Chapter 1

Summary

Chapter 2

Problem Statement

From the assignment we were given the following tasks

- Write a function that performs matrix matrix multiplication on arbitrary sized two-dimensional arrays.
- Make a separate implementation for each way the loops in the above function can be ordered and test which one is the fastest.
- Compare the result of our implementation to DGEMM(), a similar function in the BLAS library
- All the functions and the call to DGEMM() should be wrapped into a single library.
- Implement a blocked version of the matrix matrix multiplication function and identify the block size that gives the best performance in regards to L1 cache. This result should be compared to the timings for the library function.
- Compare different compiler settings to see how they affect the result.

Chapter 3

Description of Hardware and Software Used

- stuff from lscpu - compiler and libraries used. version numbers to

Theory

The matrix multiplication is a critical operation in computer science, both because it is the base of several practical problems and because it is still relatively expensive.
This operation, in fact, is really the fundamental brick of several interesting problems, from the resolution of big linear systems, often applied to physics, to resolution of discrete Fourier transforms, used for example to DNA splicing problems. On the main challenges of this operation is not related to the pure computational power, but with the amount of data that it requires: nowadays, as CPU/GPU evolved and still evolves faster than the memory chips, the real challenge is not to compute the result in a short time, but making the computational unit able to produce that result. In other words, it is more expensive bringing the right data at the right time, to the right chip, than to actually produce the result.
Algorithms in general nowadays, are so much affected by this behaviour, that the fastest ones are the algorithms that make a better usage of the caches. From this point of view, matrix multiplication is considered a good example of what a typical algorithm should look like these days. It is therefore used as a base to test the ability of high performances computers to deliver outcomes.

3.1 Loop Optimization Techniques

Since scientific computing often requires the looping over large data sets there has been developed multiple optimization strategies that targets different areas. These areas are:

spatial locality Reuse all the data fetched in a single cache line

emporal locality Reuse data in previously fetched cache lines

loop overhead

3.2 Loop Blocking

A common technique to improve temporal and spatial locality at the cost of loop

Chapter 4

Implementation

4.1 Basic matrix multiplication

The basic matrix multiplication algorithm follows the most standard n^3 implementation:

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

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

Its important to note that the C matrix initialization has been moved outside the main loop, in a separate cycle. This doesnt change the code complexity, but it helps to recognize that the core of the multiplication algorithm is completely independent from the order of the 3 nested loops that wrap it, so they can be reordered in any possible permutation (6 different ways).

In order to define with of the 6 permutations is the best, its important to understand that the 3 different matrices are actually linear array accessed by an index in the form:

X[i*c+j] with c constant, i and j loop variables.

The best performances will be reached when the matrix X will be read/written sequentially, so when itll be wrapped by an external loop on i and an internal loop on j. We can express this with the dependency i =¿ j. If we define the dependency graph for all the 3 matrices, we get:

$$C[j*n+i] : j => i : m => n$$

$$A[j*k+l] : j => l : m => k$$

$$B[l*n+i] : l => i : k => n$$

There is only 1 combination that satisfied all these constraints, and that is mkn (external to internal). We expect the best performances with this combination.

4.2 Blocked matrix algorithm

In literature, its suggested to use a block version of the naive algorithm to improve data locality: this approach intuitively do a better usage of the cache and improve the overall performances. The idea behind the algorithm is to split both A and B in squared blocks, small enough so the cache L1 can store 3 of them. In this way 2 blocks can be multiplied using the naive algorithm with high performances. Externally, other 3 nested loops will apply the same naive algorithm to combine the blocks:

```
void matmult_blk_internal(int m,int n,int k, int sm,int sn,int sk,double *A,double *B,
    double *C, int bs)
{
  int mmin = min(m,(sm+1)*bs);
  int nmin = min(n,(sn+1)*bs);
  int kmin = min(k,(sk+1)*bs);
  int smbs = sm*bs;
  int snbs = sn*bs;
  int skbs = sk*bs;

      for(int i = snbs; i < nmin; i++)
    for(int l = skbs; l < kmin; l++)
      for(int j = smbs; j < mmin; j++)

        C[j*n+i] += A[j*k+l] * B[l*n+i];
}

void matmult_blk(int m,int n,int k,double *A,double *B,double *C, int bs)
{
  //bs=50;
  int nm = n*m;
  for(int i = 0; i < nm; i++)
  {
    C[i] = 0;
  }
  int m_bs = m/bs;
  int n_bs = n/bs;
  int k_bs = k/bs;

  for(int j = 0; j <= m_bs; j++)
    for(int l = 0; l <= k_bs; l++)
      for(int i = 0; i <= n_bs; i++)
        matmult_blk_internal(m,n,k,j,i,l,A,B,C,bs);
}
```

This approach maintain the, not only the same complexity of the naive algorithm, but also the exact same number of floating point operation (the only overhead is on integers), and the same numerical precision of the previous code. The proof is left, but the reader can refer to the literature to further details.

At this point, one important detail, is making sure that the compiler will inline the internal function in order to gain the best from the method. From our test, the most common optimization settings (-fast) included that.

4.3 Scripts and tests

In order to test our results, a set of python scripts have been developed. They automatically takes care of:

- rebuilding the library
- executing the tests
- store in a file the compiler option that has been used
- store in a file the description of the machine used
- store the data in a dat file
- generating a eps plot, using a predefined template.

Each test set has been run several time to assure the numerical stability of the results, before selecting an execution as a final

Chapter 5

Results

5.1 Compiler parameters

//TODO Alexander

5.2 Comparison against dgemm

Against the dgemm library function, even with the same compiling options, the implementation provided of the matmult_nat() method has been proven poor, according to the graph shown in Figure **??**. The figure shown that, for every possible permutation of the cycle order (including so the naive implementation), dgemm outnumbers it.

//TODO figure

The reason for that ... (investigation required)

5.3 Permutations of m, k, n and performance

The three cycles performed in the naive algorithm can be rearrenged in six different ways: there are in fact six possible permutations of the three letters. The performance results provided by the execution of the six algorithms on some standards memory footprints are provided in Figure **??**. The test was made on the product of square matrices $nxnxn$, where n was calculated using the following formula, to relate it to the memory footprint:

$$n = \sqrt{\frac{F_{byte}}{8 \cdot 3}}$$

//TODO figure

The best results of the graph are the ones where the most internal cycle is cycled over $[1, n]$, and in particular the best implementation has been proved matmult_mkn, followed by kmn. The worst ones, instead, have been proved nkm and knm, the ones with the most internal cycle in m.

The explanation of this behaviour is that mkn is the best implementation because is the one that causes the less cache misses. If the most internal variable looped is n, in fact, each time we start reading an element from matrices B and C, that have both n columns, all the row is read. This causes less misses in the cache, leading to a better perfomance for both the mkn and kmn bheviours. Since the A matrix has k columns, also if we loop through k before looping though m we got a better performance, for the same reason. This explains why mkn is slighly better that the kmn implementation.

The worst case scenarios, instead, do the opposite: they privilegiate columns over rows, resulting in a better performance if the matrix was stored column-wise, like in Fortran, but leading to disastruous results in C, where the rowmajor order is preferable.

To verificate our assumption, we checked the number of cache misses for both the best and the worst implementation, resulting in the following table:

5.4 Performance of n,k,m implementations with different sizes

The results and conclusions presented in the previous section held keeping in mind that the matrices we tested were square ones, where $n = m = k$. We repeated the same tests for some odd-sized matrices, i.e. where one of the value was kept very low. The other two values were raised accordingly, in order to maintain the same memory footprint.The results we came up with are really interesting, and are presented in the three Figures ABC. The smallest valuse has always been kept to four elements.

//ABC

For small m and n the change is almost irrelevant. Surprisingly, for small ks, there is a drop in performance of the ones that before where the most performant results: mkn and kmn. This happens because

//image/scheme to explain

5.5 Blocked Matrix performance

We recall that the blocked version, introduced in section //TODO, uses the best algorithm for naive multiplication, mkn, in its inner loop. We experimented the block size against the MFLOPs generated for a generally big square matrix (with 2048kB memory footprint). The results are reported in Figure //TODO (green line). The performances of the algorithm increased at the rising of the block size. This means that the optimum is the highes possible value of the block size, i.e. the matrix itself. In conclusion, the blocking algorithm is useless, because it will be always beaten by the nkm algorithm.

To find an approximate optimum for the block size, it was necessary to substitute the algorithm in the inner loop with the worst one (knm). Repeating the test, the data showed a plateau curve, having its maximum in the range 27-37. After 40, the performances start to decrease, because the L1d cache starts to fill up. At 40, in fact, we have an estimated data cache size of $3 \cdot 40^2 \cdot 8 = 37.5kB$, which is sligthly higher than the l1d cache size ($32kB$) due to the prefetching effect.

Chapter 6

Conclusion

Bibliography

[1] T. Moeslund, *Image and Video Processing*, Aalborg University, 2 ed., 2010.