ALook: Adaptive Lookup for GPGPU Acceleration

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

ABSTRACT

Associative memory in form of look-up table can decrease the energy consumption of GPGPU applications by exploiting data locality and reducing the number redundant computations. State of the art architectures utilize associative memory as static look-up tables. Static designs lack the ability to adapt to applications at runtime, limiting them to small segments of code with high redundancy. In this paper, we propose an adaptive look-up based approach, called ALook, which uses a dynamic update policy to maintain a set of recently used operations in associative memory. ALook updates with values computed by floating point units at runtime to adapt to the workload and matches the stored results to avoid recomputing similar operations. ALook utilizes a novel FPU architecture which accelerates GPU computation by parallelizing the operation look-up process. We test the efficiency of ALook on image processing, general purpose, and machine learning applications by integrating it beside FPUs in an AMD Southern Island GPU. Our evaluation shows that ALook provides 3.6× EDP (Energy Delay Product) and 32.8% performance speedup, compared to an unmodified GPU, for applications accepting less than 5% output error. The proposed ALook architecture improves the GPU performance by 2.0× as compared to state-of-the-art computational reuse methods for the same level of output error.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning approaches**; *Supervised learning*;

KEYWORDS

Approximate computing, GPU, Associative Memory, Computational reuse

ACM Reference Format:
Daniel Peroni, Mohsen Imani, Tajana Rosing. 2019. ALook: Adaptive Lookup for GPGPU Acceleration. In *ASPDAC '19: 24th Asia and South Pacific Design Automation Conference (ASPDAC '19), January 21–24, 2019, Tokyo, Japan.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3287624.3287634

1 INTRODUCTION

In recent years the number of devices making up the Internet of Things (IoT) has rapidly increased. These devices generate massive quantities of data with over 90% produced in the last 2 years [1–3], but lack the performance required to process it in real time. Many IoT devices have limited energy resources available. In some cases, data can be sent to the cloud for processing, but this approach cannot guarantee real time results. Limitations, such as transfer speeds and bandwidth, limit the scalability of cloud-based processing. Reducing the transfer of personal information is an important aspect of ensuring user security and privacy [4–6]. As a result, the computation of large amounts of data needs to be performed locally on a wide variety of devices. Novel architectural designs are necessary to provide good performance while maintaining low power.

Many computing applications, such as multimedia, machine learning, search algorithms, and computer vision, handle large quantities of data and do not require perfect accuracy [7–11]. For example, limitations of human senses mean media applications can allow small visual or audio errors, while machine learning techniques are stochastic in nature and have inherent error [12–17]. Approximate computing involves trading output accuracy for improvements to performance and energy consumption.

Many of these applications involve a large number of redundant computations which can be exploited to save energy by using look-up tables composed of associative memory [18–21]. Commonly computed values are stored in memory rather than recomputing the same result repeatedly. Computational look-up approaches suffer from two main drawbacks. First, static tables can only cover a limited portion of small applications that do not deviate significantly from the profiling dataset. Based on our testing, in the *Sobel* application up to 75% of operations are redundant, however, a large static table with can only avoid recomputing 40% of them. Second, existing computational reuse architectures perform lookup sequentially and are tied to the GPU pipeline. These designs only improve power efficiency, not performance.

In this paper we propose an adaptive computational reuse approach, called ALook, which dynamically updates values stored in associative memory at runtime to improve the energy efficiency of GPGPU applications. To control the energy cost of ALook updates, we reduce the number of writes by only replacing values in ALook intermittently. In addition, to the best of our knowledge, we propose the first GPU-based computational reuse approach which significantly exploits associative search to improve the GPU performance. ALook duplicates the first stage of the floating point unit (FPU) pipeline enabling processing and computational reuse of two inputs simultaneously. Doubling the first stage adds less than 4.5% area overhead to a conventional FPU. Using this architecture, ALook can parallelize the FPU computation, unless ALook cannot match a stored value with either operation.

We examine the proposed techniques for a range of OpenCL GPGPU applications, as well as several machine learning benchmarks from the Rodinia benchmark suite. We test the efficiency of ALook by integrating it beside FPUs in an AMD Southern Island

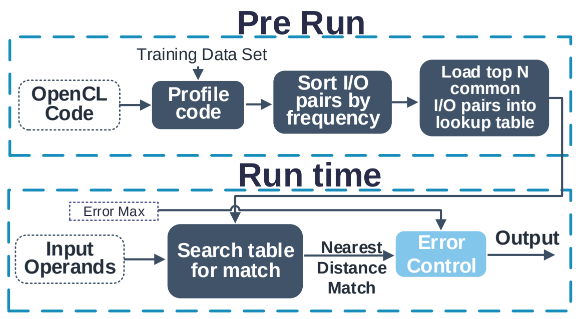
GPU. Our evaluation shows that ALOOK provides 3.6× EDP (Energy Delay Product) and 32.8% performance speedup, compared to an unmodified GPU, for applications accepting less than 5% output error. The proposed ALOOK architecture improves the GPU performance by 2.0× as compared to state-of-the-art computational reuse methods [22, 23] for the same level of output error.

2 RELATED WORK

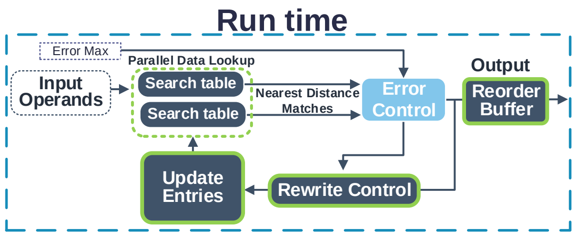
There are many existing strategies for implementing approximate hardware, such as voltage over scaling (VOS), approximate arithmetic units, and approximate memory based accelerators. The voltage over scaling (VOS) involves reducing the voltage of the circuit until logic errors begin to appear [21, 24]. The errors can be profiled and predicted in order to control error. If the voltage is decreased too drastically, critical timings errors begin to appear, potentially causing massive faults in the application. VOS rarely scales gracefully in unoptimized circuits, but careful redesigns can shorten critical paths to reduce error [25–27]. Even with specially designed circuits, VOS has limited scalability and is highly sensitive to process variation. Approximate arithmetic units simplify the hardware used in the computation of operations, such as addition, multiplication, or division. These designs may use simplified building blocks [11, 28, 29], reduced precision [11], or speculative hardware [30]. Error correction and control in approximate computational units require additional steps after computation and causes sizable energy overhead. Application output error is difficult to control as it requires runtime output sampling to adjust the rate of approximation used.

Approximate memory based accelerators store common inputs and output pairs to implement computational reuse. The associative memory searches for the nearest distance value in the table to return as the result for the given inputs [18, 21, 22, 31]. This method is effective at saving power in applications with many identical or similar computations. Associative memory is placed adjacent to FPUs to store commonly occurring data. In these designs, developers must identify and profile key regions of approximable code for common inputs and outputs to load into a static table. In [21], the authors propose a configurable associative memory which relaxes computation by applying VOS to the non-volatile associative memory to trade accuracy for energy savings. Work in [23] designed a novel associative memory based on non-volatile memory which supports searches for nearest absolute distance, rather than Hamming distance similarity. All existing work provide low hit rate on practical applications because their stored values are static and cannot change to adapt to a running application. Work in [22] proposed a method which is capable of updating associative memory through online training. However, this approach suffers from high overhead and energy cost.

We propose ALOOK, a dynamic lookup table capable of improving GPGPU performance and reducing power use. ALOOK does not require code sampling and profiling prior to runtime and reduces energy consumption by searching associative memory for previously computed results. Unlike existing approaches in which GPU performance is bound by pipeline stages, ALOOK is the first design which exploits the redundant GPU computation to improve the computation performance.



(a) Static Work Flow [23]



(b) ALOOK Work Flow

Figure 1: The process flow for a) the static lookup table [23] and b) the dynamic ALOOK.

3 ADAPTIVE LOOKUP DESIGN

Many GPGPU workloads show large amounts of computational data locality with identical or highly similar operations recomputed repeatedly. In these applications, inputs and output pairs for frequently used operations can be stored in associative memory. Inputs for operations are used to search the memory to find the nearest distance match and return the associated output.

We propose ALOOK, a dynamic lookup table capable of adapting to short-term data locality, thus improving GPGPU performance and reducing power use by eliminating redundant computations. Our design provides three major improvements over previous work. First, we eliminate the need to profile the application before runtime for common operations to store in the table. Second, ALOOK updates data entries dynamically to adapt to changes within the application over time. Third, ALOOK uses a duplicated first pipeline stage and reorder buffer to speed up applications, unlike prior designs which only offer energy improvements.

3.1 Lookup Overview

Placing a small lookup table next to each floating point unit enables approximate computational reuse [21, 23]. Figure 1a) shows the pre-run and profiling stage of a static table [23]. During the pre-run stage, the user must flag the section of OpenCL code they wish to approximate. Then, the application is profiled with a training data set and the input and outputs of each arithmetic operation is recorded. Each unique I/O pair is counted and the pairs are sorted by frequency. Prior to a non-profiling run, the top N most common pairs are loaded into the lookup table, where N is the number of entries the table can store. At run-time, the lookup table searches for each operation's input values and returns the corresponding output. The table returns the nearest distance match rather than recomputing the result. The nearest distance match is not guaranteed to be an exact match, so the final application accuracy is managed through error control. Error for a match grows as the difference between the input values and the closest match increases. Error control identifies matches with a large distance from the inputs and assigns them to hardware to compute exact results, while close matches are used directly as output, saving power.

Figure 1b) shows the improvements, in green, ALOOK offers over previous work. ALOOK entirely eliminates the prerun profiling steps required for static designs. The user must still flag code safe for approximation, but no longer needs to sample the application for I/O pairs and identify the most common ones. At run-time ALOOK searches for the nearest distance match to each input within the table. To speed up applications while still allowing accuracy control, two incoming operations are run in parallel. *Parallel data lookup* searches two duplicate tables simultaneously, then a reorder buffer ensures results are output in the order they arrived in the pipeline. The output values associated with the nearest match is used as the result for the operation rather than running on the FPU. Accuracy checks are made on each operation using error control, shown in Figure 1 in light blue, to ensure close matches for operations. ALOOK supports the closest distance search within a user set maximum error. If the input data has distance larger than the specified distance, it will not match any entries. Instead of using poor matches, the results are computed on exact hardware and the newly computed values replace stored entries in the table using an LRU policy. Rewriting on every miss is unnecessary, so *rewrite control* is added improve the energy efficiency of ALOOK. Updating the entries with recently computed values increases the search hit rate, reducing energy consumption and accelerating a wide range of applications.

3.2 Parallel Data Lookup

ALOOK is able to speedup GPGPU applications by eliminating a fraction of redundant computations, unlike existing computational reuse approaches which only reduce GPU power consumption [21]. Although these techniques improve the GPU power efficiency, they cannot exploit the input redundancy for computation speedup. They only improve performance for individual operations within stages of the GPU pipeline, but not the whole application. We use duplicate tables to speed up applications through *parallel data lookup* while still allowing accuracy control.

Figure 2 shows the placement of ALOOK within an AMD Southern Island-based 7970. The GPU has 32 compute units each with 4 SIMD units. The SIMDs have 16 cores for computing arithmetic operations. Two lookup tables are placed alongside each floating point unit to enable parallel lookup. We implement ALOOK with content addressable memory (CAM) introduced in [23]. The lookup table identifies nearest distance matches for each incoming set of input values. When a search provides a poor match for an operation the FPU computes the result exactly and the table updates dynamically using this value.

Figure 3 shows the architecture of an enhanced FPU using a single lookup table to store highly frequent patterns [23]. The table is placed adjacent to the first pipeline stage of the FPU block and search operations occur during this stage. The most similar entry to the inputs and the error distance for each incoming operation are identified. If a close approximate match is found, the output associated with the best match is used as the result and clock gating disables the FPU computation to save power. If a close match is not located in the lookup table the computation continues as normal in the second FPU stage. Lookup table matches do not improve performance as the operations are processed sequentially in the pipeline. Regardless of a hit or miss in ALOOK, the FPU spends a single cycle for each value read from the input buffer. This architecture cannot use associative memory to speed up the GPU process, only reduce energy.

We propose *parallel data lookup*, a novel architecture, which exploits the GPU computational reuse in order to accelerate the computation. Figure 4 shows the integration of parallel lookup tables next to the floating point pipeline stage. ALOOK duplicates the first stage of the FPU and uses a lookup table beside each block. In contrast to conventional FPU which read a single operand from the input buffer, ALOOK processes two operands each cycle. ALOOK reads two inputs and processes them in parallel at the first duplicated FPU block. At the same time, lookup tables next to FPU blocks check for entries closely matching the input operands. Depending on the hit or miss in these two blocks, ALOOK works in one of the following configurations:

- If both input operands hit in the lookup tables ($Hit_1 = 1$ and $Hit_2 = 1$), ALOOK accesses the pre-stored results of computation and clock gates the next FPU stages.
- If input operands hit only in one of the lookup tables (e.g. $Hit_1 = 1$ and $Hit_2 = 0$), ALOOK bypasses the computation in one of the FPUs and exploits the next FPU stages to process the missed input.
- If neither input operands match entries in the lookup tables ($Hit_1 = 0$ and $Hit_2 = 0$), they process serially in the next FPU stages. When this occurs, ALOOK does not read any input operands for a single cycle.

In the first two cases, ALOOK processes two operands at a time, doubling the FPU performance. The percentage of ALOOK time in these configurations depends on the lookup table hit rate (α). This hit rate depends on the number of entries in the lookup table and the level of approximation during the similarity check. Using a larger lookup table or working at a higher approximation level increases the hit rate and the possibility of improving GPU performance. In Section 4.3, we explore parallel lookup efficiency.

3.3 Rewrite Control

Developers can adjust the level of accuracy for individual operations for each section of code, with the default setting only providing exact matches. In this way, even applications with high precision requirements benefit from ALOOK. If the error of the closest match is greater than a user-specified error max, the output is run on exact hardware instead. Error control selects which of the two possible results to output. When a search match exceeds the maximum error, the tables must be updated, adding additional overhead due to write costs. Rewrite control determines when and how often to rewrite data in the table. We show reducing rewrite rate decreases overhead

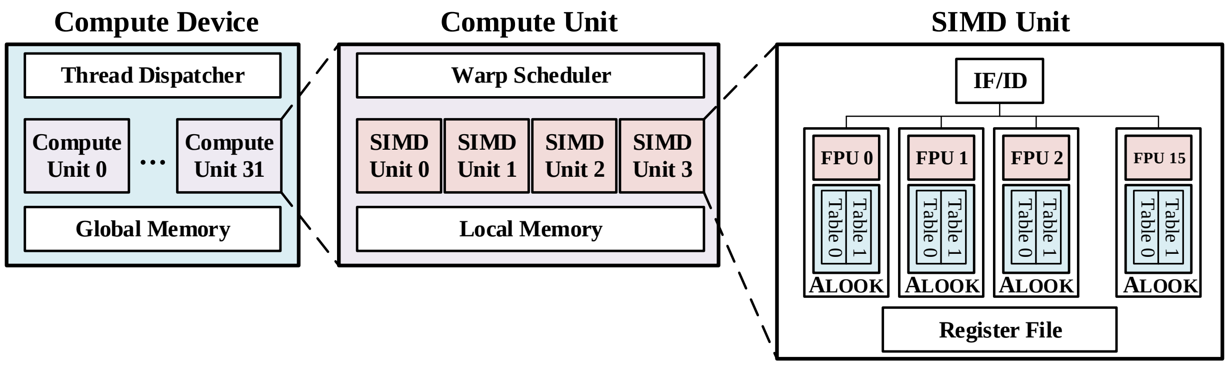


Figure 2: Implementation of ALook alongside FPU within AMD Southern Island Architecture

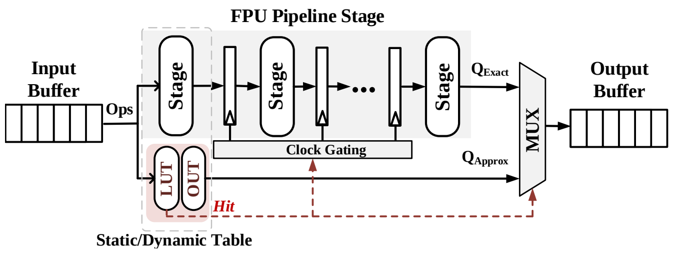


Figure 3: Computational reuse in conventional FPU [23]

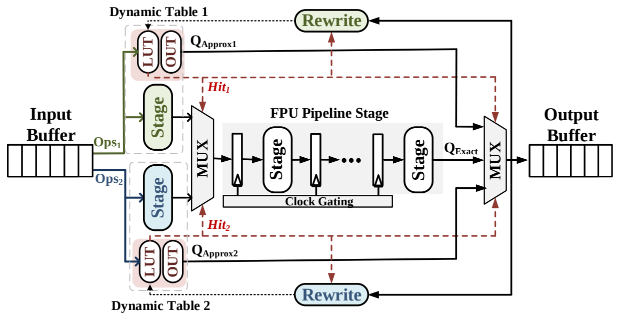


Figure 4: ALook integrates lookup tables next to FPUs

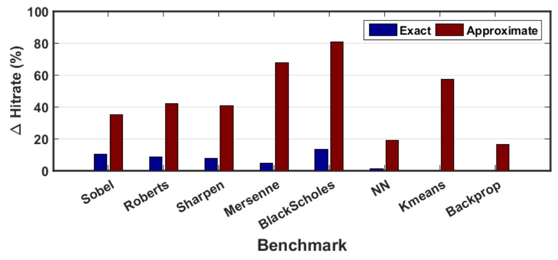


Figure 5: Increase in hitrate for a dynamic table over a static table [23] for exact matches and approximate matches with less than 5% error

without significantly impacting hitrate, resulting in higher energy savings.

We reduce the frequency of ALook rewrites in order to improve energy efficiency. Decreasing the rewrite rate saves energy, but also results in fewer hits. During periods of high hit rates, rewriting on misses can remove common operations prematurely. ALook replaces infrequently used entries with a least recently used (LRU) replacement policy. When a close match is found for given inputs, the counter is set to zero and the other counter values are incremented. When a miss occurs in ALook, the value with the highest count is evicted and replaced with the result computed on exact hardware. To enable LRU policy we use *m* bits, based on the number of entries in the table, to keep track of the values in the dynamic lookup table. When two misses occur during parallel data lookup, only the furthest distance miss is used to update the tables to ensure consistent values.

For a hardware implementation, we used a Content Addressable Memory (CAM) using Non-Volatile Memory (NVM) [32, 33]. In particular, we used memristor CAM with 2 transistors and 2 resistors (2T-2R) for our implementing. Write speeds in CAM memory are slower and consume more energy than reads. When processing code sections with low data locality, the cost of constantly rewriting data becomes high and penalizes energy savings. We decrease the number of writes by only replacing values in ALook once for every 8 misses as determined by our experimental results. In Section 4.4 we examine the trade-off between energy efficiency and update frequency.

A dynamically updating table can handle larger and more varied data sets than static designs. Static profiled data may not accurately represent the runtime data. A training set needs to be varied enough to cover a wide variety of inputs, but this generalization can provide sub-optimal hit rates and accuracy. The profiling data set may not accurately reflect all use cases and contain gaps for some real-world cases. In many workloads, the data locality changes significantly over time. The more distinct regions a data set has, the harder it is for a small static lookup table to cover the changes over time.

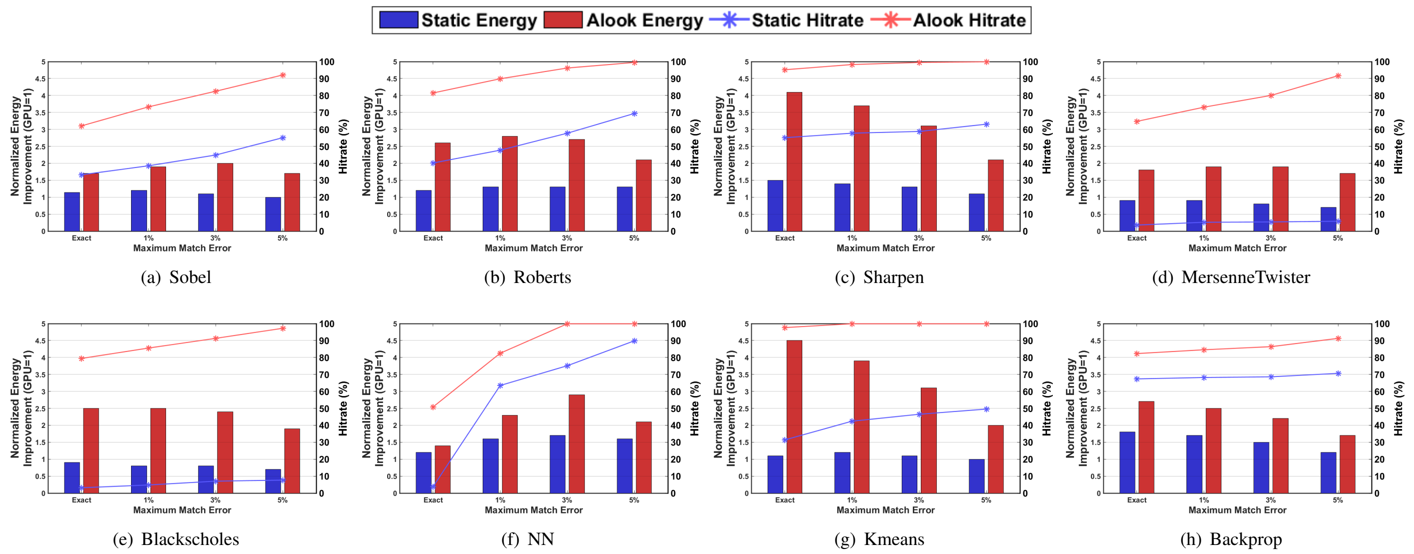


Figure 6: Operation hitrate and energy improvement over unmodified GPU for a static lookup table [23] compared to ALOOK.

4 EXPERIMENTAL RESULTS

4.1 Experimental Setup

The proposed ALOOK is implemented beside each floating point unit on an AMD Southern Island architecture GPU, Radeon HD 7970 device. We use Multi2sim, a cycle accurate CPU-GPU simulator and modify the kernel code for profiling and runtime simulation [34]. We test the design using 8 OpenCL applications: 5 from AMD APP SDK v2.5 [35] and 3 from the Rodinia 3.1 machine learning benchmark suite [36]. From AMD APP SDK we test *Sobel*, *Roberts*, *Sharpen*, *MersenneTwister*, and *BlackScholes*. From Rodinia we test *Knn*, K-means, and *BackProp*. We use the Caltech 101 computer vision dataset [37] for image processing applications. For other applications, we use application specific data sets. For image processing, we define PSNR as the accuracy metric. For general applications working with numbers, the average relative error is defined as the accuracy metric. For each machine learning algorithm, we define a unique accuracy metric to test the impact of approximation. The circuit level associative memory is simulated using HSPICE in 45-nm technology with a 1V supply voltage. We extract frequent patterns of four GPU floating point units; adder (ADD), multiplier (MUL), multiply accumulator (MAC) and square root (SQRT).

4.2 Static vs Dynamic

ALOOK adapts to previously unseen applications by updating the values over time. Figure 5 shows the improvement a dynamic table provides over a same sized static table [23]. The dynamic table updates when it cannot provide a match, replacing old entries with values computed on exact hardware based on an LRU policy. On average the dynamic table improves the hit rate of exact matches for the tested applications by 6% compared to an identical static design. When matches are guaranteed to less than 5% error, the hit rate improvement increases to 44.8%. The adaptability of allows fewer entries to produce the same or better results than static memory, resulting in significantly less area overhead and search power. In our testing, static tables require up to 8× more entries to provide comparable hit rates to ALOOK. The MersenneTwister application has temporal locality but exhibits drastic changes in computational values over its run. As shown in Figure 5, a static design using approximate matching provides 5% hit rate, while ALOOK can approximate 70% of operations. Despite 65% difference in hit rate, our adaptive approach produces less than 10% overall output error.

Figure 6 shows the hit rate and efficiency improvement of a GPU enhanced with a static table [18, 23] compared to ALOOK as the number of entries increases from 16 to 128 rows. We allow up to 5% error on individual matches. Before the application is run, the static table loads N pre-profiled pairs and these values remain fixed for the run duration. ALOOK does not require pre-profiled data, instead of updating entries on search misses allowing it to provide higher hit rates. Based on our results the best average energy improvement occurs when ALOOK has 32 rows, with an average energy improvement of 2.7× for the eight tested applications. The higher activation rate of larger tables is negated by the increased energy needed to search the additional rows. Our evaluation shows that for same sized tables with similar computational accuracy, ALOOK provides an average of 2.1× better energy savings compared to a static table [23].

4.3 Parallel Data Lookup

We examine the ability to search two incoming sets of operations simultaneously. ALOOK is adapted to allow for two parallel search operations. When both operations miss, they are computed in exact hardware in the order they arrived. If one misses and the other hits, one is moved to the reorder buffer and the other is run on an exact FPU. If both find close matches, they are moved to the reorder buffer. Figure 7 shows the lookup table hit rate and the normalized execution time for applications using a 128 entry ALOOK in exact and approximate mode. For an approximation, we select a similarity

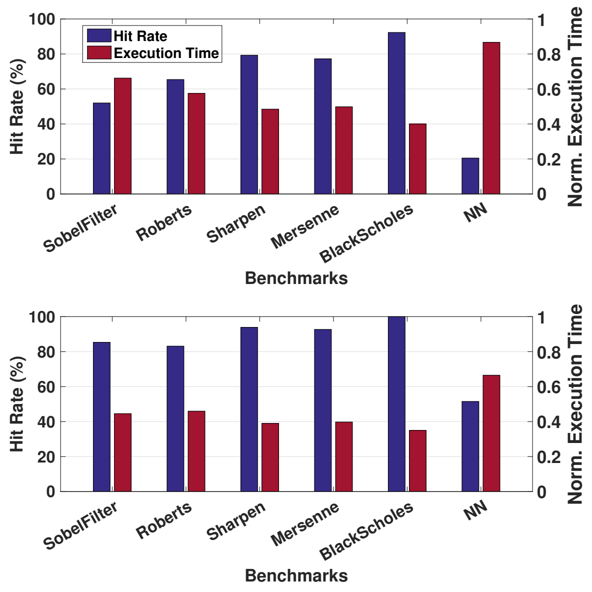


Figure 7: Average hit rate and normalized execution time of ALOOK in exact and approximate modes.

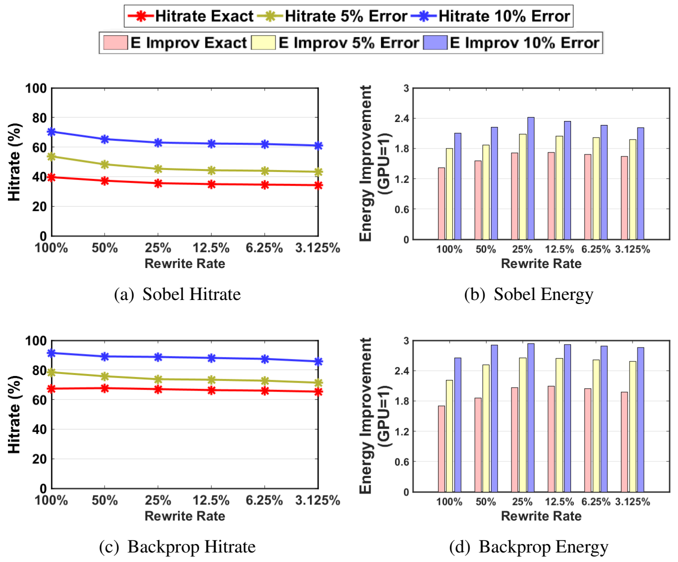


Figure 8: Hitrate and energy improvement as rewrite rate is decreased for Sobel and Backprop applications

metric which ensures less than 5% quality loss. The results show that ALOOK using exact and approximate lookup table can speed up the FPU execution time by 15.3% and 54.2% respectively.

4.4 Variable Rewrite

When a good match is not detected in ALOOK, the value must be computed on exact hardware and written to the CAM. Our design needs to have enough data entries so the effective hit rate overcomes the energy overhead for write operations. Increasing the number

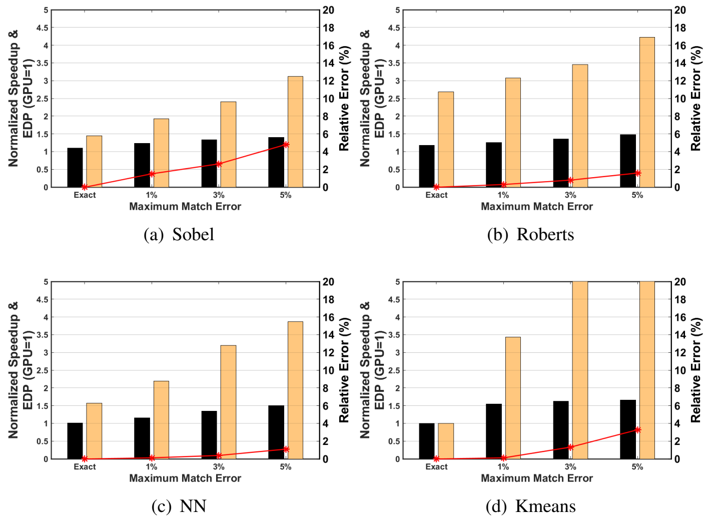


Figure 9: Normalized EDP and performance speedup of ALOOK enhanced GPU for increasing maximum error distances.

of entries shows diminishing returns for hit rate improvement and also raise the energy and time required to search the table. Figure 8 shows ALOOK hit rate and energy saving for decreased rewrite rates. Initially, a write occurs for every search resulting in a poor match. As rewrite rate decreases, the energy savings levels out and then begins to rise slightly. Decreasing rewrites past a point results in the energy savings from decreased matches impacting energy more than saving energy from fewer writes. Based on our results, then it is most efficient to rewrite once for every 8 misses, roughly 12% of misses.

4.5 Accuracy-Energy Trade-off

In this section, we discuss the impact of hardware approximation when trading accuracy for energy savings. In section 3.3 we discuss that ALOOK has the capability to control maximum match distance to control the level of approximation. Accuracy is controlled by setting the maximum acceptable distance of input data to the stored value in ALOOK. If the closest distance match error is more than the specified maximum, the input data is sent to precise FPUs to process. We adjust the maximum error distance to trade energy and accuracy in different applications. Figure 9 shows the energy-delay product (EDP) and performance speedup which each application can achieve when running different applications using an ALOOK with 32 rows. As the error distance increases, more values are computed using ALOOK resulting in better speed up and more energy savings. The overall error is represented by the red line. In the eight tested applications, ALOOK provides 3.6× EDP (Energy Delay Product) and 32.8% performance speedup, compared to an unmodified GPU, with less than 5% output error.

Figure 10 shows the accuracy of the K-means application when the ALOOK is in exact and approximate modes. For clustering algorithms, the accuracy is determined by identifying the number of

Table 1: EDP improvement of ALOOK and other approximate approaches on GPGPU with 5% maximum quality loss.

	ReRAM [21]	*ACAM* [22]	**Alook**		*ReRAM* [21]	*ACAM* [22]	**Alook**
Sobel	1.1×	1.3×	2.6×	***BlackScholes***	0.9×	1.6×	3.8×
Robert	1.2×	1.6×	4.0×	***NN***	1.6×	1.9×	3.5×
Sharpen	1.4×	1.9×	6.3×	***Kmeans***	1.2×	2.6×	5.9×
Merrnse	0.9×	1.3×	2.7×	***Bakcpropag***	1.7×	2.4×	3.5×

incorrectly grouped points. The points wrongly classified by the approximate hardware occur at the boundaries of two clusters.

4.6 Comparison

We compare the efficiency of the ALOOK with two state-of-the-art approaches enabling computational reuse in GPU architecture. First, ReCAM [21], which utilizes a static but configurable table to enable approximation, and second ACAM [22] which uses online learning to fill the lookup table values during runtime. We have implemented both ReCAM and ACAM approaches in their best configurations which results in maximum energy saving. All evaluations have been performed using 32-row table and with 5% maximum quality loss. Table 1 compares the energy-delay product improvement of different designs as compared to conventional GPU architecture. Our evaluation shows that ALOOK can achieve 2.9× and 2.0× higher EDP improvement as compared to ReCAM and ACAM respectively. The higher ALOOK efficiency comes from (i) the low hit rate of the static table in ReCAM and the significant cost of the online learning algorithm to update the lookup table values in ACAM. (ii) ALOOK is capable of speeding up the GPU computation, while ReCAM and ACAM work with the same performance as conventional GPU.

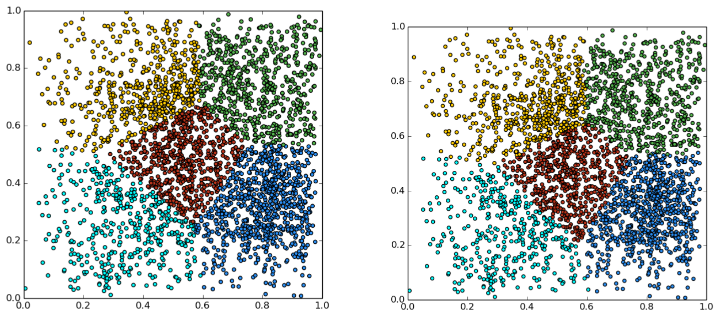
4.7 Overhead

We modify four main floating point units in GPU architecture by duplicating their first stage and adding an associative memory next to them. The tested FPUs utilize deep pipelines with 23 stages, so the duplication of the first stage adds less than 4.5% area overhead to a conventional FPU. In this paper, we exploit a crossbar lookup table that can be integrated at the top of FPUs with minor area overhead. Our evaluation shows that the peripheral circuits which enable the nearest search operation adds an extra 0.3% area overhead to the FPU (for a table with 32 rows).

The ALOOK search operations happen in a single cycle and in parallel with the FPU's computation, thus ALOOK does not add any performance overhead to the GPU. However, a miss in ALOOK adds 5.7% energy overhead to FPU operations, since we still need to pay the cost of FPU to process such data. To ensure no energy overhead of a particular GPGPU application, ALOOK needs to produce a hit rate of 17%. This hit rate is much lower than the numbers we saw in the tested applications.

5 CONCLUSION

Associative memory in form of lookup table can decrease the energy consumption of the parallel processor by exploiting data locality and reducing the number redundant computation. We propose an adaptive associative memory, called ALOOK, which accelerates GPGPU computation by searching for the nearest distance value for incoming FPU operations. ALOOK consists of a small dynamic lookup table which adapts over time to an application. Our evaluation shows that ALOOK provides 3.6× EDP (Energy Delay Product) and 32.8% performance speedup, compared to an unmodified GPU, for applications accepting less than 5% output error. The proposed ALOOK architecture improves the GPU performance by 2.0× as compared to state-of-the-art computational reuse methods for the same level of output error.



(a) Exact (b) Approx

Figure 10: Output quality comparison for *K-means* application running on (a) exact hardware, (b) ALOOK in approximate mode resulting in 2.9% error.

ACKNOWLEDGEMENTS

Authors would like to thank Hamid Nejatollahi for UC Irvine for insightful discussions. 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] "Internet of Things (Iot) Connected Devices Installed Base Worldwide from 2015 to 2025 (in Billions).," in *Statista - The Statistics Portal*, Statista.
[2] C. Ji *et al.*, "Big data processing in cloud computing environments," in *I-SPAN*, pp. 17–23, IEEE, 2012.
[3] V. N. Gudivada, R. Baeza-Yates, and V. V. Raghavan, "Big data: Promises and problems," *Computer*, vol. 48, pp. 20–23, Mar 2015.
[4] M. Zhou *et al.*, "Security and privacy in cloud computing: A survey," in *2010 Sixth International Conference on Semantics, Knowledge and Grids*, pp. 105–112, Nov 2010.
[5] M. Imani *et al.*, "A binary learning framework for hyperdimensional computing," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, IEEE, 2019.
[6] C. Wang *et al.*, "Privacy-preserving public auditing for data storage security in cloud computing," in *2010 Proceedings IEEE INFOCOM*, pp. 1–9, March 2010.
[7] M. Imani *et al.*, "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.
[8] M. Imani *et al.*, "MFBO-SSM: Multi-fidelity Bayesian optimization for fast inference in state-space models," in *AAAI*, 2019.
[9] J. Han *et al.*, "Approximate computing: An emerging paradigm for energy-efficient design," in *IEEE ETS*, pp. 1–6, IEEE, 2013.
[10] M. Imani *et al.*, "Cfpu: Configurable floating point multiplier for energy-efficient computing," in *Proceedings of the 54th Annual Design Automation Conference 2017*, p. 76, ACM, 2017.
[11] S. Hashemi *et al.*, "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.
[12] M. Imani *et al.*, "Bayesian control of large MDPs with unknown dynamics in data-poor environments," in *Advances in Neural Information Processing Systems*, 2018.

[13] Y. Kim *et al.*, "Efficient human activity recognition using hyperdimensional computing," in *Proceedings of the 8th International Conference on the Internet of Things*, p. 38, ACM, 2018.

[14] M. Imani *et al.*, "Rapidnn: In-memory deep neural network acceleration framework," *arXiv preprint arXiv:1806.05794*, 2018.

[15] M. S. Razlighi *et al.*, "Looknn: Neural network with no multiplication," in *Proceedings of the Conference on Design, Automation & Test in Europe*, pp. 1779–1784, IEEE/ACM, 2017.

[16] S. Salamat *et al.*, "Rnsnet: In-memory neural network acceleration using residue number system," in *ICRC*, pp. 1–10, IEEE, 2018.

[17] F. Imani *et al.*, "Nested gaussian process modeling for high-dimensional data imputation in healthcare systems," in *IISE 2018 Conference & Expo, Orlando, FL, May*, pp. 19–22, 2018.

[18] M. Imani *et al.*, "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.

[19] M. Imani *et al.*, "Multi-stage tunable approximate search in resistive associative memory," *IEEE TMSCS*, 2017.

[20] 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.

[21] M. Imani *et al.*, "Resistive configurable associative memory for approximate computing," in *DATE*, pp. 1327–1332, IEEE, 2016.

[22] M. Imani *et al.*, "Acam: Approximate computing based on adaptive associative memory with online learning," in *IEEE/ACM ISLPED*, pp. 162–167, 2016.

[23] M. Imani *et al.*, "Program acceleration using nearest distance associative search," in *Quality Electronic Design (ISQED), 2018 19th International Symposium on*, pp. 43–48, IEEE, 2018.

[24] P. K. Krause *et al.*, "Adaptive voltage over-scaling for resilient applications," in *DATE*, pp. 1–6, IEEE, 2011.

[25] A. B. Kahng *et al.*, "Slack redistribution for graceful degradation under voltage overscaling," in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, ASPDAC '10, (Piscataway, NJ, USA), pp. 825–831, IEEE Press, 2010.

[26] D. Mohapatra *et al.*, "Design of voltage-scalable meta-functions for approximate computing," in *2011 Design, Automation Test in Europe*, pp. 1–6, March 2011.

[27] V. K. Chippa *et al.*, "Scalable effort hardware design," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, pp. 2004–2016, Sept 2014.

[28] C. Liu *et al.*, "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.

[29] 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.

[30] V. Camus *et al.*, "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.

[31] M. Imani *et al.*, "Approximate computing using multiple-access single-charge associative memory," *IEEE TETC*, 2016.

[32] Y. Kim *et al.*, "Orchard: Visual object recognition accelerator based on approximate in-memory processing," in *ICCAD*, pp. 25–32, IEEE, 2017.

[33] S. Gupta *et al.*, "Felix: Fast and energy-efficient logic in memory," in *ICCAD*, pp. 25–32, IEEE, 2018.

[34] R. Ubal *et al.*, "Multi2sim: a simulation framework for cpu-gpu computing," in *PACT*, pp. 335–344, ACM, 2012.

[35] " AMD Accelerated Parallel Processing (APP) Software Development Kit (SDK)." http://developer.amd.com/sdks/amdappsdk/.

[36] 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.

[37] "Caltech Library." http://www.vision.caltech.edu/Image_Datasets/Caltech101/.