Learned Approximate Computing: Algorithm Hardware Co-optimization

Egor Glukhov, Tianmu Li, Vaibhav Gupta, Puneet Gupta
Department of Electrical and Computer Engineering University of California, Los Angeles
e.glu@ucla.edu, litianmu1995@ucla.edu, vaibhav22@ucla.edu, 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 and the hardware can be co-optimized to achieve the best quality-performance tradeoff. We propose LAC: learned approximate computing to optimize the algorithm and approximate hardware at the same time to maximize quality of output. Our approach allows automatic selection of approximate computing hardware while achieving similar quality as dedicated training for a single hardware configuration. Our improved training algorithm allows simultaneous hardware selection and application optimization without additional runtime overhead. Multi-hardware setup chooses a separate approximate hardware for each part of an application which allows for more hardware configurations and further improves quality.**

I. Introduction

Approximate computing trades some computation accuracy to improve area and energy efficiency. Some examples of approximate computing include introducing uncertainty by lowering voltage [2] or introducing input-dependent error to simplify logic design [3], [4]. Despite finding interest in error-tolerant applications like deep learning using neural networks [5], the inherent uncertainty in approximate computing has prevented wide adoption in mainstream applications.

Multiple approaches have been proposed to improve the accuracy of approximate computing, preserving area and energy efficiency. The methods include compensating errors with additional circuitry [6], [7], finding better accuracy-performance tradeoffs through better logic design [8] or allowing multiple accuracy levels [4], [9], [10]. As approximate computing relies on uncertainties to achieve better performance, reducing the overall error runs into diminishing returns.

To further enhance the accuracy of approximate computing in practice, there are two aspects to improve: the approximate computing hardware and the target application. Most recent works focus on optimizing approximate computing hardware for specific applications. The approaches include reducing the approximation error for computations prevalent in an application [11]–[13], or limiting approximation to specific error-tolerant components of applications [14]. Optimizing the approximate computing hardware for one or two particular applications means that the hardware may not be suitable for other applications. Efforts on optimizing the latter have been sparse [1], and most past works have focused on neural networks [15], [16]. With all the different approximate computing options, it also becomes challenging to choose the setup that is most suitable for a given application or a specific performance or quality constraint.

This is a modified and extended version of the previous work [1].

In this work, we propose LAC: learned approximate computing. Instead of optimizing the approximate computing hardware for a fixed application, we optimize the *application* kernel for both a fixed and trained hardware configuration. The key observation is that the application and the approximate computing hardware can be co-optimized to achieve optimal performance while minimizing manual intervention. During the process, we also allow searching for the approximate hardware configuration that is most suitable for the application. This way, the coefficients of an application kernel are automatically adjusted to the error properties of the most appropriate approximate hardware configuration. Our contributions are as follows:

- We develop the LAC methodology to train almost arbitrary parameterizable application kernels.
- In cases where the hardware can be changed, LAC searches for the optimal hardware while tuning the algorithm.

II. Fixed hardware LAC

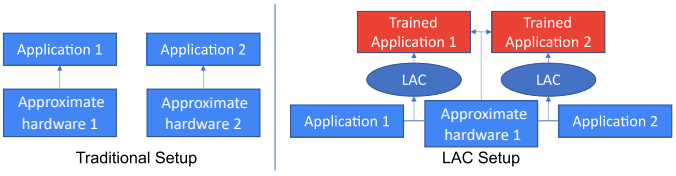


Fig. 1: Difference between traditional and LAC setups. LAC focuses on optimizing the application kernels rather than optimizing the hardware approximations.

While most previous works try to reduce the error of the approximate hardware for a particular application, the application itself is mostly untouched. The traditional setup implies that the approximate hardware may not be a good choice for other applications, and different applications require separate approximate hardware for the best quality and performance. This section will focus on a fixed hardware version of LAC where applications are trained for fixed hardware.

A. Training applications for a fixed hardware

In a fixed hardware setup, LAC tries to make the application learn the approximate computing hardware by training its pa-

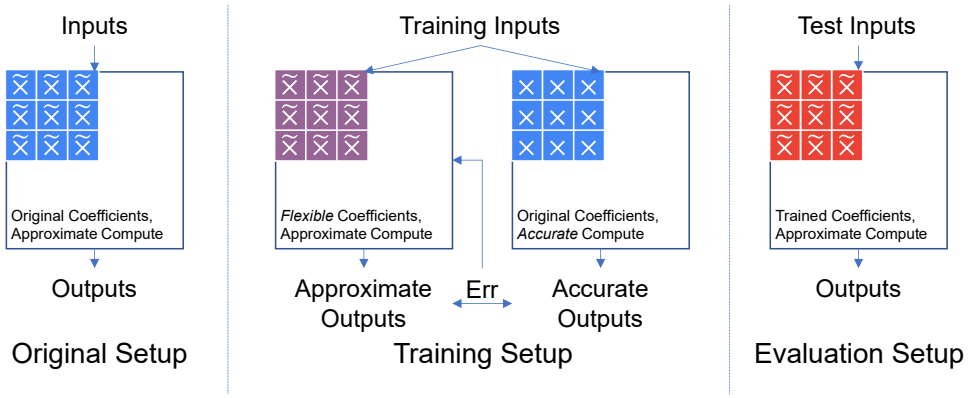


Fig. 2: Overview of LAC for fixed hardware.

rameters. Fig. 1 compares a traditional approximate computing setup with LAC. By training the application to better match the properties of the approximate multiplier, LAC aims to improve application performance for a specific multiplier and allow the reuse of approximate computing hardware across multiple applications. The motivation behind LAC is that 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. If an application is dominated by multiplying three by three, the results have very high errors. Conversely, if the application does not contain three in any 2-bit sections, multiplications can be completely accurate. This discrepancy means that approximate computing hardware does not have consistent performance across multiple applications since the values used between applications vary greatly. One way to get around this issue is by modifying the application coefficients so that the computation performed avoids the high-error regions of the approximate hardware. However, approximate compute units differ in their error characteristics. Dynamic Range Unbiased Multiplier (DRUM) [17] lowers average error at the cost of introducing error in more multiplications. Even if it is possible to achieve better overall performance with a lower average error, manually tuning the coefficients of an application to achieve that 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. The only constraint is that the application kernels should be parameterizable and hence be able to optimize using standard mathematical optimization approaches.

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 an optimization solver. 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. Evaluation on Fixed Hardware

A. Approximate hardware

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

Multiplier	Variant	Area	Power
ETM [4]	8-bit	0.14	0.04
	16-bit	0.50	0.25
DRUM [17]	16-bit-4	0.25	0.12
	16-bit-6	0.39	0.29
EvoApprox [9]	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

We choose to study approximate multipliers since they add the most energy and time delay costs, as compared to the other arithmetic operations. The multipliers we use are summarized in Table I. We use a subset of multipliers from the EvoApprox library [9] since the well-defined error metrics provided a clear baseline for comparing their performance in different applications. We also demonstrate improvements when using more widely used multipliers that were intentionally designed for high performance - the error-tolerant multiplier (ETM) [4] and the Dynamic Range Unbiased Multiplier (DRUM) [17]. We use an 8-bit ETM with the bits split at k = 4 and a 16-bit ETM with the bits split at k = 8 and use two implementations of the 16-bit DRUM with k = 4 and k = 6. Area and power numbers in Table I are normalized to accurate 16-bit multipliers.

B. Applications

Table II summarizes the applications used for evaluating LAC. Performance is first evaluated for three applications

TABLE II: Application summary

Application	Coefficients
Gaussian blur	3x3
Edge detection	3x3
Image sharpening	3x3
Discrete Cosine Transform	8x8
Discrete Fourier Transform	12x12(complex)
Inversek2j [18]	4

using 3x3 filters. The three filters include the 3x3 versions of Gaussian blur for image blurring, Sobel filter for edge detection and Laplacian filter [19] 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. Coefficients are constrained to $[0, 255]$ for applications using unsigned values, and $[-255, 255]$ for signed values. Average Structural Similarity Index (SSIM) [20] is used to measure the performance before and after training using LAC since the applications produce image outputs. To adjust the final output to the $[0, 255]$ range, bit shift is used in both accurate and approximate computes. The bit shift amount is chosen such that the maximum of bit shifted output is 255. For image sharpening using the Laplacian filter, the outputs of the filter are added to the original image for the final result.

To analyze the performance of more complicated applications, we also train the Discrete Cosine Transform (DCT) and the Discrete Fourier Transform (DFT). The DCT uses a quality level of 50, as described in [21], and requires an 8x8 filter. For DFT, we used 12x12 matrix. Both DCT and DFT contain floating-point coefficients, so the coefficients are scaled up by 2^m and then rounded to fill the integer input range. m is the bit width of the multiplier. The final values are scaled down by the 2^m for DCT and 2^{2m} for DFT since DFT is performed twice on the x and y axes. The 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. Coefficients are constrained to $[0, 2^m - 1]$ for applications using unsigned values, and $[-(2^m - 1), (2^m - 1)]$ for signed values.

We also include Inversek2j from AxBench [18] as an application that does not operate on image inputs. Inversek2j computes the inverse kinematics for a 2-joint arm, which is useful in robotic applications. The quality of Inversek2j is measured using relative error. For SSIM and PSNR, a higher value indicates a better quality, while the opposite is true for relative error.

C. Dataset

We use the CIFAR-10 dataset [22] as the input for all of the image applications. Models are trained on 100 training images, and evaluated on the 20 test images. Inversek2j uses 1000 train samples and 200 test samples from AxBench [18].

D. 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 initial LAC implementation [1] is performed using the Matlab surrogate solver that runs into runtime issues for larger applications. To resolve the runtime issue and allow integration with the search component in the trained hardware setup, described in Section IV, we migrated to the gradient-based Adam optimizer in PyTorch. During the training process, we keep a high-precision floating-point copy of the weights and quantize the weights to integers on the fly, similar to the straight-through estimator [23] used for training quantized neural networks. To further speed up the simulation of the approximate computing hardware, we implement parallel versions of the approximate multipliers to spread the work across multiple CPU cores.

E. Results

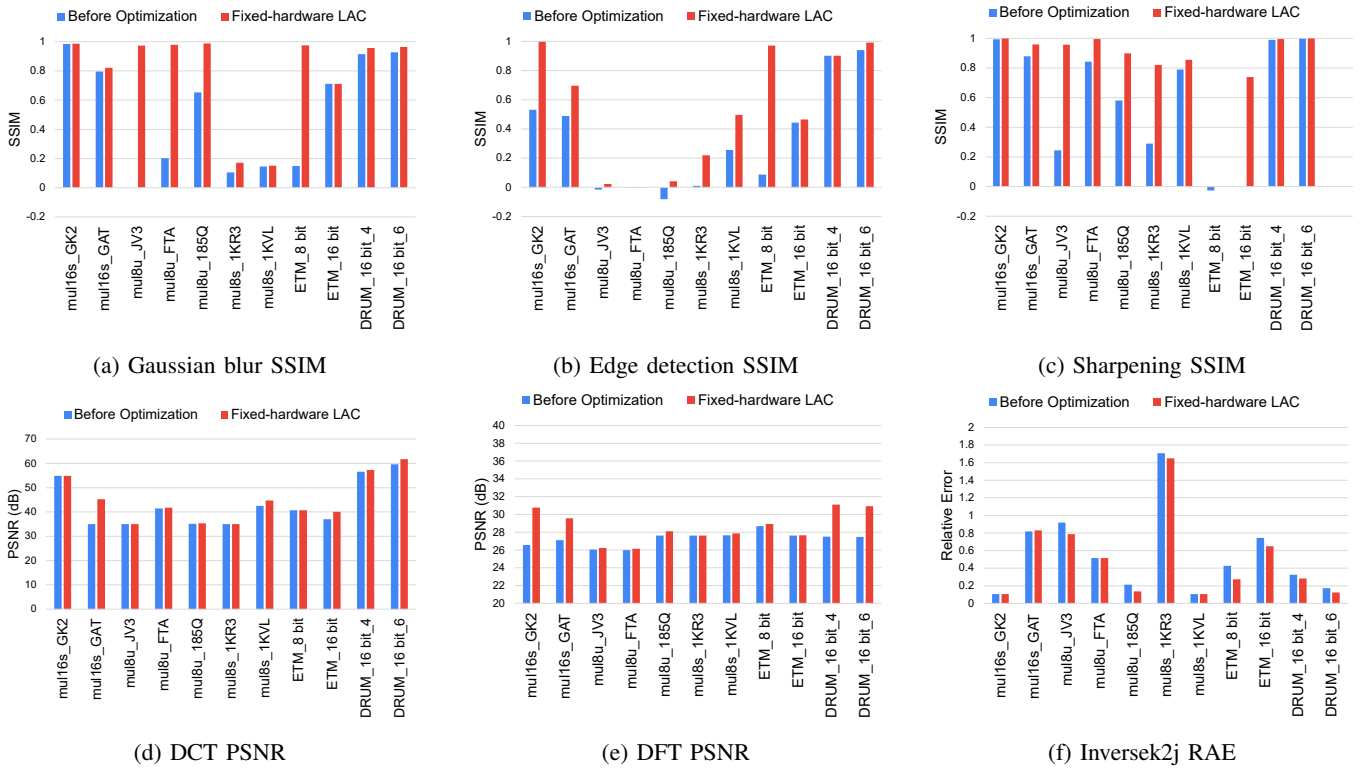
1) Application quality improvement: Figure 3 shows the quality improvements from LAC. While some applications originally use signed parameters, we use both unsigned and signed multipliers for all applications as even unsigned multipliers can benefit from LAC. As is mentioned in Section III-B, Gaussian blurring, edge detection, and image sharpening use SSIM for training and evaluation, while DCT and DFT use PSNR. On average, SSIM improves by 0.28, 0.20, and 0.24 for the three applications.[1] PSNR improves by 1.73dB and 1.36dB for DCT and DFT respectively. For Inversek2j, the relative error is reduced by 0.054 on average. Quality improvements from LAC depend on the application and characteristics of the approximate multipliers. Some multipliers with lower quality before training happen to have large errors when using the original coefficients in the application. For those multipliers, LAC is able to avoid the high-error points in those multipliers and achieve higher quality. The improvement is dramatic in many cases, making previously unusable approximate hardware acceptable.

2) Hardware efficiency impact of LAC: The improved quality from LAC results in a better quality-performance tradeoff for all the applications. Fig. 4. compares the output quality before and after LAC optimization for two approximate multipliers. While the results before optimization favor the more expensive approximate multipliers, results after LAC optimization are much closer and allow us to use the cheaper and less accurate multipliers.

IV. Trained hardware LAC

While training the application for a fixed hardware configuration enables hardware reuse between different applications, it is overly restrictive when designing hardware for an application. For hardware-application co-optimization, we allow searching between hardware configurations. For applications that can choose between different approximate computing hardware, LAC automatically chooses the optimal configuration for a given quality or performance constraint.

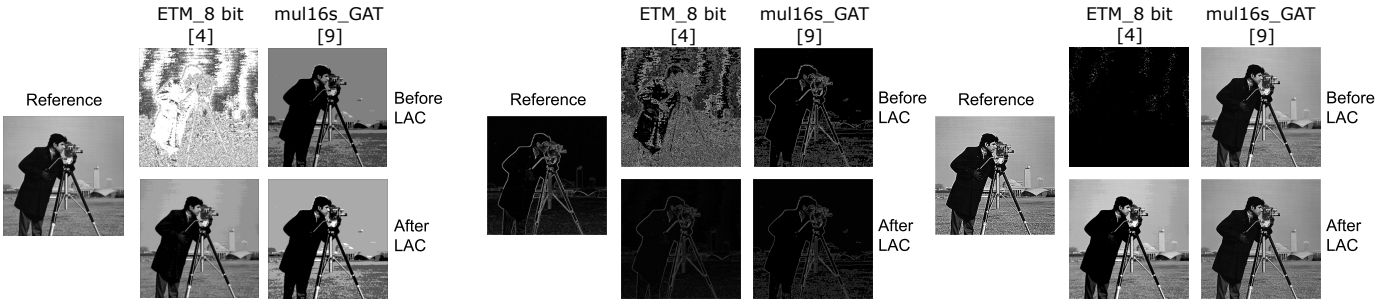
[1]Note that SSIM lies between -1 and 1 with 1 being best.



(a) Gaussian blur SSIM (b) Edge detection SSIM (c) Sharpening SSIM

(d) DCT PSNR (e) DFT PSNR (f) Inversek2j RAE

Fig. 3: Quality improvements of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j. For (a)-(e), higher is more accurate. For (f), lower is more accurate. Absence of a bar represents close to zero SSIM.



(a) Gaussian blur (b) Edge detection (c) Laplacian sharpening

Fig. 4: Quality improvements of (a) Gaussian blur, (b) edge detection, and (c)image sharpening. Figures on the right use Pareto optimal multipliers with the highest SSIM before optimization.

Fig. 5 illustrates the overall structure of LAC for a trained hardware scenario. The application training component uses the same structure as LAC for fixed hardware, which trains the coefficients of an application to minimize the difference between the outputs from approximate hardware and accurate hardware. Multiple approximate hardware are trained at the same time, each with its own set of trainable parameters. The selector at the end decides the one to choose that maximizes application performance. The optimization goals here are similar to a neural architecture search (NAS) setup. In neural architecture search, the search algorithm needs to find the optimal network architecture parameters under some constraints from several different options. To apply NAS methods to LAC, we need to replace the network architecture options with approximate hardware options.

To this end, we use the Binary Gate proposed in Proxyless-Nas [24]. Fig. 6 demonstrates the structure of the binarized gate with three inputs, which acts as the ”Search” component in Fig. 5. In the original ProxylessNas setup, each binary gate is used to choose between multiple layer alternatives for a layer. In LAC, we use it to orchestrate between different approximate multipliers. Furthermore, a binary gate is incorporated into the training process, tuning its weights while LAC optimizes application coefficients. Weight values represent the preference of each multiplier. NAS explores the design space by allowing the binary gate to stochastically select a different approximate multiplier at each new iteration. It does not guarantee convergence to the global optimum but

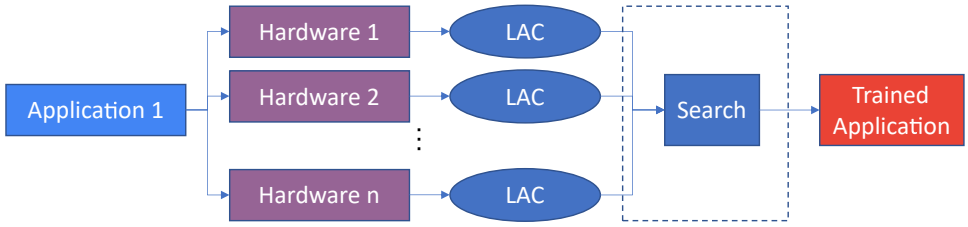


Fig. 5: Overview of LAC for trained hardware.

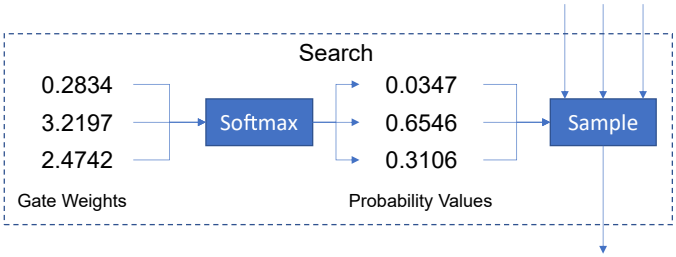


Fig. 6: Illustration of the binarized gate used for choosing between different hardware outputs.

has shown effectiveness in a variety of applications [25], [26]. Most importantly NAS enables co-optimization of applications with hardware.

Before training, the binarized gate is initialized with the same weight value assigned to each path, where each path represents the application's output using a specific approximate multiplier. In the forward pass, a Softmax function converts the weight values into probability values. We sample two of the paths in each cycle using the probability values. The sampled output is then used to calculate the loss. In the backward pass, the application coefficients and the binary gate weights are treated differently. For the application coefficients, the output loss is backpropagated to both of the sampled paths to update the coefficients of two corresponding multipliers. Compared to single path propagation, this approach improves application training, which allows NAS results to reach brute-force search results. For the binarized gate weights, the output loss is backpropagated to all the weight values weighted by the probability values. After training, a single best-performing path becomes the chosen approximate hardware configuration.

We choose the binarized gate setup for the following reasons:

- Performance. The binarized gate is a point-wise function that has a low computation cost regardless of the complexity of the applications.
- Scalability. The binarized gate setup allows scaling to more complicated applications that can be divided into multiple stages. For an application with n stages, where each stage can choose from k different approximate computing hardware configurations, there are k^n distinct options to choose from. Searching for the optimal hardware while training the application coefficients would incur a k^n time penalty compared to only training the coefficients. On the other hand, the binarized gate removes that penalty, as it is training two configurations (two paths) at each iteration regardless of n and k. NAS still explores the whole design space, since different stages are separated and are controlled by separate binarized gates.

During training, we use Eq. 1 to calculate the loss during training. x is the input value, w is the application coefficients to be trained, and w_0 is the original application coefficients to calculate the target for optimization. $f(x, w_0)$ calculates the results from accurate hardware using the original coefficients, and $f_a(\cdot)$ is the function to simulate the approximate computing hardware. $b(\cdot)$ then selects one of the possible choices based on the current weights in the binary gate. $L(\cdot)$ represents the output quality of the application and can be peak signal-to-noise ratio (PSNR), structure similarity index measure (SSIM), or something else.

$$L_a(x, w, t) = L(b(f_a(x, w)), f(x, w_0)) \tag{1}$$

The loss performance in Eq. 1 only searches for optimal accuracy without performance considerations. In the trained hardware setup, for cases where there is a performance constraint (area/power), we reduce the search space to only include multipliers that satisfy the performance constraint, instead of using the regulatory loss term as in the original ProxylessNas setup. Performance-constraint-based pruning of search space is only for the trained-hardware NAS where a single multiplier type is chosen for the entire application. Hence any multiplier that violates the performance constraint need not be considered within the NAS.

We also evaluate the search results when different components of an application can use different multipliers. Performance constraint pruning is not applied here as multipliers that are larger than the constraint may satisfy the average area constraint if mixed with cheaper ones. As approximate multipliers make distinct tradeoffs in their error profiles, different multipliers can potentially compensate for their error when used together. To achieve this, we separate applications into multiple layers and assign a binary gate to each layer. Binarized gate weights are updated using gradients from the final output that combines all selection results. This way each hardware is picked depending on errors introduced by other selections. In this setup, only a single path was back-propagated, which allowed for greater runtime benefit. We used two different approaches for layering: serial and parallel NAS.

Serial implementation features multiple stages of an application that are to be executed in series. After each stage, Binary Gate is applied to choose only one hardware option and pass the output to the next stage. The loss of the whole application

is the output of the last stage. Each gate has its own, separate weights which allows for choosing different hardware at each stage. We used serial implementation for the JPEG compression algorithm, which we divided into 3 components: the DCT computation which performs the compression, the inverse DCT computation which generates images from the compressed data, and the computation in between.

In parallel NAS, parts of an application can be executed at the same time. After completion, Binary Gate is applied to each part separately to select multipliers. Outputs of the gates are then combined to generate the final loss. We applied this method to the Gaussian Blur application. Convolution was separated into 9 matrix scalar multiplications, each involving a separate coefficient of 3x3 matrix. We allowed each coefficient in the kernel to use different approximate hardware during convolution, so a total of 9 multiplier types were used. After all matrix operations are finished, results are summed up to complete a convolution.

For evaluating loss in multi-hardware setup, we use Eq. 2. a is the area of the current configuration, a_{th} is the area threshold that is set before training and unchanged. To prevent violation of constraint, regulatory term $L_h(\cdot)$, resembling hinge loss, is added to loss function $L_a(\cdot)$ from Eq. 1. Formula for $L_h(\cdot)$ is shown in Eq. 3. Hinge loss serves to increase loss if the area is above the target, otherwise do nothing. As the regulatory term can often be traded for additional accuracy, we introduce parameter γ that lowers the constraint to ensure meeting the target area requirement. Parameter δ weights hinge loss $L_h(\cdot)$ with accuracy loss $L_a(\cdot)$, aiming to adjust the contribution of the area factor to the overall loss. Both parameters ought to be determined by experimentation.

$$L_b(x, w, t, a, a_{th}) = L_a(x, w, t) + \delta * L_h(a, a_{th}) \quad (2)$$

$$L_h(a, a_{th}) = \begin{cases} 0, & \text{if } a < \gamma * a_{th} \\ a - \gamma * a_{th}, & \text{otherwise} \end{cases} \quad (3)$$

Setups discussed so far make use of the fact that the area is unchanged throughout the training. Other use cases have constraints that are not constant. To evaluate the NAS approach on these problems, we flip the problem by optimizing the area with a lower limit of accuracy. The loss is calculated by Eq. 4, where l_{target} is the constraint. The inverted area is taken as a base loss. Modified hinge loss $L_{hm}(\cdot)$ from Eq. 5 acts as a regulatory term, resembling $L_h(\cdot)$ with a distinction of argument order as this term is designed to be positive. Additionally, we take $\gamma = 1$.

$$L_c(x, w, t, a, l_{target}) = -a - \delta * L_{hm}(L_a(x, w, t), l_{target}) \quad (4)$$

$$L_{hm}(l, l_{target}) = \begin{cases} 0, & \text{if } l > l_{target} \\ l_{target} - l, & \text{otherwise} \end{cases} \quad (5)$$

V. Evaluation on trained hardware

We use the same set of approximate multipliers as Sec. III to evaluate the performance of LAC when performing a hardware search at the same time.

A. Quality Search Results

Fig. 7 compares the output quality with and without hardware training enabled. For all of the tested applications, LAC is able to find the approximate computing setup that has the highest quality after training. Despite training for multiple multipliers at the same time, LAC does not degrade the performance of the best-performing multiplier significantly. Due to the stochastic nature of the search, there can be a slight difference in the output quality of a single multiplier.

B. Performance Search Results

To evaluate the search performance of LAC under a performance constraint, we tested the performance of LAC with area, delay, and accuracy constraints (power constraints generate similar results). Fig. 8 shows the quality-performance tradeoff with an area constraint. The training algorithm can find the approximate hardware with the highest output quality regardless of the area constraint. In Fig. 9 we showcase delay-restricted search using the same setup on a subset of applications for brevity with delay values from Table III. The slight degradation of output quality compared to fixed hardware results is due to inevitable stochastic selections of incorrect multipliers in the middle of training, causing an under-training of the correct multiplier.

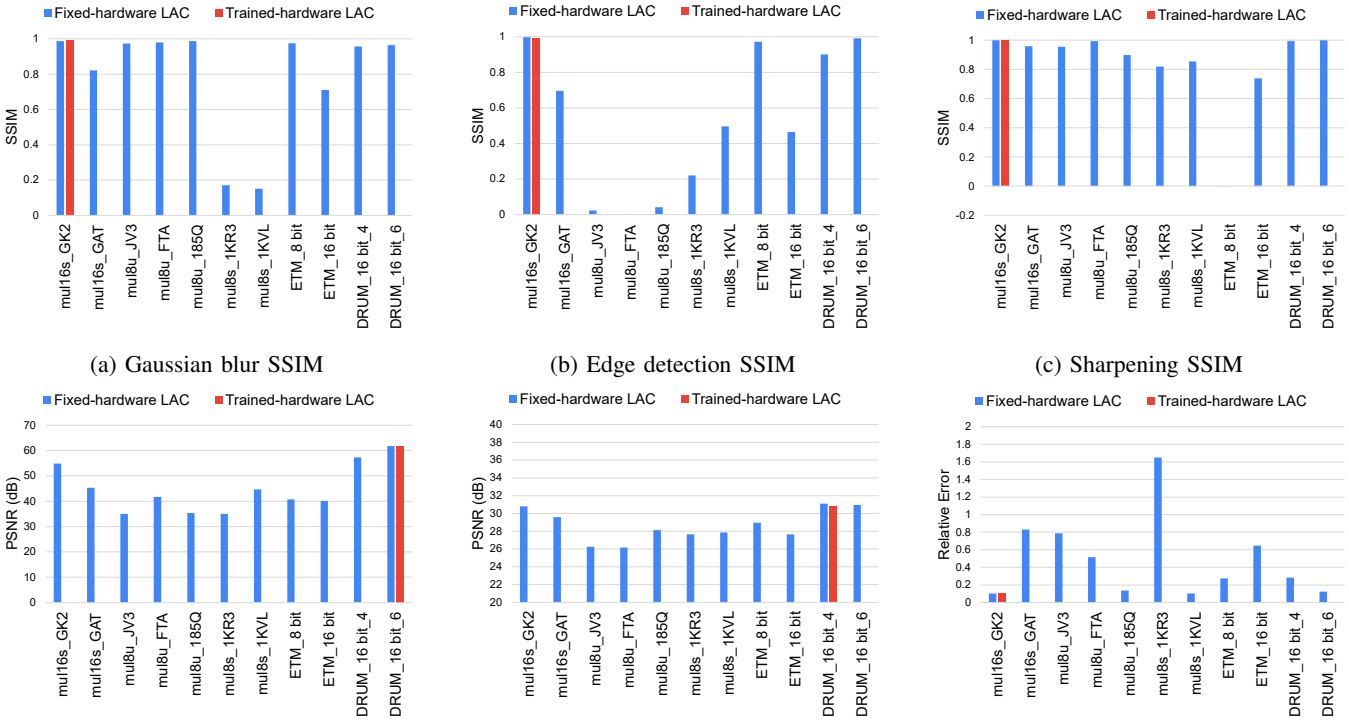
TABLE III: Delay of multipliers.

Multiplier	Variant	Delay
EvoApprox [9]	mul8u_JV3	0.58
	mul8u_FTA	0.95
	mul8u_185Q	1.41
	mul8s_1KR3	0.89
	mul8s_1KVL	1.33
	mul16s_GK2	2.95
	mul16s_GAT	2.57

Results of the accuracy-constrained search are presented in Fig. 10. An accurate multiplier was not included in the search so area values are not bounded at 1. For each of the target values of SSIM, we compare the area output of three search methods: picking a minimal satisfactory multiplier without any LAC optimization, NAS accuracy-constrained setup, and a brute-force search that optimizes all available multipliers prior to making a selection. Even though it incurs the smallest runtime cost, a search without LAC has a too scarce selection of multipliers with high accuracy. Utilizing LAC, both NAS and exhaustive search achieve the same area that is better than trivial no LAC approach. NAS offers runtime benefits that are further discussed in Section V-D.

C. Multi-hardware Search Results

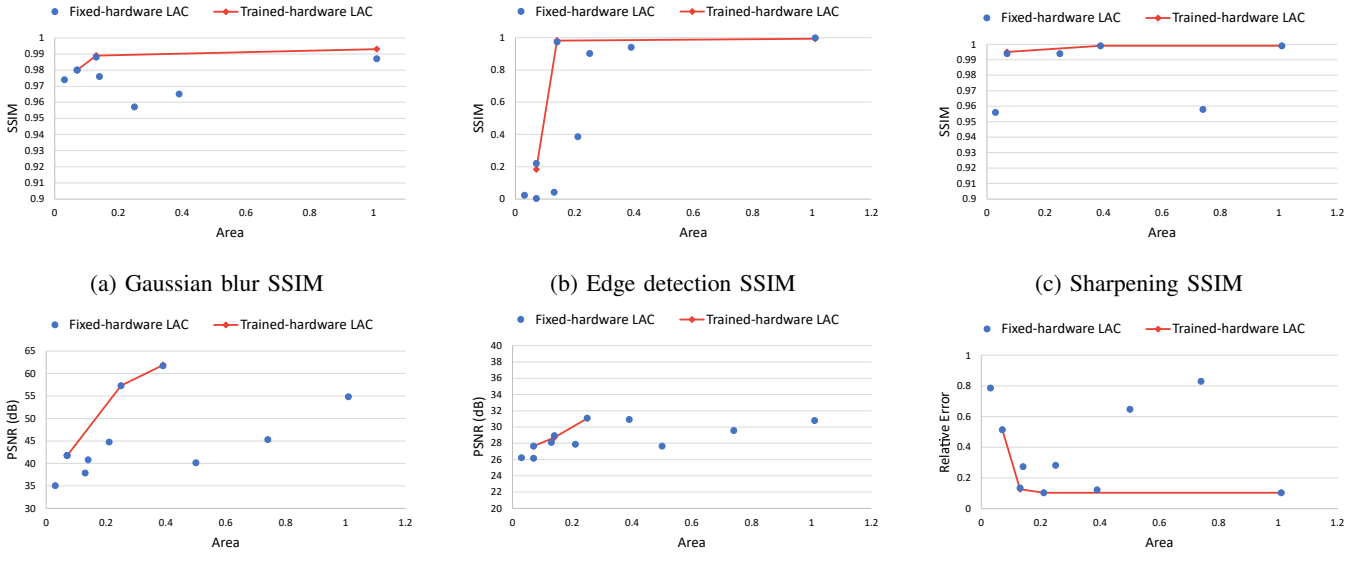
Our first multi-hardware search setup is Gaussian blur (parallel NAS). We used hyperparameter values $\gamma = 0.9$ and



(a) Gaussian blur SSIM (b) Edge detection SSIM (c) Sharpening SSIM

(d) DCT PSNR (e) DFT PSNR (f) Inversek2j RAE

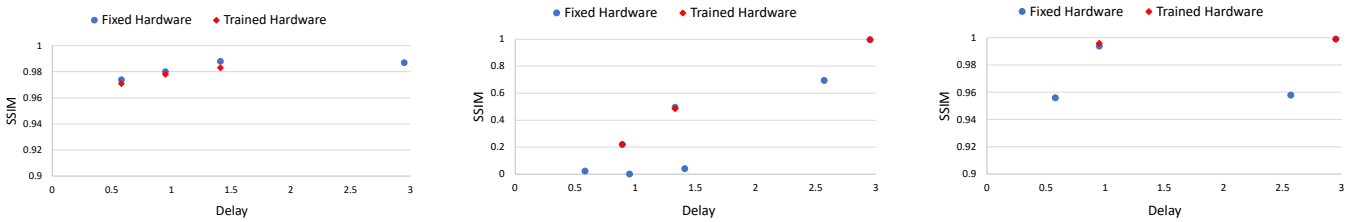
Fig. 7: LAC search results of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j.



(a) Gaussian blur SSIM (b) Edge detection SSIM (c) Sharpening SSIM

(d) DCT PSNR (e) DFT PSNR (f) Inversek2j RAE

Fig. 8: LAC performance-centric search results of (a) Gaussian blur, (b) edge detection, (c) image sharpening, (d) JPEG compression using DCT, (e) DFT, and (f) Inversek2j.



(a) Gaussian blur SSIM (b) Edge detection SSIM (c) Sharpening SSIM

Fig. 9: LAC delay constrained search results of (a) Gaussian blur, (b) edge detection, (c) image sharpening

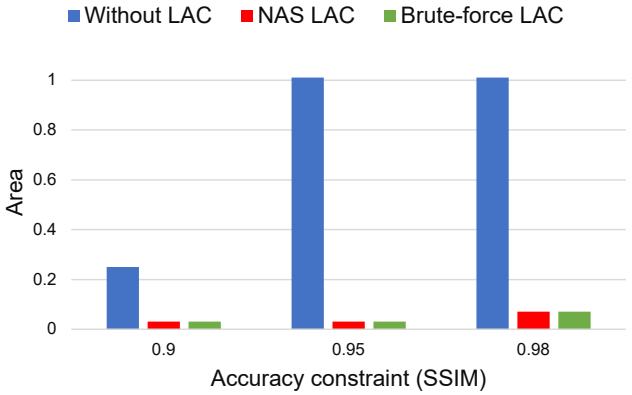


Fig. 10: LAC accuracy constrained search results of Gaussian blur

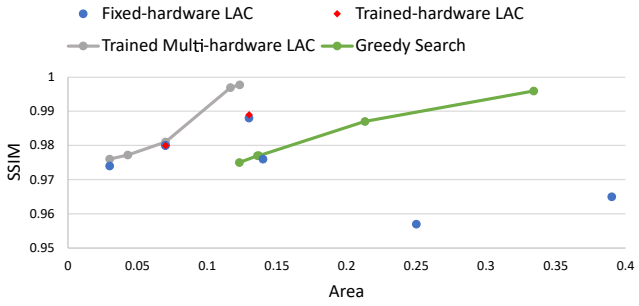


Fig. 11: Multi-hardware search results of Gaussian blur.

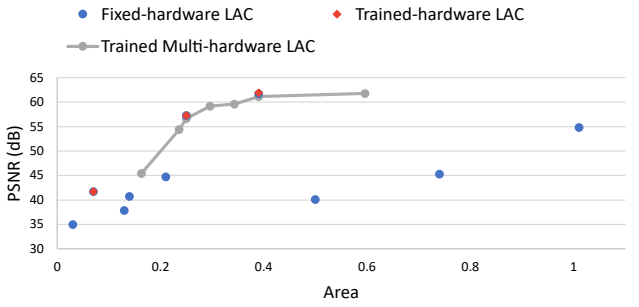


Fig. 12: Multi-hardware search results of JPEG compression using DCT.

$\delta = 1.0$. Fig. 11 shows the training results, where we take the average of multipliers as the overall area. For any given area in this range, allowing different multipliers in an application (denoted as "Trained Multi-hardware LAC") approach provides better output accuracy than the single multiplier approach. The maximum improvement in SSIM is 0.01 (from 0.988 to 0.998) achieved with a configuration with an average area of 0.123. The optimizer selected 8-bit FTA multiplier for the middle coefficient of the Gaussian blur kernel, namely $4/16$. This hardware produces more accurate results working with numerically larger inputs. Assigning it to the numerically highest value in computation increased the accuracy of the final result.

To provide a more comprehensive comparison, we implement a greedy stage-by-stage search. Starting with a random stage/part of an application, the search will select hardware for this stage by brute forcing all options, keeping the rest of the application fixed. The hardware assignment becomes permanent and an algorithm continues to the next random stage. Results are included in Fig. 11. The approach yields inferior to NAS quality, as an optimal choice for a single stage might become less performant once other multipliers are selected.

Results for our second setup, 3-stage JPEG (serial NAS), are shown in Fig. 12. In this setup, we take $\gamma = 1.0$ and $\delta = 300.0$. Combining approximate hardware allowed for area utilization, which is impossible in the single-multiplier setup. If the target accuracy is between two "Trained-hardware" points, the multi-hardware setup provides an area reduction. If both setups are compared for the same area, there is a small gap due to training noise. Gradual results were achieved by mixing above and below the constraint multipliers. Best performing configurations used well-performing DRUM "16-bit-6" and EvoApprox "mul16s_GK2" multipliers. Quality Pareto curve of the greedy stage-by-stage search for JPEG is not included as runtimes are impractically long as we discuss in Section V-D.

Both serial and parallel NAS implementations produce finer Pareto curves compared to trained hardware points. If the constraints are near these points, NAS will converge to the trained-hardware solution. New points become the optimal choice if other points are too far. For example, for the $58-59$ dB PSNR range on the JPEG application, the area is improved by 24% using the serial NAS training approach.

TABLE IV: Runtime comparison. Trained Multi-hardware Brute-force search and Greedy search values for JPEG and a Brute-force search value for Gaussian blur are estimated.

Setup	NAS	Brute-force search	Greedy search
Trained-hardware	147 sec	449 sec	449 sec
Trained Multi-hardware	462 sec	$> 10^{11}$ sec	7940 sec

(a) Gaussian blur

Setup	NAS	Brute-force search	Greedy search
Trained-hardware	11 min	57 min	57 min
Trained Multi-hardware	60 min	$>$ 6000 min	$>$ 1500 min

(b) JPEG

D. Runtime analysis

Alternatives to NAS, the brute-force and greedy approaches, require a longer runtime that is proportional to the number of hardware options. All tests are completed on the Intel i7-12700H CPU and Nvidia RTX 3070 GPU. Measurement results for Gaussian blur are presented in Table IVa. In the case of the trained hardware setup that used only a single approximate hardware, NAS always achieved same accuracy as brute-force search, while providing nearly 3 times runtime savings. Applied to a single layer, as in trained hardware, greedy search is the same as brute-force. For multi-hardware setup, the search space increases exponentially, making brute-force search infeasible. Greedy search is more practical but requires 17x running time compared to NAS. We also observed the runtime benefit of NAS for the JPEG application. As shown in Table IVb, using NAS decreased runtime by around 5 times in the trained hardware case. Multi-hardware brute-force and greedy search have impractical runtimes that are estimated to be at least 100 and 25 times the NAS runtime respectively.

VI. Conclusion

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 where the application kernels are optimized to improve application quality in the presence of approximate hardware. In a fixed-hardware setup, We have shown improvements of 0.24 in SSIM and 1.55dB 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. In a trained hardware setup, LAC allows searching for the best hardware configuration during training. With a method inspired by works on neural architecture search, LAC can find the best approximate hardware configuration even allowing for different hardware implementations for each part of the application. The multi-hardware approach enabled up to 38% area reduction compared to the single-hardware setup.

References

[1] V. Gupta, T. Li, and P. Gupta, "LAC: Learned Approximate Computing," in *IEEE/ACM Design, Automation and Test in Europe*, March 2022, p. 4.

[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] 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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

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

[16] 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.

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

[18] A. Yazdanbakhsh, D. Mahajan, H. Esmaeilzadeh, and P. Lotfi-Kamran, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design & Test*, vol. 34, no. 2, pp. 60–68, 2017.

[19] ——, "Axbench: A multiplatform benchmark suite for approximate computing," *IEEE Design and Test, special issue on Computing in the Dark Silicon Era*, 2016.

[20] 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.

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

[22] A. Krizhevsky, "Learning multiple layers of features from tiny images," pp. 32–33, 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

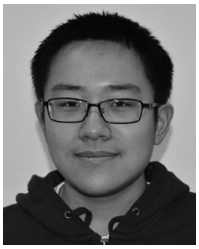
[23] Y. Bengio, "Estimating or propagating gradients through stochastic neurons," *CoRR*, vol. abs/1305.2982, 2013. [Online]. Available: http://arxiv.org/abs/1305.2982

[24] H. Cai, L. Zhu, and S. Han, "Proxylessnas: Direct neural architecture search on target task and hardware," *CoRR*, vol. abs/1812.00332, 2018. [Online]. Available: http://arxiv.org/abs/1812.00332

[25] W. Jiang, X. Zhang, E. H.-M. Sha, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Accuracy vs. efficiency: Achieving both through fpga-implementation aware neural architecture search," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3316781.3317757

[26] W. Jiang, L. Yang, E. H.-M. Sha, Q. Zhuge, S. Gu, S. Dasgupta, Y. Shi, and J. Hu, "Hardware/software co-exploration of neural architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4805–4815, 2020.

Egor Glukhov is pursuing the B.S. degree in Electrical Engineering at the University of California at Los Angeles. His research interests include computer architecture and hardware design.



Tianmu Li received the B.S. and Ph.D. degrees in electrical and computer engineering from the University of California at Los Angeles in 2017 and 2023 respectively. His current research focus is on efficient machine learning using approximate computing methods.

Vaibhav Gupta received his B.S. in Electrical Engineering from the University of California at Los Angeles.

Puneet Gupta (Fellow, IEEE) received the B.Tech. degree in electrical engineering from the Indian Institute of Technology Delhi, New Delhi, India, in 2000, and the Ph.D. degree from the University of California at San Diego, San Diego, CA, USA, in 2007. He is currently a Faculty Member with the ECE Department, University of California at Los Angeles, and has authored over 200 papers, 18 U.S. patents, a book, and a book chapter in the areas of system-technology co-optimization as well as variability/reliability aware architectures.