Comparing the performance of accelerated dense matrix multiplication on FPGA with the software version

Rudrendu Mahindar
Electrical & Computer Engineering
University of Southern California
Los Angeles, CA, USA
mahindar@usc.edu

Introduction

The multiplication of dense matrices is one of the most common and expensive computations in a variety of applications like high-performance computing, graph analytics processing, and machine learning. The operations require a lot of computation and communication of data among the processing elements. The increasing technological gap between computation and memory speeds [1] calls for developing algorithms and techniques to reduce data movement. The initial phase of research in proving I/O lower bounds for specific algorithms, e.g., Matrix-Matrix Multiplication (MMM), dates back to the 80s [8]. The results of the findings were followed by the extension to parallel and distributed machines [9]. Since then, many algorithms to minimize I/O were developed for linear algebra [2, 10, 11], neural networks [12], and general programs that access arrays [13]. The minimization of I/O not only impacts performance but also reduces bandwidth usage in a shared system. MMM is a typical component of larger applications [14, 15], where it co-exists with other algorithms, e.g., memory-bound linear algebra operations such as matrix-vector/vector-vector operations, which benefit from a larger share of the bandwidth but do not require large amounts of computing resources.

The core algorithms of various domains require significant improvement of matrix multiplication by ensuring that the essential computations fit into local memory, which may be managed by the programmer directly, expose high levels of parallelism [4], and carefully consider, memory capacity, and bandwidth. The cost [5] of loading and storing matrix elements dominates the computation time thereby degrading performance. In a single node machine [6], there is a lot of data transfer between main memory and cache due to cache misses incurred during irregular memory accesses, and reuse of elements in the cache by applying a six-level loop structure of block matrix multiplication can reduce the processor-memory traffic, which is the primary concern of dense matrix computations. The practical performance is considerably impacted due to the memory access patterns [7] and cache reuse of the modified tiled version of the algorithm to allow the matrix multiplication operations to be compute-bound instead of memory-bound, as the modern machines have larger cache sizes. Software-based matrix multiplication is slow and often becomes a bottleneck in the overall operation.

FPGA-based [19, 20] design of the matrix multiplier provides a significant speedup in computation time and flexibility as compared to software. Modern FPGAs can accommodate multimillion gates on a single chip. During the last decade, the logic density, functionality, and speed of FPGA have improved considerably. Modern FPGAs are now capable of running at speed beyond 500 MHz. Another important feature of FPGAs is their potential for dynamic reconfiguration which allows reprogramming a part of the device at run time so that resources can be reused through time multiplexing.

Matrix multiplication operation is often performed by parallel processing systems [3] that distribute computations over several processors to achieve significant gains in speedup. The existing realizations of matrix multiplication mainly differ in terms of algorithms or hardware platforms. During the last few years, research efforts towards the acceleration of matrix multiplication using FPGA have been gaining prominence. Parallel array design effectively exploits the inherent parallelism of the matrix multiplication. FPGAs can be used efficiently to implement these fine-grain arrays since they inherently possess the same regular structure. The architecture can effectively utilize the hardware resources inside FPGA like LUTs, fast carry logic, shift registers, flip flops, etc. 2-D array systolic architecture develops energy-efficient designs. For systolic array, the amount of storage per processing element affects the system-wide energy. These designs use the maximum amount of storage per processing element and the minimum number of multipliers to obtain an energy-efficient matrix multiplier. The systolic array-based algorithm [26] is highly suited for acceleration on FPGAs because the partitioning of local matrices enables parallel computations through the mode of access to array elements.

FPGAs are an excellent platform to accurately model performance and I/O for guiding algorithm implementations. In contrast to software implementations, the replacement of cache with explicit on-chip memory, and isolation of the instantiated architecture, yields fully deterministic behavior in the circuit: accessing memory, both on-chip, and off-chip, is always done explicitly, rather than by a cache replacement scheme fixed by the hardware. With ever-increasing diversity in available hardware platforms, and as low-precision arithmetic and exotic data types are becoming key in modern DNN [17] and linear solver [18] applications, extensibility, and flexibility of hardware architectures will be crucial to stay competitive. The models established so far, however, pose a challenge for their applicability on FPGAs because existing high-performance FPGA implementations [19, 20] are implemented in hardware description languages (HDLs),

https://github.com/RudrenduMahindar/EE599-FPGA-Project-Spring-2020

which drastically constrains their maintenance, reuse, generalizability, and portability.

Problem definition

Large sizes of matrices incur a lot of computation and communication costs. Matrix multiplier is the basic computational block of popular matrix-matrix, matrix-vector computations like Computer vision CNN models, graph analytics, PageRank, Solution of linear systems of equations, etc., so there is a broad scope of accelerating dense matrix computations. Acceleration increases the overall performance of systems when acceleration becomes necessary for deploying large-scale applications involving huge volumes of data, high speed, real-time low latency requirements. For faster matrix multiplications in pipelined computations and reduced communication overhead approaches we explore the FPGA based 2-D systolic array architecture (data alignment of this architecture in time shown in fig.2) as a comparison to the different versions of matrix multiplication in software.

PYNQ [23] is an open-source project from Xilinx that makes it easier for developers [16] to use Xilinx platforms. Using the Python language and libraries, designers can exploit the benefits of programmable logic and microprocessors to build more capable and exciting electronic systems. PYNQ eases development by not requiring the knowledge of AISC style design tools or expertise in using HDLs. Python is one of the most popular high-level programming languages (no need of HDL), many python libraries are available for a long-range of applications.

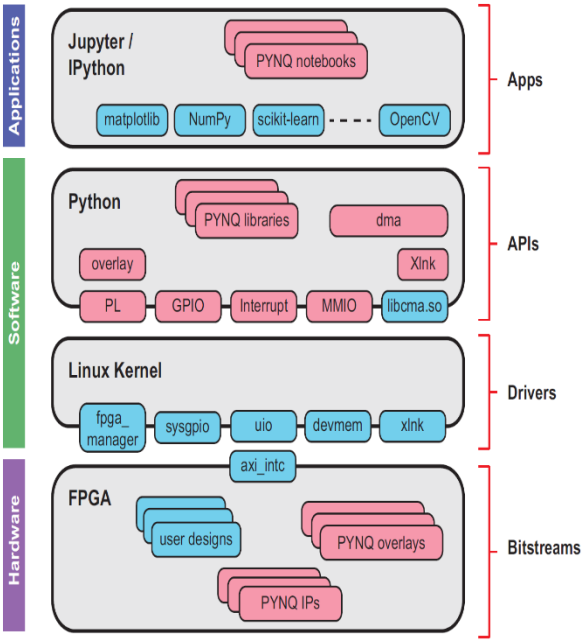

Fig.1: PYNQ- the coolest python framework

Software developers take advantage of the capabilities of Xilinx platforms by programming in python for the PYNQ framework and get an easy software interface like the Jupyter Notebook for rapid prototyping, development of their Zynq, Alveo and AWS-F1 design on FPGA.

Overlays are programmable/configurable FPGA designs that extend user applications from the Processing System into the Programmable Logic and are used to accelerate a software application or to customize the hardware for a particular application. They can be loaded to the FPGA dynamically, as required, just like a software library. PYNQ provides a python interface to allow overlays to be controlled from Python These overlays are created in HDL and wrapped with this PYNQ API. The implementation of hardware overlays can be abstracted from the developers for the faster development of their projects.

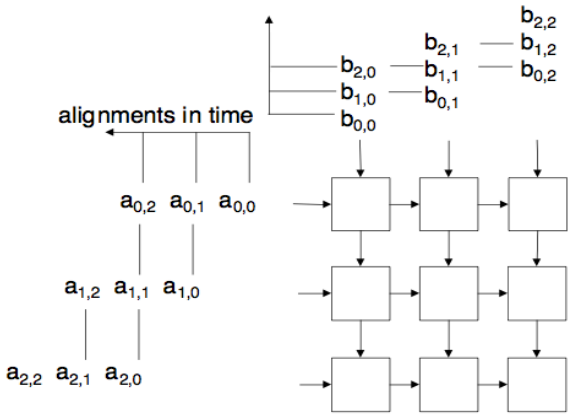

Fig.2: Data alignment in a systolic array [24]

In this work, we have used the PYNQ framework for accelerating dense matrix multiplication on FPGA for 2-D array systolic architecture. The performance of the FPGA accelerated kernel has been compared with the software version of matrix multiplication in the 3-level loop naïve method (shown in fig.3), the tiled version of the block-matrix method (shown in fig.4), and the in-built optimized software libraries.

Algorithm 1 Compute the product of two n x n matrices

```
1:  procedure NAIVE-MATRIX-MULTIPLY(A, B)
2:      n = A.rows
3:      let C be a new n × n matrix
4:      for i = 1 to n do
5:          for j = 1 to n do
6:              c_ij = 0
7:              for k = 1 to n do
8:                  c_ij = c_ij + a_ik · b_kj
9:              end for
10:         end for
11:     end for
12:     return C
13: end procedure
```

Fig.3: Naïve 3-level loop matrix multiplication [25]

```
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where
  b=n / N is called the block size
    for i = 1 to N
      for j = 1 to N
        {read block C(i,j) into fast memory}
        for k = 1 to N
            {read block A(i,k) into fast memory}
            {read block B(k,j) into fast memory}
            C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
        {write block C(i,j) back to slow memory}
```

Fig.4: Block matrix tiled version [21]

Design details

The software version of matrix multiplication has been coded in python3 inside a Jupyter Notebook. Performance has been recorded for a 3-level loop naïve method of matrix multiplication and tiled 6-level loop blocked-method of the same. Besides the above two versions, performance has also been recorded for matrix multiplication by using Numpy, which has optimized matrix multiplication functions. The sizes of matrices are 512x512, 1024x1024, and 2048x2048, and the block sizes for the tiled version are 2, 4, 8, and 16. For interesting observations, the performance of the AWS F1 host has been compared with Intel dual-core i3-6006U CPU@2GHZ, cache 3MB for matrix size 1024x1024.

Xilinx Vitis HLS in CPP for targeting systolic array architecture has been used for accelerating dense matrix multiplication on FPGA. This CPP kernel is then cross-compiled to generate a bitstream which will be run on the target FPGA. In the Jupyter Notebook overlay of PYNQ has been exploited to load bitstream into the FPGA and develop the python application, which will abstract the hardware implementation running on the FPGA. The matrix sizes start from 512x512 and keep on doubling until the allocation fails on the DDR of the AWS F1 instance and the comparison with the software version has been done for the 3 sizes 512x512, 1024x1024, and 2048x2048.

Experimental Setup

We evaluate the different versions of our implementation on AWS EC2 F1 instance, whose setup is shown in fig. 5. The f1.2xlarge has the specifications: (26 ECUs, 8 vCPUs, 2.3 GHz, Intel Xeon E5-2686v4, 122 GiB memory, 1 x 470 GiB Storage Capacity). The F1 instance is equipped with Xilinx Vitis 2019.2 and the FPGA accelerated version has been executed on platform "xilinx_aws-vu9p-f1_shell-v04261818_201920_1". The matrix input and output buffers are contiguous blocks of memory in the DDR allocated by the host processor and then synced to the FPGA card in the configurable region, input buffers of matrices are synced before starting the computation and the output buffer is synced after the computation ends to get the results in the host. On F1 instance Jupyter Notebook in anaconda3 has been used as an interactive tool for using python coding for both the software and the FPGA versions. The HLS CPP code was wrapped in python by using the PYNQ framework to map the code to FPGA hardware in the bitstream using PYNQ overlays.

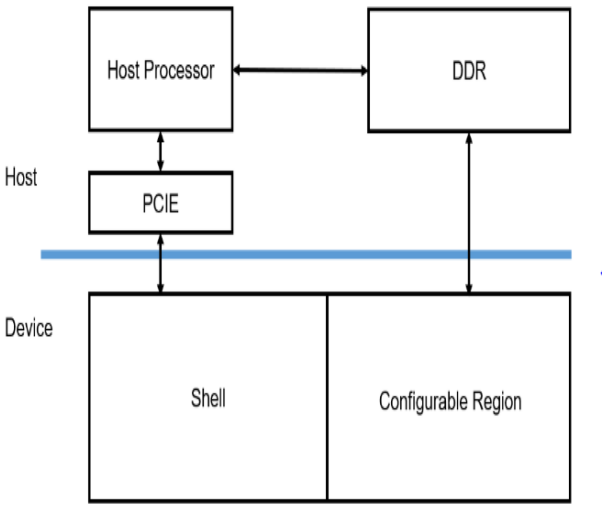

Fig.5: Setup of the AWS EC2 F1 instance [22]

For some comparisons with the performance of the host processor for the software version of matrix multiplication, another processor (P2) Intel dual-core i3-6006U CPU@2GHZ, cache 3MB (this processor has been named as P2 for the following sections in this paper), has been used for the implementation. In this P2 Intel i3 processor, python3 has been used for coding the implementation of software version algorithms.

Analysis of Results

We evaluate the computational performance and communication behavior of our FPGA accelerated version by constructing kernels with varying matrix sizes, doubling sizes until the contiguous memory allocation by the host processor on the DDR fails. Table1 shows the kernel execution time as a break-up of the input buffers sync time, output buffer sync time, and computation time on matrices. The last value of the matrix size column was the allocation failed size. The performance trend of the FPGA accelerated kernel for varying matrix sizes has been shown in fig. 6. The comparison of FPGA kernel time with the naïve 3-level software time has been shown in table 2 and the difference in their figures appears in fig.7. An interesting observation while implementing the software version of matrix

multiplication on F1 instance shows that the tiled 6-level loop version does not show any improvement compared to the performance 3-level naïve method. The data of performance comparison has been displayed in table3 for 6-level loop, 3-level loop, NumPy matmul() methods of matrix multiplication on matrix size 1024x1024, chosen comparison setup for F1 instance against P2 processor.

Table1: Time (in seconds) of FPGA accelerated kernel

Matrix size	Input sync time (sec)	Compute time (sec)	Output sync time (sec)	Total kernel execution time (sec)
512x512	0.00226	0.00334	0.00113	0.00673
1024x1024	0.00853	0.0132	0.00427	0.026
2048x2048	0.0338	0.0525	0.0169	0.1032
4096x4096	0.134	0.210	0.0672	0.4112
8192x8192	0.537	0.839	0.269	1.645
16384x16384	2.15	3.36	1.07	6.58
32768x32768				RuntimeError: Allocate failed

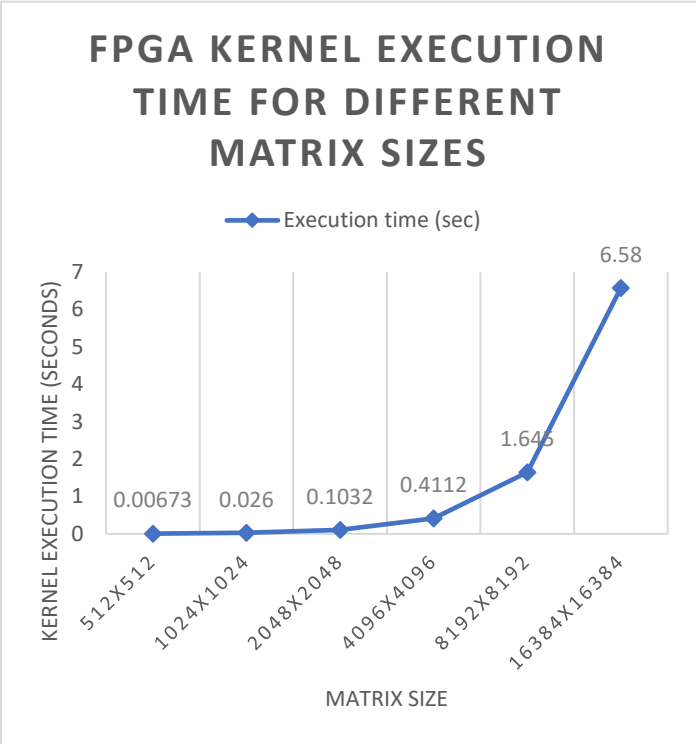

Fig.6: FPGA kernel execution time (in seconds) for different matrix sizes

Table2: FPGA time vs Software 3-level loop time

Matrix size	FPGA kernel execution time (sec)	Naïve 3-level loop execution time (sec)
512x512	0.00673	125
1024x1024	0.026	1064
2048x2048	0.1032	11685
4096x4096	0.4112	140220

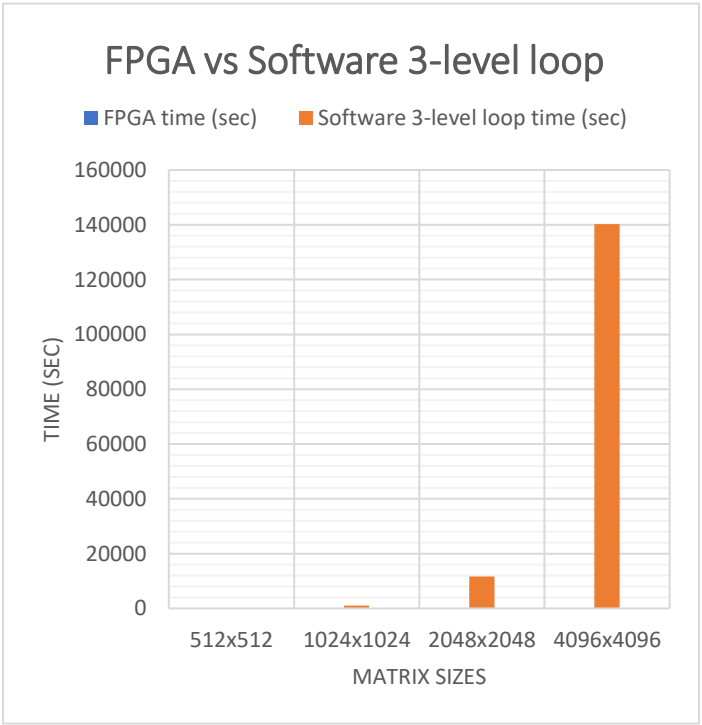

Fig.7: FPGA kernel time vs software 3-level loop time

From table3 we find that block matrix method with different sizes improves performance on P2 processor, the best for block size=8, whereas in F1 instance host processor there is a further degradation of performance with the 6-level loop tiled version. The NumPy matmul() function is significantly faster on both the processors. We hereby move towards the comparison between the fastest software method using python3 on Jupyter Notebook i.e., the Numpy matmul() and the FPGA accelerated version before arriving at the final viewpoint on the performance comparison of the FPGA vs Software versions and this comparison has been shown in table5 and fig.9.

Table3: Comparison of 3-level, 6-level loop performance (in seconds) between F1 and P2 processors for matrix size 1024x1024

Processors for software version	3-level loop (sec)	6-level loop Block size=2 (sec)	6-level loop Block size=4 (sec)	6-level loop Block size=8 (sec)	6-level loop Block size=16 (sec)	Numpy matmul() (sec)
F1 instance host processor	1064	1466	1256	1182	1154	0.368
P2 processor	38077	3410	5121	3111	3648	11.49

We observe at the figures of table3 to find that the 6-level loop block matrix method causes no performance improvement on the host processor of the AWS F1 instance. To dive deeper into the case analysis of block size variance for the tiled version 6-level loop of multiplying matrices on the F1 host processor we raise the block size in powers of 2 starting from block size=2 until the performance figures saturate at block size=128. This performance data has been represented in table4 for matrix sizes 512x512 and 1024x1024 and the behavior of different block sizes is reflected from fig.8.

Table4: Tiled version performance (in seconds) of the 6-level loop on F1 host processor for multiplying matrices with different block sizes

Matrix size	Block size=2	Block size=4	Block size=8	Block size=16	Block size=32	Block size=64	Block size=128	Block size=256
512x512	179	154	145	141	140	139	141	141
1024x1024	1466	1256	1182	1154	1144	1140	1146	1146

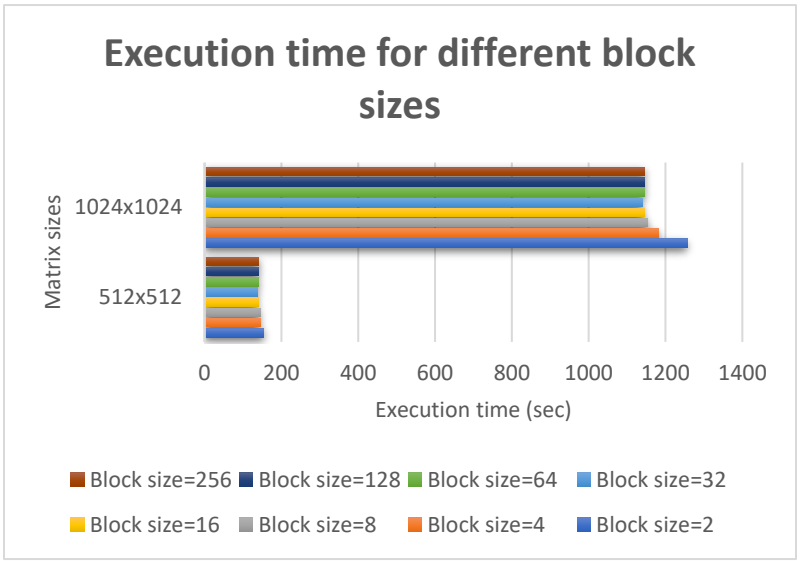

Fig.8: Behaviour of block matrix multiplication for different block sizes on the AWS F1 host processor

Table5: FPGA accelerated kernel time vs the fastest software method time (in seconds)

Matrix sizes	FPGA kernel execution time (sec)	Numpy matmult() time (sec)
512x512	0.00673	0.04804
1024x1024	0.026	0.368365
2048x2048	0.1032	2.9
4096x4096	0.4112	30

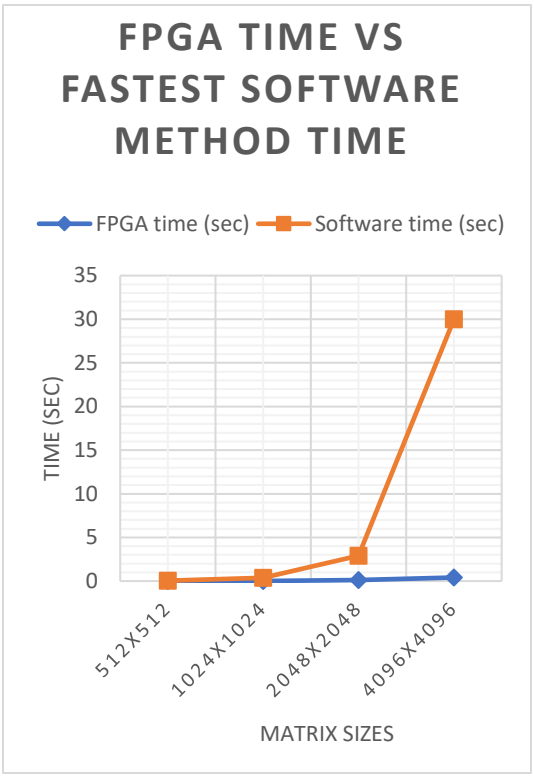

Fig.9: Performance comparison of FPGA accelerated version vs the fastest method in software version

Conclusion

We present a comparative analysis of different matrix multiplication algorithms on different platforms. The analysis uses the accelerated systolic array version on FPGA, and naïve method 3-level loop, block matrix method 6-level loop, and NumPy method on software.

The initial phase of comparison between the software version and the FPGA version on the AWS F1 instance shows that the NumPy method produces the fastest results in the software version but the FPGA accelerated version always beats the fastest software version in speed, starting from being 7 times faster than NumPy for matrix size 512x512, and then the FPGA version continues to speed up faster than NumPy for larger matrix

sizes until being 70 times faster for matrix size 4096x4096 and the FPGA execution kernel has taken under 1 second till 4096x4096.

On contrasting observations on performance improvement by tiled version 6-level loop of multiplying matrices, the best block size=64 for F1 host processor but it was worse in performance than that of the naïve 3-level loop on F1, where block size=64 was 1.07 times slower than the 3-level loop. Performance on the P2 processor improves by using the tiled version, where the best performance for block size=8 was 12 times faster than the 3-level loop on P2.

The NumPy matmul() on python3 shows a noteworthy jump in speed of execution compared to the other software methods used here, as its performance was around 3000 times better than the naïve 3-level loop on both F1 host and P2 processors.

References

[1] W. A. Wulf and S. A. McKee, “Hitting the memory wall: Implications of the obvious,” SIGARCH Comput. Archit. News, vol. 23, no. 1, pp. 20–24, Mar. 1995.

[2] E. Solomonik and J. Demmel, “Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms,” in Euro-Par 2011 Parallel Processing, E. Jeannot, R. Namyst, and J. Roman, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 90–109

[3] Berntsen, Jarle. "Communication efficient matrix multiplication on hypercubes." Parallel computing 12.3 (1989): 335-342.

[4] Rivera, Gabriel, and Chau-Wen Tseng. "A comparison of compiler tiling algorithms." International Conference on Compiler Construction. Springer, Berlin, Heidelberg, 1999.

[5] Johnsson, S. Lennart. "Minimizing the communication time for matrix multiplication on multiprocessors." Parallel Computing 19.11 (1993): 1235-1257.

[6] Johnsson, S. Lennart, Tim Harris, and Kapil K. Mathur. "Matrix multiplication on the Connection Machine." Supercomputing'89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing. IEEE, 1989.

]7] Goto, Kazushige, and Robert van De Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR02-55, Department of Computer Sciences, U. of Texas at Austin, 2002.

[8] H. Jia-Wei and H.-T. Kung, “I/O complexity: The red-blue pebble game,” in Proceedings of the thirteenth annual ACM symposium on Theory of computing. ACM, 1981, pp. 326–333.

[9] D. Irony, S. Toledo, and A. Tiskin, “Communication lower bounds for distributed memory matrix multiplication,” Journal of Parallel and Distributed Computing, vol. 64, no. 9, pp. 1017–1026, 2004.

[10] G. A. Geist and E. Ng, “Task scheduling for parallel sparse cholesky factorization,” Int. J. Parallel Program., vol. 18, no. 4, pp. 291–314, Jul. 1990.

[11] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer, “Communication-avoiding QR decomposition for GPUs,” in Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International. IEEE, 2011, pp. 48–58.

[12] J. Demmel and G. Dinh, “Communication-optimal convolutional neural nets,” arXiv preprint arXiv:1802.06905, 2018.

[13] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick, “Communication lower bounds and optimal algorithms for programs that reference arrays–part 1,” arXiv preprint arXiv:1308.0068, 2013.

[14] V. Vanhoucke, A. Senior, and M. Z. Mao, “Improving the speed of neural networks on CPUs,” in Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop, vol. 1. Citeseer, 2011, p. 4.

[15] M. Del Ben et al., “Enabling simulation at the fifth rung of DFT: Large scale RPA calculations with excellent time to solution,” Comp. Phys. Comm., 2015.

[16] Janßen, Benedikt, Pascal Zimprich, and Michael Hübner. "A dynamic partial reconfigurable overlay concept for PYNQ." 2017 27th International Conference on Field Programmable Logic and Applications (FPL). IEEE, 2017.

[17] T. Ben-Nun and T. Hoefler, “Demystifying parallel and distributed deep learning: An in-depth concurrency analysis,” ACM Computing Surveys (CSUR), vol. 52, no. 4, p. 65, 2019.

[18] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, “Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers,” in Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. IEEE Press, 2018, p. 47.

[19] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, “A customizable matrix multiplication framework for the Intel HARPv2 Xeon+FPGA platform: A deep learning case study,” in Proceedings of the 2018 ACM/SIGDA International Symposium on FieldProgrammable Gate Arrays, ser. FPGA ’18. New York, NY, USA: ACM, 2018, pp. 107–116.

[20] Z. Jovanović and V. Milutinovic, “FPGA accelerator for floating-point matrix multiplication,” IET Computers & Digital Techniques, vol. 6, no. 4, pp. 249–256, 2012.

[21] https://slideplayer.com/slide/5207726/

[22]https://github.com/Xilinx/PYNQ-HelloWorld/blob/master/pynq_helloworld/notebooks/pcie/resizer_device.ipynb

[23] http://www.pynq.io/

[24]http://ashanpeiris.blogspot.com/2015/08/digital-design-of-systolic-array.html

[25]https://www.kkhaydarov.com/matrix-multiplication-algorithms/

[26] Elfimova, L. D., and Yu V. Kapitonova. "A fast algorithm for matrix multiplication and its efficient realization on systolic arrays." Cybernetics and Systems Analysis 37.1 (2001): 109-121.