Ve370 Introduction to Computer Organization

Project 3 Report

Group 40

Dong Jing, 515370910182

Zhu Hanqi, 515370910113

Li Yuting, 515370910108

Techniques and Future Development of Approximate Computing

I. Introduction and Background

Recently, large-scale applications play a more and more important role in our daily life. But power and energy considerations is a limiting factor for digital hardware design. In the design, how to reduce energy consumption and the size of circuit becomes increasingly significant, especially for some mobile and embedded computing systems. Meanwhile, these applications also have an inherent tolerance to errors in computation, like applications that involve media processing, recognition and data mining. These applications share a common characteristic that it is not necessary to have a perfect accurate result. What they want is a result in acceptable range and small circuit size as well as low energy consumption. These applications are imprecision-tolerant. The sources of imprecision-tolerance are mainly divided into three parts: [1] perceptual limitations: these are determined by the ability of the human brain to fill in missing information and filter our high-frequency patterns; [2] redundant input data: this redundancy means that an algorithm is able to be lossy and still be adequate; [3] noisy inputs.

Approximate computing techniques have been proposed to solve this problem. The idea of approximate computing is that, use a smaller and less complex circuits to achieve the same goal of the large and complex circuit with an acceptable correct result. Approximate computing can be considered as a tradeoff between accuracy and energy. For some applications, the result need not be so accurate that we can use some approximate components to replace the accurate components in the circuit to make the size of circuit smaller and decrease the power with result in an acceptable range. We call this kind of applications error-tolerant application. To design an approximate circuit, we need some approximate units and some methods to know whether an approximate circuit works well.

In this paper, we will mainly talk about how approximate computing applies to FPGA and GPU and the future development of approximate computing. Before talking these

parts, we will introduce some basic knowledge about approximate computing.

2. Overview of Approximate Units and Metrics

To build an approximate computing circuit, we need to build approximate arithmetic units first. Many people focus on the approximate arithmetic units. There are three main kinds of approximate computing units which are used most widely, approximate adder, approximate multiplier and approximate comparator.

There are many different approximate adders with different error distributions. In this part, we will only introduce several approximate adders.

1) Approximate mirror adders: It consists of five approximate mirror adders, which is an efficient adder. It removes some transistors to decrease power and simplify circuit.

2) *Accuracy configurable adder II*: It consists of several sub adders. The length of sub adders is smaller than the ACA-II. Sub adders are used to break the long carry chain. Except the first sub adder, other sub adders use several bits to predict the carry-in. Combine all the result of all sub adders and get the result. Because the long carry chain is broken, the size of circuit will be decreased and the speed of computation will increases.

The approximate multipliers which are widely used can be classified into following two types.

1): Multipliers with approximate partial products. This multiplier uses precise adders.

2): Multipliers with precise partial products. This multiplier uses approximate adders.

The main kind of approximate comparator just ignore several bits. It will ignore k LSBs and compare the rest n-k bits. When there is difference in the k LSBs, the result is wrong. The error rate of this comparator is

$$ER = (\frac{1}{2})^{n-k} \cdot \frac{2^k - 1}{2^{k+1}} = \frac{2^k - 1}{2^{n+1}}$$

To know how an approximate unit and an approximate circuit work, we need some metrics for approximate computing

Error rate is the fraction of incorrect outputs out of a total number of inputs for an approximate circuit.

Error significance refers to the degree of error severity due to approximate operations.

Error distance is defined as the arithmetic distance between an inexact output and the correct output for a given input.

Mean error distance is the average effect of multiple inputs.

Normalized error distance is the normalization of MED for multiple-bit adders.

Probability mass function is used to see the error distribution for an approximate circuit. People always use PMF to check whether the approximate circuit gives an approximate result in acceptable range.

3. Approximate Computing Techniques for GPUs

A graphics processing unit (GPU) is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer

intended for output to a display device. They are used in embedded systems, mobile phones, personal computers, workstations, game consoles, etc. Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more efficient than general-purpose CPUs. In GPU assisting, there have been some great applications of approximation computing methods, and some typical algorithms are used to improve the performance.

In researching this area, we read 3 relative paper. To understand these topics, we also attained some new terms and ideas.

In computational complexity theory, the complexity class FP is the set of function problems which can be solved by a deterministic Turing machine in polynomial time; it is the function problem version of the decision problem class P. Roughly speaking, it is the class of functions that can be efficiently computed on classical computers without randomization. A binary relation P(x,y) is in FP if and only if there is a deterministic polynomial time algorithm that, given x, can find some y such that P(x,y) holds. The difference between FP and P is that problems in P have one-bit, yes/no answers, while problems in FP can have any output that can be computed in polynomial time. For example, adding two numbers is an FP problem, while determining if their sum is odd is in P. Just as P and FP are closely related, NP is closely related to FNP. Polynomial-time function problems are fundamental in defining polynomial-time reductions, which are used in turn to define the class of NP-complete problems. Because a machine that uses logarithmic space has at most polynomial many configurations, FL, the set of function problems which can be calculated in log-space, is contained in FP. It is not known whether FL = FP; this is analogous to the problem of determining whether the decision classes P and L are equal.

In the reduction problem, one needs to reduce a number of values to a single one, such as sum, maximum, minimum, etc. More precisely, the reduction problem takes as input a set of values $v_1,\ldots, v_n$ and outputs a single value $v = v_1 \oplus \ldots \oplus v_n$, where $\oplus$ is an associative and commutative operator. Thus, summation is a special case of the reduction problem. The reduction problem can be solved in O(logN) passes on the GPU using a fragment program. Existing algorithms can only perform global reductions, that is, reducing values of all the pixels in a texture into one single value.

In the field of computer graphics, a shader is a special type of computer program that was originally used to do shading (the production of appropriate levels of light, darkness, and color within an image) but which now perform a variety of specialized functions in various fields of computer graphics special effects or do video post-processing unrelated to shading, and even functions unrelated to graphics at all. Shaders calculate rendering effects on graphics hardware with a high degree of flexibility. Most shaders are coded for a graphics processing unit (GPU), though this is not a strict requirement. Shading languages are usually used to program the programmable GPU rendering pipeline, which has mostly superseded the fixed-function pipeline that allowed only common geometry transformation and pixel-shading functions; with shaders, customized effects can be used. The position, hue, saturation, brightness, and contrast of all pixels, vertices, or textures used to construct a final image can be altered on the fly, using algorithms defined in the shader, and can be modified by external

variables or textures introduced by the program calling the shader.

Here is a summary of the three topics.

Hsiao et al. [2013] note that reducing the precision of **FP** representation and using the fixed-point representation are two commonly used strategies for reducing energy consumption of a GPU **shader**. Reduced-precision FP provides wider numerical range but also consumes higher latency and energy; the opposite is true for fixed-point representation. They propose an automatic technique that intelligently chooses between these two strategies. Their technique performs runtime profiling to record precision information and determine feasible precisions for both fragment and vertex shading. Then both these rendering strategies are evaluated with selected precisions to find the more energy-efficient strategy for the current application. Finally, the winning strategy with its precision is used for the successive frames, except that memory access-related and other critical operations are executed in full-precision FP. They show that their technique provides higher energy saving and quality than using either strategy alone.

Zhang et al. [2014] note that the FPU and special function units are used only in arithmetic operations (but not in control/memory operations) and they contribute a large fraction of GPU power consumption. Thus, using inexact HW(short for hardware) for them can provide large energy savings at bounded quality loss without affecting correctness. Based on this, their technique uses linear approximation within a reduced range for functions such as square root, reciprocal, log, FP multiplication and division, for example, $y = 1/\sqrt{x}$ is approximated as $y = 2.08 - 1.1911x$ in the range $x \in [0.5, 1]$. They build the functional models of inexact HW and import them in a GPU simulator that can also model GPU power consumption. In the simulator, each inexact HW unit can be activated or deactivated and their parameters can be tuned. They first obtain the reference (exact) output, then run a functional simulation with inexact units to obtain the inexact output. By comparing the reference and inexact output using an application-specific metric, the quality loss is estimated. If the loss exceeds a threshold, then either an inexact unit is disabled or its parameters are adjusted, depending on the program-specific error sensitivity characterization. The simulation is performed again with an updated configuration and this process is repeated until the quality loss becomes lower than the threshold. They show that their technique allows exercising trade-off between quality and system power and provides large power savings for several compute-intensive GPU programs.

Byna et al. [2010] present an ACT for accelerating a supervised semantic indexing (SSI) algorithm, which is used for organizing unstructured text repositories. Due to the data dependencies between the iterations of SSI, parallelism can only be exploited within individual iterations. Hence, for small datasets, GPU implementation of SSI does not fully utilize the GPU HW resources. They note that SSI is an error-tolerant algorithm and the spatial locality of writes between different iterations of SSI is low, (i.e., these iterations rarely update the same part of the model). Also, after some initial iterations, only a few iterations perform any updates. Using these properties, dependencies between iterations can be intelligently relaxed, then multiple iterations can be run in parallel. Also, noncritical computation is avoided, for example, processing of common words, such as “a,” “of,” and “the,” are avoided since it does not affect the accuracy.

They show that their technique improves performance compared to both a baseline GPU implementation and a multicore CPU implementation.

4. Approximate Computing Techniques For FPGAs

One of the techniques for FPGAs is presented by Moreau et al. [1]. They present a technique for neural acceleration of approximable codes on a programmable system on chip (SoC), that is, an off-the-shelf field-programmable gate array (FPGA). Their approach explores the performance opportunity of neural processing unit (NPU) acceleration implemented on off-the-shelf FPGAs and without tight NPU–core integration, avoiding changes to the processor ISA and micro-architecture.

With the help of technology improvement on performance and energy efficiency, researchers are exploring new avenues in computer architecture. Two of the emerging trends are specialized logic in the form of accelerators or programmable logic, and approximate computing, which exploits applications' tolerance to quality degradations. Approximate computing trades off accuracy to enable novel optimizations. The confluence of these two trends leads to additional opportunities to improve efficiency. One example is neural acceleration, which trains neural networks to mimic regions of approximate code. Once the neural network is trained, the system no longer executes the original code and instead invokes the neural network model on a NPU accelerator. This leads to better efficiency because neural networks are amenable to efficient hardware implementations. However, prior work on neural acceleration has assumed that the NPU is implemented in fully custom logic tightly integrated with the host processor pipeline. While modifying the CPU core to integrate the NPU yields significant performance and efficiency gains, it prevents near-term adoption and increases design cost/complexity. This technique faces these challenges.

There are two basic ways to use their technique (SNNAP). The first is to use a high-level, compiler-assisted mechanism that transforms regions of approximate code to offload them to SNNAP. This automated neural acceleration approach requires low programmer effort and is appropriate for bringing efficiency to existing code. Approximate applications can take advantage of SNNAP automatically using the neural algorithmic transformation. This technique uses a compiler to replace error-tolerant sub-computations in a larger application with neural network invocations. The process begins with an approximation-aware programming language in which code or data can be marked as approximable. In any case, the programmer's job is to express where approximation is allowed. The neural-acceleration compiler trains neural networks for the indicated regions of approximate code using test inputs. The compiler then replaces the original code with an invocation of the learned neural network. Lastly, quality can be monitored at run-time using application-specific quality metrics. The second is to directly use SNNAP's low-level, explicit interface that offers fine-grained control for expert programmers while still abstracting away hardware details. SNNAP behaves as a throughput-oriented accelerator: it is most effective when the program keeps it busy with a large number of invocations rather than when each individual invocation must complete quickly.

Another technique is presented by Sampson and his fellows. Their approach is ACCEPT (an Approximate C Compiler for Energy and Performance Trade-offs), a framework for approximation that balances automation with programmer guidance. ACCEPT automatically applies a variety of approximation techniques, including hardware acceleration, while ensuring their safety. They apply ACCEPT to nine workloads on a standard desktop, an FPGA-augmented mobile SoC, and an energy-harvesting sensor device to evaluate the annotation process.

Recent work has shown how to accelerate approximate programs with hardware neural networks. Neural acceleration uses profiled inputs and outputs from a region of code to train a neural network that mimics the code. The original code is then replaced with an invocation of an efficient hardware accelerator implementation, the NPU. But the technique has thus far required manual identification of candidate code regions and insertion of offloading instructions. ACCEPT automates the process.

ACCEPT safely and efficiently harnesses the potential of approximate programs by combining three main techniques: (1) a programmer–compiler feedback loop consisting of source code annotations and an analysis log; (2) a compiler analysis library that enables a range of automatic program relaxations; and (3) an autotuning system that uses dynamic measurements of candidate program relaxations to find the best balances between efficiency and quality. The final output is a set of Pareto-optimal versions of the input program that reflect its efficiency–quality trade-off space. ACCEPT implements an automatic neural acceleration transform that uses an existing configurable neural-network implementation for an on-chip FPGA , which is based on the previous technique. ACCEPT uses approximate region selection (x4.2) to identify acceleration targets, then trains a neural network on execution logs for each region. It then generates code to offload executions of the identified region to the accelerator. The offload code hides invocation latency by constructing batched invocations that exploit the high-bandwidth interface between the CPU and FPGA. This techniques targets a commercially available FPGA-augmented SoC and does not require specialized neural hardware.

5. Conclusion

As we can see, approximate computing technique is applied to arithmetic circuit. Part III and Part IV show how approximate computing works on GPUs and FPGA. It shows that approximate computing technique is not only an idea to reduce energy consumption and size of circuit but also has great influence on some applications. It can accelerate the speed of program with a little error. It can improve performance of GPUs compared to both a baseline GPU implementation and a multicore CPU implementation.

Approximate techniques can be divided into five types, selective approximation, timing relaxation, functional approximation, domain specific approximation and data/information approximation. Table 1 shows the classification of different types of approximations.

Selective Approximation	Analysis of software code or instructions to find a certain accuracy mode for a part of code

Timing Relaxation	Relaxing of synchronization, timing and handshaking constraints to reduce control overhead
Functional Approximation	An approximate alternative of an algorithm that reduces size of circuit and power.
Domain Specific Approximation	Leveraging the domain specific knowledge for approximations in applications and their algorithms
Data/Information Approximation	Use of unreliable memories, load value approximation, data truncation, data decimation and more

Table 1. Classification of Types of Approximations

In layers, we can divide approximation computing techniques into three kernels. In program, the approximation technique includes loop perforation, code perforation and tunable kernels, etc. In architecture, approximation technique includes approximate storage, ISA extensions and approximate accelerators. In circuit layer, approximation technique includes imprecise logic, voltage overscaling and analog computation.

In recent years, approximate computing has developed a lot and gained significant traction. But it is still a new rising field. Most of works about approximate computing techniques only work for a small set of applications. We still need to develop a new way to measure how approximate computing works. The calculation about errors is a big problem in design of approximate circuit. And we should have some research attempts to construct new hardware and software interface for approximate computing, although it is a heavy burden for programmers due to the quality of applications. In order to apply scientific and engineering methods to designing, optimizing and manufacturing an approximate computing system (both hardware and software), we need to refine the end-user perception metric into quantifiable, testable specifications for individual components along with solutions for the following questions. How to determine whether a part on a production line passes qualifying tests for shipment. How to know the propagation of errors from inputs to outputs. What programming languages and tools are suitable for approximate computing design?

In conclusion, existing work about approximate computing has improved the performance of power and size of circuit in some applications. It can work for GPUs and FPGA. But we still need significant innovations and research efforts to enable approximate computing as a practical mainstream computing paradigm. And we still need to know more about how error propagates between approximate computing units.

Reference

Chih-Chieh Hsiao, Slo-Li Chu, and Chen-Yu Chen. 2013. Energy-aware hybrid precision selection framework for mobile GPUs. *Computers and Graphics 37, 5, 431–444.*

Hang Zhang, Mateja Putic, and John Lach. 2014. Low power GPGPU computation with imprecise hardware. *Design Automation Conference (DAC'14). 1–6.*

Surendra Byna, Jiayuan Meng, Anand Raghunathan, Srimat Chakradhar, and Srihari Cadambi. 2010. Best- effort semantic document search on GPUs. *General-Purpose Computation on Graphics Processing Units. 86–93.*

Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. 2015. *SNNAP:* Approximate computing on programmable SoCs via neural acceleration. *International Symposium on High Performance Computer Architecture (HPCA'15). 603–614.*

Adrian Sampson, Andr´e Baixo, Benjamin Ransford, Thierry Moreau, Joshua Yip, Luis Ceze, and Mark Oskin. 2015. ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing. *Technical Report UW-CSE-15-01-01*. University of Washington, Seattle, WA.

Mittal, S. (2016). A Survey of Techniques for Approximate Computing. *ACM Computing Surveys, 48(4)*, 1-33. Doi:10.1145/2893356