Wordlength Optimization for Custom Floating-point Systems

Quentin Milot, Mickaël Dardaillon, Justine Bonnot, Daniel Menard

Wordlength Optimization for Custom Floating-point Systems

Quentin Milot[1][0009−0008−9165−0821], Mickaël Dardaillon[1][0000−0001−6862−2090], Justine Bonnot[2][0000−0003−4975−8659], and Daniel Menard[1][0000−0002−5416−2393]

[1] Univ Rennes, INSA Rennes, CNRS, IETR - UMR 6164, F-35000 Rennes, France
first.last@insa-rennes.fr
[2] WedoLow, F-35510, Cesson-Sévigné, France
jbonnot@wedolow.com

Abstract. Algorithm complexity and problem size explosion during the last few years brought new challenges for hardware implementation of data-oriented applications. In particular, they have caused a large increase in memory, execution time, and power needed. One solution to address those challenges is to reduce data sizes. Custom floating-point is a good candidate that brings a large dynamic range in a compact representation. Thanks to improvement in floating-point units, it is more and more explored as an alternative to fixed-point.
This paper proposes an automatic optimization flow to optimize floating-point wordlength for a given quality metric. This flow is generic and can be used for any C or C++ algorithm. Different strategies are proposed to optimize the exponent and mantissa wordlengths. These strategies take advantages of analyzing the data dynamic range to reduce the optimization time. The obtained results show that better implementation cost can be obtained compared to 16-bit floating-point for a same quality.

Keywords: Custom floating-point · Wordlength · Optimization

1 Introduction

During the last years, with the technological progress in the semiconductor industry, digital platforms have integrated more and more transistors and have become faster and more energy efficient. With this evolution, numerous smarter systems are designed based on data-oriented applications. This technological progress leads to a significant improvement of the logical elements through the reduction of latency and dynamic energy. The energy to transport the data can now be higher than the energy to process the data by up to two to four orders of magnitude [4]. Memory access and data transportation became the new bottleneck for computation optimization. One way to face this issue is to optimize the data wordlength [6].

Two finite precision arithmetics are mainly used inside digital platforms to implement data-oriented applications: fixed-point and floating-point. The fixed-point arithmetic was favored in energy-efficient systems thanks to its lower cost.

Even if shift operations must be inserted between operations for scaling or aligning data, the processing part of fixed-point is less complex in terms of logical elements compared to floating-point. For fixed-point arithmetic, the quantization step (distance between two consecutive values) is constant due to the fixed implicit scaling factor. For floating-point arithmetic, the quantization step is adapted to the value to represent thanks to the explicit scaling factor embedded in the floating-point data. Thus, floating-point arithmetic can better compress data than fixed-point *i.e.* for a similar quantization error, fewer bits are required for encoding a floating-point data than for a fixed-point data.

However, with the reduction of floating-point's computing energy, especially compared to memory access energy [6], custom floating-point data types becomes a new contender for energy-efficient systems integrating data-oriented applications. In contrast to 64, 32 and 16-bit standard floating-point data types, customizing floating-point data requires an in-depth knowledge of the data dynamic range and application accuracy to adapt the wordlength to the need of each data. To obtain an optimized cost, the wordlength of the mantissa and the exponent must be adapted for each data inside an application. To face the complexity of modern applications, and to limit the development time, automatic frameworks to convert an application using standard floating-point data types to custom floating-point data types are mandatory.

Different methodologies and frameworks [2] have been proposed for fixed-point conversion. This conversion is composed of two main steps. First, the number of bits for the integer part is determined from the dynamic range analysis. Secondly, the number of bits for the fractional part is determined. The total number of bits is optimized such as the implementation cost is minimized subject to a quality criterion on the application output. In floating-point arithmetic, the problem is more complex because the accuracy depends on the mantissa wordlength but also of the exponent wordlength. Wordlength optimization for custom floating-point arithmetic has been proposed for specific applications like deep learning [13, 10, 8]. Generic approaches have been proposed for custom floating-point system design [3, 12]. However, the strategy to optimize the wordlength of both the exponent and mantissa is not considered in these approaches.

In this paper, a new method for custom floating-point refinement is proposed. Different strategies are proposed to optimize the exponent and mantissa wordlengths. These strategies take advantage of analyzing the data dynamic range to reduce the optimization time. The obtained results show that better implementation costs can be obtained compared to standard 16-bit floating-point for a same quality.

The rest of the paper is organized as follows. Section 2 presents previous works on finite precision conversion. The proposed finite precision conversion flow is then described in Section 3. The determination of the exponent wordlength from dynamic range analysis is presented in Section 4. The optimization strategies specified for custom floating-point are detailed in Section 5. Section 6 presents the experiment results on different use cases.

2 Background and State of the art

2.1 Finite precision arithmetic

Two finite precision arithmetics are mainly used to implement data-oriented applications: fixed-point and floating-point. Fixed-point arithmetic allows representing real values with the help of integer arithmetic. A fixed-point value x_{FxPt} is composed of two parts corresponding to the integer and the fractional part. x_{FxPt} is obtained by multiplying an integer x_{int} by a constant scaling factor 2^{-n}, where n corresponds to the fractional part wordlength.

Contrary to fixed-point arithmetic, for which the scaling factor is implicit and not embedded in the data, in floating-point arithmetic, the scaling factor is explicit and defined through the exponent field e. A normalized floating-point value x_{FlPt} is composed of three parts corresponding to the sign s, the exponent e and the mantissa m. This exponent e is embedded in the encoded data and allows selecting the range by representing the $\log_2$ of the number x to represent. Thus, with floating-point data types, very small and very high values can be represented with a good relative accuracy. The mantissa part m refines the value inside the considered range.

To avoid multiple representations of a given value and to allow interoperability between digital platforms, standards, like the IEEE 754 norm have been proposed for defining floating-point data type for 64, 32 and 16 bits. A normalized floating-point data type is under discussion for 8-bit data [11, 7]. This standard defines the exponent and mantissa wordlengths, w_E and w_M respectively [5].

The exponent e_c embedded in the floating-point data is an unsigned integer value. To consider negative exponents to represent small values, a bias Δ equal to half of the exponent range is defined with the following expression:

$$e = e_c - \Delta \qquad \text{with} \qquad \Delta = 2^{w_E - 1} - 1$$

where w_E corresponds to the number of bits to represent the exponent. The minimal and maximal exponent values are used to represent specific values (0, denormalized values, $\pm\infty$, NaN).

In hardware design, like for fixed-point arithmetic, optimizing the data wordlength and using its own wordlength for each data will reduce the implementation cost (area, energy, latency). For custom floating-point data, reducing the exponent wordlength will impact the range of the data that can be represented. Reducing the mantissa wordlength will impact the accuracy [10]. The challenge is to provide automated design tools to convert a C source code into a C++ code using custom floating-point data types. The data wordlength search space must be explored efficiently in order to reduce the implementation cost for a maximal application quality degradation.

2.2 State of the art

The challenge of creating design tools for custom floating-point data type has been explored through multiple point of view in the literature.

Domain-specific quantization Domain-specific quantization takes advantage of domain particularities to improve quantization. Neural network quantization is a good example, with current interest in machine learning driving a lot of works in both quantization-aware training and post-training quantization. In quantization-aware training, data wordlengths are added as parameters to optimize during training [8]. It improves quantization robustness by taking into account quantization noise during training, but the method is specific to machine-learning. Post-training quantization is more generic as it can be applied outside of the domain of machine learning. It relies on the algorithm designer to set an objective. These approaches usually optimize a wordlength per layer, and use standard floating-point data types based on the targeted architecture [13]. In [10], custom floating-point data types are used. The exponent and mantissa wordlength are optimized.

Bitsize [3] The Bitsize framework aims at optimizing the wordlength for fixed-point and floating-point data types. This framework uses an analytical approach for quantization error evaluation. The error model is efficient in terms of error evaluation time but limited to smooth operations. An operation is considered as smooth if the output is a continuous and differentiable function of the inputs.

Minibit+ [12] Minibit+ is an increment of Minibit approach [9] dedicated to fixed-point optimization. The transformation focuses on non-uniform optimization and floating-point optimization. The flow is based on a C++ program input. It starts with a range analysis using affine arithmetic. It then performs a coarse-grained accuracy analysis followed by a fine-grained accuracy analysis to determine the data wordlength. The main limitation is its lack of support for control structures such as conditions and loops.

In the Bitsize and Minibit+, the strategy to optimize the wordlength of both the exponent and mantissa is not detailed.

3 Finite Precision Conversion Flow

The proposed framework aims at converting automatically a C or C++ code with standard floating-point data types (float or double) to a C++ source code with custom floating-point data types for which the data wordlength has been optimized according to an application quality criterion. This output source code can then be used with high-level synthesis tools such as Vitis or Catapult for hardware implementation.

To optimize applications described with C/C++ floating-point code, the proposed framework in Figure 1 starts with a source-to-source compiler [15]. This step analyzes the application source code to determine which variables in the application source code may have their precision optimized *i.e.*, all the variables involved in the application output computation. This first step also determines if some data types are set up by other ones, so as to reduce the design space

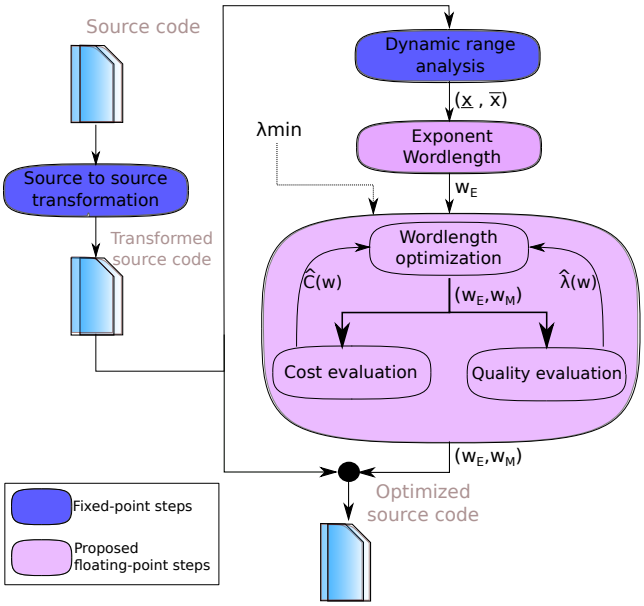


Fig. 1: Proposed framework for custom floating refinement.

to explore. Let $\mathcal{V}$ be a set grouping the N_v variables v_i to be optimized. At the output of this compiling step, the application source code is generated with generic data types that will be modified during the optimization process. This version of the application is then automatically instrumented by the framework to derive information on the variables belonging to the set $\mathcal{V}$.

The next step is a dynamic range analysis of each variable belonging to the set $\mathcal{V}$. Ranges are extracted based on a simulation running the testbench provided by the developer. The generic data types introduced in the first step are instrumented to collect all the values taken by a variable v during the source code execution and to build the histogram of $\log_2(v)$. To illustrate, an example of a histogram is given in Figure 2.

4 Exponent wordlength determination

The dynamic range associated with standard floating-point data types is predefined and depends on the exponent wordlength. Since this range is not custom for standard floating-point data types, several bits in the exponent field can be left unused, leading to supplementary implementation cost.

To obtain a suitable exponent wordlength, the data range needs to match with the representation for any mantissa wordlength w_M. A floating-point data having w_E bits for the exponent fields with standard bias Δ allows representing values in the range indicated in Equation 1 in the extreme case of mantissa

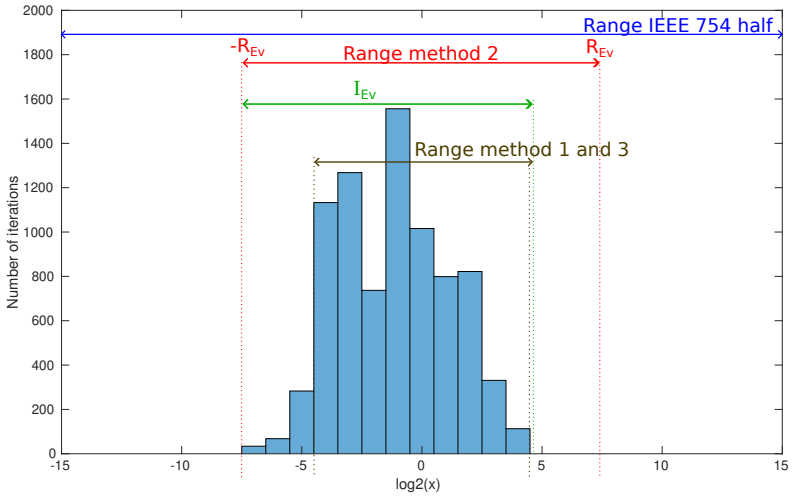


Fig. 2: Example of an histogram obtained from dynamic range analysis. Ranges for different floating-point data types are illustrated based on methods presented in Section 5.2.

wordlength $w_M = 0$.

$$[-2^{2^{w_E-1}-1}; -2^{-(2^{w_E-1}-2)}] \cup [2^{-(2^{w_E-1}-2)}; 2^{2^{w_E-1}-1}] \tag{1}$$

Let $\underline{v}$ and $\overline{v}$ be respectively the minimal and maximal values taken by the variable $|v|$ without considering the value 0. The exponent range I_{E_v} represents the values that the exponent needs to reach to handle all the values of v. Since the exponent part needs to be symmetrical with standard bias Δ and reach both values of I_{E_v}, the maximal range distance R_{E_v} determines the maximal wordlength $w_E^{\max}$. The maximal wordlength $w_E^{\max}$ needs to represent the value R_{E_v} and $-R_{E_v}$ and can be determined with the following expression:

$$I_{E_v} = [\log_2(\underline{v}), \log_2(\overline{v})], R_{E_v} = \max\left(|I_{E_v}|\right), w_E^{\max} = \lceil \log_2\left(R_{E_v}\right) \rceil + 1 \tag{2}$$

Let $\mathbf{w}_E^{\max} = \left[w_{E_0}^{\max}, \dots w_{E_i}^{\max} \dots w_{E_{N_v-1}}^{\max}\right]$ be a N_v-length vector storing for each variable v_i of $\mathcal{V}$, the maximal number of bits $w_{E_i}^{\max}$ for the exponent.

5 Wordlength optimization for custom floating-point

5.1 Wordlength optimization problem definition

Worldlength optimization aims at obtaining a minimal cost C satisfying a user-defined quality constraint λ_{min}.

$$\min_{\mathbf{w}}(\hat{C}(\mathbf{w})) \quad \text{subject to} \quad \hat{\lambda}(\mathbf{w}) \geq \lambda_{min} \tag{3}$$

With $\mathbf{w}$ being the concatenation of the vectors $\mathbf{w_E}$ and $\mathbf{w_M}$ representing the exponent and mantissa wordlengths respectively for each variable of the set $\mathcal{V}$.

Cost evaluation Different metrics like architecture area, energy consumption or memory space can be considered for the implementation cost C. For this study, an energy consumption and a memory space models are considered.

The dynamic energy of an operation depends on its type (addition, multiplication, memory transfer, ...) and the input and output wordlengths. For a given technology node for ASIC or for a given chip for FPGA, a library is obtained from a characterization process in which the operation costs are evaluated for different combinations of exponent and mantissa wordlengths. The energy consumption of the complete system is the sum of all the operation costs used in the system. For this study, a library for 28nm FDSOI technology is used [1]. This library provides the implementation cost for arithmetic operations but not for memory transfer, limiting the cost model to processing, excluding on-chip and off-chip memory. The memory cost aims at estimating the memory space to store the application variables. To do so, the number of allocations of a variable is determined and multiplied to the total wordlength of the variable.

Quality evaluation Simulation-based and analytical approaches can be considered to evaluate the quality metric λ at the output of an application according to the data wordlength $\mathbf{w}$. Analytical approaches aim at providing a mathematical expression of the quality metric. With this mathematical expression, the quality evaluation time is low but the supported applications are limited to those having only smooth operations.

Simulation-based techniques are more generic and support any application. The quality metric is statistically evaluated from data collected with a Monte-Carlo simulation carried out on the testbench input data. The source code is modified to integrate the custom floating-point data types. In this work, `ac_type` [14] are used. The confidence in the statistical estimation of the quality metric depends on the number of input samples used for the Monte-Carlo simulation. Thus, a huge number of samples can be required, leading to high quality evaluation time.

Optimization algorithm The space to explore being a subspace of natural numbers, the optimization needs to be heuristic. Numerous heuristic methods have been proposed to optimize wordlengths and especially in the context of fixed-point systems [2].

In this paper, the *min+1* algorithm [2] has been adapted for wordlength optimization problem in the context of custom floating-point. This algorithm is composed of two steps. First, an initial solution is found to start the greedy algorithm deployed in the second step. All the variables of vector $\mathbf{w}$ are set to their maximal value and one variable $\mathbf{w}_i$ is decreased until the quality constraint is no more fulfilled. The minimal value fulfilling the constraint is recorded in the vector $\mathbf{w}_{\min}$. This process is repeated for each variable of vector $\mathbf{w}$. In the second step, a mildest-ascent greedy algorithm is applied to increment by one bit one of the variables of vector $\mathbf{w}$ at each iteration. This second step starts with the initial solution $\mathbf{w}_{\min}$, for which the quality constraint is not fulfilled. At each

iteration a gradient is computed for each variable to find the best direction *i.e.* the variable for which a one-bit wordlength increment lead to the best quality improvement. The optimization algorithm stops when the quality constraint is fulfilled.

This optimization method needs to test numerous configurations. Let n_C be the number of tested configurations to obtain the final solution. Let n_S be the number of input samples required to obtain a quality evaluation with a given confidence interval. The time t_{Opt} required for this global optimization process can be determined with Equation 4. In this equation, t_S and t_{comp} represent the execution time for one sample and the compilation time of the modified source code to evaluate the quality.

$$t_{Opt} = n_C * (t_{comp} + n_S * t_S) \tag{4}$$

5.2 Global wordlength optimization strategies

In this section, the proposed strategies to optimize the wordlength of both the exponent and mantissa are detailed. As shown in Equation 4, the optimization time is proportional to the number of tested configurations. This latter depends on the optimization search space. The search space is linked to three parameters, N_v the number of variables to optimize, $\mathbb{E}_r \doteq \{\mathbf{x} \in \mathbb{N}^{N_v} : 1 \leq x_i \leq w_{E_i}^{\max}\}$ the search space for exponent wordlength and $\mathbb{M} \doteq \{\mathbf{x} \in \mathbb{N}^{N_v} : 1 \leq x_i \leq W_m\}$ the search space for the mantissa wordlength W_m represent the mantissa wordlength of the reference type (24 for single precision, 53 for double precision) and $w_{E_i}^{\max}$ the maximal number of bits required for the i^{th} variable according to the range analysis.

Exponent and mantissa wordlength simultaneous optimization This approach optimizes the wordlengths of the exponent and mantissa simultaneously. The maximal exponent wordlength is defined in vector $\mathbf{w}_E^{\max}$ computed with Equation 2 with the help of the range analysis step. The variable for the optimization process is a $2.N_v$-length vector $\mathbf{w}$ composed of the exponent wordlength w_{E_i} and the mantissa wordlength w_{M_i} for each variable v_i of the set $\mathbb{V}$. The domain to explore is then $\mathbb{E}_r * \mathbb{M}$. The number of elements n_p of the search space can be expressed with the following expression:

$$n_p = M^{N_v} * \prod_{i=0}^{N_v} w_{E_i}^{\max} \tag{5}$$

By considering the exponent wordlength as a variable to optimize, compared to the approaches described in following sections , this approach provides more degrees of freedom to find an optimized solution. Indeed, values with a small exponent can potentially be approximated without degrading too much the application quality λ. Furthermore, for these values, the unnormalized representation can help to represent a part of the smallest values. To optimize this unnormalized representation, both the mantissa and exponent need to be taken into account.

The use of the reduced space $\mathbb{E}_r$, instead of a space composed of the maximal exponent wordlength ,allows reducing the optimization time. Indeed, the values higher than $w_{E_i}^{\max}$ will lead to the same quality results for a superior cost and are irrelevant. Nevertheless, this approach leads to the largest number of possibilities and thus tends to the highest optimization time of the three proposed methods.

Mantissa wordlength only optimization This approach aims at optimizing only the mantissa wordlength. The exponent wordlength is set to the value obtained with Equation 2 ($\mathbf{w}_e = \mathbf{w}_e^{\max}$). The variable for the optimization process is a N_v-length vector $\mathbf{w}$ composed only of the mantissa wordlength w_{M_i} for each variable v_i of the set $\mathbb{V}$. This a priori information allows limiting the search space to the set $\mathbb{M}$. Compared to the first approach, the number of variables in the optimization process has been divided by two. Thus, the number of tested configurations during the optimization process and the optimization time are reduced. The number of elements n_p of the search space can be expressed with the following expression:

$$n_p = M^{N_v} \tag{6}$$

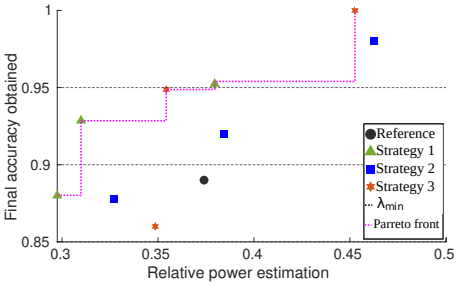
Nevertheless, this approach can lead to a solution less optimized compared to the one obtained with the first approach. This approach does not explore the potential benefit of reducing slightly the exponent wordlentgh in relation to the value obtained with Equation 2.

Exponent and mantissa wordlength sequential optimization This approach is a middle ground between the other two. The optimization problem is divided into two optimization processes carried-out one after the other. First the exponent wordlengths are optimized and then the mantissa. The variable for the first optimization process is a N_v-length vector $\mathbf{w}$ composed of the exponent wordlength w_{E_i} for each variable v_i of the set $\mathbb{V}$. The variable for the second optimization process is a N_v-length vector $\mathbf{w}$ composed of the mantissa wordlength w_{M_i} for each variable v_i of the set $\mathbb{V}$. The optimization is performed on a subset of the search space $\mathbb{E}_r * \mathbb{M}$. The number of elements n_p of the search space can be expressed with the following expression:

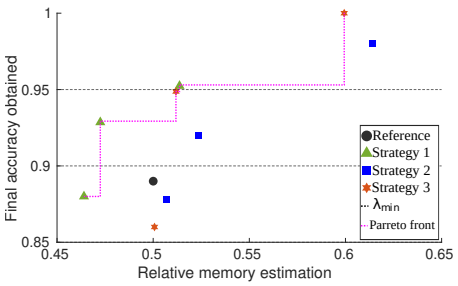
$$n_p = M^{N_v} + \prod_{i=0}^{N_v} w_{E_i}^{\max} \tag{7}$$

Since this approach is a middle ground between the two other methods in terms of exploration space, the number of configurations required tends to be a middle ground. However, optimizing with a sequential optimization algorithm such as *min+1* can lead for high λ_{min} to the exploration of more configurations.

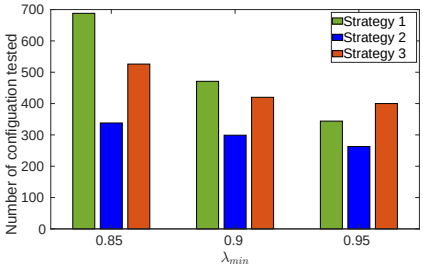
This solution brings a new hyperparameter to the optimization process. The quality degradation due to finite precision must be budgeted between the two independent optimization processes. In this work, the goal $\lambda_{E_{min}}$ for the first optimization process was set to equal the quality of the original system.



(a) Pareto chart of relative energy consumption (processing part) and final accuracy obtained according to the strategy and the minimal quality λ_{min}.



(b) Pareto chart of relative memory space and final accuracy obtained according to the strategy and the minimal quality λ_{min} .



(c) Number of configurations tested for each optimization process according to the strategy and λ_{min}.

Fig. 3: Results for the squeezenet application.

6 Experiments and results

The results are presented for two applications corresponding to a Linear Time-Invariant (LTI) system and a Convolutional Neural Network (CNN). The LTI system is an Infinite Impulse Response filter composed of four cascaded biquad cells. The optimization process is defined with $N_v = 7$. The quality evaluation metric is based on the Signal to Quantization Noise Ratio (SQNR), considering as a reference the single-precision floating-point data types (FP32). The second application is the Squeezenet CNN used for image classification. The network is composed of 14 layers. The optimization process is defined with $N_v = 15$. The CNN structure is complex due to pooling layers that add unsmooth operations. The quality evaluation metric is based on the top 1 accuracy, considering as a reference the FP32 implementation.

Firstly, the efficiency of our approach for finding an efficient solution in terms of tradeoff between cost and quality is evaluated for different quality metric constraints. For the Squeezenet, the tradeoff between the implementation cost and the quality metric is depicted in Figure 3a and Figure 3b respectively for the energy consumption and for the memory size. The x-axis represents the

Table 1: Optimization results of an IIR filter for different strategies and λ_{min}.

	Strategy	Relative power estimation	Relative memory estimation	SQNR obtained (db)	Number of configuration tested
λ_{min}	FP16	0.40	0.50	51.26	.
50	1	0.39	0.32	50.35	233
	2	0.42	0.45	51.18	154
	3	0.39	0.33	50.28	185
70	1	0.51	0.42	70.72	197
	2	0.54	0.55	70.10	125
	3	0.50	0.44	70.09	156
90	1	0.86	0.74	90.14	749
	2	0.87	0.81	90.956	112
	3	0.64	0.56	90.13	161

implementation cost normalized with the cost obtained for the reference solution with FP32. The y-axis represents the quality metric. The results are presented with the three proposed strategies and for three values of λ_{min}. The Pareto front is shown with the dotted line. These solutions can be compared with the one obtained for the IEEE 754 half-precision floating-point data type (FP16). These results show that the solutions located in the Pareto front and obtained with our approach are better than the FP16 solution. For a similar quality metric value, a difference of 7% and 3% for the relative energy consumption are measured. Indeed, FP16 is a generic data type that can not take advantage of the specificities and the limited variable dynamic range of the considered application. Moreover, customizing mantissa and exponent wordlength allows obtaining more flexibility in terms of cost and quality compared to standard data types.

When comparing the three proposed strategies, the second strategy is always less efficient than the two others. This shows that reducing the exponent wordlength compared to $w_{E_i}^{\max}$ provides gain. Decreasing the exponent wordlength of one or two bits can be acceptable in terms of quality degradation. This is illustrated in Figure 2, where the small values located in the left distribution tail will be approximated with unnormalized numbers. Even, a reduction of one bit is not negligible when the exponent wordlength is small.

The results obtained for the IIR filter are summarized in Table 1. The solution with FP16 and our solutions obtained for $\lambda_{min} = 50\ dB$ are close. The dynamic range of the variables inside the filter is very high and the five bits of the exponent are required to represent these values. As for Squeezenet, the second strategy always leads to a more conservative solution. By considering the two applications, both the first and third strategies can provide the best solution.

The number of configurations n_C tested during the optimization process are depicted in Figure 3c for the Squeezenet and summarized in Table 1 for the IIR filter. By optimizing only the mantissa wordlength the second strategy always requires testing fewer configurations compared to the two others. Then, by using a sequential process, the third solution always requires half as much configurations compared to the first solution.

7 Conclusion

This paper proposes a new method for custom floating-point refinement. Three strategies were established to optimize the exponent and mantissa wordlengths according to a user-defined quality constraint. These strategies are based on data dynamic range analysis to adapt the exponent wordlength. The results show an improvement in terms of power and memory consumption compared to a generic solution with FP16 data types. Our approaches allow obtaining different trade-offs between cost and quality.

References

1. Barrois, B., et al.: Customizing fixed-point and floating-point arithmetic — A case study in K-means clustering. In: International Workshop on Signal Processing Systems (SiPS). pp. 1–6 (Oct 2017). https://doi.org/10.1109/SiPS.2017.8109980
2. Caffarena, G.: Wordlength optimization of fixed-point algorithms. In: Approximate Computing Techniques: From Component-to Application-Level, pp. 261–284. Springer (2022)
3. Gaffar, A., et al.: Unifying bit-width optimisation for fixed-point and floating-point designs. In: Symposium on Field-Programmable Custom Computing Machines. pp. 79–88 (Apr 2004). https://doi.org/10.1109/FCCM.2004.59
4. Horowitz, M.: Computing's energy problem (and what we can do about it). In: International Solid-State Circuits Conference (ISSCC) (2014)
5. IEEE Computer Society: IEEE Standard for Floating-Point Arithmetic (Aug 2008)
6. Jouppi, N., et al.: Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product. In: International Symposium on Computer Architecture (ISCA). pp. 1–14. Valencia, Spain (Jun 2021). https://doi.org/10.1109/ISCA52012.2021.00010
7. Kuzmin, A., Van Baalen, M., Ren, Y., Nagel, M., Peters, J., Blankevoort, T.: Fp8 quantization: The power of the exponent. Advances in Neural Information Processing Systems **35**, 14651–14662 (2022)
8. Kwak, J., et al.: Quantization Aware Training with Order Strategy for CNN. In: International Conference on Consumer Electronics-Asia. pp. 1–3 (Oct 2022). https://doi.org/10.1109/ICCE-Asia57006.2022.9954693
9. Lee, D., et al.: MiniBit: bit-width optimization via affine arithmetic. In: Conference on Design automation (DAC). p. 837 (2005). https://doi.org/10.1145/1065579.1065799
10. Liu, F., et al.: Improving Neural Network Efficiency via Post-training Quantization with Adaptive Floating-Point. In: International Conference on Computer Vision (ICCV). pp. 5261–5270. Montreal, Canada (Oct 2021). https://doi.org/10.1109/ICCV48922.2021.00523
11. Micikevicius, P., et al.: FP8 Formats for Deep Learning (Sep 2022)
12. Osborne, W., et al.: Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems. In: International Conference on Field Programmable Logic and Applications. pp. 617–620 (Aug 2007). https://doi.org/10.1109/FPL.2007.4380730
13. Shah, H., et al.: KD-Lib: A PyTorch library for Knowledge Distillation, Pruning and Quantization. ArXiv (Nov 2020). https://doi.org/10.48550/arXiv.2011.14691
14. SIEMENS EDA: Algorithmic C Datatypes reference manual (Aug 2022)
15. WedoLow: Software engineering, wedolow.com