Emmerald : A Fast Matrix-Matrix Multiply Using Intel's SSE Instructions

Douglas Aberdeen
Research School of Information Sciences and Engineering
Australian National University
daa@csl.anu.edu.au

Jonathan Baxter
Research School of Information Sciences and Engineering
Australian National University
Jonathan.Baxter@anu.edu.au

July 12, 2000

Abstract

Generalised matrix-matrix multiplication forms the kernel of many mathematical algorithms, hence a faster matrix-matrix multiply immediately benefits these algorithms. In this paper we implement efficient matrix multiplication for large matrices using the Intel Pentium single instruction multiple data (SIMD) floating point architecture. The main difficulty with the Pentium and other commodity processors is the need to efficiently utilize the cache hierarchy, particularly given the growing gap between main-memory and CPU clock speeds. We give a detailed description of the register allocation, Level 1 and Level 2 cache blocking strategies that yield the best performance for the Pentium III family. Our results demonstrate an average performance of 2.09 times faster than the leading public domain matrix-matrix multiply routines and comparable performance with Intel's SIMD small matrix-matrix multiply routines.

1 Introduction

A range of applications such as artificial neural networks benefit from GEMM (generalised matrix-matrix) multiply routines that run as fast as possible. The challenge is to use the CPUs peak floating point performance when memory access is fundamentally slow. The SIMD (Single Instruction Multiple Data) architecture of Pentium III processors mean efficient use of the memory hierarchy is critical to being able to supply data fast enough to keep the CPU fully utilised. In this paper we focus on the implementation of efficient algorithms on the Pentium SIMD architecture to achieve fast matrix-matrix multiplies of large matrices. The assembler instructions available provide two natural choices for the precision of the calculations: 32-bit floating point numbers using the SSE (Streaming SIMD Extensions) and 16-bit integer numbers using MMX (Multimedia Extensions). Our SSE SGEMM code has been nicknamed Emmerald .

GEMM forms the basis for many of the level-3 BLAS (Basic Linear Algebra subroutines), including LAPACK [9]. Thus any GEMM routine is immediately and easily used in a large range of applications. In this paper we restrict ourselves to 32-bit floating point SGEMM (single precision GEMM) because an integer GEMM is not part of BLAS and would be useful only in specialised situations. However most of the techniques discussed for Emmerald would also apply to MMX integer matrix multiplication.

Two freely available research efforts in fast GEMM are PHiPAC [7] and the more recent ATLAS [14]. Both find the best GEMM for the target architecture by generating and timing many different matrix multiply kernels, using a search algorithm to converge to the optimal parameters. ATLAS in particular is competitive with vendor optimised matrix multiplication routines including the Intel ASCI Option Red GEMM which has a peak performance just 10% faster than ATLAS using double precision GEMM [3]. Because it is freely available and shows near optimal performance, especially on Intel architectures, we compare Emmerald against the performance of ATLAS. Our experiments showed that ATLAS achieves a peak of 375 MFlops/s for single-precision multiplies on a PIII @ 450MHz, or $0.83\times$*clock rate*. Our matrix-matrix multiply using SIMD instructions achieves a peak of 890 MFlops/s, or $1.98\times$*clock rate*.

The rest of this paper is organised as follows. Section 2 describes the GEMM multiplication routines in general. In section 3 we introduce the concept of blocking as the main tool used to improve memory access efficiency. Details of how we apply blocking and low level register allocation in the case of Emmerald are presented in section 4. Results and comparisons with ATLAS are given in section 5, followed in Section 6 by details of an application which achieves a price/performance ratio under USD $1/MFlop/s for distributed training of a very large scale neural network.

2 SGEMM

The GEMM (Generalised Matrix-Matrix) multiply routines form the basis of level-3 BLAS [9]. Research into matrix-matrix multiplication aims to produce the most efficient possible GEMM routines for a given architecture. These routines perform the matrix operation:

$$C \leftarrow \alpha op(A) op(B) + \beta C, \tag{1}$$

where C is an $(M \times N)$ matrix, $op(A)$ is the $(M \times K)$ matrix given by A or A^T and $op(B)$ is the $(K \times N)$ matrix given by B or B^T.

SIMD technology refers to the ability of a processor to perform a *single instruction* such as a multiply or add on *multiple data*. The SSE instructions provide an obvious way to improve matrix multiply performance since we wish to perform the same sequence of operations on large amounts of data. The Intel SSE assembler instructions allow one floating point instruction to be performed on 4 pairs of single precision floating point numbers simultaneously [5]. This means up to 4 floating point numbers are in the same stage of execution in the FPU (floating point unit) pipeline during any given cycle. The use of special 128-bit registers to hold 4 32-bit numbers at a time restricts the instructions to single precision, hence our efforts have been directed at producing a `blas_sgemm()` compatible function rather than the more commonly used double precision `blas_dgemm()` routine.

Traditional dense (square) matrix multiply algorithms require $O(N^3)$ floating point additions and multiplications. Strassen's method reduces this complexity to $O(N^{\log_2 7})$ by using a divide and conquer algorithm [11]. In practice the algorithm exhibits poor locality making it hard to reuse data in cache [12]. To compensate for this and the overhead incurred by recursing down to matrices of size one, a truncation point, typically around $M = N = K = 64$ is defined after which the traditional algorithm is used. This produces a hybrid method which does not achieve the optimal $O(N^{\log_2 7})$. Strassen's method also has complications which arise when the operand dimensions are not integer multiples of the truncation point. For these reasons and because cache re-use is critical for keeping a SIMD CPU fully utilised, we have chosen not to implement this method. However, we expect that for sufficiently large matrices a hybrid of Emmerald and Strassen's method would be advisable.

3 Blocking

In a matrix multiply each element of the result matrix is generated by a dot-product, where a sequence of floating point numbers are multiplied and accumulated, requiring two floating point operations for each pair of numbers in the dot product operands.

The 64-bit 100 MHz bus of the PIII allows a maximum throughput of 200 million single precision values per second from main memory. If we rely purely on main memory to supply values, we have an upper bound of 200 MFlops/s regardless of processor speed. Pentiums with SIMD instructions can potentially execute at a MFlop/s rate four times their clock rate, which would consume values at a rate 9 times the bandwidth of main memory for a 450 MHz machine. Our algorithm must take into account the availability of L0 (registers), L1 and L2 caches to overcome the limitations inherent in deep memory hierarchies. On the PIII the L1 data cache is 16 Kb 2-way set-associative with a 3 cycle latency to the registers and L2 cache is a unified 512 Kb 4-way set-associative with a 6 cycle latency [3]. The maximum throughput from the L2 cache is 900 million single precision values per second. Both L1 and L2 caches are write back, meaning writes go to cache and main memory is only updated when an updated cache line is overwritten. For a detailed treatment of caches and translation lookahead buffers the reader is referred to [4, §5].

A standard matrix multiply technique for deep memory hierarchy machines (used in [3, 7, 14]) is matrix blocking. The fundamental concept is to break a large matrix-matrix multiply into a series of smaller matrix multiplies where the data required will fit entirely into cache. This is illustrated in Figure 1. Let A be a matrix of dimension $(M \times K)$ and B be a matrix of dimension $(K \times N)$. Then $C \longleftarrow AB$ is the $(M \times N)$ result. If we cannot fit all of A, B and C into L1 cache simultaneously a large time

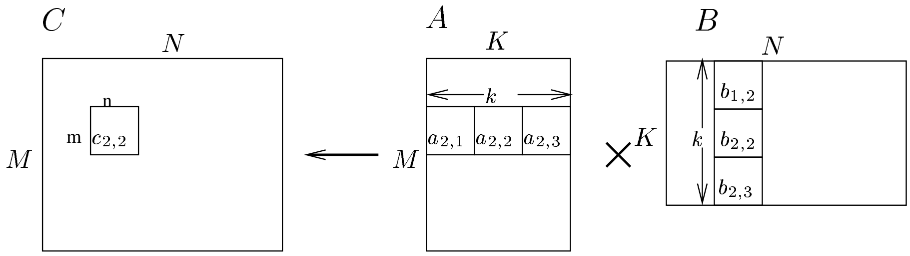

Figure 1: Blocking matrix multiplies for improved memory efficiency.

```
for (x = 0; x < m; x++) {                /* outer loop */
    for (y = 0; y < n; y++) {
        for (z = 0; z < k; z++) {    /* inner loop */
            c[x*n + y] += a[x*k + z]*b[z*n + y];
        }
    }
}
```

Figure 2: Naive matrix multiply code showing the outer loop and the inner loop.

penalty will be incurred each time we fetch an element, either from L2 cache, or worse, from main memory.

Blocking aims to minimise the calls to a lower level in the memory hierarchy by doing $M/m \times N/n$ smaller matrix multiplies $a = bc$, with dimensions $a \sim (m \times k)$, $b \sim (k \times n)$ and $c \sim (m \times n)$. We chose m, n and k so that the blocks a, b and c all fit into the current cache level. Hence all the data necessary for the current block multiply is in the same cache level and lower levels are only accessed when we move onto the next block. Figure 1 shows how the block $c_{2,2}$ is generated from the operation:

$$c_{2,2} \leftarrow a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + a_{2,3}b_{3,2}. \tag{2}$$

If M, N and K are not multiples of m, n and k respectively we must treat the boundary cases as different sized blocks or pad with zeros.

The order in which each block multiply is done is one of the blocking parameters. Figure 2 shows how we define the order of the loops. At each level in the cache hierarchy, the operation we are performing is equivalent to a matrix multiplication that operates on blocks instead of individual elements. The standard matrix multiply has 2 loops that iterate over the elements of C and 1 loop to iterate along a row of A and a column of B. This loop performs a dot product that computes one element of the result matrix. We define the outer loop at a cache level to be whichever of the three necessary loops that contains the others nested within it. The inner loop at a cache level is the most nested loop. The middle loop is implicitly whichever loop remains.

The results are accumulated in C, meaning the result is invariant under the ordering of loops. A clever ordering can take advantage of the particular cache architecture to

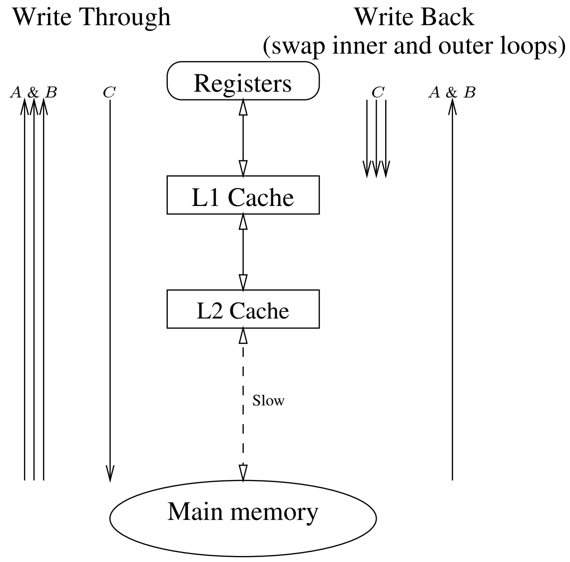

Figure 3: The cache hierarchy with the relative loads of A & B compared to stores to C for a write through scheme and a write back scheme.

obtain peak performance. For example, if the cache is write through any storage of a result in $c_{x,y}$ results in data being written *through* every cache layer to main memory. The bus operations required for this write take time away from the bus operations needed to read from the A and B matrices. In this case we would want to accumulate $c_{x,y}$ in registers for as long as possible to prevent writes. However, to accumulate like this we can generate only as many results as fit into registers. From Figure 2, the only way to limit the number of results being generated in the innermost loop is to iterate along $a_{x,1\ldots K/k}$ $b_{1\ldots K/k,y}$, i.e. we do a dot product to generate a single result $c_{x,y}$, hence avoiding any write backs until the end of the dot product because $c_{x,y}$ can remain in a register. Figure 3 shows the write through case on the left side, with many reads of A and B for few writes of C. If the cache is write back, writes only go to L1 and we can write from registers without using the main-memory bus. Thus it might be preferable to hold as many A and B elements in registers as possible, while generating and writing back all the C values the A and B elements contribute to. This would minimise the reads of A and B values, another way of reducing bus traffic. Figure 3 shows the write back case on the right, with many writes of C to L1 for few reads of A and B. To do this in Figure 2 we would move the inner loop to the outer loop, while the order of the other two loops will depend on other factors. In Emmerald we try to minimise both the reads from A and B as well as write backs to C.

Blocking can be made hierarchical to suit the hierarchical nature of the memory architecture. Thus if the blocks a, b and c reside entirely in L2 cache, we can further sub-block them into a', b' and c' which reside entirely in L1 cache. Finally we must decide which elements of a', b' and c' to put in which registers to minimise all calls to

Parameter	*Description*
M	Rows of A. Fixed.
N	Columns of B. Fixed.
K	Columns of A and rows of B. Fixed.
m	# of rows to block from A into L2.
n	# of columns to block from B into L2.
k	# of columns from A and rows from B to block into L2, $mk + kn + mn$ elements should fit into L2.
L2_{outer}	Do we increment x or y or z in the outer loop of $a_{x,y}b_{y,z}$.
L2_{inner}	Do we increment x or y or z in the inner loop of $a_{x,y}b_{y,z}$.
m'	# of rows to block from $a_{x,y}$ into L1.
n'	# of columns to block from $b_{y,z}$ into L1.
k'	# of columns from A and rows from B to block into L1, $m'k' + k'n' + m'n'$ elements should fit into L1.
L1_{outer}	Do we increment x or y or z in the outer loop of $a'_{x,y}b'_{y,z}$.
L1_{inner}	Do we increment x or y or z in the inner loop of $a'_{x,y}b'_{y,z}$.
L0	How do we allocate registers.

Table 1: Summary of blocking parameters for a 2 level cache architecture.

memory. This depends mainly on the number of registers available. Table 1 summaries the parameters to consider for blocking.

The behavior of the TLB (translation look-aside buffer) must also be taken into account when planning a blocking strategy. A new TLB entry must be created when a virtual address for a newly loaded page is referenced for the first time. If TLB misses are ignored performance drops of up to 30% have been observed for large matrices [2]. The way to minimize these misses is to limit the number of virtual addresses used by buffering parts of the matrix into a contiguous address space.

4 Floating-Point SIMD

This section describes the algorithm for using the floating point SIMD instructions to perform matrix-matrix multiplication. Special attention is paid to the blocking strategy applied. We start by describing the floating point Intel SIMD architecture. Then we discuss why the obvious approach of adding SIMD parallelism to ATLAS does not work, before presenting Emmerald beginning with register allocation and working up through L1 cache and L2 cache organisation.

4.1 Floating Point SIMD Pipeline

There are 8 128-bit special `xmm` registers on which the SSE instructions operate. The instructions in Table 2 are used at the lowest level in Emmerald to implement matrix-matrix multiplication. The `PS` suffix on most of the instructions indicates that the operation is *parallel scalar*, meaning the operation works on four 32-bit floating point

Instruction	*Cycles*	*Description*
`MULPS`	5	Multiply four floating point numbers.
`ADDPS`	3	Add four floating point numbers.
`MOVUPS`	>3	Move four floating point numbers to memory.
`MOVAPS`	>3	Move four floating point numbers to memory aligned on a 128-bit boundary.
`SHUFPS`	3	Shuffle four floating point numbers around in registers.
`MOVHLPS`	3	Shuffle and copy the high pair of values to the low pair in a new register.
`MOVLHPS`	3	Shuffle and copy the low pair of values to the high pair in a new register.
`ADDSS`	3	Add the first floating point number only.
`MOVSS`	>3	Move the first floating point number to memory.
`PREFETCHT0`	N/A	pre-fetch 32 bytes into L1 cache starting at mem. Does not cause processor stalls.
`CPUID`	1	Report on the availability of SIMD instructions.

Table 2: Summary of the floating point SIMD instructions used.

numbers in parallel. We cannot pre-determine how long a memory access will take. We only know that the minimum time to fetch data is 3 cycles, which occurs when the data resides in L1 cache.

The Intel FPU pipeline is organised in such a way that one floating point instruction can be issued per cycle as long as multiplies are followed by an add and there are no stalls due to register dependencies. The theoretical CPI (cycles per instruction) should be 1.0 on the code from Figure 4 since it follows the MULPS–ADDPS sequence and there are no dependencies shorter than the pipeline (up to 5 cycles for multiplies). However, our experiments have shown that this code, when unrolled 8 times and timed over several million iterations, achieves a CPI of 1.3, or 1350 MFlops/s on a 450 MHz CPU. The code is unrolled to amortise the extra cycles taken to do the jump at the end of each loop. If we unroll sufficiently we observe no change in the CPI upon unrolling further. This places an upper bound on the MFlops/s achievable for matrix multiplication assuming all the data can be stored in registers and dependencies are ignored. A possible explanation for the high CPI is the PIII's lack of a dedicated 4-way SIMD FPU. Instead has two 2-way units that cannot always function in parallel [10].

4.2 Converting ATLAS

ATLAS provides a state of the art SGEMM routine. The code for which is freely available [13]. Our first effort at applying the SIMD instructions was to re-write the kernel of ATLAS. The kernel uses a square matrix-matrix multiply of size $m = n = k < N_B$, assuming all the data can to fit into L1 cache. A search algorithm then computes the best value of N_B and how to arrange kernel multiplies to suit the memory

```
MULPS(1, 1);
ADDPS(2, 2);
MULPS(3, 3);
ADDPS(4, 4);
MULPS(5, 5);
ADDPS(6, 6);
MULPS(7, 7);
```

Figure 4: Optimal SIMD code for maximum MFlops/s

hierarchy of the target architecture. The re-write of the ATLAS kernel produced an improvement of 1.75 times over the non-SIMD kernel. However, we found that a much faster kernel matrix-multiply could be obtained using narrow rectangular blocks. Hence we developed Emmerald as as a complete SGEMM implementation.

4.3 L0 (Register) Blocking

Because the PIII SIMD instructions are performing 4 operations in parallel the problem of how to supply data to the processor fast enough to keep the FPU fully utilised is exacerbated four-fold. We must devise a scheme to minimise the ratio of memory accesses to floating point operations. This is done with two core strategies:

- accumulate results for as long as possible to reduce write backs;
- re-use values in registers as much as possible.

The authors of [3] perform multiple dot-products to achieve this. Each dot product generates:

$$C_{i,j} \leftarrow A_{i,1}B_{1,j} + A_{i,2}B_{2,j} + \ldots + A_{i,K}B_{K,j}. \tag{3}$$

We took the same approach and found by trial and error that the optimal number of dot-products to do simultaneously is 5. Figure 5 shows how the dot products progress. Each small black circle represents an element in the current matrix block. Each dashed square represents one floating point value in a `xmm` register. Thus 4 dotted squares together form one 128-bit `xmm` register.

`xmm0` keeps the next 4 values from the a' row we are operating on. Two registers (`xmm1` and `xmm2`) alternately keep 4 values from each of the 5 columns of the b' (L1 block), we are operating on. The use of `xmm1` and `xmm2` in an alternating fashion allows one register to be loading a value while the other is involved in floating point operations. The remaining 5 registers accumulate the results, hence using all 8 `xmm` registers. Figure 6 shows all the operations performed in one iteration of the innermost loop. It can be clearly seen that 8 floating point operations are done on each column from b' during each iteration, giving a total of 40 flops per iteration. This Figure also represents the L0 block, the equivalent of a vector-matrix multiply of dimension $1 \times 4 \times 5$. The vector-matrix multiply inner assembler loop is shown in Figure 10. The

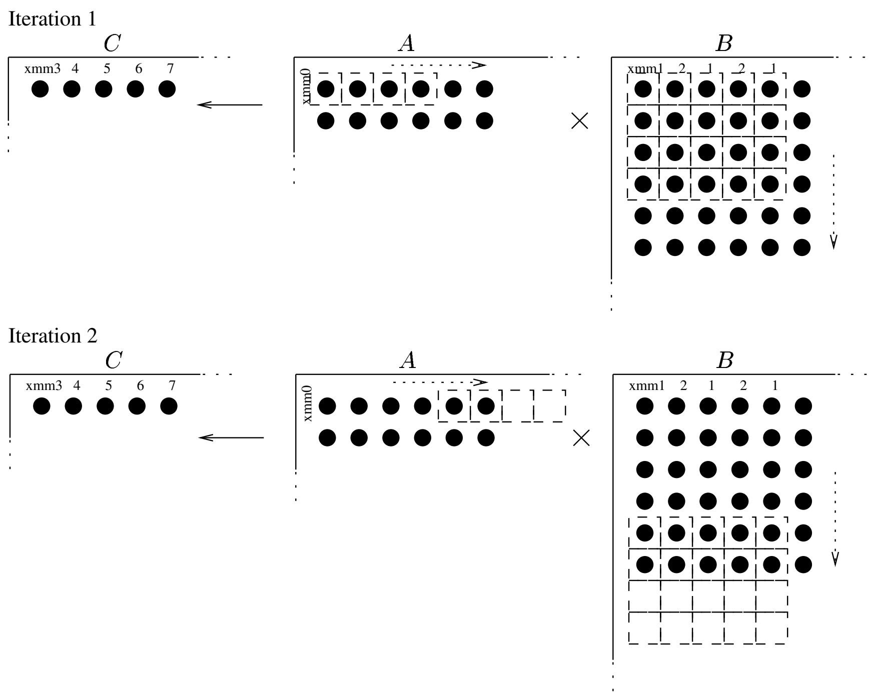

Figure 5: Allocation of `xmm` registers, showing progression of the dot products which form the innermost loop of the algorithm.

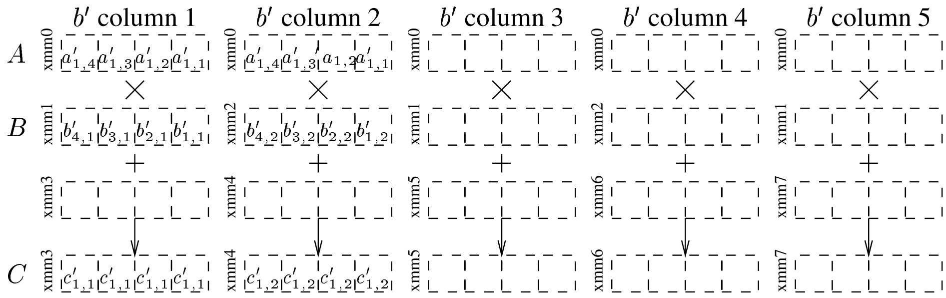

Figure 6: Floating point operations performed in one iteration of the innermost loop.

a' row is fetched in line 2, and the 5 b' columns are fetched alternately into `xmm1` and `xmm2` in lines 3, 4, 7, 10, and 13. Note that the results are not stored back immediately but continue to be accumulated into `xmm3,4...7`.

Loads of the b' values are usually done with greater than 3 cycles between the load and use. This accounts for the 3 cycle latency between L1 cache and registers. Loads from a' are done once for the next 40 floating point operations with at least 3 cycles before their first use. This can be seen in Figure 10 where the a' value is loaded in line 2 then it is not used until line 5. Other techniques used to improve pipeline performance are:

- The b' block is buffered into an area of memory assumed to be in L1 cache constantly (to be discussed further in section 4.4). This has the further advantage of reducing TLB misses [2].
- The b' values in the buffer are all aligned on 128-bit boundaries to allow use of the faster MOVAPS call instead of the MOVUPS call (see Table 2).
- The next 8 a' values in the row are pre-cached into L1 cache approximately 66 instructions in advance. This can be seen in line 1 of Figure 10. The placement of the `PREFETCHT0` commands was done by trial and error as recommended by Intel [5].
- Code generation is used to completely unroll the assembler loop (Figure 10) for all allowable values of k'. Care must be taken to ensure this does not overflow the L1 instruction cache.

Data shuffling must sometimes be done on the `xmm` registers to re-order the four floating point values. This is required if the four results in a register are accumulated into one, or the four results must be written back to non-contiguous memory. Shuffling is an expensive operation and acceptable results are only achieved if shuffles are rare.

At the end of the dot-product we have 5 registers which hold 5 elements of the result matrix to be accumulated into C. However these registers each hold four values that must first be summed into one before the result is written back. This process is

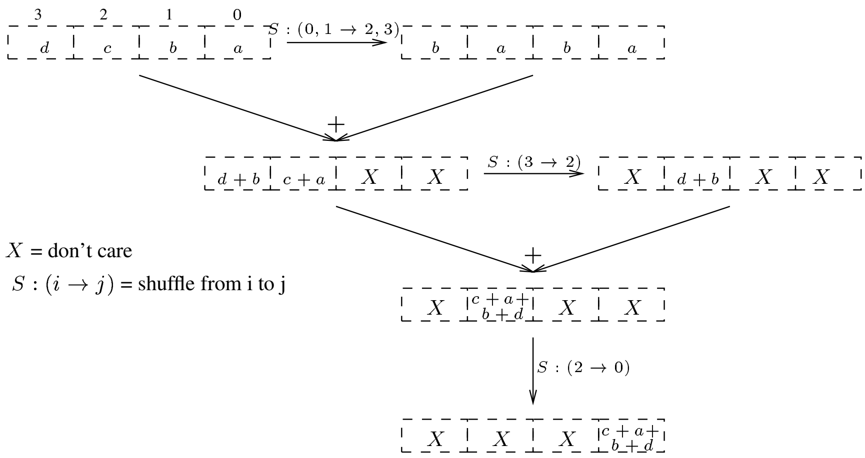

Figure 7: Shuffling operations required to accumulate parallel results into one scalar value.

illustrated in Figure 7. A minimum of 3 shuffle instructions are required, each taking a minimum of 3 cycles. Starting with one `xmm` register to be accumulated, a copy is made shuffling the low pair into the high pair. This copy is added in parallel to the original. A copy is made of the result, shuffling value 3 into value 2, before being added to the first result. Now we have the accumulated result in position 2, which is shuffled to position 0 to allow access by the `MOVSS` instruction. Note that if we shuffle the high pair into the low pair in the first step, we avoid the final shuffle. However, limitations of the `SHUFPS` instruction do not allow this. The overhead of this procedure is minimised by putting off the shuffling and write back for as long as possible using long dot products.

4.4 L1 Blocking

The L1 data cache is 16Kb 2-way set associative. This means there are two banks of cache, each of which can hold 2048 single precision values in one cache bank. The line of the cache bank into which a value is inserted is determined by the least significant bits of its address. If that line in the cache is already used in both banks, i.e. two values already share addresses with the same least significant bits, then the value is loaded into the least recently accessed bank [3]. We cannot assume 100% utilisation of both cache banks for matrix data is possible because at any one time there are intermediate variables and old data in the cache over which we have no control. It is safer to assume that we can fully utilise one bank with matrix data. We do this by ensuring the matrix data has contiguous addresses so that every element has different least-significant bits, hence making sure no matrix data overlaps in cache [2]. This leaves a cache bank free to hold sundry matrix data and variables without fear of our matrix data being constantly ejected from the cache. There is still a small chance that some matrix data will be ejected if, for instance, there are two or more intermediate variables which map

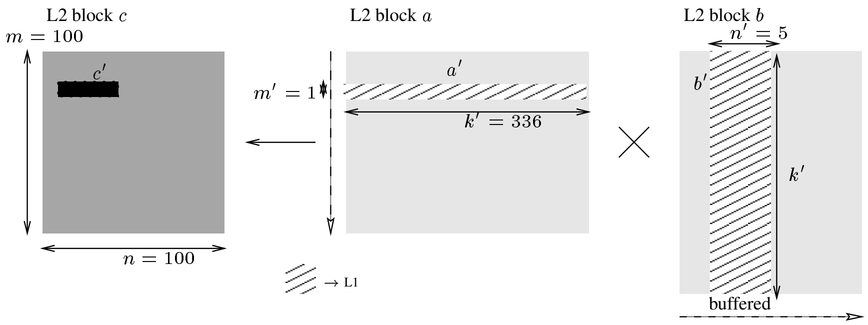

Figure 8: L1 Blocking Scheme.

to the same cache line as some of the matrix data. The random replacement policy prevents us from putting the matrix data into just one bank, but we are ensuring that we only use one of the two cache lines available for an address.

To hold just the a' and b' matrices in L1 permanently we have a maximum length of $k' = 341$ for the dot products. Because the innermost loop computes 4 elements of each dot product, we choose a multiple of four for the column length. In fact we unroll the inner loop by a multiple of four times so we want an overall multiple of 16 for k', up to a maximum of 336.

Since $b' \sim (336 \times 5)$ is large compared to $a' \sim (1 \times 336)$, b' is buffered into L1 cache. We want to re-load b' rarely, so L1_{outer} as defined in Table 2 moves along 5 columns of b' at a time. L1_{inner} iterates over one row of a at a time. We rely on the pre-fetching to get the a' values into L1 ahead of time. This is shown in Figure 8. To summarise:

- $m' = 1$,
- $n' = 5$,
- $k' \leq 336$,
- $\text{L1}_{\text{inner}} =$ rows of a,
- $\text{L1}_{\text{outer}} =$ columns of b.

4.5 L2 Blocking

The L2 cache is 512Kb, 4-way set associative. We set L2_{inner} to take k columns of A (and hence k rows of B) at a time. This is the inner loop because it allows a c block to be completely read into L2 from main memory, written back by the CPU and then stored to main memory before the next c block is accessed. Each c block is only touched once each per `blas_sgemm()` call. This is important because it minimises the amount of the C matrix that we need in L2 cache at any one time, hence minimising conflicts with the a and b blocks and main memory transactions. If we move this loop to $L2_{outer}$ or the L2 middle loop, we would generate results which map into multiple c blocks, hence requiring multiple c blocks to be in L2 cache which would increase conflicts. Put another way, caches require tight temporal or spatial locality to work effectively and our choice of $L2_{inner}$ forces the result addresses generated to have good locality. A useful heuristic is to loop over the largest block in the outer loop and the smallest in the inner loop [15]. This reflects the idea that we should re-read the largest block rarely. The c block is the smallest in L2 so we have additional motivation to iterate over it in the inner loop. We avoid reading C at all if $\beta = 0$ (3).

Now we choose $L2_{outer}$, the outermost loop of the entire algorithm. The two alternatives are to loop along m rows of A or n columns of B per iteration. Note that we buffer only b' into L1 cache, relying on pre-fetching to get the a' values from L2 to L1. Thus it is more important that a values be in L2 than b values, since b' values are assumed to be in L1. According to the LRU (least recently used) replacement policy of the Intel cache [3], the data most recently read is more likely to be in L2 than older data. Thus $L2_{outer}$ should loop along n columns of B per iteration, leaving the middle loop of the L2 blocking to move along m rows of A, hence reading a values more frequently than b values. If reading the a values ejects b values from L2 cache, we do not incur too large a penalty due to the rarity of such reads as discussed in Section 4.4. This argument only holds if a and b blocks contain an equal number of elements, otherwise we fall back on our heuristic and put the larger block in the outer loop. Figure 9 illustrates this blocking for $k = 336$. We choose m, n and k to be multiples of the L1 blocking parameters to avoid extra boundary cases for every L2 block.

We might consider buffering each L2 block in the same way we buffered b' in L1. Then we guarantee that a, b and c do not conflict in L2 cache (ignoring a few sundry variables), recalling from Section 4.4 that contiguous memory less than the size of a bank will be placed into non-overlapping cache lines. Without buffering we cannot assume that the memory used for a block is contiguous since a block is taken from different spans of the main memory allocated to a matrix. Buffering also decreases TLB misses. Emmerald combines L1 and L2 b buffering into one process. At the start of the `sgemm()` call the b matrix is copied into block major order, at the same time performing the constant α multiplication and preparation for L1 buffering which requires alignment and re-ordering of elements within L1 blocks to fit the preferred assembler loop access pattern. This requires memory and time overhead which makes Emmerald inefficient for small matrices. ATLAS uses the same technique without the re-ordering of individual elements in L1 blocks [15].

Empirically we determined that utilisation of approximately 70% of the L2 cache is optimal. Due to varying PIII L2 cache sizes, the *usable* L2 cache size is a runtime tunable parameter and L2 blocking dimensions are determined at runtime. We first choose the largest $k \leq 336$ such that k (nearly) evenly divides K. The a and b dimensions are then chosen to keep the a and b blocks the same size while filling the usable L2 cache. Allowing a and b to be different sizes reduced performance.

Our experiments showed that including L2 blocking allows average performance levels to be maintained across operands too big to fit into L2. To summarise:

- $k' = k \leq 336$
- $L2_{inner}$ = columns of A, rows of B,

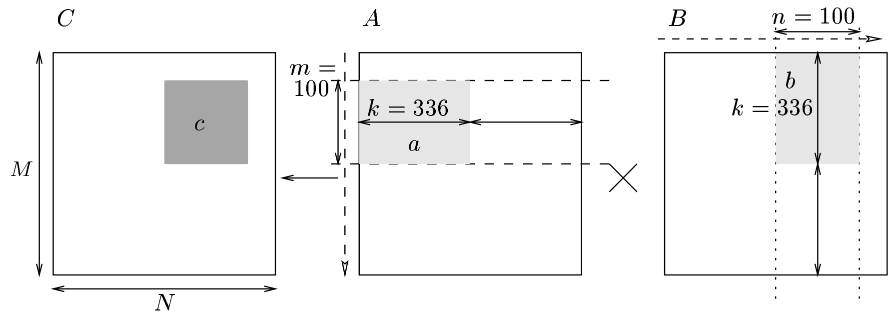

Figure 9: L2 Blocking Scheme

- $\text{L2}_{\text{outer}} =$ columns of B.

5 Results

Our experiments were carried out on PIII processors running Linux kernel 2.2.12. For the floating point experiments a kernel patch was necessary to activate the floating point SIMD instructions and registers. All code was written using `gcc` with inline assembler calls to new instructions. Version 2.9.1 or better of `gas` is required to use these instructions.

The performance of Emmerald was measured by timing matrix multiply calls with size $M = N = K = 16$ up to 700. The following steps were taken to ensure a conservative performance estimate:

- wall clock time on an unloaded machine is used rather than CPU time;
- the stride of the matrices, which determines the separation in memory between each row of matrix data, is fixed to 700 rather than the optimal value (the length of the row);
- caches are flushed between calls to `sgemm()`.

Figure 11 shows Emmerald's performance compared to ATLAS and a naive three-loop matrix multiply. The average performance of Emmerald after size 100 is 760 MFlops/s or $1.69\times$*clock rate* of the processor and 2.09 times faster than ATLAS. A peak rate of 890 MFlops/s is achieved when $m = n = k = stride = 320$. This represents $1.98\times$*clock rate*. On a PIII 550 MHz we achieve a peak of 1090 MFlops/s, demonstrating scaling of performance with clock speed. The largest tested size was $m = n = k = stride = 3696$ which ran at 940 MFlops @ 550 MHz. To put these results in context, the naive matrix multiply implemented as shown in Figure 2 achieves an average of 29 MFlops/s for large matrices. Thus Emmerald is 26 times faster than a naive multiply.

```
; Pre-fetch 8 A matrix values to L1
prefetcht0  48(ebx)
; Load four A matrix value
movups      (esi),xmm0
; Load four B matrix values from buffer col 1
movaps      (edi),xmm1
; Load four B matrix values from buffer col 2
movaps      -288(ecx),xmm7
; four A values * four B values from col 1
mulps       xmm0,xmm1
; Accumulate four values into col 1 result
addps       xmm1,xmm2
; Load four B matrix values from buffer col 3
movaps      -272(ecx),xmm1
; four A values * four B values from col 2
mulps       xmm0,xmm7
; Accumulate four values into col 2 result
addps       xmm7,xmm3
; Load four B matrix values from buffer col 4
movaps      -256(ecx),xmm7
; four A values * four B values from col 3
mulps       xmm0,xmm1
; Accumulate four values into col 3 result
addps       xmm1,xmm4
; Load four B matrix values from buffer col 5
movaps      -240(ecx),xmm1
; four A values * four B values from col 4
mulps       xmm0,xmm7
; Accumulate four values into col four result
addps       xmm7,xmm5
; four A values * four B values from col 5
mulps       xmm0,xmm1
; Accumulate four values into col 5 result
addps       xmm1,xmm6
```

Figure 10: SIMD code for 5 dot products of length 4

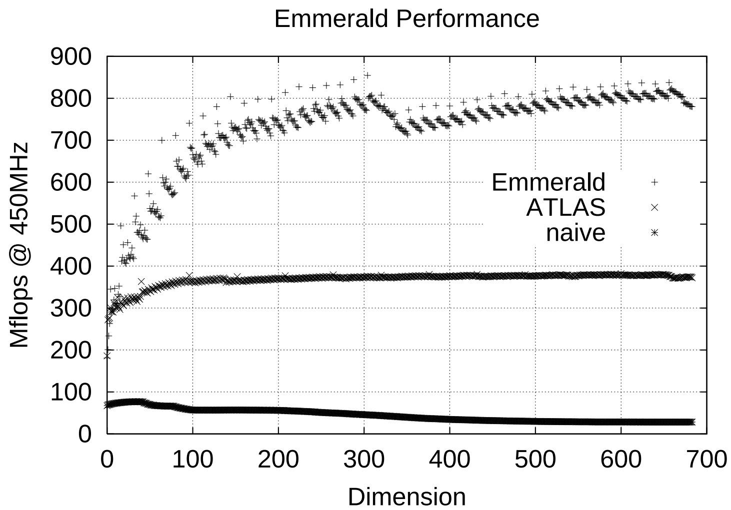

Figure 11: Performance of Emmerald on a PIII running at 450 MHz compared to ATLAS SGEMM and a naive 3-loop matrix multiply. Note that ATLAS does not make use of the PIII SSE instructions.

Emmerald's performance fluctuates considerably between consecutive matrix dimensions. This is due to boundary effects where we switch to a non SIMD SGEMM (such as ATLAS) when the matrix edges blocks are too small to be efficiently computed with Emmerald. ATLAS avoids these fluctuations by having separately optimized small matrix code [14]. We could use SIMD small matrix code such as described by Intel in [6], however decreased bus efficiency in the absence of long dot products means we gain little performance over using ATLAS for this calculation. The kink in the performance of Emmerald at dimension 336 indicates the transition point where L2 caching is required to maintain performance for very large matrices.

The SIMD matrix multiply code released by Intel is for matrices up to dimension 6×6 [6], designed with applications such as 3D transformations in mind. This code runs at 633 MFlops/s. How to compare this result to our SGEMM is not clear because these small matrices reside entirely in L1 cache, thus no performance penalty is incurred from main memory accesses. On the other hand a large proportion of the operations on small matrices are data shuffles which are not necessary when we can do long dot-product style operations on large matrices. These data shuffles prevent us from using the Intel code as the kernel multiply for a high-performance arbitrary matrix size GEMM. Assuming that the benefits of long dot-products should outweigh the penalty of fetching from main-memory when we correctly block our code, the comparison shows that we have a reasonable performance. If this assumption is not true, then we have excellent performance.

6 Application to Ultra Large Scale Neural Networks

To illustrate the use of Emmerald in a real-world application, we used it as the core of a distributed Neural-Network training algorithm for large-scale Linux clusters [1].

We performed an experiment in which a neural network with 1.8 million adjustable parameters was trained to recognize machine-printed Japanese characters from a database containing 6 million training patterns. The training was run on *Bunyip*[1], a 196 processor, Linux-based Intel Pentium III cluster consisting of 98 dual 550 MHz processor PC's.

We trained the network in a continuous run for 56 hours and 52 minutes, requiring a total of 31.2 Peta Flops (10^{15} single-precision floating-point operations), with an average performance of 152 GFlops/s (single precision). Total memory usage during training was 38.6 GBytes. The total machine cost, including the labor cost in construction, was USD$149,500, giving a final price/performance ratio of 98¢ USD / MFlops/s (single precision). We believe this to be the first time the $\$1$ / MFlops/s barrier has been broken for a supercomputing application.

7 Alternative SIMD Architectures

The blocking and memory organization techniques covered in this paper apply to all deep memory hierarchy machines. The code that requires tailoring to specific SIMD architectures covers the dot product style inner loops, L1 block element re-ordering, alignment for fast 128-bit loads, L1 block dimensions and cache prefetch commands.

In principle Emmerald can be easily ported by re-writing the inner assembly loop in Figure 10 and adjusting the L1 block sizes to suit the size of the L1 cache and the number of registers. Other SIMD architectures are more flexible than the Pentium SSE architecture, providing more instructions and more registers. Thus we expect Emmerald, or similar GEMMs, implemented on alternative SIMD platforms, to exhibit performance boosts greater than those reported in this paper. For example the Motorola AltiVec architecture has 32 Kb of L1 data cache and 32 128-bit registers [8]. In this case we might re-write the inner loop to use L1 blocks of $a' \sim (1 \times 160)$ and $b' \sim (24 \times 160)$. The inner loop would also more efficient since we can use one instruction for multiply and accumulate instead of two. Other candidate architectures for simple ports include AMD's 3DNow! and Sun's MAJC [10].

8 Conclusion

We have described the construction of a fast SIMD generalised matrix-matrix multiply for the Intel PIII architecture. Details of the assembly inner loop, L1 blocking for SIMD architectures and L2 blocking were discussed.

The Emmerald code is available from http://csl.anu.edu.au/∼daa/research.html. We compared Emmerald to ATLAS, the leading public domain non-SIMD SGEMM. The average performance of Emmerald is 2.09 times faster than ATLAS which equates to a

[1] http://tux.anu.edu.au/Projects/Beowulf/

generalized matrix multiply running at a MFlop/s rate of 1.69 times the clock speed of the processor.

To illustrate the use of Emmerald in a real-world application, we used it as the core of a distributed Neural-Network training algorithm for large-scale Linux clusters. A sustained performance of 152 GFlops/s on a 198 processor Linux Cluster was achieved, with a price/performance ratio of 98¢ USD / MFlops/s (single precision).

9 Acknowledgements

The authors would like to thank Markus Hegland and Clint Whaley for their comments and suggestions. This work was supported by the Australian Research Council.

References

[1] D. Aberdeen, J. Baxter, and R. Edwards. 98¢ /mflop, ultra-large-scale neural-network training on a PIII cluster. Sumbitted to SC2000, May 2000.

[2] B. Greer. The most imprortant technical library in the world. http://www.nag.co.uk/other/ff98_papers/greer.html, July 1999.

[3] B. Greer and G. Henry. High performance software on Intel Pentium Pro processors or Micro-Ops to TeraFLOPS. Technical report, Intel, August 1997. http:// www.cs.utk.edu/~ghenry/sc97/paper.htm.

[4] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitive Approach*. Morgan Kaufmann, 2nd edition, 1996.

[5] Intel. *Intel Architecture Optimization Reference Manual*, 1999. http://developer.intel.com/design/PentiumII/manuals/245127.htm.

[6] Intel. *Streaming SIMD Extensions – Matrix Multiplication*, June 1999. http://developer.intel.com/ design/pentiumiii/ sml/245045.htm.

[7] J.Bilmes, K.Asanovic, J.Demmel, D.Lam, and C.W.Chin. PHiPAC: A portable, high-performace, ANSI C coding methodology and its application to matrix multiply. Technical report, University of Tennessee, August 1996. http://www.icsi.berkeley.edu/~bilmes/phipac.

[8] Motorola. MPC7400 PowerPC microprocessors. http://www.motorola.com /SPS/PowerPC/products/ semiconductor/cpu/7400.html, October 1999.

[9] Netlib. *Basic Linear Algebra Subroutines*, November 1998. http://www.netlib.org/blas/index.html.

[10] J. Stokes. 3 1/2 SIMD architectures. June 2000. http://arstechnica.com/cpu/1q00/simd/simd-1.html.

[11] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[12] M. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning strassen's matrix multiplication for memory efficiency. In *Super Computing*, 1998.

[13] R. C. Whaley and J. Dongarra. *Automatically Tuned Linear Algebra Software V1.0*, September 1998. http://www.netlib.org/atlas/.

[14] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. Technical report, Computer Science Department, University of Tennessee, 1997. http://www.netlib.org/utk/projects/atlas/.

[15] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. Technical report, Dept. of Computer Sciences, Univ. of TN, Knoxville, March 2000. http://www.cs.utk.edu/ ~rwhaley/ATLAS/atlas.html.