Performance of precision auto-tuned neural networks

Quentin Ferro, Stef Graillat, Thibault Hilaire, Fabienne Jézéquel

Performance of precision auto-tuned neural networks

Quentin Ferro[1], Stef Graillat[1], Thibault Hilaire[1], Fabienne Jézéquel[1,2]

[1]*Sorbonne Université, CNRS, LIP6, Paris, France*
[2]*Université Paris-Panthéon-Assas, Paris, France*
{quentin.ferro,stef.graillat,thibault.hilaire,fabienne.jezequel}@lip6.fr

Abstract—While often used in embedded systems, neural networks can be costly in terms of memory and execution time. Reducing the precision used in neural networks can be beneficial in terms of performance and energy consumption. After having applied a floating-point auto-tuning tool, PROMISE, on various neural networks, we obtained versions using lower precision while keeping a required accuracy on the results. In this article, we present results regarding the memory and computation time gains obtained thanks to reduced precision, using vectorized and non-vectorized code. We also show the impact on the execution time of PROMISE of the parallelization of the Delta Debug algorithm it implements.

Index Terms—Precision, Neural Networks, Auto-Tuning, Floating-Point, Stochastic Arithmetic

I. Introduction

Neural Networks (NNs) are nowadays widely used and becoming larger and larger. Their requirements in terms of resource can be a problem when used in a critical situation, such as in embedded systems with limited power and memory. Therefore, it can be beneficial to optimise the numerical formats used in a neural network. To do so, we use the auto-tuning tool PROMISE[1] [1]. From a given neural network and a required accuracy on its results, PROMISE provides a mixed precision program. In this article, we show the impact of mixed or reduced precision on memory usage and execution time in programs provided by PROMISE. We also show how the execution time of the PROMISE tool itself can be reduced thanks to parallelization.

Because reduced or mixed precision offers advantages in terms of execution time, memory usage, and energy consumption, multiple mixed precision algorithms have been proposed, especially in linear algebra, as shown in the survey [2]. Designing such mixed precision algorithms requires a good knowledge of the computation involved.

Besides, precision auto-tuning tools aim at providing a mixed precision version of a program that satisfies accuracy requirements, whatever the implemented algorithms. In [3] a benchmark suite of programs is introduced for mixed precision computing analysis. Moreover the authors present the performance of various precision auto-tuning algorithms such as combinational (used in FloatSmith [4]), compositional (used in FloatSmith [4]), Delta Debug (introduced in [5], used in Precimonious [6] and PROMISE [1]), hierarchical

(used in CRAFT-HPC [7]), hierarchical-compositional (used in FloatSmith [4]) and a Genetic Search Algorithm (GA) (used in AMPT-GA [8]). The Delta Debug algorithm requires multiple executions of the user program to provide a suitable type configuration. To determine if a configuration is successful, various tests can be implemented. In PROMISE, that relies on rounding error estimation, the requirement is the result accuracy. The test in Precimonious is based on both a reference result computed in the highest precision and the execution time of the program obtained.

The contributions of this article are briefly described below.

- The memory consumption in the programs generated by PROMISE is analyzed, the type configurations in these programs depending on the required result accuracy.
- The execution time of the codes provided by PROMISE in mixed or reduced precision is analyzed. The executions with different type configurations are compared. Versions with and without SIMD vectorization are considered.
- To improve the performance of the search for a suitable type configuration, a parallel version of the Delta Debug algorithm is introduced in PROMISE. Its benefits on the execution time of PROMISE are analyzed.

Experiments are carried out on various neural networks including classification ones.

The outline of this article is as follows. Section II is a preliminary recall of the methodology described in [9] for the precision auto-tuning of neural networks. Sections III and IV analyse benefits in terms of respectively memory and execution time in the programs provided by PROMISE. Then Section V shows the performance gain in the PROMISE tool itself thanks to parallelization. Finally, concluding remarks and perspectives are given in Section VI.

II. Precision auto-tuning of neural networks using PROMISE

PROMISE (PRecision OptiMISE) is an auto-tuning tool which aims at reducing the precision of the variables in a given program. From an initial C/C++ code and a required accuracy on the result, it returns a mixed precision code, lowering the precision of the different variables while keeping a result that satisfies the accuracy constraint. To do so, some variables are declared as custom typed variables that PROMISE recognizes. PROMISE considers tweaking their precisions. Different variables can be forced to have the same precision by giving them
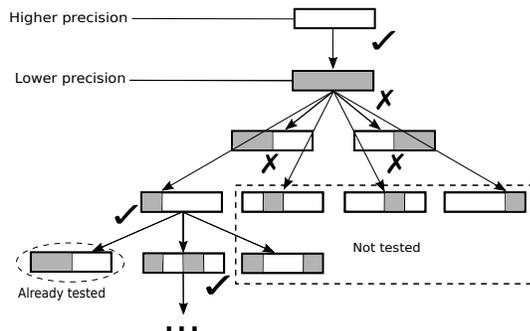
---

[1]http://promise.lip6.fr

Fig. 1: Creation of subsets by PROMISE. Subsets with single (resp. double) precision variables are represented in grey (resp. white).

the same custom type. It may be useful to avoid compilation errors or casts of variables.

PROMISE computes a reference result using the CADNA[2] library [10], [11] that enables one to estimate rounding errors in C/C++ or Fortran codes. CADNA implements Discrete Stochastic Arithmetic (DSA) [12], a probabilistic method to control rounding errors. With DSA, each arithmetic operation is performed three times with a random rounding mode. Then the accuracy can be estimated from a comparison of the three results obtained. CADNA introduces new numerical types named *stochastic types*. Each stochastic variable contains three floating-point values and one integer being the exact number of correct digits. CADNA can print each computed value with only its exact significant digits estimated thanks to DSA with a confidence level of 95%. In practice, owing to operator overloading, the use of CADNA only requires to change declaration of variables and input/output statements.

PROMISE relies on the Delta Debug algorithm [5] to test different type configurations, until a suitable one lowering the precision while satisfying the accuracy requirement is found. The Delta Debug algorithm does not perform an exhaustive search: it has a mean complexity of $O(n \log(n))$ for n variables. Figure 1 shows how PROMISE creates suitable configurations mixing two types, e.g. single and double precision. PROMISE can also provide a transformed program mixing half, single, and double precision variables. Half precision can be either native on CPUs that support it or emulated using a library developed by C. Rau[3]. PROMISE dataflow is presented in Figure 2. After computing a reference result in double precision using CADNA, PROMISE tries to lower the precision of the variables from double to single precision, then from single to half precision, using twice the Delta Debug algorithm.

Like in [9], we consider the application of PROMISE on four different neural networks (NNs) briefly described below:

- Sine: 3 layers, densely-connected interpolation network approximating the sine function

- MNIST: 2 layers, densely-connected classification network based on the MNIST database
- CIFAR: 5 layers, convolutional classification network based on the CIFAR10 database
- Pendulum: 2 layers, densely-connected interpolation network approximating the Lyapunov function of an inverted pendulum (introduced in [13] and used in [14]).

These NNs are developed in Python using either Keras[4] or PyTorch[5]. For each NN, we save the data (weights and biases of each layer) of the trained model in HDF5 (Hierarchical Data Format)[6]. Then, a Python script reads those data, and implements the inference phase in a C++ program. The implementation in C++ allows us to apply PROMISE.

Two different approaches are considered for each NN: one type per neuron or one type per layer. Both approaches reduce the precision of the parameters, taking into account the accuracy required on the output. The approach per layer leads to a more uniform precision program, because when one neuron of a layer needs to be in a specific precision, the whole layer has to be in the same precision. However, this approach provides a suitable configuration faster than the one per neuron and can still be useful depending on the application, for example to obtain a first mixed precision configuration.

As we consider the inference phase, input values are chosen and provided to the neural network. It has been observed that with both approaches input values have actually a low impact on the type configurations obtained. Moreover, because changing the precision of the input has a low impact on the configurations obtained, the input remains in double precision except for the speed gain analysis described in Section IV.

As already pointed out in [9], the results of a classification network may not require a high accuracy. However, exhaustive tests have been performed, considering all possible numbers of correct digits in the results, at most from 1 to 15 in double precision.

Except if indicated otherwise, the following results have been obtained on a 2.80 GHz Intel Core i5-8400 CPU having 6 cores with 16 GB RAM.

III. MEMORY GAIN ANALYSIS

To analyse the possible memory gains thanks to mixed precision, we consider the four different neural networks already mentioned with the different type configurations obtained with PROMISE in [9].

A. Methodology

Various tools can analyse memory usage, either the memory exchanged during the execution or the total memory used by the program. In our case, a preliminary study using Valgrind's Massif tool[7] allows us to conclude that the theoretical values can be considered, i.e. counting the number of half, single and double precision variables in the program knowing their byte

---

[2]http://cadna.lip6.fr
[3]http://half.sourceforge.net

[4]https://keras.io
[5]https://pytorch.org
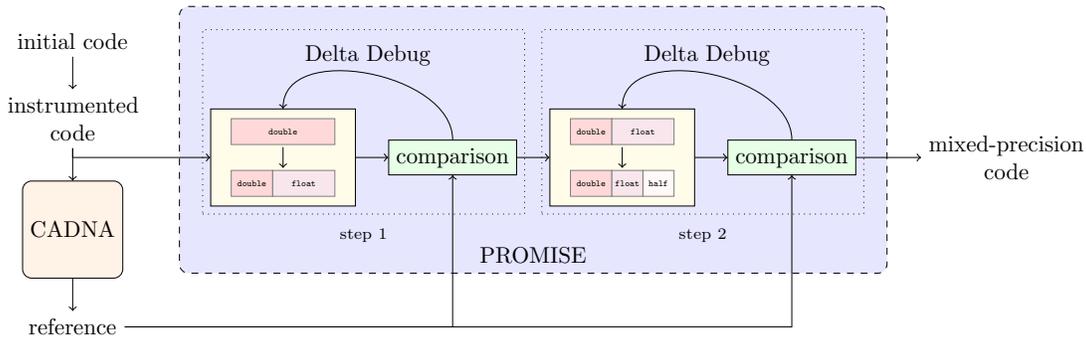[6]https://www.hdfgroup.org
[7]valgrind.org/docs/manual/ms-manual.html

Fig. 2: PROMISE dataflow

Fig. 3: Configurations and memory used for the Sine neural network with input value 0.5 and one type per neuron

Fig. 4: Configurations and memory used for the Sine neural network with input value 0.5 and one type per layer
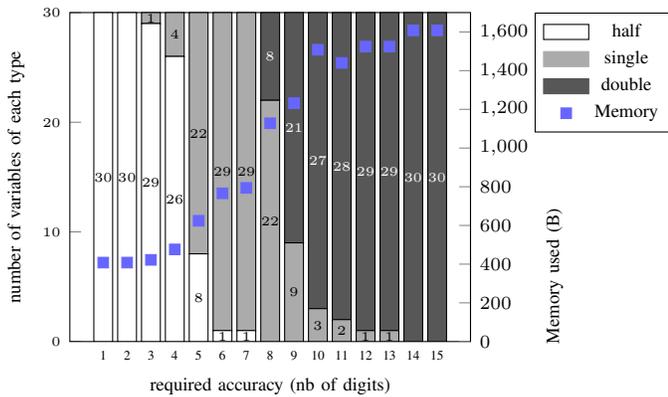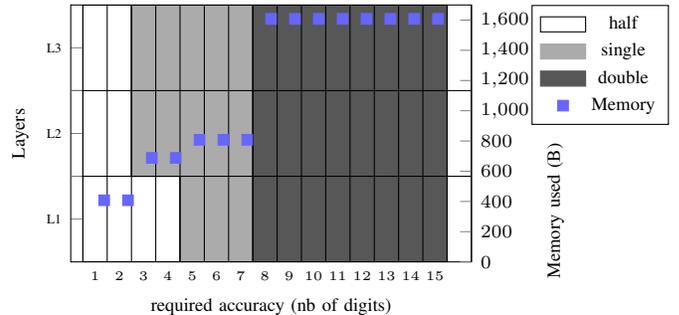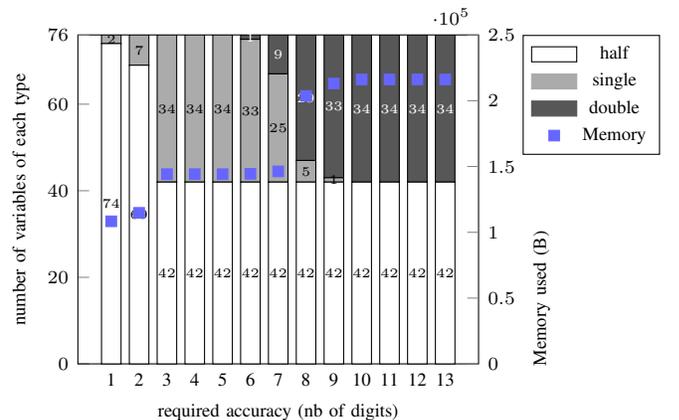

Fig. 5: Configurations and memory used for the MNIST neural network with input value test_data[61] and one type per neuron

size, instead of running Massif for each configuration. Indeed, the "useful-heap" values given by Massif are consistent with the theoretical values. Hence, the following results are the theoretical values considering the number of half, float and double precision variables in our program.

B. Results

Figures 3 to 10 present, in the programs provided by PROMISE for each neural network, both the number of variables of each type and the memory consumed, depending on the required accuracy. Figures 3, 5, 7, and 9 (respectively 4, 6, 8, and 10) have been obtained considering one type per neuron (respectively one type per layer).

As expected, when using lower precision the program requires less memory. The approach per layer tends to use more memory, gaps are often observed from one accuracy to another, because of the configurations obtained with this approach. Indeed when one variable of a layer needs to be in higher precision, the whole layer has to pass in higher precision.

The input data is mentioned in the caption of each figure. In particular, with MNIST NN the input image test_data[61] is the 62nd test data out of the 10,000 provided by MNIST, and with CIFAR NN, the input image test_data[386] is the 387th test data out of the 10,000 provided by CIFAR10. It has been

observed that input values have actually a slight impact on the type distributions and the memory consumption.

As a remark, while the theoretical values are relevant for our analysis, more memory is actually used by a program because of the overhead memory associated with every allocation. In our case it is negligible ($\approx 1\%$).
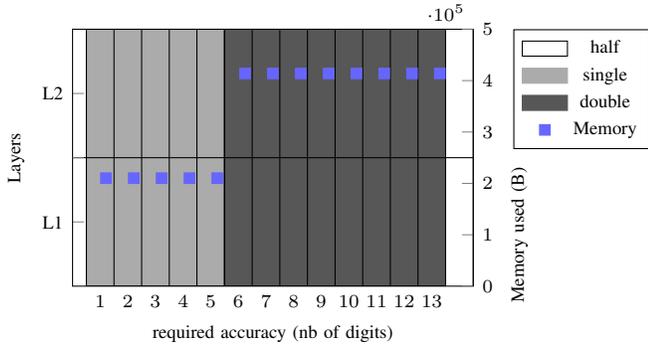
Fig. 6: Configurations and memory used for the MNIST neural network with input value test_data[61] and one type per layer
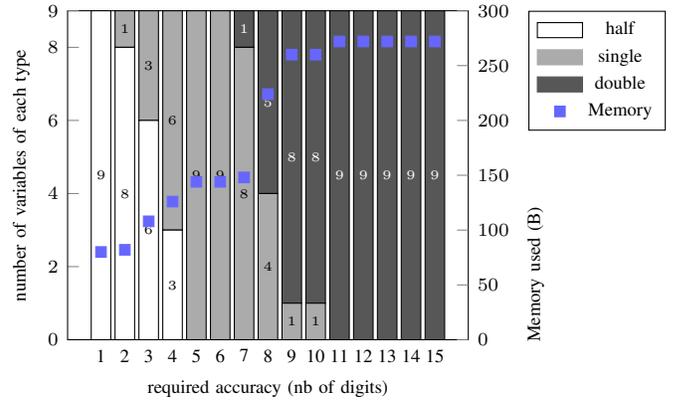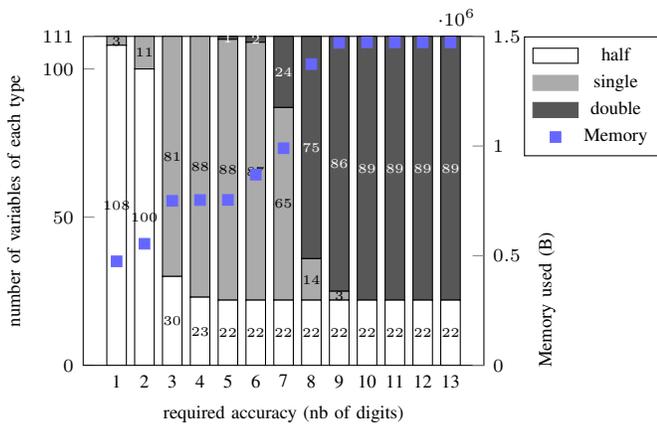

Fig. 7: Configurations and memory used for the CIFAR neural network with input value test_data[386] and one type per neuron
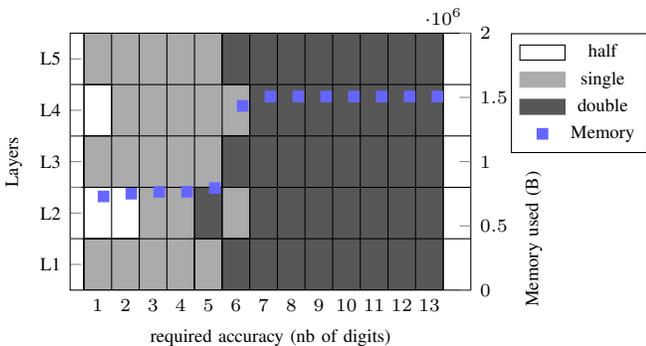

Fig. 8: Configurations and memory used for the CIFAR neural network with input value test_data[386] and one type per layer
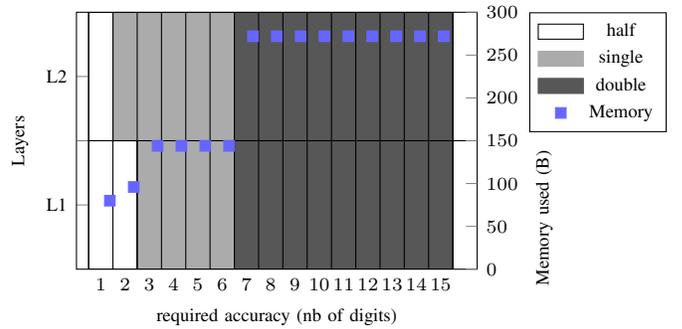

Fig. 9: Configurations and memory used for the Pendulum neural network with input value (0.5,0.5) and one type per layer

Fig. 10: Configurations and memory used for the Pendulum neural network with input value (0.5,0.5) and one type per layer

IV. Speed Gain Analysis

A. Methodology

We analyse the gain in execution time thanks to mixed precision in the configurations provided by PROMISE. For this analysis, in PROMISE we specify that configurations using single and double precision only must be provided, because half precision is not available in hardware in the architecture used in our experiments. We compare the time obtained with vectorized and non-vectorized codes, as most modern CPUs support SIMD instructions. To do so, we use OpenMP SIMD instructions and AVX2 vectorization on the loops that compute the different matrix-vector products.

For this performance analysis, the type of the input variables can be changed by PROMISE. Indeed, in our previous experiments, all input variables were in double precision and so all the arithmetic operations in the first layer were carried out in double precision, with possibly costly cast operations on the weights. Consequently, enabling the tuning of the input variables is beneficial for the execution time of the programs provided by PROMISE.

To generate the different configurations with vectorized code, we modify our translation script from NN model to

C++ program to add the appropriate OpenMP pragmas. We can then directly apply PROMISE on our vectorized C++ program. To enable vectorization, we use the flags `-fopenmp-simd -avx2`. We also compile with `-O3` option. We measure the different execution times using C++ time.h library, and considering the minimum value over 10,000 runs of the program.

Using AVX2, a vector unit is able to perform 8 operations in single precision at the same time, but only 4 in double precision. So a theoretical speedup of 2 should be observed when changing a vectorized code from double to single precision.

B. Results

We present here results on MNIST NN. Figures 11 and 12 display the different configurations and their execution time given the required accuracy with test_data[61] input, respectively with the approach per neuron and per layer. We recall that here the type of the input variable can also be changed.

First, for each configuration, we measure the execution time of the whole computation, i.e. the time of the *main* function in the code. As expected, differences can be observed between vectorized and non-vectorized codes. Furthermore the execution time depends on the precision used. As it can be observed in Figure 12, in the approach per layer, configurations in uniform precision can be obtained. Indeed, depending on the required accuracy all variables can be in single or in double precision. The speedup of a single precision execution w.r.t. a double precision one is up to 1.60 for non-vectorized codes and up to 1.86 for vectorized codes. The best speedup obtained is therefore slightly below the theoretical ratio of 2.

We can also compare mixed precision and double precision codes. We focus here on the mixed precision configuration obtained with a required accuracy from 10 to 13 digits and the approach per neuron (see Figure 11). The speedup of the mixed precision execution w.r.t. the double precision one is up to 1.30 for non-vectorized codes and up to 1.41 for vectorized codes. Nonetheless, if the configuration has a majority of double precision variables, the speedup can be worse. Indeed, in this case a significant number of operations are carried out in double precision and costly cast operations are performed.

When comparing vectorized and non-vectorized code in Figures 11 and 12, we observe a speedup of up to 1.45. This ratio is not in accordance with the theoretical speedup of vectorized codes w.r.t. non-vectorized ones on AVX2 units: 8 in single precision and 4 in double precision. This can be explained by the fact that a significant part of the code does not actually benefit from the vectorization. It is notably the case of the *softmax* activation function in the last layer, which computes an exponential using the C++ cmath library.

Therefore, we display in Figures 13 and 14 the execution time of only the matrix-vector products performed in the code. In this case, we observe a speedup of up to 7.2 when comparing vectorized and non-vectorized codes in single precision, and a speedup of up to 3.9 in double precision.
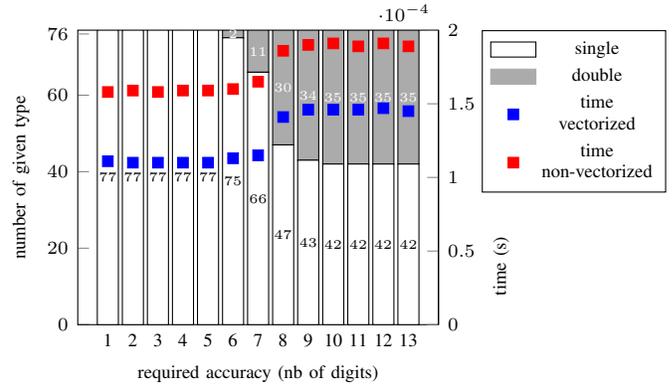

Fig. 11: Configurations and total execution time for MNIST neural network with test_data[61] input and one type per neuron
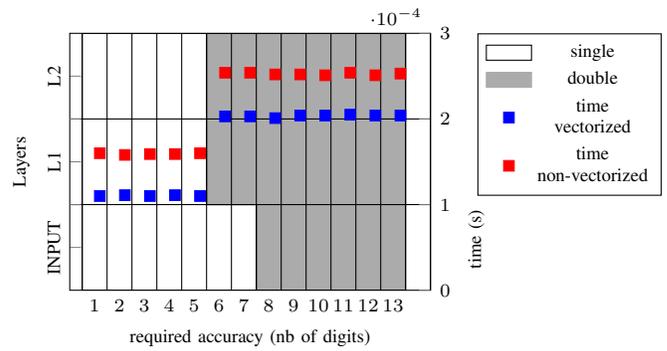

Fig. 12: Configurations and total execution time for MNIST neural network with test_data[61] input and one type per layer

When comparing single and double precision in vectorized codes, we have a speedup of up to 2, the theoretical ratio.

In Figure 14 we do not observe any time difference between the single precision and the double precision execution of the non-vectorized matrix-vector products. However, Figure 12 exhibits a ratio of up to 1.60 between the double precision and the single precision execution time of the non-vectorized code.
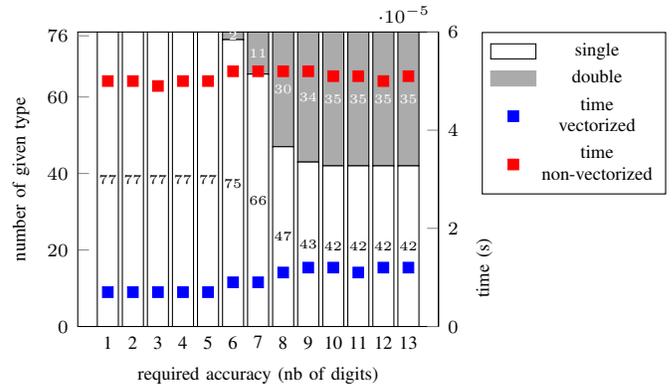

Fig. 13: Configurations and execution time of matrix-vector products with test_data[61] input and one type per neuron
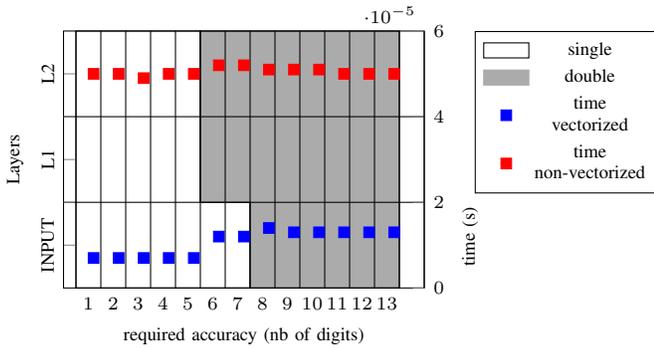
Fig. 14: Configurations and execution time of matrix-vector products with test_data[61] input and one type per layer

This speedup is due to the benefits from using single precision when declaring and initializing variables (in our case all the weights and biases). In this part of the code, the speedup of the single precision execution w.r.t. the double precision one is actually up to 2, the theoretical ratio.

V. PROMISE PERFORMANCE IMPROVEMENT

While Section IV was analyzing the performance of the codes generated by PROMISE, this section is dedicated to the performance improvement of the PROMISE tool itself. The search for a suitable configuration by the Delta Debug algorithm requires multiple compilations and executions of mixed precision versions of the user code. Therefore we have parallelized the Delta Debug algorithm inside PROMISE. We use the parallelization described in [15] and the related implementation in the tool Picire[8]. This parallelization relies on Python multiprocessing module that allows one to execute a function across multiple input values, distributing the data in different processes in parallel.

As shown in Figure 1, Delta Debug is a multi-level algorithm that passes subsets of variables in lower precision and determines whether the associated configurations satisfy the accuracy requirement on the result. The idea of the parallelization is to test multiple configurations in parallel at one level of the Delta Debug algorithm. When a configuration matches our requirement, we can stop looping over the different configurations. The number of configurations to be tested in parallel can be specified. In our case, we test 6 configurations in parallel, because 6 cores are available on our machine.

Figures 15 and 16 show the PROMISE runtime for MNIST and CIFAR using the sequential and the parallel version of the Delta Debug algorithm. We use here the approach per neuron, more costly than the one per layer. For MNIST we observe a speedup of up to 3.2, and for CIFAR a speedup of up to 2.7.

As a remark, for MNIST, if 1 or 2 digits on the result are required, the sequential algorithm performs better than the parallel one (see Figure 15). For CIFAR, when 1 digit is required, the sequential and parallel execution times are
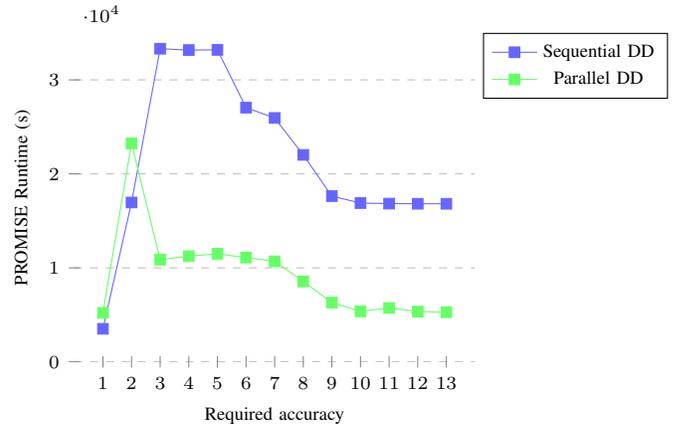
[8]https://github.com/renatahodovan/picire

Fig. 15: Computing time of PROMISE using the different Delta Debug (DD) versions on MNIST NN with test_data[61] input
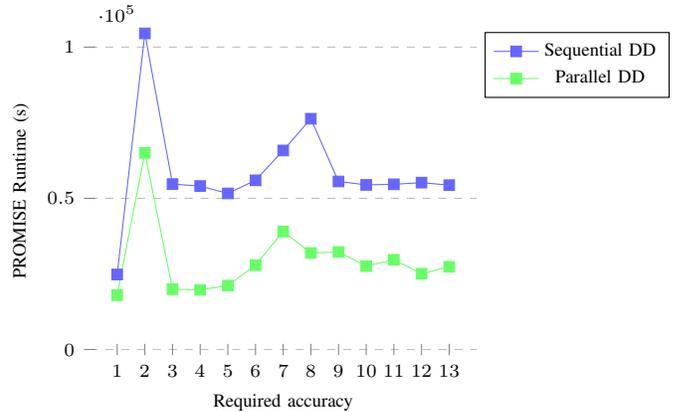

Fig. 16: Computing time of PROMISE using the different Delta Debug (DD) versions on CIFAR NN with test_data[386] input

very close (see Figure 16). Indeed the parallel version may not be satisfactory when requiring low accuracy on the result, because in this case the Delta Debug algorithm performs only a few steps and is affected by the parallelization overhead.

VI. CONCLUSION AND PERSPECTIVES

Thanks to the application of auto-tuning on different neural networks, we have shown that lowering the precision of floating-point variables can have valuable benefits in terms of memory consumption and execution time. Significant memory gains could be deduced from the length of each numerical format. Indeed, single precision (respectively half precision) enables one to consume half (respectively four times) less memory than double precision. For time measurements, PROMISE has been used in order to generate codes mixing two precisions (single and double) available in hardware for our experiments. Using single precision is beneficial for declaration and initialization of variables, and particularly advantageous in vectorized parts of the codes. PROMISE

relies on the Delta Debug algorithm that tests multiple type configurations in the user code. We have shown the impact of the parallelization of the Delta Debug algorithm on the execution time of PROMISE.

In the future, we plan to extend to different architectures the analysis of the impact of mixed precision on performance. The performance of mixed precision codes provided by PROMISE could be measured on AVX512 SIMD units. Furthermore the performance of codes generated by PROMISE possibly mixing half, single, and double precision could be analysed on architectures with native half precision, such as some ARM CPUs or recent GPUs. We also plan to extend PROMISE to other numerical formats, such as bfloat16.

A challenging perspective consists in proposing algorithms with a reasonable complexity for floating-point auto-tuning in arbitrary precision. This would enable one to automatically generate arbitrary precision codes for architectures such as FP-GAs (Field Programmable Gate Arrays). Moreover precision auto-tuning could benefit from research on mixed precision linear algebra. To reduce the number of variables to be considered, and consequently improve the performance of precision auto-tuning, linear algebra kernels could be automatically replaced by their mixed precision version.

REFERENCES

[1] S. Graillat, F. Jézéquel, R. Picot, F. Févotte, and B. Lathuilière, "Auto-tuning for floating-point precision with discrete stochastic arithmetic," *Journal of Computational Science*, vol. 36, p. 101017, Sep. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877750318309475

[2] N. J. Higham and T. Mary, "Mixed precision algorithms in numerical linear algebra," *Acta Numerica*, Jun. 2022. [Online]. Available: https://hal.science/hal-03537373

[3] K. Parasyris, I. Laguna, H. Menon, M. Schordan, D. Osei-Kuffuor, G. Georgakoudis, M. Lam, and T. Vanderuggen, "HPC-MixPBench: An HPC benchmark suite for mixed-precision analysis," 9 2020. [Online]. Available: https://www.osti.gov/biblio/1692269

[4] M. O. Lam, T. Vanderbruggen, H. Menon, and M. Schordan, "Tool integration for source-level mixed precision," in *2019 IEEE/ACM 3rd International Workshop on Software Correctness for HPC Applications (Correctness)*, 2019, pp. 27–35.

[5] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.

[6] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC'13. New York, NY, USA: ACM, 2013, pp. 27:1–27:12.

[7] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. Legendre, "Automatically adapting programs for mixed-precision floating-point computation," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 369–378.

[8] P. V. Kotipalli, R. Singh, P. Wood, I. Laguna, and S. Bagchi, "AMPT-GA: Automatic mixed precision floating point tuning for GPU applications," in *Proceedings of the ACM International Conference on Supercomputing*. Phoenix Arizona: ACM, Jun. 2019, pp. 160–170. [Online]. Available: https://dl.acm.org/doi/10.1145/3330345.3330360

[9] Q. Ferro, S. Graillat, T. Hilaire, F. Jézéquel, and B. Lewandowski, "Neural network precision tuning using stochastic arithmetic," in *NSV'22, 15th International Workshop on Numerical Software Verification,*, Haifa, Israel, Aug. 2022. [Online]. Available: https://hal.science/hal-03682645

[10] P. Eberhart, J. Brajard, P. Fortin, and F. Jézéquel, "High performance numerical validation using stochastic arithmetic," *Reliable Computing*, vol. 21, pp. 35–52, 2015.

[11] F. Jézéquel, S. s. Hoseininasab, and T. Hilaire, "Numerical Validation of Half Precision Simulations," in *Trends and Applications in Information Systems and Technologies*, Á. Rocha, H. Adeli, G. Dzemyda, F. Moreira, and A. M. Ramalho Correia, Eds. Cham: Springer International Publishing, 2021, vol. 1368, pp. 298–307, series Title: Advances in Intelligent Systems and Computing. [Online]. Available: http://link.springer.com/10.1007/978-3-030-72654-6_29

[12] J. Vignes, "A Stochastic Arithmetic for Reliable Scientific Computation," *Mathematics and Computers in Simulation*, vol. 35, no. 3, pp. 233–261, Sep. 1993. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/037847549390003D

[13] Y.-C. Chang, N. Roohi, and S. Gao, "Neural lyapunov control," *33rd Conference on Neural Information Processing Systems (NeurIPS 2019)*, Dec. 2020, arXiv: 2005.00611. [Online]. Available: http://arxiv.org/abs/2005.00611

[14] C. Lauter and A. Volkova, "A framework for semi-automatic precision and accuracy analysis for fast and rigorous deep learning," *arXiv:2002.03869 [cs]*, Feb. 2020, arXiv: 2002.03869. [Online]. Available: http://arxiv.org/abs/2002.03869

[15] R. Hodován and Á. Kiss, "Practical Improvements to the Minimizing Delta Debugging Algorithm:," in *Proceedings of the 11th International Joint Conference on Software Technologies*. Lisbon, Portugal: SCITEPRESS - Science and Technology Publications, 2016, pp. 241–248. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005988602410248