UNIVERSITY OF CALIFORNIA SAN DIEGO

CSE 260 – Parallel Computation
Programming Lab 1
High Performance Matrix Multiplication
Group ID: G-418-960

Andreas Prodromou, Samuel Wasmundt
<A530-492-30>, <A530-537-28>

January 22, 2014

1 HIGH PERFORMANCE MATRIX MULTIPLICATION

Matrix multiplication is a very common function amongst applications that require high-performance. Various performance improvements were suggested in the past and implemented in the ATLAS library. The goal of this assignment was to optimize matrix multiplication performance on a single core. This report describes the various optimizations implemented by our group and the impact of these optimizations in terms of GFlops/s.

The starter code provided was a matrix multiplication implementation with one level of blocking to improve performance over the "naive" implementation that simply traverses the matrices in order and performs multiplications and additions to compute the final result.

Our group extended the starter source code with a second level of cache blocking and a combination of other performance optimizations such as loop unrolling, array alignment, write combining, and the use of SSE intrinsic commands. SSE commands proved to play a vital role in improving the single-core performance by utilizing the core's SIMD capabilities.

The following sections provide some details over the implemented optimizations, along with some background details explaining why each method improves the performance. Results

are presented in the form graphs to visualize the optimizations' impact.

2 CACHE BLOCKING

As described in the introductory section, the starter code provided us with an algorithm that performs one level of cache blocking on the input matrices and performs the multiplications. Compared to the naive implementation we observe a noticable improvement, especially when multiplying large matrices.

The term "blocking" implies that the input matrices are divided into smaller "sub-blocks" and then these sub-blocks are multiplied with each other in a combination which assures that all the required multiplications took place. The reason for the observed performance improvement relies deep in the core's microarchitecture. During a large matrix multiplication, the core's cache cannot hold all the required memory to support the software's logic. This results in an increasing number of cache-misses, which consequently lead to a major performance hit. As expected, the size of the sub-blocks is another important decision that affects performance. Large sub-blocks might still not fit in the cache, while small sub-blocks do not utilize the cache efficiently.

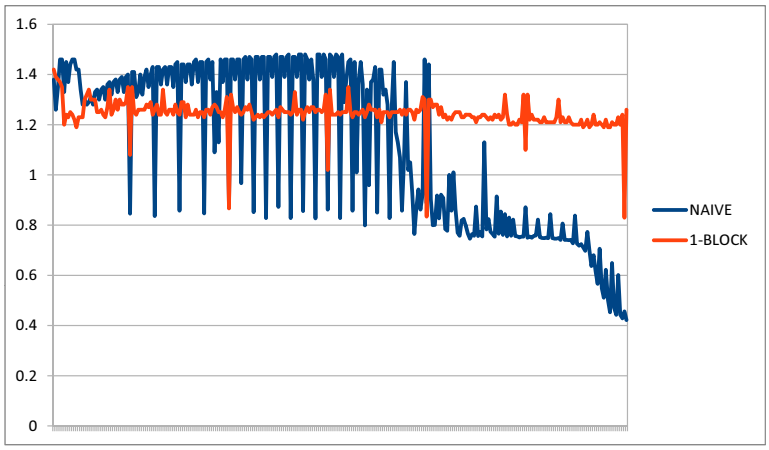

Figure 2.1: Performance with Single Blocking Level

Dividing large matrices into sub-blocks requires more effort by the programmer, however it significantly increases the number of cache-hits by utilizing locality within the code. The

downside to this is that often times the cache characteristics will vary from machine to machine, and as a result fine tuning will need to be done on the end machine. Figure 2.1 displays the performance improvement after one level of blocking over the “naive” code.

2.1 Second Level of Cache Blocking

Along the lines of the given performance optimization, we implemented a second level of cache blocking. Following the same logic, we want the first division into sub-blocks to hold as much data as possible in the L2 cache and then we further divide the sub-blocks into smaller chunks that fit in the L1 cache. This way we utilize all the cache levels of our core as much as possible.

Since this optimization is essentially a recursive call over the first optimization, our source code is essentially performing the exact same steps as the first blocking level with the only difference being that it now performs blocking on a different-sized matrix. Figure 2.2 demonstrates the prformance increase obtained by implementing a second level of cache blocking in comparison to the naive approach and the single-blocking level optimization.

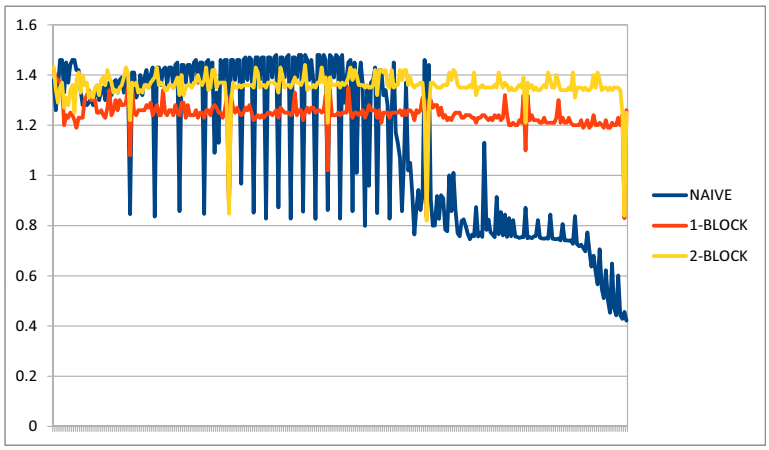

Figure 2.2: Performance with Two Blocking Levels

2.2 Optimal Cache Blocking Sizes

As described earlier, we need to determine the sizes for the two levels of cache blocking in such a way that our program utilizes the two levels of cache efficiently (i.e. Fill the caches as

much as possible without exceeding their capacities). To determine these numbers, we need knowledge of the processor's cache sizes.

In our case, the processor used is an Intel Xeon 5345 with 32KB of L1 cache and 4MB of L2 cache. We begin with determining the optimal sub-block size for the first blocking level. Since this level will position blocks of memory in the L2 cache, we have to figure out a number (BLOCK_SIZE_1) such as the L2 cache is filled as much as possible by holding three sub-blocks (A, B and C, where C = A * B). So knowing that we have an L2 of 4MB we calculate we will have $\sqrt{4MB/8/3} = 209$ entries for each A, B, and C. Likewise, for the L1 cache, which is 32KB, we have $\sqrt{32KB/8/3} = 36$. As an optimization to handle the boundary cases as described below, we made temporary variables A*, B*, and C* subject them to being a multiple of 4. Thus, the final sizes we decided on were 192 for BLOCK_SIZE_1 and 32 for BLOCK_SIZE_2.

2.3 HANDLING THE BOUNDARY CASES

A significant effort was put into handling matrix multiplication with any given M, N, and K for any BLOCK_SIZE_1 and BLOCK_SIZE_2 parameters. For performance, we were able to implement a matrix multiplication that could correctly handle any combination of these without the use of any if conditionals. The initial matrix multiplication performed as many 4x4 SSE matrix multiplications (more on this in the next section) and followed up by performing the proper amount of 'cleanup' multiplications that were unable to be performed by doing 4x4 multiplication blocks. We were able to directly improve upon this performance by actually simplifying the matrix multiplication. Instead of directly handling the boundary cases, we simply take the inputed arrays of A, B, and C and create new temporary arrays A*, B*, and C* that have been padded with 0's to allow for a perfect division into BLOCK_SIZE_1, which further by design can be broken exactly into BLOCK_SIZE_2 sized chunks. Thus, we know that the A*, B*, and C* matrices being passed to our matrix multiplication will always be divisible by 4! By padding the input A, B, and C arrays with 0's we are able to perform 4x4 block matrix multiplications without worrying about erroniously computing C. Further, these arrays are now aligned arrays which also improves performance.

3 SIMD OPTIMIZATION – SSE INSTRUCTIONS

Bang has support for 16x128 bit SSE3 registers. Since we are opperating on doubles, each variable uses 64 bits of space. This effectively gives us 32 usable double precision registers. Taking this into account, we expanded upon the 2x2 SSE Matrix Multiplication provided by Professor Baden in lecture and perform a 4x4 SSE Matrix Multiplication. This requires us to use 16 double precision registers to load and store our result, 4 to load in A, and 4 to load B, for a total of 24 double precision values. Note that the 4 values of A are forming a matrix column, and as such they have to be loaded into different registers. As a consequence, our program utilizes a total of 14 XMM registers. By replacing a naive matrix multiplication that does a simple dot product of each row and collum of A and B, respectively, and stores the result in C with the SSE intrinsics instructions we were able to see a speedup.

In an attempt to improve performance we tried several different configurations of the SSE matrix multiplication. Initially, we had a 2x2 SSE matrix multiplication implemented and contrasted this with that of a 4x4. Due to the underlying hardware on Bang, the 4x4 won out. More concretely, with a 2x2 SSE multiplication being performed it was not enough to adequately saturate the 16x128 bit registers provided by Bang. By bumping up the matrix multiplication to a 4x4, we were able to fully utilize the resources. One thing that we did not experiment with, due to time constraints, was performing multiplication on matrices such as 2x4, 3x4, 4x2, etc. Theoretically one of these could have outperformed our 4x4 is if there was an unforseen register contention.

Figure 3.1 demonstrates the speedup achieved by utilizing the processor's SIMD capabilitites. Some discussion is required to provide a better understanding of this figure. The line SSE-192 demonstrates exactly what we described up to this point in this document. BLOCK_SIZE_1 was set to 192 for the reasons described earlier and SSE intrinsics were used. We observe high spikes in our graph (higher than the SSE-16 case) however we also observe periods of very low performance. On the other hand, SSE-16 (BLOCK_SIZE_1 was set to 16) shows lower peak performance, however very few periods of bad performance compared to SSE-192. The reason for the observed behavior is the alignment of the arrays and padding them with 0's. When the block size is relatively small, less memory locations need to be padded with 0 and as a side-effect less calculations need to be performed. For example, with a block size of 192, an array of size 193×193 would be reshaped into an array of size 384×384 with the extra space being set to zero. With a small block size, the impact of alignment and padding is minimal.

4 OTHER OPTIMIZATIONS

Some of the other optimizations we looked into were transposing B to generate better cache locality, creating temporary arrays to align memory as discussed above, and write combining. Optimizations that weren't fully explored due to time constraints were optimizing TLB misses and exploring with SSE matrix multiplication dimensions. We also attempted loop unrolling within the SSE matrix multiplication portion.

4.1 MATRIX TRANSPOSITION

This proved not to be that beneficial to use because we had already created our temporary B* array that allowed us to load aligned blocks. Though we were able to get an entire line of A, and an entire collumn of B (which corresponds to an entire row of B*), we were unable to effectively use them in an SSE supported way that gained us performance benefits. This is due to the fact that we needed shift operation to get the data where it needed to be able to perform the operations supported by SSE in order to get the correct result matrix.

4.2 ALIGNED TEMPORARY ARRAYS

As discussed earlier, by creating temporary storage arrays A*, B*, and C* that were aligned we were able to see performance increases. A specific thing to note about this is that we saw our

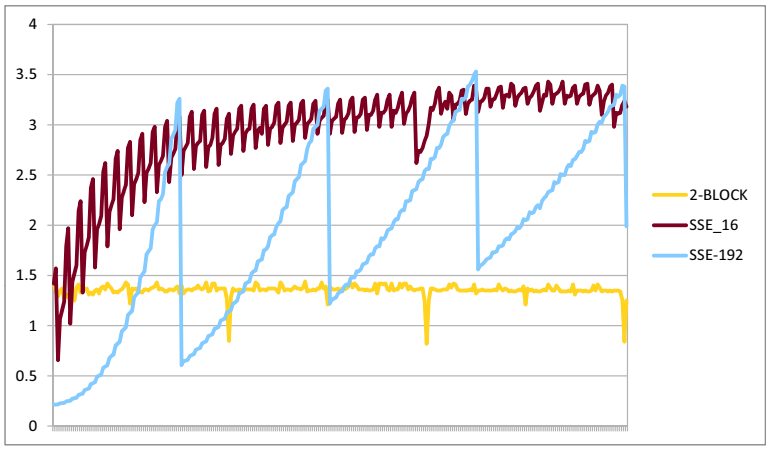

Figure 3.1: Performance using SSE Intrinsics

performance increase from this optimization as the size of the array being operated on grew. This is because the memory is an $O(N^2)$ operation, whereas the computation of the matrix multiplication is $O(N^3)$. Thus, as the size of our arrays grow, this performance optimizations plays a bigger and bigger role.

4.3 Write Combining

This was a simple two fold optimization. Instead of performing multiple writes to one memory location, we created a temporary variable that we updated and only updated the memory location after all local computation was complete. Further, we tried combining these writes whenever possible to utilize write bandwidth. An example of write combining is when we are creating our temporary A*, B*, and C* arrays we do not create them all within one loop, instead we do them in 3 separate loops that allows each to be combined, utilizing the write/read bandwidth of memory. Essentially eliminating a bottleneck that would occur if we need to access 3 different parts of memory for A, B, and C during each iteration. This way we are able to grab larger chunks at a time. Theoretically this is simple enough that the compiler should be able to handle it, but in our analysis we saw improvements having the loops broken apart.

4.4 Loop Unrolling

One last optimization that we attempted was to try unrolling the for loop within our SSE matrix multiplication. In doing so we didn't have a need for any more C registers, as the C

matrix was fully represented already, we did however need to add 4 more A and B registers which brought the total register consumption to 16. After a performance analysis, it was shown to be detrimental to unroll the for loop. We speculate the reason for this to be due to register contention.

5 VERIFICATION

We found the checking mechanism within the benchmark driver to be insufficient to detect all errors within our matrix multiplication. We implemented a self checking method that would fully check the returned C value with what was expected, erroring when an incorrect result was returned. This proved to be very beneficial as we were modifying out code.

6 CONCLUSIONS

This report presents all the optimizations that were performed by our group in an attempt to optimize matrix multiplication as much as possible. Not all the optimizations described are part of the final program, since in some cases certain optimizations ended up hurting performance in the presence of other optimizations.

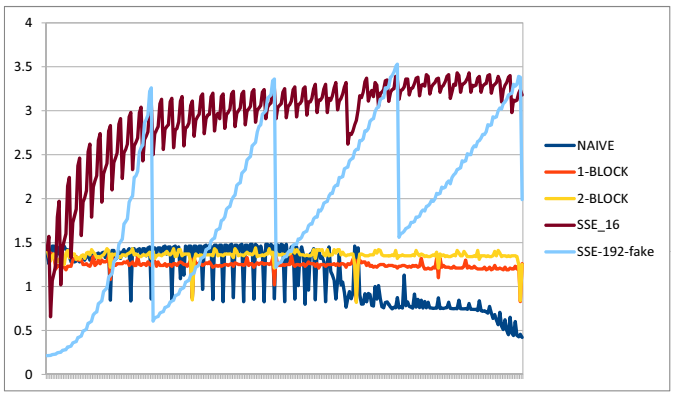

Figure 6.1: A comparison of all optimizations

Figure 6.1 demonstrates some of the optimizations performed and their impact on the overall matrix multiplication performance.