Matrix Correlation Optimization with CUDA

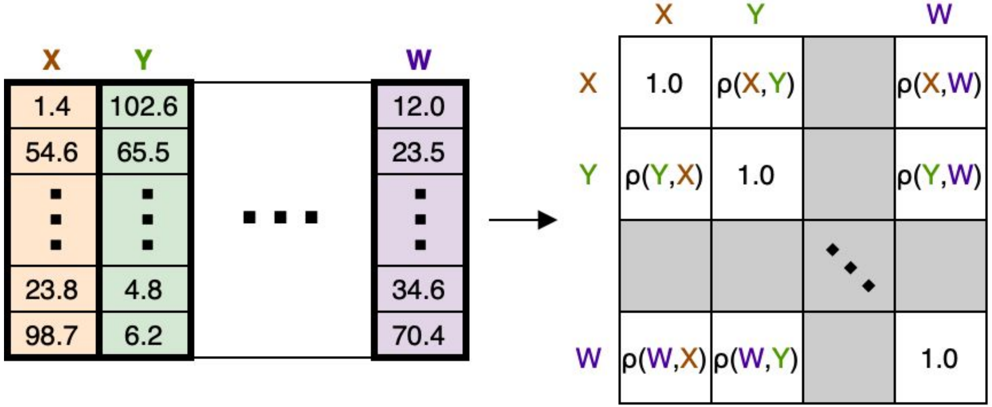
Assignment Group 2
Denny Ciccia, Francesco Mecatti, Nicola Morelli

Problem: Compute the Correlation Matrix

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

Algorithm: mean -> std. dev. -> center & reduce -> correlation coefficients.
The **hotspot** is the computation of the **correlation coefficients**.

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



Input Matrix
N x M

Correlation Matrix
M x M

Memory Configuration

We decided to use the **Unified Virtual Memory**, the optimal configuration for Jetson Nano integrated GPU.

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	91.26%	100.13ms	1	100.13ms	100.13ms	100.13ms	void gemm_v3<float>(float*, float*, unsigned long)
	7.07%	7.7556ms	1	7.7556ms	7.7556ms	7.7556ms	[CUDA memcpy DtoH]
	1.35%	1.4844ms	1	1.4844ms	1.4844ms	1.4844ms	[CUDA memcpy HtoD]
	0.32%	350.15us	1	350.15us	350.15us	350.15us	[CUDA memset]
API calls:	75.08%	335.53ms	2	167.77ms	2.4859ms	333.05ms	cudaMalloc
	24.84%	111.00ms	2	55.501ms	1.9037ms	109.10ms	cudaMemcpy

Pageable

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	97.51%	255.90ms	1	255.90ms	255.90ms	255.90ms	void gemm_v3<float>(float*, float*, unsigned long)
	1.41%	3.7036ms	1	3.7036ms	3.7036ms	3.7036ms	[CUDA memcpy HtoD]
	0.92%	2.4193ms	1	2.4193ms	2.4193ms	2.4193ms	[CUDA memset]
	0.16%	414.03us	1	414.03us	414.03us	414.03us	[CUDA memcpy DtoH]
API calls:	55.44%	343.18ms	4	85.795ms	2.6312ms	335.20ms	cudaHostAlloc
	42.14%	260.86ms	2	130.43ms	3.9333ms	256.93ms	cudaMemcpy

Pinned

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	100.00%	100.07ms	1	100.07ms	100.07ms	100.07ms	void gemm_v3<float>(float*, float*, unsigned long)
API calls:	77.07%	354.66ms	4	88.664ms	2.6910ms	346.37ms	cudaMallocManaged
	21.85%	100.57ms	1	100.57ms	100.57ms	100.57ms	cudaDeviceSynchronize

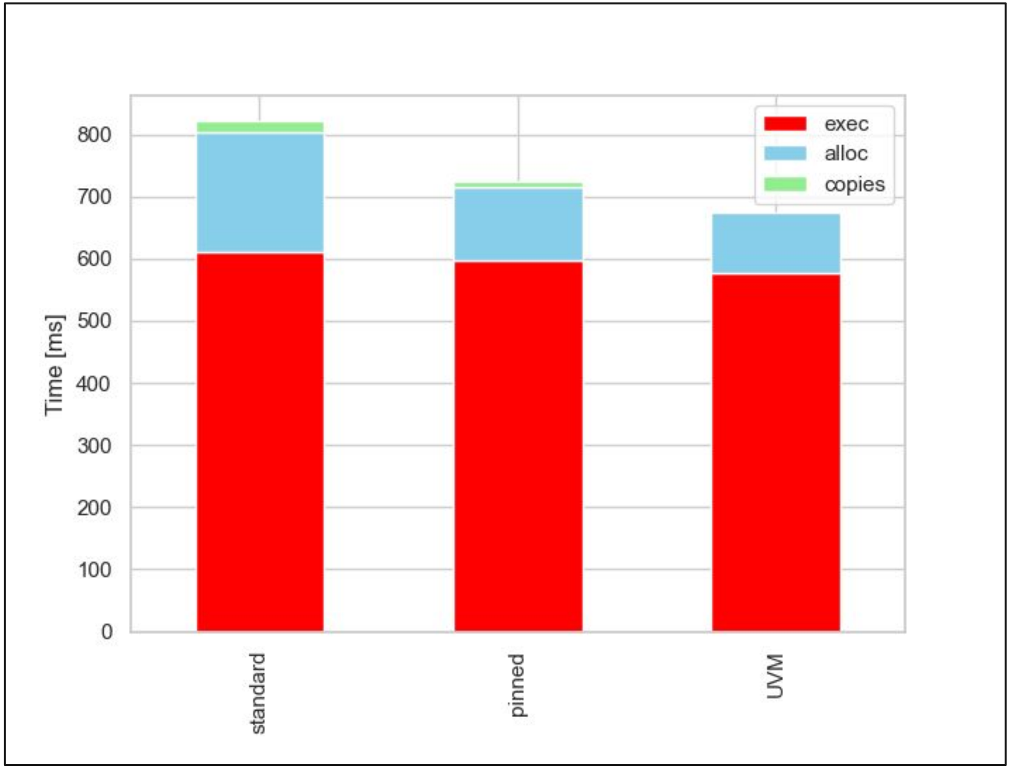
UVM

nvprof profiling with Standard Dataset

UVM Benefits

Allocation time is also reduced, since just one allocation primitive is invoked for each matrix.

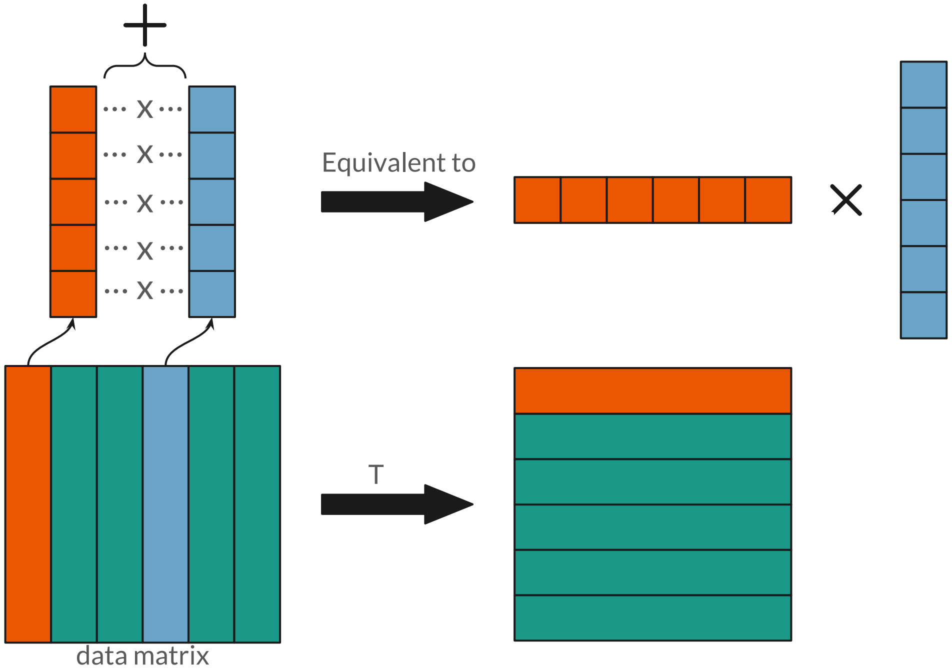
By storing data on the **UVM** copy time is zeroed, saving 150ms.



Execution times of **three** different **memory models** with Large Dataset.

Kernel Function

The **hotspot loop** is equivalent to a **matrix multiplication**.
We **transposed** the data matrix to improve spatial locality.



Kernel Function

Tiled **GEMM**:

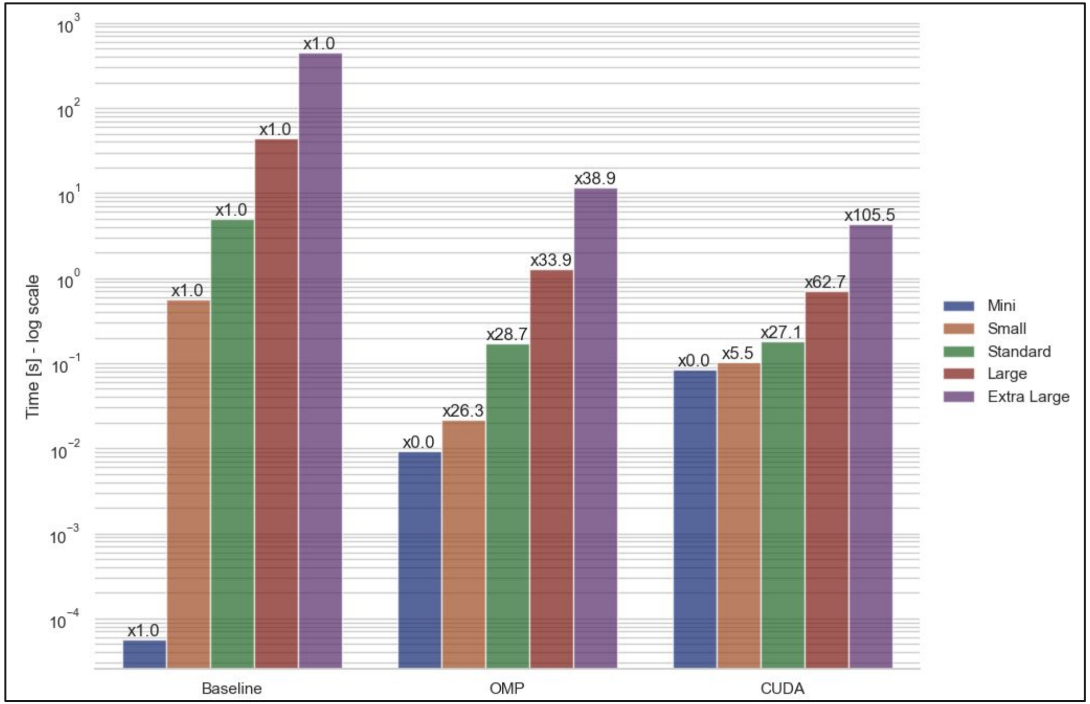
- Every block computes a tile of the total correlation matrix using the shared memory
- To avoid useless computation we made sure to compute only the higher half of the matrix since it is symmetric with respect to the principal diagonal

```
template <typename T>
__global__ void gemm_v3(T *__restrict__ a, T *__restrict__ c, size_t n)
{
    __shared__ T as[BLOCK_SIZE][BLOCK_SIZE];
    __shared__ T bs[BLOCK_SIZE][BLOCK_SIZE];

    const size_t a_row = threadIdx.y + blockIdx.y * blockDim.y;
    const size_t b_col = threadIdx.x + blockDim.x * blockIdx.x;

    T accum = (T)0;
    if ((b_col + BLOCK_SIZE) >= a_row)
    {
        for (int kb = 0; kb < (n + BLOCK_SIZE - 1) / BLOCK_SIZE; ++kb)
        {
            size_t a_col = threadIdx.x + kb * blockDim.x;
            size_t b_row = threadIdx.y + kb * blockDim.y;

            as[threadIdx.y][threadIdx.x] = (a_row < n && a_col < n) ? a[a_row * n + a_col] : (T)0;
            bs[threadIdx.y][threadIdx.x] = (b_row < n && b_col < n) ? a[b_col * n + b_row] : (T)0;

            __syncthreads();

            for (size_t i = 0; i < BLOCK_SIZE && (kb * BLOCK_SIZE + i) < n; i++)
                accum += as[threadIdx.y][i] * bs[i][threadIdx.x];

            __syncthreads();
        }

        if (a_row < n && b_col < n)
            c[a_row * n + b_col] = accum;
    }
}
```

Speedup and OMP comparison

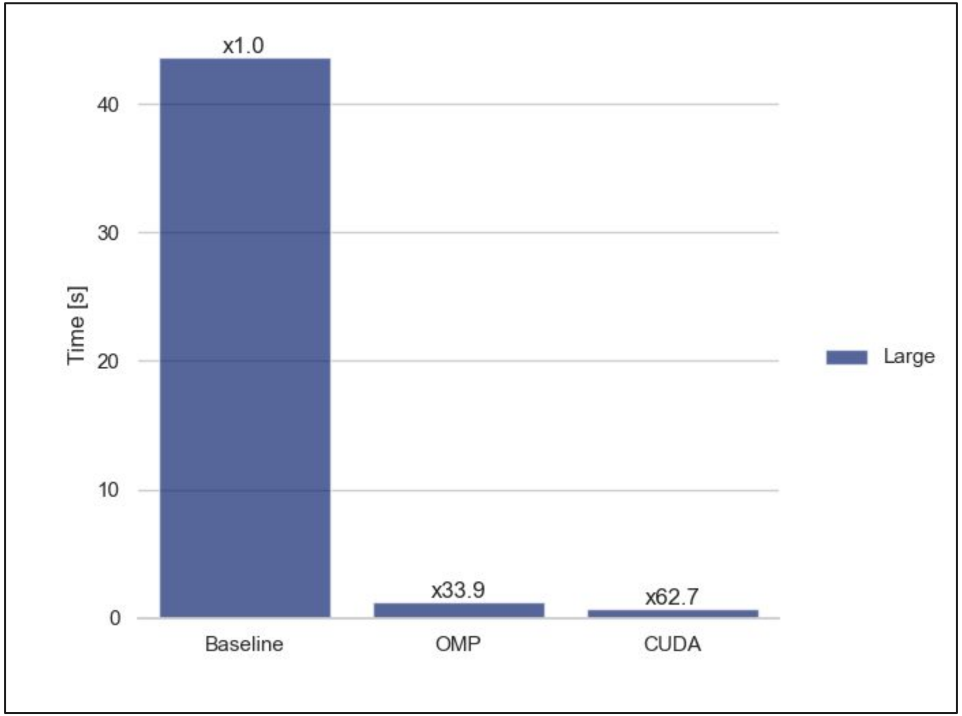
Speedups for smaller dataset sizes are higher in OpenMP than CUDA.

Baseline time is sequential time without access pattern optimization (aka no loop interchange).



Execution times of **three** different **approaches** have been profiled for **each dataset size**.
The label over each bar represents the speed up w.r.t. the baseline, sequential, time.

Speedup and OMP comparison: Large Dataset Detail



Execution times for **large** dataset. The lower the better

Results

Dataset	Baseline [sec]	Omp [sec] (speedup)	Cuda [sec] (speedup)
MINI	0.000057	0.00924141(0.006)	0.0836763(0.0006)
SMALL	0.562221	0.0214025(26.3)	0.10178(5.52)
STANDARD	4.96188	0.172975(28.7)	0.183133(27.1)
LARGE	43.6303	1.28537(33.9)	0.695554(62.7)
EXTRA LARGE	455.688	11.7039(38.9)	4.32089(105.5)