

Supporting Information: Extending a moldable computer architecture to accelerate DL inference on FPGA

Mirko Mariotti^{a,b}, Giulio Bianchini^{a,b}, Igor Neri^a, Daniele Spiga^b, Diego Ciangottini^b, Lorian Storchi^{c,b}

^a*Dipartimento di Fisica e Geologia, Università degli Studi di Perugia, Via Pascoli, Perugia, 06123, PG, Italy*

^b*INFN sezione di Perugia, Via Pascoli, Perugia, 06123, PG, Italy*

^c*Dipartimento di Farmacia, Università degli Studi G. D'Annunzio, Via dei Vestini, Chieti, 66100, CH, Italy*

1. Analysis of latency as a function of the exponential exponent

One key strength of our implementation lies in its extensive parameter customization both in terms of hardware and software. For instance, an example of software customization, users have the flexibility to modify the number of terms used in the exponential function to suit their specific needs, accordingly to the following equation:

$$e^x = \sum_{l=0}^K \frac{x^l}{l!} \quad (\text{S1})$$

This is Taylor series expansion for the exponential function around $x=0$ (Maclaurin series) that converge to e^x when $K \rightarrow \infty$. As a result, it is expected that there will be a trade-off between reduced accuracy and improved latency (i.e., see for instance results reported in Table S1). Similarly, users can also customize the hardware by directly modifying the data type used to represent the numerical values. The template-based approach combined with the use of the `bmnumbers` package allows to easily change the data type.

Email addresses: `mirko.mariotti@unipg.it` (Mirko Mariotti), `giulio.bianchini@pg.infn.it` (Giulio Bianchini), `loriano@storchi.org` (Lorian Storchi)

To perform the testing of the proposed model we used a Digilent Zedboard (Soc: Xilinx Zynq-7000 [1]) running a custom Linux image with PYNQ 2.6 [2] which made it possible to interact with the IP core of the BM. The evaluation metrics of the proposed solution focuses on two aspects: inference speed, or latency, and resource utilization, as well as an evaluation of the numerical precision of the results respect to the values computed using a standard CPU based code.

Since data transfer from the ARM processor to the FPGA is carried out using the AXI4-Lite Memory-Mapped protocol, a dedicated computing unit called benchcore within the BM ecosystem was employed to accurately measure inference speed. This setup ensures that only the time required by the implemented IP core to perform classification is considered, excluding any protocol or communication overhead. It is a component with two terminals that can be inserted in any part of a BM. It returns the exact count of clock cycles for a signal to travel from one terminal to the other. Knowing the cycles and the clock frequency of the FPGA, it is subsequently easy to trace the time in seconds as follow:

$$t = \frac{c}{f * 10^6} \quad (\text{S2})$$

where f is the FPGA clock frequency in MHz, and c is the number of clock cycles needed to complete the inference operation. Clearly, as reported in Figure S1, one should expect to perform the inference on the full test set (i.e., 275 entries) the number of clock cycles needed have a intrinsic variability. Indeed, the interactions among the terminals of the benchcore are not completely synchronous, thus the number of clock cycles needed to perform the inference on a single entry can vary according some distribution.

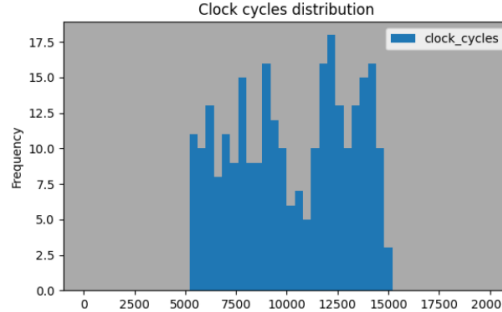


Figure S1: Clock cycles distribution

1.1. Using standard IEEE 32-bit floating point numbers

In Table S1 we are reporting the results obtained using the standard IEEE 32-bit floating point numbers using the model depicted in Figure 1 of the main text. Specifically we are reporting: K representing the number of terms used in the exponential function (see equation S1); the average latency in microseconds computed from the mean clock cycles and the board’s frequency; the mean clock cycles used to perform a classification; the accuracy percentage compared to the original starting model (100% means there is no loss of accuracy between our inference on the FPGA and the CPU/software-based one); the average error on the probabilities for class 0 and the average error on the probabilities for class 1.

K	Latency (μ s)	Clock Cycles	Acc %	Avg. Err. p0	Avg. Err. p1
20	$(1.03 \pm 0.03) \cdot 10^2$	$(10.27 \pm 0.29) \cdot 10^3$	100	$1.65 \cdot 10^{-7}$	$1.67 \cdot 10^{-7}$
16	$(0.79 \pm 0.02) \cdot 10^2$	$(7.95 \pm 2.11) \cdot 10^3$	100	$1.65 \cdot 10^{-7}$	$1.67 \cdot 10^{-7}$
13	$(0.63 \pm 0.02) \cdot 10^2$	$(6.32 \pm 1.67) \cdot 10^3$	100	$1.65 \cdot 10^{-7}$	$1.67 \cdot 10^{-7}$
10	$(0.48 \pm 0.01) \cdot 10^2$	$(4.76 \pm 1.23) \cdot 10^3$	100	$1.62 \cdot 10^{-7}$	$1.66 \cdot 10^{-7}$
8	$(0.39 \pm 0.01) \cdot 10^2$	$(3.91 \pm 1.02) \cdot 10^3$	100	$6.56 \cdot 10^{-5}$	$6.61 \cdot 10^{-5}$
5	$(0.29 \pm 0.01) \cdot 10^2$	$(2.91 \pm 0.74) \cdot 10^3$	100	$3.10 \cdot 10^{-5}$	$3.12 \cdot 10^{-5}$
3	$(0.17 \pm 0.04) \cdot 10^2$	$(1.75 \pm 0.40) \cdot 10^3$	100	$5.32 \cdot 10^{-3}$	$5.31 \cdot 10^{-3}$
2	$(0.13 \pm 0.03) \cdot 10^2$	$(1.31 \pm 0.27) \cdot 10^3$	100	$1.93 \cdot 10^{-3}$	$1.93 \cdot 10^{-3}$
1	$(0.09 \pm 0.02) \cdot 10^2$	$(0.92 \pm 0.17) \cdot 10^3$	100	$9.92 \cdot 10^{-2}$	$9.90 \cdot 10^{-2}$

Table S1: Latency, clock cycles, accuracy and the average error in probabilities per inference are analyzed as the exponent of the exponential K varies. The results were obtained using the ZedBoard, with the latency column representing the mean latency (in μ s) for classifying 275 samples. Clock cycles for each prediction are measured using the benchcore system.

From table S1 one can easily note that as K (see equation S1) decreases the latency follows suit, this is clearly expected. Concurrently, there is a noticeable increase in probability errors, although, the K decrement does not impact the overall accuracy. As previously mentioned, the latency values were calculated by counting clock cycles for inference using the benchcore. However, when starting from high-level language, it is necessary to consider the overhead due to data transfer of the full test set, which has been estimated

to be approximately $1.03 \pm 0.03 \cdot 10^2 \mu s$. Thus, the best overall inference time we can achieve, when K is equal to one, is $0.09 \pm 0.02 \cdot 10^2$ to perform the inference of 275 entries (i.e., the full test set).

1.2. Using the FloPoCo framework

One of the main features of FloPoCo [3] is the representation of floating-point numbers using an arbitrary precision, i.e., number of bits, but always considering the NN model as reported in Figure 1 of the main text. In fact, it is possible to arbitrarily choose the number of bits assigned to both the exponent and the mantissa. This leads to great freedom in selecting numerical precision, allowing us to perform various tests by varying the number of bits used to represent a floating-point number. Consistent with the tests conducted earlier, we will analyze the implementation using the same metrics while varying the exponent of the exponential function (i.e., the value of K in equation S1).

K	Latency (μs)	Clock cycles	Acc. %
20	$(0.31 \pm 0.06) \cdot 10^2$	$(3.11 \pm 0.56) \cdot 10^3$	100
16	$(0.26 \pm 0.05) \cdot 10^2$	$(2.63 \pm 0.46) \cdot 10^3$	100
13	$(0.22 \pm 0.04) \cdot 10^2$	$(2.21 \pm 0.39) \cdot 10^3$	100
10	$(0.17 \pm 0.03) \cdot 10^2$	$(1.68 \pm 0.30) \cdot 10^3$	100
8	$(0.14 \pm 0.02) \cdot 10^2$	$(1.44 \pm 0.24) \cdot 10^3$	100
5	$(0.10 \pm 0.02) \cdot 10^2$	$(0.99 \pm 0.15) \cdot 10^3$	100
3	$(0.07 \pm 0.01) \cdot 10^2$	$(0.68 \pm 0.10) \cdot 10^3$	100
2	$(0.05 \pm 0.01) \cdot 10^2$	$(0.55 \pm 0.07) \cdot 10^3$	100
1	$(0.04 \pm 0.01) \cdot 10^2$	$(0.39 \pm 0.05) \cdot 10^3$	100

Table S2: FloPoCo 32-bits, 7 bits for the exponent and 22 bits for the mantissa.

K	Latency (μs)	Clock cycles	Acc. %
20	$(0.30 \pm 0.05) \cdot 10^2$	$(2.97 \pm 0.51) \cdot 10^3$	100
16	$(0.24 \pm 0.04) \cdot 10^2$	$(2.37 \pm 0.44) \cdot 10^3$	100
13	$(0.20 \pm 0.04) \cdot 10^2$	$(2.02 \pm 0.36) \cdot 10^3$	100
10	$(0.16 \pm 0.03) \cdot 10^2$	$(1.60 \pm 0.27) \cdot 10^3$	100
8	$(0.13 \pm 0.02) \cdot 10^2$	$(1.33 \pm 0.23) \cdot 10^3$	100
5	$(0.09 \pm 0.02) \cdot 10^2$	$(0.92 \pm 0.15) \cdot 10^3$	100
3	$(0.06 \pm 0.01) \cdot 10^2$	$(0.64 \pm 0.10) \cdot 10^3$	100
2	$(0.05 \pm 0.01) \cdot 10^2$	$(0.50 \pm 0.07) \cdot 10^3$	100
1	$(0.04 \pm 0.01) \cdot 10^2$	$(0.37 \pm 0.04) \cdot 10^3$	99.63

Table S3: FloPoCo 19-bits, 5 bits for the exponent and 11 bits for the mantissa.

K	Latency (μs)	Clock cycles	Acc. %
20	$(0.28 \pm 0.05) \cdot 10^2$	$(2.79 \pm 0.54) \cdot 10^3$	98.54
16	$(0.23 \pm 0.04) \cdot 10^2$	$(2.29 \pm 0.40) \cdot 10^3$	99.63
13	$(0.19 \pm 0.04) \cdot 10^2$	$(1.93 \pm 0.35) \cdot 10^3$	99.63
10	$(0.15 \pm 0.03) \cdot 10^2$	$(1.50 \pm 0.26) \cdot 10^3$	100
8	$(0.13 \pm 0.02) \cdot 10^2$	$(1.25 \pm 0.22) \cdot 10^3$	100
5	$(0.09 \pm 0.01) \cdot 10^2$	$(0.87 \pm 0.14) \cdot 10^3$	100
3	$(0.06 \pm 0.01) \cdot 10^2$	$(0.62 \pm 0.09) \cdot 10^3$	100
2	$(0.05 \pm 0.01) \cdot 10^2$	$(0.49 \pm 0.07) \cdot 10^3$	100
1	$(0.04 \pm 0.01) \cdot 10^2$	$(0.36 \pm 0.04) \cdot 10^3$	99.27

Table S4: FloPoCo 16-bits, 5 bits for the exponent and 8 bits for the mantissa.

K	Latency (μs)	Clock cycles	Acc. %
20	$(0.26 \pm 0.06) \cdot 10^2$	$(2.64 \pm 0.59) \cdot 10^3$	96.36
16	$(0.22 \pm 0.04) \cdot 10^2$	$(2.15 \pm 0.41) \cdot 10^3$	98.90
13	$(0.18 \pm 0.04) \cdot 10^2$	$(1.84 \pm 0.40) \cdot 10^3$	98.54
10	$(0.15 \pm 0.03) \cdot 10^2$	$(1.47 \pm 0.35) \cdot 10^3$	98.90
8	$(0.12 \pm 0.03) \cdot 10^2$	$(1.23 \pm 0.32) \cdot 10^3$	98.90
5	$(0.09 \pm 0.03) \cdot 10^2$	$(0.89 \pm 0.32) \cdot 10^3$	98.90
3	$(0.07 \pm 0.04) \cdot 10^2$	$(0.68 \pm 0.42) \cdot 10^3$	98.90
2	$(0.06 \pm 0.03) \cdot 10^2$	$(0.62 \pm 0.60) \cdot 10^3$	98.90
1	$(0.04 \pm 0.03) \cdot 10^2$	$(0.38 \pm 0.28) \cdot 10^3$	96.72

Table S5: FloPoCo 12-bits, 4 bits for the exponent and 5 bits for the mantissa.

As can be seen from the reported results (see Tables S2, S3, S4 and S5), the floating-point operators generated by FloPoCo are computationally more efficient with respect to our initial implementation, also when the same number of bits is used. Indeed, using 32 bits and K equal to 20, the average latency using FloPoCo is $31.18 \mu\text{s}$ (see Table S2) compared to $102 \mu\text{s}$ for our the initial implementation (see Table S1). In Tables S2, S3, S4 and S5, we are reporting the results obtained when an increasingly higher number of bits is used for the floating point representation. In general, an improvement in inference speed is observed as the number of bits decreases. However, as expected, reducing numerical precision results in a decrease in accuracy due to increased rounding errors caused by the limited range of bits used to represent the numbers. In addition, depending on the number of bits used, we observe also an increasing number of overflow error especially when the K value is high and the number of bits is comparably low. For instance in Table S4, where 16-bits are used, the accuracy initially increase when K decrease, this is clearly related to overflow errors occurring when adding many terms in series expansion (i.e., big K values in equation S1).

In Figure S2 we are reporting: the average number of clock cycles as a function of K. While the number of bits used as, overall, quite a small impact on the latency, instead it has a big impact on resources utilization and power consumption (see Table S2).

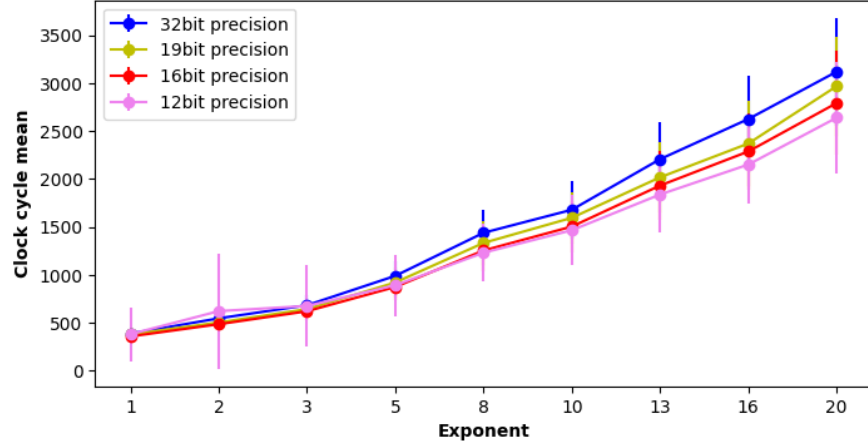


Figure S2: Average clock cycles as a function of the K values using 32bit, 19bit, 16bit and 12bit precision

Bit	Luts	Luts %	Regs	Regs %	DSPs	DSPs %
32	14306	26.89	9264	8.70	28	12.72
19	7202	13.54	5717	5.37	14	6.32
16	7738	14.54	5487	5.15	0	0
12	4133	7.76	5094	4.78	0	0

Table S6: Resource usage comparison as a function of the number of bits used for the floating-point representation (see text for details).

Table S6 shows the resource usage comparison as a function of the number of bits used for the floating point representation, in detail: "Bit" represents the number of bits used; "Luts" are the number of Look-Up Tables used estimated by Vivado [4]; "Luts (%)" column represents, instead, the percentage of Look-Up Table utilization; "Regs" are the number of registers used as estimated by Vivado; "Regs (%)" column represents instead the percentage of register utilization; "DSPs" are the number of Digital Signal Processors used as estimated by Vivado; "DSPs (%)" column represents the percentage of DSP utilization.

All the values reported have been obtained using a fixed value of K (i.e., K equal to 5), given that the K value has a negligible impact on the resources utilization.

2. Energy consumption measurements

Energy consumption is a crucial factor to take into account in the context of machine and deep learning. Both the training and inference phases come with computational and energy costs. Although there is a general tendency to believe that the training phase is more energy-intensive compared to inference, in production environments, the expenses related to inference exceed those during training due to the multiplicative factor arising from the repeated utilization of the system [5]. Indeed, it is estimated that inference accounts for 90% of compute cost [6]. The use of specialized and energy-efficient hardware such as FPGAs for this type of task is a viable solution to reduce energy costs. For these reasons, we have chosen to monitor the actual energy consumption of our solution. In particular, we aim to track variations resulting from changes in numerical precision and the number of bits, as described earlier, using FloPoCo.

In order to monitor the power consumption of the ZedBoard, the board is powered by a stabilized DC power supplies (E3634A by Agilent Technologies, Inc., Santa Clara, CA 95051 United States), set at $12\text{ V} \pm 2\text{ mV}$, in order to keep the supply voltage stable. The current absorption is measured via a benchtop digital multimeter (Tektronix DMM4020 by Tektronix Inc., Beaverton, OR, USA) with a resolution of $100\text{ }\mu\text{A}$.

With the measuring setup described above we are able to measure the current consumption of the entire board, with peripherals (e.g. LEDs) and the ARM/FPGA SoC. In the following we are interested in estimating the power consumption of only the FPGA and thus the other contribution should be separated. The total power consumption of the FPGA can be represented as the sum of leakage power (P_{leak}), which does not depends only on the state of the system, and dynamic power (P_{dyn}), which account for the power dissipated every time a signal changes[7]. The main contributors to leakage power consumption are: subthreshold leakage, reverse-biased junction leakage, gate leakage and source/drain tunneling leakage[8]. Leakage power is due both to the logic parts (e.g. inverters, multiplexer, LUTs and Flip-flop) and to the routing network. To mitigate leakage power consumption in FPGAs designers employ various techniques, such as: power gating, voltage scaling, and

architectural optimizations[9].

Dynamic consumption, on the other hand, refers to the energy consumed during state transitions and execution of the processing task. In this state the circuit dissipate power every time a signal changes, due to the charging and discharging of load and parasitic capacitance[10]. The dynamic power consumption can be modeled as:

$$P_{dyn} = \frac{1}{2} \sum_n C_n \alpha_n f_{clk} V^2$$

Where C_n is the capacitance of a net n , α_n is the switching activity factor of net n , f_{clk} is the clock frequency and V is supply voltage. Our goal is to identify the energy consumption exclusively of our IP module performing inference, excluding the PS (Processing System) part and AXI components. To this scope, we developed a firmware without our IP module as baseline. The execution of this firmware allow the estimation of the power consumption of the board with only AXI IP in the FPGA. This baseline permits the estimation of the power consumption of the design with our IP module by isolating its specific impact.

Bit	S-Power (W)	D-Power (W)	Time/inf. (μ s)	En./inf. (J)
12	$(0.20 \pm 0.17) \cdot 10^2$	$(1.20 \pm 0.17) \cdot 10^2$	6.76 ± 1.12	$(0.81 \pm 0.1) \cdot 10^{-7}$
16	$(1.17 \pm 0.17) \cdot 10^2$	$(2.4 \pm 0.17) \cdot 10^2$	6.21 ± 1.24	$(1.49 \pm 0.1) \cdot 10^{-7}$
19	$(1.13 \pm 0.17) \cdot 10^2$	$(2.2 \pm 0.17) \cdot 10^2$	6.44 ± 1.31	$(1.39 \pm 0.1) \cdot 10^{-7}$
32	$(3.7 \pm 0.17) \cdot 10^2$	$(5.5 \pm 0.17) \cdot 10^2$	6.84 ± 1.53	$(3.78 \pm 0.1) \cdot 10^{-7}$

Table S7: Power consumption as a function of the number of bits used for the floating-point representation.

Table S7 shows the resource usage comparison as a function of the number of bits used for the floating point representation, in particular: "Bit" represents the number of bits used; "S-Power (W)" represents the power consumption in Watt of our IP module when it is in idle state; "D-Power (W)" represents the power consumption in Watt of our IP module when it is actively processing data; "Time/inf. (s)" represents the time taken for a single inference in seconds; "En./inf. (J)" represents the energy usage in Joule per single inference. Evidently, we observe a distinct variation between static

and dynamic power consumption, depending also on how the arithmetic operators generated by FloPoCo are optimized. Furthermore, as expected, the use of higher precision implies higher consumption. However, the higher dynamic consumption observed with 32-bit precision represents approximately 1.4% of the total consumption.

Finally, just to give an overview and a broader comparison alongside the FPGA results, we conducted energy consumption measurements for the same task and using the same DNN model on standard CPU architectures. The two architectures used are the ARM Cortex-A9 processor (Zedboard) and the Intel i7-1260P processor. Python [11] was chosen as programming language and TensorFlow Lite [12] was used as framework to perform classifications. Regarding energy consumption monitoring, the Perf tool [13] was used for the Intel i7-1260P processor (i.e., Perf supports Running Average Power Limit for power consumption measurements [14]). Instead, in the case of the ARM Cortex-A9 processor the same monitoring system used for the FPGA measurements has been used. In both cases, the monitored energy consumption does not include the time for loading the model and dataset but only the time required by the TensorFlow Lite to invoke the predict function, which performs inference for a single sample. As for the Intel processor, we estimated an energy consumption of approximately 10^{-4} J and a time per single inference on the order of 10^{-6} s . On the other hand, regarding the ARM processor, the estimated energy consumption is of the order of 10^{-6} J, with a time per single inference on the order of 10^{-2} s. In the latter case the power consumption is generally of one or more order of magnitude bigger respect to the FPGA one, being the inference time much bigger. Instead, in the case of the Intel processor, we are getting an similar inference time but a much bigger power (i.e., almost three order of magnitude bigger).

3. Physical layout and congestion analysis

To provide further insight into the structural efficiency and scalability of our design, we examined the physical placement and routing.

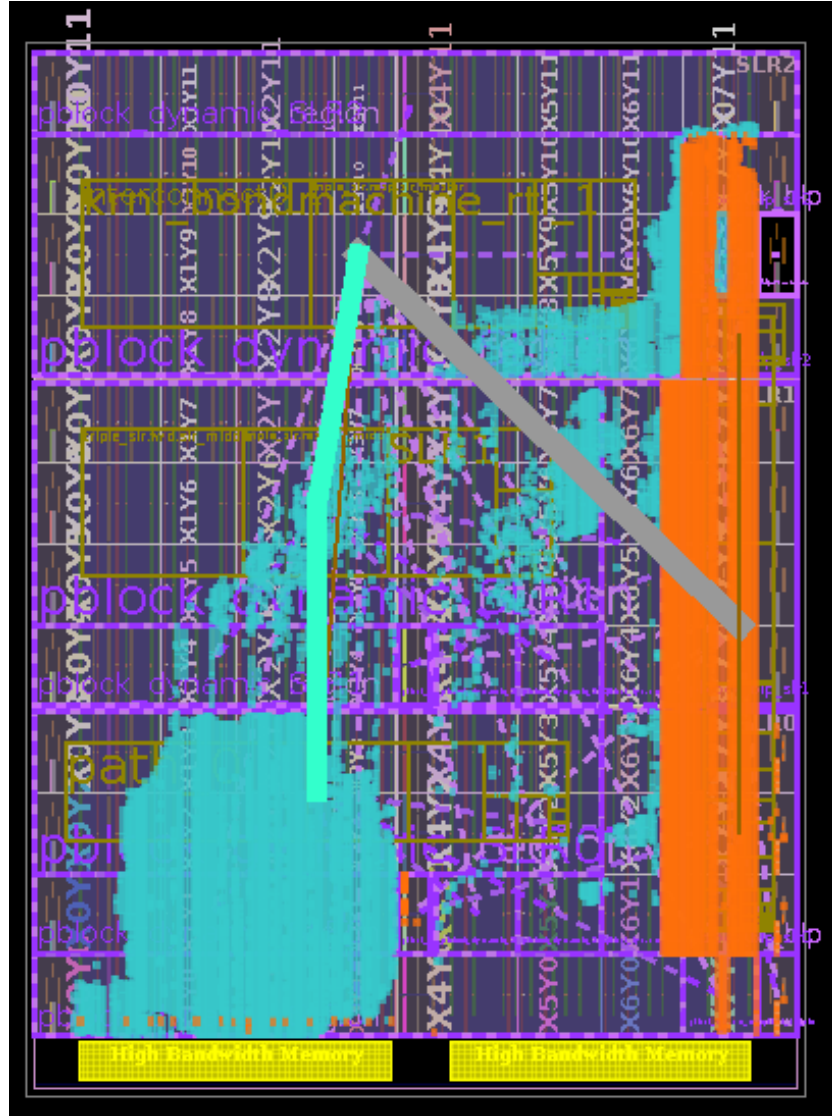


Figure S3: LUT and DSP distribution on the FPGA fabric. The map illustrates the non-clustered, sparse arrangement of logic resources, indicating low to moderate congestion levels. The localized congestion near HBM interfaces is expected due to numerous I/O connections.

The LUT and DSP distribution, illustrated in Figure S3, reveals a non-clustered, sparse arrangement of logic. This is a direct result of the BM's architectural mapping, which distributes computational tasks across the FPGA fabric and prevents the formation of a single, dense hot-spot. Although the

Vivado routing congestion map shows signs of localized congestion, particularly at the High-Bandwidth Memory (HBM) interfaces that are expected for designs with numerous I/O connections to external memory, it also indicates that the design exhibits low to moderate congestion levels overall. While the design is already robust and meets all timing constraints, it can definitely be improved through further optimization of the BM architecture and the underlying numerical operators. The design’s successful implementation on a fractional part of the available resources on the Alveo U55C, the absence of critical timing paths and a manageable congestion profile, validates our architectural approach as structurally efficient and scalable. This confirms that it can be adapted to larger models and datasets without significant performance degradation and without introducing critical congestion issues.

References

- [1] Xilinx, Zedboard fpga.
URL <https://www.xilinx.com/products/boards-and-kits/1-8dyf-11.html>
- [2] Pynq: The xilinx platform for python on fpgas.
URL <http://www.pynq.io/>
- [3] F. d. Dinechin, C. Lauter, A. Tisserand, Flopoco: A generator of floating-point arithmetic operators for fpgas, *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 2 (2) (2009) 10.
- [4] T. Feist, Vivado design suite, White Paper 5 (2012) 30.
- [5] F. Plumed, S. Avin, M. Brundage, A. Dafoe, S. hÉigeartaigh, J. Hernandez-Orallo, Accounting for the neglected dimensions of ai progress (06 2018).
- [6] D. Thomas, Reducing machine learning inference cost for pytorch models (03 2020).
- [7] A. D. G. García, L. F. G. Pérez, R. F. Acuña, Power consumption management on fpga, in: *15th International Conference on Electronics, Communications and Computers (CONIELECOMP’05)*, IEEE, 2005, pp. 240–245.

- [8] A. Agarwal, S. Mukhopadhyay, A. Raychowdhury, K. Roy, C. H. Kim, Leakage power analysis and reduction for nanoscale circuits, *IEE Micro* 26 (2) (2006) 68–80.
- [9] L. Chang, D. J. Frank, R. K. Montoye, S. J. Koester, B. L. Ji, P. W. Coteus, R. H. Dennard, W. Haensch, Practical strategies for power-efficient computing technologies, *Proceedings of the IEEE* 98 (2) (2010) 215–236.
- [10] A. Wiltgen, K. A. Escobar, A. I. Reis, R. P. Ribas, Power consumption analysis in static cmos gates, in: 2013 26th Symposium on Integrated Circuits and Systems Design (SBCCI), IEEE, 2013, pp. 1–6.
- [11] G. Van Rossum, F. L. Drake, *Python 3 Reference Manual*, CreateSpace, Scotts Valley, CA, 2009.
- [12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattemberg, M. Wicke, Y. Yu, X. Zheng, TensorFlow: Large-scale machine learning on heterogeneous systems, software available from tensorflow.org (2015).
URL <https://www.tensorflow.org/>
- [13] A. C. De Melo, The new linux’perf’tools, in: *Slides from Linux Kongress*, Vol. 18, 2010, pp. 1–42.
- [14] H. David, E. Gorbato, U. R. Hanebutte, R. Khanna, C. Le, Rapl: Memory power estimation and capping, in: *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, 2010, pp. 189–194.