GRATER: An Approximation Workflow for Exploiting Data-Level Parallelism in FPGA Acceleration

Atieh Lotfi*, Abbas Rahimi†, Amir Yazdanbakhsh‡, Hadi Esmaeilzadeh‡, Rajesh K. Gupta*
*UC San Diego †UC Berkeley ‡Georgia Institute of Technology

***Abstract*—Modern applications including graphics, multimedia, web search, and data analytics not only can benefit from acceleration, but also exhibit significant degrees of tolerance to imprecise computation. This amenability to approximation provides an opportunity to trade quality of the results for higher performance and better resource utilization. Exploiting this opportunity is particularly important for FPGA accelerators that are inherently subject to many resource constraints. To better utilize the FPGA resources, we devise, GRATER, an automated design workflow for FPGA accelerators that leverages imprecise computation to increase data-level parallelism and achieve higher computational throughput. The core of our workflow is a source-to-source compiler that takes in an input kernel and applies a novel optimization technique that selectively reduces the precision of kernel's data and operations. By selectively reducing the precision of the data and operation, the required area to synthesize the kernels on the FPGA decreases allowing to integrate a larger number of operations and *parallel* kernels in the fixed area of the FPGA. The larger number of integrated kernels provides more hardware context to better exploit data-level parallelism in the target applications. To effectively explore the possible design space of approximate kernels, we exploit a genetic algorithm to find a subset of safe-to-approximate operations and data elements and then tune their precision levels until the desired output quality is achieved. GRATER exploits a fully software technique and does not require any changes to the underlying FPGA hardware. We evaluate GRATER on a diverse set of data-intensive OpenCL benchmarks from the AMD SDK. The synthesis result on a modern Altera FPGA shows that our approximation workflow yields 1.4×–3.0× higher throughput with less than 1% quality loss.**

I. INTRODUCTION

Before the effective end of Dennard scaling, we were able to improve all three of performance, efficiency, and generality. With the end of Dennard scaling, the community is facing an iron triangle. We can only improve any two of the performance, efficiency, and generality at the expense of the third. Solutions that provide significant performance and efficiency gains while retaining as much generality as possible, are highly desirable. One promising approach is the use of programmable accelerators, such as, FPGAs to achieve higher performance and efficiency in certain application domains. Approximate computing is another promising approach that leverages tolerance of applications to imprecision and trades small losses of quality for gains in performance and efficiency [11, 12, 14, 19, 20].

The confluence of these two trends can potentially yield significant performance and efficiency gains since many of the applications that can benefit from the FPGA accelerations are amenable to approximation. However, there is a lack of techniques that exploits this opportunity. This work aims to bridge the gap between approximation and the FPGA acceleration through an automated design workflow. Altera and Xilinx recently offer high-level acceleration frameworks for OpenCL [1, 4], hence in our work we target acceleration of data-intensive computational OpenCL applications. The challenge is however devising a workflow that can be plugged into the existing toolsets and can automatically identify the opportunities for approximation while keeping the quality loss reasonably low. This paper addresses this challenge and makes the following contributions:

(1) We propose GRATER, a design workflow that automatically leverages approximation to provide more opportunities for the FPGA accelerators to utilize data-level parallelism and achieve higher throughput. GRATER automatically reduces required hardware area for synthesizing an instance of an OpenCL kernel. This required area is determined by the precision of data and operations which is specified by the kernel program. By selectively reducing the precision, a larger number of *parallel* approximate kernels can be mapped in the fixed area budget of an FPGA. GRATER provides a readily applicable workflow that exploits the inherent error tolerance of the emerging applications for higher computational throughput with off-the-shelf FPGAs without any changes to their hardware structure.

(2) GRATER systematically tunes the precision of the operations and data in the input OpenCL kernel, subject to a statistical target for quality-of-result. GRATER uses a source-to-source compiler that leverages an automated transformation to selectively reduce the precision. The precision of the data and operations are automatically inferred from the precision of their operands. We devise a genetic programming-based optimization algorithm that assigns various precision levels to different data and operations in the kernel. We use genetic programming to evolve kernel variants until one is found with optimal assignments that reduces synthesized kernel area while stochastically satisfying the quality-of-result target.

(3) We evaluate GRATER with a diverse set of data-intensive OpenCL benchmarks selected from the AMD APP SDK v2.9 [2]. The synthesis result on an Altera Stratix V FPGA shows that the reduced area of the transformed approximate kernels yields 1.4×–3.0× higher throughput with less than 1% loss of quality.

The rest of the paper is organized as follows. Section II surveys prior work in this topic area. Section III describes acceleration of OpenCL applications on the FPGAs. GRATER approximation design workflow is presented in Section IV. In Section V, we present experimental results, followed by conclusion in Section VI.

II. Related Work

SNNAP [14] focuses only on the FPGA acceleration of neural networks that mimic regions of codes that are amenable to approximation. Data precision and word-length (i.e., bit width) optimizations have been used to balance resources and quality-of-result through introducing custom data types [6, 7, 9, 10, 17, 18]. ASAC [16] identifies approximable variables in the program using a statistical sampling method for analyzing the sensitivity of the program output to perturbation in intermediate data. A synthesis method for generating approximate circuits from RTL code is proposed in [15]. It uses a greedy approach to generate approximate versions, and selects the near-optimal design after simulating and synthesizing all design variants. Circuit techniques also trade area/energy/performance for quality-of-result through imprecise implementations of specific hardware blocks such as adder, and multiplier [11, 12]. 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, 4].

In contrast, GRATER does not share the aforementioned limitations. GRATER focuses on high-level FPGA-specific pre-synthesis optimization. This optimization reduces the FPGA resource utilization when the kernel is synthesized. Performing the optimization in the pre-syntesis step enables GRATER to efficiently explore the design space of the kernel while abstracting away the details of the post-synthesis circuit. The area saving from GRATER allows integrating more kernels in the fixed-area budget of the FPGA. The resulting better utilization of the FPGA resources leads to higher data-level parallelism and system throughput. GRATER: (1) allows seamless integration of exact and imprecise data elements within a software-level kernel to cooperatively work on the same hardware fabric without requiring any modification to the structure of the FPGAs; (2) focuses on standard data types used in OpenCL that are fully supported by the high-level synthesis tools; (3) takes advantage of the portability of OpenCL code: kernel quality measurements are accelerated on the GPUs, and then the selected optimized kernel is mapped on the FPGAs; (4) deploys a simple model for estimating resource utilization on FPGA based on the complexity of operations to avoid synthesizing every possible approximate kernel.

III. OpenCL Execution Model

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 is called a work-item. These kernels execute on compute devices that are a set of compute units (CUs), each comprising of multiple processing elements having ALUs. The work-items execute on a single processing element and exercise the ALU. The OpenCL platform model from the programming model to the framework of the compute devices is illustrated in Fig. 1.

A. Mapping OpenCL Programs on FPGAs

The Altera OpenCL SDK [1] allows programmers to use high-level OpenCL kernels, written for GPUs, to generate an FPGA design with higher performance per Watt [5]. An OpenCL kernel is first compiled and then synthesized as a special dedicated hardware for mapping on an FPGA. However, GPUs and FPGAs exploit data-level parallelism differently, which leads to disparate benefit in terms of performance per watt. GPUs are single-instruction multiple-data (SIMD) devices that exploit data-level parallelism: they group processing elements in a CU to perform the same operation but on their own individual data. On the other hand, FPGAs exploit pipeline parallelism in a CU where different stages of the instructions are applied to different work-items concurrently leading to a higher performance per Watt.

FPGAs can further improve the performance benefits by creating multiple copies of the kernel pipelines (synthesized version of an OpenCL kernel).[1] For instance, this replication process can make *N* copies of the kernel pipeline. As the kernel pipelines can be executed independently from one another, the performance would scale linearly with the number of copies created owing to the data-level parallelism model supported by OpenCL. In the following sections, we describe how GRATER can reduce the amount of resources for a kernel pipeline to save area and exploit remaining area resources to boost performance by replication. GRATER systematically reduces the precision of data and operations in OpenCL kernels to shrink the resources used per kernel pipeline by transforming complex kernels to simple kernels that produce *approximate* results.

IV. GRATER: Approximation Design Workflow

GRATER supports a source-to-source 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, GRATER takes in as

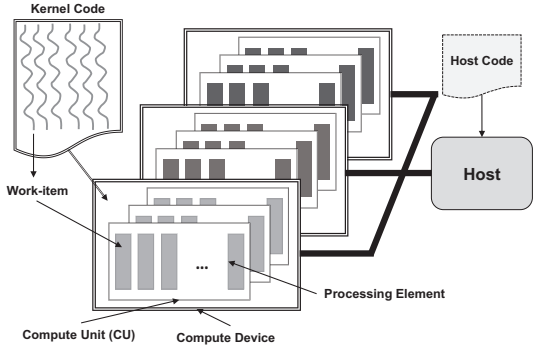

Fig. 1: OpenCL platform model.

[1]Replication is handled in Altera OpenCL by setting `num_compute_units` as a kernel attribute.

inputs, an *exact* OpenCL kernel, a set of input test cases, and a metric for measuring the quality-of-result target. GRATER 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. GRATER then automatically generates a set of approximate kernels that produce acceptable results. These approximate kernels provide improved performance benefits by reducing the area when implemented on the FPGAs. GRATER outputs an optimized approximate kernel with the least area whose output quality satisfies the quality-of-result target. Fig. 2 illustrates an overview of our workflow.

GRATER uses the precision of the operations and data to tune performance as a tradeoff against precision. The transformation investigates a set of kernels where in each version, some of these potential variables are replaced with a less accurate variable[2]. We assign a precision tag (PT) to each variable type. For example, a kernel with data types ranging from floating point to char has four levels of complexity: $\{4, 3, 2, 1\}$ are assigned to {`float`, `int`, `short`, `char`} respectively. The higher the PT, the higher the accuracy requirements, and the higher resource consumption. A brute-force methodology for exploring the approximate kernels is to generate an approximate kernel for every possible combination of the variable types. For instance, for a kernel with $|V|$ number of `float` variables, a total number of $4^{|V|}$ kernels would be generated where in each version every `float` variable is replaced by different PTs. This results in an exponentially growing design space intractable to search. To avoid this huge design space exploration, we devise an algorithm that first detects those variables that are amenable to approximation and then applies a genetic-based algorithm to approximate the kernel. We discuss the details of our algorithm in the following subsections.

A. Analysis and Pruning

In the first step, GRATER detects variables in the code that are amenable to approximation. To do so, a separate kernel is generated for testing the amenability of every variable. In each kernel, the precision of one variable is demoted by one level (a ΔPT demoting), while other variables have their exact precision, to measure the *significance* of a small precision loss of a variable on the quality of result. This test determines whether the precision of the selected variable can be reduced or not. If the output quality is less than the desired output quality, GRATER excludes this variable from the set of safe-to-approximate variables and does not modify its precision.[3] Consequently, the variable is eliminated from the candidate list of variables for approximation. The pruning algorithm continues the screening process for all the variables in the code (Line 4–10 in Algorithm 1). The pruning algorithm is executed $|V|$ times to determine approximable variables (AV).

[2]We limit the space of our optimization search across the available variable types in OpenCL, as opposed to within a type itself [17], due to the nature of a source-to-source transformer that requires to work at the same level of abstraction of the input programming language. GRATER enables Altera OpenCL synthesis tool chain to benefit from this source-to-source translation by generating standard OpenCL approximate kernels.

[3]GRATER also enables the programmer to annotate critical variables as non-approximable, so that the transcompiler would not change their precision.

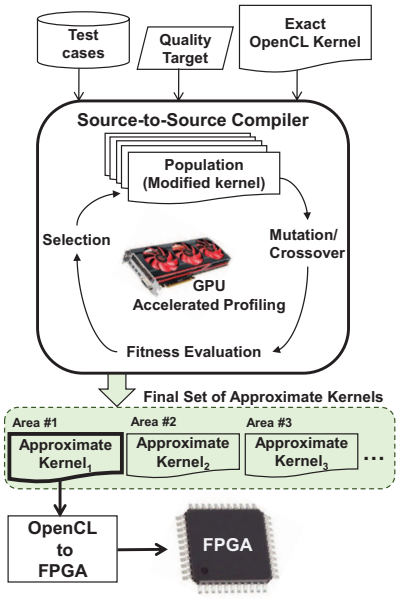

Fig. 2: Overview of GRATER, our approximation design workflow.

This sensitivity test is done with the help of profiling feedback that is accelerated on a GPU.

GRATER then finds the lowest possible precision for each variable in AV (Line 11–14 in Algorithm 1). It generates an approximate kernel for every variable in AV, where in each kernel, one variable type is replaced by the lowest possible precision (e.g. `char`) while other variables preserve their exact precision (EP) that originally have in the exact code. If the quality of the generated kernel is less than the desired output quality, then that tentative lowest precision is promoted by one level and the same quality check is repeated. This process is continued until the lower precision bound for each variable is found. At this point, PT value ranges for each approximable variable is extracted (from EP to LP).

After finding the lower precision bound for all variables in AV, another approximate kernel is generated in which all approximable variables get their lowest possible precision (LP values) found in the previous step. If this kernel meets the quality-of-result target, the solution is found (Line 15–19 in Algorithm 1). Otherwise, a genetic algorithm, described in the following, is run to find the approximate kernel.

B. Genetic-based Approximation Algorithm

Genetic algorithm is a powerful stochastic search method which is deployed to find a good solution from a large search space [8]. To operate with a genetic algorithm, we need to take into account the following components: 1) a genetic representation of solutions in a form that can be interpreted as a chromosome, 2) an initial population, 3) a fitness function which gives an evaluation of the desirability of each chromosome, and 4) genetic operators that change the composition of new generation during reproduction and a selection operator for choosing the survivors.

1) Genetic Representation of Chromosomes

We represent each individual as an array of precision tags with the length of AV list. Each gene in this representation

Algorithm 1 pseudo-code for the GRATER

```
1: function PRUNE&RELAX(ExactKernel, QualityTarget, inputSet)
2:     V = {All candidate variables in ExactKernel}        AV = {}
3:     TopPop = {}        cInput = input_0 from inputSet
4:     for all variables v_i in V do
5:         generate kernel_i s.t. v_i ← ΔPT demoting
6:         run kernel_i with cInput on GPU
7:         if (Quality(kernel_i) ≥ QualityTarget) then
8:             AV = AV ∪ v_i
9:         end if
10:    end for
11:    for all variables v_i in AV do
12:        LP_i = FindLowerPT(v_i, cInput)
13:        EP_i = getExactPT(v_i, cInput)
14:    end for
15:    generate kernel_min s.t. ∀ v_i ← LP_i
16:    run kernel_min with cInput on GPU
17:    if (Quality (kernel_min) ≥ QualityTarget) then
18:        ApproxKernel = kernel_min
19:    else
20:        ApproxKernel, TopPop =
21:            GA(ExactKernel, LP, EP, cInput)
22:    end if
23:    for all input_i in the training inputSet do
24:        run ApproxKernel with input_i on GPU
25:        if (Quality(ApproxKernel) < QualityTarget) then
26:            NeedToChangeSolution = True
27:            for all kernel_j in TopPop do
28:                run kernel_j with input_i on GPU
29:                if (Quality(kernel_j) > QualityTarget) then
30:                    ApproxKernel = kernel_j
31:                    NeedToChangeSolution = False
32:                    Break
33:                end if
34:            end for
35:            if (NeedToChangeSolution) then
36:                cInput = input_i
37:                Goto line 11
38:            end if
39:        end if
40:    end for
41:    return ApproxKernel
42: end function
```

shows the PT of each variable in AV. Every individual can be easily translated to a candidate approximate kernel. The precision of the variables and associated operations in the approximate kernel is inferred from the assigned PT value in the chromosome.

2) Population

The initial population is randomly generated. Each approximable variable can have a PT value range with different levels of complexity, started from its lowest precision bound to its exact precision level (LP and EP in Algorithm 1).

All individuals in the population should meet the desired quality-of-result requirement. This can be verified either by executing the kernel or comparing its PT values with the least precision chromosome found. The least precision chromosome found in the population is the one that the PT values of every gene in its chromosome is lower than the PT values of corresponding genes in all other chromosomes. If such a chromosome does not exist in the population, the least precision PT in the population would be the same as LP. In this case, for all generated kernels we need a kernel execution for accuracy check. When the quality measurement test is done by executing an approximate kernel, its output is compared with the exact kernel output on a representative data input. If the output of the approximate kernel cannot satisfy the quality-of-result target, the approximate kernel is ruled out from the population. Otherwise, it is considered as one of the candidates for the next generation. This kernel profiling and execution process is accelerated on a GPU. This is accomplished by decoupling the quality loss analysis and the approximate kernel mapping thanks to the platform-independent nature of OpenCL. To increase the speed of genetic algorithm, before creating and executing each approximate kernel, the generated chromosome is compared to the chromosome with the least PT in the population so far. If all PT values in the newly generated chromosome is higher than or equal to the PT values of the least precision chromosome, this new chromosome can certainly meet the quality-of-result target; otherwise, the corresponding kernel should be executed for accuracy check.

3) Fitness Function

Given a kernel, the fitness function returns a value showing the desirability of the approximate kernel. The fitness value is used by the selection operation to decide which individuals would survive to the next generation. Our main objective is to find an approximate kernel that minimizes the resource utilization on FPGA while meeting the quality-of-result requirement. To achieve this objective, our fitness function computes a weighted summation of its assigned PT values in the chromosome to estimate the area occupancy. For each variable, the weight is determined by a coefficient assigned to each precision tag multiplied by the number of times the variable is used in operations in the kernel. (The coefficients are determined through simulations which is 0, 1, 2, 6 for PT of 1, 2, 3, 4 respectively.) The higher the precision and the number of times the variable is used in operations, the higher weight it gets. With this definition, the lower the fitness value, the lower area occupancy that configuration has.

4) Selection and Genetic Operators

We use two genetic operators, crossover and mutation, to produce new chromosomes. Crossover combines the first part from one parent chromosome to the second part from the other parent chromosome to produce a child chromosome. In this implementation, the crossover point is selected randomly. Mutation operation randomly modifies PT values of approximable variables in the chromosome. The new PT value is a random value in the range of LP and EP for the approximable variable. The newly generated chromosome is only accepted if it meets the quality-of-result requirement; otherwise, the operation is applied again.

There are many possible selection algorithms to select more fit individuals from the new and old population for the next generation. To rank area occupancy of the approximate kernels without synthesizing and mapping the kernel on FPGA, we use the fitness values as an estimate of the area occupancy. The selected chromosomes are sorted based on their estimation of area occupancy (fitness value) in each iteration. The top best individuals are always transferred for the next generation (elitism selection). For the rest, individuals are selected based on the proportionate selection where some of them might change with the crossover and mutation operations. For the simulation purpose, the crossover rate, mutation rate, and elitism rate is 0.7, 0.05, and 0.25

respectively. The algorithm runs as long as the user defined number of iterations has not been passed yet or when the best fitness values stop growing any further.

Until here, the genetic algorithm finds the final solution using only one input test case. This solution is verified with the other input test cases from the training set. If it meets the quality-of-result requirement for all inputs of this set, this approximate kernel is the final solution. Otherwise, either the other top chromosomes in the population is checked or the genetic algorithm is applied again for the failed input (Line 23–40 in Algorithm 1).

When this procedure is terminated, the best chromosome with the lowest fitness value is selected and translated to its corresponding approximate kernel which has the least area estimation on FPGA. This kernel is passed to the Altera SDK tool to be synthesized and mapped on the FPGA.

V. Experimental Results

A. Experimental Setup

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. All of these applications are error tolerant and have approximable data in their kernels. The number of variables in these kernels are in the range between 11 and 17.

GRATER source-to-source compiler [3] is implemented in Python, and accepts the exact kernel, the desired quality metric, and a set of 100 training input test cases as its inputs. GRATER utilizes the AMD Evergreen Radeon HD 5870 GPU device for accelerated profiling experiments, and finally generates an optimized approximate kernel with the minimum area occupancy estimation and an acceptable output error. The approximate OpenCL kernels were synthesized for Altera DE5 board with a Stratix V FPGA using Altera OpenCL SDK 13.1 tool [1].[4] Sections V-B and V-C detail how GRATER can reduce the area and correspondingly increase the throughput for different applications.

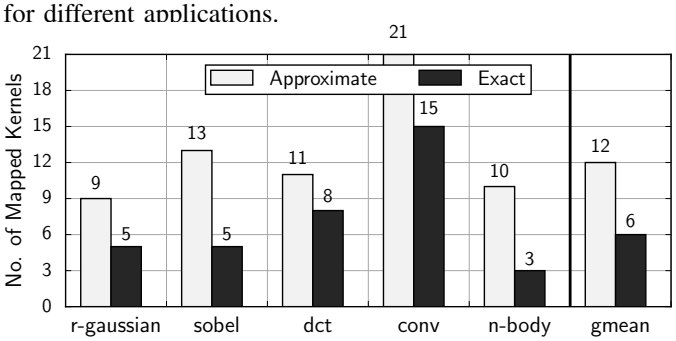

Fig. 3: Number of mapped kernel pipelines on FPGA.

[4] It should be noted that the accelerated profiling process on GPU takes order of milliseconds to determine if the kernel can meet the quality-of-result target. While it takes on average more than an hour to synthesize the approximate OpenCL kernels on Stratix V FPGA.

B. Area Savings with Approximate Kernels

Table I shows the resource utilization for an exact kernel and the optimized approximate kernel. As shown, the area utilization is reduced by an average of 14%–25% for different FPGA resources using the transformed approximate kernel instead of the exact one. This is achieved by the proper precision tuning of the kernel variable types that brings area saving without scarifying the output quality. Fig. 3 compares the maximum number of mapped kernels on the FPGA board for the exact kernels and the approximate kernels. For instance, the exact r-gaussian kernel contains 12 `float` variables in which 5 of them are converted to `short` and one of them to `char`. This precision tuning reduces DSP block utilization by 45% and logic utilization by 11%, hence enables mapping 9 approximate kernels rather than 5 exact kernels. Considering the geometric mean across all the benchmarks, GRATER is able to map 12 approximate kernels instead of 6 exact kernels in the fixed area budget of the FPGA. Given a fixed area budget of the FPGA, GRATER improves the number of mapped kernel by a factor of 2× on average (maximum 3.3× in n-body).

C. Speedup

As shown in Section V-B, GRATER reduces the synthesized area for the approximate kernels on the FPGA. Therefore, the number of parallel kernels (i.e., the kernel pipelines) that can be fitted into the FPGA is increased resulting in higher throughput. Fig. 4 shows the corresponding kernel speedup – throughput of the approximate kernel normalized to throughput of the exact kernel. As an example, for the n-body kernel the number of kernel pipelines that can be mapped into FPGA is increased from 3 for the exact kernel to 10 for the approximate kernel. Although the maximum clock frequency for this kernel is changed from `209` MHz for the exact kernel to `187` MHz for the approximate kernel, the approximate n-body kernel reaches to 2.98× higher throughput compared to the exact kernel. As another example, convolution kernel reaches 1.41× higher throughput. For this kernel, the number of kernel pipelines (and the maximum clock frequency) is increased from 15 kernel pipelines (and `191` MHz) for the exact kernel to 21 kernel pipelines (and `193` MHz) for the approximate kernel. A geometric mean of 1.82× higher throughput is achieved across these evaluated benchmarks. GRATER increases the number of kernel pipelines until one of the resources reaches its maximum limit. The limiting factor for each of the kernels are as follow: r-gaussian (Logic block, 96%), sobel (DSP block, 91.4%), dct (RAM,

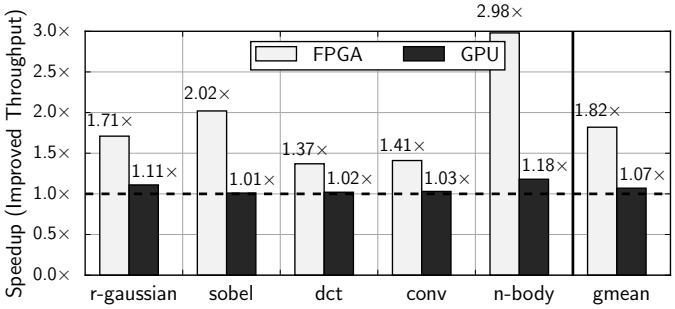

Fig. 4: Speedup with GRATER on FPGA and GPU.

TABLE I: Area utilization for exact (Ext) and approximate (Apx) kernels on StratixV FPGA

Resource utilization	r-gaussian		sobel		n-body		dct		conv		Avg. Area Reduction
	Ext	Apx	Ext	Apx	Ext	Apx	Ext	Apx	Ext	Apx	
ALUTs	70668	60008	72937	52213	119044	58333	50530	48271	53092	48573	21.5%
Registers	110668	90825	114892	91218	172894	95156	85707	80202	91062	80933	20.2%
Logic blocks	27%	24%	29%	23%	43%	25%	22%	21%	23%	21%	17.3%
DSP blocks	16.4%	9%	19%	19%	30%	21%	25%	22%	1.95%	1.17%	25.4%
Memory bits	7.91%	7.09%	3%	3%	8.1%	4%	9.56%	4.39%	3.7%	3.36%	24.8%
M20K blocks	20.23%	18.4%	18%	18%	25%	17%	22.81%	17.22%	15.03%	14.06%	14.4%

99%), convolution (Logic, 92%), n-body (DSP, 93%). Fig. 4 also summarizes the speedup for executing the approximate kernels on the GPU, normalized to the exact kernel execution time. The GPU exhibits a maximum speedup of 18% (with a geometric mean of 7%) due to its inflexible pipeline that cannot be fully customized to leverage the precision tuning for boosting throughput per unit of area.

To evaluate the quality loss, we use PSNR for image processing applications and average relative error for the other application domains [13, 16]. We compute the quality loss of each approximate kernel by comparing against the output elements from the exact kernel. For simplicity, here we report the quality loss of all applications by average relative error metric. We set the quality loss target to a maximum of 0.7% for image processing applications (which is equivalent to PSNR of a minimum 30 dB) and 1% for other applications which is conservatively aligned with other work on quality trade-offs [11, 12, 14, 20]. We verify the output quality of the optimized approximate kernel with 100 different test input patterns, other than the training input set. Fig. 5 shows the minimum, maximum, and average quality loss for all the evaluated applications. In all applications, the maximum quality loss is below the required threshold. Hence, it satisfies the target quality-of-result.

The execution time of our proposed algorithm is within few seconds (for sobel and r-gaussian that find the solution without running the genetic algorithm) to few minutes for others.

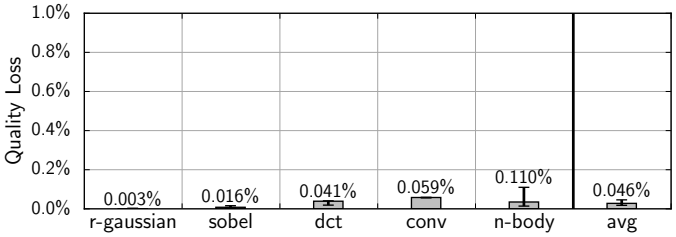

Fig. 5: Quality loss with GRATER.

VI. CONCLUSION

This work aims to address the following challenge: *how to exploit approximation in order to increase the benefits of FPGA accelerators without changing the FPGA hardware structure?* Our approach is providing more opportunities for parallel execution by reducing the precision of kernel's data and operations. To this end, we devise GRATER that systematically transforms an OpenCL kernel to an approximate version through a genetic algorithm by reducing its area on the FPGA using the state-of-the-art high-level synthesis tools. The reduction in area results in better utilization of data-level parallelism and thereby increased throughput. The results show that GRATER integrates a larger number of *parallel* kernels on the same FPGA fabric that leads to 1.4×–3.0× higher computational throughput on a modern Altera FPGA with less than 1% loss of quality. GRATER provides these significant benefits without applying any modifications to the underlying FPGA hardware. This feature confirms the efficiency of our framework in exploiting approximation with current hardware platforms. FPGA accelerators provide significant gains in performance and efficiency, yet still require relatively long design cycles to achieve those gains. Automated workflows, such as ours, that improve the benefits of FPGA acceleration are imperative to their widespread applicability.

VII. ACKNOWLEDGMENTS

This work was supported by the NSFs Variability Expedition (1029783), a Qualcomm Innovation Fellowship, NSF award CCF #1553192, Semiconductor Research Corporation contract #2014-EP-2577, and a gift from Google.

REFERENCES

[1] Altera SDK for OpenCL. http://altera.com/products/software/opencl/opencl-index.html.
[2] AMD SDK v2.9. http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/.
[3] "GRATER Transcompiler," https://bitbucket.org/act-lab/grater/src.
[4] SDAccel. http://www.xilinx.com/products/design-tools/sdx/sdaccel.html, 2015.
[5] D. Chen, et al. Invited paper: Using OpenCL to evaluate the efficiency of CPUs, GPUs and FPGAs for information filtering. FPL, 2012.
[6] G. Constantinides, et al. Numerical data representations for FPGA-based scientific computing. *Design Test of Computers, IEEE*, 2011.
[7] F. de Dinechin, et al. Designing custom arithmetic data paths with FloPoCo. *Design Test of Computers, IEEE*, 2011.
[8] A. Eiben, et al. Introduction to Evolutionary Computing. *Natural Computing Series. Springer*, 2007.
[9] A. Gaffar, et al. PowerBit - power aware arithmetic bit-width optimization. FPT, 2006.
[10] N. Herve, et al. Data wordlength optimization for FPGA synthesis. *Signal Processing Systems Design and Implementation*, 2005.
[11] A. Rahimi, et al. A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters. *CODES+ISSS*, 2013.
[12] P. Kulkarni et al. Trading accuracy for power with an underdesigned multiplier architecture. *VLSI Design*, 2011.
[13] S. Misailovic, et al. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. *OOPSLA'14*, 2014.
[14] T. Moreau, et al. SNNAP: Approximate computing on programmable socs via neural acceleration. *HPCA*, 2015.
[15] K. Nepal, et al. ABACUS: A technique for automated behavioral synthesis of approximate computing circuits. *DATE*, 2014.
[16] P. Roy, et al. ASAC: Automatic sensitivity analysis for approximate computing. *LCTES '14*, pp 95-104, 2014.
[17] E. Schkufza, et al. Stochastic optimization of floating-point programs with tunable precision. *PLDI '14*, 2014.
[18] W. Sung. et al. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *IEEE Trans. Signal Processing*, 1995.
[19] A. Yazdanbakhsh. et al. Axilog: Language Support for Approximate Hardware Design. *DATE*, 2015.
[20] A. Yazdanbakhsh. et al. Neural Acceleration for GPU Throughput Processors. *MICRO*, 2015.