# Design Space Exploration for Memory-Oriented Approximate Computing Techniques

Hugo Miomandre, Jean Francois Nezan, Daniel Ménard

# Design Space Exploration for Memory-Oriented Approximate Computing Techniques

Hugo Miomandre
*INSA Rennes*
*IETR, UMR CNRS 6164*
Rennes, France
Hugo.Miomandre@insa-rennes.fr

Jean-François Nezan
*INSA Rennes*
*IETR, UMR CNRS 6164*
Rennes, France
Jean-Francois.Nezan@insa-rennes.fr

Daniel Ménard
*INSA Rennes*
*IETR, UMR CNRS 6164*
Rennes, France
Daniel.Menard@insa-rennes.fr

***Abstract*—Modern digital systems are processing more and more data. This increase in memory requirements must match the processing capabilities and interconnections to avoid the memory wall. Approximate computing techniques exist to alleviate these requirements but usually require a thorough and tedious analysis of the processing pipeline. This paper presents an application-agnostic Design Space Exploration (DSE) of the buffer-sizing process to reduce the memory footprint of applications while guaranteeing an output quality above a defined threshold. The proposed DSE selects the appropriate bit-width and storage type for buffers to satisfy the constraint. We show in this paper that the proposed DSE reduces the memory footprint of the SqueezeNet CNN by 58.6% with identical Top-1 prediction accuracy, and the full SKA SDP pipeline by 39.7% without degradation, while only testing for a subset of the design space. The proposed DSE is fast enough to be integrated into the design stream of applications.**

***Index Terms*—Approximate Computing, Design Space Exploration, Signal Processing, Deep Neural Network**

## I. Introduction and Context

Digital signal processing and multimedia applications have to compute an ever increasing amount of data. For instance, the SKA [3], [14] is the world's largest radio telescope project in construction, planned to generate 16 Tb/s data streams.

In 2021, the digital world accounted for 4% of greenhouse gases and 10% of the electricity consumption.Memories are responsible for the major part of the power consumption of both Edge Computing [7], [15] and HPC systems [16], as the area occupied can be as large as 80% of a chip and external memories are often required to extend these internal memories.

Approximate Computing (AxC) enables improvements from circuit to system levels to reduce the energy consumption, computation time, or implementation cost as presented in [5]. In this context, three main approaches have been proposed in AxC: *Hardware-Level*, *Computation-Level* and *Data-Level* AxC techniques. The contributions of this paper belong to the latter category, which impact the processed data. Existing precision optimization techniques [10] aim at reducing the implementation cost of the processing part without taking into account the memory part.

Memory reduction AxC techniques are typically done at design time and used on top of memory handling [8] method, and may be application-specific [11]. These techniques require an in-depth knowledge of the computation and handling performed on the data, which tends to be time-prohibitive. Hence the need to develop fast and generic Design Space Exploration (DSE) algorithms to optimize the precision of data buffers.

In this paper, we propose DSE method to ease the implementation of a precision optimization technique aiming to reduce the memory footprint of an application while respecting a user-defined quality constraint. Our method, relying on [12], is fast enough to be integrated into the design stream of an application.

Our contribution is a data-driven AxC DSE method. The goal is to find the binary representation of manipulated data with the smallest number of bits, while preserving the accuracy of the final result.

The Approximate Buffer (AxB) technique presented in [12] consists in concatenating data into buffers to reduce the memory footprint. Data are represented using the Fixed-Point (FxP) format with an arbitrary bit-width, with every data from an AxB using the same format. It enables memory footprint reductions between 27% and 68% on the application examples without significant degradation of the output quality, at the cost of a manual and time-consuming DSE.

While current DSE methods enable the implementation of AxC-based techniques on applications, these methods are either architecture-specific [6] or application-specific [13], but also does not take into account the memory footprint improvement achievable with AxC [4]. To the best of our knowledge, there is no automatic and generic DSE algorithm for memory footprint reduction method based on data precision optimization AxC techniques like AxBs.

## II. Contributions

The DSE method presented in this paper automates and extends the AxB technique presented in [12], adding the support of representations in addition to the FxP format.

### A. Alternative Data Representation

By default, AxBs use the FxP format to store data. However, as the whole AxB uses the same format, FxP is poorly suited to represent values across a wide range. Hence, we added support for two additional data representations.

The custom Floating-Point (cFP) format extends the rule-set of regular FP arithmetics [1].The cFP representation uses the same four parameters to represent the sign-bit $s$, the size

of the exponent field $e$ and the mantissa $m$, and the value of the exponent bias $b$. These parameters are adjusted to reduce the number of bits required to represent a value while still maintaining a sufficient level of accuracy.

The concept of Uniform Quantazition (UQ) representation consists in re-scaling the data to store to fit a new representation space. This representation space is sub-divided in $2^n$ steps, with $n$ denoting the number of bits to represent the data. The conversion slightly differs from the conventional uniform quantization to enforce the proper storage of 0. The simplicity of the UQ format goes well with the possibility to handle data with an arbitrary bit-width.

### B. Memory Footprint Minimization Algorithm

Our generic DSE algorithm is designed to minimize the memory footprint of an application, while maintaining the result above a specified threshold of a desired quality metric. Our DSE relies on the AxB concept detailed in [12], allowing data to be represented and stored in memory on an arbitrary number of bits, without the usual $2^n$ constraints. Alternative data representations from Section II-A are added within AxBs, requiring no additional modification of the application original source code, aside from the initial modification needed for AxBs usage.

One of the main feature of our DSE is the distinction between **global buffers** and **local buffers**. Global buffers are are allocated for the whole lifespan of the application, such as the result vector of an iterative process or a reference set of value that needs to be compared against. Local buffers are not required for the whole duration of the application and whose memory space can be reused [8].

This clear distinction between **global buffers** and **local buffers** allows to opportunistically increase the bit-width of local buffers when possible, reducing the number of configuration to explore. This operation is later on referred to as buffer inflation.

A first run of the application is performed to monitor the variation of values to store within AxBs. This enables the determination of the base storage format to be used for each representation.

Algo. 1 shows the global view of our method and its separation into three subsections detailed in Algo. 2, Algo. 3 and Algo. 4.

**Algorithm 1:** Global view of the DSE algorithm.

```
1 w^min ← minValueDetermination // Algorithm 2
2 q^test ← runTest(w^min)
3 if q^test < q^constraint then
4     w^min ← IterativeProcess(w^min) // Algorithm 3
5 return bitScraping(w^min) // Algorithm 4
```

*1) Min Value Determination:* The first part (Algo. 1:1), detailed in Algo. 2, consists in finding, for each individual buffer, the minimal number of bits $b^{best}$ and the associated data-type which satisfy the quality constraint $q^{constraint}$, while every other buffer are kept at their original data-type. The goal of this step is to find an appropriate starting vector $\boldsymbol{w}^{min}$. Each bit-width $n^{bit}$ is tested with each candidate data-type. Intermediary results are kept aside for latter parts of the DSE, as well as potential future explorations with different quality constraints. This vector of individual minimums $\boldsymbol{w}^{min}$ is then tested to check if it satisfies the quality constraint (Algo. 1:2-4). If not, the algorithm continues with the second part (Algo. 1:4) detailed in Algo. 3.

**Algorithm 2:** Min Value Determination.

```
   Input: The list of buffer to process
   Output: The list of minimal sizes
1  foreach j ∈ bufGlob ∪ bufLoc do
2      q^test ← ∞
3      n^bit ← MAX_NB_BIT
4      b^best ← MAX_NB_BIT
5      for i ← log2(MAX_NB_BIT) - 1 downto 0 do
6          if q^test ≥ q^constraint then
7              n^bit ← n^bit - 2^i
8          else
9              n^bit ← n^bit + 2^i
10         q^test ← findBestType(j, n^bit)
11         if (q^test ≥ q^constraint) AND (n^bit < b^best) then
12             b^best ← n^bit
13     w_j^min ← b^best
14 return inflateBuffer(w^min)
```

*2) Iterative Process:* The main part of Algo. 3 is an iterative convergent process to find a buffer configuration $\boldsymbol{w}^{test}$ with a better *qMem* ratio ($\frac{\Delta \text{quality}}{\Delta \text{memory}}$).

The goal is to slowly increase the output quality of the application by selectively adding bits to data buffers, starting from $\boldsymbol{w}^{min}$. On every step, a bit-budget $w^{budget}$ and a $w^{offset}$ are used to determine the number of bits to add. Global buffers are handled as independent buffer-entities, while local buffers are all considered as a single unique buffer-entity. With $n^{global}$ and $n^{local}$ the number of global buffers and local buffers, the number of buffer-entity to consider for the definition of $w^{budget}$ and $w^{offset}$ is $n^{entity} = n^{global} + \min(n^{local}, 1)$. $w^{budget}$ corresponds to the number of buffer-entity to receive a single additional bit, and $w^{offset}$ to the number of bits to add to each buffer-entity.

During a step, the algorithm sets up a list $\boldsymbol{W}^{test}$ of all the configurations possible with $w^{budget}$ and $w^{offset}$ from $\boldsymbol{w}^{min}$. $\boldsymbol{W}^{test}$ inflated, creating potential duplicates to remove, and ordered by increasing memory footprint. Finally, the application is executed with every entry of $\boldsymbol{W}^{test}$.

If no satisfying *qMem* ratio can be found, the $w^{budget}$ is increased, until it reaches $n^{global} + \min(n^{local}, 1)$. $w^{budget}$ then goes back to 1, and $\boldsymbol{w}^{offset}$ is incremented and applied on every buffer (Algo. 3:9). Local buffers with non-overlapping lifespans are completely independent from one another. Consequently, as the maximal memory footprint is achieved in a

specific section of the application, local buffers that are not impacting this maximum can have their bit-width increased to the point it either reaches the memory ceiling, or reaches the number of bits used for the original data.

**Algorithm 3:** Iterative Process.

**Input:** The list of buffer to process
**Output:** A set of parameter that satisfies $q^{constraint}$

$q^{iter} \leftarrow q^{test}$, bestQMem $\leftarrow 0$
$w^{budget} \leftarrow 0$, $w^{offset} \leftarrow 0$, exit $\leftarrow$ False
**while** *exit = False* **do**
  // If no satisfactory step has been found, add an additional bit to the pool
  **if** *bestQMem* $\leq 0$ **then**
    $w^{budget}$++
  **else**
    $w^{budget} \leftarrow 1$, $w^{offset} \leftarrow 0$
  // If an additional bit has been added to every buffer, increase the pool offset
  **if** $w^{budget} > n^{entity}$ **then**
    $w^{budget} \leftarrow 1$, $w^{offset}$++
  // Provides a list of test vectors ordered by increasing footprint
  $\mathbf{W^{test}} \leftarrow$ **generateW$^{test}$**($\mathbf{w}^{min}$, $n^{global}$, $n^{local}$, $w^{budget}$, $w^{offset}$)
  **foreach** $\boldsymbol{w}^{test}$ *in* $\boldsymbol{W}^{test}$ **do**
    $q^{test} \leftarrow$ **runTest**($\mathbf{w}^{test}$)
    qMem $\leftarrow \frac{\Delta quality}{\Delta memory}$
    // Check if satisfactory solution
    **if** $q^{test} \geq q^{constraint}$ **then**
      exit $\leftarrow$ True, Break
    // Check if satisfactory next step
    **if** *qMem* $>$ *bestQMem* **then**
      bestQMem $\leftarrow$ qMem
      $q^{iter} \leftarrow q^{test}$
      $\mathbf{w}^{min} \leftarrow \mathbf{w}^{test}$
    // If buffers are already at max size, then exit without solution
    **if** $\sum \boldsymbol{w}^{test} = n^{entity} \times MAX\_NB\_BIT$ **then**
      exit $\leftarrow$ True, Break
**return** $\mathbf{w}^{min}$

To simplify the representation of the method, some of its parts have been offloaded in the form of functions, detailed hereafter, the first two being user-provided:

- **qualityEval**(): Evaluate the quality from a reference.
- **appFootprint**($\boldsymbol{w}^{test}$): Computes the memory footprint. Can be user-provided or from a development framework.
- **findBestType**($\boldsymbol{w}^{test}{}_j$, $n^{bit}$): If needed, runs the application with a single AxB for buffer $\boldsymbol{w}^{test}{}_j$ on $n^{bit}$. Returns the best type and save the result for future uses.
- **inflateBuffer**($\boldsymbol{w}^{test}$): Gradually increases the size of local buffers until it reaches the memory ceiling.
- **runTest**($\boldsymbol{w}^{test}$): Runs the application with the $\boldsymbol{w}^{test}$ parameter set. Returns the output quality value.
- **generateW$^{test}$**($\boldsymbol{w}^{min}$, $n^{global}$, $n^{local}$, $w^{budget}$, $w^{offset}$): Generates a list $\boldsymbol{W}^{test}$ with $w^{budget}$ and $w^{offset}$, inflates, prunes, and orders the list by memory footprint. Returns $\boldsymbol{W}^{test}$.

*3) Bit Scraping:* After the main part is completed (Algo. 3), a pass of bit-scraping (Algo. 4) is performed.

**Algorithm 4:** Bit Scraping.

**Input:** Vector $\mathbf{w}^{min}$ respecting constraint $q^{constraint}$
**Output:** Optimised buffer, New quality value

exit $\leftarrow$ False
**while** *exit = False* **do**
  **for** *i = 0 to* $n^{buffer}$ **do**
    $\mathbf{w}^{test} \leftarrow \mathbf{w}^{min}$
    $\mathbf{w}_i{}^{test} \leftarrow \mathbf{w}_i{}^{test}$ - 1
    **if** ***appFootprint***($\boldsymbol{w}^{test}$) $<$ ***appFootprint***($\boldsymbol{w}^{min}$) **then**
      $q^{test} \leftarrow$ **runWithTestVector**($\mathbf{w}^{test}$)
      **if** $q^{test} \geq q^{constraint}$ **then**
        $\mathbf{w}^{min} \leftarrow \mathbf{w}^{test}$
        $q^{last} \leftarrow q^{test}$
        exit $\leftarrow$ True
**return** $\mathbf{w}^{min}$, $q^{last}$

The result vector $\boldsymbol{w}^{min}$ obtained from Algo. 3 is able to satisfy the quality constraint, but may overshoot it.

Bit-scraping consists in iterating on every buffer of $\boldsymbol{w}^{min}$ to remove unnecessary bits. If the removal allows a reduction of the memory footprint and maintaining of the constraint, the change is committed, otherwise discarded. If no bits can be removed from buffers, the bit-scraping process ends.

## III. Experimental Results

The proposed DSE was tested on 2 C-based applications modified to use AxBs. Tests were performed on a x86 Intel Xeon E5-1620 CPU.

### A. SqueezeNet CNN

SqueezeNet [9] is a CNN for computer vision. Specific methods based on quantization and pruning enable a reduction of the model size, but the reference 4.7MB version is used. The model is pre-trained and does not undergo any re-training.

The proposed DSE is used to minimize the memory footprint of SqueezeNet. The constraint is identical TOP-1 classification for a dataset of 300 images. The memory footprint of the reference implementation is composed of around 4.7MB of model parameters, around 3.8MB of buffers to store data between layers and around 0.7MB of other memory allocations, for a total of 9.2MB.

Fig. 1 shows the memory allocation during the execution of each layer (left columns).

The buffers used to test our DSE method are the 10 global buffers holding the model of the CNN, and 7 of the local

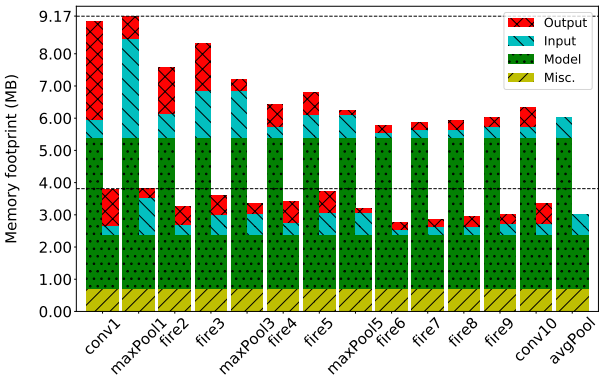


Fig. 1. Memory allocation at different stage of the reference SqueezeNet CNN (left) and with our method with a **100%** accuracy (right).

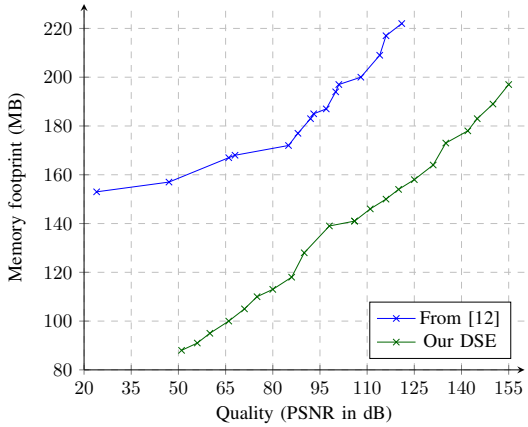


Fig. 2. Graph of the memory footprint dependant of the output quality.

buffers used between the first layers, for a total of 17 buffers. Using our generic DSE algorithm, the memory footprint of the application can be reduced to 3.81MB, a 58.6% reduction, with 100% TOP-1 accuracy. This result is obtained by exploring only 300 of the theoretical maximum $32^{17} \approx 3.87 \times 10^{25}$ possible configurations. The memory footprint obtained with our DSE are displayed on Fig. 1 (right columns).

Inference time when using AxBs is around 7% slower. Considering that a single execution of the application takes around a minute on **CPU**, our DSE method is able to provide a satisfying AxB configuration in approximately 5 hours.

### *B. SDP Imaging Pipeline*

For the second use-case, the SDP Evolutionary Pipeline (SEP) [2] application is considered. The SEP is an implementation of the Square Kilometre Array (SKA) Science Data Processor (SDP) for its most compute-intensive task. The dataset corresponds to 15 minutes of data from 512 antennae. The full SKA is expected to have hundreds of thousands of receivers producing billions of samples per second.

The reference memory footprint of the SEP with this dataset is 234.9MB. The acceptable noise level of this application is evaluated with a PSNR value at around 100dB.

Fig. 2 shows the memory footprint of the application dependant of the output quality, tested with quality constraints from 50dB to 155dB by 5dB increments. The number of configurations tested to reach a result ranges from 7 to 50.

Using our DSE with a 105dB constraint yields an output quality of 106dB with a memory footprint of 141MB, a 39.7% reduction. This result is obtained by exploring only 7 of the maximum $32^9 \approx 3.52 \times 10^{13}$ possible configurations, with a runtime of around 170 seconds per configuration. The use of AxBs on this application leads less than 1% performance penalty.

The proposed DSE is able to provide the AxB configuration for a given quality constraint in a time ranging from minutes to hours.

## IV. Conclusion

This paper presents an automatic and generic DSE method to optimize the setup of the AxB AxC technique.

We have shown that our DSE method is capable of finding the adequate AxB configuration to reduce the memory footprint of data processing application. Our DSE method has been tested on SqueezeNet CNN with a pre-trained model and on an implementation of the SKA SDP pipeline.

Precision optimization of applications' computations is not in the scope of the proposed method but are complementary and can be combined with our approach.

Future works will consist in merging this method into a code generation framework and studying the impact of its result to target other kind of hardware architecture such as FPGAs.

## References


[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2019*.

[2] SEP Pipeline Imaging - GitLab. https://gitlab.com/ska-telescope/sdp.

[3] SKA Observatory Public Website. https://www.skatelescope.org/.

[4] S. Barone, M. Traiola, M. Barbareschi, and A. Bosio. Multi-objective application-driven approximate design method. *IEEE Access*, 2021.

[5] A. Bosio, D. Menard, and O. Sentieys. A Comprehensive Analysis of Approximate Computing Techniques: From Component- to Application-Level. ESWEEK 2018, 2018.

[6] J. Castro-Godínez, J. Mateus-Vargas, M. Shafique, and J. Henkel. Axhls: Design space exploration and high-level synthesis of approximate accelerators using approximate functional units and analytical models. In *ICCAD*, 2020.

[7] E. de Greef, F. Catthoor, and H. de Man. Array placement for storage size reduction in embedded multimedia systems. In *ASAP*, USA, 1997.

[8] K. Desnos, M. Pelcat, J.-F. Nezan, and S. Aridhi. On memory reuse between inputs and outputs of dataflow actors. *ACM TECS*, 2016.

[9] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡0.5mb model size, 2016.

[10] D. Menard, D. Chillet, and O. Sentieys. Floating-to-fixed-point Conversion for Digital Signal Processors. *EURASIP*, 2006.

[11] S. Minakova and T. Stefanov. Buffer sizes reduction for memory-efficient cnn inference on mobile and embedded devices. In *DSD*, 2020.

[12] H. Miomandre, J.-F. Nezan, D. Menard, A. Campbell, A. Griffin, S. Hall, and A. Ensor. Approximate buffers for reducing memory requirements: Case study on SKA. In *SiPS*, 2020.

[13] G. Paim, L. M. G. Rocha, H. Amrouch, E. A. C. da Costa, S. Bampi, and J. Henkel. A cross-layer gate-level-to-application co-simulation for design space exploration of approximate circuits in hevc video encoders. *IEEE Transactions on Circuits and Systems for Video Technology*, 2020.

[14] J. Santander-Vela, L. Pivetta, and N.P. Rees. Status of the Square Kilometre Array. In *Proc. of International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS'17)*, ICALEPCS.

[15] Wm. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, March 1995.

[16] D. Zivanovic, M. Pavlovic, M. Radulovic, H. Shin, J. Son, S. A. Mckee, P. M. Carpenter, P. Radojković, and E. Ayguadé. Main memory in hpc: Do we need more or could we live with less? *ACM TACO*, 2017.