Masterpraktikum Scientific Computing – High Performance Computing

Gaurav Kukreja, Evangelos Drossos

Exercise Sheet 1

Exercise 4

4.a.

In the given implementation of Matrix Multiplication, the access pattern for input matrix B is very suboptimal. Input Matrix A is read linearly, and the next element to be read can be prefetched hugely assisting vectorization. However, this can not be done for B.

This is especially suboptimal, in the case where the size of the matrix is such that the one row of the Matrix B, uses one memory page. In this case, every access of the next element in Matrix B, causes a page fault. For page size 4KB, and matrix width 512, this performance impact can be felt.

The total number of writes are n^2, and the total reads are 2*n^3. Most of the accesses to elements in Matrix B will lead to a cache miss, for large matrices.

4.b.

The compiler automatically vectorizes the code. The compiler interchanges the loops in such a way, that the memory access is optimized. Manually, this can be done as follows.

```
for (i = 0; i < n; i++)
{
        for(k = 0; k < n; k++)
        {
                r = a[i *n + k];
                for(j = 0; j < n; j++)
                {
                        c[i * n + j] += r * b[i * n + j];
                }
        }
}
```

By performing this change, Matrix B is now read linearly from memory. This is optimal for vectorization.

No compiler directives were needed to vectorize the code. The compiler auto vectorizes the

code, when the optmization level is set to O3.

4.c.
Please look at attached Ex4_plot.pdf for the plots and corresponding discussions.

4.d.
Compiler options were not needed in this case. Simply specifying the optimization level provided best results.

4.e.
The performance seems to increase until matrix size ~1024. Thereafter it falls suddenly. This has been discussed in Ex4_plot.pdf too. This observation is consistent in given implementation with and without vectorization. The reason for this is that L3 cache can hold at most matrices with width ~1140.

However the cache blocked implementation is able to utilize the cache in such a way that this problem does not occur.

4.f.
The performance is not stable in the sense that it does not stay consistent with the matrix width. For some matrix width, e.g. the case shown for cache blocked implementation on matrix size 4096, there are sudden dips in performance, owing to collisions on cache lines and other such issues.

4.g.
Without interchanging loop
Without any optimization, the performance is growing rapidly until matrix size 63. The matrices are being stored in L1 cache so far. Hereafter, the matrix can no longer be stored in the L1 cache, and the performance rise becomes slower.

Performance dips suddenly beyond matrix size 1024, because of L3 misses. Matrices of width ~ 1140 can be stored in the L3 Cache. When the matrix size increases, performance dips suddenly

With Loop Interchanging
It should be noted that the MFLOPS achieved with compiler optimizations and vectorization is much much higher than the MFLOPS without optimization.

The rise of the performance, and the dip beyond matrix size 1024 follows the same pattern as for the unoptimized implementation.

With Cache Blocking

The cache blocked implementation of the Matrix Multiplication, enable us to acheive consistent performance even beyond matrix size 1024. We do see a sudden dip in performance at width 4096. This could be because of collisions on cache line.