CFPU:Configurable Floating Point Multiplier for Energy-Efficient Computing

Mohsen Imani, Daniel Peroni and Tajana Rosing
CSE Department, UC San Diego, La Jolla, CA 92093, USA
{moimani, dperoni, tajana}@ucsd.edu

ABSTRACT

Many applications, such as machine learning and data sensing are statistical in nature and can tolerate some level of inaccuracy in their computation. Approximate computation is a viable method to save energy and increase performance by trading energy for accuracy. There are a number of proposed approximate solutions, however, they are limited to a small range of applications because they cannot control the error rate of their output. In this paper, we propose a novel approximate floating point multiplier, called CFPU, which significantly reduces energy and improves performance of multiplication at the expense of accuracy. Our design approximately models multiplication by replacing the most costly step of the operation with a lower energy alternative. In order to tune the level of approximation, CFPU dynamically identifies the inputs which will produce the largest approximation error and processes them in precise CFPU mode. We showed that our CFPU can outperforms a standard FPU when at least 4% of multiplications are performed in approximate mode. In our tested applications this percentage of multiplications is substantially higher, leading to significant energy savings. Our experimental evaluation on AMD Southern Island GPU shows that replacing the proposed CFPU with traditional FPUs results in 77% energy savings and 3.5× energy-delay product improvement over eight general Open*CL* applications while providing acceptable quality of service. In addition, for the same level of accuracy, the CFPU provides 2.4× energy-delay product improvement compared to state-of-the-art approximate multipliers.

CCS CONCEPTS

•Hardware → Reconfigurable logic applications; •Theory of computation → Stochastic approximation;

KEYWORDS

Approximate computing, Floating point unit, Energy efficiency

ACM Reference format:
Mohsen Imani, Daniel Peroni and Tajana Rosing. 2016. CFPU:Configurable Floating Point Multiplier for Energy-Efficient Computing. In *Proceedings of ACM conference, Austin, Texas USA, June 2017 (DAC '17),* 6 pages.
DOI: http://dx.doi.org/10.1145/3061639.3062210

1 INTRODUCTION

In 2015, the number of smart devices around the world exceeded 25 billion. This number is expected to double by 2020 [1, 2]. Many of these devices have batteries with strict power constraints, so the need for systems that can efficiently handle the computing requirements of data-intensive workloads is undeniable [3, 4]. Running machine learning algorithms or multimedia applications on general purpose processors, e.g. GPUs, CPUs, and FPGAs, results in large energy consumption and performance inefficiencies. Many of these applications do not need highly accurate computation, so by accepting slight inaccuracy, instead of doing all computation precisely, we can get significant energy and performance improvements [5, 6].
Several data processing applications use a large range of values and require high precision. Therefore, the main computations in many traditional and state-of-the-art computing systems are based on floating point units (FPUs) [7, 8]. For example in CPUs, video processing or high performance scientific computations require large amounts of power. To cover the same dynamic range, the fixed point unit must be five times larger and 40% slower than a corresponding floating point [9].

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 streaming processes [10, 11]. Several techniques have been introduced to address multiplication costs by designing an approximate multiplication unit. Most of prior work attempted to reduce the bit-size of multiplication or use multiplication in different sizes to enable approximation [12, 13]. However, either the lack of accuracy tuning or the large area requirements of the tuned designs, significantly reduces the advantages provided by these approximation designs.

In this paper, we propose a configurable floating point multiplication, called CFPU, which significantly improves the multiplication energy consumption by trading for accuracy. CFPU avoids the costly multiplication when calculating the fractional part of a floating point number by discarding one of the input mantissa and using the second directly. To tune the level of accuracy, our design adaptively distinguishes the inputs that will result in the highest output error and assigns them to compute precisely on the CFPU. We evaluate the efficiency of proposed technique on AMD Southern Island GPU architecture by replacing the traditional FPUs with the proposed CFPU. Our evaluation shows that the proposed CFPU can achieve 77% energy savings and 3.5× energy-delay product improvement over eight general Open*CL* applications, while providing acceptable quality of service. In addition, the comparison of the proposed CFPU with previous state-of-the-art multipliers [12–15] shows that our design can achieve 2.4× higher energy-delay product while providing less error.

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

DAC '17, Austin, Texas USA

DOI: http://dx.doi.org/10.1145/3061639.3062210

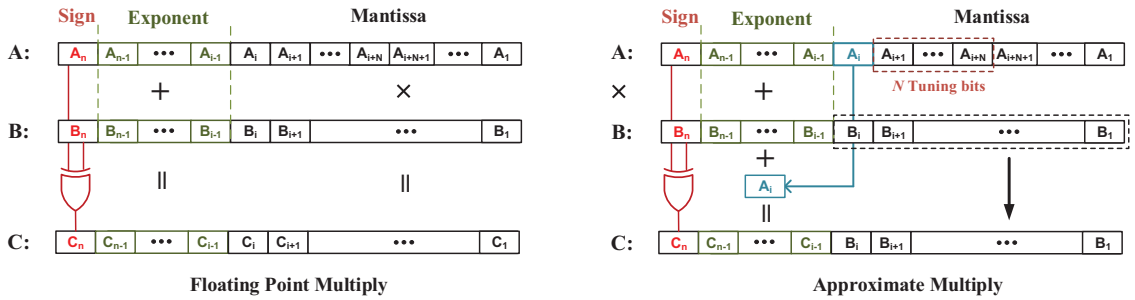


Figure 1: Approximate multiplication of proposed CFPU between A ad B operands.

threshold is reached [16, 17]. However, 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.

Another recently emerged strategy is the application of Non-volatile memories (NVM) to create approximate memory units, for energy efficient storage and computing purposes [6, 18]. 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 [19, 20]. This approach does not work well in applications without a large number of redundant calculations. Associative memory can be integrated into FPUs reduce these redundancies.

Finally approximate hardware involves redesigning basic component blocks to save energy, at the cost of accurate output [12, 14, 21, 22]. Liu *et al.* utilize approximate adders to create an energy efficient approximate multiplier [14]. Hashemi *et al.* designed a multiplier that selects a reduced number of bits used in the multiplication to conserve power [12]. Camus *et al.* propose a speculative approximate multiplier combines gate-level pruning and an inexact speculative adder to lower power consumption and shrink FPU area [21].

In contrast to previous work, we design a configurable approximate floating point multiplier which approximately processes data using an input mantissa directly in the output. Our design produces exact output for multiples of 2, but the level of accuracy for other cases can be tuned by adaptively assigning far distance data to precise cores to compute.

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, neural networks and other learning and streaming applications [5, 12]. For instance, looking at general OpenCL applications, we observed about 85% of floating point arithmetic involved multiplication. In order to make multiplication efficient, we propose a technique where costly mantissa multiplication is implemented by reusing the one of the input values in the output.

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.

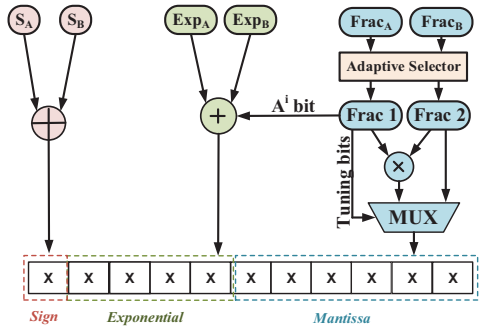


Figure 2: CFPU Integration with adaptive selector and N tuning bits

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 fall 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.

Our proposed design, which can be seen in Figure 2, uses three techniques: mantissa approximation, adaptive selection, and tuning. Mantissa approximation involves removing the costly mantissa multiplication by copying one mantissa to the output and discarding the second. Adaptive selection checks for a mantissa which will produce 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 exceeded, the calculation is run in exact mode. We further explain the three modifications in the following sections.

3.1 Mantissa Approximation

The multiplication of the mantissas is the most costly operation, taking 80% of the total energy of the multiply operation [23], so our approach removes it entirely. Instead of 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 the mantissa is

1, the output error is 0%.

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

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

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

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

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.

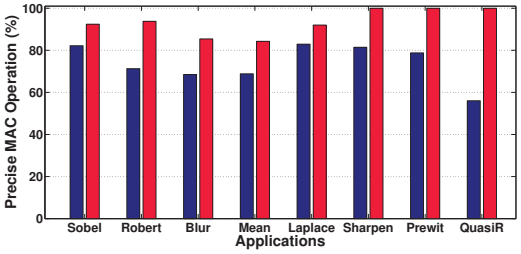
3.2 Adaptive Operand Selection

Because the approximate multiply uses both exponents in its calculation, but discards one of the mantissas, an operation always multiplies by a power of 2. Therefore, a multiplication by a power of 2 will always result 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. Multiples of 2 are common in many applications, so having hardware intelligently check both inputs and adaptively discard a mantissa with value 1 (all mantissa bits are 0) when possible to greatly reduce output error.

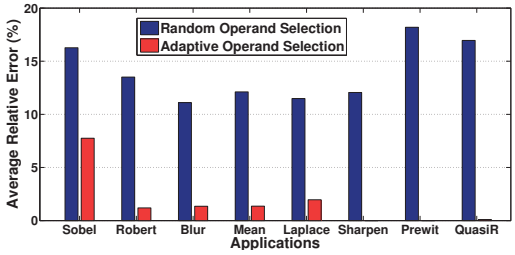
Figure 3.a compares the portion of multiplications which can run precisely on the proposed CFPU with and without adaptive operand selection for the OpenCL applications we evaluated. Consequently, Figure 3.b shows the impact of the adaptive operand selection on the computation accuracy of the proposed CFPU. The result shows that adaptive operand selection, significantly improves computation accuracy such that for all shown applications, the average relative error reduces 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 a naive approximate approach, while adaptive selection shows 92% of the outputs calculated exactly, at an overall relative error of 9%. Our evaluation over eight different applications shows that this adaptive selection technique can improve the computation accuracy by 8× compared to a naive selection.

3.3 Tuning Accuracy

Although proposed approximate multiplication provides high energy savings, the accuracy of computation is heavily impacted depending on application. For some applications, with quantized inputs, e.g., *Sharpen filter*, the proposed design can work precisely with no

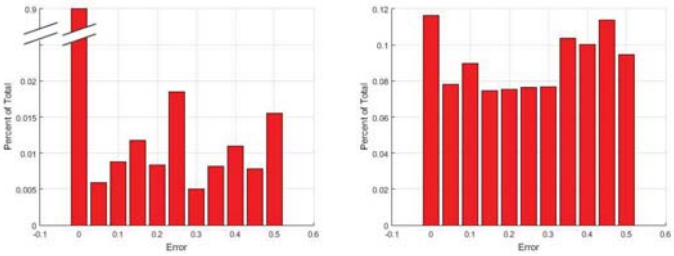


(a)



(b)

Figure 3: a) The portion of precise CFPU computation in different applications and b) the impact of smart operand selection on the computation accuracy.



(a) Sobel (b) MersenneTwister

Figure 4: Error distribution for applications.

quality loss. In addition, many recognition algorithms, such as motion tracking and plate detection applications, only need to quantify changes in the input data. Therefore, the approximate multiplication can provide high accuracy, close to precise, computations on those applications. Figure 4 shows the distribution of error rates for inputs of two applications. In the case of *Sobel*, almost 90% of the multiplies are by a multiple 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 more even distribution of error rates. While about 12% of the computations will have 0% error, the error rates are too randomly distributed to provide acceptable overall error without an additional optimization.

To generalize the functionality of proposed design for general applications (with acceptable approximation), our design requires the ability to tune the level of accuracy. To this end, our design finds inputs that will produce large output inaccuracy and then processes them on FPU precisely. The N bits after the first of input A's mantissa is used to tune the level of approximation. The case of maximum error occurs when mantissa A is furthest from a multiple of 2, which

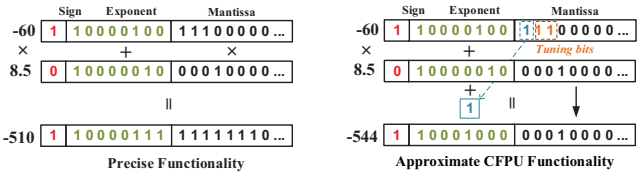


Figure 5: An example of 32-bit multiplication in conventional precise FPU and proposed CFPU using N=2 tuning bits.

occurs when its value is 1 followed by all 0s. When tuning, the goal is to ensure values close to this maximum error point are selected to run exactly over values with lower error. Therefore, the hardware must change it's selection depending on the value of the first bit of input A's mantissa. If the first bit of mantissa A is 1, the first N tuning bits are checked for 0s, where N is selected based on desired accuracy. If a 0 is found, the hardware will run in exact mode. For each guaranteed bit in the A_{i-1} to 1^{st} indexes, the maximum error is reduced by half.

$$Error = \left| \sum_{i=N}^{n-1} 2^{-((n-i)A_{n-i-1})} - 0.5 \right|. \quad (3)$$

When the first bit is 0, the first N tuning bits are checked for 1s instead. Because values closest to the non-tuned 50% error are run on exact hardware first, the overall error rate for an application can be low, with only a small number of multiplies computed exactly.

An example of CFPU multiplication is shown in Figure 5 for two 32-bit floating point numbers in precise FPU and proposed CFPU using 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. In contrast, our design checks the first mantissa bit and the N tuning bits after that. In this case, the first mantissa bit is 1, so the output exponent value is increased by 1. The next two bits are checked for against 0 to determine if the value will stay below a 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 will continue 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.

3.4 CFPU Hardware Support

Figure 6 shows the circuitry to enable CFPU adaptive operand selection and accuracy tuning. We implement adaptive operand selection by only checking the mantissa bits in one of the input operands. In the default case, our design copies the mantissa of the A operand for the mantissa of the multiplication output, unless the detector circuitry determines all mantissa bits of the A operand are zero. In that case, CFPU selects the mantissa of the operand B as the output operand.

As Figure 6 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 will detect changes in the ML voltage (ML=1). However, if all mantissa bits are zero, no current will pass through R_{sense} and the B operand mantissa will be 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

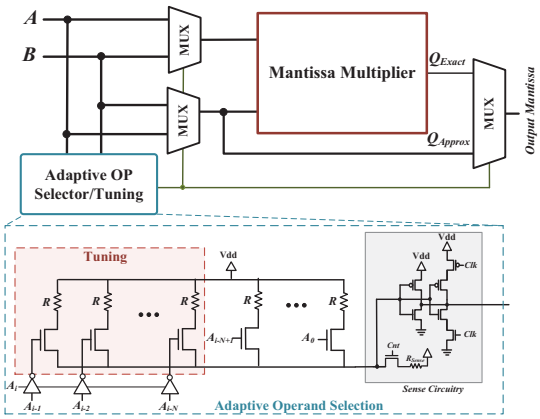


Figure 6: circuitry to support adaptive operand selector and tuning the level of approximation in CFPU.

CFPU core. The operand selection happens by using two multiplexer 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 deign area efficient, we use the same circuitry for adaptive operand selection and tuning approximation. For each application, this sampling time can individually set in order to provide target accuracy.

4 RESULTS

4.1 Experimental Setup

We integrated the proposed approximate CFPU on the floating point units of an AMD Southern Island GPU, which has been commercially used, e.g., Radeon HD 7970 device. We modified Multi2sim, a cycle accurate CPU-GPU simulator [24] to model the CFPU functionality on three main floating point units in GPU architecture: multiplier, multiplier-accumulator (MAC) and Multiplier-then-adder (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 [25]. The circuit level simulation of the CFPU design has been performed using HSPICE simulator in 45-nm TSMC technology. We test the efficiency of enhanced GPU on eleven general Open*CL* applications: *Sobel, Robert, Mean, Laplacian, Sharpen, Prewit, QuasiRandom, FFT, Mersenne, DwHaar1D* and *Blur*. In these applications, roughly 85% of the floating point operations involve multiplication.

We propose an automated framework to fine tune the level of approximation and satisfy required required accuracy while providing the maximum energy savings. Figure 7 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. Then, based on the user accuracy requirement, it dynamically decreases the level of approximation until computation accuracy satisfies the user quality of service. For each application, this framework returns the optimal number of CFPU

Table 1: Energy and performance improvement and quality of loss replacing GPU with proposed floating point multiplications.

Applications	Sobel	Robert	Mean	Laplacian	Sharpen	Prewit	QuasiR	Average Improv.
Energy savings	84%	83%	81%	76%	75%	69%	72%	**77%**
Speed up	21%	24%	27%	16%	20%	22%	12%	**20%**
EDP improvement	8.3×	7.7×	6.4×	4.7×	5.3×	4.2×	4.1×	**5.8×**
QoL (%)	9.02%	1.19%	1.36%	1.96%	0%	0%	0%	**1.93%**

Table 2: Ratio of approximate to total operations and quality of loss running applications in different tuning mode.

Tuning bits	Sobel		Robert		Mean		Laplacian		FFT		Mersenne		DwtHaar1D		Blur	
	Approx/total	*QoL*	*Approx/Total*	*QoL*	*Approx/Total*	*QoL*	*Approx/Total*	*QoL*	*Approx/Total*	*QoL*	*Approx/Total*	*QoL*	*Approx/Total*	*QoL*	*Approx/Total*	*QoL*
0 bit	100%	9.02%	100%	1.19%	100%	1.36%	100%	1.96%	100%	73%	100%	12%	100%	94%	100%	11.1%
1 bit	96%	2.70%	97%	0.35%	98%	1.04%	96%	0.50%	54%	9.8%	60%	8.8%	66%	31%	82%	3.7%
2 bits	94%	0.74%	95%	0.10%	85%	0.03%	94%	0.11%	32%	8.3%	43%	3.4%	55%	12%	76%	0.92%
3 bits	93%	0.07%	94%	0.03%	85%	0.01%	93%	0.02%	21%	4.1%	33%	1.6%	53%	8.3%	62%	0.36%
4 bits	92%	0.01%	94%	~0%	84%	~0%	92%	~0%	15%	2.3%	29%	0.7%	47%	0.7%	53%	0.21%
Exact	92%	0%	93%	0%	84%	0%	92%	0%	10%	0%	23%	0%	45%	0%	53%	0%

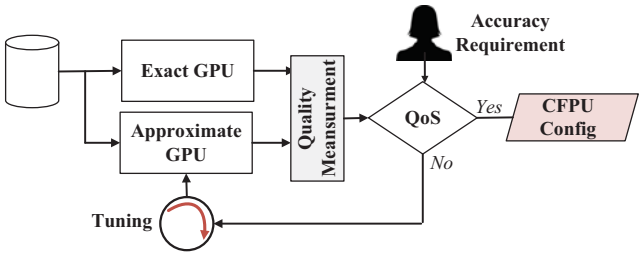


Figure 7: Framework to support tunable CFPU approximation.

tuning bits checked, providing maximum energy and performance efficiency.

4.2 Approximate CFPU

To efficiently show the application of the proposed CFPU design, 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. 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 77% energy savings, 20% speedup, and 5.8× energy-delay product compared to the traditional AMD GPU, while providing an acceptable output quality of less than 10% average relative error.

4.3 Tunable CFPU Computing

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 [26]. 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 quality of loss and portion of running multiplications in each application on exact and on approximate CFPU, when the number of tuning bit changes from none to 4 bits. Increasing the number of tuning bits improves the computation accuracy by processing the far and inaccurate multiplications in

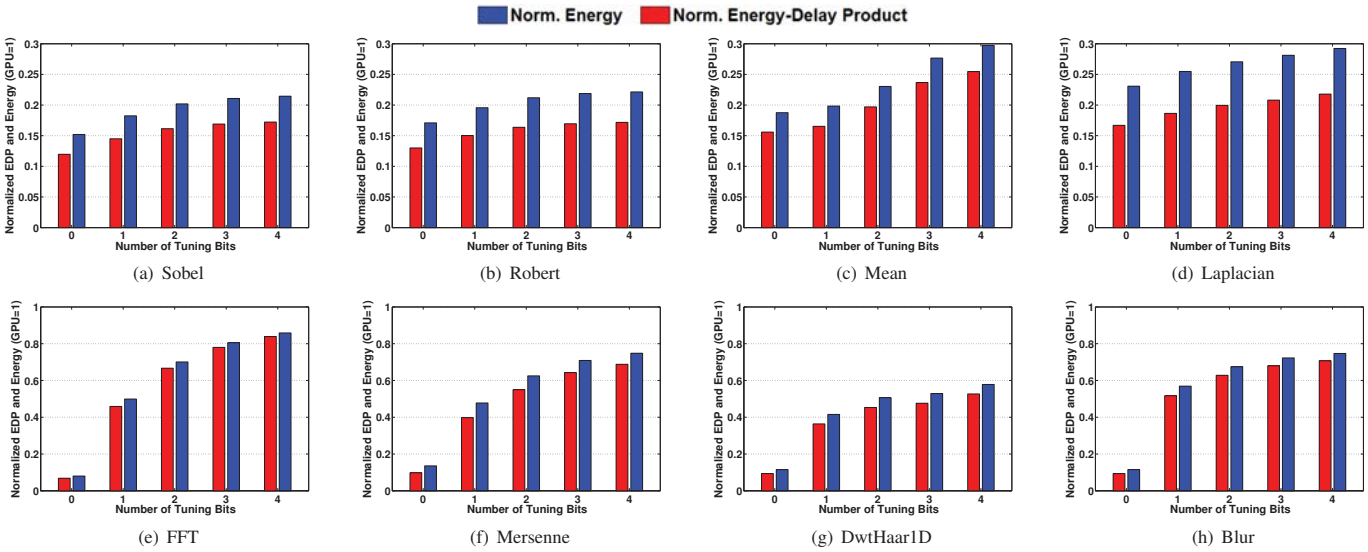
Table 3: Comparing the energy, and performance of the CFPU and previous designs ensuring acceptable level of accuracy.

	Power(mW)	Delay(ns)	EDP (pJs)	Max.Error	Tunable
CFPU 3	0.17	1.6	0.44	6.3%	***Yes***
DRUM6 [12]	0.29	1.9	1.04	6.3%	*No*
ESSM8 [13]	0.28	2.1	1.2	11.1%	*No*
Kulkarni [15]	0.82	3.5	10.0	22.2%	*No*

precise CFPU mode. On the other hand, more number of tuning bits slows down the computation, because a larger portion of data is processed on precise CFPU. Figure 8 shows the energy consumption and performance of a GPU enhanced with tunable CFPU using different numbers of tuning bits. Our experimental evaluation shows that running applications on proposed CFPU provides 3.5× energy-delay product improvement compared to a GPU using traditional FPUs, while ensuring less than 10% quality of loss. In addition, accepting less then 1% error, the CFPU can still achieve 2.7× EDP improvement compared to a GPU using traditional FPUs. To ensure the quality of computation, Figure 9 compares the visual results of *Blur* running on precise and approximate hardware. Our result shows that approximate computing create no noticeable difference between the precise and approximate result images.

To modify the standard FPU with our CFPU hardware, we need to have 3.4% area overhead, and 2.7% energy overhead (if CFPU runs in exact mode), 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 approximate mode (or in 4% of the multiplications, one of the input operands is a power of 2). This number is significantly smaller than the numbers that we observed when running the tested applications on the proposed CFPU.

To understand the advantage of proposed design, we compare the energy consumption and delay of the proposed CFPU with the state-of-the-art approximate multipliers proposed in [12, 13, 15]. The application of previous designs limits to a small range of robust and error tolerant applications, as they are not able to tune the level of accuracy in runtime. In contrast, the proposed CFPU dynamically finds the inaccurate data and processes them in precise mode. CFPU tunes the level of accuracy at runtime based on the user accuracy requirement. This makes the application of CFPU general. Table 3 lists the power consumption, critical path delay, and energy-delay product of CFPU alongside previous work in [12], [13] and [15] in their best configurations. Our evaluation shows that at the same level of accuracy, the proposed design can achieve 2.4× EDP improvement compared to the state-of-the-art approximate multipliers.



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

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

Figure 8: Energy consumption and energy-delay product of enhanced GPU with tunable CFPU normalized to GPU with conventional FPU.

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

Figure 9: 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.

5 CONCLUSION

In this paper, we propose a configurable floating point multiplier which can approximately perform the computation with significantly lower energy and performance cost. The proposed approximate multiplication has tuning capability by adaptively processing an uncertain part of data precisely. Our design considers floating point implementation of proposed approximate design. Our experimental evaluation shows that replacing the CFPU with existing FPUs results in 77% energy savings and 4.8× energy-delay product improvement, while providing less than 10% quality loss. In addition, our result shows, that at the same level of accuracy the proposed CFPU can achieve 2.4× lower energy-delay product compared to state-of-the-art approximate multipliers.

6 ACKNOWLEDGMENT

This work was supported by NSF grant #1527034 and Jacobs School of Engineering UCSD Powell Fellowship.

REFERENCES

[1] J. Gantz *et al.*, "Extracting value from chaos," *IDC iview*, vol. 1142, 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] C. Ji *et al.*, "Big data processing in cloud computing environments," in *I-SPAN*, pp. 17–23, IEEE, 2012.
[4] 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.
[5] J. Han *et al.*, "Approximate computing: An emerging paradigm for energy-efficient design," in *IEEE ETS*, pp. 1–6, IEEE, 2013.
[6] M. Imani *et al.*, "Resistive configurable associative memory for approximate computing," in *DATE*, pp. 1327–1332, IEEE, 2016.
[7] M. Courbariaux *et al.*, "Low precision arithmetic for deep learning," *arXiv:1412.7024*, 2014.
[8] M. Samragh *et al.*, "Looknn: Neural network with no multiplication," in *IEEE/ACM DATE*, 2017.
[9] J. Liang *et al.*, "Floating point unit generation and evaluation for fpgas," in *FCCM*, pp. 185–194, IEEE, 2003.
[10] A. Suhre *et al.*, "A multiplication-free framework for signal processing and applications in biomedical image analysis," in *ICASSP*, pp. 1123–1127, IEEE, 2013.
[11] M. Imani *et al.*, "Masc: Ultra-low energy multiple-access single-charge tcam for approximate computing," in *IEEE/ACM DATE*, pp. 373–378, IEEE, 2017.
[12] S. Hashemi *et al.*, "tldrum: A dynamic range unbiased multiplier for approximate applications," in *ICCAD*, pp. 418–425, IEEE Press, 2015.
[13] S. Narayanamoorthy *et al.*, "Energy-efficient approximate multiplication for digital signal processing and classification applications," *TVLSI*, vol. 23, no. 6, pp. 1180–1184, 2015.
[14] C. Liu *et al.*, "A low-power, high-performance approximate multiplier with configurable partial error recovery," in *IEEE/ACM DATE*, p. 95, IEEE, 2014.
[15] P. Kulkarni *et al.*, "Trading accuracy for power with an underdesigned multiplier architecture," in *IVLSI*, pp. 346–351, IEEE, 2011.
[16] P. K. Krause *et al.*, "Adaptive voltage over-scaling for resilient applications," in *DATE*, pp. 1–6, IEEE, 2011.
[17] M. Imani *et al.*, "Multi-stage tunable approximate search in resistive associative memory," *IEEE TMSCS*, 2017.
[18] Y. Kim *et al.*, "Cause: critical application usage-aware memory system using non-volatile memory for mobile devices," in *IEEE/ACM ICCAD*, pp. 690–696, IEEE, 2015.
[19] M. Imani *et al.*, "Acam: Approximate computing based on adaptive associative memory with online learning," in *IEEE/ACM ISLPED*, pp. 162–167, 2016.
[20] M. Imani *et al.*, "Remam: low energy resistive multi-stage associative memory for energy efficient computing," in *IEEE ISQED*, pp. 101–106, IEEE, 2016.
[21] V. Camus *et al.*, "Approximate 32-bit floating-point unit design with 53% power-area product reduction," in *ESSCIRC*, pp. 465–468, IEEE, 2016.
[22] 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.
[23] P. Yin *et al.*, "Design and performance evaluation of approximate floating-point multipliers," in *ISVLSI*, pp. 296–301, IEEE, 2016.
[24] R. Ubal *et al.*, "Multi2sim: a simulation framework for cpu-gpu computing," in *PACT*, pp. 335–344, ACM, 2012.
[25] D. Compiler, "Synopsys inc," 2000.
[26] M. Imani *et al.*, "Approximate computing using multiple-access single-charge associative memory," *IEEE TETC*, 2016.