Assignment1 – CSE536, winter 2015

Gayathri Srinivasan  
G00278495  
gssrinivasan@oakland.edu

Dense Matrix Multiplication

# Dense matrix multiplication

* Matrix multiplication is one of the most widely used computation kernels in many scientific and engineering applications. In this assignment we have implemented various functions that perform multiplication of two dense matrices.

A[M][K] \* B[K][N] = C[M][N]

* The Assignment included implementation of 5 functions, Viz.,

matmul\_base – The baseline algorithm

matmul\_base\_1 – A variant to compute the C matrix column-wise.

matmul\_base\_sub – For the multiplication of two sub-matrices.

matmul\_row1D – Row-based 1D block distribution for computing C matrix.

matmul\_col1D – Column-based 1D block distribution for computing C matrix.

matmul\_rowcol2D – Row/Column-based 2D block distribution for computing C matrix.

# Implementation and Definitions

#### The Code Implementation of the dense matrix multiplication is as attached below



#### *Idea behind the implementation*

The “matmul\_base\_sub” function is designed to compute the value of an element in the C matrix, given the initial value of i and j. It is called from various instances of the code for computing the C[i][j] values.

The “*matmul\_row1D*” takes the *num\_tasks* as one of it’s parameters, estimates if the number of rows *N* is divisible by this value and accordingly assigns to each of it’s task “*N/num\_tasks”* number of rows to be computed. This is taken row-wise using different counter values like *initiali* and *initialj.* Hence, for every task it computes the value for *num\_of\_work\_in\_one\_task*  number of row-wise elements in the C matrix. If *N* is non-diviible by *num\_tasks*, then we use the *not\_in\_cycle* (N-num\_of\_matdata\_left), which gives the value of the already computed rows, ie., not-in-cycle data gives the starting point of the left over number of rows. So, using this we set the counter values.

The “*matmul\_col1D*” works similar to that of “*matmul\_row1D”* just that this function unlike the later computes the column-wise elements in the C matrix instead. This function also uses the same kind of counter values to set the values of *i* and *j* to be sent to the *matmul\_base\_sub* function. The logic of computation is the same the counters *initiali* and *initialj*  are set according to the required specifications.

The “*matmul\_rowcol2D*” is used for the block wise decomposition of C matrix before computing the necessary values. Initially we set the block depending on the *num\_tasks* value. We set the *num\_of\_rows\_in\_block* to be *N/sqrt(num\_tasks),* depending on this value we decompose the matrix C for the computation. If the *N%num\_of\_rows\_in\_block* is 0, then the decomposition becomes simple, in a case that the C matrix can be divided as smaller square sub-matrices each of size equal to *num\_of\_rows\_in\_block.* The *num\_blocks* I computed and a simple for loop is done for computing C matrix. In case if the *N*  is non-divisible by *num\_of\_rows\_in\_block*  we encounter a need for calculating a sub-matrix which is not a square matrix, and also a matrix which is a square matrix but of size smaller than the *num\_of\_rows\_in\_block.* So we use few variables to check the pointers to the C and update the value of counters, *initiali* and *initialj* to compute the values of C matrix.

# performance Chart and analysis summary

#### Quoted below are the performance analysis of the Code

The Machine used was a No-Machines for Windows at Beatles.secs.oakland.edu server. The results of the two categories of optimization are as follows.

* The performance graph for the –O0 optimization has an advantage of optimizing the execution time and the code size. It’s a default optimization level used in Linux.
* The execution time for the computation of the C matrix is most optimized in case of row/col2D as we can see a significant difference in the computation time as the data set increases.
* For the lower data set value say, 32X32 or 64X64 the performance of the 5 different methods are almost comparable to each other.
* The change is observed for higher data set value, where the performance of base, base\_1, row\_1D, column\_1D and row/col\_2D varies. The former 4 algorithms take a longer time to execute when compared to the later.
* This change is the computation time is primarily due to the shared memory access time, and also some factors like reading and writing to the non-consecutive memory locations (non-contiguous locations). Since, there is an existence of contiguous memory access in the rowcol2D algorithm it’s performance is considered high.
* When we compare row\_1D or col\_1D with base and base\_1 respectively, the existence of contiguous memory is available in both row\_1D and col\_1D algorithms, which makes its performance better.
* The performance graph for the –O3 Optimization level is as shown above. In Linux –O3 optimization level has got the advantage of optimizing the code size and the execution. This is the reason for the overall fall in the execution time in the graph compared to the –O0 level.
* The improvement parameter –O3 contributes to the reduction in the code size during compilation and the execution time, retaining the rest of the concept intact.
* Hence the observation of the behavior of various algorithms are same as the ones discussed above.
* To add a point to the explanation, when we increase the number of task the computation time of the C matrix is improved significantly. Applying the concurrent/parallel working will prove the algorithms to function with maximum efficiency.

# Reference

The values used for the performance graph creation are as follows

* For –O0 optimization the values are

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| row/col-2D | 0.00 | 3.00 | 25.00 | 122.00 | 1116.00 |
| column\_1D | 1.00 | 3.00 | 26.00 | 181.00 | 1127.00 |
| row\_1D | 0.00 | 5.00 | 29.99 | 262.00 | 1164.00 |
| base\_1 | 1.00 | 3.00 | 25.00 | 297.00 | 1400.00 |
| base | 0 | 3.00 | 26.00 | 301.99 | 1409.00 |
|  | 32x32 | 64x64 | 128x128 | 256x256 | 512x512 |

* For –O3 optimization the values are

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| row/col-2D | 0.00 | 0.00 | 5.00 | 74.00 | 331.00 |
| column\_1D | 0.00 | 0.00 | 7.00 | 78.00 | 356.00 |
| row\_1D | 0.00 | 1.00 | 8.00 | 82.00 | 351.00 |
| base\_1 | 1.00 | 0.00 | 7.00 | 78.00 | 359.00 |
| base | 0 | 1.00 | 6.00 | 77.00 | 345.00 |
|  | 32x32 | 64x64 | 128x128 | 256x256 | 512x512 |