Exploring Direct Convolution Performance on the Gemmini Accelerator

Caio Vieira[1], Arthur F. Lorenzon[2], Lucas Mello Schnorr[1], Philippe Olivier Alexandre Navaux[1], Antonio Carlos Schneider Beck[1]

[1]Institute of Informatics - Federal University of Rio Grande do Sul (UFRGS)
Porto Alegre - Brazil

[2]Federal University of Pampa - Alegrete - Brazil

`cravieira@inf.ufrgs.br, aflorenzon@unipampa.edu.br,`

`{schnorr, navaux, caco}@inf.ufrgs.br`

Abstract. *Convolutional Neural Network (CNN) algorithms are becoming a recurrent solution to solve Computer Vision related problems. These networks employ convolutions as main building block, which greatly impact their performance since convolution is a costly operation. Due to its importance in CNN algorithms, this work evaluates convolution performance in the Gemmini accelerator and compare it to a conventional lightly- and heavily-loaded desktop CPU in terms of execution time and energy consumption. We show that Gemmini can achieve lower execution time and energy consumption when compared to a CPU even for small convolutions, and this performance gap grows with convolution size. Furthermore, we analyze the minimum Gemmini required frequency to match the same CPU execution time, and show that Gemmini can achieve the same runtime while working in much lower frequencies.*

Resumo. *Algoritmos de Redes Neurais Convolucionais (CNN do inglês Convolutional Neural Network) tem se tornado uma solução recorrente para solucionar problemas de Visão Computacional. Estas redes empregam convoluções como principal bloco de construção, o que impacta a performance, pois convolução é um operação cara computacionalmente. Devido a sua importância nos algoritmos de CNN, este trabalho avalia o desempenho de convoluções no acelerador Gemmini e os compara com a execução em uma CPU convencional com cargas leves e pesadas de trabalho. Nós mostramos que o Gemmini pode atingir melhores tempos de execução e consumos energéticos até mesmo para pequenas convoluções e a diferença de desempenho cresce com o tamanho da convolução. Além disso, nós analisamos a frequência mínima necessária que o Gemmini deve ter para obter o mesmo tempo de execução de uma CPU e mostramos que o Gemmini é capaz de atingir o mesmo resultado mesmo quando trabalhando em frequências muito mais baixas.*

1. Introduction

Neural Network (NN) algorithms have been deployed in many different applications in recent years due to its capability to solve non linear problems [LeCun et al. 2015]. A variation of NNs are the Convolutional Neural Networks (CNN), which utilizes convolution operations in its layers and is heavily used in Computer Vision problems, such as image recognition. However, this class of algorithms requires high computational costs, demanding new solutions to achieve latency and throughput requirements. Therefore, in order to tackle this problem, different ways of processing NNs using hardware have emerged.

Because NN computations are in essence matrix and vector operations, their performance in CPUs can be improved by using SIMD (single instruction, multiple data) instructions [Vanhoucke et al. 2011]. However, GPUs are better suited to handle NNs workloads due to its parallel nature. Recent GPUs have specialized units [NVIDIA 2020] to deal with these workloads, speeding up the inference and training tasks. Nevertheless, the growing number of NN applications in different scenarios have demanded new specialized architectural solutions.

Therefore, to tackle this problem, many accelerators were proposed, such as Eyeriss, NVDLA, and Gemmini [Chen et al. 2016, Zhou et al. 2018, Genc et al. 2019]. They achieve better performance by implementing common NN functions in hardware and leveraging data-reuse opportunities, which is usually accomplished by employing systolic arrays based architectures [Kung 1982]. Systolic architectures can successfully be applied in compute-bound problems. The idea behind systolic systems is to fetch data from memory once and perform as many as possible computations on it before storing data back to memory, thus minimizing memory accesses and hence, the energy consumption. Each element in the array is a Processing Element (PE), which are chained in a pipeline fashion to form the array. Gemmini [Genc et al. 2019] is an example of systolic array generator which enables creation of systolic arrays with different sizes, and thus aiming different utilization scenarios.

When analyzing NN accelerators, requirements such as latency, throughput, and power consumption are used, and the benchmark programs consist of running entire NN algorithms. However, this methodology may not capture the execution behavior of each algorithm building block. Thus, in this work, we investigate the execution behavior in a fine-grain approach by comparing performance of convolutions between Gemmini and a conventional CPU. We choose to compare convolution execution because it is a common building block in CNN algorithms and it is a costly operation, accounting for over than 90% of computations in a CNN [Chen et al. 2016, Cong and Xiao 2014]. Results show that even for small convolutions, Gemmini shows better execution time and energy consumption than CPUs while working in lower frequencies. Furthermore, we show that CPU's performance degrades when it is under heavy load while Gemmini is not affected in this situation due to its dedicated hardware nature.

2. Background

2.1. 2D Convolution

Convolution is a mathematical operation used as building block in CNN algorithms [Howard et al. 2017, Krizhevsky et al. 2012, Lecun et al. 1998]. Given two matrices, input, I and kernel, K, a 2D convolution can be applied to produce an output matrix, O, according to Equation 1, where j and k define an element in matrix K, and m and n define an element in matrices O and I. The intuition behind this equation is to slide the kernel matrix over the input matrix. At each step, the overlapping elements in both matrices are multiplied, and then the partial results are summed to create a new element in the output

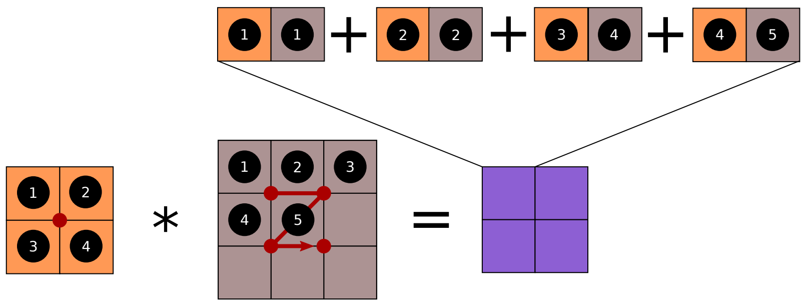


Figure 1. The kernel matrix overlaps the input matrix to produce the values of the output matrix.

matrix. This process is depicted in Figure 1, where the matrices are represented from left to right as kernel, input and output[1]. The dot in the kernel matrix represents its center and the dots inside the input matrix represent the position where the kernel matrix center must overlap. To compute the first element of the output matrix, the kernel overlaps the input matrix on the top left corner. Each overlapping element is multiplied and the partial sums are summed to produce the final output. This process is repeated until the kernel overlaps all possible positions, represented by the trajectory of the line connecting the dots.

$$O[m,n] = \sum_j \sum_k K[j,k] I[m-j, n-k] \tag{1}$$

The *stride* parameter determines the sliding step used to move the kernel. It is possible to pad zeros (*padding*) to the input matrix border in order to increase its dimension sizes, allowing the kernel to overlap more elements. Both parameters, stride and padding, determine the dimension of the output matrix, as shown in Equation 2. Where D represents the dimension size of a matrix (e.g., input, output, or kernel), p represents padding, and s the stride used. Throughout this work, we will consider only squared matrices, therefore D represents the number of rows and columns. Figure 2 shows an example of these parameters applied to a convolution, considering $s = 2$ and $p = 1$. The increased stride reduces the number of possible overlap positions. Notice that each dot is positioned two cells away of each other. The padding is represented by the increased size in the input matrix. In practice, these parameters affect D_O and hence, the total number of computations required by the convolution.

$$D_O = \left\lfloor \frac{D_I + 2p - D_K}{s} + 1 \right\rfloor \tag{2}$$

In CNNs, 2D convolutions usually occur with higher dimension tensors. Tensor is an N-dimensional array of data (matrices are 2D tensors). Both input, and kernel matrices can be extended to a third dimension conventionally named *channel*. Besides channels, it is also possible to work with a fourth dimension named *batch*. These four dimensions can be related to images. A image have two dimensions to represent the pixels, one dimension to represent the colors (channel), and the fourth is an image sample (batch).

[1]We will adopt this convention for Figures 1, 2, and 3.

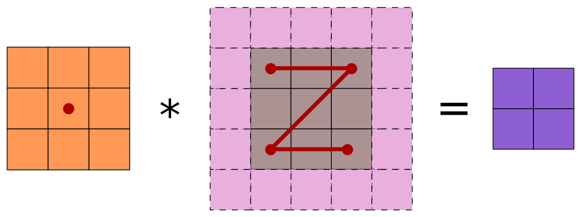


Figure 2. Convolution considering padding, p, equal to 1 and stride, s, equal to 2.

```
1  for(int n = 0; n < N; n++)
2   for(int orow = 0; orow < O_D; orow++)
3    for(int ocol = 0; ocol < O_D; ocol++)
4     for(int och = 0; och < O_C; och++)
5      for(int krow = 0; krow < K_D; krow++)
6       for(int kcol = 0; kcol < K_D; kcol++)
7        for(int ich = 0; ich < I_C; ich++) {
8         int iy = orow*s + krow - p;
9         int ix = ocol*s + kcol - p;
10        if(0 <= ix < I_D && 0 <= iy < I_D)
11         output[n][orow][ocol][och] +=input[n][iy][ix][ich]
            ↪  * kernel[och][krow][kcol][ich];
12       }
```

Listing 1: Direct convolution algorithm for 4-D tensors as 7 nested loops.

Equation 3 shows convolution complexity related to tensors' dimensions, where N represents the batch size and C the number of channels of a matrix. Figure 3 shows an example of 4D tensors used in convolutions and the relationship between its dimensions. The input tensor is determined by N, C_I, and D_I. The only free output tensor parameter is the C_O, since the batch must be the same as the input tensor and D_O is determined by Equation 2. Similarly, only D_K is a free parameter in the kernel tensor, since the number of its channels is determined by the number of channels in the input tensor and the number of its batch is the same as the number of channels in the output tensor.

A 2D convolution program can be implemented as shown in Listing 1. The program iterates over 7 dimensions and 5 of them are present in equation 3, only D_O in lines 2 and 3 are not. However, computation on the output matrix is performed only when the conditions in line 10 are satisfied, i.e. when it is possible to index an element in input tensor, explaining where the term D_I come from.

$$N \times C_O \times D_K \times D_K \times C_I \times D_I \times D_I \tag{3}$$

2.2. Gemmini

Tensor operations are compute-bound with high data reuse opportunities, either spatial, or temporal [Kwon et al. 2019]. Gemmini [Genc et al. 2019] is a systolic array generator which can be used to accelerate general matrix operations represented by Equation 4,

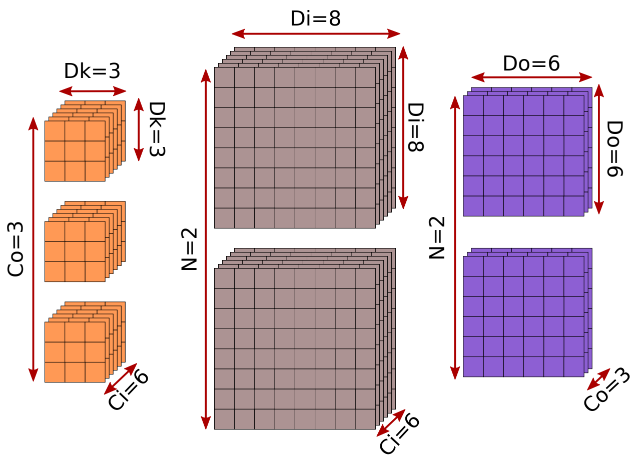


Figure 3. Tensors as present in CNN convolution layers.

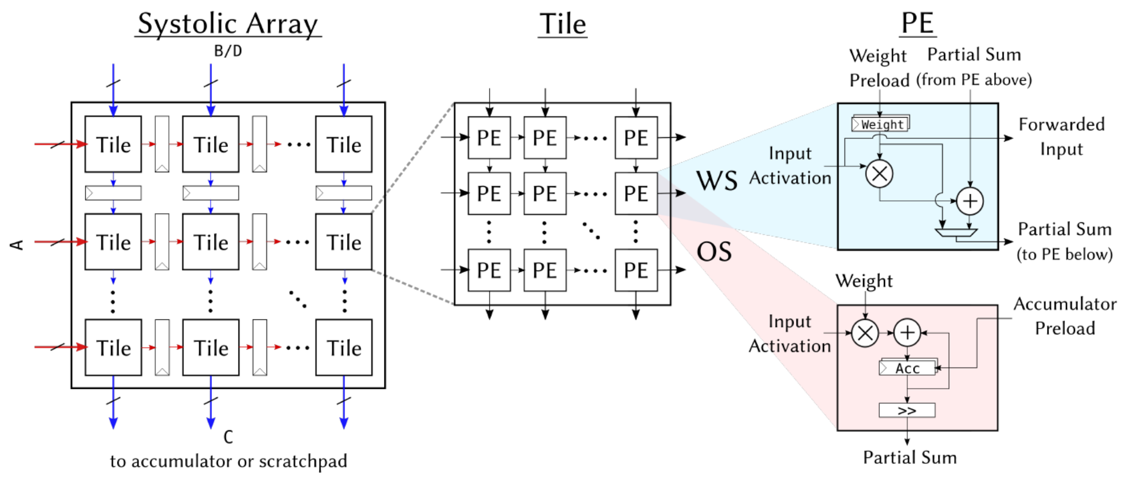


Figure 4. Overview of Gemmini's systolic array. Each tile is composed by fully-combinational PEs, and registers divide each tile. Source: H. Genc et al [Genc et al. 2019].

where A and B are multiplied matrices, D is a bias matrix, and C is the result.

$$C = A * B + D \tag{4}$$

Figure 4 depicts Gemmini systolic array internals. The basic computation unit in Gemmini is a processing element, which can perform a Multiply-Accumulate (MAC) operation. A set of fully combinational connected PEs compose a *tile*, which are arranged in a pipeline fashion to form the systolic array. Since Gemmini is a generator, it is possible to generate a new hardware by tunning its parameters (e.g. size of scratchpad memory and number of tiles) targeting a specific utilization scenario. Furthermore, it is possible to choose which type of dataflow the systolic array in Gemmini will support: weight-stationary or output-stationary, or both.

Gemmini is designed to be tightly coupled to a RISC-V [Waterman et al. 2016] processor. RISC-V is an open Instruction Set Architecture (ISA) which supports new extensions, and custom instructions. A RISC-V processor communicates with Gemmini by issuing well-defined custom instructions. To easier the accelerator programming, software libraries written in C are generated with a Gemmini hardware, matching the custom hardware's parameters. The libraries provide an API to access the accelerator, avoiding the burden of programming in assembly to control the hardware. Due to its tightly coupled nature, Gemmini's performance may depend not only on its hardware parameters, but also on the tightly coupled processor parameters, such as cache size [Genc et al. 2019].

3. Methodology

We use a single direct convolution program as benchmark[2], which can be executed either in Gemmini or in a conventional CPU. For each benchmark execution, we vary input tensor's parameters *batch size* and *in dim size* which represents the number of rows and columns (D_I) as discussed in Section 2.1. Thus, the output dimension (D_O) will change according to Equation 2. In the first set of experiments, we fix *in dim* to 56 while varying *batch size* from 1 to 10. Then, we vary *in dim* from 10 to 100 while fixing *batch size* to 4. We choose both fixed values to be similar to common values found in convolution layers [Howard et al. 2017].

For each simulation we measure total execution time, and energy required by the CPU system. Since Gemmini is provided as a Verilog description, the number of simulated cycles is measured and then converted to time considering synthesis frequency. We choose Gemmini's energy consumption and working frequencies using data of system number 7 in [Genc et al. 2019, Table 1] since its parameters are the closest to those present in the Gemmini system used in this work. Since Gemmini is a generator, and thus can be applied into many application domains, e.g., embedded computing where lower operating frequency may be required, we evaluate the minimum Gemmini necessary frequency to match the same runtime obtained by CPU. Equation 5 shows how this can be calculated, where C_{gem} is the number of cycles to run the benchmark on Gemmini and t_{cpu} is the total execution time of the same benchmark on a CPU.

$$F_{min} = \frac{C_{gem}}{t_{cpu}} \tag{5}$$

We evaluate the CPU system in two different scenarios to assess the impact of processor load. In the first scenario, named **CPU-LL**, the benchmarks are executed in a lightly loaded CPU, that is, no demanding process is executed in background. In the second scenario, named **CPU-HL**, we run CPU-bound processes to keep the rest of CPU cores busy. Since CPU benchmarks runs in a full system environment, which may impact the results, we run each benchmark 500 times, and then consider the mean value of each execution as the final result. The computer system used to run the benchmarks consists of a Ubuntu 16.04 OS and a Intel i5-4460 CPU (four physical cores). To stress the processor in CPU-HL system, we use the *stress* tool which launches CPU-bound processes, making the rest of the cores busy. The processor a 3 GHz along all experimentation process. Energy metric is measured using the Intel Performance Counter Monitor (PCM) [opcm], which measures CPU socket's energy consumption.

[2]Compiled with default baremetal parameters when not noticed. More details about the source code can be found in Section 7.

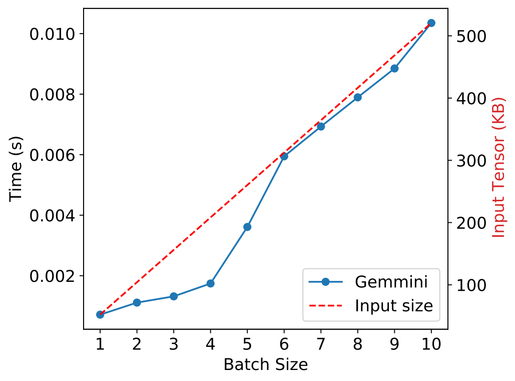
The Gemmini system used is the **GemminiRocketConfig** available in Chipyard [Amid et al. 2020], which is a framework to generate custom SoCs. The default system consists of a 5-stage in-order Rocket core [Asanović et al. 2016] tightly coupled to Gemmini. The accelerator is composed of 1 tile with 16x16 8-bit integer PEs. Besides that, the scratchpad and accumulator sizes are 256KB and 64KB, respectively. Since Chipyard provides the generated hardware as a Verilog code, we use the open-source Verilog simulator Verilator to run the benchmarks. Differently from the CPU systems, each Gemminni simulation is executed only once since the convolution program runs on top of a baremetal platform and the Verilator is a cycle-accurate simulator. For the reasons mentioned previously, we consider Gemmini system power consumption as 568.23 mW and working frequency as 500 MHz.

4. Results

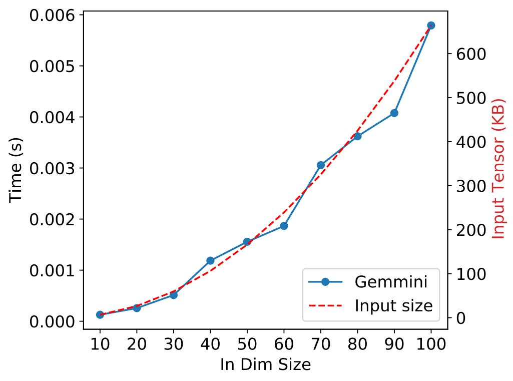
Figures 5.a to 5.d show execution when varying batch size and in dim size respectively to each platform. In these cases, execution time grows with respect to input size and in the same proportion, as expected when analyzing Equation 3. In fact, running an application in an accelerator does not change the algorithm complexity, but optimize its execution. By analyzing Figures 5.a and 5.b, it is noticeable the steep increase in execution time when batch size is 5 and in dim size is 70, which occurs due to input tensor's size being greater than the scratchpad size (256 KB). Figures 5.c and 5.d show the execution times for the CPU systems where the errors bars represent the standard deviation of the measurements. It is noticeable that CPU-HL not only took longer to finish, but also had a more unpredictable execution time in comparison with CPU-LL. Figures 5.e and 5.f depict the differences in execution time by comparing both systems. When considering batch size equal to 10, Gemmini was 4.9 times faster than CPU-LL. Time difference is more noticeable when analyzing in dim growth. Gemmini achieved 10.5 times less execution time than CPU-LL when considering in dim equal to 100.

Figure 6 shows energy consumption when running the benchmarks, which is directly proportional to execution time. As expected, Gemmini can run the same application with much lower energy consumption. The execution time and energy consumption gap is even higher when comparing Gemmini with a heavily loaded CPU since all cores are performing computation. Gemmini's big energy advantage over the CPU systems is explained by three reasons. First, Gemmini is designed to this specific niche of applications. Second, even though the Gemmini system is composed of the accelerator and the Rocket processor, the entire system can be considered as an embbeded computing platform since the CPU is a simple in-order core. Third, the power value adopted for Gemmini considers a fabrication process using TSMC 16 nm FinFET technology against the 22 nm process used in i5-4460, thus granting Gemmini a technology node advantage.

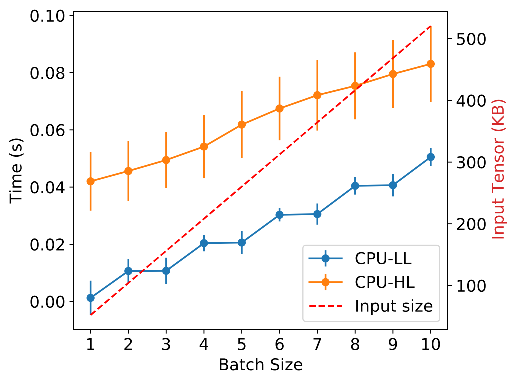
Hitherto, we only considered Gemmini working at 500 MHz. However, Gemmini's flexibility allows it to be used in many application domains, e.g. IoT devices, which lower operation frequencies may be required. Figure 7 shows Gemmini's system necessary frequency to match execution times of both CPUs scenarios. Overall, Gemmini can achieve the same execution time while working in much lower frequencies. When comparing both plots, batch growth requires higher frequencies than in dim growth. Thus, Gemmini's performance scales better when increasing the number of lines and columns of input tensor than the number of batches. When analyzing the CPU-LL behavior in Figure



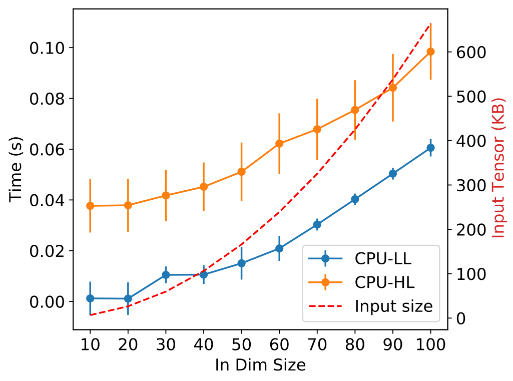
(a) Gem: Batch.



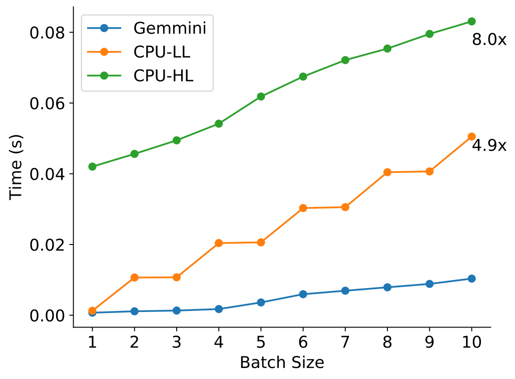
(b) Gem: In Dim.



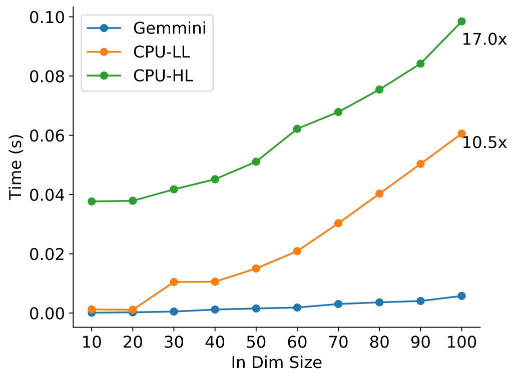
(c) CPU: Batch.



(d) CPU: In Dim.



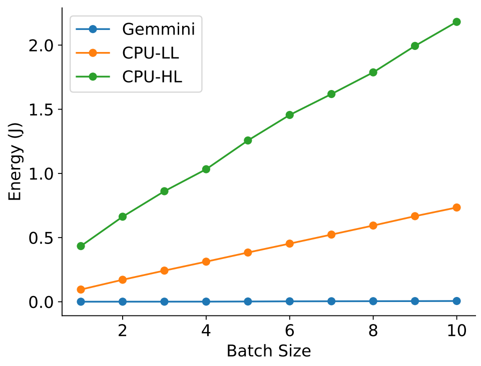
(e) Comparison varying Batch size.



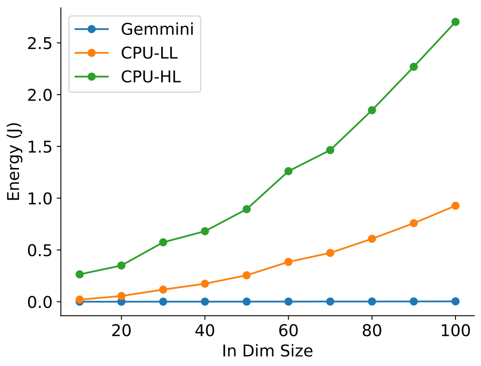
(f) Comparison varying In Dim size.

Figure 5. Execution time comparison between Gemmini and CPU systems.

7.a, it is noticeable that for some values of batch size (2, 3, and 4), the minimum frequency reaches its minimum value before increasing again. This indicates that Gemmini achieves its best performance in comparison with the CPU-LL around these values. In an analogous manner, Gemmini's worst result happens when in dim is equal to 20 in Figure 7.b, indicating that Gemmini's runtime for this point is closer to CPU-LL's execution time.

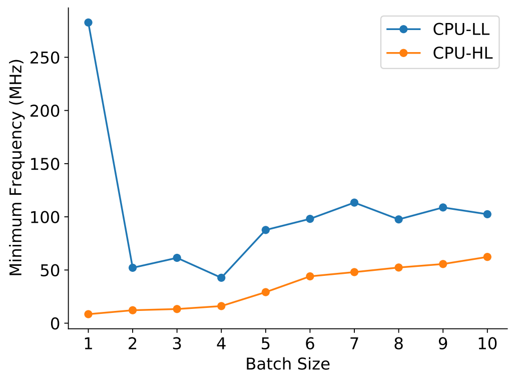


(a) Batch from 1 to 10.

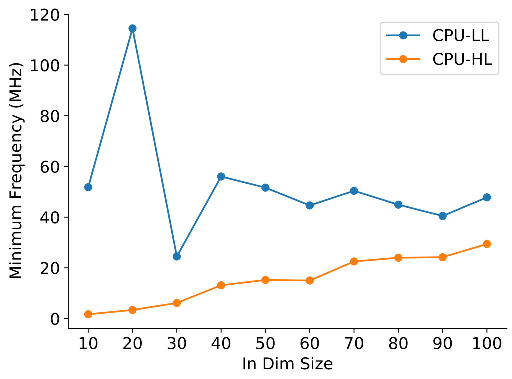


(b) In Dim from 10 to 100 with step of 10.

Figure 6. Energy comparisons.



(a) Batch from 1 to 10.



(b) In Dim from 10 to 100 with step of 10.

Figure 7. Minimum Gemmini frequencies to reach CPU execution time.

5. Related Work

In this section we discuss about other hardware accelerators, convolution algorithms and a tool to find the best hardware parameters to run a NN application.

DNN accelerators: To leverage computation-bound operations such as the convolution routine described in Listing 1, many accelerators were proposed, which exploit data reuse techniques and different dataflows. NVDLA [Zhou et al. 2018] is an open-source accelerator which offers integration with development tools such as Caffe [Jia et al. 2014], facilitating deployment of CNNs. Eyeris [Chen et al. 2016, Chen et al. 2017] proposes the Row-Stationary dataflow, which optimizes all types of data movement by maximizing the usage of the storage hierarchy, i.e., registers inside a PE, inter-PE communication, and global buffer access.

Hardware design space exploration: MAESTRO [Kwon et al. 2019] allows designers to leverage reuse opportunities, and to find optimal design points when tunning accelerator parameters. The tool receive as input a generic description of the accelerator and a description of a DNN model. Then, MAESTRO sweeps the design space to find a throughput-or-energy-optimized design. As output, the tool indicates the best accelerator's parameters to run the given DNN model.

Convolution algorithms: Besides the standard direct convolution presented in Equation 3, it is possible to reduce the computational cost by factorizing the standard convolution into depthwise and pointwise convolutions [Howard et al. 2017]. This approach can drastically reduce computation at a small reduction in accuracy. The Indirect Convolution algorithm [Dukhan 2019] is an alternative to GEMM-based convolution algorithms due to elimination of im2col transformations. Furthermore, it replaces the im2col buffer with a smaller indirection buffer. Nevertheless, it is optimized for NHWC layout, and has limited applicability to backward pass of convolution operator.

6. Conclusion

This work studied execution behavior of small convolutions in a lightly and heavily loaded CPU, and in Gemmini. Overall, Gemmini can achieve one order of magnitude less execution time than a conventional lightly loaded CPU when comparing a single convolution execution, and even better results can be achieved when comparing with a heavily loaded CPU. We show that Gemmini could run in much lower frequencies to match CPU systems execution times. The difference between the compared platforms could be greater when evaluating complete CNN algorithms, as claimed in previous work [Genc et al. 2019].

As future work, different matrix operations performance could be studied on Gemmini, e.g., matrix transposition and multiplication. Another possibility is to explore Gemmini's performance in a warehouse computing scenario as done with a Tensor Processing Unit (TPU) in Jouppi et al [Jouppi et al. 2017].

7. Reproducing the Results

We provide a git repository which contains the code and scripts to reproduce the results of this work available at: `https://gitlab.com/cravieira/wscad2020`.

8. Acknowledgment

This study was financed by the Cordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001 and the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

References

[Amid et al. 2020] Amid, A., Biancolin, D., Gonzalez, A., Grubb, D., Karandikar, S., Liew, H., Magyar, A., Mao, H., Ou, A., Pemberton, N., Rigge, P., Schmidt, C., Wright, J., Zhao, J., Shao, Y. S., Asanović, K., and Nikolić, B. (2020). Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21.

[Asanović et al. 2016] Asanović, K., Avizienis, R., Bachrach, J., Beamer, S., Biancolin, D., Celio, C., Cook, H., Dabbelt, D., Hauser, J., Izraelevitz, A., Karandikar, S., Keller, B., Kim, D., Koenig, J., Lee, Y., Love, E., Maas, M., Magyar, A., Mao, H., Moreto, M., Ou, A., Patterson, D. A., Richards, B., Schmidt, C., Twigg, S., Vo, H., and Waterman, A. (2016). The rocket chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley.

[Chen et al. 2016] Chen, Y., Emer, J., and Sze, V. (2016). Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 367–379.

[Chen et al. 2017] Chen, Y., Krishna, T., Emer, J. S., and Sze, V. (2017). Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138.

[Cong and Xiao 2014] Cong, J. and Xiao, B. (2014). Minimizing computation in convolutional neural networks. In Wermter, S., Weber, C., Duch, W., Honkela, T., Koprinkova-Hristova, P., Magg, S., Palm, G., and Villa, A. E. P., editors, *Artificial Neural Networks and Machine Learning – ICANN 2014*, pages 281–290, Cham. Springer International Publishing.

[Dukhan 2019] Dukhan, M. (2019). The indirect convolution algorithm. *CoRR*, abs/1907.02129.

[Genc et al. 2019] Genc, H., Haj-Ali, A., Iyer, V., Amid, A., Mao, H., Wright, J., Schmidt, C., Zhao, J., Ou, A., Banister, M., Shao, Y. S., Nikolic, B., Stoica, I., and Asanovic, K. (2019). Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures. *ArXiv*, abs/1911.09925.

[Howard et al. 2017] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861.

[Jia et al. 2014] Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R. B., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. *CoRR*, abs/1408.5093.

[Jouppi et al. 2017] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P.-l., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, C. R., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Killebrew, D., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Ross, M., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 1–12, New York, NY, USA. Association for Computing Machinery.

[Krizhevsky et al. 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Kung 1982] Kung, H. T. (1982). Why systolic architectures? *Computer*, 15(1):37–46.

[Kwon et al. 2019] Kwon, H., Chatarasi, P., Pellauer, M., Parashar, A., Sarkar, V., and Krishna, T. (2019). Understanding reuse, performance, and hardware cost of dnn dataflow: A data-centric approach. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 754–768, New York, NY, USA. Association for Computing Machinery.

[LeCun et al. 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521:436–44.

[Lecun et al. 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[NVIDIA 2020] NVIDIA (2020). Nvidia a100 tensor core gpu architecture.

[opcm] opcm. Processor counter monitor. Available: `https://github.com/opcm/pcm`.

[Vanhoucke et al. 2011] Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*.

[Waterman et al. 2016] Waterman, A., Lee, Y., Patterson, D. A., and Asanović, K. (2016). The risc-v instruction set manual, volume i: User-level isa, version 2.1. Technical Report UCB/EECS-2016-118, EECS Department, University of California, Berkeley.

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