Performance Optimization in Matrix Multiplication and Numerical Integration

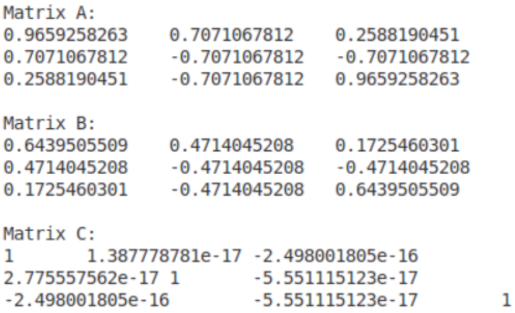
High Performance Computing

Anna Grebennikova, Anna Remizova, Kseniia Ovchinnikova

UGA

April 8, 2025

Lab Sheet 1. Matrix Initialization



```
Matrix A:
0.9659258263    0.7071067812    0.2588190451
0.7071067812    -0.7071067812   -0.7071067812
0.2588190451    -0.7071067812   0.9659258263

Matrix B:
0.6439505509    0.4714045208    0.1725460301
0.4714045208    -0.4714045208   -0.4714045208
0.1725460301    -0.4714045208   0.6439505509

Matrix C:
1       1.387778781e-17 -2.498001805e-16
2.775557562e-17 1       -5.551115123e-17
-2.498001805e-16        -5.551115123e-17        1
```

Figure: Validation of matrix multiplication

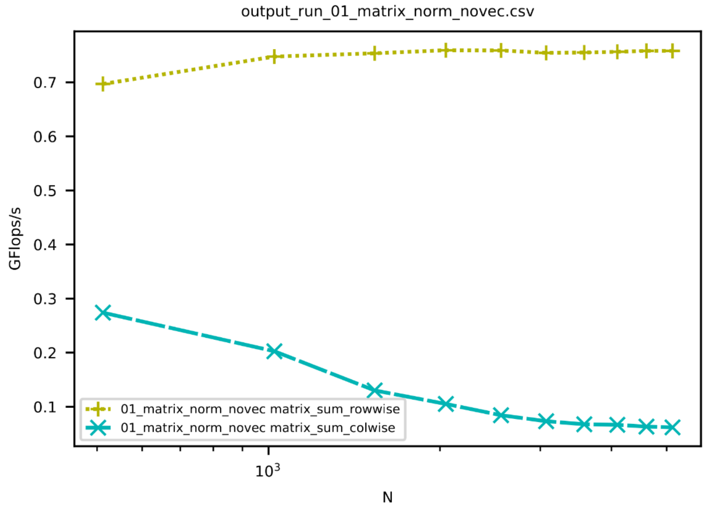
- The matrices A and B are filled according to the rule:

$$A_{i,j} = \cos\left(\frac{(j+\frac{1}{2})(i+\frac{1}{2})\pi}{N}\right)$$

$$B_{i,j} = \cos\left(\frac{(i+\frac{1}{2})(j+\frac{1}{2})\pi}{N}\right) \cdot \frac{2}{N}$$

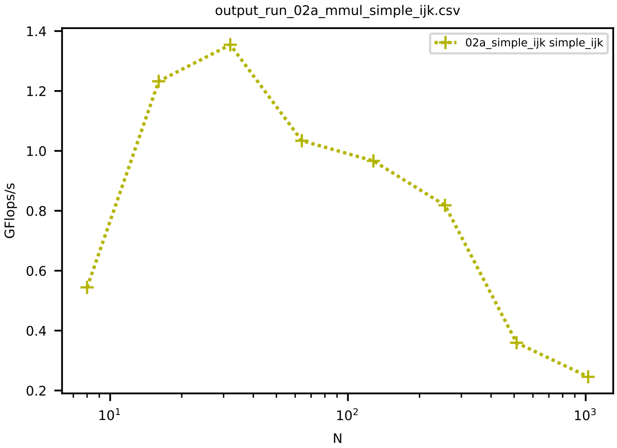
- Confirmed that result of multiplication of A and B results in the identity matrix C

Lab Sheet 1. Row wise vs column wise summation



- C/C++ uses row-major ordering
- Consequently, row-wise summation is observed to perform better than column-wise summation.

Lab Sheet 1. Matrix multiplication using a "*ijk*" algorithm

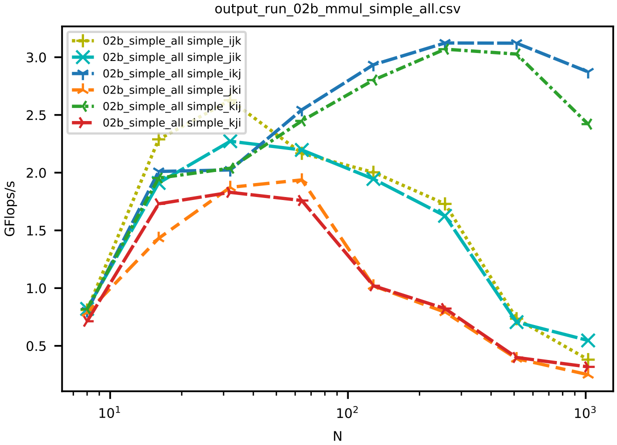


- We have both row wise and column wise operations:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

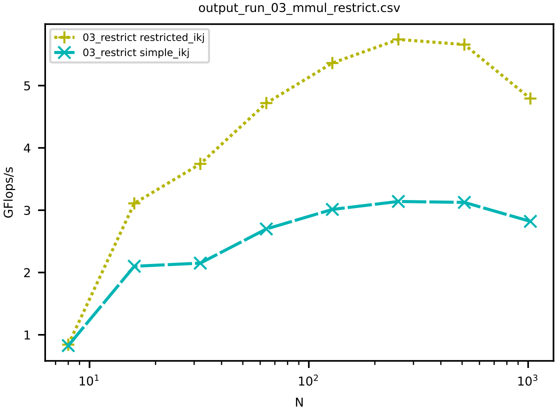
- Performance improves initially (from $N = 8$ to 32) due to better CPU cache utilization.
- Performance drops for large matrices due to cache limitations (8 MB).

Lab Sheet 1. Matrix multiplication with different loop orders



- In *ijk* order: accesses $A[i][k]$ (row-major) and $B[k][j]$ (column-major) for each k.
- In *ikj* order: constant for i, k and row-major for j.
- In *kij* order: column-major for i and row-major for j.

Lab Sheet 1. "*ikj*" algorithm vs restricted "*ikj*"



- Restricted *ikj* improves performance by informing the compiler that pointers are not aliases.
- This prevents reloading on each iteration.

Lab Sheet 2. Variable vs Constant block size

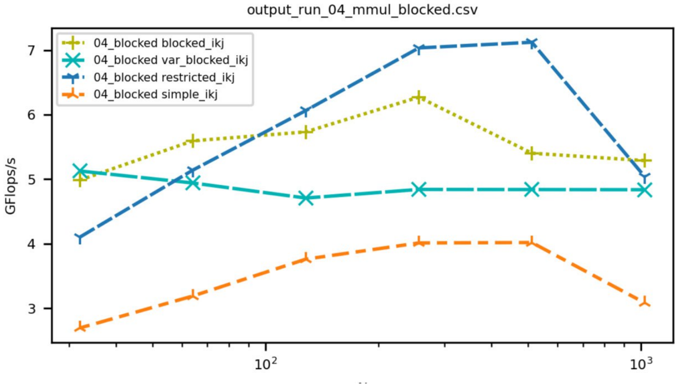


Figure: Variable block size = 16

- Variable block size performs slightly worse than compile-time constant size.
- Runtime variable block sizes hinder compiler optimizations, leading to extra computations.
- Restricted version remains better.

Lab Sheet 2. Experiment with larger matrices

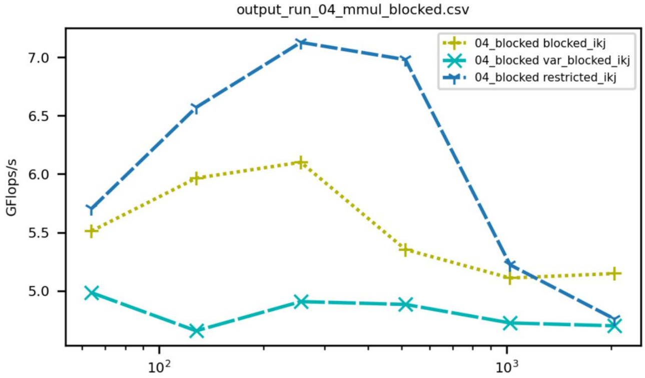


Figure: Matrix size = 2048

- From matrix size 2048, the simple blocked method outperforms the restricted method.

Lab Sheet 2. SIMD experiments

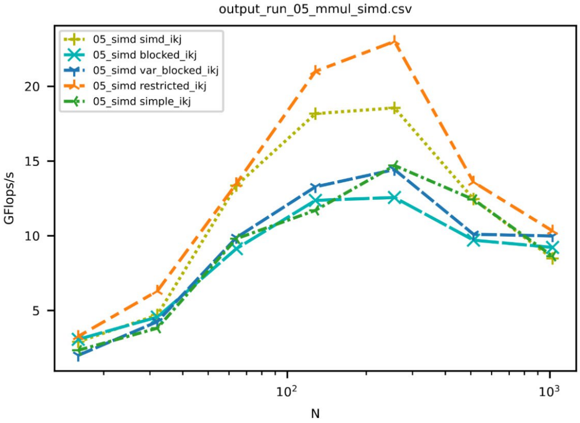
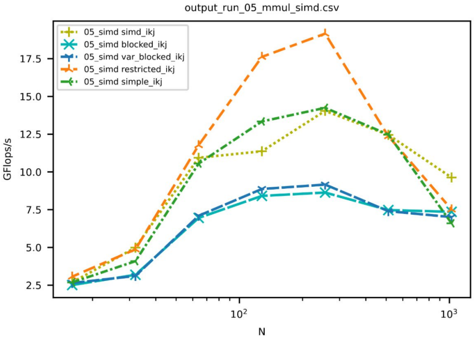


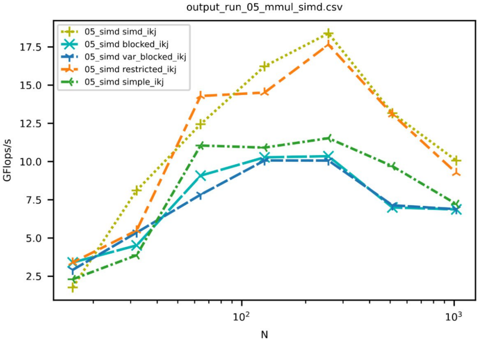
Figure: std::size_t alignment = 512

- Memory allocated using `posix_memalign((void*)A, alignment, size)` with `alignment = 512` and `size = N * N * double`.
- In the loop: `#pragma omp simd aligned(i_A, i_B, o_C : 512)`.
- The restricted method still performs better.

Lab Sheet 2. SIMD experiments



- PRAGMAs were defined in the inner loops
- Simple multiplication is now "leveled" with the SIMD

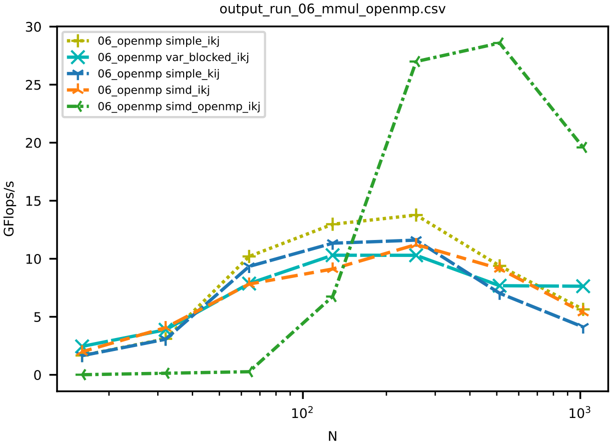


- No alignment memory allocation
- SIMD and restricted are now "leveled", as expected

Lab Sheet 3. Parallelization

We implement parallelization with:

```
    #pragma omp parallel for collapse(2)
```



Parallelization performance improves significantly, but slightly declines for 1024 by 1024 matrices due to cache overloading.

Lab Sheet 3. Numerical Integration

Numerical integration of π was performed by evaluating the following definite integral using Riemann sums:

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi \approx 3.141592654$$

Another integral considered was the *dilogarithm function*, defined as:

$$\mathrm{Li}_2(x) = -\int_0^x \frac{\ln(1-t)}{t}\,dt$$

At the point $x = 1$, this evaluates to:

$$\mathrm{Li}_2(1) = \frac{\pi^2}{6} \approx 1.64493406685$$

To ensure numerical accuracy, a large number of intervals (starting from 10,000) was used. Using fewer intervals resulted in insufficient precision (less than 10^{-8}).

Lab Sheet 3. Code Implementation

Implementation of the hand-written reduction:

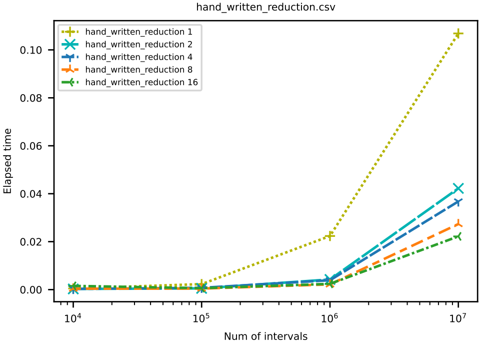
```
#pragma omp parallel
{
    // Private variable for partial sum
    double partial_sum = 0.0;
    // Parallel for loop
    #pragma omp for
    for (long i = 0; i < num_intervals; i++) {
        // Compute x for the current interval
        double x = weight * ((double)i + 0.5);
        // Apply the provided function
        double y = func(x);
        // Accumulate partial result
        partial_sum += weight * y;
    }
    #pragma omp critical
    std::cout << "Thread " << omp_get_thread_num() <<
    " partial sum = " << partial_sum << std::endl;
    sum += partial_sum;
}
```

Implementation of the omp reduction:

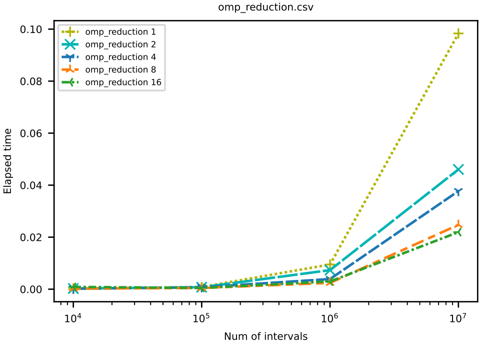
```
#pragma omp parallel for reduction(+:sum)
for (long i = 0; i < num_intervals; i++) {
    // Compute x for the current interval
    double x = weight * ((double)i + 0.5);
    // Apply the provided function
    double y = func(x);
    // Accumulate partial result
    sum += weight * y;
}
```

- A private variable was created for each thread.
- To synchronize access to the shared variable (sum) and avoid race conditions, the `#pragma omp critical` directive was used.
- Printing from threads ensures that only one thread accesses the critical section at a time.

Lab Sheet 3: Results



Implemented reduction



Original reduction

Performance is nearly identical for both the implemented and original reductions. As the number of threads increases, the calculation time decreases, while larger interval sizes lead to longer performance times.