Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 98–109

Matrix Multiplication: A Case Study of Algorithm Engineering

Nadav Eiron
Computer Science Department, Technion — Israel Institute of Technology
Haifa, 32000, Israel
e-mail: nadav@cs.technion.ac.il

Michael Rodeh
Computer Science Department, Technion — Israel Institute of Technology
Haifa, 32000, Israel
e-mail: rodeh@cs.technion.ac.il

and

Iris Steinwarts
Computer Science Department, Technion — Israel Institute of Technology
Haifa, 32000, Israel
e-mail: iriss@cs.technion.ac.il

ABSTRACT

Modern machines present two challenges to algorithm engineers and compiler writers: They have superscalar, super-pipelined structure, and they have elaborate memory subsystems specifically designed to reduce latency and increase bandwidth. Matrix multiplication is a classical benchmark for experimenting with techniques used to exploit machine architecture and to overcome the limitations of contemporary memory subsystems.

This research aims at advancing the state of the art of algorithm engineering by balancing instruction level parallelism, two levels of data tiling, copying to provably avoid any cache conflicts, and prefetching in parallel to algorithmic operations, in order to fully exploit the memory bandwidth. Measurements show that the resultant matrix multiplication algorithm outperforms IBM's ESSL by 6.8-31.8%, is less sensitive to the size of the input data, and scales better.

The techniques presented in this paper have been developed specifically for matrix multiplication. However, they are quite general and may be applied to other numeric algorithms. We believe that some of our concepts may be generalized to be used as compile-time techniques.

1. Introduction

As the gap between CPU and memory performance continues to grow, so does the importance of effective utilization of the memory hierarchy. This is especially evident in compute intensive algorithms that use large data sets, such as most numeric problems. The problem of dense matrix multiplication is a classical benchmark for demonstrating the effectiveness of techniques that aim at improving memory utilization. Matrix multiplication involves $O(N^3)$ scalar operations on $O(N^2)$ data items. Moreover, this ratio can be preserved when performing the multiplication operation as a sequence of operations on sub-matrices. This feature of the problem, which is shared by other numeric problems, allows efficient utilization of the memory subsystem.

The efficient implementation of compute-intensive algorithms that use large data sets present a unique engineering challenge. To allow the implementation to exploit the full potential of the

program's inherent instruction level parallelism, the adverse effects of the processor-memory performance gap should be minimized. A well engineered compute-intensive algorithm should:

- Manage with small caches;
- Avoid cache conflicts;
- Hide memory latencies associated with "cold-start" cache misses.

A broad set of techniques has been suggested to adapt numeric algorithms to the peculiarities of contemporary memory subsystems. These techniques include software pipelining [8], blocking (tiling) [9] and data copying [9, 17]. Each of these techniques was designed as a solution to one of the first two engineering challenges presented above. The relatively new method of selective software prefetching [2, 11, 12] aims at the third challenge. Software prefetching attempts to hide memory latencies by initiating a prefetch instruction sufficiently early, before the data item is used. However, the implementation should be carefully designed to avoid cache pollution by the prefetched data (see [10]). If the prefetch instruction is non-blocking, the memory access will be executed in parallel with the computations carried out by the CPU. Previous attempts to improve the performance of BLAS-2 routines are reported in [1]. Similar work on BLAS-3 routines is presented in [13]. Both these efforts employ a variety of techniques in order to overcome the memory hierarchy limitations. However, so far, they do not meet all three challenges simultaneously.

We propose a new cache-aware $O(N^3)$ matrix multiplication algorithm which builds upon known techniques to meet all of the three engineering goals. Our algorithm is based on two observations: (i) The fact that in matrix multiplication the ratio between the number of scalar operations and the data size remains high, even when the problem is divided into a sequence of multiplications of sub-matrices. (ii) The system bus is a valuable hardware resource that should be taken into account in the algorithm design.

Our algorithm uses a new blocking scheme that divides the matrices into relatively small non-square tiles, and treats the matrix multiplication operation as a series of tile multiplication phases. The data required for each phase is designed to completely fit in the cache. In addition, our scheme maintains a high ratio of scalar operations to the number of data items for each phase.

To maintain conflict-free mapping of the data regardless of the associativity level of the cache, the algorithm restructures the matrices into an array of interleaved tiles. The copying operation can be carried out during the multiplication process, using only a small copying buffer.

To cope with memory latency, all data required during phase i must be prefetched into the data cache during phase $i-1$. This is done simultaneously with the actual computation. The two activities must be well balanced. In particular, the smaller the latency and the higher the memory bandwidth — the smaller the portion of the cache needed. The order and timing of the prefetch instructions is designed to make sure that relevant data is not flushed from the cache. When doing so, the associativity level of the cache must be taken into account.

Our cache-aware $O(N^3)$ matrix multiplication algorithm does not suffer memory latency when running on an architecture that fits the assumptions of our machine model. The performance of our algorithm is not influenced by the size or layout of the input matrices. Assuming that the data set fits in the main memory of the machine, our algorithm maintains its behavior regardless of the data set size. In addition, unlike traditional blocking based algorithms, our algorithm shows little sensitivity to small changes in the input size.

We implemented our algorithm on an IBM RS/6000 PowerPC 604 based workstation. Our implementation allows instruction level parallelism by using tiling at the register level, combined with loop unrolling and software pipelining. The scheduling of machine instructions builds on the fact that in our algorithm, memory access operations in the inner-most loop are always serviced by the cache. Our implementation outperforms IBM's BLAS-3 matrix multiplication function by roughly 21.5%, on the average, for double precision data. For some values of N, our implementation runs 31.8% faster.

In this work we demonstrat memory hierarchy oriented optimizations for the $O(N^3)$ matrix multiplication algorithm. However, the same ideas can be used to achieve similar improvements in many other linear algebra algorithms that exhibit similar features (see [16]).

In the following section we describe the assumptions that we make on the machine architecture. In Section 3 we outline our techniques and their application for matrix multiplication, while in Section 4 we present our implementation for the IBM RS/6000 platform.

2. The Machine Model

When optimizing an algorithm, it is important to properly choose a simple, but sufficiently accurate machine model. The objective is then to define an abstract model such that an algorithm optimized for it will perform well in practice.

In our case, we deliberately decide to ignore the effects of the virtual memory subsystem (see, for example, [7]). Specifically, we ignore the paging mechanism, the use of virtual versus physical addresses for cache indexing, and the use of a Translation Look-aside Buffer (TLB) to shorten the address translation process. As a consequence, the negative effects of page faults and TLB misses are not taken into account. Furthermore, we assume a virtually-indexed data cache. This assumption is required to allow the algorithm to use virtual addresses when it restructures data in a manner that will assure conflict-free mapping into the cache. However, the algorithm may be adapted, under certain circumstances, to use physically-indexed data caches. Indeed, out implementation uses such a cache.

Other assumptions that we make regarding the target machine are:

- The memory subsystem includes at least one level of data cache. Our optimization techniques target only the first level (L1) data cache. We assume that slower caches do not degrade the performance of the L1 cache. Specificaly, we require that they do not affect the L1 replacement policy.
- The L1 data cache write policy is copy-back.
- The L1 data cache replacement policy is Least Recently Used (LRU).
- The processor supports a non-blocking cache fetch instruction. This may be a specialized prefetch instruction, or a simple non-blocking load instruction [4].
- The CPU follows a load/store (register-register) architecture.

We use the following parameters in the description of our algorithm: The L1 data cache holds C bytes arranged in lines of size L. The cache is K-way set associative. We denote by M the number of machine cycles required to fetch a complete cache line from memory (contemporary machines have $20 \leq M \leq 100$).

3. The Algorithm

Our algorithm is designed to carry out matrix multiplication of the form

$$\mathcal{C} = \mathcal{A} \cdot \mathcal{B}$$

where $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ are real matrices of sizes $N_1 \times N_2$, $N_2 \times N_3$ and $N_1 \times N_3$, respectively. We denote by I the matrices' element size, in bytes. We make no assumption regarding the layout, or relative location, of the input matrices in memory. In the following subsections we outline the algorithm, describing the use of each of the optimization techniques and the way in which the algorithm combines them. We end up with a matrix multiplication algorithm, that by construction, does not suffer memory latency when running on an architecture that fits the assumptions of our machine model.

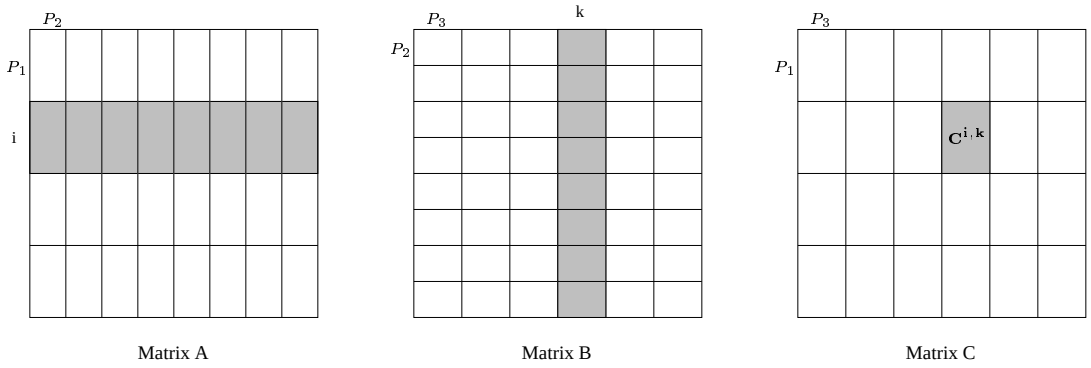


Figure 1: Tile multiplication

3.1. Tolerating Capacity and Cold Start Misses

In this subsection we assume that the cache is fully associative. We postpone the discussion on eliminating cache conflicts and pollution to the following subsections.

The algorithm partitions the input matrices into tiles (see Figure 1). The matrix $\mathcal{C}$ is divided into tiles of size $P_1 \times P_3$. Each one of these tiles is generated by a sum of products of a horizontal stripe of $\mathcal{A}$-tiles and a vertical stripe of $\mathcal{B}$-tiles. The matrix $\mathcal{A}$ is thus divided into tiles of size $P_1 \times P_2$ while $\mathcal{B}$ is divided[1] into tiles of size $P_2 \times P_3$. Denoting by $\mathcal{M}^{i,j}$ the (i,j)th tile of the matrix $\mathcal{M}$, we have:

$$\mathcal{C}^{i,k} = \sum_{j=0}^{N_2/P_2-1} \mathcal{A}^{i,j} \cdot \mathcal{B}^{j,k}$$

Since we have $N_1N_3/(P_1P_3)$ $\mathcal{C}$-tiles to compute and since the computation for each tile requires N_2/P_2 tile multiplications, we have a total of $(N_1N_2N_3)/(P_1P_2P_3)$ tile multiplication phases. In each phase we multiply an $\mathcal{A}$-tile of size $P_1 \times P_2$, by a $\mathcal{B}$-tile of size $P_2 \times P_3$, updating a $\mathcal{C}$-tile of size $P_1 \times P_3$, using $P_1P_2P_3$ scalar multiplications and $P_1P_2P_3$ scalar additions. The number of data items accessed in each phase is $P_1P_2 + P_2P_3 + P_1P_3$.

In order to achieve optimal performance, all data used in a specific phase must be present in the data cache at the beginning of the phase. If this condition is indeed met, the system bus remains unused by the tile multiplication code and can instead be used to bring the data required for the *next* phase into the cache. Let W denote the number of machine cycles it takes to multiply an $\mathcal{A}$-tile by a $\mathcal{B}$-tile and store the result in $\mathcal{C}$. Then:

$$W = \Theta(P_1P_2P_3)$$

The total amount of data (in bytes) required for each phase is:

$$I \cdot (P_1P_2 + P_2P_3 + P_1P_3)$$

Assuming that each prefetch instruction fills a single line of the cache, the number of prefetch instructions we must issue is

$$\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + P_1P_3).$$

If

$$M\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + 2P_1P_3) \leq W \tag{1}$$

[1] Naturally, this implies that P_1, P_2 and P_3 must all divide the respective dimension, N_i. Padding of the matrices may be used to meet this requirement.

```
for (i = 0 ; i < N1/P1 ; i++)
    for (j = 0 ; j < N3/P3 ; j++)
        for (k = 0 ; k < N2/P2 ; k++) {
            C^{i,j} = C^{i,j} + A^{i,k} · B^{k,j};
        }
```

Figure 2: The algorithm's outer loops.

then a single phase runs long enough to allow us to prefetch all the data required by the next phase on time. Note that in Equation (1) the size of the $\mathcal{C}$-tile is multiplied by 2. This is because $\mathcal{C}$-tiles are modified and therefore must be written back to memory, occupying the system bus.

To allow for the latency-free operation of the algorithm, the cache must be large enough to concurrently hold all the data required for a certain phase (i.e., one tile of each matrix: $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$) as well as the data that will be used in the next phase. All in all, the cache must be able to hold two tile triplets, so the total cache size must be at least

$$2I \cdot (P_1P_2 + P_2P_3 + P_1P_3) \leq C. \tag{2}$$

Note that Equation (2) places an upper bound on P_1, P_2 and P_3, while the timing considerations of Equation (1) place a lower bound on these values.

For machines with a large CPU-memory performance gap, it might happen that Euqations (1) and (2) cannot hold simultaneously for any value of P_1, P_2 and P_3. However, in Equation (1) we calculated the number of prefetch instructions required in a single phase assuming that all three tiles should be replaced. The number of prefetch instructions required may be reduced by ordering the phases so that one of the tiles is reused. The code in Figure 2 replaces $\mathcal{C}$-tiles only once in every N_2/P_2 phases. The reuse of $\mathcal{C}$-tiles is beneficial, since these tiles are the only ones that are modified and therefore must be written back to memory. Using the scheme presented in Figure 2, the number of prefetch instructions needed in a single phase is reduced to:

$$\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + \frac{P_1P_2P_3}{N_2}).$$

This means, that the condition of Equation (1) may be relaxed to:

$$M\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + 2\frac{P_1P_2P_3}{N_2}) \leq W \tag{3}$$

In case this does not yield feasible values of P_1, P_2 and P_3, the implementation's performance may degrade.

Assuming a memory bus that does not allow for pipelined memory access or outstanding requests, all prefetch instructions must be spaced at intervals of at least M units of time, to prevent stalling the CPU. With a more advanced bus that allows multiple outstanding memory requests, this requirement can be relaxed, as long as the request queue never overflows. For buses that allow pipelined memory access, a more detailed calculation of the effective value of M should be carried out.

3.2. Avoiding Cache Conflicts

Our algorithm is based on the assumption that any two triplets of tiles (one from each matrix) that are used in two consecutive tile multiplication phases can simultaneously reside in the cache. So far, we have assumed full associativity of the cache. Most practical caches have a limited degree

of associativity. Restructuring the data by copying it to properly designed locations can be used to avoid cache conflicts. [17] discusses the utilization of copying to avoid cache conflicts. In the case of matrix multiplication, copying is potentially beneficial, as it takes only $O(N^2)$ time while the computation takes $O(N^3)$ time, assuming that every data element is copied only once.

When using a K-way set associative cache ($K > 1$, K is even[2]), the following condition allows any two triplets of tiles to be mapped into the cache simultaneously:

Condition 3.1. *Associate with each of the three matrices a multi-set of cache set indices of size sufficient to hold one tile of that matrix. Use copying to map each tile of a specific matrix to the multi-set of cache sets that is associated with that matrix. The mapping is conflict-free if the multi-set union of the multi-sets for the three matrices does not contain any index more than $K/2$ times.*

This condition is sufficient but not necessary, since it is equivalent to having *any* two triplets fit in the cache, and not just the $(N_1N_2N_3)/(P_1P_2P_3)$ triplets used in matrix multiplication. Note that a scheme that complies with Condition 3.1 may be easily designed to copy each data element only once.

When using a 2-way set associative cache, Condition 3.1 may be met by mapping the tiles so that all tiles of a single matrix are mapped to the same sets in the cache, and each cache set is mapped only once. Such a mapping may be formed by interleaving the matrices' tiles, so that the offset between two tiles of the same matrix is a multiple of the way size of the cache. Since the cache is assumed to have two ways, we can have two tiles from each of the matrices resident in the cache simultaneously.

For a direct mapped cache, Condition 3.1 cannot be satisfied. To allow conflict-free mapping for direct mapped caches, we must use the fact that not every combination of three tiles from the matrices $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$ is used. Each matrix will have two possible sets of cache set indices for its tiles, with half of the tiles using one set and the other half using the second set. The tile mapping is chosen so that whenever a tile is replaced by a tile from the same matrix, the two tiles use different sets, and therefore, do not conflict.

To design such a mapping, we divide the cache lines into two equal-sized subsets: a "black" subset and a "white" subset. Each of these subsets is designed to hold one tile from each matrix. Each of the matrices will have half of its tiles mapped to black cache sets and the other half mapped to white cache sets. For the sake of this explanation, we assume that each dimension of the matrices is divided into an even number of tiles (i.e., N_1/P_1, N_2/P_2 and N_3/P_3 are all even). The matrix $\mathcal{A}$ is copied so that all even-numbered *vertical stripes* of tiles are mapped to the black part of the cache and all odd-numbered vertical stripes of tiles are mapped to the white part of the cache. The matrices $\mathcal{B}$ and $\mathcal{C}$ are copied so that all even-numbered *horizontal stripes* of tiles are mapped to the black part of the cache and all odd-numbered horizontal stripes of tiles are mapped to the white part of the cache. It can now be easily verified, that when multiplying tiles in the order of Figure 2, whenever a tile is replaced by a tile from the same matrix, the tile replacing it is colored differently, and therefore the two tiles do not conflict. Note that this method can also be implemented while copying each data element only once.

While it is possible to copy the matrices to their new locations before engaging in the multiplication process itself, it is also possible to interleave the copy operation with the computation. For an elaborate discussion of copying on the fly, see [16].

3.3. Cache Pollution

As observed in [10, 13] care must be taken, when performing prefetch operations, in order to assure that essential data is not flushed out of the cache. Similarly, we have to assure that fresh prefetched data is not flushed out before it is used for the first time. Therefore, we have to examine

[2]This assumption on K is used for simplicity. If it does not hold, use $\lfloor K/2 \rfloor$ instead of $K/2$ in the discussion that follows.

what cache lines are chosen for flushing by the replacement policy when a new cache line is brought into the cache. Since the replacement policy is applied to each set separately, we limit our discussion to a single set.

Consider a K-way set associative cache. Recall that $K/2$ lines of each set are being used by the active triplet of tiles, and the other $K/2$ lines are used to fetch the next triplet before the current phase is over. Let $\{p_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which prefetch instructions have been executed during the jth phase, $\{f_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which the *first* access to each line holding data for the jth phase has been executed, and $\{l_i^j\}_{i=0}^{K/2-1}$ be the sorted time instances in which the *last* access to each such line during the jth phase took place.

Let us now describe the conditions on the access pattern that will allow pollution-free prefetching. The first condition governs the prefetching code:

Condition 3.2. *Every block of the tiles used in the jth phase is accessed (prefetched) only once during phase $j-1$.*

Assuming that the above condition holds, the following condition is sufficient and necessary for pollution-free prefetching:

Condition 3.3. *For every $0 \leq i \leq K/2-1$ and every phase j, denote by r the least index for which $f_r^j > p_i^j$ and by s the least index for which it holds that $p_s^{j-1} > l_i^{j-1}$. If no such r exists, the prefetch instruction p_i^j is pollution-free. Otherwise, denote by F the cache lines accessed by $\{f_k^j\}_{k=0}^{r-1}$ and by P the cache lines accessed by $\{p_k^{j-1}\}_{k=0}^{s-1}$. The prefetch is pollution-free iff $P \subseteq F$.*

4. Implementation

To experiment with our approach, we implemented our matrix multiplication algorithm for both single and double precision square matrices. The target platform was a 133MHz PowerPC 604 based IBM RS/6000 43P Model 7248 workstation. The algorithm was implemented in C and was compiled using the IBM XL-C compiler [15]. Where necessary, hand-tuning of the resulting machine code was carried out. To gauge the performance improvements over known techniques, we compared our results to IBM's Engineering Scientific Subroutine Library (ESSL) [3], which is the state of the art implementation of BLAS provided by IBM for this platform.

4.1. Platform Description

The PowerPC 604 [14] is a superscalar, super-pipelined RISC processor. The CPU contains one floating point unit (FPU) and one load-store unit (LSU). It has 32 architectural floating point registers and 32 architectural integer registers. The processor has a 16KB on-chip instruction cache and a separate 16KB on-chip data cache ($C = 16384$). Both caches are *physically indexed*, four-way set associative ($K = 4$), and have a line size of 32 bytes ($L = 32$). The write policy for the on-chip data cache is write-back and the replacement policy is LRU. Access to the cache is done via non-blocking load/store instructions. Note that the PowerPC 604 processor adheres to all of the assumptions of our machine model, except for the use of a physically indexed L1 data cache.

In addition to the L1 cache the machine has an off-chip L2 *directly mapped and physically indexed,* unified cache of size 512KB. The line size of the L2 cache is 32 bytes. The L2 cache controller implements full inclusion of both on-chip L1 caches. Note that this L2 cache design contradicts our assumption that slower caches do not interfere with the replacement policy of the L1 cache.

The AIX operating system uses the PowerPC 604's MMU to implement demand paged virtual memory. The page size is 4KB, the same as the size of a way of the L1 data cache. This allows us to ignore the page-number part of the virtual address when mapping data to the L1 cache, making the distinction between physically and virtually indexed caches irrelevant.

To complete the picture, let us now examine the features of the PowerPC 604 instructions, which are relevant to our work. The PowerPC architecture supports a floating point multi-add (`fma`) instruction which performs two floating point operations: a multiplication and an addition. This instruction has four register operands and performs the following calculation: fp1 $\leftarrow$ fp2 $\cdot$ fp3 + fp4. The pipelined structure of the PowerPC CPU supports issuing one independent floating point instruction in every cycle. The usage of the `fma` instruction is most appropriate for matrix multiplication, since it allows the PowerPC 604 to complete two floating point operations in every cycle. By implementing the tile multiplication code using `fma` instructions, that use the FPU while loads and stores execute in parallel in the LSU, we assume that the value of W (the amount of work for a single tile multiplication phase) is roughly equal to $P_1P_2P_3$ machine cycles.

Floating point load instructions that hit in the on-chip L1 data cache usually complete in 3 cycles. Since the L1 data cache access is pipelined, one load/store instruction may complete in every cycle.

A load instruction that misses the L1 data cache but hits in the L2 data cache completes in approximately 20 cycles. A load instruction that misses both caches takes roughly 80 cycles (we therefore assume that $M = 80$). While the PowerPC 604 may have up to four outstanding load/store instructions, memory access is not pipelined.

As far as prefetching is concerned, the PowerPC 604 supports a special Data Cache Block Touch (`dcbt`) instruction that fetches the cache line which corresponds to its virtual effective address (VEA) into the cache. While our algorithm seems to be designed to use such an instruction, we decided to use a standard non-blocking register load instruction instead (see [16] for a discussion of this issue).

4.2. Implementation Details

We implemented the algorithm for both single and double precision floating point numbers. As described in Section 3.1, when breaking up matrix multiplication into phases, the algorithm designer can sequence the phases to maximize tile reuse. Since our target platform suffers from high memory latency, we indeed took advantage of this observation. In particular, we choose to reuse every $\mathcal{C}$-tile in every N/P_2 consecutive phases before replacing it. For the single precision implementation, we choose the following values for the tile-size parameters: $P_1 = P_3 = 32$ and $P_2 = 16$.

Proposition 4.1. *For single precision implementation on the IBM RS/6000 43P Model 7248, the choice $P_1 = P_3 = 32$ and $P_2 = 16$ complies with both Equations (2) and (3), assuming the input matrices are at least of size 6×6.*

Proof. Each $\mathcal{A}$-tile and each $\mathcal{B}$-tile have 512 elements (or 2KB in size), while each $\mathcal{C}$-tile has 1024 elements (or 4KB in size). We see that a triplet has 2KB + 2KB + 4KB = 8KB. Since the cache is 16KB in size, Equation (2) is satisfied.

The total amount of work per phase is $W = P_1P_2P_3 = 32 \cdot 16 \cdot 32 = 16384$. Plugging in Equation (3) the values for M, P_1, P_2, P_3, I, and L, we have:

$$M\frac{I}{L} \cdot (P_1P_2 + P_2P_3 + 2\frac{P_1P_2P_3}{N_2}) = 80\frac{4}{32}(512 + 512 + 2\frac{16384}{N}) \leq 16384 = W$$

This inequality holds for all $N \geq 16/3$. Since we assume $N \geq 6$, Equation (3) is satisfied. ∎

When choosing to reuse $\mathcal{C}$-tiles the frequency of prefetches is dependent on N. Since N is an input parameter of our algorithm, prefetching data from $\mathcal{C}$ at the correct frequency would have required the use of code that checks, at relatively high granularity, whether a prefetch should be executed. For the efficiency of implementation, we design $\mathcal{C}$ prefetch code to execute outside of the inner loops, prefetching only $\mathcal{A}$- and $\mathcal{B}$-tiles inside the inner-most loop. Since we have 4KB of data, which fills 128 cache lines, to prefetch during 16384 cycles, a line should be prefetched once every 128 cycles. To efficiently interleave the prefetch instructions in the tile multiplication code, we unroll the inner-most loop so that its computations take 256 cycles, long enough for one prefetch of $\mathcal{A}$ and one

```
struct Triplet {
       float A_tile[P1][P2];
       float C_half1[P1][P3/2];
       float B_tile[P2][P3];
       float C_half2[P1][P3/2];
};
```

Figure 3: Conflict free mapping

prefetch of $\mathcal{B}$. As the processor supports outstanding memory requests, the timing of the relatively few prefetch instructions for the next $\mathcal{C}$-tile, executed outside of the inner loop, is not crucial.

To allow conflict-free mapping of two tile triplets into the cache, the matrices are first copied into a page-aligned array of the `Triplet` structure, as shown in Figure 3.

Proposition 4.2. *For single precision implementation on the IBM RS/6000 43P Model 7248, copying the tiles of the three matrices to a page-aligned array of the* `Triplet` *structure satisfies Condition 3.1.*

Proof. The array is page-aligned, each structure is exactly the size of two pages, and the page size is the same as the L1 way size. This implies that every $\mathcal{A}$-tile is mapped into sets 0 to 63 of the cache, exactly once, and so is every $\mathcal{B}$-tile. For sets 64 to 127 of the cache, there are two lines of each $\mathcal{C}$-tile mapped into each set. As the cache is 4-way set associative, Condition 3.1 is satisfied, allowing the simultaneous mapping of any two triplets of tiles into the cache. ∎

To implement the tile multiplication code, we used tiling at the register level. We divided $\mathcal{B}$-tiles into sub-tiles of size 4×4 elements each. $\mathcal{A}$- and $\mathcal{C}$-tiles are divided accordingly into vertical stripes. In our inner-most loops we load a single sub-tile of $\mathcal{B}$ into 16 registers and then traverse a 4-element wide vertical stripe of both $\mathcal{A}$ and $\mathcal{C}$. In each iteration, we multiply four elements of $\mathcal{A}$ by a sub-tile of $\mathcal{B}$, while updating four elements of $\mathcal{C}$, totaling 16 scalar multiplications (this loop is unrolled 16 times, so that it runs in 256 cycles).

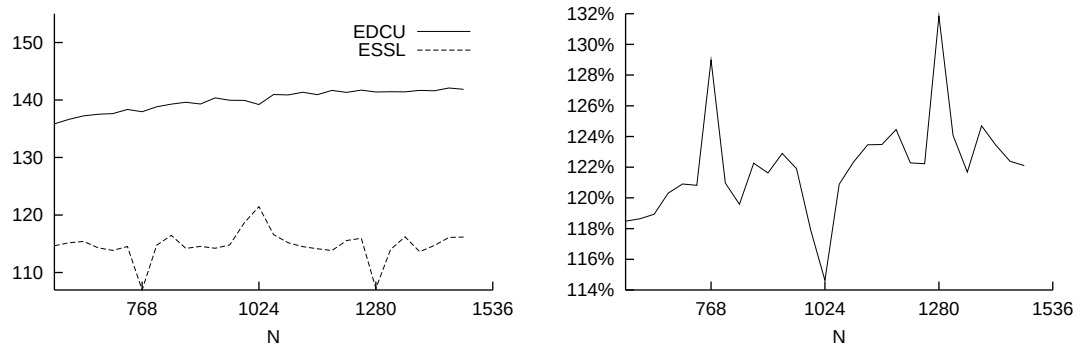
Mapping of elements within the tile storage area is designed such that first access made by the tile multplication code occur in increasing address order. $\mathcal{A}$ and $\mathcal{C}$-tiles are copied in vertical stripes, four element wide. Each such 4-element sub-row occupies consecutive memory addresses, and these sub-rows are ordered in column-major order. $\mathcal{B}$-tiles are copied such that every sub-tile of size 4×4 occupies a contiguous area of memory. These sub-tiles are again arranged in column-major order within the tile.

Prefetches are carried out according to the tiles' layout in memory, i.e., in order of increasing addresses. The prefetches of $\mathcal{A}$ and $\mathcal{B}$ are interleaved within the inner-most loop; one from $\mathcal{A}$ and then one from $\mathcal{B}$. Prefetches for the two halves of $\mathcal{C}$ are interleaved, with each half accessed in increasing adderss order. Every block is prefetched only once, satisfying Condition 3.2.

Proposition 4.3. *For single precision implementation on the IBM RS/6000 43P Model 7248, our implementation complies with Condition 3.3.*

For a complete proof of Proposition 4.3, see [16].

For double precision data, the bandwidth requirements are higher by a factor of 2 as compared to single precision. In addition, the space taken up in the cache is also doubled. This aggrevates the problem of hiding the memory latency. No values for P_1, P_2 and P_3 allows us to hold two triplets of tiles in the cache simultaneously and to hide memory latency completely. Therefore, we *forgo the*



(a) (b)

Figure 4: Performance of double precision matrix multiplication on the RS/6000 43P Model 7248: (a) In MFLOPS, (b) Relative to ESSL

prefetching of data from $\mathcal{C}$, relaxing Equation (2) to require that the cache be large enough to hold two pairs of $\mathcal{A}$- and $\mathcal{B}$-tiles and a single $\mathcal{C}$-tile simultaneously, and relaxing Equation (1) to require that a single phase runs long enough to prefetch one $\mathcal{A}$-tile and one $\mathcal{B}$-tile.

For the double precision implementation we choose the following values for the tile size parameters: $P_1 = P_3 = 32$ and $P_2 = 8$. Given these tile parameters, each $\mathcal{A}$-tile and each $\mathcal{B}$-tile is 2KB in size, while each $\mathcal{C}$-tile is 8KB in size. The total space required by two pairs of $\mathcal{A}$- and $\mathcal{B}$-tiles and one $\mathcal{C}$-tile is therefore 16KB, the same as the cache size.

The total amount of work per phase is $W = P_1 P_2 P_3 = 8192$. During this time we prefetch only one $\mathcal{A}$-tile and one $\mathcal{B}$-tile, totalling in 128 cache lines. This means that prefetch instructions should be executed once every 64 cycles. Since memory latency is $M = 80$ machine cycles, the prefetch instruction hide only 80% of the memory latency for accesses to $\mathcal{A}$ and $\mathcal{B}$.

To provide partial remedy to the performance penalty observed in the double precision case, we developed a variant that departs from our generic cache-aware matrix multiplication algorithm. We took advantage of the fact that $\mathcal{C}$ is used for output only, and therefore the initial values of $\mathcal{C}$ are all zeros. We therefore used a single memory buffer to hold $\mathcal{C}$-tiles, which is always resident in the cache. Whenever a new $\mathcal{C}$-tile is required, the old one is saved into the original $\mathcal{C}$ matrix and the buffer is cleared. By using a single, cache-resident, buffer to hold $\mathcal{C}$-tiles, we compensate for our inability to prefetch the data from $\mathcal{C}$. However, since the L1 cache uses allocate on write policy, copying back of the $\mathcal{C}$-tiles allocates cache lines to hold the modified data of the $\mathcal{C}$ matrix. This allocation may cause flushing of prefetched data from $\mathcal{A}$- and $\mathcal{B}$-tiles that is intended for use in the next tile multiplication phase. Since the combined size of an $\mathcal{A}$-tile and of a $\mathcal{B}$-tile is only 4KB, compared to 8KB for a single $\mathcal{C}$-tile, the saving of delays for access to $\mathcal{C}$-tiles is advantageous.

4.3. Results

Figure 4 (a) shows the performance in MFLOPS of our EDCU (Enhanced Data Cache Utilization) single $\mathcal{C}$-buffer double precision matrix multiplication vs. IBM's BLAS-3 (shown as ESSL). Figure 4 (b) shows the relative performance of these two implementations.

For double precision data we get the following average performance figures for the sizes of inputs we checked: IBM's ESSL implementation achieves 115 MFLOPS, our standard implementation achieves 130 MFLOPS, and our single $\mathcal{C}$-buffer implementation achieves 140.8 MFLOPS, a 21.5% average advantage over ESSL. Another feature of our algorithm that can be clearly seen from Figure 4 is that, performance-wise, our design is less sensitive to changes in the size of the data in comparison to ESSL. The performance instability evident in ESSL allows the single $\mathcal{C}$-buffer implementation to outperform the ESSL implementation by up to 31.8% on some double precision matrices.

For single precision data, the tile sizes we use are sufficient to allow prefetching of all the required data. However, the memory bandwidth is not taxed as heavily in the single precision implementation, as it is in the double precision implementation, and so the potential for performance improvement via memory latency hiding is smaller. This is clearly demonstrated by the performance achieved by the naive 3-loop $O(N^3)$ matrix multiplication algorithm. While for double precision the naive 3-loop implementation achieves only 13.6 MFLOPS on average, allowing us to achieve 935% performance increase over it, its performance is almost doubled to 23.7 MFLOPS on average for single precision numbers. For the single precision implementation, we got the following average performance figures for the sizes of input we checked: IBM's ESSL implementation achieves 154.9 MFLOPS while our EDCU implementation achieves 165.5 MFLOPS, a 7% average increase over ESSL, and up to 13% for some single precision matrices.

The main reasons for not reaching the full potential of the CPU are related to the specific machine which does not fit the assumptions taken by the algorithm. First, the machine has a relatively small L1 cache, when considering its memory latencies. Our double precision implementation could not prefetch the $\mathcal{C}$-tile, forcing us to use instead, a single $\mathcal{C}$ buffer that is copied back on every N/P_2 tile multiplication phases. The copy back operation pollutes the cache, causing delays in subsequent accesses to $\mathcal{A}$- and $\mathcal{B}$-tiles. In addition, since the tile sizes we use allow only 64 cycles between prefetch instructions, we cannot fully hide memory latency, which is as high as 80 cycles. Therefore, the inner-most loop of the double precision implementation is prolonged by roughly 25% (the difference between 80 and 64). Second, the direct mapped L2 cache forces a full inclusion policy on the instruction and data L1 caches. Therefore, some data may be flushed out of the L1 cache because of conflicts in the L2 cache, which may even result from instruction access. Third, as noted in Section 3.2, before engaging in the actual multiplication process, we copy our input matrices into an array of interleaved tiles. For the sake of simplicity, we choose to carry these copying operations off-line. These copying operations take time and our measurements indicate that the overhead for the input sizes we used is at least 14% of the peak for double precision data. Clearly, since this overhead is $O(N^2)$ its relative influence on performance diminishes as the size of the data increases.

5. Conclusion

To achieve good performance, numeric algorithms should balance computation with data movement. We have presented a new cache-aware $O(N^3)$ matrix multiplication algorithm. This algorithm can be proven to suffer no memory latency when running on an architecture that fits the assumptions of the machine model introduced. Furthermore, this algorithm uses only the smallest part of the cache that can still balance the memory bandwidth. Using a larger cache than the minimum required will have no further impact on performance.

Our experiments show that, even for platforms that are not idealy suited for the suggested techniques, the implementation of the matrix multiplication algorithm we present is competitive: It achieves performance that is higher by 6.8–31.8% than that attained by the vendor's state of the art implementation, which also uses advnaced memory hierarchy oriented optimizations [5].

While the technique presented in this work was demonstrated for $O(N^3)$ matrix multiplication, generic guidelines and conditions for cache-aware design of compute intensive algorithms were formulated and are presented in full in [16]. Furthermore, insight regarding effective exploitation of modern hardware has been gained. We believe that some of our concepts may be generalized for use as compile-time techniques.

6. Acknowledgements

Many members of the IBM Haifa Research Lab helped us significantly during this research. We are specifically indebted to David Bernstein for his helpful advice, that contributed greatly to this

work.

References

[1] R. C. Agarwal, F. G. Gustavson and M. Zubair, *Improving Performance of Linear Algebra Algorithms for Dense Matrices, Using Algorithmic Prefetch*, IBM J. Research & Development, **38(3)** (1994) 265–275.

[2] D. Callahan, K. Kennedy and A. Porterfield, *Software Prefetching*, proceedings of ASPLOS'91, 1991, 40–52.

[3] *IBM Engineering and Scientific Subroutine Library for AIX, Version 3 – Guide and Reference*, IBM Corp. 1997.

[4] K. Farkas and N. Jouppi, *Complexity/Performance Tradeoffs with Non-Blocking Loads*, Proceedings of ISCA, 1994, 211-222.

[5] F. G. Gustavson, *Personal Communications*, 1998.

[6] S. Hoxey, F. Karim, B. Hay and H. Warren (eds.), *The PowerPC Compiler Writer's Guide*, IBM Microelectronics Division, 1996.

[7] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitive Approach, 2 ed.*, 1996.

[8] M. S. Lam, *Software Pipelining: An Effective Technique for VLIW Machines*, SIGPLAN'88, 1988, 318–328.

[9] M. S. Lam, E. E. Rothenberg and M. E. Wolf, *The Cache Performance and Optimizations of Blocked Algorithms*, proceedings of ASPLOS'91, 1991, 63–74.

[10] J. H. Lee, M. Y. Lee, S. U. Choi and M. S. Park, *Reducing Cache Conflicts in Data Cache Prefetching*, Computer Architecture News, **22(4)**, (1994) 71–77.

[11] T. Mowry, *Tolerating Latency Through Software-Controlled Data Prefetching*, Ph.D. Thesis, Stanford university, 1994.

[12] T. C. Mowry, M. S. Lam and A. Gupta, *Design and Evaluation of a Compiler Algorithm for Data Prefetching*, proceedings of ASPLOS'92, 1992, 62–73.

[13] J. J. Navarro, E. García-Diego, J. R. Herrero, *Data Prefetching and Multilevel Blocking for Linear Algebra Operations*, proceedings of ICS'96, pp. 109–116, 1996.

[14] *PowerPC 604 RISC Microprocessor User's Manual*, IBM Microelectronics and Motorola Inc. 1994.

[15] K. E. Stewart, *Using the XL Compiler Options to Improve Application Performance*, PowerPC and POWER2, Technical Aspects of the new IBM RISC System/6000, IBM Corp. 1994.

[16] I. Steinwarts, *Matrix Multiplication: A Case Study of Enhanced Data Cache Utilization*, M.Sc. thesis, Department of Computer Science, The Technion — Israel Institute of Technology, 1998.

[17] O. Temam, E. D. Granston and W. Jalby, *To Copy or Not to Copy: A Compile-Time Technique for assessing When Data Copying Should be Used to Eliminate Cache Conflicts*, SUPERCOMPUTING'93 1993, 410–419.