SqueezCL: Squeezing OpenCL Kernels for Approximate Computing on Contemporary GPUs

Atieh Lotfi
UC San Diego
alotfi@cs.ucsd.edu

Abbas Rahimi
UC San Diego
abbas@cs.ucsd.edu

Hadi Esmaeilzadeh
Georgia Institute of Technology
hadi@cc.gatech.edu

Rajesh K. Gupta
UC San Diego
gupta@cs.ucsd.edu

ABSTRACT

Approximate computing provides an opportunity for exploiting application characteristics to improve performance of computing systems. However, such opportunity must be balanced against generality of methods and quality guarantees that the system designer can provide to the application developer. Improved parallel processing in graphics processing units (GPUs) provides one such means for data-level parallel applications. We propose SqueezCL a software method to reduce the hardware resources used by an OpenCL kernel. SqueezCL transforms an exact OpenCL kernel to an approximate OpenCL kernel by squeezing dimensions of its data elements. The core of SqueezCL leverages bitwidth reduction to shrink the hardware resources. Selectively reducing the precision and size of data elements generates approximate kernels that can be executed faster at a cost to quality loss. Exploiting this opportunity is particularly important for GPU accelerators that are inherently subject to memory resource constraints. We evaluate SqueezCL on a diverse set of data-level parallel OpenCL benchmarks from the AMD APP SDK v2.9. Experimental result on the AMD Radeon HD 5870 shows that SqueezCL yields on average 1.1× higher performance with less than 10% quality loss without requiring any changes to the underlying GPU hardware.

1. INTRODUCTION

With the diminishing returns from transistor scaling [6] maintaining the current abstraction of near-perfect accuracy in hardware design imposes significant costs in efficiency and performance. Emerging applications including graphics, multimedia, web search, and data analytics offer massive *parallelism* and degrees of tolerance to *approximate computing* to go beyond primarily numerical computations for interacting with sensory interfaces. As such, these present an opportunity for less fully accurate computation results to be considered 'acceptable' for the end user. A programmable parallel architecture, such as those found in GPUs, can jointly exploit these two key application characteristics to improve performance.

Several recent efforts have focused on designing circuits for approximate computing. These circuit techniques focus on optimizing area/energy/performance through imprecise implementations of specific hardware blocks such as adder [8], multiplier [9], and dot product [10]. Others propose algorithms for approximate synthesis through circuit simplification, gate pruning, and voltage overscaling [19, 5, 10]. These techniques trade quality of results for significant gains in performance and efficiency but are fundamentally limited to ASIC designs for a fixed functionality and a predetermined degree of accuracy. The challenge is however exploiting this opportunity with existing toolsets and contemporary programmable parallel architectures. This paper addresses this challenge and makes the following contributions:

(1) We propose SqueezCL, a software method to reduce the hardware resources used by an OpenCL kernel for approximate computing on GPUs. SqueezCL automatically reduces required hardware resources for executing an instance of an OpenCL kernel to achieve higher performance. This required resources is determined by the precision of data elements within the kernel program. Selectively reducing the precision results in approximate kernels that can be executed faster. SqueezCL provides a readily applicable OpenCL kernel transformation that exploits the inherent error tolerance of the emerging applications for higher performance with off-the-shelf GPUs without any changes to their hardware structure.

(2) SqueezCL systematically tunes the precision of data elements in the input OpenCL kernel, subject to a statistical target for quality-of-result. SqueezCL uses a source-to-source compiler that leverages bitwidth reduction optimizations. We devise an optimization algorithm that partitions the kernel's data elements to different sets and assigns various precision levels to each set. The optimization algorithm searches the space of these assignments and finds an optimal assignment that reduces bitdidth while satisfying the quality-of-result target. (3) We evaluate SqueezCL with a diverse set of data-level parallel OpenCL benchmarks selected from the AMD APP SDK v2.9 [2]. Execution of these approximate kernels on an AMD Radeon HD 5870 GPU shows on average 10% higher performance (18% at maximum) with less than 10% loss of quality. Exploiting this opportunity is particularly important for GPUs accelerators that are inherently subject to memory bandwidth constraints.

The rest of the paper is organized as follows. Section 2 surveys prior work in this topic area. Section 3 describes OpenCL execution model. SqueezCL approximation design workflow is presented in Section 4. In Section 5, we present experimental results followed

by conclusion in Section 6.

2. RELATED WORK

A growing body of recent works explore the applicability and significant benefits of approximation computing at different levels of stack. We briefly discuss the most relevant recent works.

Approximate programming languages: EnerJ [17] provides a set of type qualifier to enable programmers to explicitly declare *all* the approximate variables in the program. Rely [4] requires programmers to manually annotate variables and operations as approximate. It allows programmers to define a reliability specification to verify these annotations. There have also been various custom directive extensions to OpenMP that allow developers to specify regions of a program that can be executed approximately [15]. Although these annotations ease the burden of isolating program parts that can be approximated, an automated process is required to identify where and how much approximation can be applied.

Approximate GPU architectures: It has been shown that spatial [12] and temporal [13] instruction reuse can be exploited for approximate computing in GPUs. For certain image processing applications, approximate instruction reuse can be extended up to accepting two Hamming distance mismatches between the operands, but limiting the search space within a tiny associative memory [14]. A software-only technique generates parameterized approximate kernels that benefit mostly from computational reuse available in data-level parallel programs [16].

Approximate circuits: Data precision (i.e., bitwidth) optimizations have been used to balance resources and quality-of-result through introducing custom data types [7, 18]. Circuit techniques also trade area/energy/performance for quality-of-result through imprecise implementations of specific hardware blocks such as adder [8], and multiplier [9]. However, these techniques either target ASIC designs with the fixed functionality, or utilize custom data types and operations that are not supported with the state-of-the-art high-level commercial synthesis tools [1, 3].

In contrast, SqueezCL does not share the the aforementioned limitations. SqueezCL focuses on data precision optimizations with few allowable data types as a special form of approximate computing, and utilizes the saved resources to boost performance on GPUs. SqueezCL: (1) allows seamless integration of exact and imprecise data elements within a kernel to cooperatively working on the same hardware fabric without requiring any modification to the structure of the GPUs; (2) focuses on standard data types used in OpenCL that are fully supported by state-of-the-art toolsets.

3. OPENCL EXECUTION MODEL

Open computing language (OpenCL) is a platform-independent framework for writing programs that execute across a heterogeneous system consisting of multiple compute devices including CPUs or accelerators such as GPUs, DSPs, and FPGAs. OpenCL uses a subset of ISO C99 with added extensions for supporting data and task-based parallel programming models. The programming model in OpenCL comprises of one or more device kernel codes in tandem with the host code. The host code typically runs on a CPU and launches kernels on other compute devices like the GPUs, DSPs, and/or FPGAs through API calls. The instance of an OpenCL kernel or an instance of a single-program multiple-data kernel is called a work-item. Work-items are arranged into work-groups for data-level parallel execution. GPUs handle this parallelism in a SIMD fashion by grouping work-items together to perform the same operation but on their own individual data. A work-item executes on a single processing element. The openCL platform model from the programming model to the framework of the compute devices is illustrated in Fig. 1.

In the following sections, we describe how SqueezCL can reduce the required amount of resources for a kernel and exploit it for increasing performance. SqueezCL systematically reduces precision of data and operations in OpenCL kernels to shrink the resources used during execution, i.e., we transform a complex kernel that requires expensive memory accesses and computations to a simpler kernel that produces *approximate* results.

4. SqueezCL: SQUEEZING OPENCL KERNELS

SqueezCL supports a cross compiler to generate approximate kernels via source-to-source OpenCL kernel transformation. The transformation algorithm automatically detects and simplifies parts of the kernel code that can be executed with reduced precision while preserving the desired quality-of-result. To achieve this goal, SqueezCL takes in as inputs, an *exact* OpenCL kernel and a metric for measuring the quality-of-result target. SqueezCL compiler investigates the exact kernel code and detects data elements, i.e., OpenCL kernel variables, that provide possible opportunities for increased performance in exchange of accuracy. SqueezCL then automatically generates a set of approximate kernels that produce acceptable results. These approximate kernels provide improved performance benefits by reducing resources used by kernel. SqueezCL outputs an optimized approximate kernel with the most performance whose output quality satisfies the quality-of-result target. Fig. 2 illustrates an overview of our workflow.

A brute-force methodology for exploring the approximate kernels generates an approximate kernel for every possible combination of the variable types. Considering four variable types in a kernel with $|V|$ number of variables, a total number of $4^{|V|}$ approximate kernels would be generated where in each version every variable is replaced by another type. This results in an exponentially growing design space for the approximate kernels, intractable to search. To reduce this space, SqueezCL uses a grouping algorithm to detect and bundle dependent variables together for the precision tuning. SqueezCL further applies a screening process to prune those groups of the variables that are not amenable to approximation. The grouping and pruning algorithm along with profiling make the precision tuning feasible for large kernels.

4.1 Grouping and Pruning Algorithm

Algorithm 1 first groups dependent variables into candidate groups (CG). Variables with data dependency, or those that are used

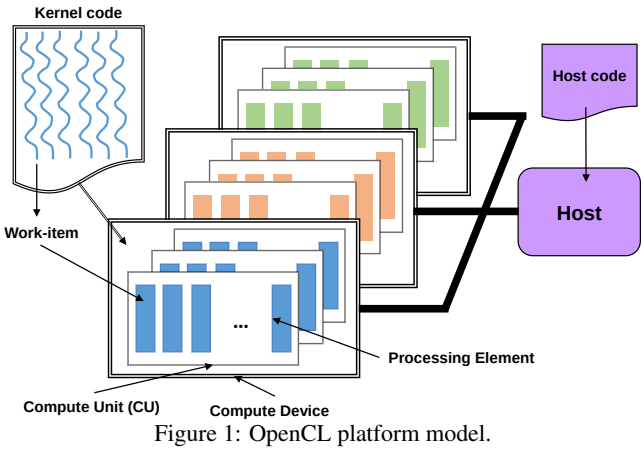

Figure 1: OpenCL platform model.

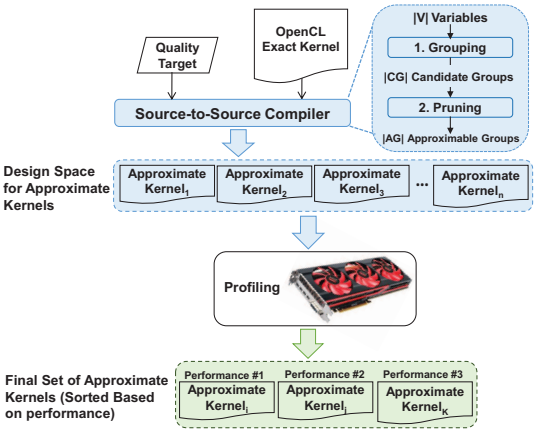

Figure 2: Overview of SqueezCL, our approximation design workflow.

together to compute a partial result for the final output reside in the same group. The algorithm detects these dependent variables and merges their corresponding groups together as shown from Line 1 to 10 in Algorithm 1. Finally, the grouping algorithm partitions |V| variables to |CG| groups that their precision can be tuned independently. Fig. 3 shows a code snippet for clarification of these dependencies in r-gaussian kernel. In this region of kernel code, there are 5 `float` variables where `xc` (`yc`) holds the current value of input (output) and `xp` (`yp`, `yb`) holds the value in the previous level(s). This dependency results in a |CG| of 2 as shown in Fig. 4. Due to this dependency using a more accurate type for the other variables cannot bring higher quality if the accuracy of any of these variables is reduced. To ensure the safety of execution, the algorithm never applies approximation on the variables that affect control flow conditions.

Once the grouping algorithm generates the candidate groups, a pruning algorithm determines a subset of these groups that are amenable for reducing the precision. We assign a precision tag (PT) value to each variable type of `char`, `short`, `int`, and `float` in an increasing order.This type conversion follows mapping of built-in data types on a semi-lattice with ordering based on cost of implementation (e.g., bit width). The higher the PT value, the higher the computational accuracy, and the higher resource utilization and execution time. The pruning algorithm selects a group from the CG set without backtracking and generates an approximate kernel by reducing precision and estimating its effect on quality as an acceptance criterion. The pruning algorithm reduces the original precision of the selected group by one level (a ΔPT demoting). The algorithm then performs a test to dynamically measure the sensitivity of the group to the quality loss. This test determines whether the precision of the selected group can be altered or not. If the output quality is less than the quality-of-result target, the precision of the selected group cannot be reduced since it is not tolerant to a ΔPT demoting. Consequently, the group is eliminated from the candidate list of the groups for approximation. The pruning algorithm continues this screening process for the next candidate group. This process is executed |CG| times to determine approximable groups (AG) among the candidate groups (Line 11–18 in Algorithm 1). The immutability test of the groups is done with the help of profiling feedback on a GPU described in Section 4.2. Finally, the approximable variables are merged into |AG| groups that significantly reduces the search space from $4^{|V|}$ to $4^{|AG|}$ where AG $\subset$ CG $\ll$ V.

Algorithm 1 Grouping and Pruning Algorithm

```
1:  function GROUP&PRUNE(ExactKernel)
2:      F = {All float variables in ExactKernel}
3:      for all variables v_i in F do
4:          Group_i = {v_i}
5:      end for
6:      for all variables (v_i, v_j) in F do
7:          if v_i and v_j are dependent then
8:              merge Group_i and Group_j
9:          end if
10:     end for
11:     for each Group_i do
12:         generate kernel_i s.t. ∀ v_j ∈ Group_i ← ΔPT demoting
13:         run kernel_i on GPU
14:         if (Quality (kernel_i) < Quality Target) then
15:             eliminate Group_i
16:         end if
17:     end for
18: end function
```

4.2 Profiling for Approximation

At this point, SqueezCL can automatically populate the approximate kernels by assigning PT values to the approximable groups. Each approximable group can have a PT value ranges from its original precision level of maximum four, to the last level of one. The precision of the variables and associated operations in a group is inferred from the assigned PT value. This leads to a design space with a maximum of $4^{|AG|}$ approximate kernels. The goal of profiling is to rule out some of these generated approximate kernels that can not meet the quality-of-result target. This quality measurement test is done by executing an approximate kernel and comparing its output with the exact kernel output on a representative data input set. If the output of the approximate kernel cannot satisfy the quality-of-result target, the approximate kernel is ruled out from the final set of kernels. Otherwise, it is selected as one of the promising kernels in the final set as shown in Fig. 2.

The output of this profiling is the final set of approximate kernels that can meet the quality-of-result target. These approximate kernels are sorted based on their execution time and the one with

```
float4 xp, yp; // previous input and output
float4 yb; // previous output by 2
for (int y = 0; y < height; y++){
    float4 xc = (float4)(input[pos].x, ...);
    float4 yc = (a0 * yb) + (a1 * yp) - ... ;
    output[pos] = (uchar4)(yc.x,...);
    xp = xc;    yb = yp;    yp = yc;
}
```

Figure 3: Code snippet for exact r-gaussian kernel

```
short4 xp, xc;
float4 yp, yb, yc;
for (int y = 0; y < height; y++){
        xc = (short4)(input[pos].x, ...);
        yc = (a0 * yb) + (a1 * yp) - ...;
        output[pos] = (uchar4)(yc.x, ...);
        xp = xc;    yb = yp;    yp = yc;
}
```

Figure 4: Code snippet for approximate r-gaussian kernel

the highest performance is selected.

5. EXPERIMENTAL RESULTS

We focus on a diverse set of application domains, including image processing (recursive gaussian, sobel), signal processing (convolution, dct), and physical simulation (n-body). These benchmarks are selected from the AMD accelerated parallel processing (APP) SDK v2.9 [2]. That is, a complete development platform created by the AMD to leverage accelerated compute using OpenCL. SqueezCL source-to-source compiler takes in the exact OpenCL kernel and the quality-of-result target. Each exact kernel is then converted to a set of approximate versions using the compiler transformations; some of these kernels are ruled out by the quality checking using the pruning and profiling algorithms. SqueezCL utilizes the AMD Evergreen Radeon HD 5870 GPU device, and generates the optimized approximate kernel with the highest performance that consumes less resources than the exact kernel.

Table 1 summarizes SqueezCL design space and profiling results for different benchmarks. The row *# Possible Kernels* indicates the number of approximate kernels that could be generated by reducing the precision of all variable types. The row *# Kernels after Grouping* shows the reduced number of kernels in the design space after applying the grouping and pruning algorithm. This design space is further shrunk by the profiling process that performs a quick screening to drop those kernels with an unacceptable quality results as shown in row *# Kernels after Profiling*. The row *GPU Profiling Time (ms)* shows the average time for the Radeon HD 5870 GPU device to run each version of the approximate kernel.

As shown in Table 1, the grouping and pruning algorithm considerably reduce the number of kernels that are generated by SqueezCL. For example, for r-gaussian kernel, this number is reduced from 4^{12} kernels to 1024 which saves time and space by more than three order of magnitude.

Table 1: Effect of grouping and profiling on size of design space for the approximate kernels.

	r-gaussian	**sobel**	**n-body**	**dct**	**conv**
# Possible Kernels	4^{12}	4^9	4^{17}	4×3^7	4×3^{14}
# Kernels after Grouping	1024	81	4096	324	8748
# Kernels after Profiling	36	27	1024	81	972
GPU Profiling Time (ms)	2.70	0.92	0.66	0.25	0.01

We use average relative error as a metric to measure the quality loss [11]. This metric measures the accuracy between the output elements of the exact kernel and the approximate kernel. We set the quality loss target to a maximum of 10% which is conservatively aligned with other work on quality trade-offs [11, 8, 9].

Fig. 5 shows the quality loss vs performance speedup (normalized to the exact kernel) achieved for different kernel configurations generated by SqueezCL for r-gaussian benchmark. The leftmost configuration shows the exact kernel (with speedup of 1 and no quality loss) and the rightmost configuration shows the results for the most relaxed kernel (which has the best speedup but has the most quality loss). The minimum, maximum, and average quality loss for a diverse set of input patterns is shown for each configuration. As seen, although the last two configurations with higher speedups pass most of test experiments, they can not guarantee the quality-of-result for all input patterns. Therefore, the kernel with speedup 1.12 that has acceptable quality loss is selected as the best kernel.

Fig. 6 summarizes the performance speedup for executing the best approximate kernels on the GPU, normalized to the exact kernel execution. The best approximate kernel exhibits a maximum speedup of 18% for the n-body application on GPU. On average, SqueezCL could achieve 10% performance speedup for these kernels.

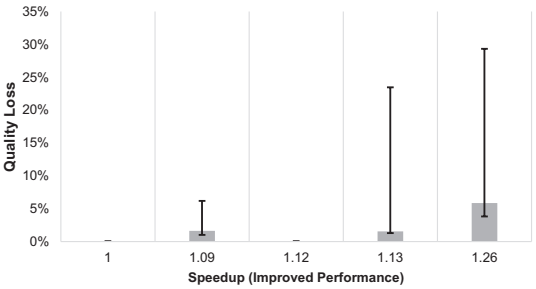

Figure 5: Quality loss vs speedup for the exact and 4 generated approximate kernels for r-gaussian

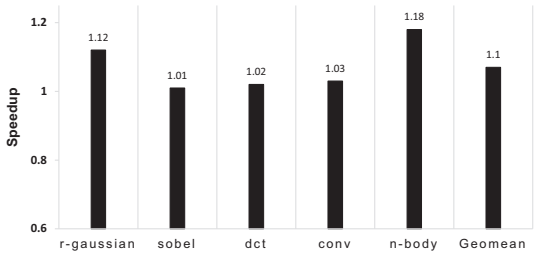

Figure 6: Speedup with SqueezCL on 8570 GPU.

Figure 7: Quality loss with SqueezCL.

We verify the output quality of the optimized approximate kernel with different test input patterns, other than the representative input set used for the kernel profiling. Fig. 7 shows the minimum, maximum, and average quality loss measurements for these test inputs. As shown, the maximum quality loss is always below 1% for all the test inputs, therefore satisfying the quality-of-result target.

6. CONCLUSION

In this work, we propose a software method to boost performance and reduce hardware resources used by an OpenCL kernel on GPUs. To this end, we devise SqueezCL that systematically transforms an OpenCL kernel to an approximate version by selectively reducing the precision of kernel's data and operations. It allows seamless integration of exact and imprecise data elements within a kernel to cooperatively working on the same hardware fabric without requiring any modification to the structure of the GPUs. We evaluate SqueezCL on a diverse set of data-level parallel OpenCL benchmarks from the AMD APP SDK v2.9. Experimental result on the AMD Radeon HD 5870 shows that SqueezCL yields $1.01\times$–$1.18\times$ higher performance with less than 10% quality loss without requiring any changes to the underlying GPU hardware.

7. REFERENCES

[1] Altera sdk for opencl. http://www.altera.com/products/software/opencl/opencl-index.html.

[2] Amd app sdk v2.9. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/.

[3] Sdaccel. http://www.xilinx.com/products/design-tools/sdx/sdaccel.html.

[4] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 33–52, New York, NY, USA, 2013. ACM.

[5] L. N. Chakrapani, P. Korkmaz, B. E. S. Akgul, and K. V. Palem. Probabilistic system-on-a-chip architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3):29:1–29:28, May 2008.

[6] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[7] A. Gaffar, J. Clarke, and G. Constantinides. Powerbit - power aware arithmetic bit-width optimization. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 289–292, Dec 2006.

[8] A. Kahng and S. Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 820–825, June 2012.

[9] 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*, pages 346–351, Jan 2011.

[10] D. Mohapatra, V. Chippa, A. Raghunathan, and K. Roy. Design of voltage-scalable meta-functions for approximate computing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, March 2011.

[11] 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*, pages 603–614, Feb 2015.

[12] A. Rahimi, L. Benini, and R. K. Gupta. Spatial memoization: Concurrent instruction reuse to correct timing errors in simd architectures. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 60(12):847–851, Dec 2013.

[13] A. Rahimi, L. Benini, and R. K. Gupta. Temporal memoization for energy-efficient timing error recovery in gpgpus. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–6, March 2014.

[14] A. Rahimi, A. Ghofrani, K.-T. Cheng, L. Benini, and R. K. Gupta. Approximate associative memristive memory for energy-efficient gpus. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 1497–1502, 2015.

[15] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. A variability-aware openmp environment for efficient execution of accuracy-configurable computation on shared-fpu processor clusters. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, Sept 2013.

[16] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 35–50, New York, NY, USA, 2014. ACM.

[17] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 164–174, New York, NY, USA, 2011. ACM.

[18] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 53–64, New York, NY, USA, 2014. ACM.

[19] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. Salsa: Systematic logic synthesis of approximate circuits. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 796–801, June 2012.