Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **2021**

Student: Naga Venkata Sai Jitin Jami Discussed with: Ashutosh Singh, Valentin Bacher, Iyuele Alemu Korsaye

Solution for Project 2

Due date: 17.10.2021 (midnight)

HPC 2021 — 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)*

1.1. Parameters of memory hierarchy

Main Memory	65.58 GB
L3 Cache	25 MB
L2 Cache	256 KB
L1 Cache	32 KB

1.2. Memory access pattern

1.2.1. $csize = 128$ and $stride = 1$

In the program *membench.c*, the array is of type *int*. So in the array, each element occupies 4 bytes of memory space. So for an array of $csize = 128$, the memory allocation is as follows:

$$memory = csize \times 4 \tag{1}$$

$$\implies memory = 128 \times 4 = 512bytes \tag{2}$$

$$\implies memory = 0.5kB \tag{3}$$

We know from the memory hierarchy that L1 cache has a memory of 32kb. Since the memory required to process the concerned array is lower than L1 cache, the compiler will read all elements of the array into fast memory from slow memory for processing. So irrespective of *stride* value, the whole array get read into L1 memory in each memory access. The average time to access the array for all values of *stride* will be more or less the same.

1.2.2. $csize = 2^{20}$ and $stride = \frac{csize}{2}$

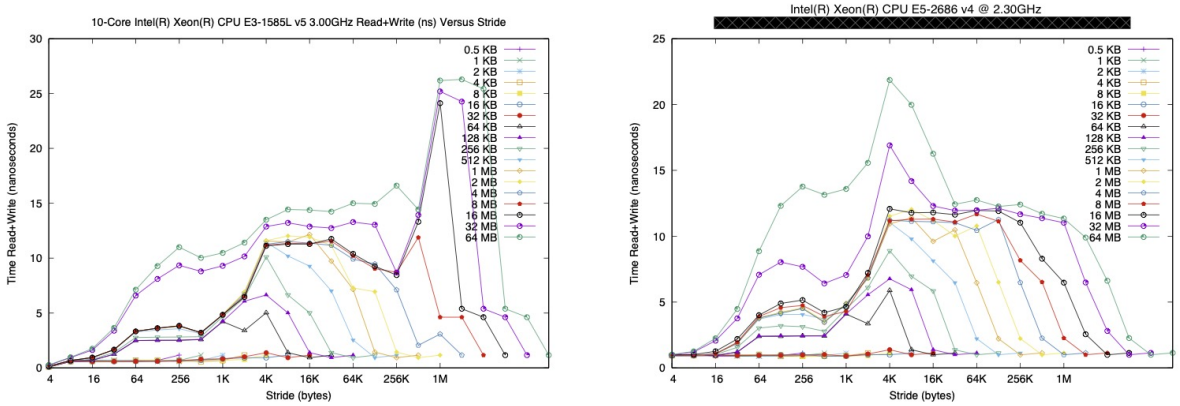
$$memory = 2^{20} \times 4 = 4MB \tag{4}$$

In this case, the memory occupied by the array is more than our fast memory. So hence, the stride value is key.

$$stride = \frac{csize}{2} = 2MB \tag{5}$$

So by declaring $stride = 2MB$, we're passing the first 2MB into fast memory and then the next 2MB. But 2MB is still big for our L1 and L2 memory. So the entire array doesn't get read into this fast memory. Instead, the array is first read into L3 cache (which can read 2MB) from slow memory. Subsequently into L2 followed by L1 cache. That is the memory access pattern.

1.2.3. Temporal Locality



(a) ICS CPU

(b) AWS CPU

Figure 1: Read + Write time vs Stride

Here we can see the *generic.ps* plot generated from the ICS cluster. As noticed in the previous section, all values of *csize* that occupy $memory \leq 32kb$ have fast $Read + Write$ times irrespective of *stride* value. This is because the whole array is being read into fast memory. So every time a new query is made, the concerned element is found in fast memory itself because of spatial locality. The whole array is in the fast memory for the whole loop.

For arrays that occupy $memory > 32kb$ there is a clear effect of stride value. Read-write times increase with increasing stride but starts dropping as we reach a stride of the order of the array. For small values of stride upto $32kB$, the number of read and write operations are not high since the stride value is small. So from 1 request to the next the effect of spatial locality comes into play. The whole cache line is already in the L1 fast memory and memory reads don't have to be as often. However the average read time keeps increasing as stride increases because as the value of stride

increases because the advantage of spatial locality is reducing.

However, as the stride value goes to the order of array size the number of read-write operations are significantly lower. The $stride = \frac{csize}{2}$ case has the best temporal locality for high array sizes because part of the array that is read into fast memory can be reused.

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

Performing matrix multiplication is a fundamental mathematical problem that comes into mind when you talk about almost every kind of regression, interpolation or optimization problem. So naturally, optimising the basic naive algorithm to perorm better on our memory architectures is key to High Performance Computing.
The naive algorithm goes like this:

```
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
    end
end
```

The above code is pretty straightforward, but the problem is that for the compiler, this is how the code looks:

```
for i = 1 to n
    for j = 1 to n
    Load C[i,j] into fast memory
        for k = 1 to n
            Load A[i,k] into fast memory
            Load B[k,j] into fast memory
            C[i,j] = C[i,j] + A[i,k] * B[k,j]
        end
    Store C[i,j] to slow memory
    end
end
```

These memory read-write operations take a lot more time from the processor perspective because fetching data is a time costly affair. The idea is to use our knowledge of memory hierarchy, memory access patterns to make the same calculation faster.

2.1. Transpose

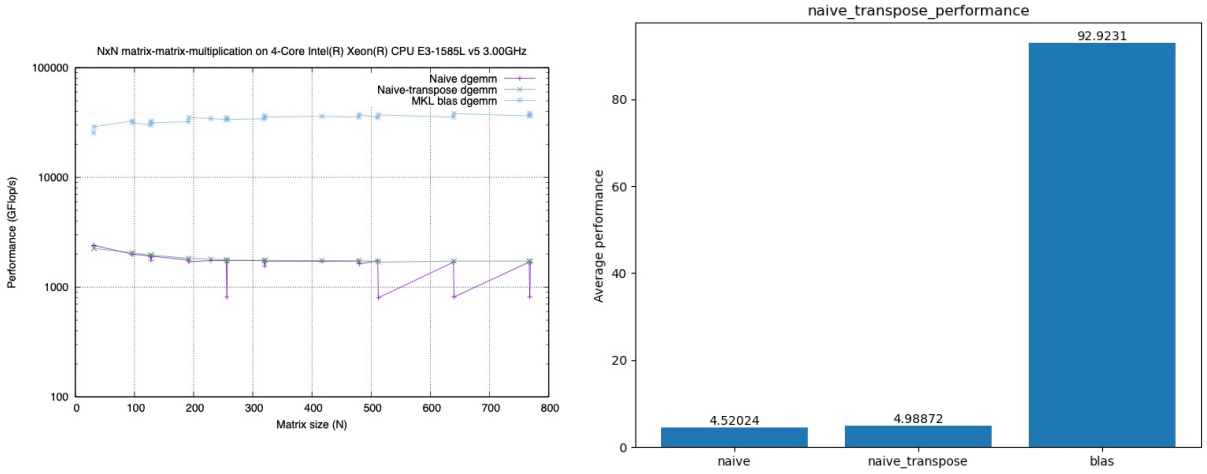
The matrices A, B and C are stored in column major scheme. So in the matrix multiplication when we are getting the dot product of $rowsofA$ with $columnsofB$ we are skipping n elements along the column of A to get to the next element in the row. But to make use of the spatial locality properties ideally we should try to access elements that come one after another in the column. So to take advantage of spatial locality we transpose matrix A and perform the naive algorithm we posted before. it goes like this:

```
A_t = Transpose(A)
for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            C[i,j] = C[i,j] + A_t[k,i] * B[k,j]
```

```
        end
    end
end
```

2.1.1. Performance

Comparing the performance between *naive* implementation, $naive - transpose$ implementation and vendor provided *blas* implementation for varied number of elements in the matrices gives us the following plot: Here you can see that the transpose method gives a certain consistency in



(a) Performance

(b) Average Performance

Figure 2: Naive Transpose

performance when it comes to matrices of large sizes. This naturally leads to an increase in average performance.

2.2. Blocking

Block multiplication of matrices is to divide up the matrices into blocks and perform matrix multiplication on each of these blocks to get the overall product in the end. The way we can extract performance through this is by making sure the block size of all 3 matrices is determined by the size of fast memory.
If A_{block}, B_{block} and C_{block} are small enough to fit on L1 cache, then using the benefits of fast memory access, we can reduce the overall $read+write$ operations and thus making it faster. Calculation for BLOCKSIZE:

$$CacheSize = 32 \times 1024bytes \tag{6}$$

$$SizeOfDouble = 8bytes \tag{7}$$

$$DoublesOnCache = \frac{CacheSize}{SizeOfDouble} \tag{8}$$

$$BLOCKSIZE = \sqrt{\frac{DoublesOnCache}{3}} \approx 36 \tag{9}$$

We use the above value to divide up the matrix into square matrices of $dimension = 36$. The remaining rows and columns in the end will undergo a regular dot product scheme. The math behind block multiplication can be found here:

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix} \tag{10}$$

$$B = \begin{bmatrix} B_1 & B_2 \\ B_3 & B_4 \end{bmatrix} \tag{11}$$

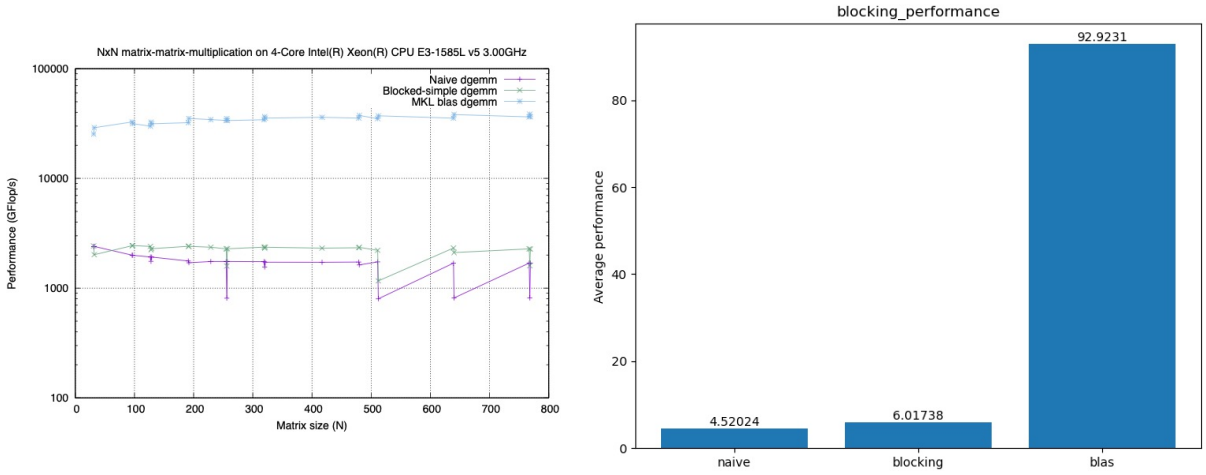
$$C = \begin{bmatrix} C_1 & C_2 \\ C_3 & C_4 \end{bmatrix} = \begin{bmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{bmatrix} \tag{12}$$

Code snippet:

```
blocksize = s;
for i = 0; i<n; i+=s
    for j = 0; j<n; j+=s
        for k = 0; k<n; k+=s
            dim_A = min(blocksize, n-i)
            dim_B = min(blocksize, n-j)
            dim_C = min(blocksize, n-j)
            naive_multiply(n, dim_A, dim_B, dim_C, A, B, C)
        end
    end
end
```

2.2.1. Performance

There is an obvious increase in performance due to blocking. The advantages of using spatial and temporal locality in the fast memory is clearly shown in the performance metrics.



(a) Performance

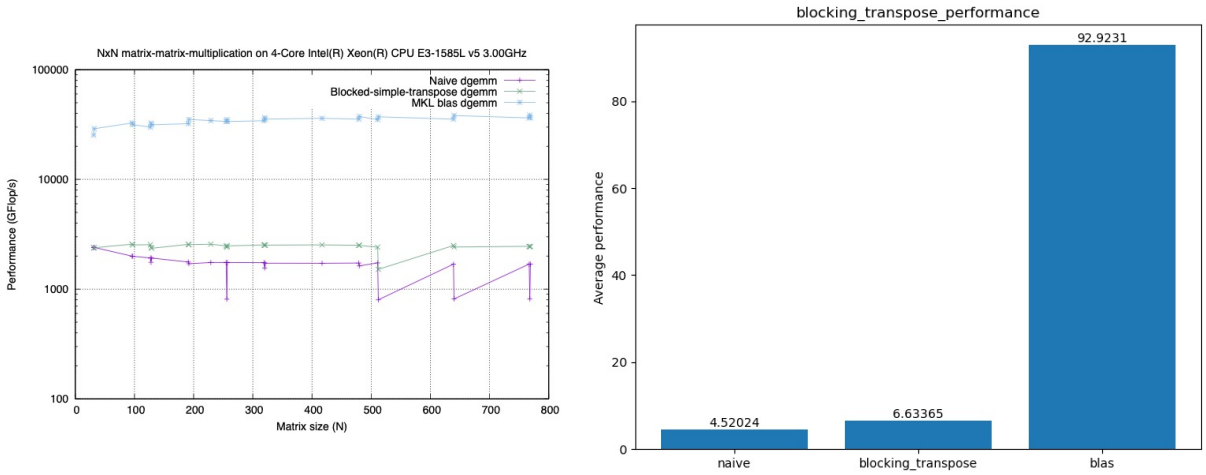
(b) Average Performance

Figure 3: Blocking

Upon implementing the transpose method in the block multiplication we see a similar effect of consistent performance for larger matrices which leads to a better average performance.

2.3. Loop unrolling

Loop unrolling takes advantage of how fast memory reads data from slow memory in the form of cache lines. The concept goes that when fast memory requests some data and gets a hit (*acachehit*) it copies data that are found in addresses around the requested data too. This happens under the assumption that if we are requesting some data from an address there is a good chance we will request data found in the corresponding addresses.
So we will take advantage of this by implementing loop unrolling up to a total of 3 variables inside blocking we carried out in the previous step. So in 1 iteration of 1 block, we are carrying out A_{row} $\cdot B_{column}$ for 3 rows and 3 columns simultaneously in one inner loop as opposed to doing 1 loop (1 read-write operation) for every scalar multiplication operation.

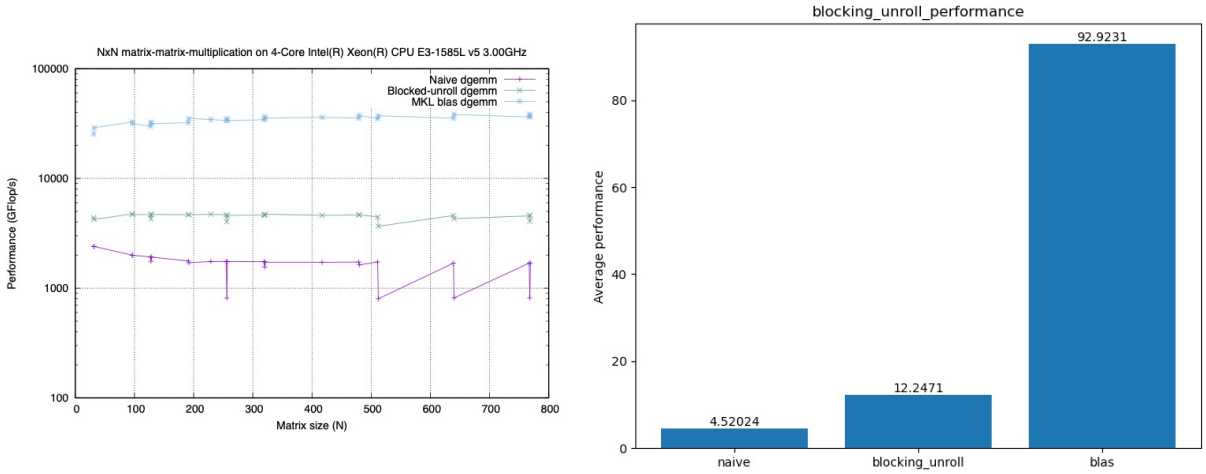


(a) Performance (b) Average Performance

Figure 4: Blocking+Loop Unrolling

2.3.1. Performance

Implementing loop unrolling within the blocked implementation gives a huge performance boost and a level of consistency for large matrices too.



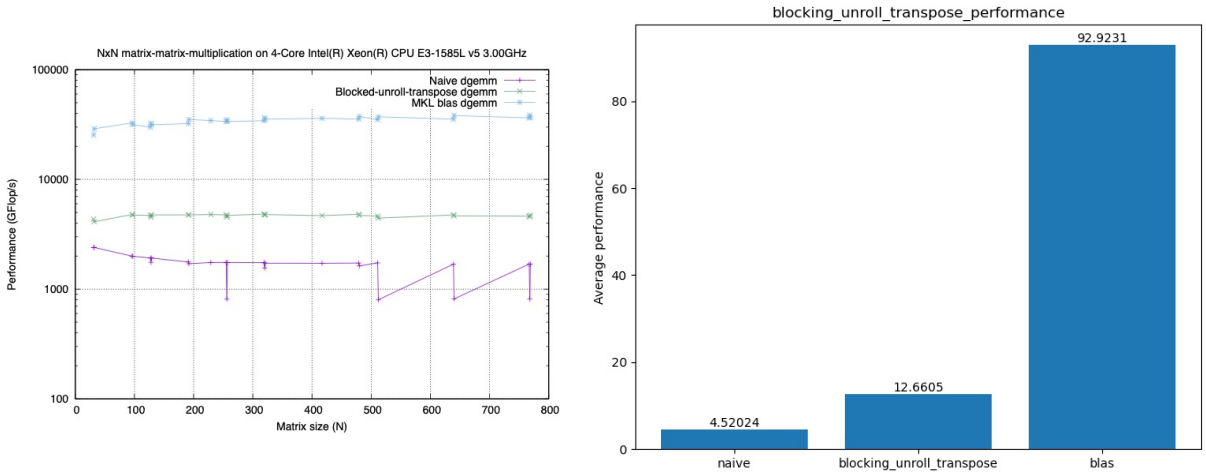
(a) Performance (b) Average Performance

Figure 5: Blocking

There is a slight performance increase when using the transpose method on the $Blocked_Loop_Unrolling$ framework but a small drop in performance was also observed on occasion. But similar observation on how performance was consistent when transposed in the case of larger matrices.

3. Final thoughts

In a pure mathematical perspective, the naive method is the most obvious way to calculate a product of 2 matrices. But from a high performance computing perspective there are ways to improve computing performance since the problem we are tackling is reducing read-write operations. There are many methods to improve apart from the ones we discussed like optimizing our code for all levels of cache and not just L1. Paralleling for multiple CPUs is also a possibility if we have a multi-core CPU.
Amongst the methods explored, blocking with loop unrolling (with or without transpose) gives the best performance.



(a) Performance

(b) Average Performance

Figure 6: Blocking+Loop Unrolling+Transpose

References

[1] R.A.v.d. Geijn, Huang, J., " How to Optimize Gemm"

[2] K. Goto and R.A.v.d. Geijn, " Anatomy of High-Performance Matrix Multiplication", ACM Trans. Math. Software, vol. 34, no. 3, pp. 1-25, 2008.