Università della Svizzera italiana | Institute of Computing CI

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

Student: Fabian Goebt Discussed with: Ekkehard Steinmacher, Henrique Gil

Solution for Project 1

HPC Lab — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

In this project you will practice memory access optimization, performance-oriented programming, and OpenMP parallelizaton on the ICS Cluster .

1. Explaining Memory Hierarchies *(25 Points)*

To proceed with the tasks in demand for this project, I have used the laptop provided by the faculty: a MacBook 16 pro, running the Ventura 13.5.2 OS on an Apple 16 Pro chip.

Since *likwid* is not available for macOS distributions, the values for the chip specifications had to be researched over the internet. The memory specification regarding the cache sizes within the chip are as follows:

Table 1: Memory Specifications of MacBook 16 Pro [1]

Main memory	16 GB
Cache	
L1 (per core)	192 KB
L2 (shared)	36 MB
L3 (shared)	23 MB

Once this information was retrieved, I proceeded to follow the exercise script, executing the commands:

```
$ cd membench
$ source /opt/intel/oneapi/setvars.sh
$ make
$ source run_membench.sh
```

The previous commands generated the *generic.ps* file, which was then converted into a PDF using the command

```
$ pstopdf generic.ps
```

From here, we can see the following graph containing a plot of stride sizes in an exponential scale, $x - axis$, against the average time per cycle in nanoseconds, $y - axis$.

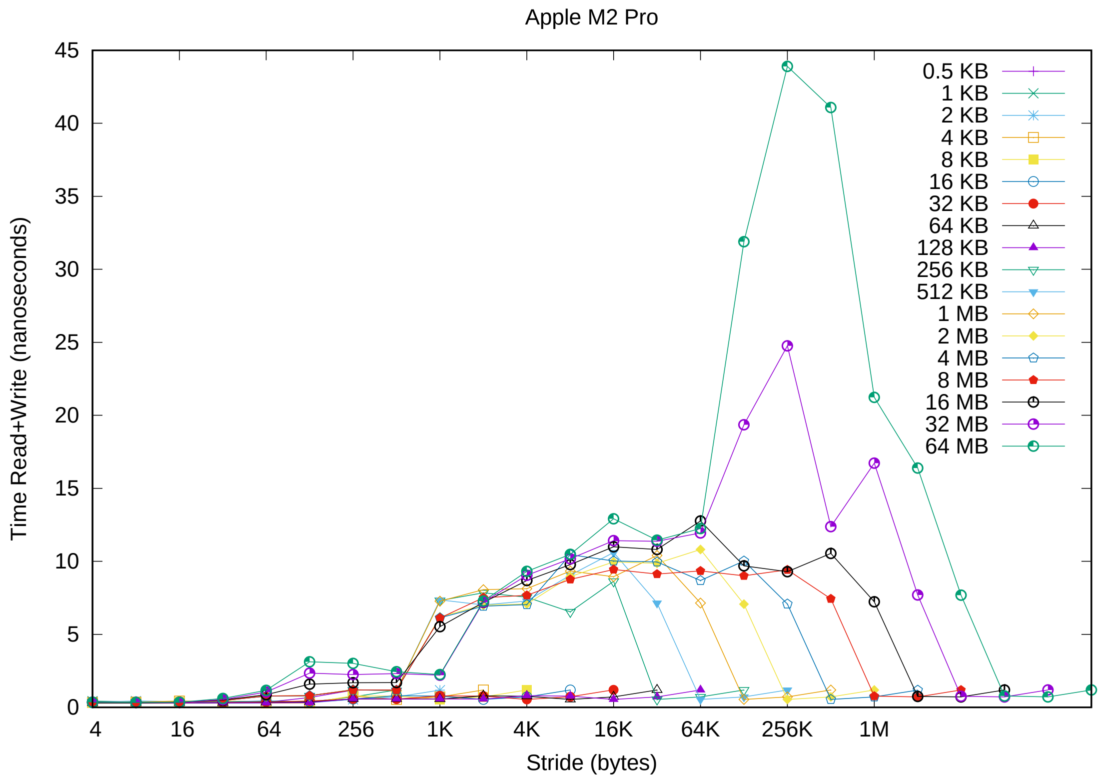

Figure 1: Memory access performance across strides and array sizes

From this graph, taking into account the previously researched cache sizes, Table 1, and the fact that other running processes in the laptop also make use of the CPU and its cache, we may derive the following conclusions:

- For every read-write access, if the memory(s) slot(s) is not yet in the cache, a whole memory block containing the desired slot is moved from the main memory to the cache. This is a common procedure verified in computer architectures, where the probability of accessing contiguous slots of memory is high, and therefore more efficient to move whole blocks instead of specific selected slots.

- According to the provided code on membench.c, the number of iterations through arrays with a given stride is dependant on the number of cycles, rather then on the array itself.

- Given the first two points, we can now understand that even for big array sizes, since strides are so low, iterations will mostly happen through slots of memory that are loaded once into the cache. In other words, a block of memory is loaded into cache for a read-write process and by design (low stride size) the next slots to be accessed will also be on that block of memory. This process, from a memory access performance perspective, may also be understood as **temporal locality**.

- From 64 Bytes to 512 Bytes stride sizes we can already see a slight rise in time. According to the values of Table 1, we can justify the rise by pointing out accesses to lower levels of the cache (L2 and L3), and for bigger array sizes also some acesses to main memory.

- From 1 KiloBytes to 64 KiloBytes stride sizes We can see a a substantial increase in time for bigger array sizes. This happens because strides get bigger, which means the iterations are happening less throughout the blocks of memory loaded into cache and more so on blocks that have to be loaded from the main memory. For lower array sizes, since these fit on its whole in the cache, read-write times remain overall the same.

- From 64 KiloBytes stride sizes on wards we can see at first a great increase in read-write times for bigger sized arrays. The same principle referred on the previous point is applied but now to a bigger degree, strides iterate more over blocks of memory that need to be loaded into cache. For smaller sized arrays, the performance is maintained.

- Finally, for strides closer to $cize/2$, where $csize$ is the array size, very few blocks of memory have to be loaded into cache to access a single memory slot. That is, for a stride of $csize/2$ only 2 blocks would have to be loaded. This means that by holding these two blocks in cache and only iterating between them, most of the computations happen within memory slots that are cached, and therefore read-write times tend to decrease.

In conclusion, we can see that really small arrays always offer the best temporal locality despite the stride size because the whole array can be loaded into cache and accessed in a low read-write time frame. Up to 16 MB of array size, we may see some increase in read-write times because some slots of memory have to be transferred form the main memory to the cache. For higher than 16 MB array sizes, there is a great increase in read-write times for mid sized (in accordance to the $x - axis$) strides because more blocks of memory have to be interchanged between the main memory and cache. For all array sizes, strides close to half the array size imply iterations through memory blocks mostly present in the cache, therefore very low read-write times may be observed.

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

Taking into consideration the unavailability of the ICS cluster, this project was fully developed using the MacBook 16 Pro 16, as mentioned in Section 1. Given this, for the second part of this assignment it is necessary to calculate the theoretical peak regarding this laptop, so I could then plug that value onto the MAX_SPEED constant located in benchmark.c.

The Apple M2 Pro chip with 12 cores has a clock speed of 3.5GHz [1], but no information could be found regarding bus size to CPU and number of instructions processsed per cycle, and extra CPU modules like the FMA from the ICS cluster. For this reason I also researched possible benchmarks in terms of FLOPS/s for this CPU, coming up with the value ≈ 74 GFLOPS/s [2].

After setting up these values I proceeded to write the code for the blocking technique in the benchmark-blocked.c file, end up with the following algorithm

```
const int block_size = 32;

for (int kk = 0; kk < n; kk += block_size) {
    for (int jj = 0; jj < n; jj += block_size) {

        // upper bounds for blocks; clamps if over n
        int k_max = kk + block_size > n ? n : kk + block_size;
        int j_max = jj + block_size > n ? n : jj + block_size;

        for (int i = 0; i < n; i++) { // iterate through each row of A

            // iterate through rows of current block of A
            for (int k = kk; k < k_max; k++) {

                // prefetch value to reduce memory access
                double r = A[i + k * n];

                // iterate through columns of current block of B
                for (int j = jj; j < j_max; j++)
                    C[i + j * n] += r * B[k + j * n];
            }
        }
    }
}
```

Note that I have chosen to write the $block\_size = 32$**. This value will be later explained.**

After setting up the code, I ran the commands

```
$ cd matmul
$ source /opt/intel/oneapi/setvars.sh
$ make
$ source run_matrixmult.sh
```

Having as console output:

```
==== benchmark-naive ======================
#Description:   Naive, three-loop dgemm.

Size: 31        Mflop/s:  6894.74       Percentage:  9.32
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
    SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math
    Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (
    Intel(R) AVX) instructions.
Size: 32        Mflop/s:  6459.72       Percentage:  8.73
Size: 96        Mflop/s:  4200.64       Percentage:  5.68
Size: 97        Mflop/s:  4158.41       Percentage:  5.62
Size: 127       Mflop/s:  3918.77       Percentage:  5.30
Size: 128       Mflop/s:  2418.73       Percentage:  3.27
Size: 129       Mflop/s:  3909.54       Percentage:  5.28
Size: 191       Mflop/s:  3739.88       Percentage:  5.05
Size: 192       Mflop/s:  3703.31       Percentage:  5.00
Size: 229       Mflop/s:  3557.36       Percentage:  4.81
Size: 255       Mflop/s:   3400.5       Percentage:  4.60
Size: 256       Mflop/s:  2756.05       Percentage:  3.72
Size: 257       Mflop/s:   3374.9       Percentage:  4.56
```

```
Size: 319        Mflop/s:  3110.98       Percentage:  4.20
Size: 320        Mflop/s:  2670.42       Percentage:  3.61
Size: 321        Mflop/s:  3094.48       Percentage:  4.18
Size: 417        Mflop/s:  2861.13       Percentage:  3.87
Size: 479        Mflop/s:  2736.52       Percentage:  3.70
Size: 480        Mflop/s:  2549.17       Percentage:  3.44
Size: 511        Mflop/s:  2686.43       Percentage:  3.63
Size: 512        Mflop/s:  854.977       Percentage:  1.16
Size: 639        Mflop/s:   2626.6       Percentage:  3.55
Size: 640        Mflop/s:  2542.73       Percentage:  3.44
Size: 767        Mflop/s:  2551.86       Percentage:  3.45
Size: 768        Mflop/s:  2482.29       Percentage:  3.35
Size: 769        Mflop/s:  2549.13       Percentage:  3.44
#Average percentage of Peak = 4.45994

==== benchmark-blas ======================
#Description:   Reference dgemm.

Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
    SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math
    Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (
    Intel(R) AVX) instructions.
Size: 31         Mflop/s:  19352.9       Percentage: 26.15
Size: 32         Mflop/s:  21508.4       Percentage: 29.07
Size: 96         Mflop/s:  24132.2       Percentage: 32.61
Size: 97         Mflop/s:    22811       Percentage: 30.83
Size: 127        Mflop/s:  23863.4       Percentage: 32.25
Size: 128        Mflop/s:  24789.7       Percentage: 33.50
Size: 129        Mflop/s:  23599.7       Percentage: 31.89
Size: 191        Mflop/s:  24635.6       Percentage: 33.29
Size: 192        Mflop/s:  25341.6       Percentage: 34.25
Size: 229        Mflop/s:  24515.6       Percentage: 33.13
Size: 255        Mflop/s:  24929.3       Percentage: 33.69
Size: 256        Mflop/s:  25533.1       Percentage: 34.50
Size: 257        Mflop/s:  24376.7       Percentage: 32.94
Size: 319        Mflop/s:  25160.2       Percentage: 34.00
Size: 320        Mflop/s:  25579.2       Percentage: 34.57
Size: 321        Mflop/s:  24836.3       Percentage: 33.56
Size: 417        Mflop/s:  25147.8       Percentage: 33.98
Size: 479        Mflop/s:  25482.2       Percentage: 34.44
Size: 480        Mflop/s:  25832.8       Percentage: 34.91
Size: 511        Mflop/s:    25505       Percentage: 34.47
Size: 512        Mflop/s:  25812.8       Percentage: 34.88
Size: 639        Mflop/s:    25535       Percentage: 34.51
Size: 640        Mflop/s:  25828.7       Percentage: 34.90
Size: 767        Mflop/s:  25700.3       Percentage: 34.73
Size: 768        Mflop/s:  25912.1       Percentage: 35.02
Size: 769        Mflop/s:  25556.2       Percentage: 34.54
#Average percentage of Peak = 33.3305

==== benchmark-blocked ===================
#Description:   Naive, three-loop dgemm.

Size: 31         Mflop/s:  5648.08       Percentage:  7.63
Intel MKL WARNING: Support of Intel(R) Streaming SIMD Extensions 4.2 (Intel(R)
    SSE4.2) enabled only processors has been deprecated. Intel oneAPI Math
    Kernel Library 2025.0 will require Intel(R) Advanced Vector Extensions (
    Intel(R) AVX) instructions.
Size: 32         Mflop/s:  3145.74       Percentage:  4.25
Size: 96         Mflop/s:  3046.48       Percentage:  4.12
```

```
Size:  97          Mflop/s:   5372.52          Percentage:   7.26
Size:  127         Mflop/s:   5495.01          Percentage:   7.43
Size:  128         Mflop/s:   3068.24          Percentage:   4.15
Size:  129         Mflop/s:    5442.3          Percentage:   7.35
Size:  191         Mflop/s:   5507.42          Percentage:   7.44
Size:  192         Mflop/s:    3104.7          Percentage:   4.20
Size:  229         Mflop/s:   5494.53          Percentage:   7.43
Size:  255         Mflop/s:   5660.77          Percentage:   7.65
Size:  256         Mflop/s:   2843.82          Percentage:   3.84
Size:  257         Mflop/s:   5406.86          Percentage:   7.31
Size:  319         Mflop/s:   5508.27          Percentage:   7.44
Size:  320         Mflop/s:   3113.83          Percentage:   4.21
Size:  321         Mflop/s:   5578.06          Percentage:   7.54
Size:  417         Mflop/s:   5592.94          Percentage:   7.56
Size:  479         Mflop/s:   5577.48          Percentage:   7.54
Size:  480         Mflop/s:   3118.56          Percentage:   4.21
Size:  511         Mflop/s:   5403.34          Percentage:   7.30
Size:  512         Mflop/s:   1050.49          Percentage:   1.42
Size:  639         Mflop/s:    5600.7          Percentage:   7.57
Size:  640         Mflop/s:   3114.38          Percentage:   4.21
Size:  767         Mflop/s:   5596.26          Percentage:   7.56
Size:  768         Mflop/s:   3018.42          Percentage:   4.08
Size:  769         Mflop/s:   5478.73          Percentage:   7.40
#Average percentage of Peak = 6.08045
```

Having as a comparison graph:

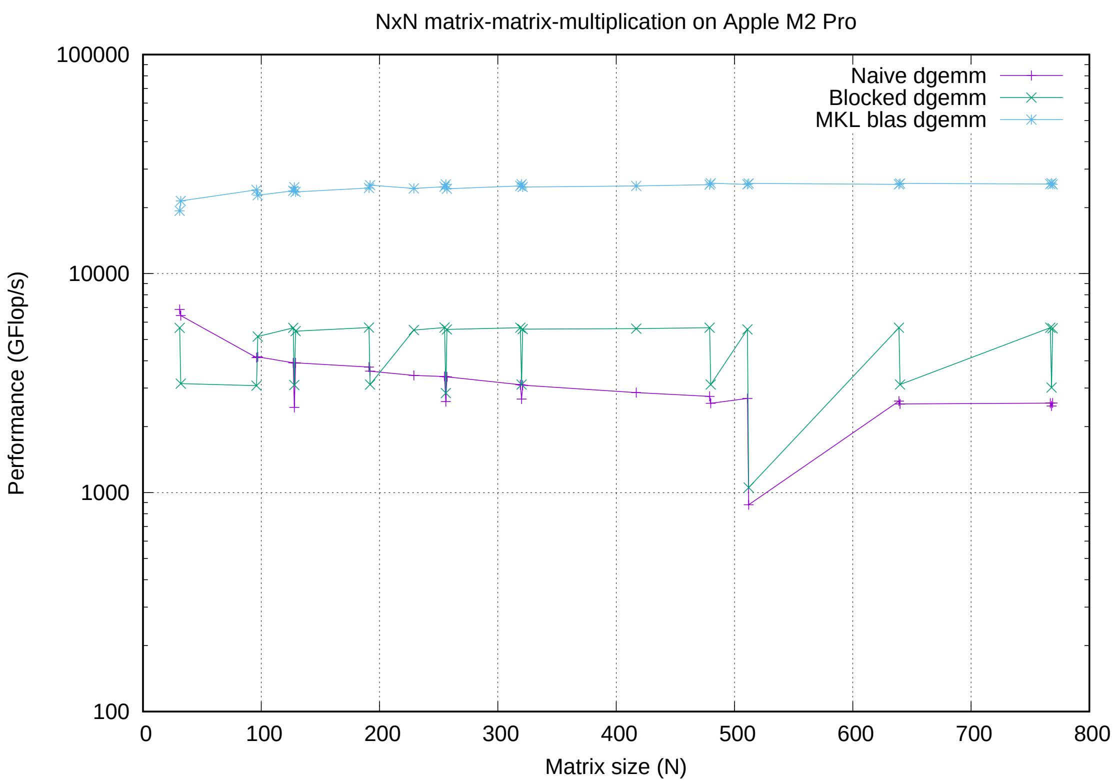

Figure 2: Memory access performance across strides and array sizes

A range of values for the block size variable were used, being that the size of 32 presented the best graphed values and average percentage of peak.

By analysing the graph we may infer that for N sizes below 100 the conventional straight forward use of matrix multiplication performs better than the blocked strategy. I suspect that this anomaly may be intrinsic to the device in use, such that an overall better performance was seen in another device running the same algorithm.

Due to the machine state, in particular the cache values, it is not possible do directly infer a relationship between the block size and the cache size. We know that for 32 sized blocks, for 3 arrays of doubles we need at least $8 * 32 * 32 * 3 = 24,576\,Bytes$, that is $24\,KB$, at any given time, which constitutes 12.5% of the total L1 cache per core.

By blocking we were able to get an average increase of 2% GFLOPS/s in comparison with the naive implementation. This fact can be translated into more cache hits (and there fore less misses) regarding the blocks loaded into memory. Still, this value is nowhere close to the performance of the BLAS algorithms, leaving space to a lot of improvement and further understanding of the underlying mechanisms inherent to memory access and matrix multiplication.

References

[1] Notebookcheck. *Apple M2 Pro Processor Benchmarks and Specs.* Accessed: September 29, 2023. 2023. URL: `https://www.notebookcheck.net/Apple-M2-Pro-Processor-Benchmarks-and-Specs.682450.0.html`.

[2] PassMark Software. *Apple M2 Pro Processor Benchmarks.* Accessed: September 29, 2023. 2023. URL: `https://www.cpubenchmark.net/cpu.php?cpu=Apple+M2+Pro+12+Core+3480+MHz&id=5189`.