Lab 1: Optimizing Matrix Multiplication
Team ONE

Daniel Putt (ID:1654588)
Vignesh Jeganathan (ID:1552354)

September 17, 2018

Introduction

- Your task in this assignment is to write an optimized matrix multiplication function for TACC Maverick supercomputer (https://portal.tacc.utexas.edu/user-guides/maverick). We will give you a generic matrix multiplication code (also called matmul or dgemm), and it will be your job to tune our code to run efficiently on Mavericks processors.
- Write an optimized single-threaded matrix multiply kernel. This will run on only one core.
- Write your program only for square matrices.
- Matrices are in column-major order.
- DGEMM $\equiv$ Double-precision GEneral Matrix-Matrix multiply
- Our results will be compared to the DGEMM provided by the BLAS library; INTEL MKL

Requirements for lab:

We are required to:

- Optimize square_dgemm() using blocking, for N x N square matrices.
- Only use one core.
- Only use the $3N^3$ algorithm. Algorithms with fewer or less expensive calculations will not be accepted.
- Use double-precision to represent real numbers.

- Only use compiler intrinsic functions and only call into the standard C library.
- Cannot use flags that detect DGEMM operations and replace them with BLAS calls.
- Run on a different machine and note the differences in efficiency and why.

Optimizing

Naive Implementation

Method:

- Solves each C_{ij} in order.
- Can optimize a bit by loop reordering, which can reduce cache misses.

Efficiency:

- Original: $\approx 5\%$, Loop reordering: $\approx 9\%$
- Poor retrieval from memory.
- Poor cache memory usage.
- Poor data reusage.
- Poor register usage.

Blocked:

A note on blocked dgemm:
We quickly found that sizing the block size for the L1 cache didn't help our efficiency. Increasing the block size larger gave a greater increase in speed. We found that blocking for the L2 cache was much more efficient, in some cases.

'Naive' Blocked(given blocked version)

Method:

- We split the given A,B,C matrices into smaller submatrices, in order to utilize different cache memories.
- We want to make the A,B,and C submatrices as large will fit on the cache, while leaving a bit of room to allow some shifting of data.
- We also want to perform as many calculations as possible on the data we move to the L2 cache before we loading in new data.
- The given code/algorithm is organized to keep each C submatrix in the L2 cache until it is completely calculated before loading in a new C submatrix. Although this means that some data from A and B will be reloaded more than once throughout the entire calculation, working with C requires both reading and writing, while A and B only read in data.
- Calculations are done on the submatrices as normal. The given code once again uses the 'naive' method for solving the blocks.

Specifically:

- The actual size of our blocking depends on the system architecture. Mavericks has an L2 cache size of 256,000 bytes.
- N = 103 is a good choice of maximum block side length.

Efficiency Ratings:

- given method: $\approx 11\%$
- Good overall memory usage. Uses L2 cache efficiently.
- Better data reusage.
- Poor register usage.
- Works about as well as optimized naive when matrix size is relatively small.

Our blocked version; 2x2 tiles; polished first attempt

- A good first place to look when attempting to optimize a program is to focus on the lines your program spends the most time on. Particularly, this means the innermost loops and calculations in our program are crucial.
- The given blocked version has poor register usage and reusage. This problem has persisted through all of the versions of code we have analyzed so far.
- There are many 64 bit (8 byte) registers at our disposal. We can do more calculations at a time.
- We implemented register blocking, blocking for 2x2 blocks.
  - For the actual blocking, we used kij loop ordering.
  - For processing and calculating the submatrices, we used a jik loop ordering, where i and j are the rows and columns, respectively. Each calculation is $C_{ij} += A_{ik} * B_{kj}$.
  - We incremented i and j by 2 for every calculation.
  - This means that instead of doing one calculation per iteration of the inner loop, we perform four, and increase the indices i, j by two at each iteration.
  - We are not explicitly coding the register usage here, but rather writing the code in such a way that the gcc compiler can easily vectorize and load the registers efficiently.
  - Pseudocode:

```
for(j = 0; j < N; j+ = 2)
    for(i = 0; i < M; i+ = 2){
        // locally store values for C_{i,j}, C_{i+1,j}, C_{i,j+1}, C_{i+1,j+1}
        for(k = 0; k < K; k + +){
            c_{i,j} + = A(i,k) * B(k,j);
            c_{i,j+1} + = A(i,k) * B(k,j+1);
            c_{i+1,j+1} + = A(i+1,k) * B(k,j+1);
            c_{i+1,j} + = A(i+1,k) * B(k,j);
        }
        //write back these values for C to memory
    }
```

- The best efficiency we achieved with this code was about 28.5% of peak

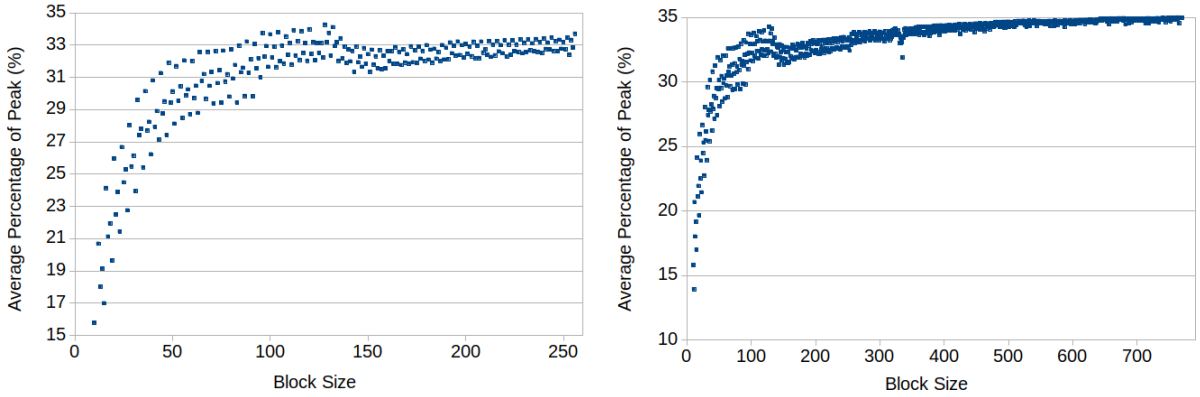
Dealing with cases of odd M or N submatrices

- When we unroll our loops, we need to take care for the event that we encounter matrices/blocks of uneven size.
- We have chosen to include cases that ensure we avoid trying to multiply with elements that don't exist, while still calculating over the odd row/column in the algorithm.
- This has the benefit of not having to reload data to subsequently calculate the fringe rows and columns.
- This also avoids the overhead work of 'padding' the matrices to superficially increase the size.
- It is worth exploring the effect of padding the matrix. It is certainly possible that the process of padding the matrices to make them of even size would be faster than the extra if-else conditionals we have introduced to avoid padding.

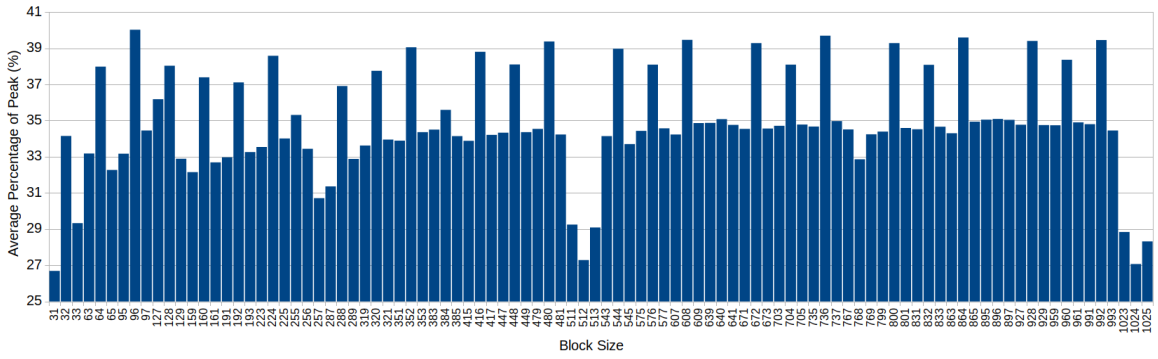
Our blocked version; 2x2 tiles, with third loop unrolled; polished second attempt

- This version reorders the loops into jki ordering.
- It also unrolls i,j,k by 2, instead of just i and j.
- This uses the registers more efficiently.

Different Block sizes vs total avg % of peak:



Block Size 128, for different matrix sizes NxN:



GCC, Flags, and Auto-vectorization

- Using flags: -O3 -fpeel-loops -funroll-loops -fopt-vec-info-all

Building algorithm to support auto-vectorization

- By unrolling our loops to increase the number of calculations per iteration of each loop, we achieve a few things.
- First, the compiler can vectorize more calculations per cycle.
- Second, in the code that isn't unrolled, there are extra registers not being used at each calculation. Unrolling allows for more efficient use of the available registers per cycle.

Blocking for cache levels

- This is where we have some unexpected results.
- We initially thought that choosing a block size that allowed 3 blocks to fit into the L1 register at any given time would give the best results.
- However, we underestimated the effect of block size and calculation time. A larger block size reduces the number of calls from memory for each section of the matrix, reducing the total time spent pulling data from memory.
- We then resized our blocks for the L2 cache instead of the L1 cache/caches.
- This is where our results for our different
- We still didn't see a decrease in efficiency at block sizes greater than 103. Our data, from our jki loop ordering, shows that we finally get a dropoff in efficiency after a block size of 128.

Blocking for registers

- Unrolling our loops allowed us to do more calculations at a time, making it easy for the gcc compiler to autovectorize and use the registers efficiently.
- We tried different block layouts, (1x2, 2x1), but neither gave an increase in efficiency.
- We could try other loop unrolling block sizes in the future. (2x3, 3x2, 1x4,4x1, etc)
- We have listed 2 dimensions above, but we can also move in k by different stepsizes.

Vectorization

- There are registers specifically used for vectorized calculations. Depending on the structure, they can hold 256 bytes instead of the 64 bytes utilized by non-vectorized calculations.
- We have written our code to help hint at the compiler that our innermost loops should be vectorized.
- However, we could use AVX, SSE, or inline assembly to vectorize our code more explicitly.

What we tried and didn't work?

- Register Blocking for 2x1 and 1x2 worked faster than code that wasn't unrolled, with 2x1 slightly more efficient. It still wasn't better than 2x2 tiling.
- We tried some other loop orderings that didn't help with the efficiency.
- We tried reordering the order that blocks were retrieved in our larger blocking loops. This didn't help either.
- We added the restrict keyword for C, and made the elements in A and B constants, in case that would help upon compile time. It didn't affect much.

Comparison of results

- Our second code works better than our first, by better utilization of the registers. We could perform more calculations per cycle this way.
- Our first code was more heavily impacted by cache blocking. A good choice for cache blocking made a great impact on our efficiency.

What we didn't try, but could improve our efficiency

- We could try to improve our efficiency by using inline assembly, avx, or sse.
- This would give us more control in efficiently vectorizing and calculating our matrix multiplication.