How to Answer “Haven’t We Done This Already?”, and Challenges/Opportunities in Approximate Computing

Luis Ceze, University of Washington

Approximate computing is about trading off application output accuracy for better efficiency across the entire stack, from algorithms down to devices. This is appealing because it may provide a viable way forward given the current performance, energy and reliability trends of transistor technology. Being able to better exploit fundamental behavior of material physics all the way up to the application output has the potential to lead to many orders of magnitude improvement in performance and energy efficiency. I see approximate computing research as being *all about the hardware, but most, if not all challenges are in the software.*

In this position paper I will first comment on common positioning questions that I get when talking about approximate computing — and I am sure many others in this workshop get or have similar questions. I will then move on to listing a set of research questions that attempt to summarize two workshops that I helped organize in early 2014.

“Haven’t we done this already?” Below are questions that I’ve heard many times. They are all fair questions, albeit I can see them being a tad irritating for people doing work on approximate computing. Attempting to answer them helps us understand what approximate computing is really about.

- “Haven’t we been doing approximation for a long time? What about floating point?”

  Indeed approximation is pervasive in systems design, and a perfect example is floating point number representation and operations. Floating point numbers are an approximation of real numbers, done brilliantly. We generally know how to deal with them (though often come across surprises), and numerical analysts have been dealing with convergence and precision issues for a while. However, a very important aspect of FP representation and operations is that they are *deterministic and very well defined.* This underpins everything related to FP approximation.

  Approximating computing often includes *non-deterministic* hardware (e.g., sub-threshold operation or analog techniques). This leads to interesting questions of whether numerical analysis techniques will work for non-deterministic approximations and my hunch is that they will not. Moreover, approximate computing can involve changing the underlying execution model (e.g., from von Neumann to a Neural model) or dramatically changing the semantics of the program in a much less clearly defined way (e.g., loop perforation), both of which are typically out of the radar of what numerical analysis deals with (and would probably many numerical analysts cringe).

- “Wait, fault tolerance is all about dealing with non-deterministic faults!”

  An inherent aspect of approximate computing is *trading off accuracy for better efficiency*, whereas fault-tolerance is broadly about *hiding* faults and exposing a perfect, reliable computing environment.

- “What about probabilistic programming and computing, isn’t that the same thing as approximate computing?”

  Probabilistic programming is about specifying computation over probabilities and/or dealing with uncertainty in data. For example, declaring a probabilistic graphical model and performing inference over it. I see this as largely orthogonal to the output accuracy trade-offs in approximate computing. For example, you can compute a precise evaluation of a probabilistic model. Case in point, Lyric semiconductor (now part of Analog Devices) offered chips for probabilistic computing, which was essentially an ASIC to speed up computing with probabilities, but *not approximately.*

  Nevertheless, one can use approximate computing to improve the efficiency of computing with probabilities, and one can use probabilistic reasoning techniques to deal with the execution uncertainty brought on by computing approximately (e.g., Sampson et al.’s Probabilistic Assertions work in PLDI 2014).

- “Aha, but the machine learning folks have been dealing with quality trade-offs for a long time!”

  Indeed! They do so because because it is very hard to write down a program to compute what they want to compute, so they statistically derive models that generalize the mapping of inputs to outputs in the examples. Those models are inherently approximate, consequently ML practitioners deal with quality issues all the time, it is part of their normal operation. They can certainly use approximate computing to improve efficiency of the computation involved in their magic. But I believe approximate computing can greatly benefit from techniques ML practitioners use to deal with quality, like feedback from users, very methodical testing, etc.

With all that said, I think we can make approximate computing better and more interesting if we bring together lessons from all the areas discussed above. Can we reuse numerical analysis techniques to deal with non-deterministic approximations? Can we bring quality control techniques from machine learning to more general programs? Can we use probabilistic graphical models to reason and deal with uncertainty in the *execution* of a program?

A community view of research questions on approximate computing. In early 2014 I helped organize two major workshops focused on approximate computing. One was an invitation-based DARPA ISAT workshop (about 35 people across the stack), and the other was a workshop held with ASPLOS based on peer reviewed papers. Below I summarize some of the recurring questions discussed in these workshops:

Abstractions. What should the HW/SW interface include? How does one express/verify/monitor quality-of-results (QoR) of an approximate program? How should we reason about quality composability of components, each with their own quality trade-offs?

Algorithms. What is the relationship between approximation inherent to Machine Learning, numerical algorithms and approximation at the execution level? What are algorithmic transformations that would be synergistic with approximation at the execution level?

Tools. How do we test approximate programs (i.e., when are they ready to ship)? How do we debug approximate programs (i.e., how do we find where the quality degradation stems from)? How do we assist programmers in finding and exploiting approximation opportunities in their programs?

Hardware. The true potential of approximate computing can only be unlocked with the right hardware support. What should the hardware look like? What mechanisms should it offer? Are they going to be non-deterministic, deterministic or a mix of both? Importantly, such hardware support is likely to involve both specialization and approximation. Where does most of the benefit come from? Specialization or approximation?

System. Approximation opportunities exist in computation, storage and communication. How should we think about end-to-end quality guarantees? How can the system adapt to address end-to-end quality issues in an efficient way? How do we avoid Admahl's law for approximation (i.e., benefits limited to portions that can indeed be approximated)?