Università della Svizzera italiana

Institute of Computing
CI

High-Performance Computing Lab **Institute of Computing**

Student: Lorenzo Varese Discussed with: Kaan Sen

Solution for Project 1

In this project we will practice memory access optimization on a local machine.

1. Explaining Memory Hierarchies *(25 Points)*

1.1. Memory specification

In Table 1 there is an overview of the memory specifications of the Macbook Pro featuring the M2 chip.

Memory Size	Specification
Main Memory	16 GB
L3 Cache	Dismissed in M2 architecture
L2 Cache (Performance core shared)	32 MB
L1 Cache (Performance core)	128 KB
L2 Cache (Efficiency core shared)	4 MB
L1 Cache (Efficiency core)	64 KB

Table 1: Memory specifications

Acquiring official data regarding the memory specifications of the M2 Pro chip can be challenging. To retrieve this information, I have employed various resources and methods, including:

- System Profiler Command: the "system_profiler SPHardwareDataType" command provides a comprehensive overview of general system information.
- Sysctl Command: the "sysctl -a — grep hw.cache" command was utilized to obtain partial information pertaining to the cache hierarchy.
- Third-Party Apps: tools such as "Device Information - CPU-Z" from the App Store were consulted to glean additional details about the processor.
- Web Articles and Pages: various online articles and web pages were explored in the quest for information about the memory hierarchy of this new processor.

It should be noted that despite these efforts, there exists a notable absence of official resources that offer a comprehensive and explicit clarification of the memory hierarchy.

1.2. Benchmark on the local machine

The analysis of memory access latency across different cache sizes and access patterns is employed with code that conducts both read and write operations on an array while utilizing CPU cycle counters for time measurement. The resulting findings offer valuable insights into the dynamics of the memory hierarchy and cache performance.

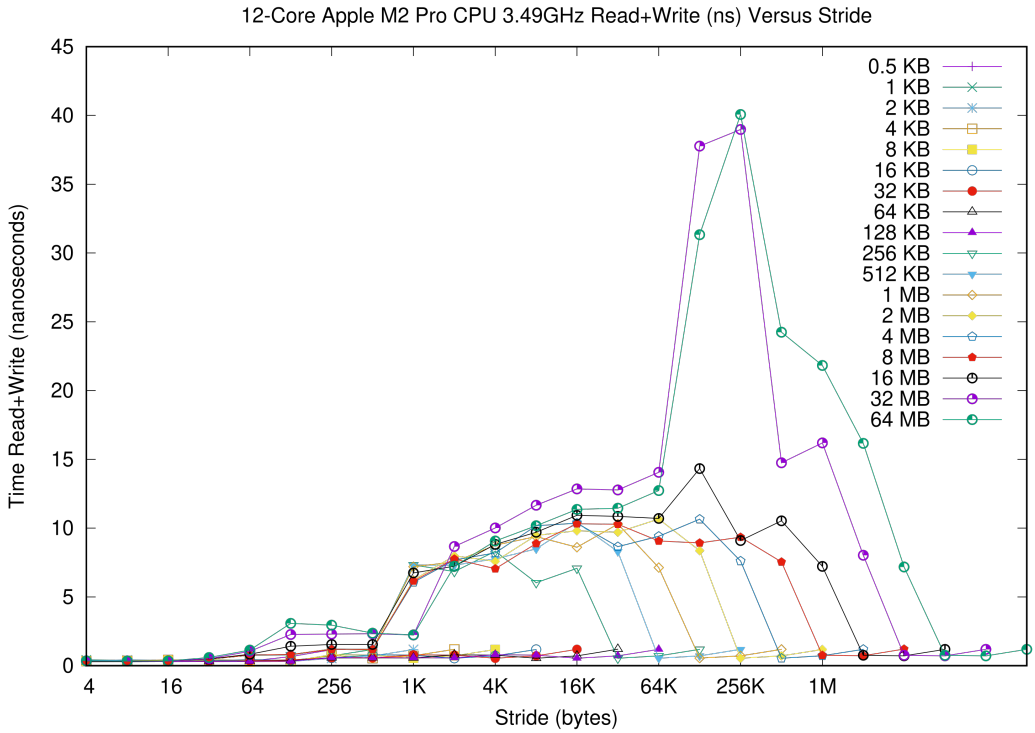

Figure 1: Memory access latency for various array sizes

1.3. Comparison between different stride and memory size

Executing data operations incurs varying memory costs, depending on cache size and the approach to data storage and retrieval within the cache memory. Specifically, we will explore two distinct scenarios:

- For the configuration where csize = 128 and stride = 1:
  - A cache size (csize) set at 128 represents a relatively modest allocation of cache memory, capable of accommodating the entire array of data.
  - A stride of 1 signifies that memory elements are accessed in a sequential manner.
  - This memory access pattern adheres to a regular and sequential sequence, facilitating an abundance of data being consistently available in the cache. Consequently, this leads to a diminished occurrence of cache missing, thereby enhancing performance.
- In the scenario where csize = 220 and stride = csize/2 = 110:
  - A cache size (csize) of 220 is larger when compared with the 128 case.
  - The employment of a stride value equal to csize/2 (110) implies that there exists a gap in the access pattern between elements stored in the cache.
  - This particular memory access pattern deviates from the previous sequential pattern. In instances where the cache capacity falls short of accommodating the complete array of data, this difference leads to a cache missing, resulting in a decrement in performance.

The data has been collected for two specific configurations in membench, which closely align with the two previously discussed scenarios. (Noting that the 4x reported stride and size values is a result of the integer size being 4 bytes in the C programming language). The outputs for these cases are as follows:

- Output Case 1
  - Configuration: csize = "128" and stride = "1"
  - Size: 512
  - Stride: 4
  - Read+write time: 0.391 ns
  - Execution time (seconds): 1.000
  - Cycles: 986,511,266
  - Steps: 2,000,148
- Output Case 2
  - Configuration: csize = "220" and stride = csize/2
  - Size: 4,194,304
  - Stride: 2,097,152
  - Read+write time: 1.207 ns
  - Execution time (seconds): 1.000
  - Cycles: 1,001,073,955
  - Steps: 79

The access times exhibit a significant change, transitioning from 0.391 ns in Case 1 to 1.207 ns in Case 2.

1.4. Consideration on temporal locality

A comprehensive analysis can be conducted by using the 'generic.ps' file generated by the 'membench' tool on the local system. The resultant data is also graphically represented in Figure 1.

In the chart's legend, it is evident that multiple lines correspond to varying sizes of arrays utilized in the benchmarking process. Arrays with sizes equal to or less than 128 KB, denoted by the purple line with a triangular marker, exhibit relatively low execution times in nanoseconds for both read and write operations.

However, as the array size is increased, it becomes increasingly challenging to retain the entirety of the array within the cache memory. This situation is aggravated when the stride, denoting the step size for memory accesses, is also augmented. Consequently, the frequency of cache missing rises due to the necessity of accessing data that has not been preloaded into the cache memory, resulting in deteriorating performance.

Hence, optimal temporal locality, which concurrently maximizes the amount of data storable within cache, is attained when utilizing an array size of 128 KB.

2. Optimize Square Matrix-Matrix Multiplication *(60 Points)*

2.1. Naive matrix multiplication

Various algorithms superior to the conventional method for matrix multiplication can be found in the literature. Nonetheless, our focus remains on optimizing the naive algorithm for enhanced memory efficiency. To this end, we outline the classical algorithm using the following pseudocode:

```
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
        end for
    end for
end for
```

In the following sections, we will endeavor to enhance its performance from a cache-friendly standpoint.

2.2. Tiled implementation of matrix multiplication

The application of tiled matrix multiplication presents a viable technique for achieving a cache-friendly solution to the aforementioned problem. In practical terms, this approach involves dividing large matrices into manageable blocks and performing computations on these blocks. The primary objective is to reducing memory access to slower storage and enhance cache locality. However it is important to note that:

- The effective size of these blocks, which can be efficiently accommodated within the cache, depends on the magnitude of data contained within the matrices.
- To implement this strategy, we must enhance the naive algorithm to ensure precise indexing within both the matrix and the block.

The forthcoming two paragraphs will introduce the two primary focal points that have been mentioned.

2.2.1. Ideal stride calculation

In an ideal scenario, the cache should possess the capacity to accommodate all the data associated with the three matrices. Practical limitations prevent this from being the case. Therefore, it becomes necessary to calculate the stride parameter (s) to determine the maximum value of s that can be employed without necessitating the storage of blocks outside the cache memory. Mathematically, this condition can be expressed as follows: for three square blocks, each having a side length of s, the combined space they occupy must not exceed the cache's maximum capacity (M). Formally, this condition is represented as $3s^2 \leq M$. Consequently, it follows that $s \leq \sqrt{M/3}$

The L1 cache is specified to have a capacity of 128 KB. Given that the size of a single double value is 8 bytes, the cache's theoretical storage capacity can be calculated as $M = 128KB/(8bytes)$, which equals $16{*}10^3$ double values. To establish an appropriate stride (s) for cache optimization, the inequality $s \leq \sqrt{M/3}$ is applied, resulting in a maximum permissible stride value of $s_{max} = 73.02$. For the practical implementation, the stride value is rounded down to $s_{max} = 73$.

2.2.2. Algorithm implementation

The provided pseudocode for the blocked matrix multiplication offers a foundational concept essential for the actual implementation. However, it diverges from the actual implementation in one key aspect. In the actual C code, the loading of matrices A and B into fast memory is performed implicitly by accessing the appropriate data using the correct combination of indexes. In contrast, the loading and storing procedure for matrix C is explicitly defined like in the pseudocode.

```
for i=1 to n/s
    for j=1 to n/s
        Load C_{i,j} into fast memory
        for k=1 to n/s
            Load A_{i,k} into fast memory
            Load B_{k,j} into fast memory
            NaiveMM (A_{i,k}, B_{k,j}, C_{i,j}) (fast memory)
        end for
        Store C_{i,j} into slow memory
    end for
end for
```

Here is my C implementation, which has been condensed for readability in this report, with basic operations like clearing and copying of C[] removed. It's important to note that the A, B, and C matrices are organized in a column-major format and represented as 2D linearized array.

```
void square_dgemm(int n, double *A, double *B, double *C)
{
  for (int i = 0; i < n; i += block_size){
    int i_next_block = MIN(i + block_size, n);
    for (int j = 0; j < n; j += block_size){
      int j_next_block = MIN(j + block_size, n);

      //Clear C_temp (Note: C_temp is block_size * block_size)
      //...

      for (int k = 0; k < n; k += block_size){
        int k_next_block = MIN(k + block_size, n);

        // Naive matrix multiplication (using only fast memory)
        for (int i_block = i, c_row_index = 0;
             i_block < i_next_block;
             i_block++, c_row_index++)
        {
          for (int j_block = j, c_col_index = 0;
               j_block < j_next_block;
               j_block++, c_col_index++)
          {
            double temp = 0.0;
            for (int k_block = k; k_block < k_next_block; k_block++)
            {
              temp += A[k_block * n + i_block]
                        * B[k_block + j_block * n];
            }
            C_temp[c_row_index + c_col_index * block_size] += temp;
          }
        }
      }

      // Store C_temp in C
      // ...
    }
  }
}
```

It is evident from the code that the "Naive matrix multiplication" shares the same underlying structure as the previous approach. The primary challenge lies in precisely defining the indices to access the appropriate data and ensuring the correct movement within a block during the iterations of the loops.

2.2.3. Comparison of benchmarks

The provided script, "run_matrixmult.sh," conducts multiple benchmarking iterations on matrices of varying sizes. It calculates the average performance value in GFlops/s for each matrix size within three distinct scenarios: utilizing naive matrix multiplication, employing the Intel OneMKL library for high-performance computing, and applying the blocked matrix multiplication algorithm that we have implemented.

On a computer with an M2 Pro processor (Performance core): the processor runs at a clock speed of 3.49 GHz. It can perform 4 calculations simultaneously due to its vector width. Each calculation involves 1 floating-point operation (FMA). As a result, the processor has a peak performance

of 13.96 billion floating-point operations per second (GF/s). This peak performance metric is established as a constant named "MAX_SPEED" with a value of 13.96 GF/s, and it is defined into the benchmark.c file.

The chart in the Figure 2 is the outcome of benchmarking conducted on my local machine after compiling with the -O3 optimization flag in case of a stride value of 24 (further comments on stride value in Section 2.2.4).

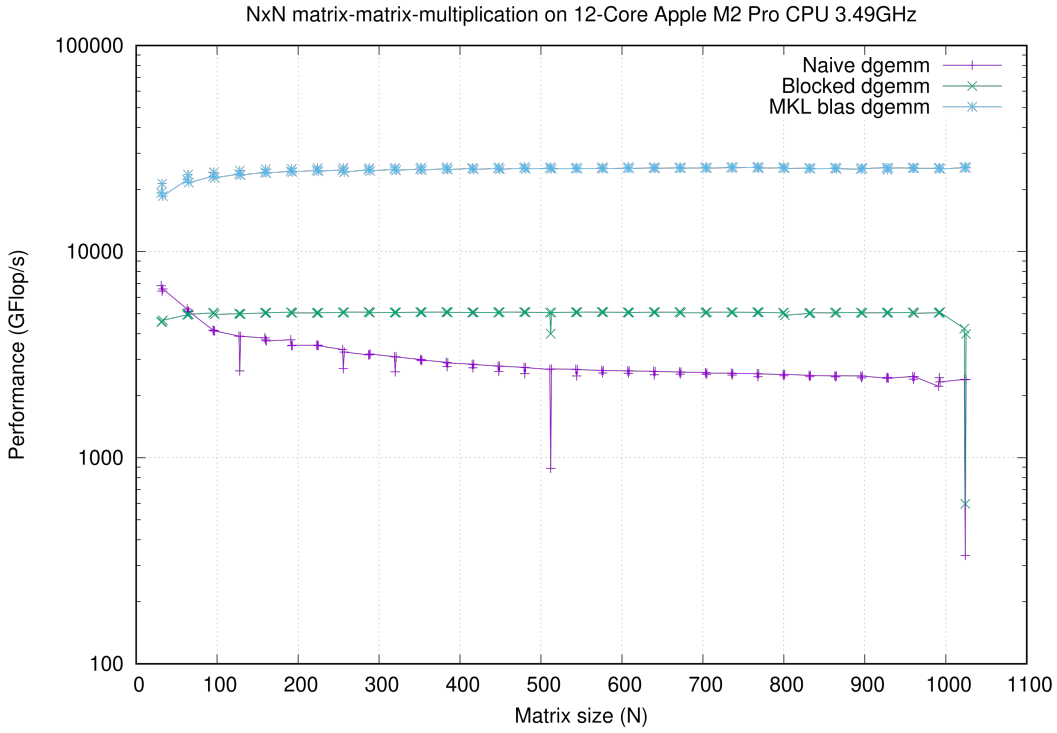

Figure 2: Comparison of matrix multiplication benchmarks

2.2.4. Further consideration on stride values and compiler

During the process of solving the assignment, I observed a relatively modest improvement when using the calculated stride value of 73. In an attempt to optimize performance further, I experimented with various stride values and documented the results in Table 2. This exploration ultimately led me to conclude that, particularly in the case of tests conducted with the -O3 optimization flag and a limited set of matrix sizes, the stride value of 24 yielded the most significant average percentage performance improvement.

It's essential to note that the benchmarked performance to surpass, which represents the baseline established by the naive matrix multiplication, stands at $23,5358$ GFlop/s.

Upon reflection, it is worth noting that further analysis of this result may be warranted. One potential explanation, which was discussed during practical sessions in class, revolves around the compiler's role in executing the benchmark using the OneMKL library. The compiler is in charge of converting our code, which was originally intended for ARM architecture, into a format that can run smoothly on Intel-based systems. To accomplish this, we utilize the -target x86_64-apple-darwin flag during compilation, which ensures that the code is specially adapted for compatibility with Intel architecture. This process could introduce unexpected behaviors that could impact the utilization of cache memory in a manner that differs from our initial expectations.

Stride	Avg. Perf. Peak (GFlop/s)
1	5,4127
2	18,1462
4	25,0201
8	32,6096
12	31,9692
16	34,485
20	35,4924
24	36,0452
28	35,511
32	35,3201
32	35,3201
53	31,2365
73	26,6031

Table 2: Average performance of peak for different stride values