~~Matrix multiplication is key in scientific computing. Previously, the basic, block-optimized, algorithms were optimized using cache-aware techniques. Today, this paper investigate how parallezation can improve the performance of matrix multiplication algrithms.~~

Matrix multiplication is a fundamental operation in scientific computing, with cache-aware techniques previously optimizing basic and block-optimized algorithms for better performance. This paper explores the impact of parallelization using OpenMP to further accelerate these algorithms on modern multi-core processors. By distributing computations across multiple cores, the study aims to enhance both speed and scalability.

~~We use OpenMP API to parallize the code in shared-memory programming, use Likwid to hardware performance counter to record the runtime and memory utilization. These algorithm are tested in problem sizes[128,512,2048], the block optimization tested in two block sizes[4,16], basic and block MMUL are tested within threads numbers[1,4,16,64], and comparison to a standards reference MM implementation (CBLAS) with is one thread.~~

This paper uses OpenMP to parallelize matrix multiplication and LIKWID to monitor hardware performance, including runtime and memory utilization. The algorithms are tested with problem sizes of 128, 512, and 2048, and block optimization is evaluated with block sizes of 4 and 16. Basic and block-optimized algorithms are run with 1, 4, 16, and 64 threads, and compared to a reference CBLAS implementation using one thread to assess performance improvements.

~~This research use OpenMP to parallelize the basic and block MMUL algorithm, use LIKWID to record each thread’s performance in including Runtime (RDTSC) the maximum multi threads, and , number of L2 Accesses, L3\_CACHE\_REQ and RETIRED\_INSTRUCTIONS in one-threads, and compared with the blas algorhtm.~~

This research employs OpenMP to parallelize the basic and block matrix multiplication algorithms, utilizing LIKWID to record performance metrics such as runtime (RDTSC) for each multithreaded execution, as well as L2 accesses, L3 cache requests, and retired instructions for single-thread execution. The performance of these parallelized algorithms is then compared to that of the BLAS algorithm to evaluate their effectiveness.

The thread 4 has between 3 to 4 speed up compared with thread1 among all the problem size. Thread 16 has between 12 to 14 speedup compared with thread1 among all the problems size. For the thread 64 , the speed up increase from 12 to 53 with the increase problem size.

The speedup of thread 4 for both block4 and block 16 are from 3 to 4 among all the sizes. The speedup of thread 16 for both block4 and block 16 increase slightly from 9 – 14 with the probolem size increases. The performance between block 4 and block 16 are not largely different. The performance of thread 64 increase significantly below 10 to over 50 with the problem size increases.

L2 cache access show the the percentage of L1 cache utilization. The larger of number L2 cache access means much more L1 cache miss, demonstrade lower L1 cache utilization. In this table, we can see basic is 48 or 148 larger than cblas, which is worst L1 cache utilization. Block4 is 4 – 8 times more than cblas. The block is best l1 cache tulization from 0.87 – 1.7.

L3 cache request indicated the l2 cache utilization. The higher l3 cache request means the higher l2 cache miss, means lower efficiency.

In table 2, for small problem isze 128, both basic and block matrix multiplication shows low l3 cache request .but with bigger problem isze, the basic the l3 cache request is amost 300 – 570 times than cblas. Block 4 have around 20 times l3 cache request. Block 16 has the better performance wich is 3-4 times more than cbals.

For smaller size, all the data can be fit in the L1 cache and L2 cache, so there is no much different among these algorithms. But when the problem size grows, , This behavior can be attributed to better cache blocking in Block16, which more effectively exploits data locality

For smaller problem sizes, all the data can fit into the L1 and L2 caches, leading to minimal differences in performance across the algorithms. However, as the problem size increases, Block16 shows better performance due to more effective cache blocking, which improves data locality and reduces cache misses, particularly in the L2 cache, allowing it to maintain higher efficiency compared to the other methods.

In table 3, the basic and block 16 has around 10 times more retired instruction then the cblas, while the block 4 has around 25 times more retired instruction. Which shows the block 4 is less efficient and require more computational steps, likely due to their less optimized memory access patterns and higher overhead for parallel execution management.

These paper presents multithread programming indeed increase the efficiency of basic and block MMUL alrotihm. Lots of threads with small problems size sometimes will decrease the performance due to overhead, but with problem size grows, the advantages is very clear. For serial one thread functions, the block 16 performance better than block 4 with lower L2 cache access and lower L3 cache request and lower retired instruction, utilizing cache better. The basic performance worse.