Integration of a systolic array based hardware accelerator into a DNN operator auto-tuning framework

Federico Nicolás Peccia
FZI Research Center for Information Technology
Karlsruhe, Germany
peccia@fzi.de

Oliver Bringmann
University of Tübingen
Tübingen, Germany
oliver.bringman@uni-tuebingen.de

***Abstract*—The deployment of neural networks on heterogeneous SoCs coupled with custom accelerators is a challenging task because of the lack of end-to-end software tools provided for these systems. Moreover, the already available low-level schedules and mapping strategies provided by the accelerator developers for typical tensor operations are not necessarily the best possible ones for each particular use case. This is why frameworks which automatically test the performance of the generated code on a specific hardware configuration are of special interest. In this work, the integration between the code generation framework TVM and the systolic array-based accelerator Gemmini is presented. A generic schedule to offload the GEneral Matrix Multiply (GEMM) tensor operation onto Gemmini is detailed, and its suitability is tested by executing the AutoTVM tuning process on it. Our generated code achieves a peak throughput of 46 *giga-operations per second* (GOPs) under a 100 MHz clock on a Xilinx ZCU102 FPGA, outperforming previous work. Furthermore, the code generated by this integration was able to surpass the default hand-tuned schedules provided by the Gemmini developers in real-world workloads.**

***Index Terms*—RISC-V, FPGA, TVM, Gemmini, Accelerator, Code Generation**

I. Introduction

Heterogeneous SoCs generators like Chipyard [1] or HEROv2 [2] are quickly being adopted for an increasing amount of use cases thanks to their great adaptability. This kind of generator exposes an enormous amount of possible configurations to the user, thus enabling the generation of SoCs tailor-made for specific applications. Thanks to the open nature of these projects, hardware accelerators like Gemmini [3], VTA [4] or NVDLA [5] are being developed to offload specific workloads from the CPU, allowing the deployment of complex algorithms onto edge platforms.

Although the developers of these accelerators normally provide hand-tuned kernels to offload commonly used tensor operations into the accelerator, the high configurability of these SoC generators becomes a problem: there is no guarantee that these default schedules will provide the best throughput across all possible SoC configurations.

This is why automatic code generation and evaluation tools are becoming increasingly popular [6], [7], [8]. These present enormous advantages, as they can easily generate different mapping options for each tensor operator and test them to automate the process of finding the best schedule parameters given a particular SoC configuration (a process known as *auto-tuning*). By measuring on a physical hardware platform, the impact of other system components on the accelerator's operation is also taken into account.

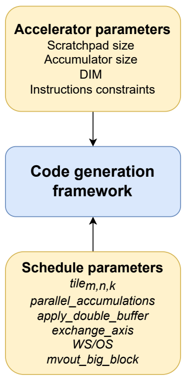
To demonstrate the advantages of this auto-tuning process for hardware accelerators, this work presents the integration of the Gemmini accelerator into the TVM deployment framework. The paper is organized as follows: first, a scheduling search space for a GEMM operation on a generic systolic array accelerator is proposed in Section II. Then, Section III presents the integration of the Gemmini accelerator into the TVM framework. [1]. Finally, Section IV compares the auto-tuning of different GEMM workloads against a reference implementation [9], and demonstrates that our scheduling definition achieves improvements in terms of *giga-operations per second* (GOPs) for all workloads. The measurements are expanded by presenting the autotuning results for operators extracted from the Baidu DeepBench Workloads, outperforming the default handcrafted kernels provided by the Gemmini developers.

II. Scheduling a GEMM operation on a systolic array

In 1982, Kung [10] presented the advantages of systolic array architectures for the execution of matrix operations. These have been especially attractive to accelerate neural network operators because of their high data reuse. Several accelerators were built using systolic arrays (or similar processing element distributions) as its core [11], [12], [13], [14], [15], [16].

The research leading to these results was supported by the Competence Center Karlsruhe for AI Systems Engineering (CC-KING https://www.ai-engineering.eu) sponsored by the Ministry of Economic Affairs, Labour and Tourism Baden-Württemberg. It was conducted within the project KIsSME (Artificial Intelligence for selective near-real-time recordings of scenario and manoeuvre data in testing highly automated vehicles, Grant No. 19A20026C), funded by the German Federal Ministry for Economic Affairs and Climate Action.

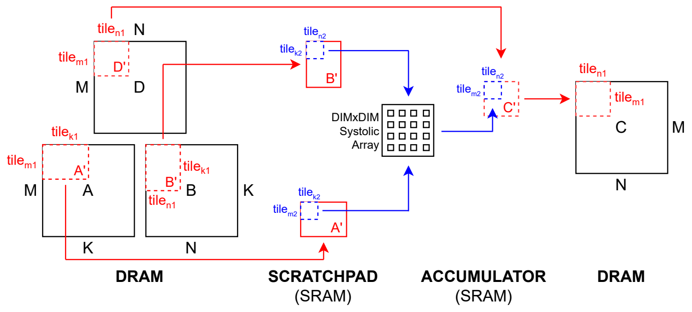
[1]As of the date of submission of this paper, the merging of this integration into the main TVM branch is still in progress



(a)

```
config.accel()
for $i_o = 0$ to $(M/tile_{m_1}) - 1$ do
    for $j_o = 0$ to $(N/tile_{n_1}) - 1$ do
        move.in($D'$)
        for $k_o = 0$ to $(K/tile_{k_1}) - 1$ do
            move.in($A'$)
            move.in($B'$)
            for $i_i = 0$ to $(tile_{m_1}/tile_{m_2}) - 1$ do
                for $j_i = 0$ to $(tile_{n_1}/tile_{n_2}) - 1$ do
                    for $k_i = 0$ to $(tile_{k_1}/tile_{k_2}) - 1$ do
                        gemm.DIMxDIM($C'$,$A'$,$B'$)
                    end for
                end for
            end for
        end for
        for $i_i = 0$ to $(tile_{m_1}/tile_{m_2}) - 1$ do
            for $j_i = 0$ to $(tile_{n_1}/tile_{n_2}) - 1$ do
                move.out($C'$)
            end for
        end for
    end for
end for
```

(b)



(c)

Figure 1: (a) shows the proposed GEMM schedule parameters for an accelerator based on a $DIM \times DIM$ systolic array able to execute a generic GEMM with form $C = A \times B + D$. (b) shows an example generated pseudocode for the operation, and (c) shows a graphical representation of the move of data in and out of the accelerator.

To correctly schedule a tensor operation on a hardware accelerator, one should be aware of the kind of instructions provided by the accelerator. These can be classified into two distinct categories:

- *Low level*: these instructions allow the programmer fine control of the behaviour of the accelerator. These are typically instructions that enable moves of data in and out of the accelerator's internal memory, and others that execute/dispatch the most basic supported computation. An example of this kind of instructions can be found on the Angel-Eye [17] or the VTA [4] accelerators.
- *Layer wide*: these instructions take care of the burden of managing the individual instructions, by exposing to the programmer a higher-level interface to execute an entire tensor operation, like the NVDLA [5].

For a systolic array-based accelerator, we are interested in the first kind of instructions, because they allow the programmer or the tuning software fine-grained control over the generated schedules. To correctly schedule them, the code generator framework should take these 4 factors into account:

1) *Configuration of the hardware*: these instructions should be generated only when the configuration of the accelerator changes, and not on each new operator, to avoid unnecessary reconfigurations.
2) *Move of data into the accelerator's SRAM*: if the input matrices fit entirely on the accelerator's internal memory, one could choose to first move the entire matrices in, and only then start to generate the compute instructions. But perhaps interleaving move and compute instructions can actually avoid idle times and thus generate faster compute schedules. The proposed schedule should be able to achieve this load balancing between different kinds of instructions.
3) *Computation*: the idle time of the systolic array should be minimized. There should always be data available in the accelerator's SRAM for the systolic array to use for its computation.
4) *Move of data out into the external DRAM*: two different options could be chosen when generating these instructions: either patches of results are moved out as soon as they are ready, or multiple patches are stored in the accumulator before bulk moving all of them out.

To limit the explosion of the schedule parameter search space, certain hardware limitations and basic assumptions can be taken into account during this stage:

- Schedules which do not respect the maximum limitation of columns and rows for each move should automatically be dropped.
- The schedule cannot generate configurations where data would overflow the accelerator's SRAM: hardware information should be used as a limit for the generated move instructions source and destination addresses.
- Each generated GEMM should try to utilize the systolic array to its full extent, to prevent idle processing elements.

The configuration parameters of the proposed schedule presented in Fig. 1 along with the accelerator parameters can effectively generate code that considers all the previous mentioned points:

- $tile_{m,n,k}$: two-level tiling for each corresponding computation axis. The first level moves data into the accelerator's memory, and the second partitions the moved data into smaller GEMMs to fit the systolic array, trying to maximize its usage.
- $parallel\_accumulations$: denotes how many output patches are accumulated simultaneously in the accumulator, before moving them out. Modifies the position of the *move.out* instruction in the generated code.
- $apply\_double\_buffer$: represents if double buffering should be applied on the *move.in* of data, weights, both or none. Takes into account the bank size of the scratchpad,

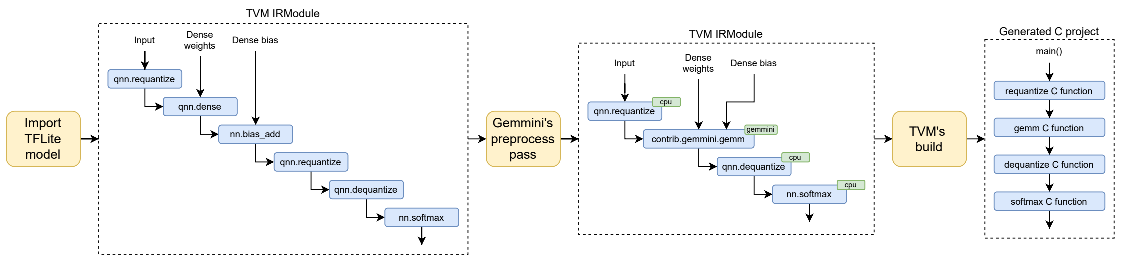


Figure 2: Integration workflow example for a neural network formed by a fully connected layer and a softmax layer

so that double-buffered data is written to different banks, thus preventing bank access conflict.

- $exchange\_axis$: allows to reorder the computation of axis M and N, useful when M and N are not equal.
- WS/OS: if supported by the accelerator, configures the systolic array to work on output stationary mode or weight stationary mode [18].
- $mvout\_big\_block$: if supported by the accelerator, enables the generation of *move.out* instructions which move more than $DIM \times DIM$ size patches. Useful to explore the trade-off between burst of smaller *move.out* instructions versus bigger instructions.

III. Integrating Gemmini into TVM

To validate the aforementioned schedule search space, the TVM [7] framework was chosen to take advantage of its AutoTVM module. This *auto-tuning* process goes over the parameter search space, measures each configuration on physical hardware, and exports the best schedule configuration. To avoid an infeasible amount of measurements, XGB model-based tuners are available [19].

The Gemmini [3] systolic array-based accelerator was selected as a test platform because of its open-source nature. Gemmini was developed by the UC Berkeley and is part of the Chipyard ecosystem. It works in a tightly-coupled manner with a RISC-V CPU, using the Rocket Chip Coprocessor (RoCC) interface to control the accelerator with help from custom instructions.

Gemmini uses a systolic array of $DIM \times DIM$ *multiply-and-accumulate* (MAC) processing elements to perform matrix multiplications. The data is consumed from a scratchpad made up of banked SRAMs and is stored in a series of banked SRAMs equipped with adder units known as the accumulator. A DMA engine connected to the System Bus (directly to the L2 cache) is used to get data in and out of the accelerator's SRAMs. Gemmini is also able to apply scaling factors during the move in and out of data, and also other common operations like ReLu or max pooling.

Gemmini's RoCC instructions can be grouped into the following categories:

- *Configuration*: these instructions configure the input, output and execution pipelines of the accelerator.
- *Move*: these instructions move specific amounts of rows and columns of data into the SRAM or out into the DRAM.
- *Execution*: dispatch the actual execution of the $DIM \times DIM$ GEMM to the systolic array.
- *Flush and fence*: general maintenance instructions.
- *Loop*: "CISC" type instructions for commonly used operations. They take away the burden of manually scheduling the intrinsic instructions by generating them directly on the hardware using FSMs. Gemmini developers also provide some handcrafted layer-wise C functions, which internally call these loop instructions.

Gemmini uses a decoupled access-execute architecture, with dedicated controllers to manage the *move.in*, execute and *move.out* instructions independently. A ROB is included to detect hazards between instructions and to issue them to their respective controller.

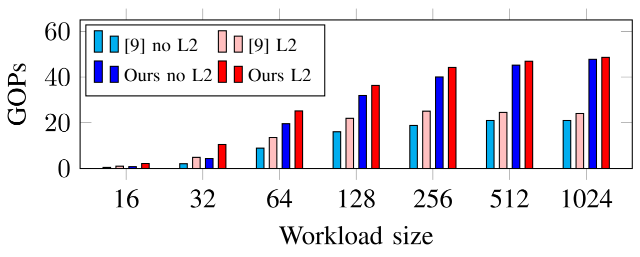
Fig. 2 shows how the developed integration between TVM and Gemmini works. First, the TensorFlow Lite quantized model is imported. Here, the model is transformed into the Relay IR dataflow graph representation of TVM. Then, pattern matching is used to replace subgraphs of operations with custom operators supported by Gemmini. These operators consist of a computation definition and its schedule: a set of parametrized loop transformations. For the GEMM operator, two different schedules were developed: one that generates the calls to the intrinsic instructions parametrized as presented in Section II, and one that replaces the entire loops with a call to the default handcrafted layer-wise C function.

During the schedule transformation, several *pragmas* are used to tag specific loops and then replace them with the Gemmini intrinsic instructions in the following compilation pass. The *tensorization* feature of TVM was used to insert the two instructions (*preload* and *compute*) needed by Gemmini to compute a GEMM. TVM's C code generator was used to create the source code file that executes the operators, and the standard header file provided by Gemmini was included in the generated file by TVM to correctly reference Gemmini's instruction macros.

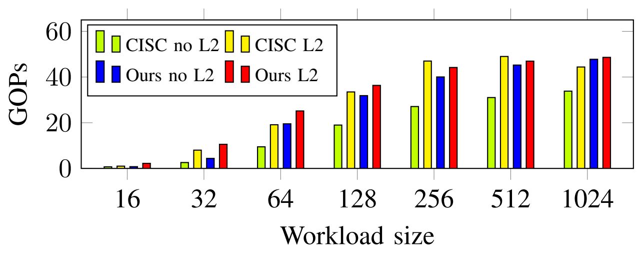
A. Quantization management

TensorFlow Lite's quantization scheme [20] uses symmetric quantization for the weights and biases of each layer, and asymmetric for the input and output of each layer [2]. Because the multiplication of two quantized matrices with different zero points is not straightforward, correction terms have to be

[2]see https://www.tensorflow.org/lite/performance/quantization_spec?hl=en



(a) Best schedules found by AutoTVM in our implementation against previous work.



(b) Best schedules found by AutoTVM in our implementation against the default "CISC instruction" based schedules.

Figure 3: Results across different GEMM workloads. For each workload, $M = N = K = workload\ size$

subtracted to get the correct result as seen in Eq. (1). Finally, to transform Q'_C into the output quantization regime of the layer, a *requantization* operator is inserted by TVM described by Eq. (2).

An easy approach would be to accelerate only the matrix multiplication using the systolic array (the term 1 of Eq. (1)) and then execute the correction terms subtraction and the re-quantization operator on the CPU. But this is slow and doesn't exploit the benefits of TVM's compilation framework, so a transformation of the GEMM operation is done during the compilation pass, allowing TVM to fold the remaining constants into the bias of the layer. The final proposed solution for a quantized matrix multiplication with bias addition is described in Eq. (3) and (4). The scaling factor s_d/s_c is inserted as output scaling factor in the configuration of the *move.out* instruction of Gemmini, thus achieving the acceleration of the entire operator, without extra tensor operations executed on the CPU.

$$Q'_{C_{(m,n)}} = \sum_0^k \left(Q_{A_{(m,k)}} - zp_a\right) * Q_{B_{(k,n)}} = \sum_0^k Q_{A_{(m,k)}} * Q_{B_{(k,n)}} - \sum_0^k zp_a * Q_{B_{(k,n)}} \quad (1)$$

$$Q_C = zp_c + \frac{s_d}{s_c} * Q'_C \quad (2)$$

$$Q_{C_{(m,n)}} = \frac{s_d}{s_c}\left[\left(\sum_0^k Q_{A_{(m,k)}} * Q_{B_{(k,n)}}\right) + Q'_{D_{(m,n)}}\right] \quad (3)$$

$$Q'_{D_{(m,n)}} = Q_{D_{(m,n)}} - \sum_0^k zp_a * Q_{B_{(k,n)}} + \frac{s_d}{s_c} * zp_c \quad (4)$$

IV. Experiments

Table I: Selected Baidu DeepBench workloads

Id	M	N	K
15	64	1	1216
49	128	1	1024
63	512	1	512
73	512	2	512
84	1024	4	512

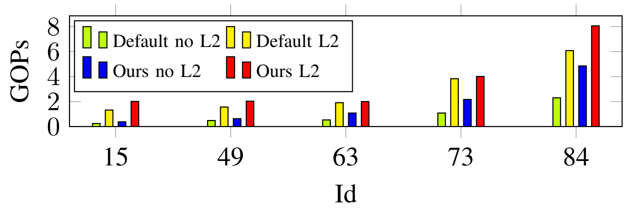


Figure 4: Best schedules found by AutoTVM for the Baidu DeepBench dataset using our implementation.

In order to be able to compare our results against the ones reported by [9], our test setup was also built using the Rocket Core [21] together with a 16×16 Gemmini accelerator, with a 256 KB scratchpad (4 SRAM banks with 4096 rows each) and a 64 KB accumulator (1 SRAM bank with 1024 rows). The inputs and weights are represented using 8 bits and the accumulated values using 32 bits. In the implemented design which uses the L2 cache, the SiFive inclusive L2 cache was used. Like the original paper, the resulting hardware was also implemented for a Xilinx Zynq UltraScale+ ZCU102 FPGA running at 100 MHz. The only difference between our setup and the one from [9] is the size of the L2 cache because they did not report the selected size. In our work, the default size used in the Chipyard project was selected: 512 KB.

Fig. 3 presents the best schedule performance found by AutoTVM, and compares it with the previous work and the default Gemmini schedules. An XGB tuner [19] with early stopping equal to 500 iterations was used to traverse the search space. The amount of operations for a generic GEMM of form $C_{[M,N]} = A_{[M,K]} \times B_{[K,N]} + D_{[M,N]}$ was defined as $OP = 2 \times M \times N \times K + M \times N$.

Although Fig. 3b shows that our AutoTVM schedules are better than the CISC-based schedules for almost all workloads, it fails to find better schedules for workloads 256 and 512 when the L2 cache is activated. There are two possible explanations for this behaviour. The first one would be that the XGB tuner is stuck in a local optimum, but this was verified to **not** be the case by executing an exhaustive grid-search of all the schedule parameter search space: the XGB found schedules are indeed the best possible schedules our implementation can generate for that workloads. The second possible explanation lies in a *load balancing* feature of the Gemmini's CISC instructions. The hardware FSMs monitor the proportion of each instruction type in the ROB and can pause the generation of each kind of instruction, to maximize the overlap between move and execute operations. This behaviour can not yet be replicated using our TVM integration, and further work needs to be done to analyse how to implement a similar feature.

To show the effectiveness of this schedule search space on real-world examples, a set of dense layer workflows taken from the Baidu DeepBench dataset [22] were selected (Table I). Fig. 4 presents the result of the AutoTVM tuning process executed on these workloads, using the same tuner parameters and FPGA bitstreams as in the previous measurements.

V. Conclusions

This work proposed a schedule parameter space for a GEMM tensor operation. The proposed schedule was configured into the TVM deep learning compiler, and integrated with the Gemmini systolic array hardware accelerator. We demonstrate that this schedule parameter space allows the autotuning process to find faster schedules than previous work, and also faster schedules than the hardcoded schedules provided by the expert developers of Gemmini on almost all tested workloads.

Future works will also add other operators to the integration, like convolutions and depthwise convolutions. The results of the acceleration of entire neural networks will also be presented.

This work did not try to improve how much time the autotuning process takes. The search space of scheduling parameters was traversed using an XGB tuner approach, which can be improved to converge faster as shown in [23]. In future works, optimization techniques to speed up this process should be investigated, to be able to find the best scheduling parameters using the minimum amount of actual hardware measurements.

References

[1] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[2] A. Kurth, B. Forsberg, and L. Benini, "Herov2: Full-stack open-source research platform for heterogeneous computing," *CoRR*, vol. abs/2201.03861, 2022. [Online]. Available: https://arxiv.org/abs/2201.03861

[3] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.

[4] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A hardware-software blueprint for flexible deep learning specialization," 7 2018. [Online]. Available: http://arxiv.org/abs/1807.04188

[5] G. Zhou, J. Zhou, and H. Lin, "Research on nvidia deep learning accelerator," in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2018, pp. 192–195.

[6] L. Zheng, C. Jia, M. Sun, Z. Wu, C. H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, J. E. Gonzalez, and I. Stoica, "Ansor : Generating high-performance tensor programs for deep learning," *CoRR*, vol. abs/2006.06762, 2020. [Online]. Available: https://arxiv.org/abs/2006.06762

[7] T. Chen, T. Moreau, Z. Jiang, H. Shen, E. Q. Yan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: end-to-end optimization stack for deep learning," *CoRR*, vol. abs/1802.04799, 2018. [Online]. Available: http://arxiv.org/abs/1802.04799

[8] S. Zheng, Y. Liang, S. Wang, R. Chen, and K. Sheng, "Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 859–873. [Online]. Available: https://doi.org/10.1145/3373376.3378508

[9] P. Xu and Y. Liang, "Automatic code generation for rocket chip rocc accelerators," *CARRV*, 2020.

[10] Kung, "Why systolic architectures?" *Computer*, vol. 15, pp. 37–46, 1 1982. [Online]. Available: http://ieeexplore.ieee.org/document/1653825/

[11] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, 2017.

[12] Y. H. Chen, T. J. Yang, J. S. Emer, and V. Sze, "Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 9, pp. 292–308, 6 2019.

[13] Z.-G. Liu, P. N. Whatmough, and M. Mattina, "Systolic tensor array: An efficient structured-sparse gemm accelerator for mobile cnn inference," 5 2020. [Online]. Available: http://arxiv.org/abs/2005.08098

[14] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," vol. Part 128280, 2017.

[15] R. Xu, S. Ma, Y. Wang, X. Chen, and Y. Guo, "Configurable multi-directional systolic array architecture for convolutional neural networks," *ACM Transactions on Architecture and Code Optimization*, vol. 18, 2021.

[16] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-datacenter performance analysis of a tensor processing unit," *ACM SIGARCH Computer Architecture News*, vol. 45, 2017.

[17] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, 2018.

[18] A. Samajdar, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic CNN accelerator," *CoRR*, vol. abs/1811.02883, 2018. [Online]. Available: http://arxiv.org/abs/1811.02883

[19] T. Chen, L. Zheng, E. Yan, Z. Jiang, T. Moreau, L. Ceze, C. Guestrin, and A. Krishnamurthy, "Learning to optimize tensor programs," vol. 2018-December, 2018.

[20] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *CoRR*, vol. abs/1712.05877, 2017. [Online]. Available: http://arxiv.org/abs/1712.05877

[21] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[22] S. Narang and G. Diamos, "Baidu deepbench," *GitHub Repository*, 2017.

[23] D. Rieber, M. Reiber, O. Bringmann, and H. Fröning, "Hw-aware initialization of dnn auto-tuning to improve exploration time and robustness," 5 2022. [Online]. Available: http://arxiv.org/abs/2205.15568