Assignment 1: Matrix Multiplication

02614 High-Performance Computing

Due 6th of January 2012

Henning Weiss - s062407

Rasmus Bo Sørensen - s072080

Evangelia Kasapaki - s114598

1 Summary

In this report we will implement three different matrix-matrix multiplication functions and evaluate the performance of the functions in the number of floating point operations per second. The functions use different algorithms for multiplying the two matrices. The performance evaluation is done using the Analyzer from the Sun Studio. We have visualized the results of the performance evaluation in graphs, which are presented in this report.

2 Introduction

Many physical problems can be reduced to systems of linear equations, thus they can be solved efficiently with linear algebra. Matrix multiplications are very calculation intensive and therefore they are a good benchmark for HPC systems.

We are to implement and evaluate three different matrix-matrix multiplication function used in a HPC system. The three different implementations are:

1. A simple matrix multiplication function, with three nested loops.
2. Invoking the DGEMM function from the sun performance library.
3. A blocked matrix multiplication function, with the block size adjusted for the processor characteristics.

The functions are evaluated in MFLOP/s as a function of memory footprint, to give an idea of the efficiency and to compare.

3 Theory

3.1 Matrix multiplication

Multiplying two matrices $C = A \times B$ can be done by the following simple algorithm:

$$C_{ij} = \sum_{k=1}^{p} A_{ik} \cdot B_{kj} \tag{1}$$

This algorithm has the time complexity $\mathcal{O}(n^3)$, for A and B being squared matrices. Whereas an matrix addition has the time complexity $\mathcal{O}(n^2)$, for A and B being squared matrices.

3.2 Simple_mm

The simple_mm function uses the algorithm described in section 3.1. This algorithm is very easy to understand, mostly because it is similar to the way we multiply matrices by hand. The simple_mm is therefore a good baseline for evaluation the performance of other implementation. An easy optimization of our simple_mm is to check whether an element in the current row is zero, in this case there is no need to go through all the elements in the corresponding column.

The simple algorithm has three nested loops, the two outer loops can be interchanged and the algorithm still reaches the correct result. The compiler can interchange these loops to optimize the memory accesses for the cache.

3.3 Block_mm

Block_mm function performs block matrix multiplication. Matrices A and B are partitioned in a number of submatrices, i.e. blocks, and the product is evaluated involving only operations on the smaller submatrices. For block dimension size S matrices A and B are partitioned in blocks as follows:

$$A = \begin{bmatrix} A_{11} & A_{12} & ... & A_{1r} \\ A_{21} & A_{22} & ... & A_{2r} \\ ... & ... & ... & ... \\ A_{p1} & A_{p2} & ... & A_{pr} \end{bmatrix}$$

$$B = \begin{bmatrix} B_{11} & B_{12} & ... & B_{1q} \\ B_{21} & B_{22} & ... & B_{2q} \\ ... & ... & ... & ... \\ B_{r1} & B_{r2} & ... & B_{rq} \end{bmatrix}$$

The multiplication product C is formed as the $p \times q$ block elements matrix

$$C = \begin{bmatrix} C_{11} & C_{12} & ... & C_{1q} \\ C_{21} & C_{22} & ... & C_{2q} \\ ... & ... & ... & ... \\ C_{p1} & C_{p2} & ... & C_{pq} \end{bmatrix}$$

Each submatrix $C_{ij}$is evaluated as seen in equation 1 varying k from 1 to r.

The function is implemented according to this algorithm and has two nested loops to get each submatrix product. For each product requires a loop and a matrix multiplication and an addition inside the loop. In total there are six nested loops.

Since block matrix multiplication is only performed on submatrices it takes advantage of data locality. Considering the memory hierarchy and size, it is expected to be faster than the simple version. By choosing a good block size the memory hierarchy could be effectively exploited resulting in less cache misses and better performance.

The block size is expected to give a better performance when two input blocks fit in the cache at once. This is the case when matrices are larger than the cache size, for small sizes of matrices the block size does not affect the performance.

3.4 DGEMM_mm

The DGEMM function is a highly optimized matrix-matrix multiplication function written in FORTRAN. FORTRAN saves data in multidimensional arrays column-wise compared to the row-wise manner of C. Because we are writing in C our wrapper must take care of the conversion between the C-style and the FORTRAN-style matrices. A matrix constructed in C style is interpreted as the transposed in FORTRAN.

$$\begin{aligned} A_C &= A_F^T \\ B_C &= B_F^T \\ C_C &= C_F^T \end{aligned} \tag{2}$$

The FORTRAN function takes the three input arguments A, B and C. Transposing the output of the function to get the desired matrix implies:

$$\begin{aligned} C_F &= A_F \times B_F \\ C_F^T &= (A_F \times B_F)^T \\ C_F^T &= B_F^T \times A_F^T \end{aligned}$$

Using the equations 2 gives us the favorable result:

$$C_C = B_C \times A_C$$

Which means that we only need to swap the parameters of the FORTRAN function.

4 Experimental setup

We have used the Sun studio collect tool to generate our results. The collect tool allows us to get a summary of time spent in each function without the need for modifications to the program, enabling us to get fairly good performance measures. Additionally the tool allows us to query the hardware performance counters of the CPU to obtain supplemental performance data. We have used this facility to query the "fp_ops" (total floating point operations) and the data cache misses counters.

The matrix multiplication subroutines must run for a couple of seconds to allow *collect* to query the program and get the measurements. For this purpose

Table 1: Machine configuration for the performed experiments.

Vendor	OS	Arch	# of cores	L1 Cache	L2 Cache	Clock
AMD	Linux 2.6.32	x86_64	24	64kb	512kb	1.9 GHz
Intel	Linux 2.6.32	x84_64	8	32kb	256kb	2.67 GHz

we have created loops around the matrix calculation functions in our program that achieve a runtime of at least three seconds for each of the three algorithms. The number of times the algorithms runs may vary, but by normalizing the performance counters with the actual time spent we can generate comparable results. As a positive side effect this will give us an average for all data generated of our functions.

Our main program takes several parameters as input, allowing us to control the size of the matrices and the block size used by the block matrix-multiply algorithm. This allowed us to write a wrapper script that varies the size of the matrices created, making it easy to generate many measurements. *Collect* will generate several binary files containing the measurements, which can be accessed by *er_print*. Further manipulation by bash scripts, awk and gnuplot allows us to fully automatically generate performance graphs, which are presented in the following sections.

5 Machine description

We used two types of machines for the performance evaluation, AMD and Intel. Both machines are 64-bits processors of x86 architecture. The AMD machine has 2 AMD Opteron 6168 processors with a total of 24 cores reaching 0.8-1.9GHz. It has 64KB of L1 cache and 512KB of L2 on each core. It also has 12MB L3 cache on each processor. Intel machine has 2 Intel Xeon X5550 processors with a total of 8 cores reaching 1.6-2.67GHZ. It has 32KB of L1 cache and 256KB of L2 cache on each core. It also has a L3 cache of 8MB per processor. The both have separate instruction and data L1 caches, while cache is unified in L2 and L3. Both machines run Linux 2.6.32 as OS.

For the experiments on performance evaluation on variable matrix size we used the AMD machine. For variable block size we used both machines. The difference in the memory hierarchy sizes is considered to affect the performance of the function by changing the block size.

5.1 Compilers

We compile our program using the Sun C 5.10 Linux_i386(suncc) compiler using the following options *-fast, -xlic_lib=sunperf, -xrestrict, -xO5* flags. Additionally we use the *-xchip=opteron* flag on the Linux AMD machines to get the chip set optimizations. Fast is a macro that enables many other optimization options and the *-xO5* enables the highest optimization level to enable good performance. The *-xlic_lib=sunperf* specifies the linking against the Sun performance library

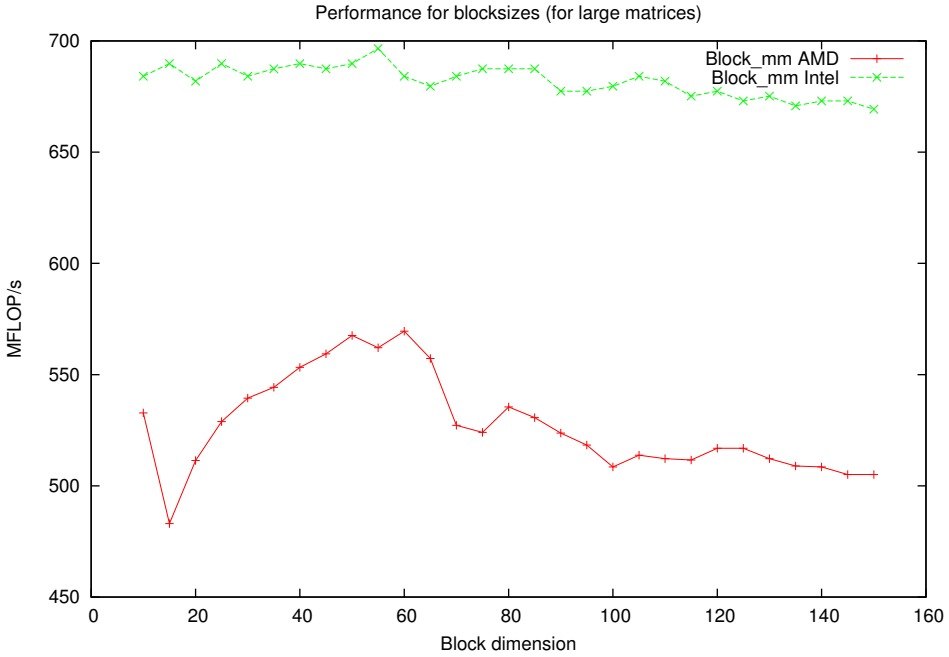

Figure 1: Varying the dimension of the block of the block_mm.

which is used for *DGEMM*. Finally the *-xrestrict* flags tells the compiler that we have no overlapping pointers in our code. Overlapping pointers prevent the compiler from doing loop unrolling.

When using the Sun studio analyzer, it is shown that the row and column loops in our functions are interchanged and the loops are unrolled up to eight times to increase performance. Additionally pre-fetching is enabled automatically in the compiler, allowing us to circumvent some of the cache misses.

6 Results

We now evaluate the three matrix-matrix multiplication functions we have implemented. To evaluate the performance of the block_mm we need to find a good block size for the cache. To find a good block size for the block_mm we vary the block size for a fixed size matrix-matrix multiplication. In figure 1 we have plotted the performance of the block_mm when varying the dimension of the block size. For the experiments we used a fixed matrix of size 1000×1000 we vary the block size from 10 to 150 in steps of 5.

In figure 1 we see that the AMD run has its maximum around 60. The block dimension of 60 allows for 2 block to be placed in the cache at once. This comes from the fact that AMD has 64 Kb of L1 cache and two blocks. Two blocks

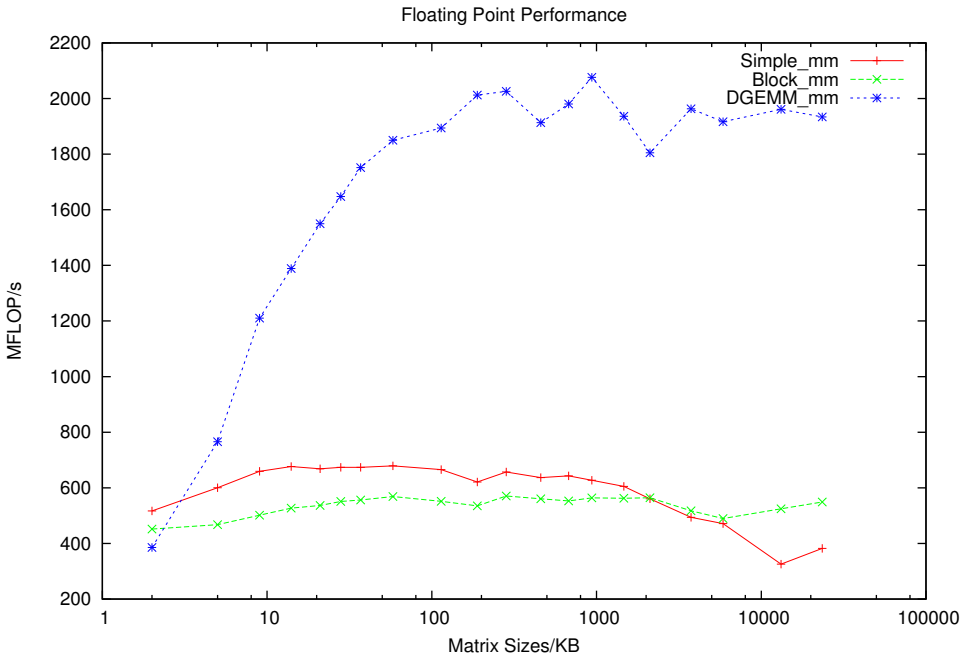

Figure 2: Floating point performance of the three matrix-matrix multiplication implementations.

requires:

$$60^2[\text{elements}] * 8[\text{bytes}] * 2[\text{blocks}] = 57.600[\text{Kb}]$$

When two blocks are placed in the L1 cache the program can do the multiplication of those two blocks right from the cache, writing the result back into memory reducing cache misses.

The AMD curve drops right after the block dimension reaches 65. This is because the two block no longer fits in the L1 cache. There is also a second drop right after the block dimension reaches 85, which is where one block takes up the entire L1 cache.

The Intel curve follows the same trend as the AMD curve, but with less variation. The reduced variation could be explained by a better cache prefetching of the Intel processor.

With the block size set to 60 we now evaluate the performance of all the three functions. The performance of the three functions can be seen in figure 2.

We can see that the library function DGEMM outperforms our homemade implementations by far. This is because the DGEMM function is highly optimized. The simple_mm performs slightly better than the block_mm for smaller matrix sizes. When the matrix sizes become larger the block_mm becomes more efficient than the simple_mm. The extra overhead in block_mm, in form of

more nested loops and a larger number of branches, could explain the performance difference between simple_mm and block_mm for small matrix sizes.

We can see that our own implementations will never be as efficient as the DGEMM library function.

The performance of simple_mm relates to the size of the different cache levels. The performance can be seen to drop at the cache level sizes. Even though the drop for the L1 cache is very insignificant.

7 Conclusions

We have shown the three different implementation of the matrix-matrix multiplication functions, and evaluated the performance of them. We have shown that our implementations of the function cannot compete with a highly optimized library function of an HPC library. However different implementations that take advantage of data locality and machine properties can affect performance. The blocking matrix-matrix multiplication algorithm is faster for greater matrix sizes, but it never reaches the performance of the DGEMM library function.