Approximate Computing

By- Prashant Gupta

With over the top Thermal budgets and an immediate need to reduce the power requirements of the chip a lot of recent research has been shifted what is called as energy efficient computing. Optimization techniques like Dynamic Voltage scaling (DVFS) and approximate computing are gaining traction with a hope to somewhat solve this problem.

Approximate computing is basically allowing machines to have some sort errors/approximations in the computational model because the system is error resilient to some extent i.e. it will have about the same result even with some minor errors. The advantage of this kind of model will be using much less power as compared to having precise computation. If one looks closely, there are a lot of applications in today's world were approximate computing can be beneficial like mobile games, virtual reality, robotics, neural networks, large scale physical computations etc. and it doesn't compromise the usefulness of these program. The important thing to note here to strike a good tradeoff between the energy saved and the amount of approximations/errors caused in the meantime.

Let's take an example of Mobile Gaming, no one will mind a drop in few frames at the runtime while playing a game if it promises to save some portion of the energy that is required to render those frames. As long as the approximations are related to pixel data or audio samples there is no problem with approximations, but now if we start doing the same with let's say pointers, references etc. the game starts to behave arbitrarily with frequent crashes, which is completely undesirable. Hence, it is important to define which parts of the program can or cannot be approximated. Techniques like approximate-aware programming languages and approximate-aware compliers come in handy here. The approximate-aware programming languages allows programmers to express randomness (abstraction) in the program and isolate the parts that are needed to be precise. Some other checks that languages like *EnerJ* [1] provide are making sure that the data from a precise computation can flow to approximate computation if needed but not the other way around. Features like making precise computations implicit and then allowing to add approximate qualifies also helps in easy adoption of the language. Approximate-aware compliers can automatically transform the programs i.e. change semantics, take approximate search methods etc. The challenge here is to make the underlying optimization more and more automated to reduce the burden on the programmer. On the hardware side one can have a Dual Voltage system with High Voltage supplied for precise computations while a lower voltage supplied for an approximate computation. So, when the programmer defines some part of the program as approximate and the other part as precise the compiler dynamically allocates the approximate part to the registers supplied with lower power while the precise instruction to the registers supplied with higher power. The compiler can further adjust the order of placement of approximate instructions to make the program run faster. This Dual Voltage system optimization further adds to the complexity of the SRAM structure.

Another interesting addition to the approximate programming language and approximate compilers is approximate architecture. As we know advanced architectural optimizations like

Branch Prediction, Out-of-order speculative execution improves the performance of processor at the cost of increased architectural complexities and high energy consumption. The approximate architecture aims to strike a balance between computing quality and energy efficiency. The two major types of architecture of approximate computing are Coarse-Grained and Fine-Grained architecture. While the coarse-grained approximate computing is somewhat unreliable the fine-grained approximate computing is more disciplined. Processors can have heterogeneous cores, some of which are dedicated to precise computing while other are dedicated to the approximate computing using Dynamic voltage and Frequency scaling (DVFS).

An overhead that comes with grained approximate computing is the quality management which translates to checking the errors in the results of approximate programs. The number of invocations that are used to test the accuracy of the approximate computation add to energy overhead. It is practically not possible to test the accuracy of all the computations and hence, checking the invocations after every *N* approximate computations and adjusting the approximation mode of the subsequent instructions based on the results seem to be a reasonable choice.

Artificial Neural Networks can also be used in the approximation-aware compliers, which can accelerate transforming the code to approximate and precise parts using a trained neural model. They can be used to accelerate regions of approximate code and are useful for many applications like image classification, speech recognition etc. These applications usually also require a specialized hardware co-processor (like FPGA or GPU) called Neural Processing Unit (NPU) coupled with the main processor. Thus, the software needs to communicate effectively with the hardware to get the best energy efficiency under acceptable quality constrains which requires the programmer to have to both software and hardware skill to fully optimize the system. Hence, efforts are being focused towards making them more automated to reduce the burden on programmer and increase potential adaptability. The throughput and operating frequency between the processor and co-processor are two important parameters to look for while dealing co-processors. Implementations like *SNNAP* [2], which use neural transformation on computationally demanding regions of code in a program, and approximates it with a neural network promise an average 3.8x speedup of saving 2.8x energy.

References:

1. Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. 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). ACM, New York, NY, USA, 164-174. DOI=http://dx.doi.org/10.1145/1993498.1993518.
2. Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, Mark Oskin. SNNAP: Approximate Computing on Programmable SoCs via Neural Acceleration. HPCA 2015.