CSE 260 Programming Assignment 1

Honam Bang, Myron Liu

Sunday 11th October, 2015

1 Notation

In this document we will use the following notation:

- A, B, C refer to the original matrices
- A', B', C' refer to a *big* block
- A'', B'', C'' refer to a *small* block
- A_R, B_R refer to the rearranged matrices of A, B (respectively)
- n is the width of A, B, C
- b is the width of an untruncated *big* block
- s is the width of an untruncated *small* block

2 Blocking (Rearrangement) and Padding

With blocking, we break the matrix multiplication into a set of smaller problems. The motivation for doing so is that we can fit each sub-problem's data into cache. In this light, naive blocking itself offers some gains, since each block can be made smaller than the cache size (more precisely 1/3 of the cache-size, since each operation $C' = C' + A'B'$ uses three blocks).

This usually isn't enough though. The reason is that the matrices are typically stored row-major, such that the elements of A' will not be contiguous in memory (and similarly for B' and C'). Hence A' will likely spill onto more L2 cache lines than necessary. To prevent the long strides between successive rows, we can make a contiguous copy A'_{copy} of A' in the following manner...

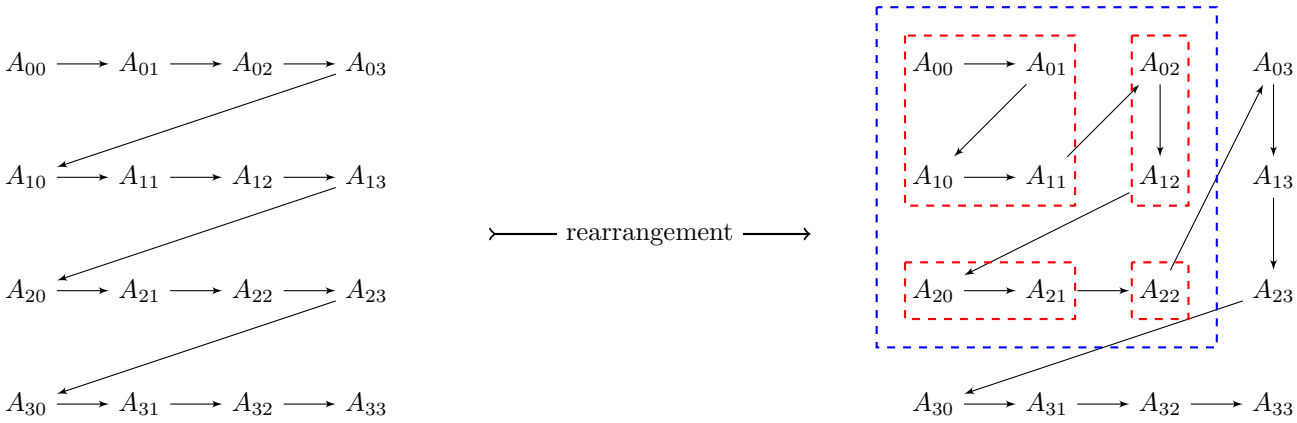
$$A' = \begin{bmatrix} a & b & c & \dots \\ d & e & f & \dots \\ g & h & i & \dots \end{bmatrix} \longrightarrow A'_{\text{copy}} = \begin{bmatrix} a & b & c & \\ d & e & f & \\ g & h & i & \dots \end{bmatrix}$$

... where we have simply removed the stride between each row. Subsequently, we use A'_{copy} for our computations. For two level blocking, we might consider making a copy of A'' as well, so that A''_{copy} fits continuously into L1 cache. However, copying itself incurs some overhead. Shuttling data from L2 to L1 is substantially faster than shuttling from DRAM to L2 though, so while it is important to make copies of A', making copies of A'' might result in a net loss of performance.

More importantly, copying at deeper levels introduces excessive redundancies. Suppose we divided each matrix *evenly* into a grid of $N \times N$ big blocks, and each big block *evenly* into $M \times M$ small blocks. Each big block gets used N times, within which each small block gets used M times. Therefore, we would need to copy each small block NM times, which is more than the N times we would need to copy the big block in which it resides.

We actually went a step further than block-copying. As aforementioned, copying A' and B' more than once is actually a bit wasteful if we can just do it once. With this consideration, we *rearrange* A, B at the start, so that we no longer need to deal with copy operations. Since we block at two levels, we rearrange recursively. More concretely,

suppose the matrix size is $n = 4$, the big block size is $b = 3$, and the small block size is $s = 2$. In this case, the rearrangement proceeds as follows...



... where the arrows indicate the order of elements. As can be seen, by performing this rearrangement a single time, we ensure that each block (regardless of size) is stored contiguously in memory. When rearranging B, the blocks are traversed in column major order. **However** the individual elements within the smallest blocks of B are still ordered row-major. The reason for this is that we need the same orientation for the vectorized 2×2 matrix multiply that was described in lecture.

C is the exception. We opted to make a working copy C'_{copy} of C', which we write back to C' after completing the blocked-contraction. Unlike copying A' or B', copying C' is not wasteful, since each block of C is only accessed once.

Although the figure above uses an odd width for the big block, in practice we always use even block sizes at both levels to facilitate the vectorized 2×2 multiply. For similar reasons, if A and B themselves have odd-size, we pretend they are padded by an extra row and column of zeros when we perform the rearrangement. In the next section, we discuss these restrictions.

3 Vectorization

For vectorization, Bang provides a set of 128 bit wide registers. It is substantially faster to load and store from these registers if the data is aligned on multiples of the register width. Since two 64 bit doubles fit snuggly into a single register, we would like each small block (within which the 2×2 multiplies actually take place) to start at an **even offset** from the beginning of the A_R, which we manually align on 128 bits.

One way to enforce this condition is to pad A with *internal* rows/columns of zeros, but this clutters the matrix with meaningless entries. It is much cleaner to simply avoid odd block dimensions, and pad the matrix minimally if needed.

Another reason for enforcing even block dimensions is so that we can rely fully on the 2×2 vectorized multiply for all the arithmetic. With odd block sizes, we will be left with a 2-vector outer-product, which must be handled separately. We want to avoid vacillating between methods, which can becomes a source of inefficiency.

All of this being said, we would like to note that odd dimensions are less of a problem with block-copying. This is because the copy can be manually aligned and padded. However, our rearrangement approach demands a different solution.

4 Other Optimizations

We also performed some loop unrolling to speed up our implementation. For the most part, we unrolled the more inner loops of each function, Unrolling the outer loops in addition to the inner ones led to excessively verbose code, which negatively impacted performance. What we find strange is that cachegrind doesn't show any increase in instruction misses when we unroll aggressively, so the impact on performance must be due to some other aspect. It is possible that the compiler cannot effectively inline function calls, but we did not test this hypothesis. Another point

of uncertainty is that user-specified inlining, whether with the *inline* keyword, or by manual insertion, sometimes slowed down performance (again, evidently not due to instruction misses).

5 Results

We were able to achieve up to 3.53 Gflops/s with the aforementioned optimizations. We would like to note the relatively uniform performance between varying block sizes, and also the gradual increase in performance for larger matrices (see Fig.1). We discuss possible sources for these phenomena in the next section.

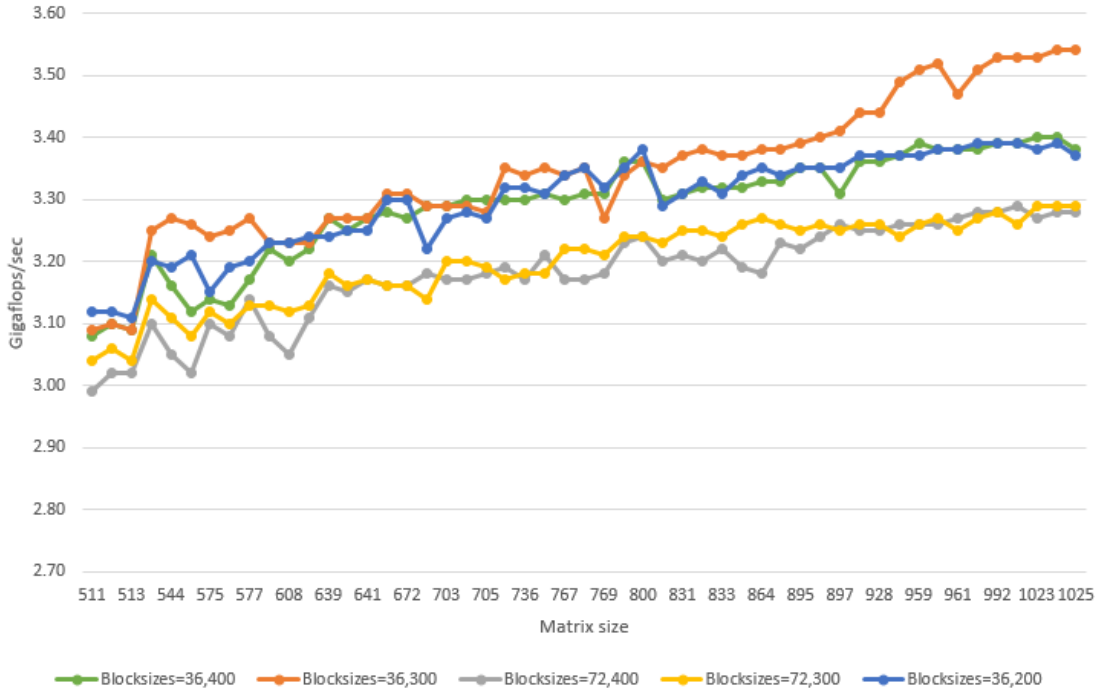


Figure 1: Matrix multiplication speed for various combinations of block sizes

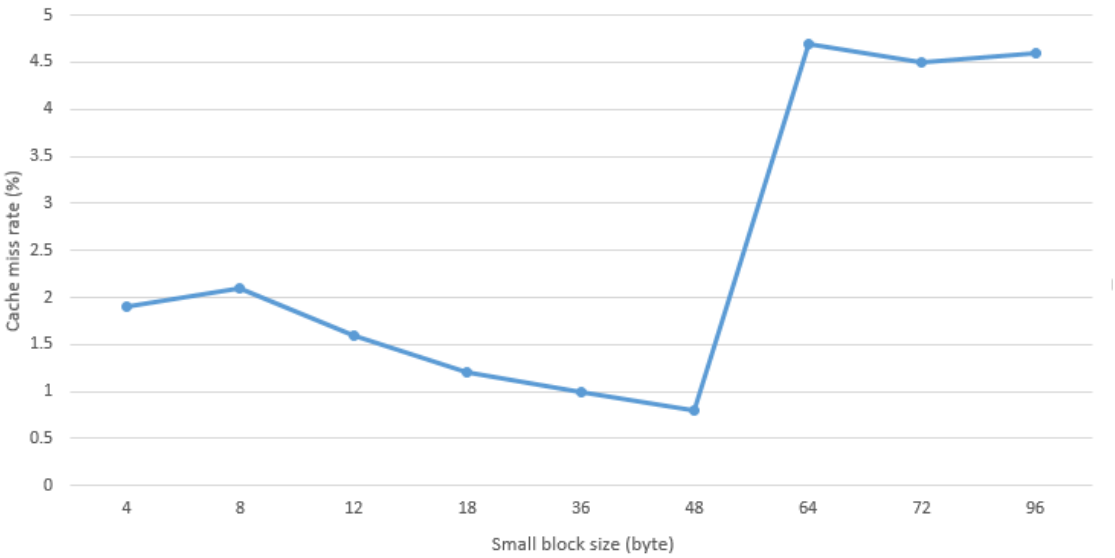


Figure 2: L1 cache miss rates for various small block sizes with fixed big block size $b = 418$

6 Discussion

When discussing with some of our peers (who used block-copying), we noticed that our implementation was somewhat more robust to varying block dimensions. We aren't completely sure, but the following observation might shed some light on the issue.

With block-copying, it's not clear where the next block's copy will be allocated. Therefore, the machine will necessarily load each copy into cache only *after* its creation. In contrast, since we have rearranged A and B so that successively-accessed blocks follow one another tightly (recall B_R's blocks are arranged column-major), a sufficiently clever machine will know to load a section of the next block in the row/column while we are working with the current block.

In addition, consider what happens when we load three copied 418×418 blocks into L2 cache. We still have 928 Bytes of unused memory in L2 cache. In reality, the machine probably uses this space for other purposes. But at least in principle, we would like to be able to use it to hold matrix elements. By rearranging, this space will not go wasted, since portions of the next blocks to be used will go into these 928 Bytes.

We believe our suspicions to be valid because changing the big block size b to something substantially smaller than 418 (which is the upper bound for fitting three $b \times b$ blocks into a 4 MByte L2 cache), has only a minute impact on performance. For instance we went from $\{b = 418, s = 36\}$ all the way down to $\{b = 180, s = 36\}$ without noticing too much change in Gflops/s (below this size, we sometimes exceed the error-bounds, but we believe this is due to the non-commutative nature of finite precision arithmetic).

Changing the size of the small block s seems to have a larger impact: decreasing it from the expected optimum of $s_{\max} = 36$ results in a noticeable performance hit. By our previous reasoning, this should not be the case, since the small blocks within a big block are also accessed contiguously in rearrangement. Our best guess for the source of this discrepancy is that the machine becomes much less smart the closer we get to the CPU. The module that controls data-transfer between L1 and L2 is like a kindergartner; the module that controls data-transfer between L2 and DRAM is like a college student; and the compiler is like Carl Friedrich Gauss.

Using cachegrind, we observed that the *naive* matrix multiplication doesn't give any L2 cache misses. This is probably because a single row and a single column can easily fit into L2 cache. However, we do get $\sim 0.2\%$ L2 cache misses for $n = 511$ with our implementation (independent of b, s). Upon investigating the cachegrind output file, we discovered that most of our L2 cache misses occur in the rearrangement phase. This is likely because we allocate new memory for A_R and B_R, and $sizeof(A_R) + sizeof(A)$ far exceeds the size of L2 cache. We have to go through each element of both matrices in rearrangement, so the number of memory accesses is independent of b, s. However, in principle, the choice of b, s probably does have a subtle impact, since they affect the order in which we access elements, which can itself affect performance.

This is a point for improvement. If we can perform the rearrangement in place, then we can reduce the number of L2 cache misses at the cost of undoing the rearrangement later on.

As can be seen from Fig.2 $s = 36$ gives near minimal L1 cache miss rate. This is to be expected because three 36×36 fit snuggly into 32 KByte L1 cache.

The steady increase in performance with increasing matrix size is a bit of a mystery to us. All we can say is that cachegrind tell us that the L2 cache miss rate is smaller for the larger matrices, with almost all misses occurring in the rearrangement pass, as always. For all we know, the effect may actually be periodic as we increase the matrix dimensions even further.

7 Conclusion

Rearranging, as opposed to copying, seems to provide some unique tradeoffs. On the one hand, all memory accesses are made contiguous in a single pass, so rather than optimizing the block size so that three blocks can fit tightly into cache, we can optimize the block size so that a block-row of A and a block column of B fit into cache. Rearrangement makes more efficient use of memory because L2 cache can be filled with useful data to the brim. This manifests itself in more stable performance that depends only weakly on the block size. The downside to the rearrangement strategy is that the pre-processing has high memory demands, resulting in cache misses. We suspect that the reordering phase is where our implementation suffers the most, and is our greatest priority for future optimization.