

CS267 Spring 2000

Assignment 2: Matrix Multiplication

D. Bonachea, J. Chapman, N. Putnam

Introduction

In optimizing a matrix multiplication routine, we attempted improvements that fell into two general categories: management of memory access patterns, and pipelining for full utilization of the processor. The optimizations we attempted interacted with each other significantly, making it hard to evaluate the success of individual optimizations and to tune the optimization parameters. In fact, we didn't have time to test every combination of the optimizations attempted, and wound up discarding several of the optimizations described below in our final submission. These testing difficulties were compounded by having imprecise timings, although in the end we were able to get better timings by writing our own timer function.

Notation

$$C_{ij} = A_{ik} B_{kj}$$

The use of column major format makes indexing over k correspond to a stride of 1 in B , and a large stride in A .

Platforms

To test the peak performance for floating point operations of the two platforms we used, we wrote a simple test program with no memory access requirements beyond the registers and easily pipelined instructions. We found that the NOW's SUN SPARC Ultra 1s had a peak speed of 166 Mflops/sec for adds or multiplies separately, and 333 Mflops/sec when adds and multiplies are in a 1 to 1 ratio. 166 Mflops/sec were measured for a 300MHz Pentium II with 128 Mbytes of physical memory.

NoW SUN SPARC Ultra1 / 170¹²

128 megs main memory

166 MHz clock

16Kb L1 d-cache, 32 byte lines, associativity 1

512Kb L2 i/d-cache, 64 byte lines, associativity 1

45-60 MB free

16 64-bit FP registers

Measured peak speed for double-precision floating point ops: 333.3 MHz Mflops/sec

Pentium II

300 MHz clock

512K unified non-blocking L2 cache.

16K data, 16K inst. L1 cache

Measured peak speed for double-precision floating point ops: 166.6 Mflops/sec

¹ Sun Microsystems, Data Sheet STP5110A "UltraSPARC-1 CPU Module", July 1997.

² Sun Microsystems "The UltraSPARC Processor Technology White Paper: The UltraSPARC Architecture"

Description of optimizations:

Standard Optimizations

Strength Reduction

We tried to eliminate array index calculations involving expensive integer multiplies by maintaining array indices and updating them with a light weight integer add at each loop iteration. For example:

```
for (bj = 0; bj < n_blocks; ++bj) {
    j = bj * BLOCK_SIZE;
    x = f(a[j]);
}
```

can be replaced by:

```
j=0;
for (bj = 0; bj < n_blocks; ++bj) {
    j += BLOCK_SIZE;
    x = f(a[j]);
}
```

Induction variable elimination

We can further reduce loop overhead and register pressure by eliminating one of the loop variables. For instance, we can eliminate bj from the example above:

```
j=0;
for (bj = 0; bj < n_blocks; ++bj) {
    j += BLOCK_SIZE;
    x = f(a[j]);
}
```

becomes:

```
for (j=0; j<n_blocks*BLOCK_SIZE; j+=BLOCK_SIZE) {
    x = f(a[j]);
}
```

Loop Invariant Code Hoisting

We attempted to remove from the loops all calculations of quantities that don't change between loop iterations. For example, in unrolling loops, we eschewed expressions such as:

```
t_a2 = Aik[lda * 2];
```

and instead used

```
t_a2 = Aik[lda2];
```

where we have previously computed $lda2 = 2 * lda$;

Loop unrolling

We attempted to unroll the loops once along i and j (ie. to compute a 2 by 2 block of C) and 16 times along k . Our rationale was that unrolling in i and j should be important for instruction scheduling, because unless elements of A and B can be used more than once each time they are loaded into a register, the optimal balance between memory loads, adds and multiplies of 1:1:1 cannot be achieved. In the end, we weren't able to achieve a significant speedup with this, and the code we're submitting uses 8-way unrolling in the loop over k only.

Our code's relatively slow performance (even at small matrix size) is a good indication that these were not optimally implemented.

Compiler Optimizations

`-fast, -fsimple=2` (not used)

These options turn off IEEE compliance in the processor's floating point system, and allow faster, floating point operations by disabling denormalized floating point numbers and relaxing the floating point exception model. Turning these options on did allow our code to run about twice as fast while still giving correct results for the test matrices given. However, for matrices with very small or very large entries, it would be likely to give wrong answers.

`-dalign`

This compiler option instructs the compiler to assume that pointers to doubles are properly aligned. Because the memory for the three matrices A, B and C is allocated statically as three arrays of doubles, and we don't do anything weird with the pointers to the data, it is safe to allow the compiler to make this assumption. We found that failing to specify this option caused each double to be loaded from memory with two `ld` operations (one for each half of the double), as opposed to a single `ldd` operation, resulting in the doubling of the number of floating point loads.

`-x05`

This option asks the compiler to use as many optimizations as it can to make the code go faster. We found that specifying it significantly improves the speed of our code.

`-xtarget=native -xarch=v8plusa`

These options tell the compiler what instruction set and which architecture to target in compiling the code.

5.0 vs. 4.2

We used the Sun cc 5.0 compiler because it seems to do a better job of instruction scheduling and loop unrolling. Specifically, 5.0 does a better job of dependency analysis inside our k-loop and reorders the operations.

Data Layout

Loop interchange

By exchanging the loops over I and J so that the outer loop is over J, we ease the memory access pattern. At each J, we access a single column of B and C. Because we are constrained to column major format, cache lines lie along the columns and columns can be loaded into memory without loading other memory locations (except sometimes at the beginning and end of the column). In contrast, when loading a single row (for matrix sizes greater than 4x4), neighboring rows are also loaded into the cache. So putting the loop over J on the outside lets us work on a single column of B and C at a time, leaving more space in the cache for many rows of A. This allows us to effectively use a larger block size.

Transpose on the fly

If A and B are divided into b by b arrays of “blocks”, each block of A must be operated on b times. The first time we visit this block, we transpose it to a row-major format and cache it in a private memory area. This is done as we compute the first column of C, using the first column of B. The rest of the rows of C in that first block, and all of the rest of the b-1 block multiplies involving that block of A reuse the cached copy.

Tiling loop interchange

To make most efficient use of our transposed, cached copy of each block of A, we make the inner loop over blocks run over blocks of B. That way we use the cached block of A b times and then invalidate it.

Matrix-Specific Optimizations

Adaptive block size

We systematically tested the performance of our code over a range of block sizes. We found that the optimal block size was different for different matrix sizes. We have exploited this fact by using different block sizes depending on the size of the input matrices. (See figure 3)

We chose block sizes based on this test:

matrix size	block size
1 to 87	16
88 to 183	88
184 to 299	184
300 to 599	88
>599	56

(we tested matrix sizes out to 2040, and still get about 80 Mflops.)

Multilevel blocking

We tested code that used two levels of blocking, and found that the speedup was insignificant. This optimization is not part of the final version of the code we submitted.

Constant block size for non-fringe code

We compile separate realizations of the block multiply routine for fringe and full-sized blocks. In the routine that multiplies the full-sized blocks, the block size is a constant, which allows the compiler to make more optimizations, and higher speed is realized. We generated a different function for each block size so the block size would be a manifest constant within the function, assisting in pointer manipulations.

Logarithmic decomposition of fringe

The same principle can be applied to the fringes as well. Routines for multiplying fringes of BLOCKSIZE by m can be compiled where m is, for example powers of 2. Fringes are then broken into no more than $\log_2(\text{BLOCKSIZE}-1)$ fringes, each of which is computed with a fixed size routines. We considered this optimization, but didn't implement it.

Conclusions:

Block size matters! So does memory access pattern and cache behavior. Lastly, if you want to save a lot of trouble and get the fastest matrix multiply, just download BLAS.

Figures

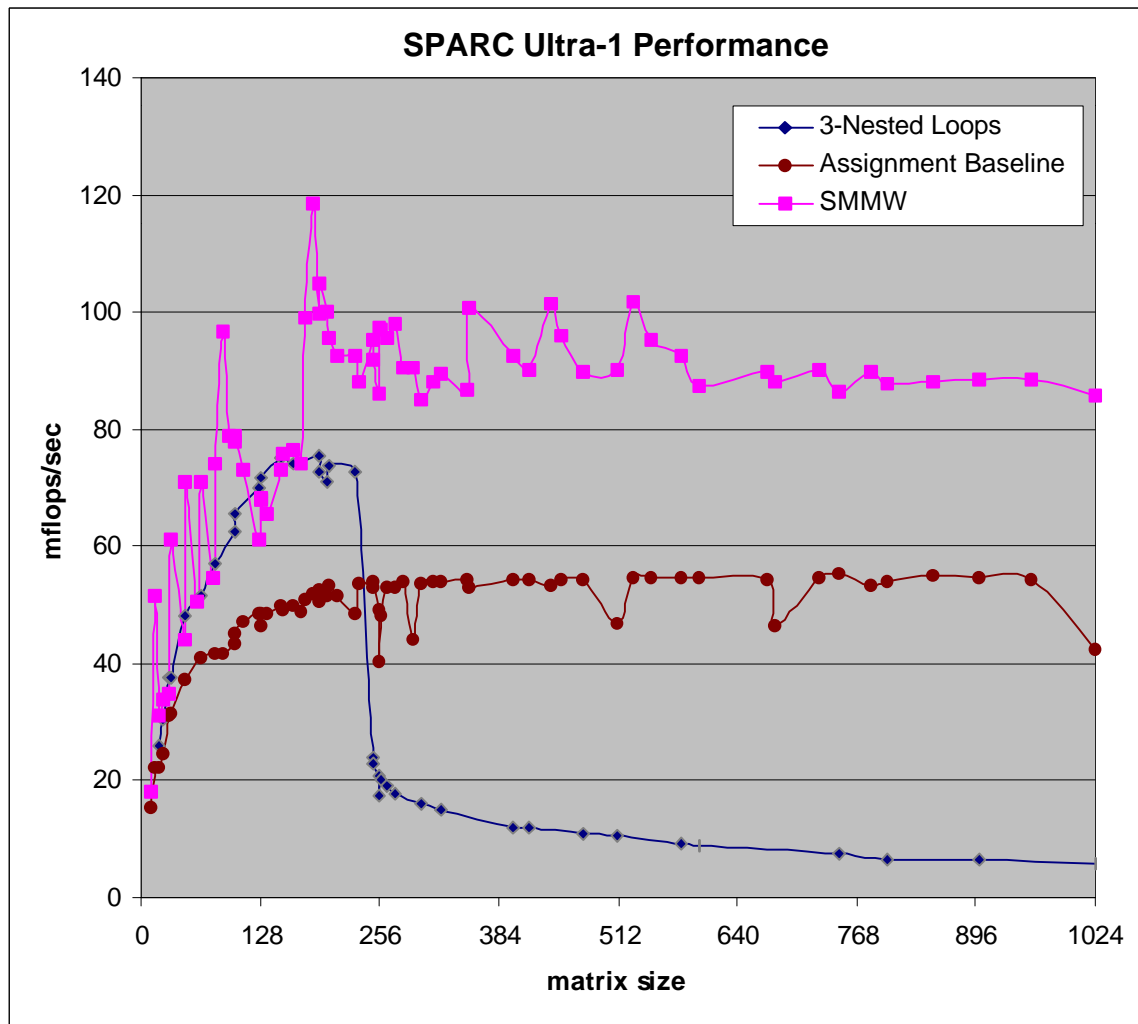


Figure 1 shows performance vs. matrix size for the simple three nested loops, the sample blocked code, and our routine. The three-nested-loops routine performs well up to a matrix size of about 230 by 230. The marked decline in performance coincides to the matrix size at which the level two cache cannot hold all of A and one column of B, when some overhead and aliasing effects are allowed for. The performance of the SMMW routine is consistently faster than both three nested loops and the baseline blocked code which uses the constant sample block size of 16. It has peaks at matrix sizes that are multiples of the block size used. (See also the discussion of figure 3.)

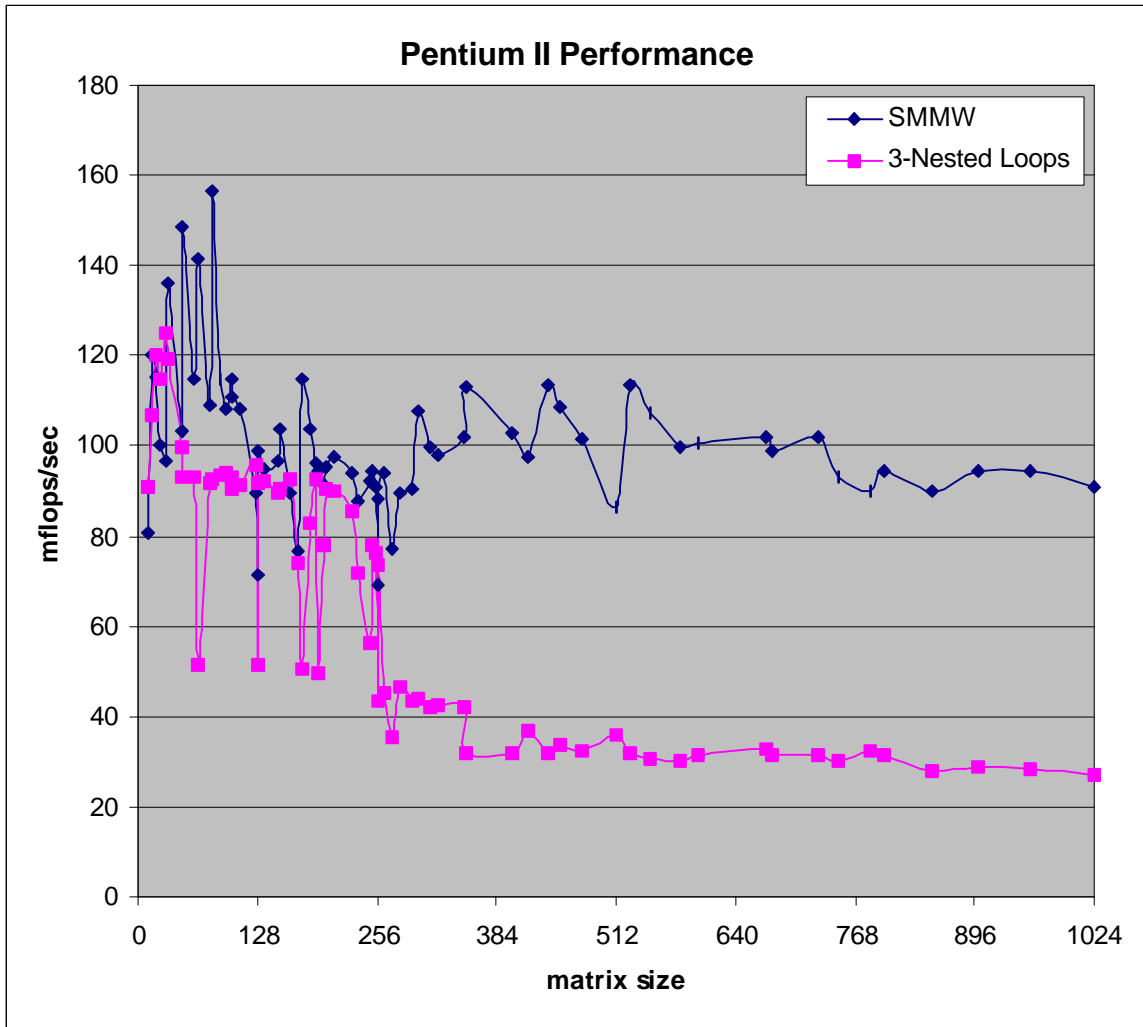


Figure 2 compares SMMW with three nested loops running on a Pentium II. The code actually runs faster, and at a higher fraction of the theoretical peak speed of the processor. The Pentium II has level one and level two caches of equal capacity as those on the SUN, but not necessarily of the same speed. This higher speed could be attributed to the memory system running at a higher speed, or to having a more highly optimized compiler. (Microsoft 32 bit C++ compiler V 12.00.8168)

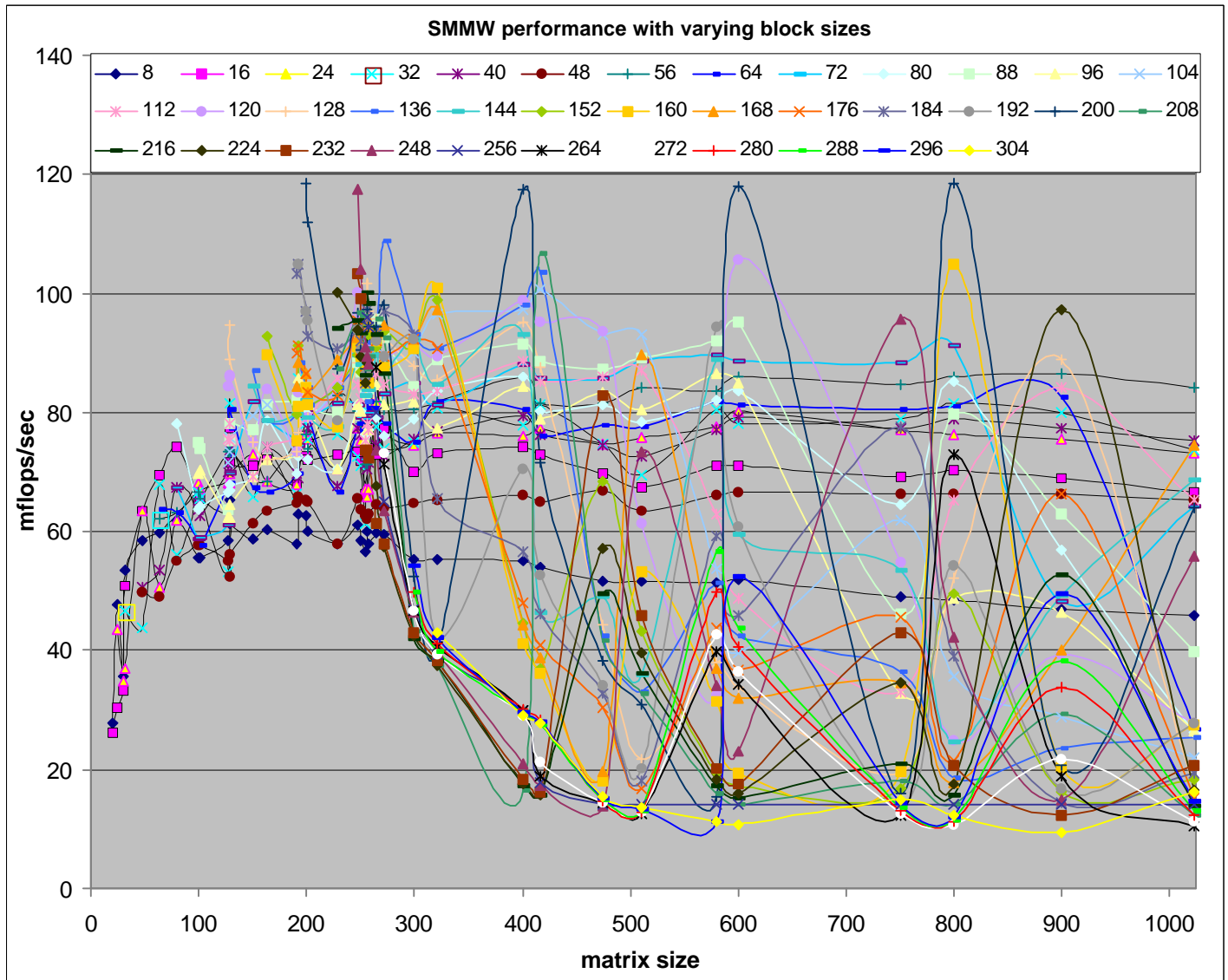


Figure 3 shows the performance of SMMW vs. matrix size with various fixed block sizes. Peaks in performance for matrices that have a size that is a multiple of the block size are due to the relatively lower speed of the fringe-multiplying code. Larger block sizes exceed the cache capacity and lead to poor performance. Note the best choice of block size varies based on the size of the matrix being computed.