A Strassen-based CUDA Implementation of A^tA Matrix Multiplication

Davide Sforza

Sapienza University of Rome, Italy

1 Introduction

Matrix multiplication A^tA is a linear algebra operation which represent an intermediate step in the solution of a large number of scientific problems. The aim of this work is to provide a GPU implementation of the algorithm proposed by Arrigoni et al. [1]. In their work, they designed a fast and parallel Strassen-based algorithm to compute the product A^tA and implemented it exploiting the MPI paradigm.

Strassen's algorithm was the first matrix multiplication algorithm with an asymptotic complexity lower than $O(N^3)$ and it is still one of the most effective. Given the central role of matrix multiplication in linear algebra applications, a wide set of GPU implementation of Strassen's algorithm have been proposed during the years. In particular, the A^tA product implementation presented in this document is based on the CUDA implementation of the Strassen's algorithm proposed by Lai et al. [3].

2 Design and Implementation

This work can be viewed as an extension of the Strassen's algorithm implementation proposed by Lai et al. [3]. Their code is available on GitHub (`https://github.com/paiweilai/strassen-cuda`). All matrix operations, i.e. matix addion, multiplication and transposition, are accelerated on the GPU exploiting the *cuBLAS* library, an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVIDIA CUDA run-time.

Algorithm 1 AtA-GPU

```
procedure ATA($A$, $C$, $m$, $n$, depth)
    $m_2 \leftarrow \lfloor m/2 \rfloor$
    $n_2 \leftarrow \lfloor n/2 \rfloor$
    define $W_1, W_2 \in \mathbb{R}^{n_2 \times n_2}$

    ▷ Cutoff criteria
    $mm \leftarrow CUTOFF/m_2$
    $nn \leftarrow CUTOFF/n_2$
    stop $\leftarrow (mm + nn) \leq 2$

    if depth $\leq 1$ or stop then
        $W_1 \leftarrow A_{11}^t \cdot A_{11}$
        $W_2 \leftarrow A_{21}^t \cdot A_{21}$
        $C_{11} \leftarrow W_1 + W_2$
        $W_1 \leftarrow A_{12}^t \cdot A_{12}$
        $W_2 \leftarrow A_{22}^t \cdot A_{22}$
        $C_{22} \leftarrow W_1 + W_2$
        $W_1 \leftarrow A_{12}^t \cdot A_{11}$
        $W_2 \leftarrow A_{22}^t \cdot A_{21}$
        $C_{21} \leftarrow W_1 + W_2$
    else
        ATA($A_{11}$, $W_1$, $m_2$, $n_2$, depth$-1$)
        ATA($A_{21}$, $W_2$, $m_2$, $n_2$, depth$-1$)
        $C_{11} \leftarrow W_1 + W_2$
        ATA($A_{12}$, $W_1$, $m_2$, $n_2$, depth$-1$)
        ATA($A_{22}$, $W_2$, $m_2$, $n_2$, depth$-1$)
        $C_{22} \leftarrow W_1 + W_2$
        STRASSEN($A_{12}^t$, $A_{11}$, $W_1$, $n_2$, $m_2$, $n_2$, depth$-1$)
        STRASSEN($A_{22}^t$, $A_{21}$, $W_2$, $n_2$, $m_2$, $n_2$, depth$-1$)
        $C_{21} \leftarrow W_1 + W_2$
    end if

    ▷ Dynamic peeling fix-up
    $\begin{pmatrix} c_{21} & c_{22} \end{pmatrix} \leftarrow \begin{pmatrix} a_{12} & a_{22} \end{pmatrix}^t \cdot A$
    $C_{11} \leftarrow a_{21}^t \cdot a_{21} + C_{11}$

end procedure
```

cuBLAS library enables matrix addition and multiplication on GPU simply calling the **cublas$\langle t\rangle$geam** and the **cublas$\langle t\rangle$gemm** functions, respectively. Moreover, it is possible to transpose one or both matrices given in input passing the flag **CUBLAS_OP_T** to the API. Computations can be performed both in single precision (setting $\langle t\rangle$ = S) and double precision ($\langle t\rangle$ = D).

The algorithm proposed by Arrigoni et al. [1] computes the product A^tA in a recursive fashion. It works dividing the input matrix A and the output matrix C into four sub-matrices at each recursive step. The four sub-matrices are defined as:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} \tag{1}$$

Given $A \in \mathbb{R}^{m\times n}$, all sub-matrices of A have size $\lfloor \frac{m}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$, while all sub-matrices of C have size $\lfloor \frac{n}{2} \rfloor \times \lfloor \frac{n}{2} \rfloor$. Since the product A^tA is always symmetric, only the lower triangular part of C needs to be computed. Thus, the algorithm never works on C_{12}, being $C_{12} = C_{21}^t$.

The remaining three sub-matrices are obtained as shown in the pseudo-code reported algorithm 1. In the base-case, three matrix additions and six matrix multiplications are performed exploiting the *cuBLAS* API, while in the recursive one standard matrix multiplications are substituted by four recursive calls to AtA-GPU and two to the Strassen's algorithm.

The base-case is entered if the maximum depth of recursion is reached or if the the input matrix is smaller than a fixed value denoted *CUTOFF*. This threshold has been set empirically to 1536 for single-precision and to 256 for double-precision computations (see section 3).

Odd-sized matrices are handled through a technique known as *dynamic peeling*. If matrices A and C are odd-sized, they partitioned as follow:

$$A = \begin{pmatrix} A_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \quad C = \begin{pmatrix} C_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \tag{2}$$

where A_{11} and C_{11} are even-sized matrices, a_{12} and c_{12} are column vectors, a_{21} and c_{21} are row vectors and a_{22} and c_{22} are scalars. The majority of the computation, $C_{11} = A_{11}^t \cdot A_{11}$, is performed using the main algorithm, while the contribution of vectors and scalars are calculated separately, according to the formulas below:

$$C_{11} = a_{21}^t \cdot a_{21} + C_{11} \tag{3}$$

$$\begin{pmatrix} c_{21} & c_{22} \end{pmatrix} = \begin{pmatrix} a_{12} & a_{22} \end{pmatrix}^t \cdot A \tag{4}$$

Remembering we only need to compute the lower triangular part of C, we can ignore the column vector c_{12} (we have $c_{12} = c_{21}^t$).

3 Experiments

The implementation has been tested using an NVIDIA GeForce GTX 1050 GPU with 4GB of GDDR5 memory (CUDA *Compute Capability* 6.1). Experiments have been carried out running the algorithm on a set randomly generated square matrices, both even and odd-sized to take into account the overhead due to the *dynamic peeling* fix-up. Performances have been evalueted in term of speed-up with respect to the execution time of *cuBLAS* matrix multiplication.

First experiments have been run to set the cutoff point on which to enter the base-case. As we can notice in figure 1, AtA-GPU algorithm performs better than standard matrix multiplication with matrices with more than 1536 rows in single precision and with more than 256 rows in double precision.

Similarly, the algorithm has been evaluated using different values for the maximum depth of recursion. Figure 2 shows the speed-up on random matrices of size $10^4 \times 10^4$. Best performances are achieved with a depth of three recursive calls in single precision and four in double precision.

Average execution time and speed-up to vary the input matrix size are reported in figures 3 and 4. The recursive implementation results faster than the one-level one, in particular for double-precision computations. The maximum speed-up is reached in

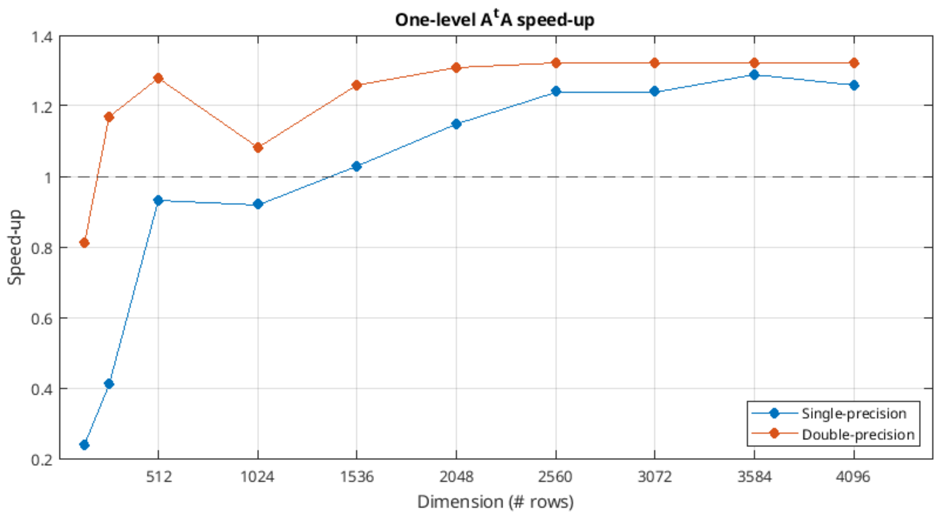

Figure 1: Speed-up of one-level AtA-GPU w.r.t. *cuBLAS gemm* on matrices of different size.

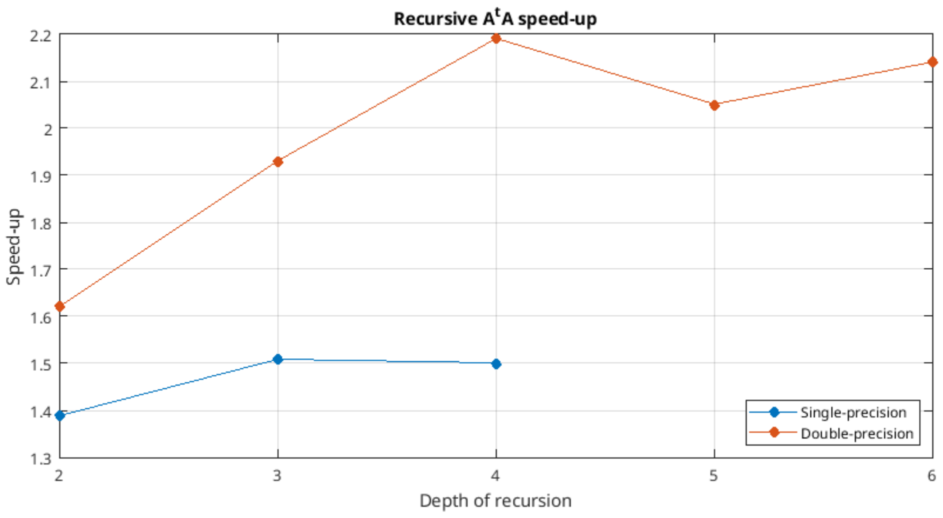

Figure 2: Speed-up of AtA-GPU with differt depth of recursion w.r.t. *cuBLAS gemm* on $10^4 \times 10^4$ random matrices.

double precision (1.33x for the one-level implementation, 2.18x for the recursive one), while the gain in performances is a bit lower, but still good, in single precision (1.31x one-level version, 1.65x recursive). *Dynamic peeling* seems to affect more single-precision computations and its impact tend to become less relevant as the input matrix size increases.

4 Conclusion

In this short document a Strassen-based CUDA implementation of the product A^tA has been presented. It works with rectangular matrices of arbitrary size and offers good performances with a speed-up up to 1.65x for single precision and 2.18x for double precision with respect to *cuBLAS* matrix multiplication.

However, some recent implementation of Strassen's algorithm, such as the one proposed by Huang et al. [2], have shown that using custom CUDA kernels instead of *cuBLAS* subroutines can ensure better performances, particularly for relatively small matrices. A possible future improvement to the current implementation could be to switch to custom CUDA kernels to compute matrix additions and multiplications.

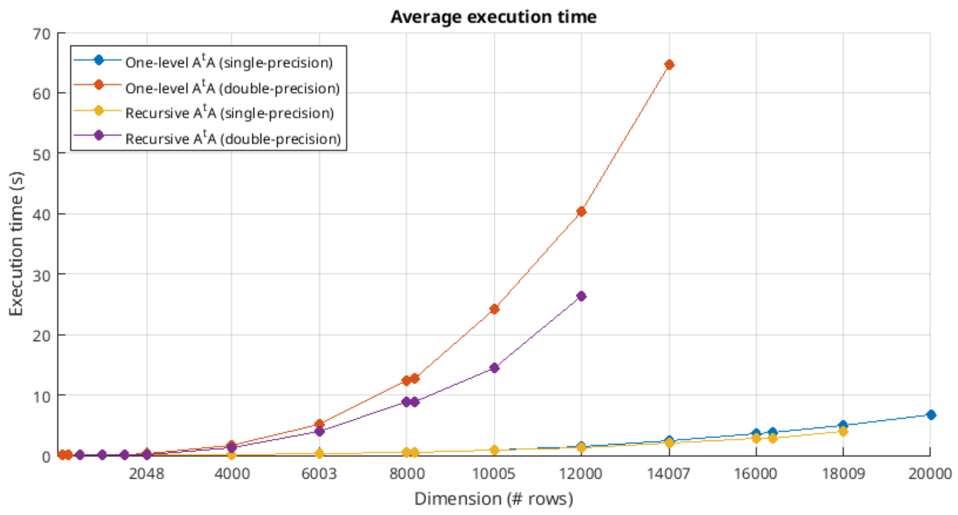

Figure 3: Average execution time of AtA-GPU on matrices of different size.

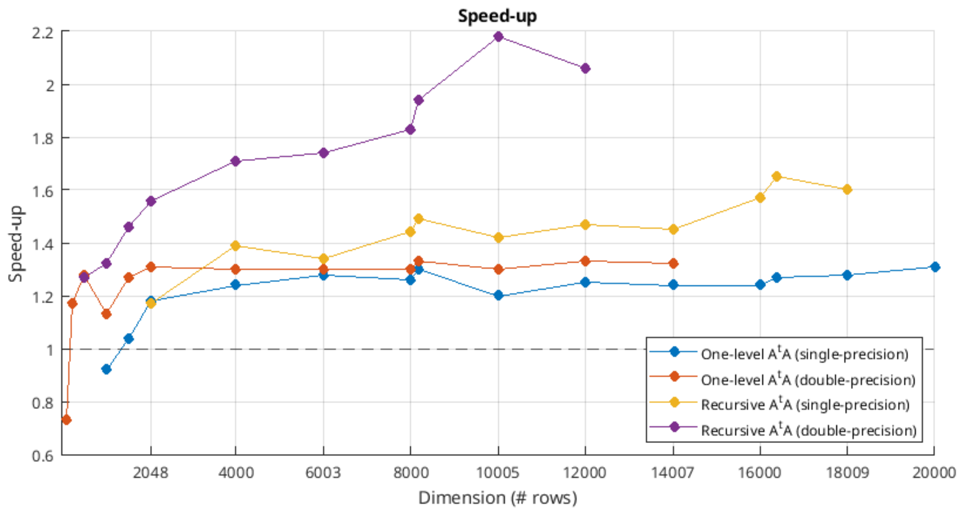

Figure 4: Speed-up of AtA-GPU w.r.t. *cuBLAS gemm* on matrices of different size.

References

[1] Arrigoni, V., and Massini, A. Fast strassen-based A^tA parallel multiplication. *arXiv preprint arXiv:1902.02104* (2019).

[2] Huang, J., Yu, C. D., and Geijn, R. A. v. d. Strassen's algorithm reloaded on gpus. *ACM Transactions on Mathematical Software (TOMS) 46*, 1 (2020), 1–22.

[3] Lai, P.-W., Arafat, H., Elango, V., and Sadayappan, P. Accelerating strassen-winograd's matrix multiplication algorithm on gpus. In *20th Annual International Conference on High Performance Computing* (2013), IEEE, pp. 139–148.