An Open-source HLS Fully Parameterizable Matrix Multiplication Library for AMD FPGAs

Angelos Athanasiadis
School of Electrical and Computer Engineering,
Aristotle University of Thessaloniki,
54124, Thessaloniki, Greece
angelathan@ece.auth.gr

Nikolaos Tampouratzis
Department of Industrial Engineering and Management,
International Hellenic University,
57400, Sindos, Greece
ntampouratzis@ihu.gr

Ioannis Papaefstathiou
School of Electrical and Computer Engineering,
Aristotle University of Thessaloniki,
54124, Thessaloniki, Greece
ygp@ece.auth.gr

Abstract—**One common characteristic of High-Performance Computing (HPC) and Cyber-Physical Systems (CPS) is their need for heterogeneous energy-efficient solutions. In this work we present a library for FPGA-accelerated dense matrix multiplication which is flexible, open-source, written in purely synthesizable C and has no dependencies on the actual hardware implementation tools. Our library is designed so as to support arbitrary array sizes and accuracy, making it a versatile and adaptable solution that meets the diverse computational requirements of applications all the way from CPS to HPC. Our approach provides an adaptable solution that efficiently exposes the flexibility and performance of the FPGAs to both novice and expert developers which is not the case with the black-box libraries provided by the FPGA manufacturers. Our approach has been evaluated in a number of state-of-the-art AMD FPGAs; the end results demonstrate that the presented implementations can achieve 9x, 34x and 3x gains, in terms of energy efficiency, when compared with embedded, high-end CPUs and GPUs respectively. Moreover, our solution matches or slightly outperforms the most advanced similar FPGA-tailored approach while also being much more flexible and designer-friendly while also library-independent.**

Keywords-High-Performance; Computing; Neural Networks; Matrix Multiplication; AMD FPGA; Vitis

I. INTRODUCTION

In recent years, the complete computing continuum has undergone a significant transformation, fueled by the growing demand for computing power in a range of domains, such as scientific simulations, machine learning, and data analytics. At the same time FPGAs have become a captivating solution to tackle the increasing demand for computing power at a relatively low power envelope, thanks to their parallel processing capabilities and flexibility.

One very widely used function in numerous CPS and HPC applications is the multiplication of dense matrices (e.g. crucial for linear algebra computations). The efficient execution of dense matrix multiplication on FPGAs has been the subject of extensive research [1], [2] because it has a direct significant impact on the overall computational throughput and energy efficiency of FPGA-based HPC and embedded systems.

Although several optimization techniques for both CPU and GPU architectures have been presented [3], [4], a comparable set of guidelines and principles for code optimizations in High-Level Synthesis (HLS) design flows has yet to be established. Furthermore, due to the low clock frequency, lack of cache, and fine-grained configurability of FPGAs, naive HLS implementations, very often, have low performance and relatively high-power consumption and require significant transformations so as to surpass the multi/many core CPUs and GPUs in terms of speed and/or energy efficiency.

This paper presents a generic open-source solution for HLS design flows allowing for the implementation of high-performance dense matrix multiplication on modern AMD FPGAs. Specifically, the contribution of this paper can be summarized as follows:

- An **open-source library**[1] designed to accelerate dense matrix multiplication of any size and datatype by extracting parallelism in two dimensions. This purely Synthesizable C library provides very high configurability and flexibility in order to take full advantage of the resources of modern AMD FPGAs **without any dependency** on the hardware implementation tool (e.g. version of the tools) or any **external library**.
- An innovative flow that enables designers to easily develop FPGA-accelerated applications, that involve matrix handling, using the presented fundamental structures minimizing the development and verification time while achieving high performance and energy efficiency.
- The effectiveness and performance of this library have been rigorously and comprehensively evaluated when implemented on both small and high-end FPGAs and it has been proved that our solution outperforms CPUs, GPUs and even relevant FPGA-tailored approaches.

The presented library will be part of a larger library for efficient matrix handling (which is under development) while it has already been used in a full-precision Convolutional Neural Network (CNN) acceleration framework implemented in multiple FPGAs.

II. RELATED WORK

Ahmad and Pasha [2] investigated several optimization techniques for hardware-accelerated general matrix multiplication on FPGAs, with a specific focus on CNNs. In contrast to [2], our approach provides a comprehensive library

[1] https://github.com/angelosathanasiadis/Gemm-HLS-Fully-Parametrizable

that empowers users to customize FPGA-based matrix multiplication to their unique computational requirements, irrelevant of the application domain. Although Haghi et al. [5] present a reconfigurable FPGA assistant for in-network computations, with an accompanying case study on distributed matrix multiplication, their study focuses on reconfigurable compute-in-the-network FPGA assistance for collective support. Our work is orthogonal to this since it mainly aims at taking full advantage of the hardware resources of a single FPGA while providing adaptability for i) implementation in different FPGA devices and ii) easy integration with design flows which focus on distributed FPGAs.

De Fine Licht et al. [1], [6] presented their research results on transformations of HLS code for HPC and flexible communication respectively. However, their approach has certain limitations : i) they are dependent on the HW implementation tool (e.g. they can be used only up to AMD Vitis 2021.1[2]), which limits their use and ii) they rely on a hardware library developed by the authors, tailored to the FPGA implementation tool, which introduces a certain tool dependency and thus undermining the adaptability and scalability of their approach, while they do not exploit state-of-the-art HBM2 memories. In comparison to those studies, we present a methodology that is independent of specific hardware tools and libraries and is implemented purely in C with HLS pragmas. We, thus, establish a more robust and sustainable solution for enhancing computational performance in HLS designed FPGA systems.

Moving to the GPUs area, several studies have explored the use of cuBLAS for efficient matrix multiplication on GPUs, as well as optimizations for power efficiency and fault tolerance [3]. For AMD GPUs, rocBLAS high-performance library has been developed for matrix operations exploiting the specific architectural features of AMD hardware. Authors in [4] present a robust framework built on rocBLAS that efficiently handles batched matrix multiplications, even with unbalanced input sizes, showcasing rocBLAS's flexibility and efficiency. While our performance metrics indicate slower computation times compared to GPU-based solutions such as cuBLAS and rocBLAS, our approach demonstrates significantly higher energy efficiency. This makes our approach particularly advantageous in power-constrained environments, where energy consumption is critical.

III. Implementation

A. Reference Implementation

In order to demonstrate our optimization flow, we start with a basic reference matrix multiplication implementation, in which we adopt a simple effective HLS pipeline approach aiming at maximizing the computational efficiency on the targeted FPGAs. The algorithm comprises of nested loops that meticulously traverse matrices A, B, and C, and calculate the dot product of a row from matrix A with a corresponding column from matrix B. The key element is the HLS pipeline directive, which allows the execution of one multiplication and one addition in a single clock cycle achieving a relatively high throughput and performance. However, the main drawback of this method is the excessive memory accesses to external DRAMs & HBMs (i.e. in total M*N*K accesses in all arrays) preventing further parallelization.

B. Vectorization

In traditional FPGA matrix multiplication implementations, the frequent accesses to external DRAMs have been a serious performance bottleneck since each DRAM access adds latency and consumes valuable energy. In our approach we propose the use of 512bit vector elements so as to dramatically reduce the number of DRAM accesses. This reduction is achieved by aggregating multiple data points into a single, wide uint512_t element, effectively consolidating data transfers and parallel computations. The use of uint512_t elements results in improved FPGA Block RAM (BRAM) utilization since by using wider data elements we increase the utilization of each BRAM memory. In addition, since modern FPGAs are connected to multiple memory banks with dedicated channels (e.g., multiple DRAM modules or HBM lanes), we split data transfers into a parameterizable number of memory banks/lanes to increase the effective external memory bandwidth.

C. Innovative Optimization flow

Moreover, as illustrated in Figure 1, we utilize the internal BRAMs in conjunction with HLS streams, to optimize also the on-chip memory access patterns and further increase the computational efficiency. We allocate BRAM resources for matrix BRAM_B and use HLS streams for matrices A and C. Stream_A and Stream_C_in are used to read matrices A and C respectively from external memories (DRAMs/HBMs) and Stream_C_out to transfer the data from the internal computation modules back to the external memories.

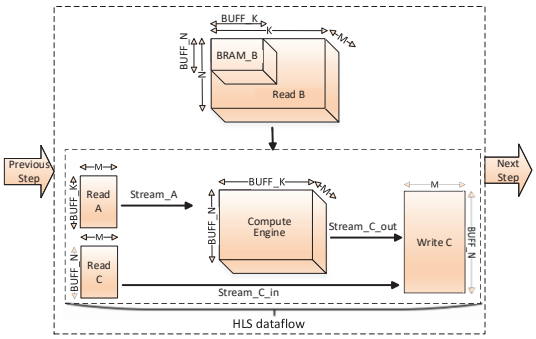

Figure 1: Architecture of Innovative Optimization flow

The aforementioned technique leads to a significant decrease in the effective latency associated with off-chip memory retrieval, thereby diminishing idle time caused by external memory latency; Listing 1 presents the optimized external memories total accesses which are at least 3 orders of magnitude less than the reference implementation. In addition, streams enable a continuous flow of data between processing elements, without the need for intermediary storage in BRAMs, minimizing the latency and resource overhead. By leveraging streams, data are transferred directly between producer and consumer processes, facilitating pipelined execution and

[2] https://github.com/spcl/gemm_hls/issues/25

enhancing parallelism. This direct transfer mechanism reduces the need for on-chip memory for temporary storage, leading to more efficient resource utilization and higher throughput. Furthermore, streams can handle variable data rates more effectively ensuring that processes are not overwhelmed by the rate of data production.

Our optimization approach is based on four (4) fully configurable parameters: buffer sizes BUFF_K and BUFF_N, which are used for the on-chip memory utilization for arrays B and C, as well as loop unrolling parameters UNROLL_N and UNROLL_K, which show the level of unroll that will happen in each dimension. These parameters enable us to customize the architecture to fully utilize any FPGA architecture/size and memory technology/topology. Through the careful selection of buffer sizes and unrolling factors, we can enhance memory access patterns and computational parallelism, achieving an optimal balance between resource usage, in case there are additional modules that are also placed on the same device, and throughput.

```
initialize BRAM_B[BUFF_K][BUFF_N]
initialize BRAM_C[BUFF_N]

for ex_k from 0 to K with step BUFF_K:
  for ex_n from 0 to N_512 with step BUFF_N:
    // Read B - K*N/VECTOR Accesses
    for k from 0 to BUFF_K, n from 0 to BUFF_N:
      BRAM_B[k][n] = DRAM_B512[];
#pragma HLS DATAFLOW
    // Read A - M*K*N/(VECTOR*BUFF_N) Accesses
    for m from 0 to M, k from 0 to BUFF_K/VECTOR:
        stream_A << DRAM_A512[]
    // Read C_in - M*K*N/(VECTOR*BUFF_K) Accesses
    for m from 0 to M, n from 0 to BUFF_N:
        stream_C_in << DRAM_C512[]
    // Main Loop
    for m from 0 to M:
      for k from 0 to BUFF_K/VECTOR:
      #pragma HLS pipeline
      //Perform Efficient Parallel MM Computation
      //UNROLL_N*UNROLL_K*VECTOR elem. in 1 cycle
      for n from 0 to BUFF_N:
        stream_C_out << BRAM_C[n]
    // Write C
    for m from 0 to M, n from 0 to BUFF_N:
        DRAM_C512[] << stream_C_in + stream_C_out
```

Listing 1: Innovative Memory Access

To provide further clarification, as illustrated in Figure 2, Stream_A is read in uint512_t quantities in order to reduce latency even further. After that, Stream_A and BRAM_B are fed to the computational part in order to calculate the outcomes. The outcomes for the whole BUFF_K dimension are stored in BRAM_C array and then streamed through Stream_C_out in WriteC-sized blocks.

Moreover, we strategically utilize the #pragma HLS pipeline in the matrix multiplication computation tile. This directive plays a critical role in enabling fully pipelined processing for the entire tile. In terms of the implemented Synthesizable C code, each iteration of the loops becomes a distinct processing step executed within a single clock cycle. In that respect we reduce pipeline stalls, guaranteeing an uninterrupted stream of data to the multiplication and addition hardware and thus achieving efficient parallel fully-parameterizable processing of **UNROLL_N*UNROLL_K*VECTOR elements in a single (1) clock cycle**. Finally, our library can further parallelize the kernel operations by allocating matrices among the different Sliced Logic Regions (SLRs) of the recent AMD Alveo devices achieving even better performance and energy efficiency.

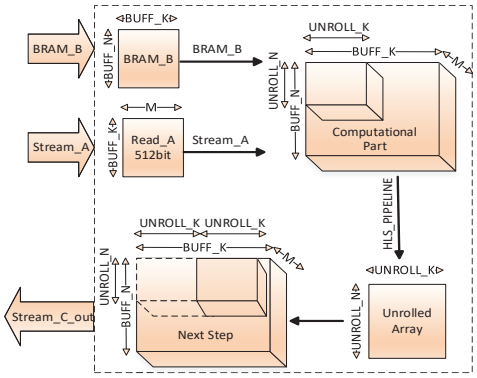

Figure 2: Architecture of Compute Engine

IV. EVALUATION

Table 1: Results with multiple matrix dimensions

M	K	N	U55 [GFLOPS]	U55 [Exec.]	KR260 [GFLOPS]	KR260 [Exec.]
512	1024	2048	207	0.0104s	24	0.89s
1024	2048	4096	215	0.08s	22.6	0.76s
1024	4096	2048	219	0.078s	22.6	0.76s
2048	2048	2048	215	0.08s	22.3	0.77s
2048	4096	16384	229	1.2s	22	12.5s
2048	16384	4096	249	1.1s	22	6.25s
4096	4096	4096	229	0.6s	22	12.5s

We have evaluated the efficiency of our open-source, fully parameterizable, purely-C library for dense matrix multiplication when implemented in numerous FPGA boards. In order to comprehensively evaluate the efficacy and robustness of our proposed matrix multiplication approach, multiple experimental runs were executed across varying dimensions as presented in Table 1. The experimental results demonstrate that we can achieve high performance in diverse dimensional configurations which highlights the wide applicability of our approach. The maximum performance achieved is 249 GFLOPS for matrix sized of M=2048, K=16284 and N=4096 on a high-end FPGA (AMD Alveo U55C) and 24 GFLOPS for matrix sized of M=512, K=1024 and N=2048 on an embedded one (Kria KR260), while the performance is also relatively high in smaller matrix sizes, in contrast to the GPU implementations.

To demonstrate further the effectiveness of the presented approach we compare the non-optimized reference model, the fully optimized model on both a high-end FPGA (AMD Alveo U55C) and an embedded one (Kria KR260), the parallel execution of matrix multiplication using OpenMP in a multicore CPU and using CUDA on an NVIDIA T4 GPU. In all experiments the array dimensions which are M=2048, K=4096, and N=16384 are selected so, as to be different from our optimal configuration, demonstrating also the flexibility of our approach. Furthermore, the results obtained from numerous experiments with different dimensions are fully inline with those presented in Figure 3. As illustrated in Figure 3, our implementation in the embedded FPGA is two orders of magnitude faster than the reference implementation and 9x times faster compared to the fully parallelized algorithm executed on an embedded ARM 4-core CPU (Cortex-A53); those numbers include all the memory accesses and the external memory technologies and topologies are exactly the same in both cases. Similarly, our fully optimized approach when implemented on the Alveo U55C board is appro-

Table 2: Comparison to previous FPGA implementations

	Device	Logic Util. BRAM	Logic Util. DSPs	Perf. FP32 [GFLOPS]	Perf. FP64 [GFLOPS]	Power Effic. [GFLOPS/W]	Tool/Library Independency	Open Source
D'Hollander [7]	Zynq 7000	32%	99%	5	-	-	×	×
Guan [8]	Stratix V	67%	17%	100	-	2.92	×	×
gemm_hls [6]	Alveo U200	58%	44%	211	74	8.49	×	✓
This work	Alveo U55C	56%	44%	229	80	9.02	✓	✓
This work	Kria KR260	94%	60%	22	10.1	2.44	✓	✓

ximately three orders of magnitude faster than the reference implementation and 10x times faster compared to the fully parallelized algorithm executed on an Intel Xeon E5-2620 v4 (8 cores). In order to further compare the overall efficiency of the presented approach with that triggered by the software implementation, the GFLOPS/Watt for each implementation were also measured. Based on our measurements we achieve 34x and 9x higher energy efficiency than the best CPU parallel implementation in an Alveo U55C and a Kria KR260 respectively. Moreover, our design, when implemented on the Alveo, is 3x more power efficient than the CUDA implementation on an NVIDIA T4 GPU which is also implemented on a better CMOS technology (12nm for T4 vs 16nm for U55c).

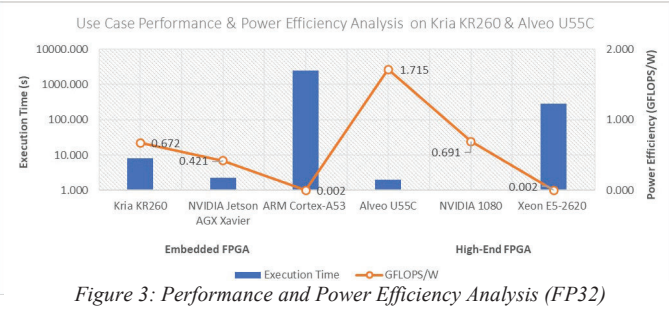

Figure 3: Performance and Power Efficiency Analysis (FP32)

Table 2 presents the comparison of our approach with other FPGA-tailored similar systems. From those, the only open source widely used library for dense Matrix Multiplication is the one in [6] so we tried to implement it in our reference modern FPGAs. However, even though it is an open-source library, it has rather limited applicability because it requires AMD Vitis v2021.1 or older as also referred in Section II. As a result, we implemented it in the largest FPGA supported by this version of the tool we had available (i.e. Alveo U200). In addition, [6] utilizes a specialized hardware library, which also makes it less flexible, than our purely C-based approach. As shown in the table, our design achieves 9.02 GFLOPS/Watt, while gemm_hls' implementation triggers 8.49 GFLOPS/Watt for the same utilization percentage (229 GFLOPS compared to 211 GFLOPS from gemm_hls).[3]

More importantly, our methodology has been developed with a focus on simplicity and wide compatibility, eliminating the need for any specific external libraries and HW implementation tools. The designer-friendliness and the flexibility of our approach enables even non-advanced designers to seamlessly develop matrix-multiplication modules, probably within broader systems (e.g. CNNs or Deep Neural Networks).

V. Conclusions and Future Work

This paper presents an open-source, FPGA-tailored library for dense matrix multiplication that is implemented in purely synthesizable C, thus providing very high flexibility and adaptability. The easily customizable parameters allow users to maximize resource utilization and performance and/or energy efficiency for any given FPGA device from low resources to high-end ones. Our evaluation demonstrates that our library outperforms both embedded and high-end multi-threaded CPUs and GPUs. The pioneering nature of the tool is highlighted by the absence of a similar open-source solution, positioning it as a valuable resource for those looking for easily programmed, yet high-performing FPGA acceleration.

VI. Acknowledgements

This work is supported by the Key Digital Technologies Joint Undertaking (KDT-JU) under the powers delegated by the European Commission and its members, including the top-up funding by National Authorities in the context of the REBECCA (Reconfigurable Heterogeneous Highly Parallel Processing Platform for safe and secure AI) project (grant agreement #101097224).

VII. References

[1] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," IEEE Trans. Parallel Distrib. Syst., vol. 32, no. 5, pp. 1014–1029, May 2021.

[2] A. Ahmad and M. Pasha, "Optimizing hardware accelerated general matrix-matrix multiplication for cnn on fpgas," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 67, pp. 1–1, 2020.

[3] S. Wu, Y. Zhai, J. Liu, J. Huang, Z. Jian, B. Wong, and Z. Chen, "Anatomy of a high-performance semi-automatic fine-tuned tolerance on gpus," in Proceedings of the 27th International Conference on Supercomputing, ser. ICS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 360–372.

[4] R. Wang, Z. Yang, H. Xu, and L. Lu, "A high-performance batched matrix multiplication framework for gpus under unbalanced input distribution," The Journal of Supercomputing, vol. 78, no. 2, p. 1741–1758, Jun 2021.

[5] P. Haghi, A. Guo, T. Geng, J. Broaddus, D. Schafer, A. Skjellum, and M. Herbordt, "A reconfigurable compute-in-the-network fpga assistant for high-level collective support with distributed matrix multiply: case study," IEEE Conference on Field Programmable Technology.

[6] J. de Fine Licht, G. Kwasniewski, and T. Hoefler, "Flexible communication avoiding matrix multiplication on fpga with high-level synthesis," ser. FPGA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 244–254.

[7] E. H. D'Hollander, "High-level synthesis optimization for blocked floating-point matrix multiplication," SIGARCH Comput. Archit. News, vol. 44, no. 4, p. 74–79, Jan 2017.

[8] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates," 04 2017, pp. 152–159.

[3] It is very difficult to compare the actual resources in each FPGA since AMD is listing differently the resources for the Alveo U200 and the U55c boards (i.e. CLBs and Registers vs System Logic Cells).