# LAC: Learned Approximate Computing

Vaibhav Gupta
*Electrical and Computer Engineering*
*UCLA*
Los Angeles, USA
vaibhav22@ucla.edu

Tianmu Li
*Electrical and Computer Engineering*
*UCLA*
Los Angeles, USA
litianmu1995@ucla.edu

Puneet Gupta
*Electrical and Computer Engineering*
*UCLA*
Los Angeles, USA
puneetg@ucla.edu

***Abstract*—Approximate hardware trades acceptable error for improved performance and previous literature focuses on optimizing this trade-off in the hardware. We show in this paper that the application (i.e., the software) can be optimized for better accuracy without losing any performance benefits of the approximate hardware. We propose LAC: learned approximate computing as a method of tuning the application parameters to compensate for hardware errors. Our approach showed improvements across a variety of standard signal/image processing applications delivering an average improvement of 5.82db in PSNR and 0.23 in SSIM of the outputs. This translates to up to 87% power reduction and 83% area reduction for similar application quality. LAC allows the same approximate hardware to be used for multiple applications.**

***Index Terms*—approximate computing, machine learning**

## I. Introduction

Approximate computing trades off quality for better performance in error-tolerant applications like digital signal processing, multimedia, and neural networks [1]. By purposefully lowering voltage [2] or introducing error into computation to simplify the logic [3], [4], the area, energy consumption, and latency of a computation can be significantly reduced at the cost of computation accuracy. However, directly using approximate computing elements in an application can result in noticeable output quality reduction.

Previous works have tried to reduce the overall error of approximate computing elements. This includes finding better ways to simplify the compute logic [5], generating error-compensation circuits [6], [7], or by having parameterized designs [4], [8], [9]. Some works have tried to improve the error profile for a specific application, including finite impulse response [10], [11] and Gaussian smoothing filters [12]. This approach limits the generalizability of the proposed approximate multipliers or adders. Another work has attempted to map approximate computing to specific parts of an application to prevent quality loss [13]. Some recent works [14], [15] have observed that customized training of neural networks can improve inference accuracy with inaccurate hardware. We generalize this observation to near arbitrary applications.

In this work, we propose LAC: learned approximate computing, where the *application* kernel learns the approximate computing element. Using machine learning techniques, the coefficients of an application kernel automatically adjust to the error properties to maximize output quality without changing the computation. Our contributions are as follows:

- We develop the LAC methodology, not limited to just machine learning, to train almost arbitrary parameterizeable application kernels.
- We show that LAC, for example in applications trained for approximate multipliers, improves structural similarity (SSIM) by up to 0.55 and peak signal-to-noise ratio (PSNR) by up to 9.68dB. The improved quality allows 87% power reduction and 83% latency reduction for the same application quality.
- Learned approximate computing approach allows optimal performance and quality without needing to change the approximate computing hardware or computations performed across different applications.

In Section II, we give an overview of LAC. Section III describes our evaluation setup while Section IV discusses LAC experimental results and we conclude in Section V.

## II. Learned Approximate Computing Methodology

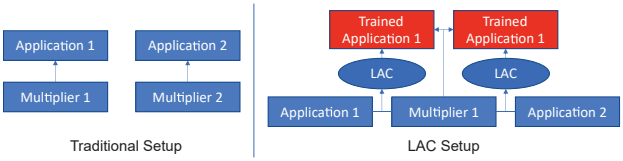



Fig. 1: Difference between traditional and LAC setups. LAC focuses on optimizing the application kernels rather than optimizing the hardware approximations.

Most previous works try to reduce the error of the approximate hardware for a particular application, sacrificing the performance of the approximate hardware in other applications. As a result, different applications require separate approximate hardware for the best quality and performance, as is shown in Figure 1. In this paper, we focus on modifying the application coefficients given the approximate hardware. The overall application algorithm still looks the same in terms of the types and order of computation performed, and the same approximate hardware can be used for different applications improving the versatility of approximate computing.

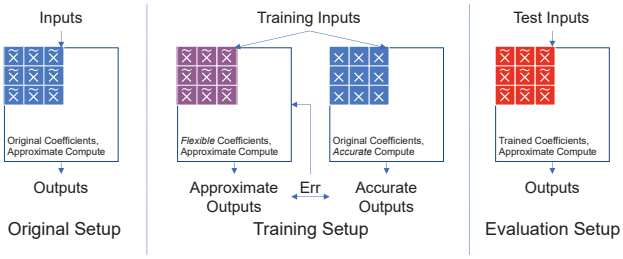



Fig. 2: Overview of LAC.

Error distributions in approximate computing units are strongly input-dependent. An example is the Kurkarni [3] multipliers, which only have errors when multiplying three by three within

a 2-bit multiplier and no error for any other multiplication. One way to get around this issue is by modifying the application coefficients so that computations avoid the high-error regions of the approximate hardware. However, approximate compute units differ in their error characteristics. Dynamic Range Unbiased Multiplier (DRUM) [16] lowers average error at the cost of introducing error in more multiplications. As a result, manually tuning the coefficients of an application to achieve optimal accuracy becomes difficult. LAC tries to simplify this process by training the coefficients. If a learning algorithm has access to the expected accurate result and all the properties of the approximate hardware, it should be possible to avoid the high-error regions. LAC is *not* limited to machine learning type applications and works as long as the application kernel is parameterizable.

Figure 2 demonstrates the overall setup of LAC. Before training, application performance is verified by using the traditional setup, where computation uses the approximate units and the application itself is unaltered. During training, inputs enter an approximate branch and an accurate branch. The accurate branch keeps the original application coefficients and produces precise results. The approximate branch accurately models the approximate hardware while using flexible coefficients. The difference between the two branches is then used as the error to train the coefficients in the approximate branch. Training is performed using any optimization solver (Matlab surrogateopt optimizer in our case). The optimizer used will depend on the size of the application kernel and the nature of the approximate computing unit involved (e.g., integer vs. floating point).

## III. Experimental Setup

### A. Approximate hardware

| Multiplier | Variant | Area | Power |
|---|---|---|---|
| Kulkarni [3] | 8-bit | 0.17 | 0.10 |
| | 16-bit | 0.60 | 0.68 |
| ETM [4] | 8-bit | 0.14 | 0.04 |
| | 16-bit | 0.50 | 0.25 |
| DRUM [16] | 16-bit-4 | 0.25 | 0.12 |
| | 16-bit-6 | 0.39 | 0.29 |
| EvoApprox [8] | mul8u_JV3 | 0.03 | 0.02 |
| | mul8u_FTA | 0.07 | 0.04 |
| | mul8u_185Q | 0.13 | 0.09 |
| | mul8s_1KR3 | 0.07 | 0.02 |
| | mul8s_1KVL | 0.21 | 0.12 |
| | mul16s_GK2 | 1.01 | 0.89 |
| | mul16s_GAT | 0.74 | 0.58 |

TABLE I: Multiplier summary. Performance numbers are normalized to 16-bit multipliers.

We chose to focus on approximate multipliers since they add the most energy and time delay costs. The multipliers we use are summarized in Table I. We used a variety of multipliers with different error and performance characteristics including multipliers from the EvoApprox library [8], the Kulkarni multiplier [3], the error-tolerant multiplier (ETM) [4] and the Dynamic Range Unbiased Multiplier (DRUM) [16]. We used both an 8-bit and 16-bit implementation of the Kulkarni multiplier, an 8-bit ETM with the bits split at k = 4 and a 16-bit ETM with the bits split at k = 8, and used two implementations of the 16-bit DRUM with k = 4 and k = 6. Signed multiplications are achieved by using the method proposed in [16], which computes the absolute value and sign of the result separately. Area and power numbers in Table I are normalized to accurate 16-bit multipliers. Coefficients are constrained to $[0,2^m-1]$ for applications using unsigned values, and $[-(2^m-1),(2^m-1)]$ for signed values, where $m$ is the bit width of the multiplier.

### B. Applications

| Application | Coefficients |
|---|---|
| Gaussian blur | 3x3 |
| Edge detection | 3x3 |
| Image sharpening | 3x3 |
| Discrete Cosine Transform | 8x8 |
| Discrete Fourier Transoform | 12x12(complex) |

TABLE II: Application summary

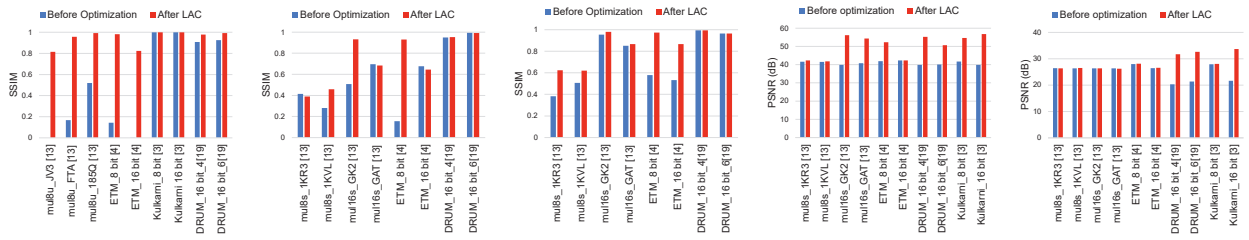
Table II summarizes the applications used for evaluating LAC. Performance is first evaluated for three applications using 3x3 filters. The three filters include the 3x3 versions of Gaussian blur for image blurring, Sobel filter for edge detection and Laplacian filter [17] for image sharpening. Gaussian blur uses unsigned values, so the unsigned multipliers are used for the experiments, while the other two use signed values in the filters. For image sharpening using the Laplacian filter, the outputs of the filter are scaled to $[0,255]$ and then added to the original image for the final result. Average Structural Similarity Index (SSIM) [18] is used to measure the performance.

To analyze the performance of more complicated applications, we also trained the Discrete Cosine Transform (DCT) and the Discrete Fourier Transform (DFT). The DCT uses a quality level of 50, as described in [19], and requires an 8x8 filter. The size of the DFT had to be reduced to 12x12 as larger sizes failed to converge. The scalability of our approach beyond the problem sizes demonstrated here might be addressed in future studies. The coefficients for DCT and DFT are scaled up by $2^m$ and then rounded to fill the integer input range. The final values are scaled down by $2^{2m}$ since they are performed twice - on the x and y axes. Quality of DCT and DFT are measured using the peak signal-to-noise ratio (PSNR) between outputs using approximate multipliers and outputs using accurate multipliers.

### C. Optimization Solvers

The optimization problem was framed as pure integer optimization - for the blurring, edge detection, and sharpening - since in these cases all the variable weights are constrained to being integers by the input requirements of the multipliers. In the case of the DFT and DCT, weights are scaled to force them into integers. These integer or range constraints were also provided to the optimizer.

The Matlab optimizer used is a gradient-free optimizer that works well for applications with a relatively small number of coefficients, but it can run into runtime issues for larger applications. We also tried using PyTorch for more efficient training. The training setup is similar to training binarized neural networks [20]. The forward pass uses the approximate compute units to accurately model computation in the approximate hardware. The backward pass uses straight-through estimators and treats the approximate units as if they are accurate. The PyTorch version finishes the training using 55% less time compared to the Matlab version when both are run on an Intel i5-11400 CPU when training a 12x12 DFT (more details in Section III). The applications used in Section III all finish in a reasonable time, so all the evaluations use the Matlab




(a) Gaussian blur SSIM (b) Edge detection SSIM (c) Sharpening SSIM (d) DCT PSNR (e) DFT PSNR

Fig. 3: Quality improvements of Gaussian edge detection, image blur and image sharpening

optimizer, but PyTorch or other gradient-based optimizers are also possible for larger applications that run into runtime issues.

## IV. Results and Discussion

In this section, we experimentally demonstrate the benefits of learned approximate computing. We show that LAC dramatically improves application quality on the same approximate hardware or conversely allows for use of cheaper/more energy-efficient approximate hardware for the same application quality. We also show that, as expected, LAC learns characteristics of hardware approximation via training rather than learning the characteristics of the input data itself. Unless otherwise mentioned, application kernels are trained on 100 images drawn randomly from the CIFAR10 dataset, and application quality is measured on 20 CIFAR10 images different from the training set.

### A. Application quality improvement

Figure 3 shows the quality improvements from LAC. As is mentioned in Section III, Gaussian blurring, edge detection, and image sharpening use SSIM for training and evaluation, while DCT and DFT use PSNR. On average, SSIM improves by 0.55, 0.16, and 0.14 for the three applications.[1] PSNR improves by 9.68dB and 3.50dB for DCT and DFT respectively. Quality improvements from LAC depend on the application and characteristics of the approximate multipliers. The improvement is dramatic in many cases, making previously unusable approximate hardware acceptable (for example, see results on one sample image in Figure 5).

### B. Hardware efficiency impact of LAC

The improved quality from LAC results in a better quality-performance tradeoff for all the applications. Figure 4 shows the tradeoff before and after optimization using LAC. The dashed line shows the Pareto optimal points. The SSIM values are taken as the geometric mean of Gaussian blur, Sobel filter, and image sharpening algorithms, and the PSNR values are taken as the geometric mean of DCT and DFT. Optimization with LAC results in 87%, 84%, and 63% power reduction and 83%, 64%, and 44% area reduction with similar SSIM values. For DCT and DFT, LAC achieves 4.1dB improvements for the same area and power on a Pareto optimal point or 9.1dB peak PSNR improvement. The same behavior can be seen in the sample images in Figure 5. LAC reduces the gap between expensive and cheap approximate multipliers and allows us to use cheaper and less accurate multipliers.

### C. LAC Sensitivity to Training Data

To check whether our approach is sensitive to the training dataset, we used three training datasets: 100 images from the CIFAR-10 dataset [21], 100 images from the MNIST dataset, and 100 random 32x32 values using the same [0,255] range as the other two. Validation is performed on the same 20 CIFAR-10 images separate from the training set for all three cases. Figure 6 shows the comparison results without a clear trend separating the three choices. This indicates that LAC training learns the hardware approximations rather than the data itself, as intended. Expanding the size of training and validation dataset size had little impact on the results, so we stuck to the smaller size for faster training.

### D. Importance of application-specific training

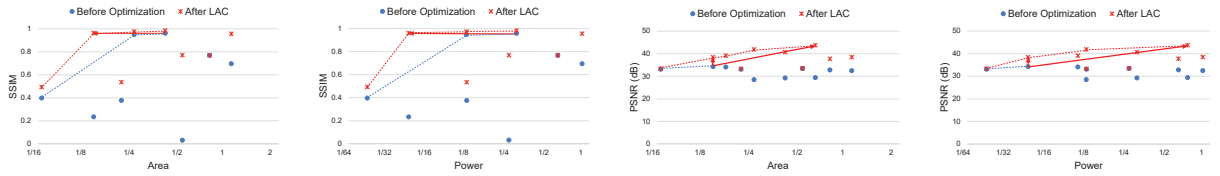
We used DCT to demonstrate the importance of application-specific training. The DCT is trained as a stand-alone operator and as a part of the JPEG compression algorithm. When training alone, LAC tries to maximize the PSNR between DCT outputs using accurate and approximate multipliers. When training as part of the JPEG compression, LAC tries to maximize SSIM between JPEG compressed images using approximate multipliers for the DCT transforms and the original image.

Figure 3d shows the performance of training the DCT alone, where the PSNR of DCT outputs improves after training. However, training the coefficients within JPEG compression improves SSIM by 0.05 on average while training DCT only improves SSIM by 0.008, as seen in Figure 7. Training the coefficients within an application allows improvements that aren't achievable when training the approximate kernel alone. This highlights the importance of application-specific training.

## V. Conclusions

Approximate arithmetic units are a promising method to reduce energy, cost, and latency in error-tolerant applications. So far, research has focused on the optimization of the hardware approximation to minimize application quality loss. In this work, we flip the argument and propose the learned approximate computing approach wherein the application kernels are optimized to improve application quality in presence of approximate hardware. We have shown improvements of 0.23 in SSIM and 5.82db in PSNR on average across a variety of signal/image processing applications, showing the utility of LAC as an approach to making the use of approximate hardware more broadly viable. The improved performance enables 87% power

[1]Note that SSIM lies between 0 and 1 with 1 being best.




(a) SSIM-area improvements (b) SSIM-power improvements (c) PSNR-area improvements (d) PSNR-power improvements

Fig. 4: Aggregated performance improvements. SSIM plots are for Gaussian blurring, Sobel filter, and image sharpening that are trained using SSIM. PSNR plots are for DCT and DFT that are trained using PSNR

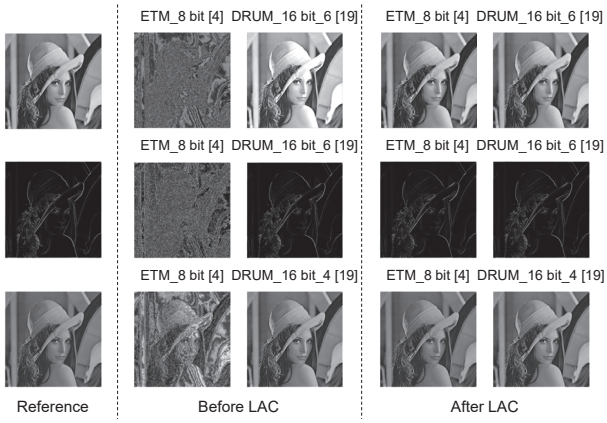



Fig. 5: Quality improvements of Gaussian edge detection, image blur and image sharpening. Figures on the right uses Pareto optimal multipliers with the highest SSIM before optimization.

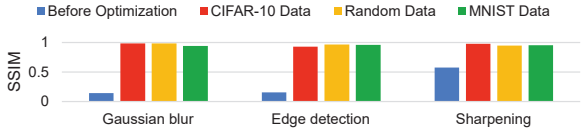



Fig. 6: Sensitivity of results to training data choice. The multiplier used here is the 8-bit ETM multiplier [4].

reduction and 83% area reduction with no drop in application quality. LAC bridges the gap between different approximate hardware and enables the same hardware in multiple applications.

## References


[1] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan, "Macaco: Modeling and analysis of circuits for approximate computing," in *ICCAD 2011*, 2011, pp. 667–673.

[2] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *ISLPED 1999*, 1999, pp. 30–35.

[3] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSID 2011*, 2011, pp. 346–351.

[4] Khaing Yin Kyaw, Wang Ling Goh, and Kiat Seng Yeo, "Low-power high-speed multiplier for error-tolerant application," in *EDSSC 2010*, 2010, pp. 1–4.

[5] F. Farshchi, M. S. Abrishami, and S. M. Fakhraie, "New approximate multiplier for low power digital signal processing," in *CADS 2013*, 2013, pp. 25–30.

[6] M. Masadeh, O. Hasan, and S. Tahar, "Using machine learning for quality configurable approximate computing," in *2019 DATE*, 2019, pp. 1575–1578.

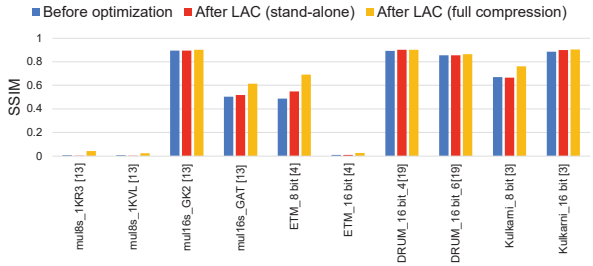




Fig. 7: JPEG compression comparison between training the DCT alone and training the DCT in JPEG compression


[7] ——, "Machine learning-based self-compensating approximate computing," in *2020 SysCon*, 2020, pp. 1–6.

[8] V. Mrazek, R. Hrbacek, Z. Vasicek, and L. Sekanina, "Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods," in *DATE 2017*, 2017, pp. 258–261.

[9] Z. Vasicek and L. Sekanina, "Evolutionary design of approximate multipliers under different error metrics," in *DDECS 2014*, 2014, pp. 135–140.

[10] A. Bonetti, A. Teman, P. Flatresse, and A. Burg, "Multipliers-driven perturbation of coefficients for low-power operation in reconfigurable fir filters," *TCAS1*, vol. 64, no. 9, pp. 2388–2400, 2017.

[11] G. Sreegul and T. S. Bindiya, "An approximation algorithm for reducing the number of non-zero bits in the filter coefficients," in *SCEECS 2020*, 2020, pp. 1–6.

[12] A. Jaiswal, B. Garg, V. Kaushal, and G. K. Sharma, "Spaa-aware 2d gaussian smoothing filter design using efficient approximation techniques," in *VLSID 2015*, 2015, pp. 333–338.

[13] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," *SIGPLAN Not.*, vol. 49, no. 10, p. 309–328, Oct. 2014. [Online]. Available: https://doi.org/10.1145/2714064.2660231

[14] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: Energy-efficient neuromorphic systems using approximate computing," in *ISLPED 2014*, 2014, pp. 27–32.

[15] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "Approxann: An approximate computing framework for artificial neural network," in *DATE 2015*, 2015, pp. 701–706.

[16] S. Hashemi, R. I. Bahar, and S. Reda, "Drum: A dynamic range unbiased multiplier for approximate applications," in *ICCAD 2015*, 2015, pp. 418–425.

[17] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design and Test, special issue on Computing in the Dark Silicon Era*, 2016.

[18] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, 2004.

[19] K. Cabeen and P. Gent, "Image compression and the discrete cosine transform." [Online]. Available: https://www.math.cuhk.edu.hk/ lmlui/dct.pdf

[20] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[21] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009.