CPSA: Compute Precisely Store Approximately

Animesh Jain[1], Parker Hill[1], Michael A. Laurenzano[1], Md E. Haque[1], Muneeb Khan[2], Jason Mars[1], Lingjia Tang[1], Scott Mahlke[1]

University of Michigan, Ann Arbor[1], Uppsala University[2]

{anijain,parkerhh,mlaurenz,mdhaque,profmars,lingjia,mahlke}@umich.edu[1], muneeb.khan@it.uu.se[2]

Abstract

We propose a new approximate-computing paradigm, where computations are performed precisely while the data is stored approximately in the memory using data packing. This lets us reduce the memory traffic, improving application memory behavior. It achieves 85% memory savings for an accuracy target of 90%.

1. Introduction

Several applications that form a significant percentage of the server and datacenter workloads are increasingly using image processing, data mining and numerical analysis algorithm [6, 2, 4]. These applications require significant amount of computing and memory resources making it harder to achieve performance/latency targets. However, such applications also show error tolerance i.e. they can provide acceptable output quality when the application is subjected to some amount of error. Approximate computing [7, 1, 3], an emerging computing paradigm, takes advantage of this inherent characteristic to trade-off application accuracy with substantial performance gains or energy savings.

In this research, we focus on the memory characteristics of an application. Memory capacity and bandwidth play a major role in deciding the performance of an application. An application can spend substantial number of idle cycles waiting on the memory response. This is observed in applications such as Kmeans and matrix multiplication that are at the core of data mining and numerical analysis applications. In this paper, we apply approximation to the data to alleviate memory bottlenecks in memory hierarchy.

Specifically, we observed that the applications do not need all the bits in the input data. We substantiate this observation by studying Kmeans application output as shown in Figure 1. In Figure 1(a), we use precise 32-bit IEEE single precision representation, whereas (b), (c) and (d) use limited precision where inputs elements are represented using 24, 16 and 8 bits respectively. Here, only the bits in the input elements are approximated while the *computation is still happening in full precision*. We observe that the output quality is still close to full precision for 24 and 16 bits, however going to 8 bits drops the accuracy below acceptable levels. This experiment suggests that we can cut the memory usage substantially (by half in this case). These memory savings in turn can result in substantial performance gains. In this research, we aim to move only the necessary bits in the memory subsystem.

One way of achieving this in hardware is to add limited precision capability across the whole hardware stack i.e both computation and memory. However, this will be a hardware-expensive approach requiring multiple functional units with complex control logic to work on arbitrary precision. Another alternative is to perform this in software, for example, by using *half* data type instead of *float* data type. But, such data types provide very limited flexibility in the number of bits we can use. In addition, currently the instructions that use data types like *half* are expensive (5-7 cycles) which eat into the performance benefits achieved by memory savings.

In this paper, we propose a new computing paradigm, Compute Precisely Store Approximately (CPSA), where we remove the unnecessary bits from the data elements and store the data approximately while the computation is done precisely. This lets us i) pack more data elements in smaller memory space, and ii) apply approximation with simpler hardware additions as opposed to approximating complete hardware stack. The hardware changes are small because computing precisely lets us reuse the existing functional units obviating the need of supporting arbitrary precision calculation.

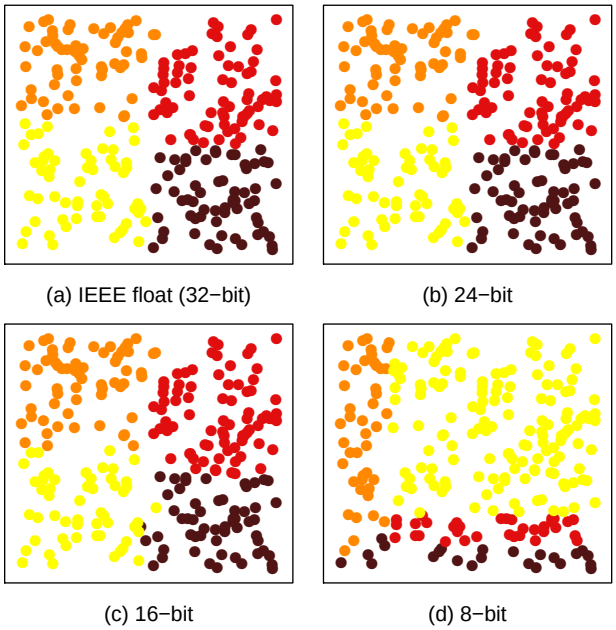

Figure 1: Kmeans input approximation

CPSA increases the effective memory capacity and bandwidth because the data elements are always packed in the memory subsystem. It can help applications whose dataset was earlier not able to fit in caches but now it fits because it consumes much lower space. Therefore, it can potentially reduce the number of idle cycles that are spent waiting for the memory response. In addition, it can also alleviate bandwidth bottleneck, because CPSA can bring more data elements in a single memory request. In this paper, we evaluate how much memory savings are obtained from CPSA. In future, we plan to provide an end-to-end system to convert these memory savings to application speedup.

2. Motivation

CPSA proposed an asymmetric store and computing paradigm where computations are done precisely while the data is stored approximately in the memory. Here, we focus on IEEE 32-bit single precision point values (1 sign, 8 exponent and 23 mantissa bits), however CPSA can be applied to any data type. We use custom limited precision format to represent the input elements. CPSA utilizes data packing to convert an IEEE floating point 32-bit in-

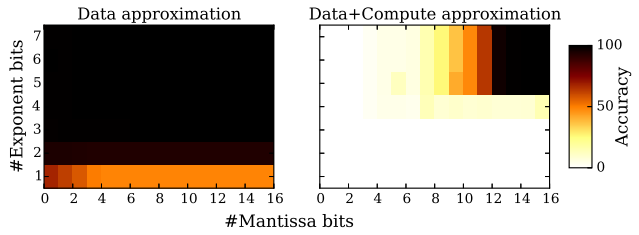

Figure 2: Comparison of accuracy between data approximation vs data+compute approximation

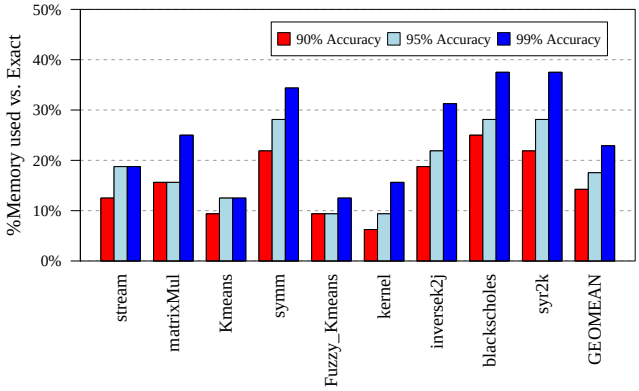

Figure 3: Memory savings obtained with CPSA

put element to a smaller representation custom precision format. This data packing allows us to pack more input elements in same memory space.

An alternative to CPSA is to add approximation throughout the hardware stack by adding functional units that can support computation at any arbitrary precision. We assert that if we alienate the approximation between compute and storage and limit it to just storage then we can further reduce the minimum number of bits required to achieve a target accuracy. We substantiate this claim by studying application Kmeans with a varying customized limited precision format. Figure 2 shows how accuracy varies with exponent and mantissa bits for data approximation (CPSA) against data+compute approximation for usps SVM dataset from UCI Machine Learning Repository [5]. We observe that we achieve 99% accuracy for a mere 5 bit input for CPSA as opposed to 17 bit input for data+compute approximation. Therefore, computing precisely enables more compact data packing.

CPSA needs both packing and unpacking support. Data packing is required to store the input data approximately. The input elements are then unpacked just before they are fed to the execution units for precise computation. Another alternative to CPSA is to perform these operations in software. However, supporting arbitrary precision packing and unpacking requires several operations which lead to significantly high overhead. We believe that packing and unpacking can be performed with simpler hardware units.

3. Tradeoff Study

We use 9 benchmarks picked from different domains like data mining and numerical analysis to study accuracy vs memory savings trade-offs. In this experiment, we use software simulation to represent the input elements with limited precision. The computation is performed precisely. For a certain accuracy target, we choose the best limited precision configuration that satisfies the accuracy constraint.

The findings of this experiment is shows in Figure 3. The figure shows how much memory is used for three accuracy targets: 90%, 95% and 99% relative to the exact execution of the application. We observe significant memory savings across all the applications. On an average, CPSA enables memory usage to a mere 15% (85% memory savings) compared to the exact execution for an accuracy target of 90%. This study shows that CPSA is capable of improving the memory behavior of the applications amenable to approximation.

4. Challenges and Future Work

In order to realise CPSA in hardware, we propose to add ISA support for two new instructions: *store-trimmed* and *load-trimmed instruction*. A load-trimmed instruction is responsible for loading a packed value from the memory and unpacking it to 32 bits before it can be fed to the functional units. Similarly, a store-trimmed instruction packs a 32-bit value and store it in the memory. Precision information with which data values have to be packed/unpacked is stored in the instruction encodings by the compiler.

The tradeoff study illustrates that CPSA achieves significant memory savings. But, there are several challenges involved in converting those memory savings into application speedup and create an end-to-end system. Here, we list the challenges and our plan on how we will tackle them.

- Unpacking needs to be done as close to the functional units as possible to reap the maximum benefits of memory savings. This way, the data is always stored in the packed format in the memory. In addition, unpacking hardware needs to be fast to allow application speedup.

  We plan to add a *data unpacker unit* between the caches and the register file which converts a limited precision value to a 32-bit value. This unpacked value is now passed on to the functional units for execution. Our preliminary study shows that the hardware for data unpacker unit is simple and it can perform the unpacking within one typical microprocessor clock cycle.

- The tradeoff study shows that the best precision configuration can be any integer and it is not restricted to a multiple of 8 bits(byte). This gives rise to byte-vs-bit addressability issue because the memory is byte-addressable while the packed data elements might not be byte-aligned.

  We plan to add a *dedicated address generation unit* which generates a byte-level address for the memory but provides bit-level access in the data returned from the memory. In this way, there are minimal changes to the existing processor design.

- Finally, the programmer needs assistance in finding out which variables are amenable to approximation. In addition, the programmer also needs to figure out what is the best precision level for an application. For an individual variable, this results in a search space of $8 * 23 (= 184)$ configurations. This space is exploded when multiple variables are approximated. Infact, it requires studying cross-effects of approximating several variables simultaneously.

  Currently, we propose a naive approach of applying a binary search over $\#exponent$ and $\#mantissa$ bits to reduce the search space from 184 to 8. But, it still requires improvement when several variables need approximation simultaneously.

References

[1] W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 198–209, New York, NY, USA, 2010. ACM.

[2] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE*, pages 1–9, May 2013.

[3] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. *SIGPLAN Not.*, 47(4):301–312, Mar. 2012.

[4] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[5] M. Lichman. UCI machine learning repository, 2013.

[6] J. Meng, S. Chakradhar, and A. Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[7] A. Sampson, J. Nelson, K. Strauss, and L. Ceze. Approximate storage in solid-state memories. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 25–36, New York, NY, USA, 2013. ACM.