Approximate Accelerators: A Case Study using Runtime Reconfigurable Processors

Fabian Lesniak, Tanja Harbaum, and Jürgen Becker
Karlsruhe Institute of Technology
Karlsruhe, Germany
{lesniak, harbaum, becker}@kit.edu

***Abstract*—Dennard Scaling and Amdahl's Law have ensured the continuous improvement of computing architectures for years, but even after they fade, the demand for ever-increasing performance continues to rise. Highly specialized architectures have further increased processing power for specific use cases, but versatility is then severely limited. Reconfigurable processors leverage FPGA technology to close the gap between general purpose processors and highly specialized architectures. Loading application-specific accelerators accelerates the current application, but limited available resources and configuration time must be taken into account. In this paper, we demonstrate the use of approximate computing techniques within runtime reconfigurable processors. Many algorithms can tolerate a certain degree of inaccurate calculations, which can improve the required area, resources, energy and processing time of a hardware accelerator. Showcasing an edge-detection algorithm, we can show a runtime improvement of $67\,\%$ by using an approximate implementation, while reducing the required accelerator resources by $75\,\%$. Our evaluation of runtime reconfiguration shows a processing time improvement of $14.1\,\mathrm{ms}$ per iteration, while accounting for a one-time reconfiguration period of $5.61\,\mathrm{ms}$. This delay represents less than $6.87\,\%$ of the processing time of our test image, yet it allows for the concurrent acceleration of other algorithms on the same framework, which we demonstrated by means of a use case running AES-256 encryption in parallel with edge detection.**

***Index Terms*—run-time reconfiguration, reconfigurable processors, approximate computing**

I. Introduction

One trend has been present for many years already and will certainly persist in the future: the demands on processing power of computing systems are steadily increasing. However, the approaches of both science and industry to meet these increasing requirements varied over time. Dennard Scaling and Amdahl's Law have been the leading force in semiconductor engineering for many years, but both are not applicable no more. Micro-architectureal optimizations allow for additional performance gains, whereas the integration of several accelerators for different types of applications improve the processing power in corner cases at the expense of greatly increased chip area. An example of such accelerators in processors is the implementation of SIMD instructions into the instruction set, like the SSE and AVX extensions commonly seen in x86 processors. However, the increased chip area for these special accelerators is often unused, as long as no corresponding application is being executed, and only algorithms which are known at design time can be accelerated by such extensions. An approach to overcome these limitations is the use of reconfigurable hardware to create processor architectures which can adapt to a given workload at design time.

Choosing the hardware platform for a specific computing problem comes down to a trade-off between flexibility, performance and resources. While using a processor brings high flexibility, its performance is considerably lower than a hardware implementation. Alternatively, a specialized ASIC implementation is inherently immutable but provides highest performance. Reconfigurable hardware aims to combine the best of both worlds, typically reaching higher performance than CPU-based solutions and allowing reconfiguration. However, the performance of an ASIC generally can't be matched and FPGA designs are generally more difficult to build than a software solution. This paper showcases a runtime reconfigurable processor, which comprises a standard GPP core and an adaptive accelerator framework, and presents multiple approaches to improve the performance and resource usage of algorithms on this processor by utilizing approximate computing.

Approximate computing is a paradigm based on the property of many algorithms to tolerate partially incorrectly evaluated equations. In particular, scenarios that rely on image data, such as classical audio and image processing or AI-based methods, can profit from approximations. By accepting certain errors in the evaluation, arithmetic operations can be performed more efficiently, reducing processing time, energy consumption, and resource requirements. The main contribution of this paper is to evaluate the impact of approximate computation methods on reconfigurable processors, showing in particular that approximation allows to accelerate multiple algorithms in parallel by overcoming resource constraints and I/O bottlenecks.

II. Related work

A. Reconfigurable Architectures

Reconfigurable processors try to bridge the gap between the flexibility of von-Neumann processors and the computing power of specialized hardware [1]. Coarse-grained reconfigurable architectures (CGRAs) differ from classical, control-flow oriented processors in that the programming is primarily data-flow driven [2]. They are characterized by the presence of the computational elements as fixed logic, while allowing to adapt the data flow between the elements. CGRAs feature a very low reconfiguration time, however their flexibility is

limited to a specific domain based on the available processing cores. CGRA-ME is a framework to model and explore specific CGRA designs by specifying the architecture on a higher level [3]. The resulting hybrid platform commonly incorporates a RISC-V core to improve the versatility and allows the execution of standard software.

In contrast to CGRAs, fine-grain reconfigurable architectures can map any hardware function, i.e. both data path and logic can be adapted. This makes them particularly flexible and applicable in a wide variety of domains. One example for such architectures are extensible processors, which are based on a general-purpose processor (GPP) and extend the instruction set with custom instructions. Standard software can be executed, but computationally intensive operations can be identified, for which algorithms are being actively researched, and mapped to an FPGA-based logic block for faster execution [4]. Ali et. al demonstrate a RISC-V processor with specific instruction set extensions for machine learning applications [5]. This implementation is more efficient compared to software processing and shows an advantage of 1.2x over a GPU implementation. However, the performance gain is limited, as the design uses a vector processing unit instead of a more specific accelerator. Reconfigurable processors with accelerator fabric consisting of field-programmable gate arrays (FPGAs) allow for highly specialized accelerators while maintaining the programmability of a GPP [6].

B. Approximate Computing

Traditionally, computing architectures are built to produce results as accurately as possible. However, several applications can tolerate a certain amount of errors and still behave correctly from the user's point of view. Li et. al. provide an analysis for the required application-level correctness of a program, where the necessary level of quality depends on the application [7]. Allowing approximate results can result in faster processing and lower demands on logic cells, thereby it can also increase the energy efficiency. Han et. al. propose approximate computing as a paradigm for energy efficient design, including the design of approximate arithmetic blocks to allow for improved efficiency through methods like voltage overscaling [8]. An apparent approach lends itself to iterative procedures, which can be terminated prematurely at the expense of accuracy. Systematic approaches can be used to generate approximated adders as Shafique et. al. propose, as breaking up the carry chain significantly improves timing. In addition, the type of target platform must also be considered [9]. Ahmad et. al. present an efficient implementation of approximate adders on FPGAs that is particularly focused on the architecture of FPGAs in the form of lookup tables (LUTs), which makes it even more efficient [10].

III. Approximate Accelerators for Reconfigurable Processors

Reconfigurable processors leverage FPGA-based hardware blocks to adapt to different workloads during runtime. In this way the advantages of processors and hardware accelerators

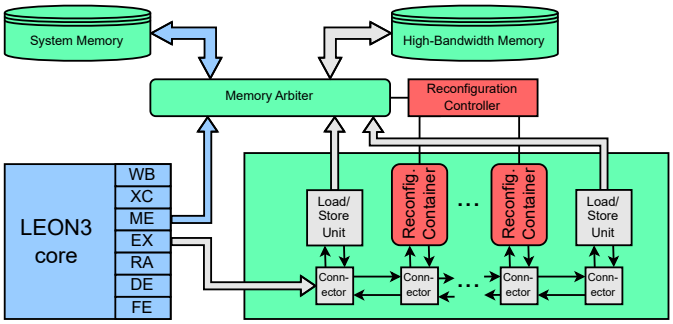


Fig. 1. Reconfigurable processor architecture with RCs and High-Bandwidth Memory, based on a LEON3 core

are combined: while the GPP allow for easy implementation and re-programming of software with a focus on control flow, a hardware accelerator excels at processing of computational kernels. Our reconfigurable processor prototype depicted in Fig. 1 features a LEON3 processor core implementing the SPARC v8 instruction set. It can decode additional instructions, so-called special instructions (SIs), to access the attached accelerator fabric. The accelerator fabric consists of multiple reconfigurable containers (RCs) which are connected through a linear data bus with a width of four 32-bit words. In addition to the RCs, two load/store units are connected at each end of the bus. They include an address generator to generate different address patterns and allow high-bandwidth access to memory with a 128-bit interface each. The RCs can be configured at runtime with application-specific accelerators and commonly contains combinatory logic only. As one RC is limited in resources like LUTs, registers and memory blocks, it is necessary to split the accelerators of complex SIs into multiple RCs and to move data between them during execution. The interconnect between RCs is configured using a micro-program which we call very large configuration word (VLCW), as it controls the data flow of each junction in the data bus. When a SI is encountered, the pipeline stalls in the execute stage and starts processing the VLCW step-by-step. These steps, which we call phases of the SI execution, usually consist of initial data loading, routing data to the respective RCs, moving data between related RCs and writing result data back to memory using the load/store units. An advantage of the modular RCs is the possibility to load the accelerators for multiple SIs at once, if the number of RCs is sufficient. Depending on the SI, different VLCWs are selected which use the required accelerators only. If the number of RCs is not sufficient to load the accelerators for all required SIs, the application can decide at runtime if reconfiguration is feasible or a software implementation of the algorithm has to be used [6]. This creates a trade-off between reconfiguration time, number of required accelerators and slow-down of a pure software implementation.

By applying approximate computing techniques, the performance of a hardware accelerator can be improved and the resource requirements are reduced. This is based on the

property of many algorithms to provide acceptable results even if they are not computed exactly. When looking at hardware accelerators, different approximate computing techniques can be applied [11]:

- approximation through simplification on the algorithmic level, like early termination of an iterative algorithm or omitting
- using approximate circuits like inexact adders and multipliers
- quantization or other accuracy reduction of data types, e.g. reducing bit width of links or LUT contents

It depends strongly on the algorithm which technique can be applied successfully and what impact they have. The applicability of different approximation techniques is also very dependant on the hardware architecture. When looking at our reconfigurable processor design, some aspects are particularly important:

- reducing required accuracy of data being exchanged through the interconnect can reduce the number of phases required for execution of a SI
- memory traffic can be reduced if less input data is required or less output data needs to be written back
- reduced resource utilization resulting in a smaller bitstream allows for faster reconfiguration of a RC during runtime
- saving enough resources to combine two related accelerators into a single RC significantly eases data transfer between these accelerators and allows other traffic on the interconnect during execution

It is especially useful to merge related RCs, as other accelerators can be loaded into a vacant RC to allow execution of another SI in parallel. As image processing algorithms can often benefit from approximation, a case study using the SUSAN edge detection algorithm is shown in Section III-A. A closer look at the arithmetic of this algorithm is taken and ways to apply different approximation techniques to the exact hardware implementation are highlighted. Section IV discusses the results and benefits from applying approximate computing to a reconfigurable processor architecture.

A. Case Study: SUSAN Edge Detection

In this section, we demonstrate the impact of approximation on an implementation of the SUSAN edge detection algorithm running on our reconfigurable processor design. The SUSAN algorithm is suitable to detect edges and corners in a greyscale image [12]. A common approach to edge detection is to calculate the brightness gradient through derivation, as it is used in the Canny edge algorithm [13] along with other factors. SUSAN uses a different approach, where the similarity of each pixel to its individual neighbours is computed. The similarity factor $c(\vec{r}, \vec{r_0})$ is calculated according to Eq. (1) using a threshold t.

$$c(\vec{r}, \vec{r_0}) = e^{-(\frac{I(\vec{r}) - I(\vec{r_0})}{t})^6} \quad (1)$$

$$n(\vec{r_0}) = \sum_{\vec{r}} c(\vec{r}, \vec{r_0}) \quad (2)$$

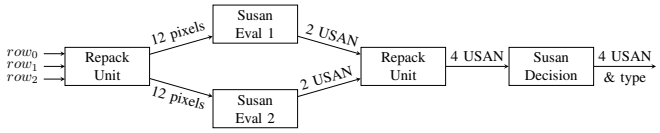


Fig. 2. Data flow graph of the exact SUSAN SI implementation with three different RC types

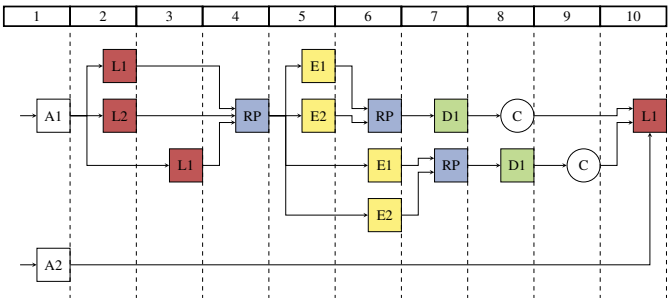


Fig. 3. VLCW execution schedule of the exact SUSAN SI implementation (D/E=SUSAN decision/evaluation, RP=data repacking)

The homogeneity of the area surrounding each pixel, named univalue segmented area nucleus (USAN), is computed to denote the similarity of the pixels within this area (Eq. (2)). An edge is found based on the difference between two neighboring USANs and their relative location, if the USAN value $n(\vec{r_0})$ is sufficiently small. To determine the direction of an edge, a weighted average of neighboring USANs is used:

$$\bar{\vec{r}}(\vec{r_0}) = \frac{\sum_{\vec{r}} \vec{r} \cdot c(\vec{r}, \vec{r_0})}{\sum_{\vec{r}} c(\vec{r}, \vec{r_0})} \quad (3)$$

$\vec{r}$ can be expressed as $\sqrt{x^2 + y^2}$ using the Euclidean norm.

To implement this algorithm within the reconfigurable processor, it can be decided which parts of the algorithm are most suitable for hardware acceleration. The USAN and the edge direction (Eq. (3), in most cases) needs to be independently calculated for all pixels and can be processed in parallel. Fig. 2 illustrates the data flow of the hardware implementation and shows the evaluation, decision and data repacking modules involved. The interconnect bandwidth of 128 bits limits image data transfer to two adjacent kernels per phase, as a kernel consisting of nine 8-bit pixels is required for each USAN. In total, four kernels are evaluated per SI execution to maximize usage of the 128 bit image rows fetched by the load/store units. In addition, the RC interface of 64 bits is not sufficient to transfer a full kernel at once, requiring two phases for the initial data transfer as shown in Fig. 3.

Three computationally intense operations are necessary to evaluate each USAN: a division operation and the natural exponential function in Eq. (1), and calculation of the square root in Eq. (3). A straightforward HLS implementation of the algorithm includes a radix-2 divider and a pipelined 6-stage CORDIC core for square root operations, as well as a LUT with fixed-point values of the natural exponential function. However, by analysing the algorithm in detail and allowing

slightly higher errors, all CORDIC cores can be avoided and processing time is greatly reduced.

The threshold t in Eq. (1) is constant and can be included in the LUT for e^x, saving one division operation while maintaining accuracy. The centre of gravity (Eq. (3)) is used if the edge is on the border between two USANs. It requires the calculation of a square root for the Euclidean norm of $\vec{r}$, but since the result is only used in comparison to the *geometric threshold* g, it can be simplified as follows:

$$\sqrt{x^2 + y^2} > g = 0.4 \cdot n \tag{4a}$$

$$x^2 + y^2 > 0.16n^2 \tag{4b}$$

$$25 \cdot (x^2 + y^2) > 4 \cdot n^2 \tag{4c}$$

While there is no need for a square root operation, Eq. (4c) can even be evaluated using integer operations only.

Finally, the ratio of $\overline{(y - y_0)^2}$ to $\overline{(x - x_0)^2}$ is used to determine the orientation of the edge by comparing it to 0.5 and 2.0 for the vertical and horizontal edge case, respectively. This can be rearranged to $2 \cdot |y| < |x|$ and $|y| > 2 \cdot |x|$, trading a division for two base-2 multiplications. These optimizations allow to reduce the hardware utilization and the number of cycles without sacrificing accuracy.

B. Appromation of the SUSAN algorithm

Our optimized SUSAN hardware implementation requires the use of four RCs due to resource constraints. Approximate computing techniques can be applied to improve performance, further reduce resource usage and potentially merge related RCs. Fig. 4 shows the execution schedule with approximation for the exponential function LUT and the quadratic multipliers applied, which allows merging the evaluation and decision RCs. Calculating the euclidean norm of $\vec{r}$ involves two quadratic multiplications. Using the technique proposed by Yao et. al. [14], they are replaced with approximate multipliers optimized for FPGAs. Similarly, adders are replaced by the approximate *GeAr* adder as proposed by Shafique et. al. [9]. Both the adders and the multipliers feature a configurable level of approximation, which allow to generate different variants of the hardware accelerated implementation.

In addition to the optimization of computational operations, advantages are also gained from the approximation of data types and their transfer between RCs. The exact implementation works on 8bpp image data, which is also the source format in memory. An RC has two 32-bit inputs, thus two phases are required to load a 3x3 kernel into the evaluation RCs (phases 5+6 in Fig. 3). By reducing the bit width to 7bpp using the repack unit, data can be loaded in a single phase and the subsequent repack step is no longer required. Nevertheless, the initial loading of data still takes up a large portion of the runtime. Processing a 3x3 kernel requires three rows of image data to be loaded, taking two phases (phases 2+3 in Fig. 4) as only two load/store units are present. An *interleaved processing mode* is used to compute the first kernel from only two rows of image data by using the data from the second row for the third as well. The second kernel is then calculated with higher accuracy in the next phase, after the third row has also been loaded. The resulting schedule is shown in Fig. 5.

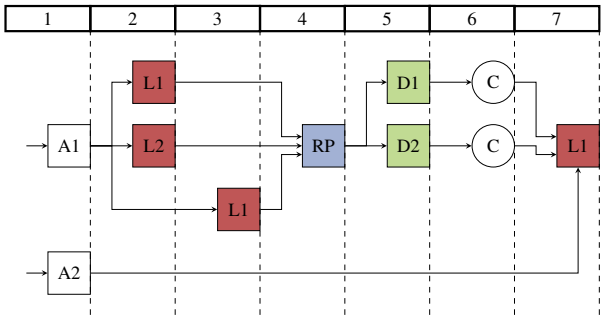


Fig. 4. VLCW execution schedule of the first-stage approximated SUSAN SI implementation with merged evaluation and decision RCs

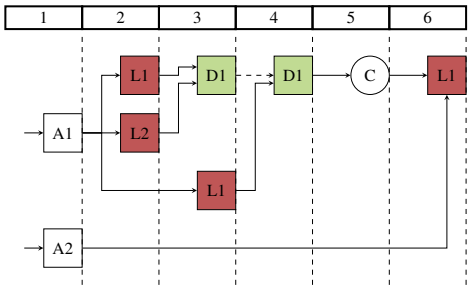


Fig. 5. VLCW execution schedule of the second-stage approximated SUSAN SI implementation with reduced bit width RCs

IV. Evaluation

A. Experimental Setup

The reconfigurable processor prototype is implemented on a Virtex-7 evaluation board (VC707) and runs at 50 MHz. We envision an ASIC implementation where the GPP core, interconnect and load/store units of the accelerator fabric are implemented as hard logic, whereas the RCs are realized using an embedded FPGA. To compensate for the difference in maximum clock frequency, the length of a SI execution phase is set to four CPU cycles to allow enough timing budget within the RCs. The reconfigurable processor uses a Gaisler LEON3 core [15], which is extended to support decoding SIs in unused regions of the original SPARCv8 instruction set [16]. SIs are executed using the attached accelerator framework, consisting of five RCs of 1200 LUTs and 20 DSP slices each. Both load/store units feature a 128bit memory interface to 4 MiB of SRAM, which is also accessible through the main AHB bus of the processor. Runtime reconfiguration of the RCs is possible using the reconfiguration controller, which also allows automatic on-demand loading of bitstreams.

The SUSAN implementation of the MIBench suite is used as baseline software reference [17]. Edge evaluation and classification is replaced by a corresponding SI to invoke the hardware implementation. Finally, edges are marked in the input image in software. Compared to processing in software,

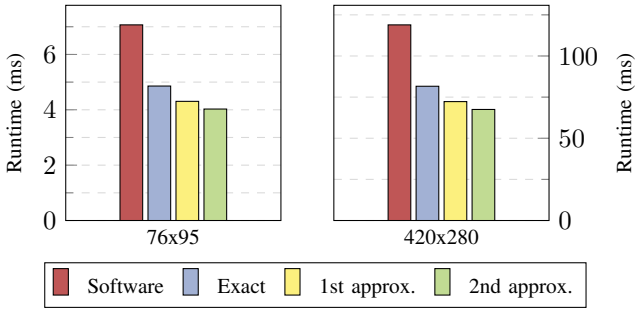
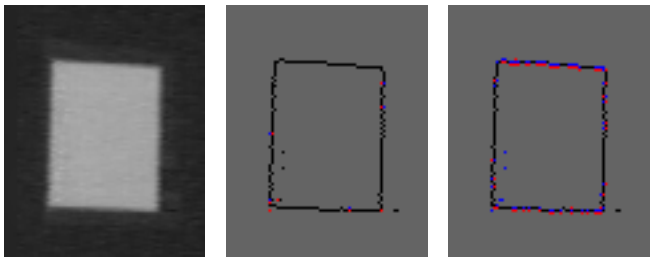


Fig. 6. Execution time in SW and different HW implementations



(a) Rectangular test image, 76x95 px (b) First stage approximation (c) Second stage approximation

Fig. 7. Edges detected in a small test image, misdetected edges marked in red/blue (additional/missing w.r.t. exact implementation)

TABLE I
ERROR ANALYSIS FOR A 420X280 IMAGE (*"Hundertwasserhaus"*)

	1st stage approx.		2nd stage approx.	
	absolute	relative	absolute	relative
edge direction	2807	2.39%	3805	3.24%
edge presence	1924	1.64%	2657	2.27%
false-positive edges	976	0.83%	1365	1.16%
false-negative edges	948	0.81%	1292	1.11%

TABLE II
FPGA RESOURCE USAGE BY IMPLEMENTATION VARIANT

Variant	Module	# of RCs	LUTs	DSPs
Exact implementation	Evaluation	2	762	1
(4 RCs total)	Decision	2	748	0
1st stage approximate	Combined	2	837	0
2nd stage approximate	Combined	1	653	0

the exact hardware accelerated implementation has a speedup of 43%.

B. The approximate SUSAN implementation

In addition to the exact hardware implementation, which yields the same results as the software implementation, two approximate versions are evaluated: The first-stage approximate implementation uses LM-2 multipliers [14] and adders with medium approximation settings[1]. The second-stage uses LM-NC multipliers, adders with twice the number of approximated bits, and the interleaved processing mode is used.

Comparing both a simple 76×95 pixels image showing a single rectangle and a 420×280 image (*"Hundertwasserhaus"*), the first and second-stage approximations reduce runtime by $11.4\,\%$ and $17.1\,\%$ (Fig. 6). Note that the total runtime of the algorithm is measured, including image read-in and highlighting of detected edges in the input image. The unaccelerated operations last $3.6\,\mathrm{ms}$ and $99.1\,\mathrm{ms}$ for the small and large test patterns, respectively. When only taking the process of edge detection and classification into account, the performance is improved by $44.6\,\%$ and $67.0\,\%$ for the 1st/2nd-stage approximations. It is noticeable that the pixel processing rate is independent of the image size and almost constant, being 1.42, 1.61 and $1.72\,\mathrm{pixels/s}$ for the exact, 1st-stage and 2nd-stage approximated cases, respectively.

Table I shows an error analysis for the large test image. The number of misdetected edges (edge presence) and incorrectly detected edge directions is evaluated separately. It can be seen that fewer edges are incorrectly predicted because fewer approximation operations are required compared to determining the edge direction. In contrast, the evaluation result of the edge direction is only used if an edge is present, whereby many errors are suppressed.

A visualization of the impact of the two approximation stages is shown in Fig. 7. The input image 7a contains a single rectangle, filtered with Gaussian blur of $x = y = 0.5$ and $2\,\%$ of random noise. The first-stage approximation misdetects a total of 13 edges, with 5 incorrectly detected and 8 not detected. In comparison, the second stage misdetects 27 edges, with 12 false positive and 15 false negative edges. Edges that are not present in the output of the exact implementation are marked in blue, superfluous edges are colored in red. It can be seen that the second-stage approximation misdetects a considerable number of edges one pixel above the correct location, which can be explained as a result of the inverleaved processing mode. Since in such cases the third image row is a duplicate of the second row, the center of mass for an edge is moved to the center of the two upper rows.

The greatest impact of the approximate implementations is seen in resource use as displayed in Table II. The exact implementation requires a total of 4 RCs, consisting of two evaluation modules and two decision modules. Due to the resource limitations of a RC, the modules can't be merged further. The approximate versions however show a greatly reduced resource utilization, therefore both the 1st and the 2nd stage allow to combine the evaluation and decision modules. The first-stage approximate implementations requires 2 RCs for parallel processing of two neighboring kernels. As the second-stage approximation makes use of the interleaved processing mode, one kernel can be evaluated earlier and one single RC is used in a pipelined way. Hence the approximate implementations are especially useful, if the amount of available RCs is not sufficient to hold all modules required

[1] Adder settings $N, R, P = 8, 2, 2$ according to Shafique et. al. [9]

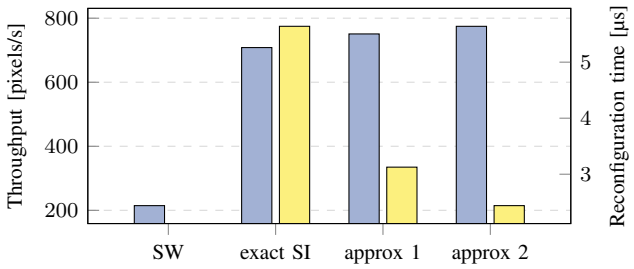


Fig. 8. Performance of exact and approximate SUSAN compared to reconfiguration time

by the application running on the reconfigurable processor. By selecting an approximate implementation, less RCs are required for SUSAN acceleration and allow implementations for other SIs to be loaded concurrently. The resource limit of the RCs can of course be adjusted in the FPGA prototype, but in a potential implementation as an ASIC one will always reach resource limits at a certain point.

C. Runtime Reconfiguration

Reconfiguration during runtime allows to switch between exact and approximate implemenentations when necessary. Our experimental use-case is a mixed-critical application running SUSAN edge detection with a constant frame rate. In addition, the system needs to apply AES-256 encryption to an incoming data stream [18]. The AES implementation requires two RCs to be available, whereas four RCs are used by SUSAN. Fig. 8 shows the required reconfiguration time to switch between the exact, first-stage and second-stage approximate implementations. Loading the second-stage approximate implementation to free up two RCs takes $5.61\,\mathrm{ms}$, which is just $8.31\,\%$ of the processing time of a 420×280 image. In our use-case the additional delay is acceptable for one frame, allowing the encryption to be accelerated by $70\,\%$. Implementing different approximation levels offers a high degree of flexibility, allowing to choose a pareto-optimal operating point between desired accuracy and number of available RCs.

V. Conclusion

In this paper, the applicability of different approximate computing techniques to reconfigurable processors has been investigated. We have shown a significant impact especially when encountering hardware limitations and bottlenecks. Approximate computing allows to overcome such limitations and enables simultaneously acceleration of different algorithms. We performed a detailed analysis of an image processing algorithm and, by applying different approximation methods, achieved significantly improved processing performance of the hardware accelerator while greatly reducing resource consumption. It can be seen that particularly good results can be achieved by approximation techniques specifically adapted and optimized for a given algorithm. For future work, we plan to investigate automatic analysis and approximation of specific hardware accelerator blocks in conjunction with HLS. Based on our results in this paper, we expect that automatic approximation will bring significant performance improvements and substantial resource savings, enabling the acceleration of a wide range of algorithms while reducing implementation effort.

References

[1] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proceedings of the IEEE*, vol. 103, no. 3, pp. 332–354, 2015.

[2] L. Liu, J. Zhu, Z. Li, Y. Lu, Y. Deng, J. Han, S. Yin, and S. Wei, "A survey of coarse-grained reconfigurable architecture and design," *ACM Computing Surveys*, vol. 52, no. 6, pp. 1–39, 2020.

[3] J. Anderson, R. Beidas, V. Chacko, H. Hsiao, X. Ling, O. Ragheb, X. Wang, and T. Yu, "CGRA-ME: An open-source framework for cgra architecture and cad research," in *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2021, pp. 156–162.

[4] C. Xiao and E. Casseau, "Efficient custom instruction enumeration for extensible processors," in *ASAP 2011 - 22nd IEEE International Conference on Application-specific Systems, Architectures and Processors*, 2011, pp. 211–214.

[5] M. Ali and D. Göhringer, "Application specific instruction-set processors for machine learning applications," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022, pp. 1–4.

[6] T. Harbaum, C. Schade, M. Damschen, C. Tradowsky, L. Bauer, J. Henkel, and J. Becker, "Auto-SI: An adaptive reconfigurable processor with run-time loop detection and acceleration," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*, 2017, pp. 153–158.

[7] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 181–192.

[8] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," in *2013 18th IEEE European Test Symposium (ETS)*, 2013, pp. 1–6.

[9] M. Shafique, W. Ahmad, R. Hafiz, and J. Henkel, "A low latency generic accuracy configurable adder," in *Proceedings of the 52nd Annual Design Automation Conference*, ser. DAC '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2744769.2744778

[10] W. Ahmad, B. Ayrancioglu, and I. Hamzaoglu, "Low error efficient approximate adders for fpgas," *IEEE Access*, vol. 9, pp. 117 232–117 243, 2021.

[11] J. Bonnot, A. Mercat, E. Nogues, and D. Ménard, *Approximate Computing at the Algorithmic Level*. Cham: Springer International Publishing, 2022, pp. 109–142. [Online]. Available: https://doi.org/10.1007/978-3-030-94705-7_5

[12] S. M. Smith and J. M. Brady, "Susan — a new approach to low level image processing," *International Journal of Computer Vision*, vol. 23, no. 1, pp. 45–78, May 1997. [Online]. Available: https://doi.org/10.1023/A:1007963824710

[13] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[14] S. Yao and L. Zhang, "Hardware-efficient fpga-based approximate multipliers for error-tolerant computing," in *2022 International Conference on Field-Programmable Technology (ICFPT)*, 2022, pp. 1–8.

[15] Aeroflex Gaisler AB, "LEON3: Multiprocessing CPU Core," https://www.gaisler.com/doc/leon3_product_sheet.pdf.

[16] L. Bauer, M. Shafique, and J. Henkel, "RISPP: A run-time adaptive reconfigurable embedded processor," in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 725–726.

[17] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.

[18] F. Lesniak, F. Kreß, and J. Becker, "Transparent near-memory computing with a reconfigurable processor," in *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, S. Derrien, F. Hannig, P. C. Diniz, and D. Chillet, Eds. Cham: Springer International Publishing, 2021, pp. 221–231.