Runtime Efficiency-Accuracy Trade-off Using Configurable Floating Point Multiplier

Daniel Peroni, *Student Member, IEEE,* Mohsen Imani, *Student Member, IEEE,* and Tajana Rosing, *Fellow, IEEE*

Abstract—Many applications, such as machine learning and sensor data analysis, are statistical in nature and can tolerate some level of inaccuracy in their computation. Approximate computing is a viable method to save energy and increase performance by controllably trading off energy for accuracy. In this paper, we propose a tiered approximate floating point multiplier, called CFPU, which significantly reduces energy consumption and improves the performance of multiplication at a slight cost in accuracy. The floating point multiplication is approximated by replacing the costly mantissa multiplication step of the operation with lower energy alternatives. We process the data by using one of the three modes: a basic approximate mode, an intermediate approximate mode, or on the exact hardware, depending on the accuracy requirements. We evaluate the efficiency of the proposed CFPU on a wide range of applications including twelve general Open*CL* ones and three machine learning applications. Our results show that using the first CFPU approximation mode results in 3.5× energy-delay product (EDP) improvement, compared to a GPU using traditional FPUs, while ensuring less than 10% average relative error. Adding the second mode further increases the EDP improvement to 4.1×, compared to an unmodified FPU, for less than 10% error. In addition, our results show that the proposed CFPU can achieve 2.8× EDP improvement for multiply operations as compared to state-of-the-art approximate multipliers.

Index Terms—Approximate computing, Floating point unit, Energy efficiency, GPU

1 Introduction

In 2017, the number of smart devices around the world exceeded 20 billion. This number is expected to exceed 40 billions by 2022 [1], [2]. Many of these devices have batteries with strict energy constraints, so the need for systems that can efficiently handle the computing requirements of data-intensive workloads is undeniable [3]–[7]. Running machine learning algorithms or multimedia applications on general purpose processors, e.g. GPUs and CPUs, results in large energy consumption and performance inefficiency. Many applications do not need highly accurate computation, so accepting slight inaccuracy, instead of doing all computation precisely, results in significant energy and performance improvements [8]–[15]. Approximate computing seeks to alleviate this problem. While there are a number of proposed approximate solutions, they are limited to a small range of applications because they cannot control the error rate of their output.

Several data processing applications use a large range of values and require high precision. Therefore, computations in many traditional and state-of-the-art computing systems use floating point units (FPUs) [13], [16]–[18]. For example, on GPUs, frame rendering or high-performance scientific computations require many FPU operations and use a large amount of power. We observed over 85% of floating point arithmetic involved multiplication in the general OpenCL applications we tested. To cover the same dynamic range as with floating point, the fixed point unit must be five times larger and 40% slower than a corresponding floating point [19], [20].

Multiplication is one of the most common and costly FP operations, slowing down the computation in many applications such as signal processing, neural networks, and stream processing [21]–[25]. There are a number of approximate multiplication units designed to save power through different techniques. Several prior publications truncate the operands of the multiplication or use different sized blocks to enable approximate multiplication [26]–[28]. However, a lack of accuracy controls and large area overhead reduce the advantages provided by these approximate designs.

In this paper, we propose a configurable floating point multiplication, called CFPU, which significantly reduces the floating point multiplication energy consumption by trading off accuracy. CFPU avoids the costly multiplication when calculating the fractional part of a floating point number in one of two ways:

1) **Mantissa Discarding** - In floating point multiplies, the bottom 23 bits represent the mantissa. To calculate the result of a multiply operation, the mantissa from each operand are multiplied together, a step which consumes the majority of the total power and bottlenecks the operation. A substantially faster and lower energy approach is to discard one of the input mantissa and use the second one directly. Using one mantissa directly has the potential to generate high error rates for individual multiplies, so we provide two modifications that increase the final output accuracy.

- *Adaptive operand selection* finds and discards the mantissa which results in the lowest error. Minimizing individual operation error allows us to increase the number of calculations performed on the CFPU while maintaining the same output error.
- *Tuning* examines the first N bits of the discarded mantissa to predict error in the output result.

2) **Shift and Add** - When the mantissa discarding cannot produce results with the error below a user-specified requirement, we run the operation in a more accurate approximate mode. We examine

• *D. Peroni, M. Imani and T. Rosing are with the department of computer science and engineering, University of California San Diego, La Jolla, CA, 92093.*
E-mail: {dperoni, moimani, tajana}@ucsd.edu

the discarded mantissa to find the location of the first '1' bit. We shift the non-discarded mantissa based on this bit's position and add it to itself to create the new mantissa for the result value. This approach requires more energy and computation time compared to the first stage but is more accurate. An operation run in the second stage has at least 50% lower error than one run in the first stage for the same input operands. If neither of these approaches produces an output with an acceptable error, our design can control the level of output accuracy by identifying the inputs that result in the highest output error and assigning them to compute precisely on the CFPU.

We evaluate the efficiency of the proposed technique on AMD Southern Island GPU architecture by replacing the traditional FPUs with the proposed CFPU. We test OpenCL workloads and our results show that using first stage CFPU approximation results in 3.5× energy-delay product (EDP) improvement, compared to an unmodified GPU, while ensuring less than 10% average relative error. Adding the second stage further increases the EDP improvement to 4.1× for the same level of accuracy. Comparing the proposed CFPU with previous state-of-the-art multipliers [26], [28]–[30] shows that our design can achieve 2.8× higher energy-delay product with lower error.

We also examine the impact of CFPU on several machine learning algorithms. With these algorithms becoming increasingly popular, there is a strong need to improve their performance without sacrificing output error. They are often naturally stochastic, allowing them to accept an error in their output. We use our design to run three Rodinia [31] machine learning benchmarks: *K-Nearest Neighbor (KNN), Back Propagation* and *K-means*. *K-means* and *K-nearest neighbor* are used in data mining applications and involve dense linear algebra computation, while *Back Propagation* is used for training weights in neural networks. For machine learning algorithms CFPU design can achieve 2.4× energy saving and 2.0× speedup compared to an unmodified GPU while ensuring less than 1% average relative error. These benchmarks have 50% energy savings and 40% speedup when running on CFPU with two stages rather than the previously proposed one stage design [32].

The rest of paper is organized as the following. Section 2 reviews the related work. Section 3 describes the proposed approximate multiplications. The experimental results are presented in Section 4. Finally, Section 5 concludes the paper.

2 Related Work

There are several commonly examined approaches to approximate computing: voltage over scaling (VOS), use of approximate hardware blocks, and use of approximate memory units. VOS involves dynamically reducing the voltage supplied to a hardware component to save energy, but at the expense of accuracy. Error rates for VOS can be modeled to determine the trade-off between energy and accuracy for applications, allowing voltage to be lowered until an error threshold is reached [18], [33]–[37]. The circuit is sensitive to any variations, and if the operating voltage of a circuit is decreased too far, timing errors begin to appear which are too large to correct. A configurable associative memory was proposed by [9] which can relax computation by using VOS on the TCAM rows/bitlines to trade between energy and output accuracy. These techniques suffer because they are bound by GPU pipeline stages and therefore cannot improve computation performance. Because they make use of Hamming distance, a metric which does not consider bit position's impact on error, output accuracy is difficult to predict.

Another recently emerged strategy is the application of non-volatile memories (NVM) to create approximate memory units, for energy efficient storage and computing purposes [9], [38]–[40]. In computing, the goal of this approach is to store common inputs and their corresponding outputs. This style of associative memory can retrieve the closest output for given inputs in order to reduce power consumption [41], [42]. This approach does not work well in applications without a large number of redundant calculations. Associative memory can be integrated into FPUs to reduce these redundancies. Resistive CAMs have been used in order to accelerate application level computation as shown in [43]. This work uses the bit position insensitive metric of hamming distance resulting in poor accuracy in GPU usage.

Finally, approximate hardware involves redesigning basic component blocks to save energy, at the cost of accurate output [26], [29], [44], [45]. Liu *et al.* utilize approximate adders to create an energy efficient approximate multiplier [29]. Hashemi *et al.* designed a multiplier that selects a reduced number of bits used in the multiplication to conserve power [26]. Speculative designs are a recently explored route. Work in [46] proposed a speculative adder with error recognition to perform approximate operations. These types of adders can be utilized in approximate multipliers. Camus *et al.* propose a speculative approximate multiplier combines gate-level pruning and inexact speculative adders to lower power consumption and shrink FPU area [44]. Deep neural networks can tolerate approximate hardware, as shown in [47], which examines the use of variable fixed point in deep neural networks.

Compared to the previous work [26], [28], [30], we focus on optimizing floating point multiplication by eliminating mantissa multiplication. Our design computes common power of 2 multiplies exactly. Our configurable approximate floating point multiplier predicts accuracy based on the incoming inputs and runs high error operations on exact hardware rather than correcting results after computation. We extend our previous design [32] to offer two levels, instead of just one, of approximation to a tradeoff between approximation error and energy benefits. The first stage approximately multiplies two values by using the mantissa from one operand directly in the output. If the first stage error is too high, in the second stage the kept mantissa is shifted based on the position of the first '1' bit of the discarded mantissa and added to itself to produce an approximate result. If the second stage error is also too high, operations can be computed on exact hardware. For the same level of application accuracy, our modified multiplier reduces application energy by up to 45% compared with our work in [32] and shows a 2.8× EDP improvement for multiply operations when compared to state of the art approximate multipliers [26], [28], [30].

3 Approximate FPU Multiplier

Compared to integer computing units, FPUs are usually costly and energy-hungry components, due to the complex way floating point numbers are stored. Multiplication based components are inefficient and slow down many current applications including multimedia, streaming, neural networks, and other machine learning applications [8], [26]. Figure 2 shows the breakdown of ALU operations for the Sobel image filter application which has a high prevalence of multiply and multiply-add (muladd) operations

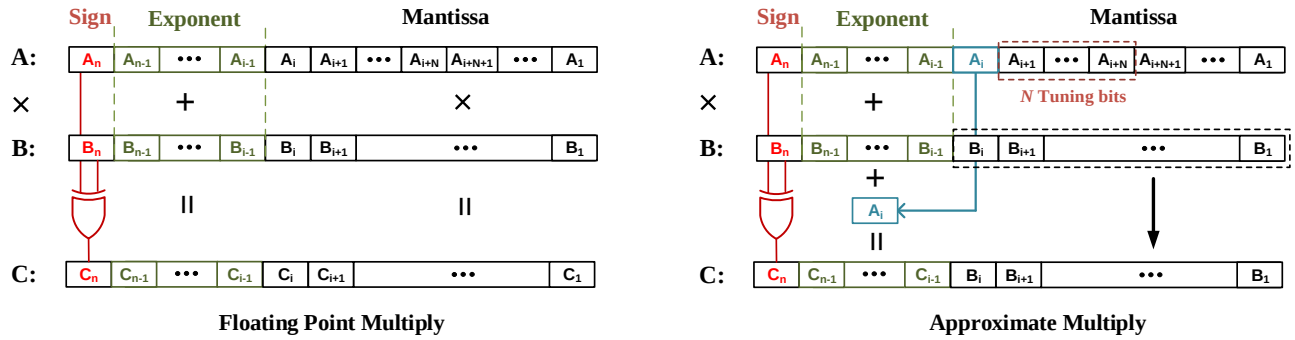


Fig. 1. Approximate multiplication of proposed CFPU between A ad B operands

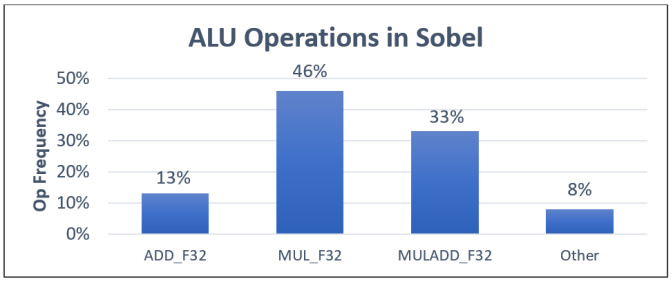


Fig. 2. ALU operation breakdown for the *Sobel* application

compared to additions. Horowitz et al [48] estimates in 45nm a floating point multiply consumes 3.7pJ of energy compared to a floating point add which consumes 0.9pJ. A floating point multiply consumes only 20% more energy than an integer multiply, so performing all operations as fixed point does provide significant energy savings. Based on these power estimates, the floating point multiply and muladd operations consume over 90% of the ALU energy for the Sobel application, making these operations good targets for energy optimization based on these values.

In order to make multiplication more efficient, we propose a two-stage floating point multiplier. The first stage optimizes mantissa multiplication by reusing one of the input mantissa directly in the output. The second stage seeks to reduce error further by shifting and adding the retained mantissa to itself.

3.1 IEEE 754 Floating Point Multiply

In floating point notation, a number consists of three parts: a sign bit, an exponent, and a fractional value. In *IEEE 754* floating point representation, the sign bit is the most significant bit, bits 31 to 24 hold the exponent value, and the remaining bits contain the fractional value, also known as the mantissa. The exponent bits represent a power of two ranging from -127 to 128. The mantissa bits store a value between 1 and 2, which is multiplied by 2^{exp} to give the decimal value.

FPU multiply follows the steps shown in Figure 1. First, the sign bit of $A \times B = C$ is calculated by XORing the sign bit of the A and B operands. Second, the effective value of the exponential terms are added together. Finally, the two mantissa values are multiplied to provide the result's mantissa. Because the mantissa ranges from 1 to 2, the output of the multiplication always falls between 1 and 4. If the output mantissa is greater than 2, it is normalized by dividing by 2 and increasing the exponent by 1.

3.2 CFPU Usage

CFPU is highly transparent to the application. A user running an error-tolerant application specifies the maximum error for any given multiply operation, $Error_{max}$, prior to execution. CFPU uses this value to ensure operations producing error greater than the value are sent to either a more accurate approximation mode or the exact hardware.

The flow chart for the proposed design is shown in Figure 3. Adaptive selection checks for a mantissa which, when discarded, produces an exact output when possible. The selector controls a multiplexer (MUX) between the two inputs A and B, and copies the selected mantissa to the output while discarding the other. Tuning utilizes the first N bits from the discarded mantissa to check against a threshold value and, if the threshold is not exceeded, the computation is complete. If the threshold is exceeded, the operation is run in the second stage which uses shift and add to increase accuracy. If the error of the second stage still exceeds the specified $Error_{max}$, the output is computed on the exact FPU hardware. We further explain the modifications in the following sections.

3.3 First stage Approximation

3.3.1 Mantissa Discarding

The multiplication of the mantissas is the most costly operation, taking over 80% of the total energy of the multiply operation [49], so the first stage approximation removes it entirely. Rather than multiplying the two mantissas, the unmodified mantissa from one of the input operands (e.g. input B) is used for the output value.

The error of any approximate multiply is $Mantissa_{discarded} - 1$. In the case where an operand is a power of 2, the mantissa is 1 and there is no error in the result.

$$Error = \sum_{i=0}^{n-1} 2^{-((n-i)A_{n-i-1})}. \tag{1a}$$

$$MaxError = \sum_{i=0}^{n-1} 2^{-(n-i)} = 0.999..9. \tag{1b}$$

Because the largest value a mantissa can be multiplied by is 2, the deviation from the kept mantissa and the correct answer is at most 100%. However, the maximum error can be reduced down to 50% by adding the first bit of the discarded mantissa to the sum of the exponent values. When the discarded mantissa is greater than 1.5 (the first mantissa bit is 1), the error is less than 50% if the kept mantissa is multiplied by 2 instead of 1. This is the functional equivalent of increasing the exponent by 1. By doing

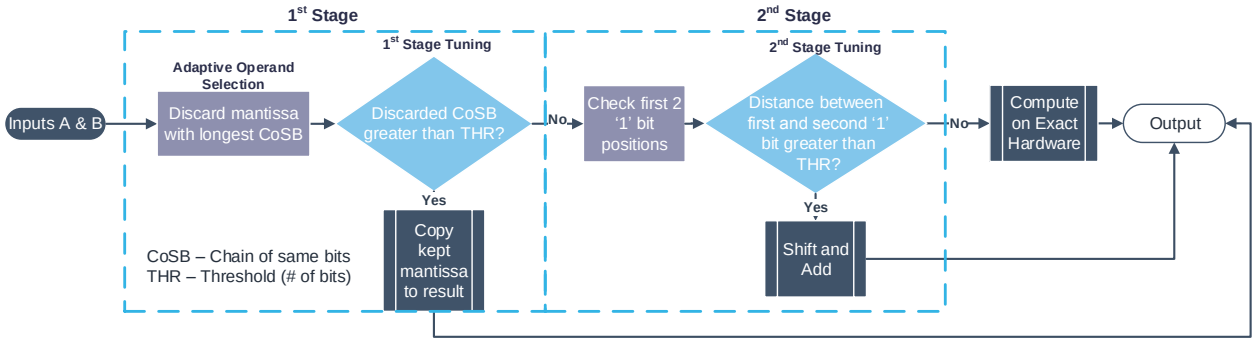


Fig. 3. CFPU operating flow including 1st and 2nd stage approximation

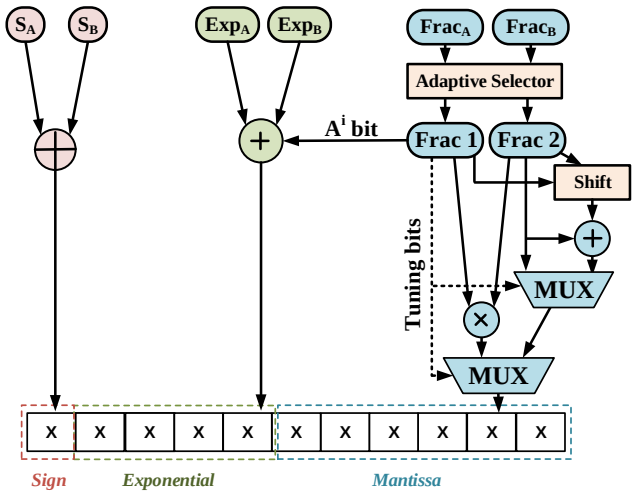


Fig. 4. CFPU Integration with adaptive selector and N tuning bits

this, the error range is shifted to be -50% to 50% instead of 0% to 100% as shown in Eq 2.

$$MaxError = abs(\sum_{i=0}^{n-1} 2^{-((n-i)A_{n-i-1})} - 0.5). \quad (2)$$

The additional logic needed to perform approximate floating point multiplication is shown in Figure 1. The B mantissa is used directly as the output mantissa, and the first bit of the discarded mantissa A is added with the two exponent values.

Figure 4 shows the flow for approximation. The sign bit for the result is computed by XORing the two input sign bits. The exponent for the result is computed by adding the two input exponents and the MSB of the discarded mantissa. CFPU ensures all operations compute their results below the user specified $Error_{max}$ through the use of adaptive selection and tuning. In the first level of approximation, Adaptive selection identifies the best mantissa to use in the output and discards the other. The upper N bits of the discarded mantissa are used to predict error and select between copying the other mantissa directly to the output, using the second level of approximation, or multiplying the two mantissas together.

3.3.2 Adaptive Operand Selection

Choosing which mantissa should be used for the result can significantly impact the accuracy of the CFPU output. For example, if the values 2.0 and 3.0 are multiplied, the result will be either 6.0 (exact) or 8.0 (33% error) depending on which mantissa is discarded.

In [32], we only use adaptive operand selection to identify mantissa values of zero. We improve adaptive operand selection to find the best mantissa to discard for all operations. This approach increases hit rate as more operations can be approximated, and reduces error for individual operations.

We compare the two mantissa values to determine the value which produces the lowest error when discarded. The output result uses the best mantissa. The benefits of this approach are twofold: 1) error for individual operations decreases and 2) the percentage of operations run in the first stage approximate mode increases because more operations are below the $Error_{max}$. The worst case error occurs when the discarded mantissa is closest to 1.5, so the preferred mantissa can be identified by detecting the longest continuous series of identical bits starting from the MSB. The mantissa with the longest series of either '1's or '0's gives a lower error when discarded. If all mantissa bits are '0', the error is 0. If both mantissas have an identical length series of bits, mantissa A is discarded.

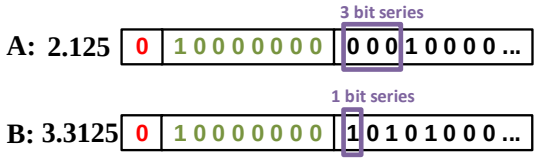


Fig. 5. Comparison of first mismatch position. Using adaptive selection mantissa A is discarded as it results in lower error. Exact answer: 7.039, Approx discarding A mantissa: 6.625, Approx result discarding B mantissa: 8.5

Figure 5 illustrates the distance calculation. If the mantissa from A is discarded, the resulting approximate multiply produces a value of 6.625 with an error of 5.9%. By comparison, if the mantissa from B is discarded instead, the output value is 8.5 with an error of 20.7%. Mantissa A has a series of 3 '0's compared to the single '1' of B, so discarding A results in a lower error.

3.3.3 Tuning Control

It is possible that neither mantissa produces output lower than $Error_{max}$ when discarded. CFPU automatically detects cases where the error exceeds the specified requirement and computes the result in the more accurate second stage. For example, if the

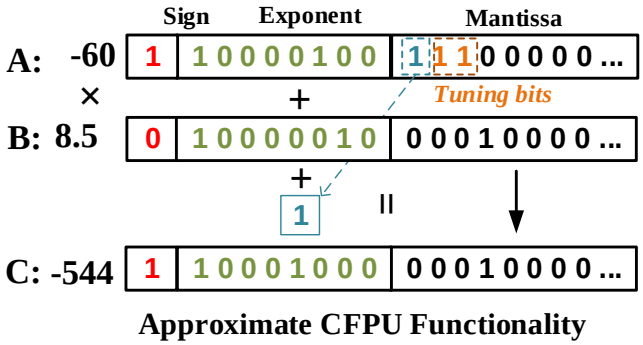


Fig. 6. An example of 32-bit multiplication running on proposed CFPU first level approximation using N=2 tuning bits.

maximum desired error is 5%, then the multiplication of A and B in Figure 5 cannot be computed using the first stage mantissa drop approximation.

The N bits after the MSB of the discarded mantissa are checked to tune the level of approximation. The case of maximum error occurs when the discarded mantissa is furthest from a power of 2, which occurs when its value is '1' followed by all '0's. When tuning, the goal is to ensure values over $Error_{max}$ are selected to run in the second stage, so the hardware must change its selection depending on the value of the first bit of the discarded mantissa. If the first bit of the discarded mantissa is '1', the first N tuning bits are checked for '0s', where N is selected based on $Error_{max}$.

$$N = log_2(\frac{1}{Error_{Max}}) - 1. \tag{3}$$

If a '0' is found, the hardware runs in exact mode. Similarly, when the first bit is '0', the first N tuning bits are checked for '1's instead. For each guaranteed bit in the A_{i-1} to 1^{st} indexes, the maximum error is reduced by half. Checking only one bit corresponds to a maximum error of 25%, two bits is 12.5%, etc.

$$MaxError = \left| \sum_{i=N}^{n-1} 2^{-((n-i)A_{n-i-1})} - 0.5 \right|. \tag{4}$$

An example of CFPU multiplication is shown in Figure 6 for two 32-bit floating point numbers in precise FPU and proposed CFPU with $Error_{max}$ set to 12.5%. An $Error_{max}$ of 12.5% requires CFPU to check N=2 tuning bits. The conventional FPU finds the correct solution of -510 by adding the exponents and multiplying the two mantissa, while XORing the sign bit to find three parts of the output data. Our design first compares the mantissas, which both contain a series of 3 identical bits, so operand A is selected for discarding. Next CFPU checks the first mantissa bit and the N tuning bits after that. In this case, the first mantissa bit is '1', so the next two bits are checked for '0' to determine if the value is below the desired error rate. When two tuning bits are checked, the maximum error is 12.5%. In this example, both tuning bits are '1', so the calculation continues in approximate mode and the mantissa from the value 8.5 is copied to the output value. The resulting output is -544, which deviates 6.67% from the correct value of -510 and falls below the desired $Error_{max}$.

3.4 Second stage Shift and Add

If the first stage produces a result which has the error greater than $Error_{max}$, CFPU activates the second stage instead. It has greater accuracy than the first stage but higher energy cost. The energy draw of the second stage is less than that of the exact hardware.

Some applications when running with only a single level of approximation, they require a high percentage of multiplies to be run in exact mode. FFT, MersenneTwister, and DwtHaar1D require almost 50% of their operations to be run on the CFPU exact mode to maintain output error below 10% as shown in Table 2. Running this high a percentage of operation in exact mode limits potential energy savings, so the addition of the second level of approximation to CFPU allows more of the multiply operations to be approximated and while maintaining acceptable output error.

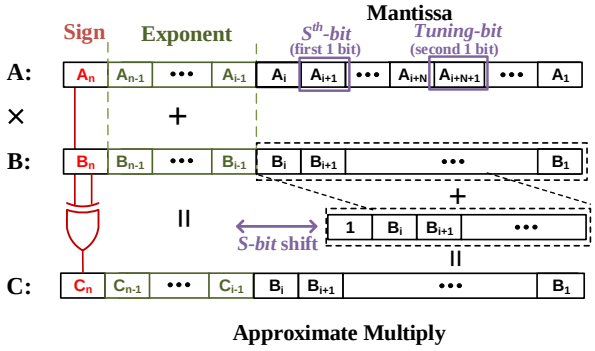


Fig. 7. Second stage shift and add in CFPU

Our 2nd stage approximation shifts and adds the kept mantissa with itself to produce a value closer to the exact output. Our hardware detects the positions of the first two '1' bits in the discarded mantissa. Figure 4 shows the overall flow of CFPU. The first level uses the fractional bits directly, Frac 2, while the second level uses the shifted value of Frac 2 summed with the original value of Frac 2. Frac 1 provides tuning bits to predict if the error for the first level is too large. If it is, then the second level error is predicted. If this predicted error is also too great, the two mantissa values are multiplied together to produce an exact result.

Figure 7 demonstrates the shift and add design. The first '1' bit position, P1, is detected by hardware and determines S, the shift amount, where S is the number of mantissa bits minus the bit position. The mantissa used in the output is shifted S positions right and added its unshifted value. The second '1' bit is used for tuning and helps determine the maximum error between the calculated result and the approximate result. The maximum error decreases the further the first '1' bit is from the MSB because of the shift and added mantissa decreases relative to the original mantissa. The closer the second '1' bit, P2, is relative to the first, the higher the error in the result. If the two '1' bits are adjacent, the shift and add value will be up to 50% smaller than necessary to reach an exact result.

$$MaxError = \sum_{i=0}^{P1} 2^{-(B_{P1-i-1})}. \tag{5}$$

The maximum error for the second level is based on the P1 within the discarded mantissa. The further P1 is from the MSB, the lower the maximum error will be.

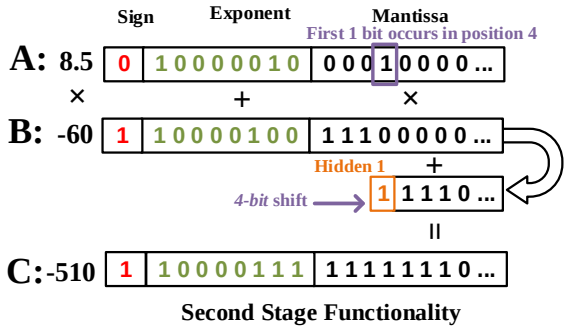


Fig. 8. Example of shift and add. Produces exact result, while first stage does not.

Figure 8 provides an example of the computation which the first stage design computed with 6.67% error as shown in Figure 6. In this example, P1 is in the fourth MSB, so the shift value is 4. Each mantissa contains a hidden '1' bit left of the MSB which must be accounted for and shifted in. The mantissa from B is shifted left by 4 bits to create the value '0001111...' and added the unshifted value '1110000...' to produce the approximate output. In this example, the second stage design calculates the output exactly. The error decreases from 6.67% calculated by the first stage to 0% from the second stage while consuming less energy than the exact multiply operation.

In cases where there is only one '1' bit in the mantissa, the shift and add approach produces exact results. In this case, the discarded mantissa is effectively a power of 2 and as such resolves to an integer value by which to shift the kept mantissa.

3.4.1 Running on Exact Hardware

The output error produced by the second stage approximation can still exceed $Error_{max}$, so some of the computations must be run on the exact hardware. Similar to the mantissa discarding approach, we use a threshold value to determine which values are run on the exact hardware. We search the discarded mantissa for the second occurrence of a '1' bit, $P2$. The position of $P2$ relative to the first bit, and $P1$ relative to the MSB determines the threshold value, V. The threshold is calculated as $V = S + (P1 - P2)$, where S is the shift amount. The minimum threshold occurs when $P1 = 22$ and $P2 = 21$, the upper MSBs, giving $S = 1$ and $V = 2$. $V = 2$ corresponds to a maximum output error of 25% on an individual operation. As V increases, the maximum output error becomes $50\% \times 2^{-V}$.

$$V > N = log_2(\frac{1}{Error_{Max}} - 1). \quad (6)$$

V must be greater than N, the number of tuning bits, in order to guarantee the result with produce an error below $Error_{max}$ If the output error still exceeds $Error_{max}$, the operation runs on exact FPU hardware instead.

The shift and add approach is more complex and power intensive when compared to the basic mantissa discard approach. The additional logical overhead and extra power draw make it viable as an intermediate option to reduce output error without requiring the full power consumption of the exact hardware. Shift and add is more accurate than mantissa drop, but if a user requires even greater output accuracy, operands producing highly erroneous results are sent to the exact hardware. The user can configure output accuracy by adjusting $Error_{max}$ value for both stages.

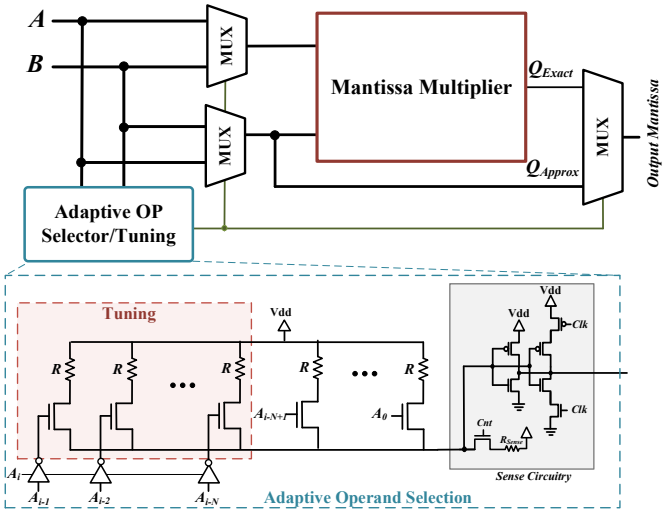


Fig. 9. Circuitry to support adaptive operand selector and tuning the level of approximation in CFPU.

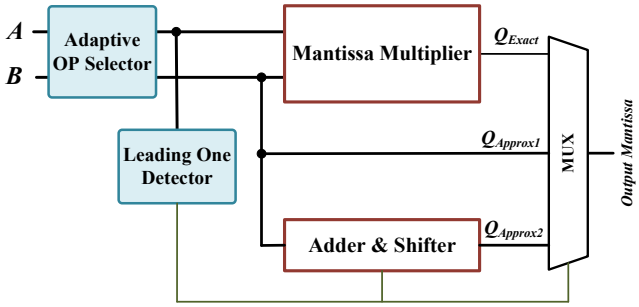


Fig. 10. Circuitry to support CFPU with two level approximation.

4 CFPU Hardware Support

Figure 9 shows the circuitry to enable CFPU adaptive operand selection and accurate tuning. We implement adaptive operand selection by checking the mantissa bits in both input operands. Our design compares the mantissas to determine which produces the lowest output error. If one mantissa has bits which are all zero, the second mantissa is copied to the output to produce an exact result. To ensure the mantissa which produces the greatest error is discarded, the hardware must locate and discard the mantissa furthest from 1.5. The further the discarded mantissa is from 1.5, the lower the output error. The circuit examines the two input mantissas and detects the one with the longest chain of continuous 1s or 0s starting from the first bit. A chain of zeros represents a mantissa close to 1, and similarly, a chain of ones is closer to a mantissa of 2. The mantissa with the longest consecutive chain of either zeros or ones is copied to the output and the other discarded.

As Figure 9 shows, the detector circuitry is a simple transistor-resistor circuitry which samples the match-line (ML) voltage to detect the $A_{i-1}, A_{i-2}, ..., A_0$ input operand. In case of any '1' bit in a mantissa, the sense amplifier detects changes in the ML voltage (ML=1). However, if all mantissa bits are zero, no current passes through R_{sense} and the B operand mantissa is selected as the output mantissa. To detect the '1' bit on A^{th}_{i-1}, ..., A^{th}_0 indices on CFPU, the sense amplifier Clk needs to be set to 250ps. Based on the

results, we can dynamically change the sampling time to balance the ratio of the running input workload on the approximate CFPU core. The operand selection happens by using two multiplexers which are controlled with our detector hardware signal. Similarly, to tune the level of approximation, our design uses N bits (after the first mantissa bit) of the selected mantissa to decide when to perform mantissa multiplication or approximate it. The number of tuning bits sets the level of approximation, with each additional bit reducing the maximum error by half. The goal is to check the value of the $A_{i-1},...,A_{i-N}$ to make sure they are same as the A_i. For this purpose, the circuitry selects the original value or inverted values of the tuning bits for the circuitry to search. To make the design area efficient, we use the same circuitry for adaptive operand selection and tuning approximation. For each application, sampling time can be individually set in order to provide target accuracy.

CFPU with two level approximation requires similar hardware to perform adaptive operand section. The two leading '1' bits in selected input operand are detected and their position used to calculate the maximum error. If the estimated error is less than $Error_{max}$, shift and add is used, otherwise the result is computed on exact hardware. Figure 10 shows the proposed hardware supporting two level approximating. The adaptive operand selector identifies the best mantissa for use in the result and the tuning bits of the discarded mantissa are examined. If the error is low, the best mantissa is copied directly to the output. Otherwise, a leading '1' but detector identifies the bit positions of the discarded mantissa to predict the 2nd stage error. If the 2nd stage error meets the accuracy requirement, a shift block and an adder block compute the result mantissa. If the error of the 2nd stage is too great, the standard mantissa multiplier hardware computes the result.

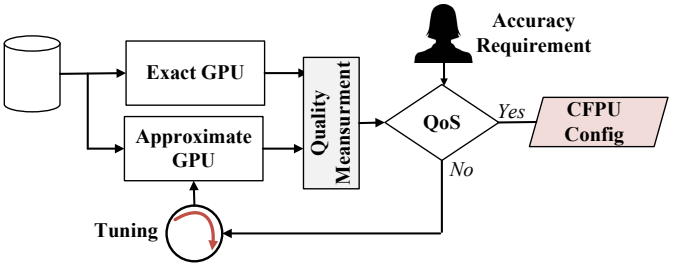


Fig. 11. Framework to support tunable CFPU approximation.

5 Results

5.1 Experimental Setup

We integrated the proposed approximate CFPU in the floating point units of an AMD Southern Island Radeon HD 7970 GPU. We modified Multi2sim, a cycle accurate CPU-GPU simulator [50], to model the CFPU functionality in three main floating point operations in GPU architecture: multiplier, multiplier-accumulator (MAC) and multiply-add (MAD). We evaluated energy of traditional FPUs using Synopsys Design Compiler and optimized for power using Synopsys Prime Time for 1 ns delay in 45-nm ASIC flow [51]. The circuit level simulation of the CFPU design has been performed using HSPICE simulator in 45-nm TSMC technology. We first test the efficiency of enhanced GPU on twelve general Open*CL* applications from AMD OpenCL SDK [52]: *Sobel, Robert, Mean, Laplacian, Sharpen, Prewit, QuasiRandom, FFT, Mersenne, DwHaar1D, Blur* and *Blackscholes*. In these applications, roughly 85% of the floating point operations involve multiplication.

Our design is then tested using machine learning applications. Machine learning algorithms are often error-tolerant allowing them to be run more efficiently on approximate hardware. We examine three OpenCL benchmarks from the Rodinia 3.1 machine learning suite [31]. These benchmarks are *K-Nearest Neighbor(KNN), Back Propagation* and *K-means*.

- K-means - Highly parallelizable clustering algorithm used in many data mining applications.
- K-nearest neighbor - Calculates the K nearest neighbors from given data. Calculates euclidean distance from many data points in parallel.
- Backpropagation - Used to train the weights in a neural network. Error values are propagated through the network to retrain nodes and reduce output accuracy.

We propose an automated framework to fine-tune the level of approximation and satisfy required accuracy while providing the maximum energy savings. Figure 11 shows the proposed framework, consisting of the accuracy tuning and accuracy measurement blocks. The framework starts by putting the CFPU in the maximum level of approximation when no tuning bits are checked. Then, based on the user accuracy requirement, it dynamically decreases the level of approximation 1 tuning bit at a time until computation accuracy satisfies the user quality of service. The tuning is adjusted using a custom assembly instruction to set the approximation level of the CFPU. For each application, this framework returns the optimal number of CFPU tuning bits checked, providing maximum energy and performance efficiency. In future runs, the detected optimal configuration is set using the custom assembly instruction prior to running approximable code and disabled using the same instruction after completion of the code.

5.2 First stage CFPU

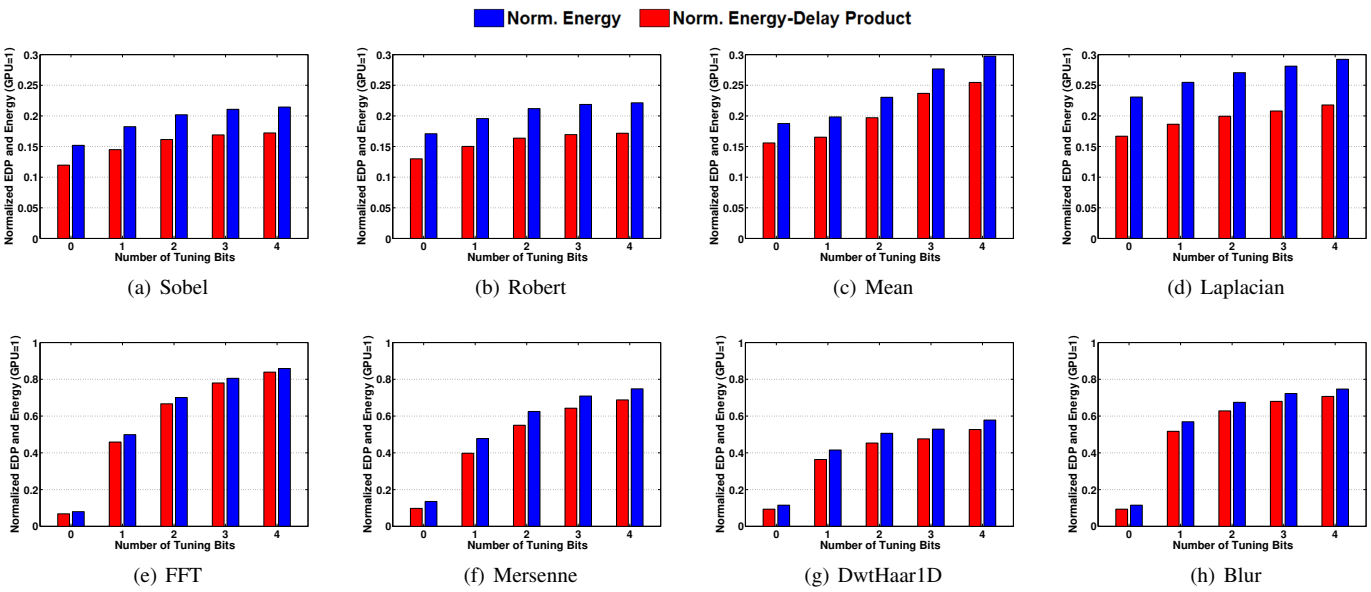
We first look at approximate multiplication. The proposed modified FPU can run entirely in approximate mode while providing a level of accuracy that is still acceptable for many applications. Table 1 shows the computation accuracy, energy savings, and speedup of running eight general OpenCL applications on the approximate GPU. These applications achieve error below 10% while using only first stage approximation. The energy and performance of proposed hardware are normalized to the energy and performance of a GPU using conventional floating-point units. Our experimental evaluation shows that our approximate hardware can achieve to 72% energy savings, 19% speedup, and 5.4× energy-delay product for these applications compared to the traditional AMD GPU, while providing an acceptable output quality less than 10% average relative error.

5.2.1 Adaptive Operand Selection

The approximate multiply uses both exponents in its calculation, but discards one of the mantissas, making an operation effectively a multiply by a power of 2. Therefore, a multiplication by a power of 2 always results in an exact answer on our hardware. It is possible to reduce error by ensuring the value of the discarded mantissa is equal to 1. This occurs when all the mantissa bits are 0. In the 11 OpenCL applications we tested an average of 52% of multiplies involved at least one power of 2. Hardware intelligently

TABLE 1
Energy and performance improvement and average relative error replacing GPU with proposed floating point multiplications.

Applications	Sobel	Robert	Mean	Laplacian	Sharpen	Prewit	QuasiR	Blackscholes	Average Improv.
Energy savings	84%	83%	81%	76%	75%	69%	72%	53%	**72%**
Speed up	21%	24%	27%	16%	20%	22%	12%	10%	**19%**
EDP improvement	8.3×	7.7×	6.4×	4.7×	5.3×	4.2×	4.1×	2.7×	**5.4**×
Error (%)	9.02%	1.19%	1.36%	1.96%	0%	0%	0%	6.79%	**2.54%**



(a) Sobel (b) Robert (c) Mean (d) Laplacian

(e) FFT (f) Mersenne (g) DwtHaar1D (h) Blur

Fig. 12. Normalized energy consumption of enhanced GPU with a tunable single stage checking N tuning bits for application run.

TABLE 2
Ratio of approximate to total CFPU operations and average relative error running applications in CFPU with single level approximation.

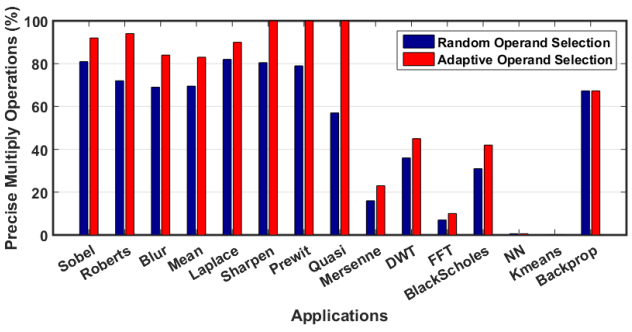
Tuning bits	Sobel		Robert		Mean		Laplacian		FFT		Mersenne		DwtHaar1D		Blur	
	Approx/total	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*
0 bit	100%	9.02%	100%	1.08%	100%	1.36%	100%	1.96%	100%	73%	100%	38%	100%	94%	100%	11.1%
1 bit	96%	2.70%	97%	0.35%	98%	1.04%	96%	0.50%	54%	9.8%	60%	13%	66%	31%	82%	3.7%
2 bits	94%	0.74%	95%	0.10%	85%	0.03%	94%	0.11%	32%	8.3%	43%	8%	55%	12%	76%	0.92%
3 bits	93%	0.07%	94%	0.03%	85%	0.01%	93%	0.02%	21%	4.1%	33%	5.2%	53%	8.3%	62%	0.36%
4 bits	92%	0.01%	94%	¯0%	84%	¯0%	92%	¯0%	15%	2.3%	29%	3.2%	47%	0.7%	53%	0.21%
Exact	92%	0%	93%	0%	84%	0%	92%	0%	10%	0%	23%	0%	45%	0%	53%	0%

TABLE 3
Ratio of approximate to total CFPU operations and average relative error running applications on CFPU with two level approximation.

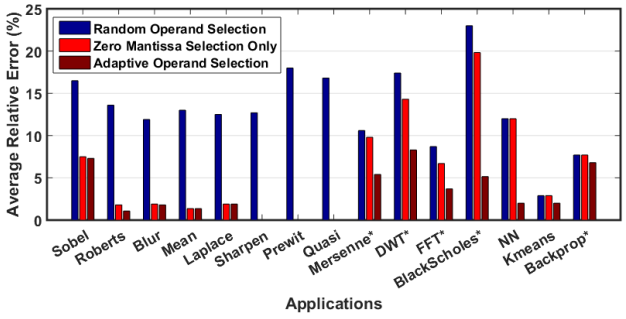
Tuning bits	Sobel		Robert		Mean		Laplacian		FFT		Mersenne		DwtHaar1D		Blur	
	Approx/total	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*	*Approx/Total*	*Error*
0 bit	100%	6.02%	100%	0.9%	100%	1.1%	100%	1.2%	100%	27.4%	100%	26%	100%	35.3%	100%	7.4%
1 bit	98%	2.3%	99%	0.12%	99%	0.3%	96%	0.50%	72%	7.3%	78%	5.4%	81%	18%	91%	3.5%
2 bits	96%	0.34%	97%	0.03%	93%	0.01%	97%	0.06%	64%	3.5%	66%	4.7%	77%	8.4%	79%	1.3%
3 bits	96%	0.34%	96%	0%	95%	¯0%	96%	¯0%	50%	3.2%	54%	2.6%	75%	7.9%	71%	0.59%
4 bits	95%	0%	96%	0%	95%	¯0%	96%	¯0%	41%	2.8%	47%	1.8%	59%	2.5%	66%	0.71%
Exact	95%	0%	96%	0%	93%	0%	95%	0%	28%	0%	38%	0%	45%	0%	60%	0%

checking both inputs and adaptively discarding mantissas results in more exact computations and greatly reduced overall output error.

Figure 13a compares the portion of multiplications which runs precisely on the proposed CFPU with and without adaptive operand selection for the evaluated applications. Figure 13b shows the impact of the adaptive operand selection on the computation accuracy of the proposed CFPU. In random operand selection, the mantissa of the first input is always selected for the output without comparing the potential error of each mantissa. The result shows that adaptive operand selection significantly improves computation accuracy such that for all shown applications, the average relative error decreases to less than 7%. This improvement is due to increasing the portion of multiplications which are run precisely on the CFPU. We verify this by looking at the percentage of precise CFPU operations using the adaptive selection technique. For example, in *Sobel* application, 82% of the outputs are calculated exactly, with an overall relative error of 16% when using random operand selection, while adaptive selection shows 92% of the outputs calculated exactly, at an overall relative error of 9%. All operations in Sharpen, *Prewit*, and *QuasiRandom* contain a mantissa of zero, so CFPU computes all results exactly. The image



(a)



(b)

Fig. 13. a) The portion of precise CFPU computation in different applications and b) the impact of adaptive operand selection on the computation accuracy. (Applications run partially on exact hardware).

processing applications *Sobel*, *Roberts*, *Blur*, *Mean*, and *Laplace* also involve many operations that can be computed exactly with CFPU. Using adaptive operand selection to only select and discard zero mantissas improves application accuracy by up to 8×.

Our work in [32] only utilizes a zero mantissa selection policy, which does not reduce error significantly for many applications. the computation accuracy by up to 13× (8×) compared to random operand selection (zero mantissa only).

The application showing the best accuracy improvement, Roberts, decreases from 13.6% application error using random operand selection to 1.85% using zero mantissa only selection [32]. The improved adaptive operand selection further decreases the output error to 1.08%. Excluding the three applications that only contain zero mantissa operations and get the best possible results, our evaluation for 12 different applications shows adaptive selection reduces average error by 2.2× more than our previous work [32].

5.2.2 Tuning

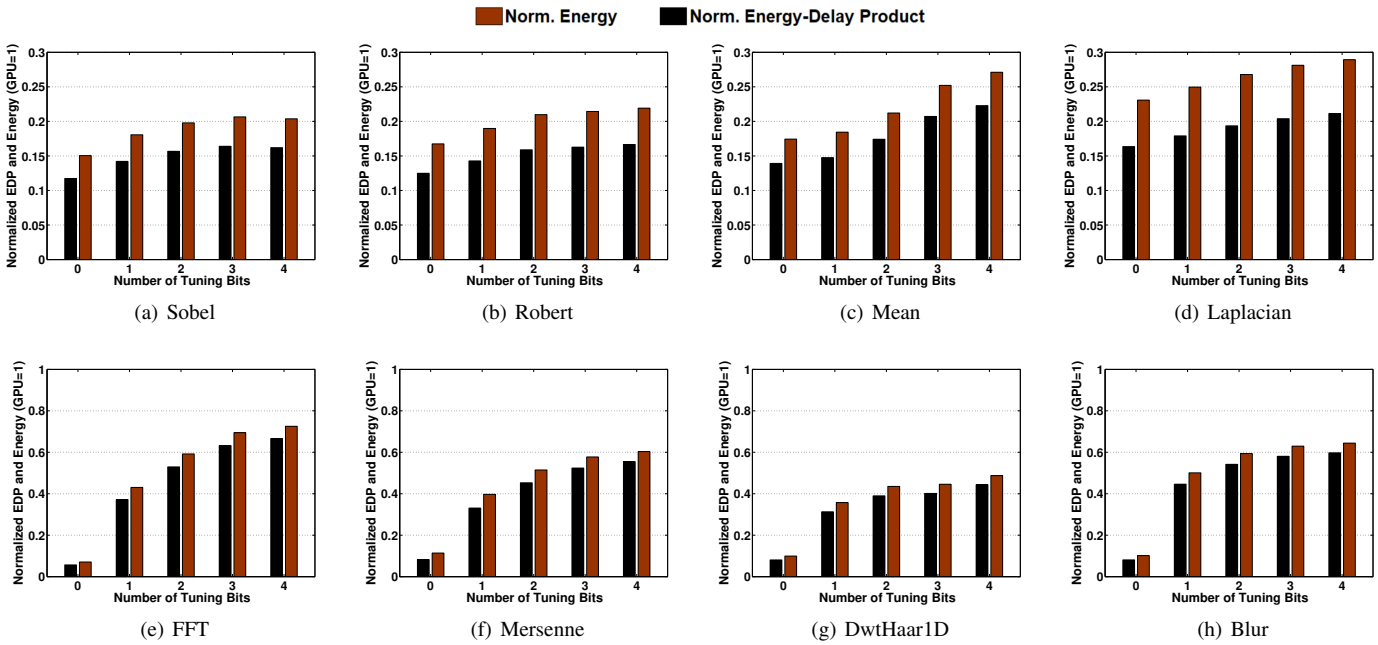
We show the efficiency of the proposed CFPU by running different multimedia and general streaming applications on the enhanced GPU architecture. We consider 10% average relative error as an acceptable accuracy metric for all applications, verified by [53]. We tune the level of approximation by checking the *N* bits of mantissa in one of the input operands. If all *N* tuning bits match with the first mantissa bit, the multiplication runs in approximate mode, otherwise, it runs precisely by multiplying the mantissa of input operands. For each application, Table 2 shows the average relative error and portion of running multiplications in each application on exact and on approximate CFPU, when the number of tuning bit changes from 0 to 4 bits. Increasing the number of tuning bits improves the computation accuracy by processing the far and inaccurate multiplications in precise CFPU mode. Increasing the number of tuning bits slows down the computation because a larger portion of data is processed on precise CFPU. Figure 12 shows the energy consumption and energy-delay product of a GPU enhanced with tunable CFPU using different numbers of tuning bits. Our experimental evaluation shows that running applications on proposed CFPU provides respectively 3.5× and 2.7× energy-delay product improvement compared to a GPU using traditional FPUs, while ensuring less than 10% (1%) average relative error.

5.3 Second stage CFPU

Although proposed first stage approximate multiplication provides high energy savings, the accuracy of computation depends on the application. For some applications, with quantized inputs, e.g., *Sharpen filter*, the proposed design can work precisely with no average relative error. Other applications, such as recognition algorithms like motion tracking and detection applications, quantify changes in the input data allowing them to tolerate small amounts of error. An approximate multiplier must be able to control the level of output error to ensure close to exact results are calculated for these applications. Figure 16 shows the distribution of error rates for each multiply operation of two applications. In the case of *Sobel*, almost 90% of the multiplies are by a power of 2 and are handled exactly by our approximate solution. The remaining 10% of operations have incorrect values with error rates ranging up to 50%. The Mersenne Twister application, on the other hand, has a evener distribution of error rates. While about 12% of the computations have no error, the error rates are too randomly distributed to provide acceptable overall error without additional optimization. For this application, the first stage approximation does not provide sufficient accuracy on its own, so over 50% of operations must be run on exact hardware to keep error below 1%.

Table 3 lists the hit rate of approximate hardware and average relative error for different applications running on hardware using two levels of approximation. The result shows that CFPU using two-level approximation can provide significantly higher accuracy compared to single level approximation. This efficiency comes from the ability of CFPU to assign input data to an approximation hardware which better classifies input data. Figure 5.2.1 shows the energy consumption and energy-delay product of CFPU using a two-level approximation. The result shows that accepting 10% (1%) average relative error, CFPU can provide 4.1× (3.2×) energy-delay product improvement as compared to a GPU using traditional FPUs.To ensure the quality of computation, Figure 15 compares the visual results of *Blur* running on precise and approximate hardware. Our result shows that approximate computing creates no noticeable difference between the precise and approximate result images.

Table 4 and Table 5 list the energy efficiency improvement, speedup, and average relative error of running Rodinia applications on CFPU with one and two levels of approximation. The results are listed when the number of tuning bits changes from 0-bits to 4-bits. For machine learning algorithms, CFPU with a single stage achieves 1.6× energy savings and 1.4× speedup while ensuring less than 1% average relative error. Enabling 2nd stage approximation increases energy savings to 2.4× and speedup to 2.0×, 50% and 40% improvements respectively. Figure 18 shows the energy and accuracy improvements for each optimization to the CFPU. In the first case, mantissa discarding is used for every



(a) Sobel (b) Robert (c) Mean (d) Laplacian

(e) FFT (f) Mersenne (g) DwtHaar1D (h) Blur

Fig. 14. Normalized energy and energy-delay product of enhanced GPU with tunable two stage CFPU.

TABLE 4
Impact of one level CFPU approximation on accelerating Rodinia applications.

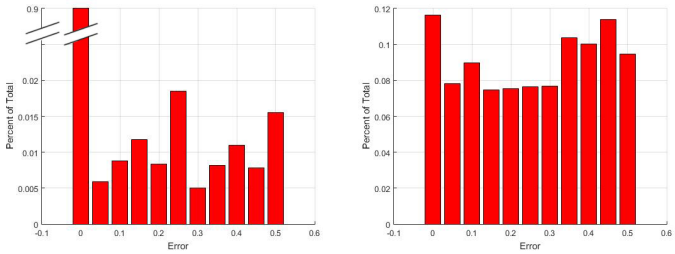
Tuning bits	Backpro				K-nearest neighbor				*K-means*			
	Approx/ Exact	**Error**	**Energy Improv.**	**Speedup**	**Approx/ Exact**	**Error**	**Energy Improv.**	**Speedup**	**Approx/ Exact**	**Error**	**Energy Improv.**	Speedup
0-bit	100	25.3%	3.25	2.25	100.00	2.5%	2.91	2.21	100.00	2.1%	2.70	1.80
1-bit	80.3	22.7%	2.65	1.84	55.0	2.1%	1.59	1.43	51.6	0.9%	1.68	1.31
2-bits	73.6	22.5%	2.32	1.70	32.0	0.6%	1.32	1.23	26.1	0.3%	1.19	1.14
3-bits	71.6	16.9%	2.16	1.67	17.4	0.3%	1.15	1.11	13.8	0.1%	1.33	1.21
Exact	67.3	0%	1.87	1.60	0.6	0%	1.09	1.06	0.01	0%	1.00	1.00

TABLE 5
Impact of two level CFPU approximation on accelerating Rodinia applications.

Tuning bits	Backpro				K-nearest neighbor				*K-means*			
	Approx/ Exact	**Error**	**Energy Improv.**	**Speedup**	**Approx/ Exact**	**Error**	**Energy Improv.**	**Speedup**	**Approx/ Exact**	**Error**	**Energy Improv.**	Speedup
0-bit	100	25.3%	3.25	2.25	100.00	2.5%	2.91	2.21	100.00	2.1%	2.70	1.80
1-bit	98.7	7.89%	3.16	2.22	87.8	1.0%	2.36	1.93	90.2	0.7%	2.31	1.67
2-bits	93.5	3.35%	2.84	2.08	59.7	0.6%	1.65	1.49	61.9	0.1%	1.64	1.38
3-bits	87.7	0.1%	2.54	1.95	40.2	0.2%	1.35	1.28	40.0	0.1%	1.33	1.21
Exact	67.3	0%	1.87	1.60	3.8	0%	1.03	1.02	0.01	0%	1.00	1.00

(a) Exact (b) Approx (c) Tuned

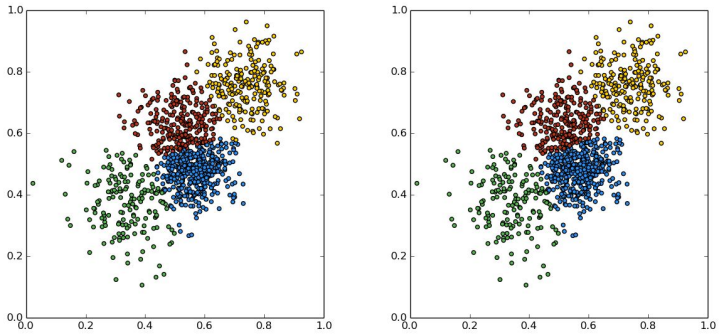
Fig. 15. Output quality comparison for *Blur* application running on (a) exact computing, (b) approximate mode ($PSNR = 25dB$), and (c) tuned computing with $PSNR = 34dB$ and 13% run on precise CFPU.



(a) Sobel (b) MersenneTwister

Fig. 16. Error distribution for applications.

operation, resulting in the highest energy savings, but poor accuracy. Adaptive selection reduces error but adds a small additional overhead. Tuning is used to reaching accuracy requirements, but



(a) Exact (b) Approx

Fig. 17. Output quality comparison for *K-means* application running on (a) exact computing, (b) approximate mode with 100% run on CFPU and 2.05% error.

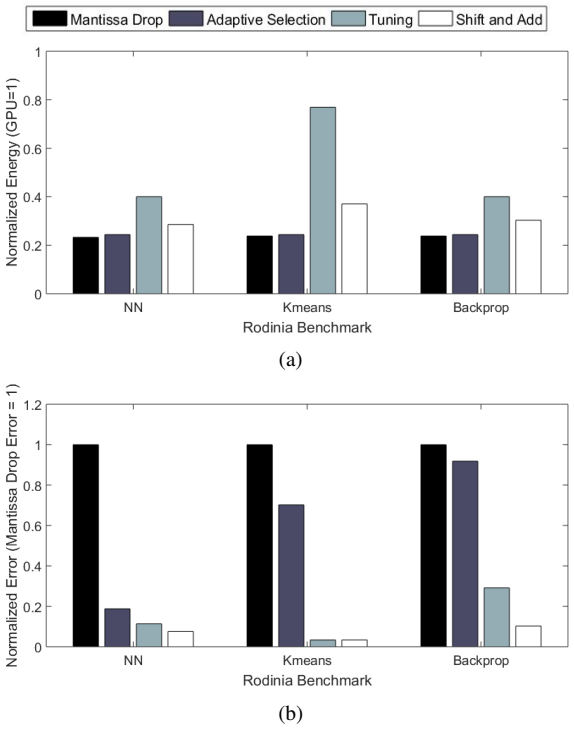


Fig. 18. Improvements from CFPU optimizations for (a) energy and (b) output error.

energy savings is decreased drastically because a large portion of operations run on exact hardware. Finally, the second stage shift and add are used to reduce energy, while still maintaining accuracy. Our evaluation shows that the proposed CFPU design can achieve 4.1× (3.2×) EDP improvement while ensuring less than 10% (1%) average relative error. The tested algorithms performed well when coupled with approximate hardware. Figure 17 compares the output of *K-means* on exact and approximate hardware. All classification errors occur around the boundaries of the clusters. All operations are computed approximately and only 2% of the points are incorrectly classified.

5.4 Overhead & Comparison

The first stage adds a 3.4% area overhead to the FPU, while the second stage adds an extra 6.2% area overhead. The energy overhead of a multiply operation when running on CFPU in exact mode is 2.7%, which is negligible compared to efficiency and tuning capability that CFPU can provide. In order to outperform the standard FPU, our design needs to run at least 4% of the data in the 1st or 2nd stage. We observed significantly higher percentages in all of the applications tested on the proposed CFPU.

TABLE 6
Comparing the energy, and performance of the CFPU using 3 tuning bits and previous designs ensuring acceptable level of accuracy.

	Power(mW)	Delay(ns)	EDP (pJs)	Max Multiply Error	Tunable
CFPU (3 Bits)	0.17	1.6	0.44	6.3%	***Yes***
DRUM6 [26]	0.29	1.9	1.04	6.3%	*No*
ESSM8 [28]	0.28	2.1	1.2	11.1%	*No*
Kulkarni [30]	0.82	3.5	10.0	22.2%	*No*

To understand the advantage of the proposed design, we compare the energy consumption and delay of the proposed CFPU with the state-of-the-art approximate multipliers proposed in [26], [28], [30]. Previous designs are limited to a small range of robust and error-tolerant applications, as they are not able to tune the level of accuracy at runtime. In contrast, our CFPU dynamically predicts the inaccurate results and processes them in precise mode. CFPU tunes the level of accuracy at runtime based on the user accuracy requirement. The ability to run in exact mode and save power increases the range of applications that benefit from it. Table 6 lists the power consumption, critical path delay, and energy-delay product of CFPU alongside previous work in [26], [28] and [30] in their best configurations. We set CFPU to use 3 tuning bits, so the maximum output error per operation is the same or less than the multipliers we compare against. Tuning requires bit checks which increase energy consumption, so a CFPU configured to check 3 bits has slightly fewer energy savings than one configured to predict error. Our evaluation shows that at the same level of accuracy, the proposed design can achieve 2.8× EDP improvement compared to the state-of-the-art approximate multipliers for a multiply operation.

6 Conclusion

In this paper, we propose a configurable floating point multiplier which can approximately perform the computation with significantly lower energy and performance cost. CFPU controls the level of approximation by processing the data in one of the three tiers: basic approximate mode, intermediate approximate mode, and on the exact hardware. The first stage approximate mode discards one input's mantissa and uses the second's directly in the output to save energy. Accuracy is tuned by examining the discarded mantissa to estimate output error. When error exceeds a user-specified maximum, CFPU uses a 2^{nd} level of approximation. This mode uses a shift and add to increase accuracy. If the approximate output error is too high, the multiply is run on exact hardware. Our results show that using first stage CFPU approximation results in 3.5× energy-delay product (EDP) improvement compared to an unmodified FPU, while ensuring less than 10% average relative error [32]. Adding the second stage further increases the EDP improvement, compared to the base FPU, to 4.1× for that same level of accuracy. In addition, our results show the proposed CFPU achieves 2.8× EDP improvement for multiply operations as compared to the state-of-the-art approximate multipliers.

ACKNOWLEDGMENT

This work was partially supported by CRISP, one of six centers in JUMP, an SRC program sponsored by DARPA, and also NSF grants #1730158 and #1527034.

REFERENCES

[1] J. Gantz and D. Reinsel, "Extracting value from chaos," *IDC iview*, vol. 1142, no. 2011, pp. 1–12, 2011.

[2] L. Atzori *et al.*, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[3] M. K. Tavana, A. Kulkarni, A. Rahimi, T. Mohsenin, and H. Homayoun, "Energy-efficient mapping of biomedical applications on domain-specific accelerator under process variation," in *Low Power Electronics and Design (ISLPED), 2014 IEEE/ACM International Symposium on*, pp. 275–278, IEEE, 2014.

[4] C. Ji *et al.*, "Big data processing in cloud computing environments," in *I-SPAN*, pp. 17–23, IEEE, 2012.

[5] N. Khoshavi, X. Chen, J. Wang, and R. F. DeMara, "Read-tuned stt-ram and edram cache hierarchies for throughput and energy enhancement," *arXiv preprint arXiv:1607.08086*, 2016.

[6] K. Kambatla, G. Kollias, V. Kumar, and A. Grama, "Trends in big data analytics," *Journal of Parallel and Distributed Computing*, vol. 74, no. 7, pp. 2561–2573, 2014.

[7] M. Nazemi and M. Pedram, "Deploying customized data representation and approximate computing in machine learning applications," *arXiv preprint arXiv:1806.00875*, 2018.

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

[9] M. Imani, A. Rahimi, and T. S. Rosing, "Resistive configurable associative memory for approximate computing," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 1327–1332, IEEE, 2016.

[10] Q. Zhang, T. Wang, Y. Tian, F. Yuan, and Q. Xu, "Approxann: an approximate computing framework for artificial neural network," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pp. 701–706, EDA Consortium, 2015.

[11] M. Imani, M. Samragh, Y. Kim, S. Gupta, F. Koushanfar, and T. Rosing, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.

[12] S. Venkataramani, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Approximate computing and the quest for computing efficiency," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 120, ACM, 2015.

[13] M. Imani, D. Peroni, Y. Kim, A. Rahimi, and T. Rosing, "Efficient neural network acceleration on gpgpu using content addressable memory," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1026–1031, IEEE, 2017.

[14] M. Samragh, M. Ghasemzadeh, and F. Koushanfar, "Customizing neural networks for efficient fpga implementation," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pp. 85–92, IEEE, 2017.

[15] M. Imani *et al.*, "Exploring hyperdimensional associative memory," in *IEEE HPCA*, IEEE, 2017.

[16] M. Courbariaux *et al.*, "Low precision arithmetic for deep learning," *arXiv:1412.7024*, 2014.

[17] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1775–1780, IEEE, 2017.

[18] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, p. 62, 2016.

[19] J. Liang *et al.*, "Floating point unit generation and evaluation for fpgas," in *FCCM*, pp. 185–194, IEEE, 2003.

[20] S. P. Gnawali, S. N. Mozaffari, and S. Tragoudas, "Low power spintronic ternary content addressable memory," *IEEE Transactions on Nanotechnology*, 2018.

[21] S. Venkataramani, A. Ranjan, K. Roy, and A. Raghunathan, "Axnn: energy-efficient neuromorphic systems using approximate computing," in *Proceedings of the 2014 international symposium on Low power electronics and design*, pp. 27–32, ACM, 2014.

[22] A. Suhre, F. Keskin, T. Ersahin, R. Cetin-Atalay, R. Ansari, and A. E. Cetin, "A multiplication-free framework for signal processing and applications in biomedical image analysis," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pp. 1123–1127, IEEE, 2013.

[23] M. Imani, S. Patil, and T. S. Rosing, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, pp. 373–378, EDA Consortium, 2016.

[24] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "Snnap: Approximate computing on programmable socs via neural acceleration," in *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pp. 603–614, IEEE, 2015.

[25] W. José, A. R. Silva, H. Neto, and M. Véstias, "Efficient implementation of a single-precision floating-point arithmetic unit on fpga," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–4, IEEE, 2014.

[26] S. Hashemi, R. Bahar, and S. Reda, "Drum: A dynamic range unbiased multiplier for approximate applications," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*, pp. 418–425, IEEE Press, 2015.

[27] M. Imani, R. Garcia, S. Gupta, and T. Rosing, "Rmac: Runtime configurable floating point multiplier for approximate computing," in *Proceedings of the International Symposium on Low Power Electronics and Design*, p. 12, ACM, 2018.

[28] S. Narayanamoorthy *et al.*, "Energy-efficient approximate multiplication for digital signal processing and classification applications," *TVLSI*, vol. 23, no. 6, pp. 1180–1184, 2015.

[29] C. Liu, J. Han, and F. Lombardi, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 95, European Design and Automation Association, 2014.

[30] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, pp. 346–351, IEEE, 2011.

[31] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pp. 44–54, Ieee, 2009.

[32] M. Imani, D. Peroni, and T. Rosing, "Cfpu: Configurable floating point multiplier for energy-efficient computing," in *Design Automation Conference (DAC), 2017 54th ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2017.

[33] M. K. Tavana, M. H. Hajkazemi, D. Pathak, I. Savidis, and H. Homayoun, "Elasticcore: enabling dynamic heterogeneity with joint core and voltage/frequency scaling," in *Proceedings of the 52nd Annual Design Automation Conference*, p. 151, ACM, 2015.

[34] P. K. Krause and I. Polian, "Adaptive voltage over-scaling for resilient applications," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pp. 1–6, IEEE, 2011.

[35] M. Imani, A. Rahimi, P. Mercati, and T. Rosing, "Multi-stage tunable approximate search in resistive associative memory," *IEEE Transactions on Multi-Scale Computing Systems*, 2017.

[36] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: imprecise adders for low-power approximate computing," in *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, pp. 409–414, IEEE Press, 2011.

[37] M. Imani, S. Patil, and T. Rosing, "Approximate computing using multiple-access single-charge associative memory," *IEEE Transactions on Emerging Topics in Computing*, 2016.

[38] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *Design Automation Conference (DAC), 2016 53nd ACM/EDAC/IEEE*, pp. 1–6, IEEE, 2016.

[39] X. Yin, M. T. Niemier, and X. S. Hu, "Design and benchmarking of ferroelectric fet based tcam," *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 1444–1449, 2017.

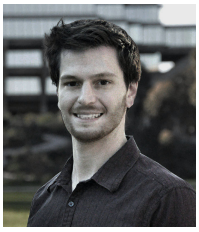
[40] X. Chen, X. Yin, M. Niemier, and X. S. Hu, "Design and optimization of fefet-based crossbars for binary convolution neural networks," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pp. 1205–1210, IEEE, 2018.

[41] M. Imani, Y. Kim, A. Rahimi, and T. Rosing, "Acam: Approximate computing based on adaptive associative memory with online learning.," in *ISLPED*, pp. 162–167, 2016.

[42] M. Imani, P. Mercati, and T. Rosing, "Remam: low energy resistive multi-stage associative memory for energy efficient computing," in *Quality Electronic Design (ISQED), 2016 17th International Symposium on*, pp. 101–106, IEEE, 2016.

[43] M. Imani *et al.*, "Resistive cam acceleration for tunable approximate computing," *IEEE Transactions on Emerging Topics in Computing*, 2017.

[44] V. Camus, J. Schlachter, C. Enz, M. Gautschi, and F. K. Gurkaynak, "Approximate 32-bit floating-point unit design with 53% power-area product reduction," in *European Solid-State Circuits Conference, ESSCIRC Conference 2016: 42nd*, pp. 465–468, Ieee, 2016.
[45] C.-H. Lin and C. Lin, "High accuracy approximate multiplier with error correction," in *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pp. 33–38, IEEE, 2013.
[46] S. A and R. C, "Carry speculative adder with variable latency for low power vlsi," in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, pp. 2400–2402, March 2016.
[47] M. Nazemi and M. Pedram, "Deploying Customized Data Representation and Approximate Computing in Machine Learning Applications," *ArXiv e-prints*, June 2018.
[48] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14, Feb 2014.
[49] P. Yin, C. Wang, W. Liu, and F. Lombardi, "Design and performance evaluation of approximate floating-point multipliers," in *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pp. 296–301, IEEE, 2016.
[50] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2sim: a simulation framework for cpu-gpu computing," in *Parallel Architectures and Compilation Techniques (PACT), 2012 21st International Conference on*, pp. 335–344, IEEE, 2012.
[51] D. Compiler, "Synopsys inc," 2000.
[52] " AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK)." http://developer.amd.com/sdks/amdappsdk/.
[53] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 449–460, IEEE Computer Society, 2012.



Daniel Peroni completed his B.S. in Computer Engineering at California Polytechnic State University in 2015. As of 2018, he is a PhD student in the Department of Computer Science and Engineering at the University of California San Diego. He is a member of the System Energy Efficient Laboratory (SeeLab), where he is investigating approximate computing using non-volatile memory solutions, GPU acceleration, and machine learning.

Mohsen Imani received his M.S. and BSc degrees from the School of Electrical and Computer Engineering at the University of Tehran in March 2014 and September 2011 respectively. From September 2014, he is a Ph.D. student in the Department of Computer Science and Engineering at the University of California San Diego, CA, USA. He is a member of the System Energy Efficient Laboratory (SeeLab). Mr. Imani research interests are in brain-inspired computing, approximation computing, and processing in-memory architectures.

Tajana imuni Rosing is a Professor, a holder of the Fratamico Endowed Chair, IEEE Fellow, and a director of System Energy Efficiency Lab at UCSD. Tajana imuni Rosing is a Professor, a holder of the Fratamico Endowed Chair, IEEE Fellow, and a director of System Energy Efficiency Lab at UCSD. Her research interests are in energy efficient computing, cyber-physical, and distributed systems. She is leading a number of projects, including efforts funded by DARPA/SRC JUMP CRISP program, with focus on design of accelerators for analysis of big data, a project focused on developing AI systems in support of healthy living, SRC funded project on IoT system reliability and maintainability, and NSF funded project on design and calibration of air-quality sensors and others. She recently headed the effort on SmartCities that was a part of DARPA and industry-funded TerraSwarm center. During 2009-2012 she led the energy efficient datacenters theme as a part of the MuSyC center. From 1998 until 2005 she was a full time research scientist at HP Labs while also leading research efforts at Stanford University. She finished her PhD in 2001 at Stanford University, concurrently with finishing her Masters in Engineering Management. Her PhD topic was Dynamic Management of Power Consumption. Prior to pursuing the PhD, she worked as a Senior Design Engineer at Altera Corporation.