HW 3 Report

Kelvin Silva

AMS 148 GPU Programming

Matrix Vector Product , CPU vs Parallel Timing Comparison

Methodology: Used sys/time.h timestamp and associated functions.

Size	CPU (us)	CUDA (us)
16	116	78
128	590	66
2048	47 823	625
32768	1,1098 791	129,705

Conclusion:

CPU serial matrix vector product time complexity scaled exponentially with size of array. However we see the CUDA Parallel version as having a faster timing in the matrix vector product as the size of the matrices scale up.

Matrix Transpose, No Shared Memory vs Shared Memory usage

Methodology: same as previous

Size	No Shared Memory (us)	Shared Memory (us)
16	98	92
2048	90	87
128	2333	1147
2048	744,119	707,124
32768	1,334,720	1,273,845

Conclusion:

The shared and no shared memory implementations seem to have similar running time. However the shared memory implementation was slightly faster. Over many iterations or function calls to Matrix Transpose, these small differences can add up.

Matrix Multiplication, Non Transposed Multiplication vs Transposed Multiplication

Methodology: Use sys/time.h library to measure time. In the transposed version, I measured the time for both the transpose operation and the multiplication operation.

Size	Matrix Multiplication No Transpose (us)	Matrix Multiplication With Transpose (us)
1024	605,911	629,980
2048	719,954	739,960
16384	5,406,028	5,671,278

32768	35,333,640	34,281,935

Conclusion:

Note that the matrix multiplication with transpose time was measured with both the multiply operation and transpose operation.

Here we see very similar times for both the Matrix Mul Transpose and Matrix Mul without transpose. However, getting to the 32k sizes we see that the transpose multiply performance beats the non transpose performance. Even though the times appear to be similar we must take note that the transpose was measured with the matrix multiplication operation in the "Matrix Multiplication with Transpose" sample.

Had the transpose been calculated and the matrix multiplication routine been timed by itself, it would outperform the matrix multiplication without transpose.

Thus, suppose that in a Linear Algebra task a transpose was needed for other reasons and the programmer needed to finally perform a matrix multiplication.

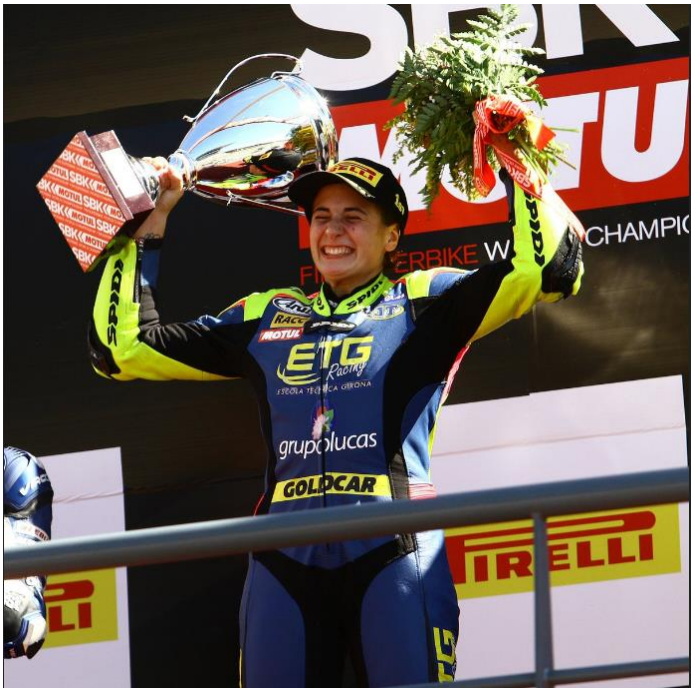
If the Matrix Multiplication with transpose was used, it would outperform the naïve Matrix Multiplication.

However there is no benefit gained by transposing a matrix solely to perform a multiplication due to the overhead of transposing.

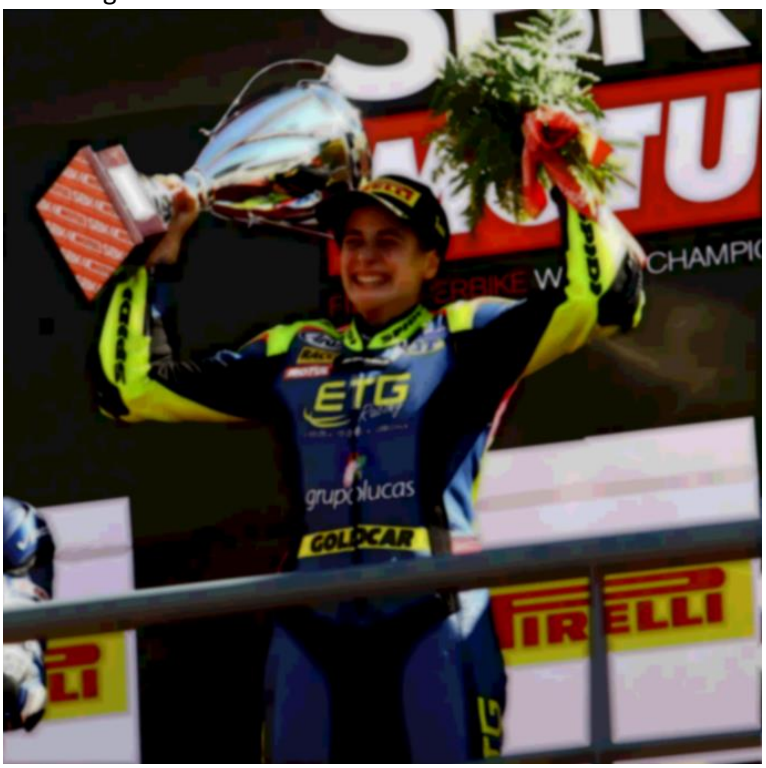
The overhead of creating a transpose is only worth it if the transpose of the matrix is necessary and used in other parts of a Linear Algebra calculation task and the Matrix Multiplication with Transpose routine was used in place of a normal Matrix Multiplication.

I chose to do the Gaussian Blur Program. Code is attached to a compressed folder.

Non blurred image:



Blur Image CUDA:



Conclusion:

I implemented this program using the stencil provided in the PDF. Stencil code was straightforward, however the main challenge was on using CImg , learning how it worked, and learning how the RGB data

was laid out, as well as figuring out how to setup the proper CUDA kernel for a 3 dim (RGB) CImg data structure.

Conclusion For Whole HW:

Overall it was a fun experience and this HW assignment helped me to become more proficient in CUDA programming.