Strassen's Matrix Multiplication Algorithm Is Still Faster

PAOLO D'ALBERTO, AMD

Recently, reinforcement algorithms discovered new algorithms that may replace Strassen's original fast matrix multiplication. Likely, there are even more hidden gems by way of factoring the operand matrices. The discovery really jump-started a wave of excitement and a flourishing of publications. However, little work on implementations and applications.

We present *Matrix Flow*, this is a simple Python project for the automatic formulation, design, implementation, code generation, and execution of fast matrix multiplication algorithms for CPUs and GPUs. We play with no module-2 algorithms, because even less general, and for square matrices, for simplicity.

We have an original way to combine fast algorithms by means of factorizing the operand matrices and prove them correct. We describe these algorithm as a Data flow and matrix data partitioning: a Directed Acyclic Graph. We show that Strassen's original algorithm is still the top choice even for modern GPUs. We also address error analysis in double precision, because integer computations are correct, always.

ACM Reference Format:
Paolo D'Alberto. 2023. Strassen's Matrix Multiplication Algorithm Is Still Faster. *J. ACM* –, –, Article — (March 2023), 8 pages.

1 INTRODUCTION

In practice, there are quite a few algorithms for *fast* matrix multiplication. We used to have a few pages of references; now institutions are actually stepping in and providing a complete list. Although the list of algorithms is long, the list of implementation for old and new architectures is limited (if not existent). We are interested in the computation with matrices, not just a single element, and we compare the trade matrix multiplication for pre and post matrix additions. This could be important when we tray to translate fast algorithms into practical implementations.

The researchers at DeepMind presented a new methodology to find new and faster matrix multiplications [1]. They provide the algorithms in matrix forms and, easily, we can generate, and execute element-wise algorithms. There are new algorithms and They show the relative advantage performance with respect to the classic matrix multiplication for GPUs and TPUs.

Here, we show that Strassen's algorithm [2] is still king of the hill. We explore a comparison across several algorithms using the standard arithmetic and in particular double precision matrices (no $\mathbb{Z}_2$ modular arithmetic because Strassen's algorithm does not have equivalent). In this scenario, we can determine a priori the relative advantages of all algorithms (7% improvement per recursion we shall expand). More interesting, we can show that *deeper* algorithms such as the one recently discovered do not have all the advantages (they are cranked up to have) when other constraints are present.

In the following, we present a python project that we call *Matrix Flow*. Here, you can take any of the algorithms for square matrices, we use the DeepMind's algorithms as repository, and create

Author's address: Paolo D'Alberto, , AMD, 2100 Logic Dr, San Jose, California, 95124.

0004-5411/2023/3-ART— $15.00
https://doi.org/

complex implementations you can run in CPU, GPU and in principle extend to any accelerator using the appropriate interface.

We have a few goals in mind for this work. First, we want to make available algorithms and Python implementations: both proven correct. Second, we want to provide tools for the implementation of these algorithms for highly efficient systems. Python is flexible, with such a nice abstraction level that writing code for matrix algorithms is a breeze. Efficiency and performance is not always Python forte, however writing interfaces for C/C++ is relatively simple. Interfaces are not a very good solution either and they are more a nice intermediary solution. So we provide tools to build entire algorithms in C/C++ for CPU and GPUs and still prove them correct and validate them easily.

So this is a research paper and a software project introduction. Eventually, we would like any one to play with the project and we would like to squeeze into the presentation and share a few lessons learned that are not trivial, sometime original, and currently important for other applications where compiler tools are designed to optimize artificial intelligence applications.

2 MATRICES AND PARTITIONS.

We start by describing the basic components of our algebra: matrices, operations, and matrix partitions . This will help us understand and describe fast matrix multiplications constructively.

In this work we work mostly with matrices and scalar, but vector are easily added. A matrix $\boldsymbol{A}$, this is a square matrix of size $\mathbb{R}^{n\times n}$. Every element of the matrix is identified by a row i and a column j as $\boldsymbol{A}_{i,j}$. A scalar α is number in $\mathbb{R}$.

- $\boldsymbol{B} = \alpha\boldsymbol{A} = \boldsymbol{A}\alpha$ so that $\boldsymbol{B}_{i,j} = \alpha\boldsymbol{A}_{i,j}$ and the real multiplication.
- $\boldsymbol{B} = \alpha\boldsymbol{B} + \beta\boldsymbol{C} = \beta\boldsymbol{C} + \alpha\boldsymbol{B}$ is matrix addition and $\boldsymbol{B}_{i,j} = \alpha\boldsymbol{B}_{i,j} + \beta\boldsymbol{C}_{i,j}$
- $\boldsymbol{B} = \alpha(\boldsymbol{B} + \boldsymbol{C}) = \alpha\boldsymbol{B} + \alpha\boldsymbol{C}$
- $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$ is matrix multiplication and $\boldsymbol{C}_{i,j} = \sum_{k=1}^{n} \boldsymbol{A}_{i,k}\boldsymbol{B}_{k,j}$
- $\boldsymbol{y} = \alpha\boldsymbol{A}\boldsymbol{x}$ is matrix by vector multiplication and $\boldsymbol{y}_i = \alpha \sum_{k=1}^{n} \boldsymbol{A}_{i,k}\boldsymbol{b}_k$ (you may find this in the software package).

Great, we have the operands and the operations. Let us introduce the last component for the description of our algorithms: Partitions. The double subscript for the definition of elements it is for exposition purpose and we do not use element operation any where. We introduce here the single subscript that we use a lot. Consider a matrix $\boldsymbol{A} \in \mathbb{R}^{n\times n}$, we can consider the matrix as a composition of $\boldsymbol{A}_i \in \mathbb{R}^{\frac{n}{2}\times\frac{n}{2}}$ sub-matrices.

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_0 & \boldsymbol{A}_1 \\ \boldsymbol{A}_2 & \boldsymbol{A}_3 \end{pmatrix} = \{\boldsymbol{A}_0, \boldsymbol{A}_1, \boldsymbol{A}_2, \boldsymbol{A}_3\} \tag{1}$$

The property is that $\boldsymbol{A}_i$ are non overlapping matrices and this is easily generalized to any factor of n for example p so that $n = k * p$. For $n = p = 3$ and $\boldsymbol{A}_i \in \mathbb{R}^{\frac{n}{3}\times\frac{n}{3}}$ and $0 \le i \le p^2 - 1$:

$$\boldsymbol{A} = \begin{pmatrix} \boldsymbol{A}_0 & \boldsymbol{A}_1 & \boldsymbol{A}_2 \\ \boldsymbol{A}_3 & \boldsymbol{A}_4 & \boldsymbol{A}_5 \\ \boldsymbol{A}_6 & \boldsymbol{A}_7 & \boldsymbol{A}_8 \end{pmatrix} = \{\boldsymbol{A}_0, \boldsymbol{A}_1, \boldsymbol{A}_2, \boldsymbol{A}_3, \boldsymbol{A}_4, \boldsymbol{A}_5, \boldsymbol{A}_6, \boldsymbol{A}_7, \boldsymbol{A}_8\} \tag{2}$$

In practice, if we have a partition $\{\boldsymbol{A}_i\}$ and we know the factor p, we know the matrix $\boldsymbol{A}$ completely.

Let us represent the simplest matrix multiplication using $2x2$ partition:

$$\begin{pmatrix} \boldsymbol{C}_0 & \boldsymbol{C}_1 \\ \boldsymbol{C}_2 & \boldsymbol{C}_3 \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}_0 & \boldsymbol{A}_1 \\ \boldsymbol{A}_2 & \boldsymbol{A}_3 \end{pmatrix} * \begin{pmatrix} \boldsymbol{B}_0 & \boldsymbol{B}_1 \\ \boldsymbol{B}_2 & \boldsymbol{B}_3 \end{pmatrix} \tag{3}$$

Using the factor $p = 2$, we have in general

$$\begin{pmatrix} \boldsymbol{C}_0 = \sum_{k=0}^{1} \boldsymbol{A}_k \boldsymbol{B}_{2k} & \boldsymbol{C}_1 = \sum_{k=0}^{1} \boldsymbol{A}_k \boldsymbol{B}_{2k+1} \\ \boldsymbol{C}_2 = \sum_{k=0}^{1} \boldsymbol{A}_{2+k} \boldsymbol{B}_{2k} & \boldsymbol{C}_3 = \sum_{k=0}^{1} \boldsymbol{A}_{2+k} \boldsymbol{B}_{2k+1} \end{pmatrix} \tag{4}$$

The single subscript is the original format in fast algorithms describing a matrix operand as partitioned into smaller matrices. The authors in [1] use a double subscript because the intend to identify a single element. Strangely enough, the single subscript has an implementation advantage.

Equation 4, as matrix computation, represent naturally a recursive computation.

A Strassen's algorithm or any fast algorithm has the following format:

$$C_i = \sum_{j=0}^{6} c_{i,j} P_j = \sum_{j=0}^{6} c_{i,j} T_j * S_j = \sum_{j=0}^{6} c_{i,j} \Big[T_j = \big(\sum_{k=0}^{3} a_{k,j} A_k \big) \Big] * \Big[S_j = \big(\sum_{\ell=0}^{3} b_{\ell,j} B_\ell \big) \Big] \tag{5}$$

There are seven products, each product can be done recursively, but first we must combine submatrices of the operands accordingly to a specific set of coefficients. The coefficients $c_{i,j}$, $a_{k,j}$, and $b_{\ell,j}$ belong to three sparse matrices. Let us take the Strassen's algorithm but represented as DeepMind would use it for the computation of Equation 5. We choose to use single subscripts for C_i, A_j, B_k, because the notation is simpler (i.e., fewer indices), and because we implicitly use a row major layout of the sub-matrices making the single index completely determined.

$$\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}^t = f[\text{`}2, 2, 2\text{`}] \tag{6}$$

Where the matrices $\boldsymbol{a}, \boldsymbol{b}$, and $\boldsymbol{c}$ are integer matrices with coefficients [-1,0,1], but they do not need to be integers, described as follow plus some useful notations:

$$\boldsymbol{a} = \left(\begin{array}{cccccccc} A_0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ A_1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ A_2 & 1 & 1 & 1 & 0 & 1 & 0 & 0 \\ A_3 & -1 & -1 & -1 & 0 & 0 & 0 & 1 \\ \hline & T_0 & T_1 & T_2 & T_3 & T_4 & T_5 & T_6 \end{array}\right), \boldsymbol{b} = \left(\begin{array}{cccccccc} B_0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ B_1 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ B_2 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ B_3 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ \hline & S_0 & S_1 & S_2 & S_3 & S_4 & S_5 & S_6 \end{array}\right), \tag{7}$$

and

$$\boldsymbol{c}^t = \left(\begin{array}{cccccccc} C_0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ C_2 & 0 & -1 & 0 & 0 & 1 & -1 & -1 \\ C_1 & -1 & 1 & -1 & -1 & 0 & 0 & 0 \\ C_3 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ \hline & P_0 & P_1 & P_2 & P_3 & P_4 & P_5 & P_6 \end{array}\right) \tag{8}$$

Take matrix $\boldsymbol{c}$ and consider the first row (C_0).

$$\boldsymbol{C}_0 = P_3 + P_5 = (T_3 * S_3) + (T_5 * S_5) = (A_1) * (B_3) + (A_0) * (B_0) \tag{9}$$

and for better effect the last row (C_3).

$$\boldsymbol{C}_3 = P_0 + P_6 = (T_0 * S_0) + (T_6 * S_6) = (A_2 - A_1) * (B_1) + (A_3) * (B_1 + B_3) \tag{10}$$

If we take the matrix $\boldsymbol{c}^t$, the columns are the products of the algorithms, in this case 7. The row explains how to sum these product so that to achieve the correct result. The matrix $\boldsymbol{a}$, each column i represents the left operand of product P_i. The matrix $\boldsymbol{b}$ represent the right operand. Notice, that with the proper matrices we could achieve Equation 4

We shall use only square matrices for simplicity. Thus the key for the full determination of the Strassen algorithm is 2. This is the common factor we use to split both sides (i.e., row and column) for all matrices. If the split factor determines the algorithm with minimum number of products, so

we are going to *play* with 2 as 7 products, 3 as 23 products, 4 as 2x2 and 49 products, 6 = 2x3 and 3x2 as 7*23 products, 9 = 3x3 as 23*23 products, 12=2x2x3, 2x3x2, 3x2x2 as 7*7*23 products.

Intuitively and if we are using the recursive idea as in Equation 5, take the algorithm with factor 6. We can take fist the factor 2, and each of the 7 product, recursively use an algorithm with factor 3 and 23 products. A pure recursive algorithm would compute the operands T_i and S_i first, recursively solve the result P_i and distribute it. The sequence 2 and 3 specifies an algorithm that is different from the sequence 3 and 2. The space requirement is different: literally if we take a 6x6 matrix, if we split the matrices by 2, the temporary space to hold T_i and S_i are two matrices of size 3x3. The recursive call will use temporary space of size 1x1. Ohterwise, if we split by 3, we need a temporary space of size 2x2. The computation order is different and (in double precision) there is a computational difference.

Interestingly, the original problem (6x6x6), which is the complete reference in [1] splits the problem (M, N, K) into $(\frac{M}{6}, \frac{N}{6}\frac{K}{6})$ basic products can be done using non square ones: (2x3x3) and (3x2x3). Thus, we can solve directly using a factor 6 and square matrices but with a proper factorization of $\boldsymbol{a}, \boldsymbol{b}$, and $\boldsymbol{c}$ and the order is important.

In the project there is a complete proof and a few implementations: In practice, if we use a factor 3 and we have $\boldsymbol{a}_3, \boldsymbol{b}_3, \boldsymbol{c}_3$ and we have a factor 2 with $\boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{c}_2$, then the algorithm with factor 6 is completely defined succinctly as $\boldsymbol{a}_3 \otimes \boldsymbol{a}_2, \boldsymbol{b}_3 \otimes \boldsymbol{b}_2, \boldsymbol{c}_3 \otimes \boldsymbol{c}_2$. Where $\otimes$ is the Kronecker product (used also in [1]). This factorization and algorithm generation is really neat: If we have a problem size that can be factorized exactly with the set of fast algorithms, then the fast algorithm require just constant temporary space (last factor squared). We shall show application of this idea in the experimental results.

3 MATRIX FLOW

Take a matrix operation such as $\boldsymbol{C} = \alpha\boldsymbol{A}\boldsymbol{B}$, and describe this computation by means of the matrix partitions by factors: recursively if you like by a factor of 2

$$\begin{pmatrix} \boldsymbol{C}_0 & \boldsymbol{C}_1 \\ \boldsymbol{C}_2 & \boldsymbol{C}_3 \end{pmatrix} = \alpha \begin{pmatrix} \boldsymbol{A}_0 & \boldsymbol{A}_1 \\ \boldsymbol{A}_2 & \boldsymbol{A}_3 \end{pmatrix} * \begin{pmatrix} \boldsymbol{B}_0 & \boldsymbol{B}_1 \\ \boldsymbol{B}_2 & \boldsymbol{B}_3 \end{pmatrix} \tag{11}$$

If we use the definition of matrix multiplication to perform the computation, we can build a sequence of instructions as follows:

```
## Declarations AD partition of A, BD of B, CD of C
decls = [[alphai],      AD ,        BD,          CD ]

###
## Computation as a sequence of assignment statements
## and binary operations.
###
V = []
for i in range(Row=2):
    for j in range(Col=2):
        T = Operation("p0", "*", AD[i*Col] ,BD[j])
        for k in range(1,K=2):
            T1 = Operation("p%d"%k, "*", AD[i*Col+k],BD[k*Col+j])
            T  = Operation('c','+',T,T1)
        R = Operation("s0", '<<',CD[i*Col+j],T)
        V.append(R)
```

Operation is simply an abstraction of the matrix operations $*$ and +, we have only binary computations plus a constant multiplication. Such a format is basic form in BLAS libraries. In practice, the declarations and the instruction sequence define the computation completely. In matrix flow this is a *Graph*. A Graph is an extension of a *function* where there can be multiple inputs and outputs. A Function is an extension of an operation: one output and two operands.

The idea behind a graph is simple: we have clearly spelled out inputs and ouputs, we can build a clear declaration of the matrix partitions partition and constants, we have a list of operations. The computation is summarized by a directed acyclic graph and the schedule by the sequence of instructions.

In the last ten years, the compiler technology for data flows computations and DAGs has had great incentives because it is the foundations artificial intelligence applications for CPU but especially for custom accelerators (among them GPUs).

The main features of a graph at the time of this writing are the following.

(1) The self description of the graph is a readable code. If we create a graph using Equation **??** and the classic algorithm to write the computation operations, the print of the graph is literally

```
(Pdb) print(G1)
CDP[0] << (ADP[0]) * (BDP[0]) + (ADP[1]) * (BDP[2])
CDP[1] << (ADP[0]) * (BDP[1]) + (ADP[1]) * (BDP[3])
CDP[2] << (ADP[2]) * (BDP[0]) + (ADP[3]) * (BDP[2])
CDP[3] << (ADP[2]) * (BDP[1]) + (ADP[3]) * (BDP[3])
```

(2) We can compute the graph by executing the list of operation in sequence. So the graph is a computational data structure.

(3) The self description is valid Python code and it can be compiled and executed.

The Graph is a self contained structure that can be used to execute the algorithm, to compile it, and thus validate easily.

Our goal here is to describe how this could be used to validate correctness and relative performance of any fast algorithms and to show that by customizing the description of the graphs as code we can generate native codes for CPU and GPU that can be easily compiled and validated in the same environment. Part of Matrix Flow but not presented here: Data dependency, scheduling rewrite for single accelerator or multiple accelerators. We want to focus on how we can build efficient codes for GPUs.

3.1 CPU and Python performance and validation

We have a systematic way to retrieve fast algorithms by factors, we can create computational graphs, for a specific problem size, and we can execute them. The first thing it is to get a feel of the performance, like in Figure 1.

There is a lot to unpack. There are two plots, the matrix sizes go from the largest to the smallest so we can highlight the maximum speed up as the first spot in the graph. This is a Thread-ripper CPU with 16 cores. We can afford to give one core to the base line GEMM (and for all fast algorithms) and we can give them 16. The first plot show the performance when we use 1 core. The Base line is steady at 30 GFLOPS. It is clear that Strassen 2 and 2x2 is faster for any problem size 2500 and above. All algorithms with three or more factors are behind and they never break even but the trajectory is promising.

The situation get worse as soon as the GEMM get more efficient: In the second plot no fast algorithm can hold a candle to the base line. The main reason is that the base line does not have a sharp increasing and, then, steady performance. In practice, the additions are constantly bad

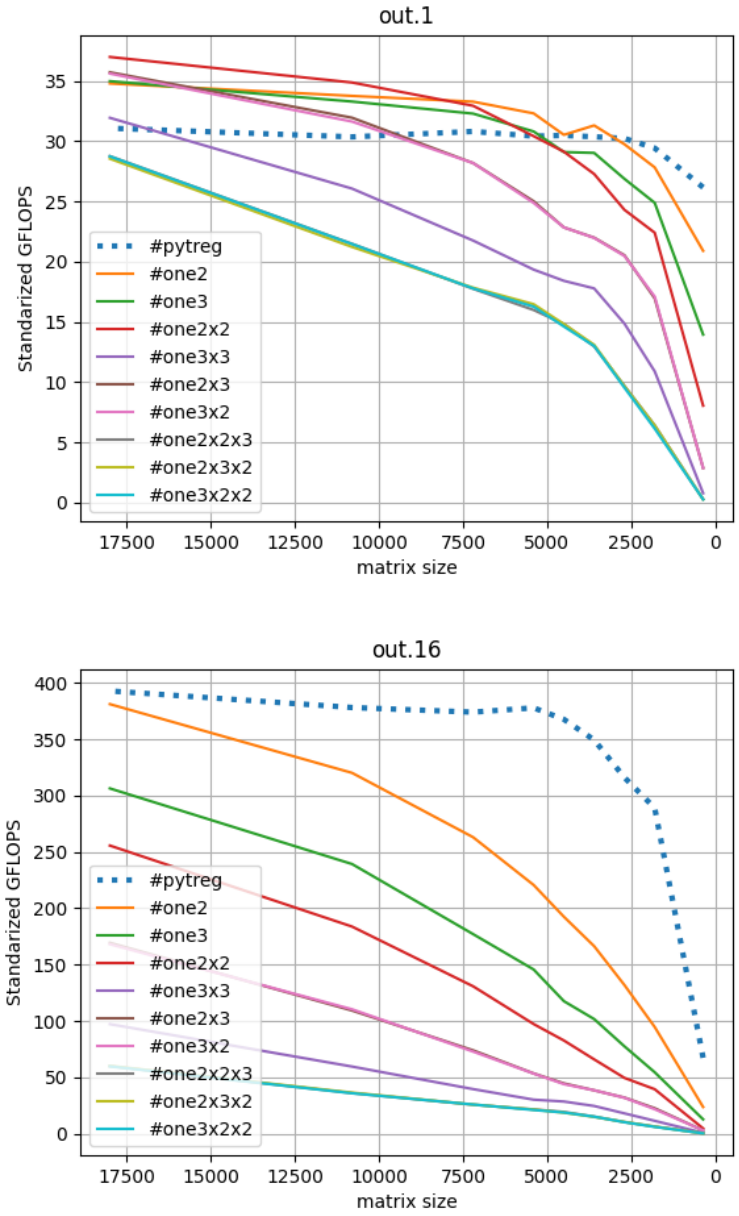

Fig. 1. Performance with 1 CORE and 16 cores from GEMM

performing and the multiplications we save are small and inefficient, we are better off solving a large problem using the standard GEMM.

Of course, the 16 core example is skewed towards the single call GEMM algorithm, which is never slower down by the several function call for each operations.

3.2 GPUs

Matrix Flow can use the same methodology to create C++ code using rocBLAS interface. In practice, we create rocBLAS calls into a single C++ package that we can compile and import in Python. Again, the interface and module idea is not efficient, it is not meant to replace any of the current implementations in C/C++/Fortran. It is just so mach easier to compare and validate the correctness of the computation and report the numerical stability of the code

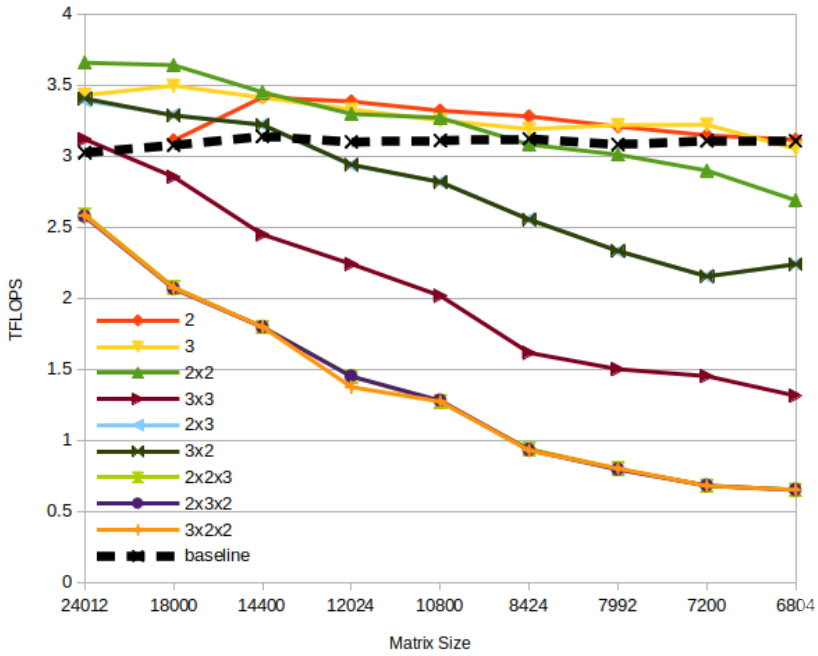

Fig. 2. Performance with VII Radeon 3.5TFLOPS DGEMM

The GPU we use here is a Vega20 known as VII Radeon. The performance is for double precision and the peak performance is 3.5TFLOPS (tera = 10^{12} floating point per second). This GPU has a 16GB of HBM2 main memory with 1TB/s bandwidth, top of the class. In Figure 2, we show the performance again from the largest to the smallest: so we can appreciate better the maximum speed up for the fast algorithms. We report kernel wall clock time that is the computation of only GEMM and GEMA without any CPU device compaction and transmission.

The good news is that the GPU performance is identical to the 1CORE plots (instead of the 16). Once again three factors algorithm are behind but the trajectory is pretty good. The 2 factor algorithms go above the baseline and Strassen 2 and 2x2 are safely in the lead.

Two caveats: the sizes we test are designed so that all algorithms have the proper factors but this does not mean that are suitable for the GEMM kernel. For the last problem size, 24012, the problem size is about 13GB (which was measured at about 80% of teh VRAM available). Due to the temporary space, Strassen with factor 2, does not fit into the device (requiring about extra 3GB).

REFERENCES

[1] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, David Silver, Demis Hassabis, and

Pushmeet Kohli. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature* 610, 7930 (October 2022), 47—53. https://doi.org/10.1038/s41586-022-05172-4

[2] V. STRASSEN. 1969. Gaussian Elimination is not Optimal. *Numer. Math.* 13 (1969), 354–356. http://eudml.org/doc/131927