



Matrix Correlation Optimization

Assignment Group 2

Denny Ciccio, Francesco Mecatti, Nicola Morelli

Problem: Compute the Correlation Matrix

Calculation of the **correlation matrix** of a given sequence of column vectors, denoted X, Y, \dots, W in the below image.

Correlation matrix = **square** matrix that contains the **correlation coefficient** between column X and Y at the **intersection** of row X and column Y .

The correlation matrix is an **upper triangular** matrix, the lower triangle (or vice versa) carries **duplicate information**.

X	Y		W
1.4	102.6		12.0
54.6	65.5		23.5
⋮	⋮	⋮	⋮
23.8	4.8		34.6
98.7	6.2		70.4

Input Matrix
 $N \times M$

	X	Y		W
X	1.0	$\rho(X,Y)$		$\rho(X,W)$
Y	$\rho(Y,X)$	1.0		$\rho(Y,W)$
			\ddots	
W	$\rho(W,X)$	$\rho(W,Y)$		1.0

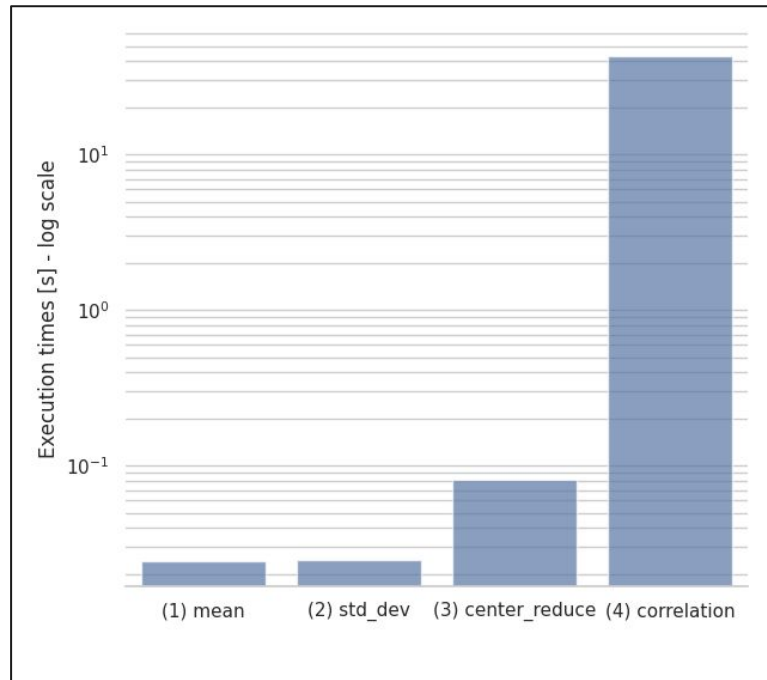
Correlation Matrix
 $M \times M$

Starting Point

$$\rho_{X,Y} = \frac{\text{cov}(X, Y)}{\sigma_X \sigma_Y} = \sum_{i=1}^N \frac{(X_i - \bar{X})}{\sigma_X \sqrt{N}} \frac{(Y_i - \bar{Y})}{\sigma_Y \sqrt{N}}$$

The formula is implemented according to this algorithm:

1. compute **mean** for each column vector
2. compute **std. dev.** for each column vector
3. **center and reduce** each column vector
4. compute **correlation** coefficients between each pair of column vectors



Data collected with `gprof` on **large** dataset

Loop Interchange

```
for (size_t j1 = 0; j1 < _PB_M - 1; j1++) {
    symmat[j1][j1] = 1.0;

    for (size_t j2 = j1 + 1; j2 < _PB_M; j2++) {

        for (size_t i = 0; i < _PB_N; i++)
            symmat[j1][j2] += (data[i][j1] * data[i][j2]);

        symmat[j2][j1] = symmat[j1][j2];
    }
}
```



```
for (size_t j1 = 0; j1 < _PB_M - 1; j1++)
    for (size_t j2 = j1 + 1; j2 < _PB_M; j2++)
        symmat[j1][j2] = 0.0;

for (size_t i = 0; i < _PB_N; i++) {
    for (size_t j1 = 0; j1 < _PB_M - 1; j1++) {
        symmat[j1][j1] = 1.0;

        for (size_t j2 = j1 + 1; j2 < _PB_M; j2++)
            symmat[j1][j2] += (data[i][j1] * data[i][j2]);
    }
}

for (size_t j1 = 0; j1 < _PB_M - 1; j1++)
    for (size_t j2 = j1 + 1; j2 < _PB_M; j2++)
        symmat[j2][j1] = symmat[j1][j2];
```

Performance counter stats for './correlation_acc':

7.494.184.899	cycles		
3.706.045.620	instructions	#	0,49 insn per cycle
999.027.913	cache-misses		
5,003132178 seconds time elapsed			



Performance counter stats for './correlation_acc':

2.287.939.837	cycles		
4.200.460.190	instructions	#	1,84 insn per cycle
39.891.848	cache-misses		
1,482564826 seconds time elapsed			

Improved **spatial locality** by turning column-major order into a **row-major order** according to the cache layout, which resulted in **25 times less cache misses**.

Parallelization with tasks

```
size_t i, j1, j2;

#pragma omp task
for (i = 0; i < _PB_N; i++)
    symmat[i][i] = 1.0;

for (j1 = 0; j1 < _PB_M - 1; j1++)
#pragma omp task
#pragma omp simd
    for (j2 = j1 + 1; j2 < _PB_M; j2++)
        symmat[j1][j2] = 0.0;

int unroll_size_ = 4;
int blocks = _PB_N / unroll_size_;
#pragma omp taskwait
```

```
for (size_t i = 0; i < blocks; i += 1)
#pragma omp task
    for (j1 = 0; j1 < _PB_M - 1; j1++)
#pragma omp simd
        for (j2 = j1 + 1; j2 < _PB_M; j2++)
        {
            size_t idx = i * unroll_size_;
            symmat[j1][j2] += (data[idx][j1] * data[idx][j2]);
            symmat[j1][j2] += (data[idx + 1][j1] * data[idx + 1][j2]);
            symmat[j1][j2] += (data[idx + 2][j1] * data[idx + 2][j2]);
            symmat[j1][j2] += (data[idx + 3][j1] * data[idx + 3][j2]);
        }
#pragma omp taskwait

for (size_t i = unroll_size_ * blocks; i < _PB_N; i++)
#pragma omp task
    for (size_t j1 = 0; j1 < _PB_M - 1; j1++)
#pragma omp simd
        for (size_t j2 = j1 + 1; j2 < _PB_M; j2++)
            symmat[j1][j2] += (data[i][j1] * data[i][j2]);
#pragma omp taskwait

for (size_t j1 = 0; j1 < _PB_M - 1; j1++)
#pragma omp task
#pragma omp simd
    for (size_t j2 = j1 + 1; j2 < _PB_M; j2++)
        symmat[j2][j1] = symmat[j1][j2];

symmat[_PB_M - 1][_PB_M - 1] = 1.0;
```

Parallel for

Parallelization on **host cores** combined with **SIMD** instructions showed to be the best-performing approach.

Code organized in such a way that **race conditions** are **prevented** without synchronization mechanisms.

Explicit **loop unrolling** increases work for each iteration, **amortizing** thread activation cost.

```
#pragma omp parallel for
for (size_t j1 = 0; j1 < _PB_M - 1; j1++) {
    symmat[j1][j1] = 1.0;

    for (size_t j2 = j1 + 1; j2 < _PB_M; j2++)
        symmat[j1][j2] = 0.0;
}

int unroll_size_ = 4;
int blocks = _PB_N / unroll_size_;

for (size_t i = 0; i < blocks; i += 1) {
    #pragma omp parallel for schedule(dynamic)
    for (size_t j1 = 0; j1 < _PB_M - 1; j1++) {
        #pragma omp simd
        for (size_t j2 = j1 + 1; j2 < _PB_M; j2++) {
            size_t idx = i * unroll_size_;
            symmat[j1][j2] += (data[idx][j1] * data[idx][j2]);
            symmat[j1][j2] += (data[idx + 1][j1] * data[idx + 1][j2]);
            symmat[j1][j2] += (data[idx + 2][j1] * data[idx + 2][j2]);
            symmat[j1][j2] += (data[idx + 3][j1] * data[idx + 3][j2]);
        }
    }
}

for (size_t i = unroll_size_ * blocks; i < _PB_N; i++)
    for (size_t j1 = 0; j1 < _PB_M - 1; j1++)
        for (size_t j2 = j1 + 1; j2 < _PB_M; j2++)
            symmat[j1][j2] += (data[i][j1] * data[i][j2]);

#pragma omp parallel for
for (size_t j1 = 0; j1 < _PB_M - 1; j1++) {
    #pragma omp simd
    for (size_t j2 = j1 + 1; j2 < _PB_M; j2++) {
        symmat[j2][j1] = symmat[j1][j2];
    }
}
```

Execution times and speedups

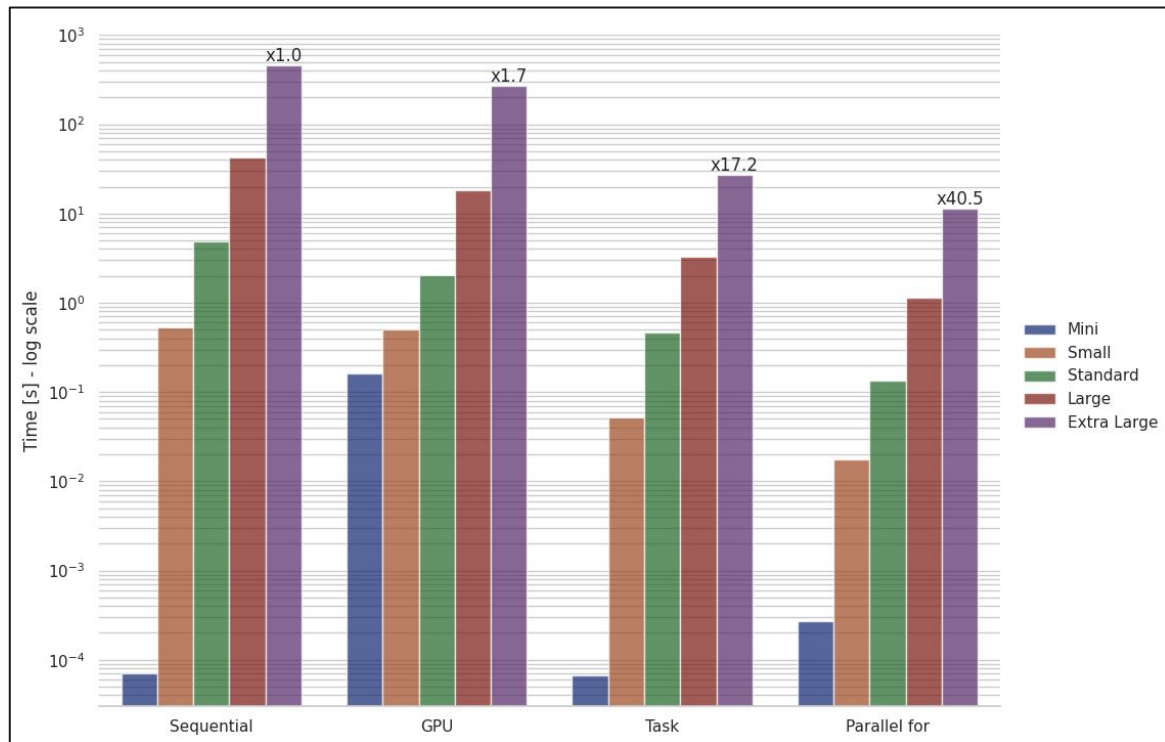


Dataset dimension	BASE_HASHES	Sequential	Task optimization	Parallel for optimization
Mini	962.000005	0,000070s	0,000066s	0,000268s
Small	249002.014697	0,526751s	0,051703s	0,017396s
Standard	998001.983040	4,916482s	0,464806s	0,135351s
Large	3996002.722686	42,652974s	3,293214s	1,151873s
Extra large	15991995.60760	460,815627s	26,792492s	11,367700s

Execution times and speedups

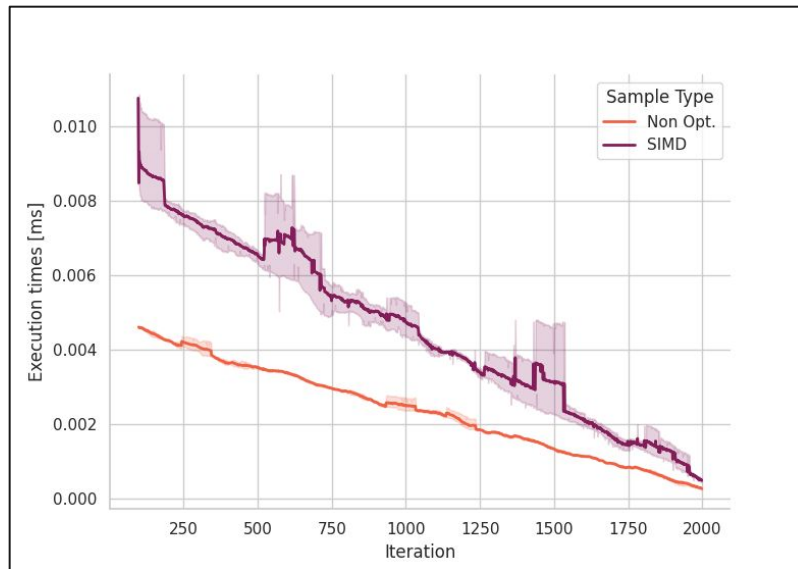
Alternative approaches based on **tasks** and **GPU offloading** tested but led to minor speedups.

The baseline on the graph refers to a **sequential** execution with **loop interchange** transformation applied, which is **4.5 times faster** than **original sequential** code.



Execution times of **four** different **approaches** have been profiled for **each** dataset size.

Final Considerations



Innermost loops exec. time vs outermost iteration @ large dataset

- (1) **Dynamic scheduling** works better than static one since **workload varies** across **iterations**.
- (2) **GPU offloading underperforms** due to the **memory copyin/copyout** between host and target, given than **OpenMP** does **not exploit** Ampere's **unified memory** architecture. In our case, memory copy is a huge overhead compared to the much smaller computation time.