

Optimization of Matrix Multiply on the Itanium 2

Amir Kamil, Guillermo Diez Canas, Wei Xiao
University of California, Berkeley

February 16, 2004

1 Introduction

Naïve matrix multiply implementations perform dismally on the Itanium 2, achieving less than a fifth of the peak floating point performance of the machine. Optimized implementations such as BLAS do exist, achieving upwards of eighty percent of peak performance. In this assignment, we attempted to tune a simple blocked implementation in order to approach the performance of BLAS.

2 Attempted Optimizations

We tried a multitude of optimizations, most offering very little performance increases. Some, however, such as register blocking, provided large gains in performance.

2.1 L2 Cache Blocking

Since the Itanium 2 does not allow floating point values in its L1 cache, the first level of cache blocking possible is for the L2 cache. Though the original blocked implementation found 56 to be optimal the block size, no block sizes larger than 64 were attempted, so we tried larger block sizes. To simplify matters, we only tested block sizes that were powers of 2. Figure 1 shows that 128 is the optimal L2 block size, agreeing with the result computed by the formula in [1].

2.2 L3 Cache Blocking

Having settled on an L2 block size of 128, we then attempted to block for the L3 cache. Again, we only tried powers of 2, with 256 giving the best performance as shown in figure 2. The performance advantage over single-level blocking was only marginal, however. We believe that the L2 block size is large enough and the L3 block size small enough that the additional level of blocking is redundant.

2.3 Padding and Fringe Handling

Performance of both the simple and blocked matrix multiply algorithms falls considerably at and near multiples of 256, likely due to cache interference. For such cases, we padded the original matrix so that its new size was at least three more than a multiple of 256, reducing but not eliminating the performance decrease as shown in figure 3.

Close to multiples of our block size, the existence of fringes results in an additional performance dip. In order to counteract this phenomenon, we combined multiple fringe blocks when the individual size was less than half a normal block. Figure 3 shows that this provides small improvements in performance.

Once register blocking was implemented, however, we no longer experienced performance dips at multiples of 256. Rather, performance peaked at these locations, due to the lack of register fringe blocks.

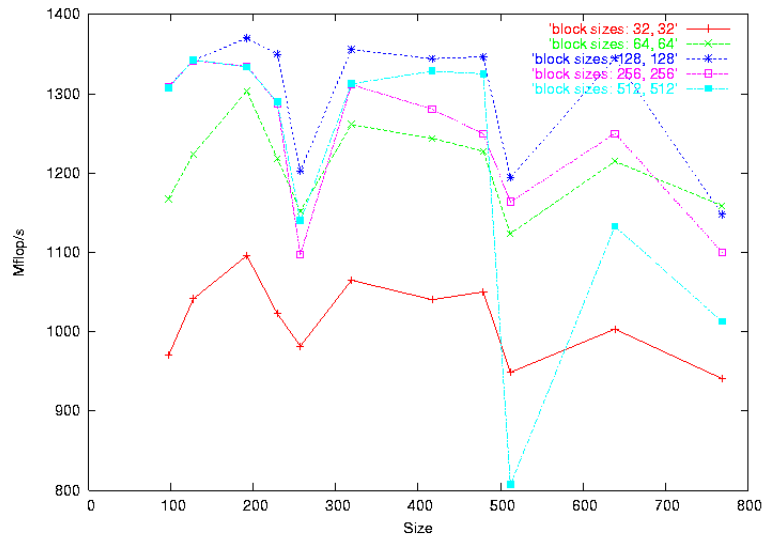


Figure 1: Comparison of block sizes for single-level blocking.

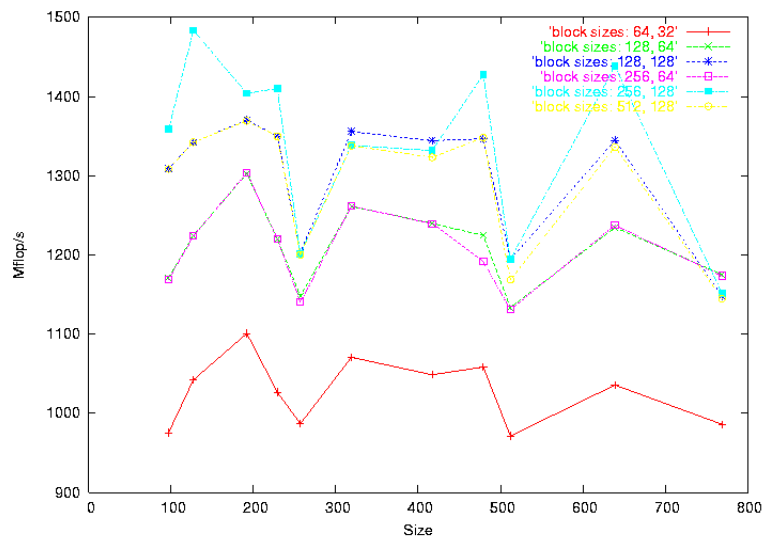


Figure 2: Comparison of block sizes for multi-level blocking.

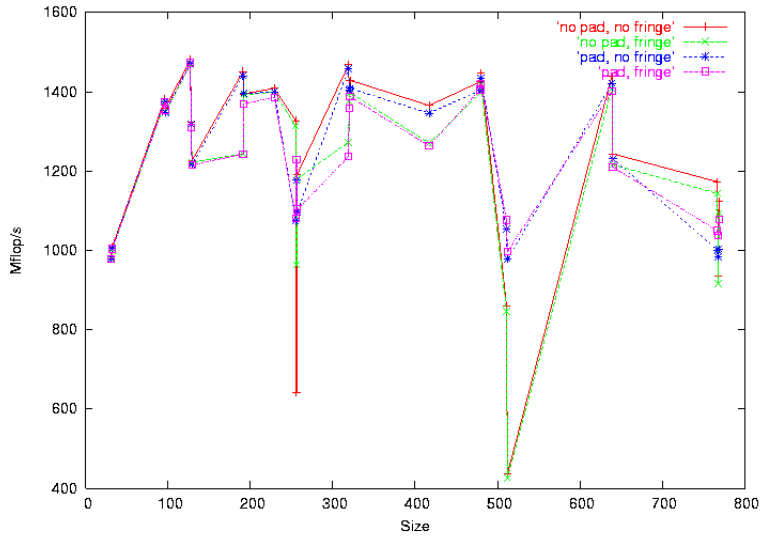


Figure 3: The effects of padding and simple fringe handling on performance.

2.4 Register Blocking

Register blocking provided by far the largest performance improvement of any optimization, taking performance from about 1400 Mflop/s in figure 3 to about 2800 Mflop/s in figure 4.

In order to determine the best size for register blocks, we wrote a code generator to output code for various block sizes. We varied three parameters in our testing, r , the number of rows in a register block in the target matrix, c , the number of columns in a register block in the target matrix, and k the number of columns in the first source matrix and the number of rows in the second source matrix. Performance results for different variations of these parameters are in figure 4, with $8 \times 4 \times 2$ providing the best results.

As indicated above, register blocks of size $8 \times 4 \times 2$ do not require padding and fringe handling for cache blocks. However, in order to minimize our code footprint, we padded all matrix sizes such that no register fringe blocks existed. The performance difference was negligible.

2.5 Loop Unrolling and Software Pipelining

Our code generator already unrolled all loops in their entirety, and implemented some measure of software pipelining. Room for improvement of the latter existed however, and we hand-tuned the generated code, pre-computing repeatedly used values and increasing the amount of pipelining. Figure 5 compares the performance of different versions of our hand-tuned code. Initial attempts at hand-tuning failed, though subsequent attempts were more fruitful.

2.6 Static Array Allocation

In order to pad matrices to eliminate register fringe blocks, new arrays must be allocated and the data copied to them. Originally, we allocated these arrays dynamically using `new`. We later modified our code to use statically allocated arrays, improving performance by about 200 Mflop/s as shown in figure 6. Since the matrix size is not known beforehand, dynamic allocation can't be completely avoided, so we dynamically allocate arrays for matrix sizes larger than 1024.

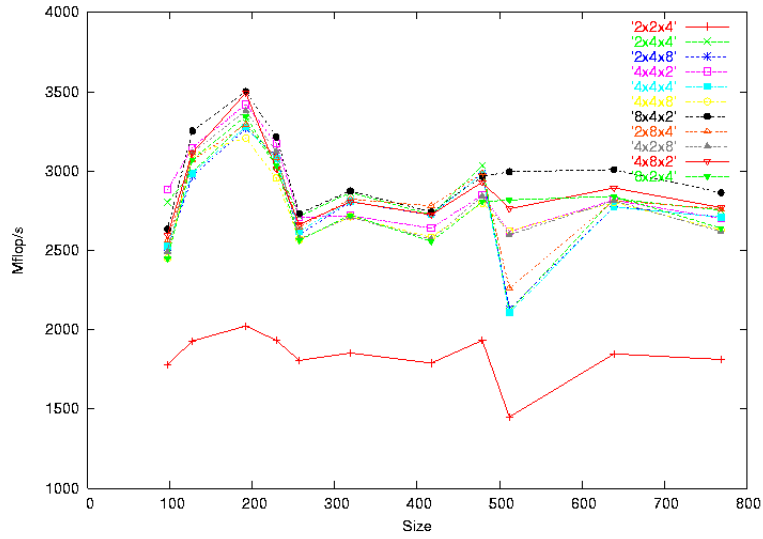


Figure 4: Performance of different register block sizes. Labels are given as $r \times c \times k$, where the block size in the target matrix is $r \times c$ and in the source matrices are $r \times k$ and $k \times c$.

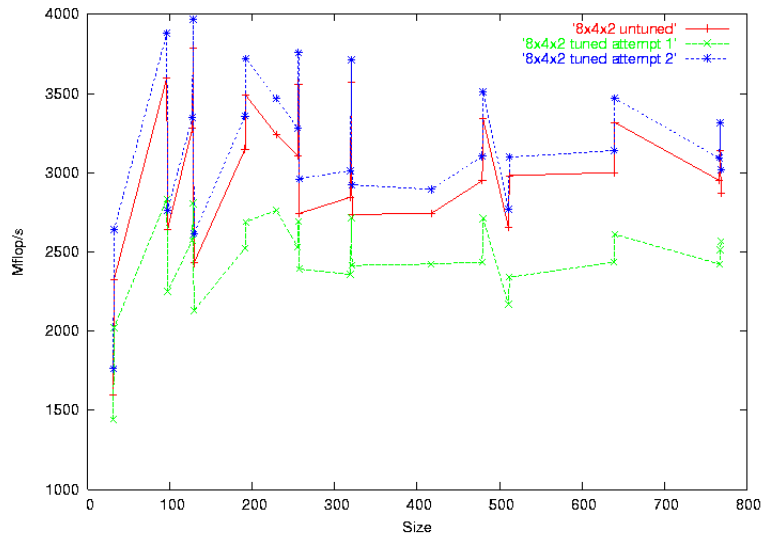


Figure 5: Comparison of untuned and tuned versions of register blocking.

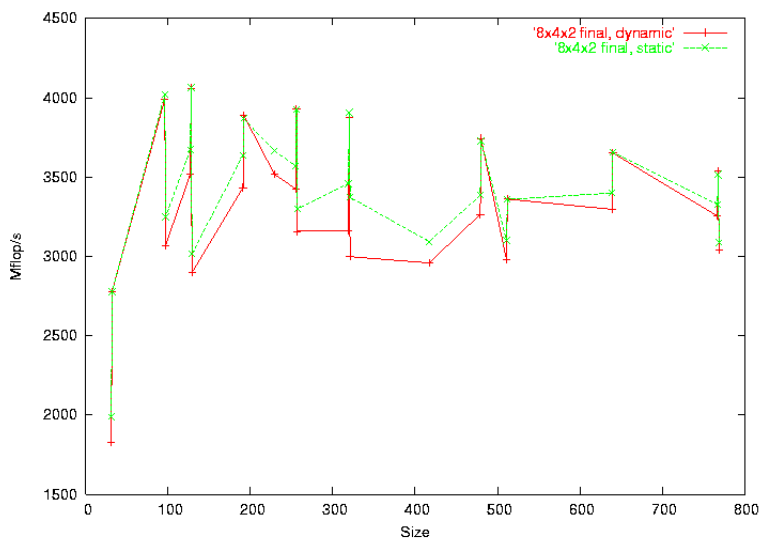


Figure 6: Comparison of static and dynamic allocation of padding arrays.

2.7 Compiler Optimization

We used `ecc` to compile our code, with flags `-tpp2 -IPF_FMA -mcpu=itanium2 -IPF_fltacc -IPF_fp_speculationfast -unroll -Kc++ -O2 -restrict -fno-alias -ipo`, and used `restrict` pointers in our code. We tried to use maximum optimization, `-O3`, but as figure 7 shows, this only decreased performance.

3 Analysis of Results

Initial optimization attempts through L3 cache blocking, padding, and fringe handling yielded only marginal performance improvements, as the L2 block size was already quite large. Register blocking, loop unrolling, and software pipelining provided much larger performance advantages, taking advantage of the Itanium 2’s 128 floating point registers and its pipeline.

The performance of register blocking is not consistent, however. Performance peaks at multiples of 8, since no register fringes exist. In between, performance is significantly lower, with smaller peaks at multiples of 2 and 4, corresponding the other register block parameters. This periodicity is illustrated in figure 8.

We additionally ran our optimized code on both the slower 900 Mhz Itanium 2 and the 2.4 Ghz Pentium 4, with the results in figure 9. The former showed performance characteristics nearly identical to the faster Itanium 2, though performance was lower due to the decreased clock speed. The latter performed much worse, despite its almost two-fold clock speed advantage. This came as no surprise, however, as our code was tuned specifically for the cache size and register set of the Itanium 2 and lacked L1 blocking.

Overall, as figure 10 shows, our code performed about three times as fast as the original blocked version, peaking at 4 Gflop/s and averaging about 3.4 Gflop/s. It is still not as fast as BLAS, which averages about 4.2 Gflop/s. We believe that with further optimization, such as implementing Strassen’s algorithm, further hand-tuning of register blocking, and a good register fringe handling algorithm, our code can approach the BLAS performance.

4 References

- [1] Lam, M., Rothberg, E., Wolf, M., “The Cache Performance and Optimizations of Blocked Algorithms,” *ASPLOS IV*, 1991.

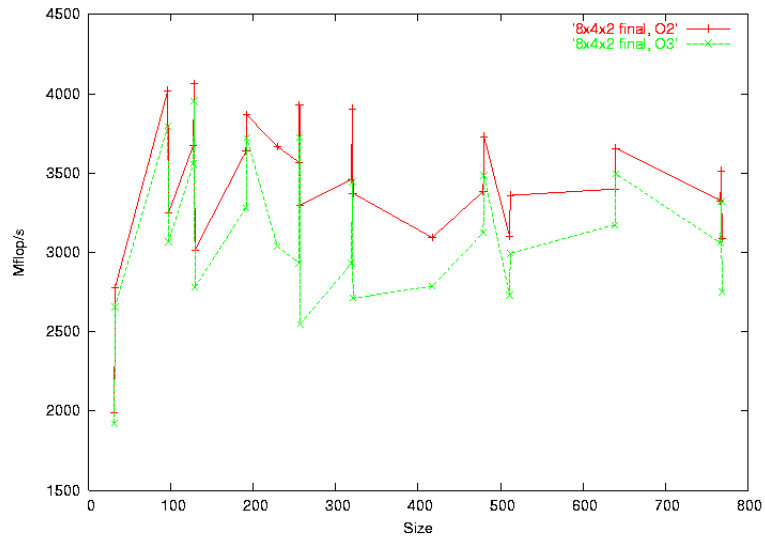


Figure 7: Effects of different compiler optimization flags on performance.

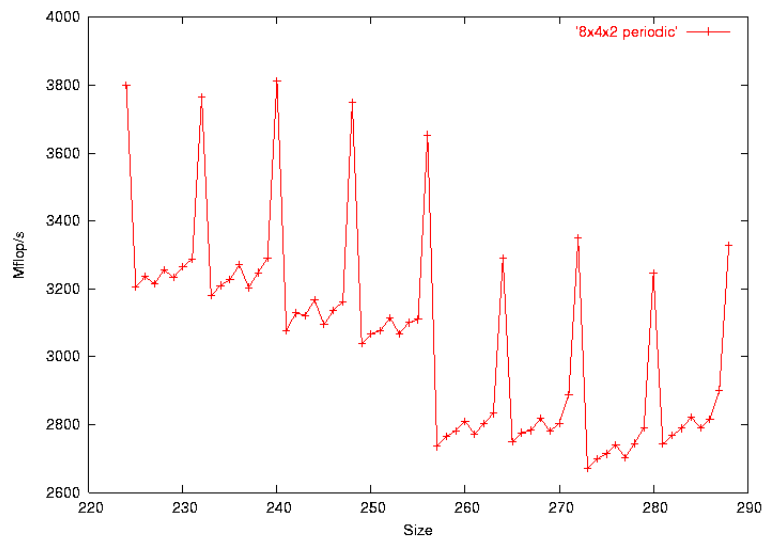


Figure 8: Performance periodicity due to register blocking.

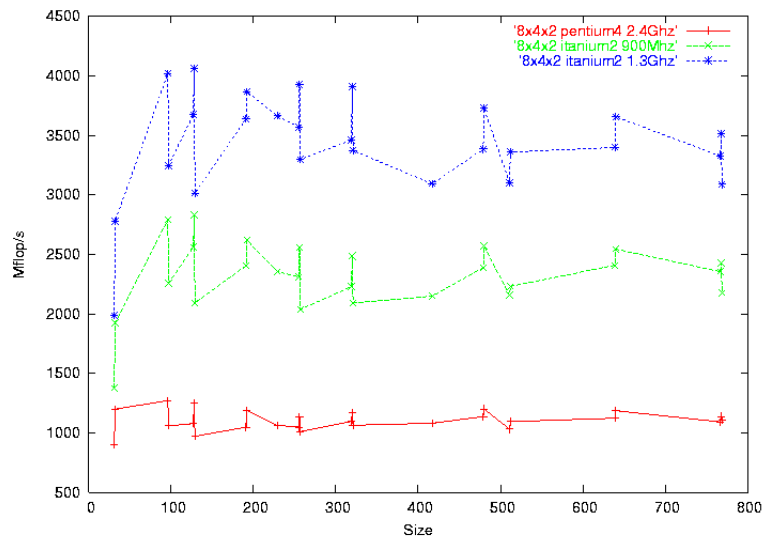


Figure 9: Performance of Itanium2-tuned code on different platforms.

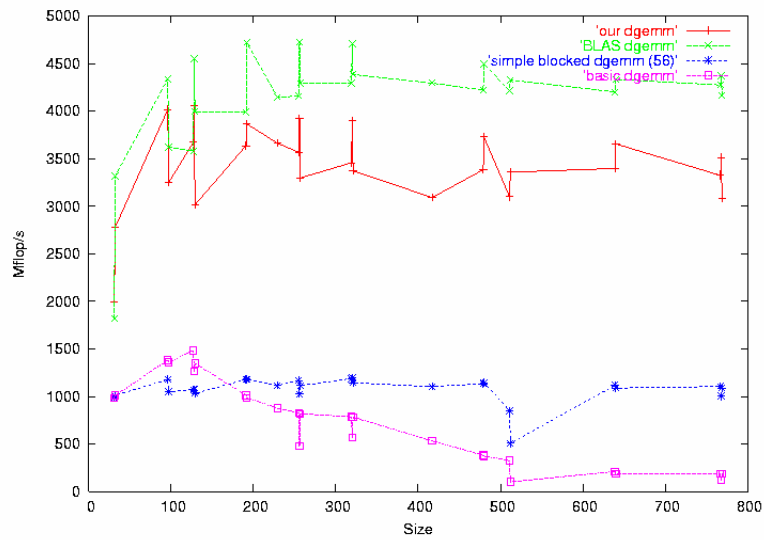


Figure 10: Performance of basic, blocked, BLAS, and final matrix multiply implementation. BLAS numbers provided by Mark Hoemmen.