Technical University of Denmark

High-Performance Computing

Course 02614

Assignment 1

Authors:
Oskar Hint, s161559
Mikkel Jensen, s123184
Philip Rasmussen, s103124

January 6, 2017

Contents

1 Introduction

2 Assignment

2.1 Nat

The Sun Studio compiler is used, and therefore the driver required is matmult_c.studio.

First, the goal is to write a native function, which performs a matrix-matrix multiplication. The shape has to be suitable for the operation, but it is arbitrary within the limits. Regarding matrix indexing, we use the double pointer representation A[i][j].
The general function prototype used is displayed below:

Listing 1: Function Prototype

```
void matmult_NNN(int m, int n, int k, double ** A, double ** B, double ** C)
```

The function prototypes are all declared in the header file ass1_lib.h. The functions themselves are described in the ass1_lib.cpp file, where the ass1_lib.h library is included.

The native function takes the integer arguments m, n and k, as well as the double arguments **A, **B and **C. The double arguments are the actual matrices, which in the case of matrix A and B are generated within the supplied driver based on the submitted shape parameters. In the case of C, only the memory is allocated. To perform the matrix-matrix multiplication a for loop for each of the 3 shape parameters is required. Three loop variables i, j and l are introduced for the matrix leading dimensions m, n and k. The C matrix has the leading dimensions of m and n, C[i][j]. The function is displayed below:

Listing 2: Function Prototype

```
void matmult_nat(int m, int n, int k, double ** A, double ** B, double ** C){
        for(int i = 0; i < m;i++){
                for(int j = 0; j < n;j++){
                        C[i][j] = 0;
                        for(int l = 0; l < k;l++){
                                C[i][j] += A[i][l]*B[l][j];
                        }
                }
        }
}
```

Notice the += sign, which is caused by the fact, that when the function loops over l, the multiplications for the different values of l are added to the

2.2 Comparing the permutations

The performance of the six different functions resulting from the possibilities of permuting mnk are tested with square matrices with memory footprints ranging from a few kB to hundreds of MB. The results are seen in figure 1 with compiler option −fast enabled. The performance of the six functions behave in pairs depending on the last index which corresponds to the innermost for-loop. The functions pair up depending on this index because it is the loop being executed the most times, and so any performance issue coming from this loop is executed the most times as well. As seen in the figure 1, the permutations with m as the last index is a factor of ~ 2 slower than the other four permutations for memory footprints smaller than L2 cache. This is because m is the column size of matrices A and C, so by having the m loop as the innermost loop, a lot of cache misses will happen due to C being a row major language, and looping along the columns as the fast loop will reduce the speed significantly. When the memory footprint exceeds the L2 cache size, the permutations with k as the inner loop drops to the performance level of the permutations ending with m. The k-parameter is the row size of A and column size of B, so in theory we should observe low performance, but it is likely that the compiler has optimized the structure in a way that is only beneficial as long as the data structures involved will fit on the L2 cache. Finally, the permutations with the n-loop as the fast loop perform well even when exceeding the L2 cache size. n is the row size of B and C, so looping over this variable fits well with C, and it gives the best possibilities for the compiler to optimize. When the L3 cache size is exceeded, the performance drops due to the fact that the data must now be transferred from the memory to the caches, and the bandwidth from the memory to the caches is much smaller than from cache to cache. The drop is not large, so it is possible that the compiler utilizes prefetching to reduce the effect of the decreased bandwidth. It is also worth noting that mkn performs better than kmn for all memory footprints, and the reason here is that the loop over m is the slowest, and thus to achieve the best performance, this loop must placed as the outmost loop.

2.2.1 Analyzer tool

We expect better-performing permutations to have less cache misses. To validate our hypotheses regarding cache hits we conducted a profiling experiment. That is, we fixed the matrix dimension size to 724 which corresponds to 12MB memory footprint, ran each of the six permutations (without compiler optimizations) using the collect command and viewed the results using Oracle Solaris Studio Performance Analyzer. To be able to make direct comparisons, we set the minimum runtime to zero and max iterations to one. This way each permutation is run for a single iteration and we can compare cache hits in absolute terms.
Level 1 and 2 cache hits and misses as well as CPU time of the permutations are presented in table 1.

Table 1: Cache counters in millions with 12MB memory footprint

	L1d hit	L1d miss	L2 hit	L2 miss	CPU time
nkm	6849	755	704	50	2.410
nmk	7229	373	373	1	2.030
mkn	7599	0	0	0	1.810
mnk	7229	376	328	47	2.310
knm	6849	755	704	51	2.350
kmn	7599	0	0	0	1.820

INTERPRET RESULTS

When the memory footprint

- Introduce functions

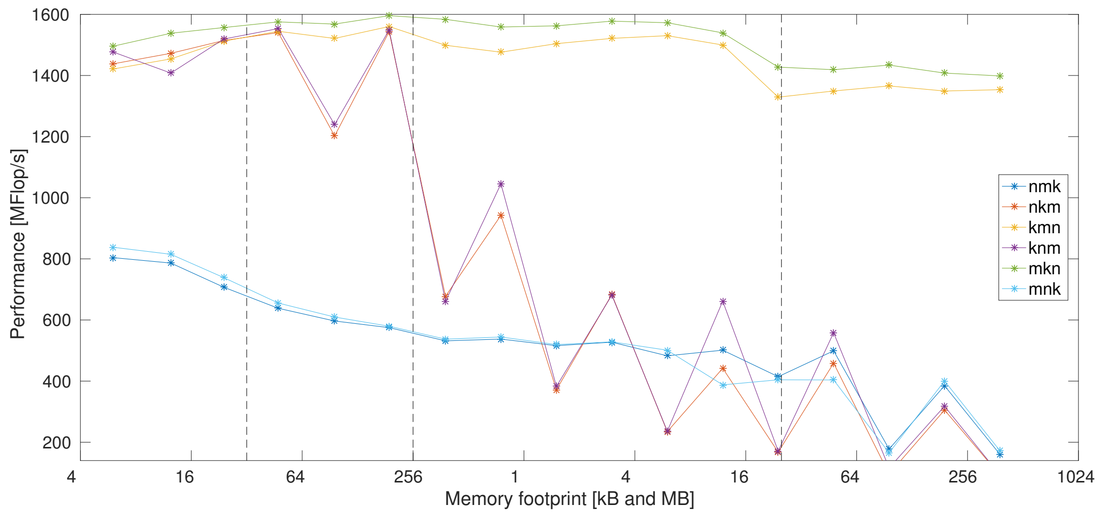

Figure 1: This is a caption

- Show library vs. native
- introduce the permutations WITHOUT -fast and show the performance vs. memory footprint for all permutations
- Optimize by compiler options (-fast, -O3, -prefetch etc. etc.)
- Performace Analyzer tool experiment(s), relate results to memory footprint-performance plot.
- Blocking and find optimum block size
- Compare blocking to the best w/o blocking

3 Conclusion