Matrix Multiply Report

Jaime Luengo Rozas : jl3752

February 16, 2019

1 Introduction

In this report, three types of matrix multiplication strategies are discussed:

- Single threaded row by column multiplication,
- Single threaded tailed row by colum multiplication,
- Multithreaded tailed and cyclic multiplication.

together with my own implementation of them. The tests where all implemented in a machine part of the ugclinux cluster, from the CS department.

2 Overall comparison

In Fig.1 we can see a comparison of the three different methods versus the size of the matrix. Here we can see how the single threaded tiled type outruns the row by column by at least one order of magnitude. Then the multithreaded tiled type outruns the row by column by almost 2 orders of magnitude. As the dimension of the matrixes grow, the difference is more evident.

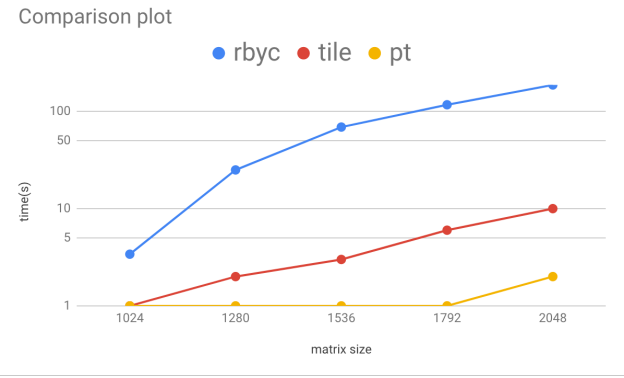

Figure 1: Comparison in time performance between three methods. Time is log scaled in the y-axis and in seconds.

3 Tiled multiplication

For the tiled single threaded type of multiplication, the size of the blocks is studied. For a matrix dimension of $2048x2048$, the block size is varied from 1 to 2048, multiplying by 2. The results in Fig.2 show that the best block sizes are in the range 16-256. Below 16 block size, most likely the caches are under utilized, not filling them completely, so switching from one block to another effectively works as in row by column multiplication that new words are brought in from DRAM and data in the cache is evicted. Higher than 256, the data held by L1, L2 and LLC caches is not enough for the block size so it is also efectively the same as row by column multiplication.

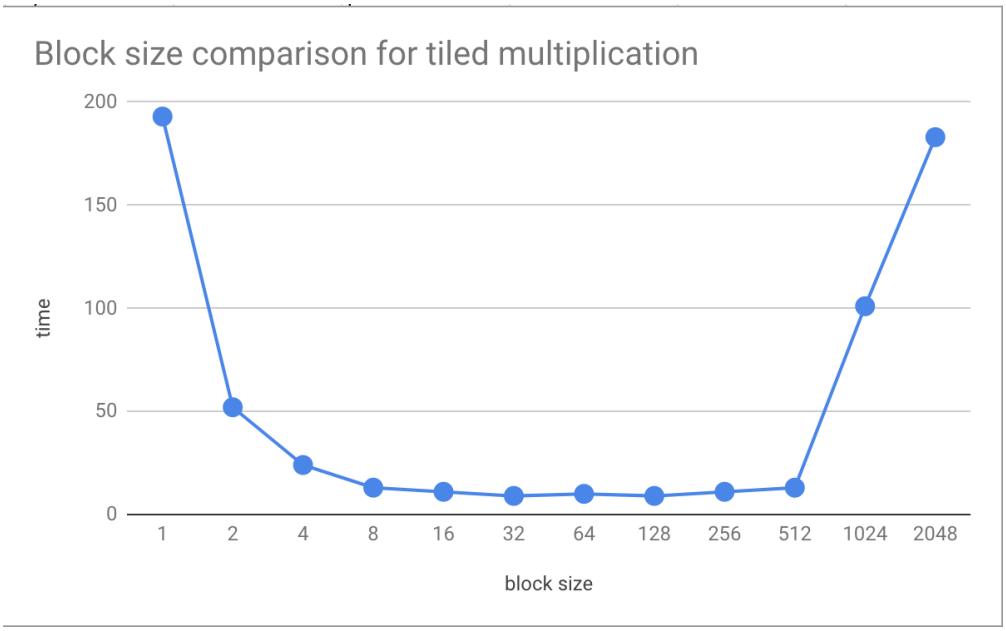

Figure 2: Block size comparison for tiled multiplication with squared matrixes of size 2048x2048

4 Multi-threaded multiplication

For the Multi-threaded multiplication, mutexes are not used to avoid its overhead. Instead each element of the resulting matrix is computed by a single thread.

Moreover, my first implementation involved a cyclic multiplication where for $C[i, j]$, one thread would compute $A[i, :]B[:, j]$; where A and B are the input matrix and C the output. However this implementation resulted being having similar performance to the tiled multiplication for my 4-core machine. This indicated that a single threaded cache-conscious implementation can compete with brute-force implementation of a multithreaded one. Hence, my final implementation is uses the tiled multiplication, where each block of C is computed by a single thread.

Since the input to *netid_mm_pt.c* si just the matrix size and number of threads; I introduce a function *getBlockSize(int n)* that tries to see if there is a block size in the optimal range

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 3: Index representation of blocks within a matrix

detailed in the previous section, that divides the matrix dimension. If there is, then the tiled version is executed, otherwise the cyclic one is executed.

The threads are assigned to blocks in a first come first served basis. If there are p blocks in a row, then there are p^2 blocks in total. This blocks are numbered as shown in Fig.3 So as we create threads $\{1 \cdots n\}$, thread i will get blocks $\{i + tk | k \in \mathbb{N}, i + tk < p^2\}$, where t is the number of threads. If the multiplication is cyclic then the assignment of threads is the same but to cells instead of blocks. Then, k can go up to $i + tk < n^2$ where n is one dimension of the matrix.

In Fig.4 a comparison between the number of threads for the multi-threaded program is presented. Here a $4096x4096$ matrix is used, so the tiled method is executed. The results show that after 4 threads, increasing them doesnt really affect the performance. This means that, since my machine is 4-core processor, the overhead of context switching between threads is neglectable vs the actual computation time.

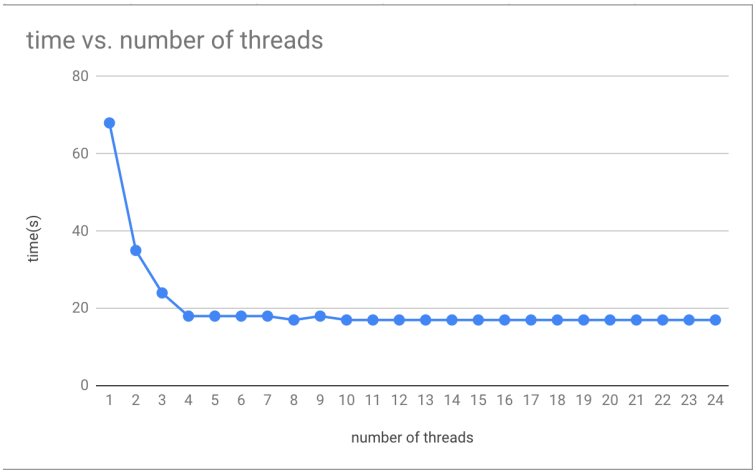

Figure 4: Comparison of number of threads for matrix size of 4096x4096