Concise Loads and Stores: The Case for an Asymmetric Compute-Memory Architecture for Approximation

Animesh Jain[†], Parker Hill[†], Shih-Chieh Lin[†], Muneeb Khan[§], Md E. Haque[†],
Michael A. Laurenzano[†], Scott Mahlke[†], Lingjia Tang[†] and Jason Mars[†]
University of Michigan[†], Uppsala University[§]
{anijain, parkerhh, shihclin, mdhaque, mlaurenz, mahlke, lingjia, profmars}@umich.edu, muneeb.khan@it.uu.se

***Abstract*—Cache capacity and memory bandwidth play critical roles in application performance, particularly for data-intensive applications from domains that include machine learning, numerical analysis, and data mining. Many of these applications are also tolerant to imprecise inputs and have loose constraints on the quality of output, making them ideal candidates for approximate computing. This paper introduces a novel approximate computing technique that decouples the format of data in the memory hierarchy from the format of data in the compute subsystem to significantly reduce the cost of storing and moving bits throughout the memory hierarchy and improve application performance. This *asymmetric compute-memory extension* to conventional architectures, ACME, adds two new instruction classes to the ISA – load-concise and store-concise – along with three small functional units to the micro-architecture to support these instructions. ACME does not affect exact execution of applications and comes into play only when concise memory operations are used. Through detailed experimentation we find that ACME is very effective at trading result accuracy for improved application performance. Our results show that ACME achieves a 1.3× speedup (up to 1.8×) while maintaining 99% accuracy, or a 1.1× speedup while maintaining 99.999% accuracy. Moreover, our approach incurs negligible area and power overheads, adding just 0.005% area and 0.1% power to a conventional modern architecture.**

I. Introduction

Data-intensive applications from domains that include machine learning, numerical analysis and data mining are emerging as key processing bottlenecks in datacenter and server applications [1, 2, 3]. The sheer volume of processing needed to handle these workloads suggests that alternative computing paradigms may be needed to keep pace. One such paradigm, approximate computing, is a technique for exploiting inherent application tolerance for inaccuracy for significant performance improvements. Prior work has shown that a range of applications have this tolerance [4, 5, 6, 7, 8] having reasonable-quality outputs even when some processing is performed approximately.

One of the main processing bottlenecks among data-intensive applications is the memory subsystem, where capacity and bandwidth can be critical factors in determining application performance. Prior work has made this observation, resulting in a class of techniques focused on the problem of identifying and building systems that take advantage of replication and redundancy across different data elements in the memory hierarchy [9, 10, 11, 12, 13].

978-1-5090-3508-3/16/$31.00 © 2016 IEEE

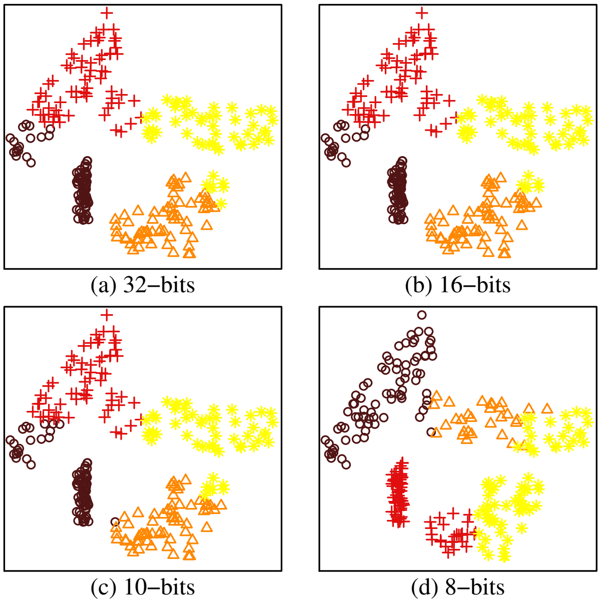

(a) 32–bits
(b) 16–bits
(c) 10–bits
(d) 8–bits

Fig. 1. Kmeans clustering output when applying a range of different storage formats. Similar accuracy for 32 (precise), 16 and 10 bits but poor accuracy for 8 bits

This work takes a new approach to addressing the problem, focusing on *marginal bits* – bits within the data representation that add little extra information among elements in a data structure while consuming a significant fraction of the memory and cache resources. Motivating this work is the observation that a number of applications (1) are tolerant to the removal of marginal bits, where the accuracy of results is minimally impacted and (2) stand to benefit significantly in performance and energy when the burden of storing and moving those additional bits is removed.

This opportunity is illustrated in Figure 1, which shows the output accuracy of Kmeans across a spectrum of different input bit counts. Figure 1(a) uses the “precise” 32-bit single-precision format, while (b), (c) and (d) use input elements represented in 16, 10, and 8 bits, respectively. Note that these experiments simply drop input bits; the computation still happens at 32-bit single precision. We observe that using 16 or 10 bits changes the cluster membership of a few points, but the results remain almost indistinguishable from the exact results. However, further reducing the input representation to 8 bits results in incorrect cluster membership for the majority of points. We have observed a similar trend in numerous applications, where *dropping marginal bits from the input has little impact on application accuracy but can significantly improve performance*. Our further investigation, as we will show in Section II, provided two more insights – 1) storing data with fewer bits while performing computation at full precision removes more marginal bits compared to the approach where fewer bits are used for both memory and compute, and 2) the remaining bits after removing the marginal bits often do not fit neatly into `double`, `float` or `half`, or any other representation that is a multiple of 8.

The goal of this work is to take advantage of this opportunity, reducing the pressure on the memory subsystem by enabling *concise storage* – a storage paradigm where the data elements are stripped of their marginal bits, removing the movement and storage costs associated with those bits in the memory subsystem. However, several challenges emerge in designing an approach that enables concise storage:

1) **Flexibility –** different applications need different numbers of bits to achieve satisfactory accuracy. Therefore, the design of a concise storage approach needs to have the flexibility to capture the wide spectrum of design points required by different applications and design objectives.
2) **Highly Concise Storage –** the approach should be able to identify as many marginal bits as possible, and avoid storing those bits throughout the memory subsystem while still delivering high-quality computational results.
3) **All Memory Levels –** techniques focused on a particular level of cache, or those focused solely on DRAM, only alleviate pressure on part of the memory subsystem. A better solution should reduce the burden of marginal bits throughout all levels of memory.
4) **Modular –** the approach should reuse as much existing compiler, architectural and micro-architectural infrastructure so that it can be easily built into those infrastructures. It should also be backward compatible and should have minimal impact on exact applications.

To address these challenges and enable concise storage throughout the memory hierarchy, this work motivates and describes ACME, an *asymmetric compute-memory extension* for conventional architectures. In ACME, data can be treated *asymmetrically*; computation is done on conventional 32-bit IEEE 754 single precision [14] values – while data is stripped of its marginal bits before being used in the memory hierarchy. ACME includes a simple ISA extension that can be leveraged by the programmer and compiler, adding two new instruction classes to the ISA to operate on concise data – *load-concise* and *store-concise* – to perform conversions between concise and single precision format via three small additional micro-architectural units. The asymmetric approach significantly increases the ability to achieve concise storage with small precision loss.

This asymmetric approach is flexible, allowing the application programmer and compiler make clear choices as to how much space is used to store data. The approach results in highly concise storage, significantly outperforming prior approaches based on leveraging redundancy across data elements or cache lines. The approach impacts all memory levels, converting between concise and full-precision formats at the boundary of the memory hierarchy, ensuring that data is stored concisely throughout the hierarchy. Finally, the approach is backward compatible and reuses existing hardware, adding three small additional micro-architectural units on top of existing designs to perform address generation for concise data accesses and to perform format conversion between concise and native data formats.

The specific contributions of this paper are as follows:

1) **Asymmetric Compute-Memory Extension –** we introduce ACME, a novel asymmetric compute-memory extension to conventional architecture that facilitates storing data *concisely* – without the marginal bits that add little to the accuracy of computation while significantly increasing the cost of storing and moving data. These concise formats allow bit-level specification of the exponent and mantissa components of float-point data formats, providing significant improvements in effective storage capacity and bandwidth while delivering high accuracy.
2) **Format Selection Assistant –** we introduce a Format Selection Assistant (FSA), a compiler component that automatically identifies the concise format given an accuracy specification from the application developer.
3) **Compiler, ISA and Hardware Support –** we describe compiler, ISA and hardware support needed to enable ACME. In particular, our approach adds two new classes of instructions, *load-concise* and *store-concise* that operate on concisely stored data. These instructions utilize three small additional hardware units responsible for performing address generation and for converting between concise and native data formats.

We perform an evaluation of ACME on 10 applications covering a range of data-intensive and compute-intensive applications. We find that the approach is able to achieve speedups that average 1.3× (up to 1.8×) while losing a maximum of 1% end-to-end application accuracy.

II. Background And Motivation

In this section, we discuss the limitations of prior work in achieving highly concise storage, and make the case for an asymmetric compute and storage technique.

A. Limitations of Prior Work

Lossless cache compression techniques [9, 10, 11, 12] focus on removing redundant bits by reducing the incidence of replicated values in last-level caches (LLCs). These approaches are designed to work with fixed-point and integer programs. However, cache compression has been shown to achieve negligible compression ratios for floating-point data because floating-point data lacks the value-level replication

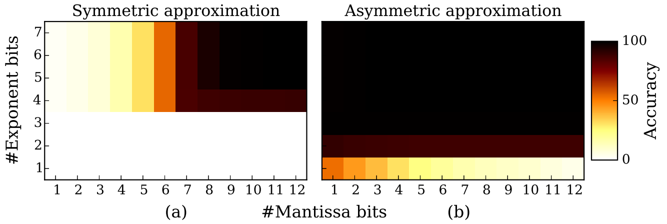


Fig. 2. Accuracy comparison between (a) symmetric approximation and (b) asymmetric approximation for Kmeans, showing that asymmetry achieves same accuracy with significantly fewer bits as compared to symmetric approach

that is often found in integer and fixed-point data [9, 13]. Others have explored extending the definition of replication to include softer definitions of replication, treating LLC lines of similar floating point data as replicas [13]. These techniques achieve better compression for floating-point values than lossless compression techniques. However, as we show later in Section V, these softer definitions of replication still leave large numbers of marginal bits in cache. Moreover, the narrow focus of prior work on last-level cache only partially addresses this problem, leaving all data in place in private caches and DRAM.

In addition, different applications need different numbers of bits to achieve satisfactory accuracy. Therefore, the design of a concise storage approach needs to have the flexibility to capture the wide spectrum of design points required by different applications and design objectives. Current architectural designs that include support for `double`, `float` and (occasionally) `half` precision floating-point configurations are of limited applicability, as they do not capture a rich enough range of options and leave a significant opportunity on the table. Moreover, recent prior work focused on building approximate storage structures also does not provide sufficient flexibility because its approximation settings are built into the hardware at design time [13].

To address the limitations of prior techniques, our approach uses custom-precision floating-point formats. In our approach, each number still has sign, exponent and mantissa fields, however the number of mantissa and exponent bits are not fixed. This makes our approach highly flexible, providing a rich spectrum of design points with different numbers of mantissa and exponent bits to choose from, resulting in a highly concise storage. Our approach is fundamentally different from previous works [9, 10, 11, 12, 13] as it identifies marginal bits by carefully characterizing the impact of bits in the data elements, while the previous works apply softer definitions of data replication across LLC lines, missing the opportunity to remove all the marginal bits.

B. The Problem with Asymmetry

One might posit that concise storage could be achieved via a system using custom precision formats that is *symmetric* in compute and memory, using a concise format in both memory and compute. However, there are two reasons that make such an approach impractical.

First, it would be extremely invasive and hardware-intensive, requiring major changes to the functional units, pipeline, datapaths and so forth to support using concise data formats throughout.

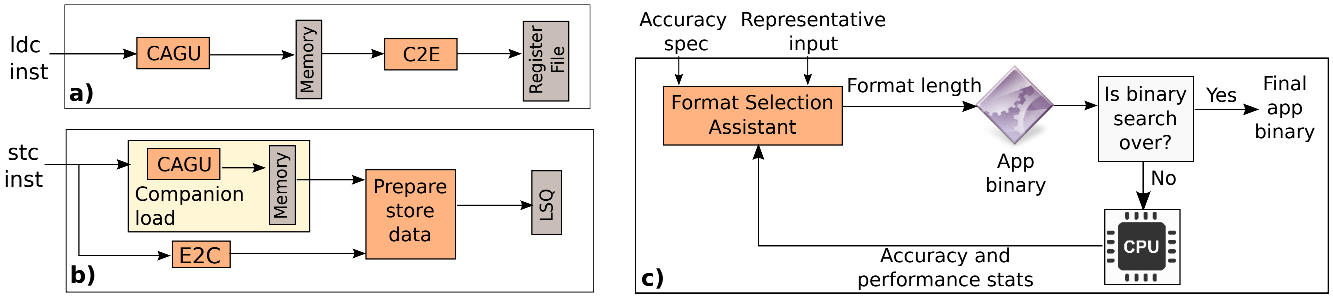
Second, we have observed in our experiments that a symmetric approach tends to lose accuracy very quickly as the number of bits in the data format are reduced. This is illustrated in Figure 2(a) and (b), which show the result accuracy of running Kmeans using symmetric and asymmetric approaches, respectively, for a range of different exponent and mantissa lengths. The asymmetric approach stores data concisely throughout the memory hierarchy while performing computation at full precision. Each plot shows the accuracy of a range of different formats, where darker colors indicate higher accuracy results. The key observation is that the asymmetric approach can achieve a particular level of accuracy with far fewer bits. Value saturation causes steep dropoffs in accuracy when reducing the number of bits in the symmetric approach (e.g., going from 4 to 3 exponent bits in Figure 2a). Such reductions in the number of bits reduce the range of values supported by the functional units, frequently leading to saturated intermediate and output values and highly inaccurate computation. For example, the symmetric approach requires 15 bits to achieve 99% accuracy, while the asymmetric approach requires just 5. This trend holds true across applications – on average across 10 test applications, we find that the symmetric approach requires 1.7× as many bits as the asymmetric approach to attain 99% accuracy.

C. Bridging the Format Divide

An asymmetric approach has significant benefits over a symmetric approach in terms of hardware simplicity and accuracy, but there remains one main difficulty to solve to enable the asymmetric approach – *bridging the format divide* by converting between precise and concise data formats at the boundary of the memory hierarchy.

An obvious way to perform these conversions to extract precisely formatted data from concise data is to leverage existing software mechanisms such as shifts, masks and other operations. Such an approach would work by loading concise data using conventional memory operations, then convert and distribute it (potentially across multiple registers) by shifting, masking and other bit-level operations. The main difficulty making software conversion approach impractical is that many such operations may be needed per memory operation, introducing significant amounts of additional processing overhead to support concise storage.

While such an approach may reduce capacity and bandwidth requirements in the memory hierarchy, through experimentation (not shown here) we have observed that it significantly undermines the ability of the approach to improve application performance on net, often introducing non-negligible slowdowns due to the cost of converting data every time it is loaded. This suggests that the key to enabling an effective



Hardware support: Executing concise loads and stores

Software support: ACME compiler

Fig. 3. ACME system architecture; a) ACME compiler finds a suitable precision for the application and produces concise loads and stores for the annotated variables. b) and c) show execution of these concise loads and stores in hardware.

approach to leveraging an asymmetric compute-memory approach lies in efficiently bridging the format divide.

III. Overview of ACME

ACME is designed to address these problems. ACME is based on an asymmetric compute-memory architecture; the data is stored concisely in memory while computation happens on full precision. ACME reduces pressure on the memory subsystem exploiting marginal bits to reduce the cost of storing and moving data.

A. Challenges

However, there are several challenges in converting these savings in memory storage and bandwidth to performance improvements.

Quick Format Conversion. ACME is based on an asymmetric compute and storage paradigm, resulting in a format divide between compute and memory. Therefore, each concise load requires conversion from the concise format to the single precision format. Similarly, each concise store requires conversion from the single precision format to the concise format to bridge this format divide. These conversions must be fast to extract maximum performance benefit from the concise storage.

Bit-level Interactions in Byte-addressable Memory. Achieving highly concise storage requires storing values of arbitrary length in the memory. This gives rise to situations in which the concise data element might not start at a byte boundary. Since conventional memory subsystem is byte-addressable, ACME needs to support certain bit-level interactions in a byte-addressable memory environment.

Choosing Precision. Different applications have varying accuracy requirements, and thus varying format requirements. Finding a suitable precision requires navigating through a non-trivial search space (`23` mantissa * `8` exponent = `184` for each variable). Therefore, ACME requires quickly finding the right level of precision for the application.

B. Key Components

We introduce these components to address the challenges outlined earlier.

Fast Conversion Units. ACME introduces two small additional units, *Concise to Exact (C2E) and Exact to Concise (E2C)*, to bridge the format divide between compute and storage. These units perform format conversions in a single cycle. The C2E unit converts the concise data element into single precision format before writing it into the register file. Similarly, the E2C unit converts the data element format from single precision format to concise format before sending it to memory.

Concise Address Generation Unit. ACME uses a Concise Address Generation Unit (CAGU) to calculate the memory address of concise data elements. Our approach keeps the memory byte-addressable. CAGU generates a byte-level memory address that is closest preceding to the concerned concise data element. It works in concert with the E2C and C2E to access the memory response at a bit-level granularity.

Format Selection Assistant. ACME employs a Format Selection Assistant (FSA) to find an appropriate format for an application. For a specified accuracy target, ACME performs a binary search over the number of exponent and mantissa bits to quickly identify a suitable precision for each approximated variable.

ISA Support. We propose two ISA extensions in the form of *load-concise* (`ldc`) and *store-concise* (`stc`) instructions. These instructions support arbitrary length storage in the memory hierarchy, leveraging the CAGU, E2C and C2E units to realize the asymmetric compute and storage architecture.

IV. Design and Implementation

ACME is an end-to-end system that stores data concisely by removing marginal bits while performing computations at full precision, an approach that improves performance of memory-intensive applications by increasing effective cache size and effective memory bandwidth. In this section, we describe the details of the ACME system architecture.

A. System Architecture

Figure 3 illustrates a high level overview of ACME. illustrating the hardware support (left) and the software support (right).

Hardware Support. Concise loads and stores are supported in the hardware via the CAGU, E2C and C2E units. For the `ldc` instruction, as shown in Figure 3a, the processor sends a load request for the memory address generated by CAGU.

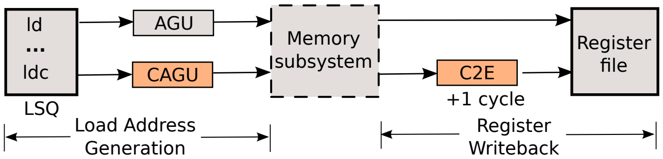


Fig. 4. Execution of exact and concise loads

The data response is passed through the C2E unit to convert the data element format from concise to single precision, before writing it into the register file. For `stc` instructions (Figure 3b), the processor first performs a *companion load* to find the data contents at the requested memory address. In parallel to the companion load, the processor removes marginal bits from the store value using the E2C module, converting the data element format from single precision to concise. The concise data is then inserted at the appropriate location in the companion load response, which is later written to the load-store queue (LSQ).

Software Support. The ACME compiler allows the programmer to annotate those variables that are amenable to approximation, as is done in prior work [15, 16, 17, 18, 19]. In ACME, these take the form of `#pragma` directives in order to ensure the compatibility of ACME-enabled code with NON-ACME compilers. The ACME compiler takes the annotated application, an accuracy specification and a representative input dataset as input and generates an application executable that uses `ldc` and `stc` to enable concise storage. As illustrated in Figure 3c, the compiler generates `ldc` and `stc` for the annotated variables with the precision information (format length) as instructed by FSA. The resulting executable is then profiled and the accuracy and performance statistics are sent to FSA. FSA uses this information to decide the format length of the next step of binary search. In addition, the ACME compiler provides a `cmemcpy` (concise `memcpy`) function that uses concise memory operations to remove marginal bits from the approximated input variables, after the variables have been initialized.

B. Hardware Execution

ACME uses `ldc`/`stc` instructions to enable precise computation on concise data elements. These instructions reuse most of the existing processor micro-architecture with the help of three small additional hardware units - CAGU, C2E and E2C.

1) Execution of Concise Loads: Every load instruction (with or without ACME) has 2 steps as shown in Figure 4 – i) Load address generation, where Address Generation Unit (AGU) calculates the effective address to be sent to the memory, and ii) Register file writeback, where the data response from the memory is written back into the register file. ACME introduces additional hardware units in both of these steps to bring concise data elements into the processor and convert them to the single precision format.

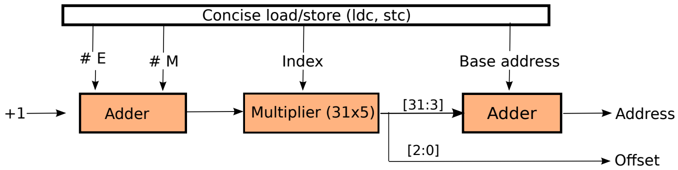


Fig. 5. Block diagram of Concise Address Generation Unit (CAGU)

Load Address Generation. Conventional processors have dedicated functional units to calculate the effective memory address for loads and stores. These functional units are called Address Generation Units (AGU). By adding dedicated AGUs, memory instructions do not use integer ALUs for address generation, creating opportunities for executing more independent integer instructions in parallel. The compiler encodes the necessary information to perform address generation into the memory instructions while generating the application binary. This information is extracted by the instruction decoder and passed on to the AGUs. For example, in x86, array traversal uses a base register, index register for the array and the data size of each element (in bytes). In this case, the AGU performs the following integer arithmetic operation to generate the effective memory address: `(base_register) + data_size * index_register`.

However unlike conventional loads, ACME requires the capability of storing a data element of any arbitrary length in the memory, breaking the assumption that data elements are byte-aligned. This gives rise to situations where ACME requires bit-level access while the memory is byte-addressable. ACME solves this challenge by introducing the CAGU and C2E unit, allowing bit-level access in the data response of the concise loads while the caches and memory remain byte-addressable. These units thus serve as a transparent layer between the processor and the memory where everything else is byte-addressable by design while ACME has bit-level access in the data response.

To accomplish this, the CAGU generates a byte-level memory address and a bit-offset to completely specify the address of a concise data element. The byte-level memory address is the closest byte preceding the requested concise data element. The bit-offset is the number of bits that are present between the above byte-address and the concise data element location. As shown in Figure 4, the CAGU first sends this byte-level address to the memory. The C2E unit then extracts the relevant bits from the data response using the bit-offset, converts them into the single precision format and stores the final 32-bit value into the register file.

Figure 5 illustrates the design of the CAGU. The instruction decoder extracts the precision information (the number of exponent and mantissa bits) from concise loads and stores to calculate the length of the concise format. CAGU multiplies the length of the concise format with the index register. Since the maximum length of a concise data element is 31, the format length can be encoded using 5 bits. Therefore, the above multiplication requires a 32x5 (for the index register

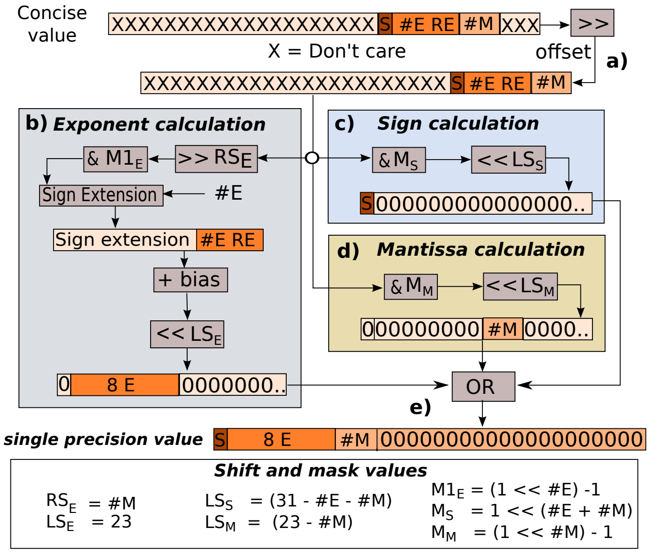


Fig. 6. Block diagram of Concise to Exact (C2E) unit

and format length, respectively) integer multiplication unit. This intermediate value is the number of bits between the base address and the concise data element. Therefore, masking off the last 3 bits of this value results in a byte-level memory address which is closest byte-level memory address preceding the requested concise data element. Moreover, the least significant 3 bits of the intermediate value form the bit-offset, i.e. the number of bits to ignore in the memory response to get to the requested data.

While sending the concise load request, the CAGU also sends the bit-offset and the precision information along with the request. These are required later by the C2E unit to extract the relevant bits from the data response. If a cache miss happens, then the bit-offset and the packing information gets stored in the MSHR entries. The bit-offset requires 3 bits and the packing information requires $\lceil log_2(8\ exponent \times 23\ mantissa)\rceil = 8$ bits of storage. Therefore, each MSHR entry needs extra 11 bits of storage.

Note that exact loads also go through AGUs which have their own integer arithmetic units. Therefore, the CAGU does not add to the critical path of the processor for non-concise memory operations. We synthesize and report the timing characteristics of the CAGU in Section V.

Register File Writeback. On receiving a concise load memory response, the C2E unit extracts the relevant bits using the bit-offset and precision information present in the memory response. The data is converted to the single precision format and stored into an intermediate register before being written to the register file. In the next cycle, ACME performs a lookup on the LSQ to find the destination register and performs a writeback into the register file.

Since each concise load performs format conversion from concise to single precision, this conversion has to be fast to provide the maximum performance benefits of concise storage. We introduce a Concise to Exact (C2E) unit to address this challenge. It converts the concise data into the single precision format in a single cycle. Figure 6 gives a step-by-step walk-through of this conversion process in the C2E unit. The process can be broken down into 5 steps – a) the C2E unit shifts the data response by bit-offset to align the relevant bits at the end, b) it masks and shifts this value to get the sign bit at the right position, c) similar operations are performed to put mantissa bits at the right position, d) the concise data has a raw (unbiased) exponent. This raw exponent is extracted, sign-extended and a bias of 127 is added to it to calculate the final exponent value. This exponent is then shifted and put at the correct position e) lastly, the C2E unit performs a logical OR operation on the sign, exponent and mantissa portions to get the final value in the IEEE floating-point format. This final value is written to an intermediate register.

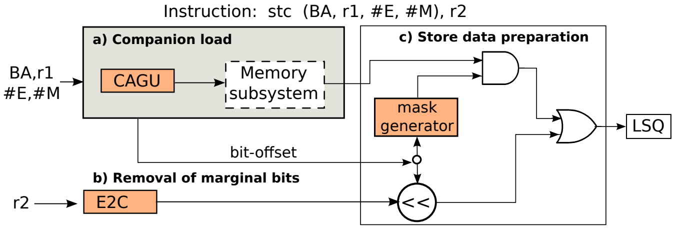

Instruction: stc (BA, r1, #E, #M), r2
a) Companion load
CAGU
Memory subsystem
BA,r1
#E,#M
c) Store data preparation
mask generator
LSQ
bit-offset
b) Removal of marginal bits
r2
E2C
<<

Fig. 7. Execution of concise stores

In the next cycle, an LSQ look up is performed to find the destination register and the data is written back into the register file. From this point, the data is in single precision format and the computation happens precisely.

2) *Execution of Concise Stores:* Supporting arbitrary length concise stores in hardware is challenging because concise stores require partial byte modifications, while memory is typically byte-addressable. Concise loads solve this problem by reading the memory first and performing bit manipulations later. However, concise stores need to preserve parts of a byte in memory while modifying another part of the byte.

We solve this problem by performing a *companion load* to the relevant memory location alongside every concise store. This data returned by the companion load is then used to prepare the final store data to be written back to the memory. In our experiments, we have observed that extra companion loads have minimal performance impact, as they are greatly outnumbered by concise and conventional loads.

Concise store execution can be broken down into 3 steps as shown in the Figure 7 – a) Performing a companion load, b) removing the marginal bits from the register value, and c) preparing the store value.

Companion Load. Concise stores perform a companion load using the CAGU as shown in Figure 7a. This is performed to keep track of the bits (other than the required data element) that need to be preserved at the time of storing in the memory.

Removal of Marginal Bits. In parallel to companion load execution, the register value that needs to be written to the memory is stripped of its marginal bits using the Exact to Concise (E2C) unit. Figure 8 gives a step-by-step walk-through of this process – a) The register value is first rounded.

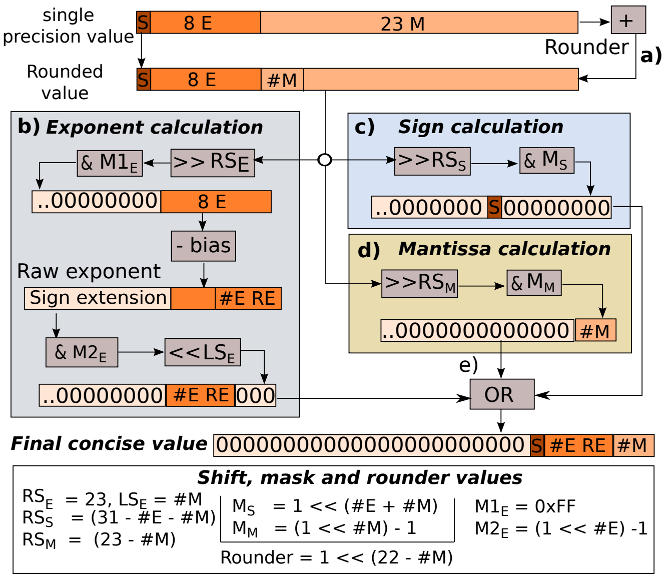


Fig. 8. Block diagram of Exact to Concise unit

This rounded value is used to find b) exponent, c) sign and d) mantissa portions separately which are then e) logically ORed to generate the concise value. Intuitively, these calculations are reverse of C2E calculations described earlier. In case the register value is beyond the range supported by current precision, it is clamped at the format-supported maximum/minimum value, whichever is closer. For representing value 0, we set all the bits in the concise format to 1.

Store Data Preparation. Finally, ACME prepares the store value to be written back to the memory as shown in Figure 7c. ACME first left shifts the concise value by the bit-offset and brings it to the right position. The data response from the companion load is masked at the bit-locations that are going to be written by the concise value. These two values are then logically ORed. This value is then written to the LSQ. Finally, the value in the LSQ is written to the memory on instruction commit.

C. Software Support

ACME provides the flexibility to handle many levels of approximation by adding concise loads and stores; load-concise (`ldc`) and store-concise (`stc`) instructions. The ACME compiler is responsible for generating these concise loads and stores for the annotated variables. These instructions support storage of the concise data in memory with precise computation using the CAGU, E2C and C2E units. In addition, the compiler adds support for a `cmemcpy` function to remove marginal bits from the input dataset in the application code .

ISA extension. We use x86 assembly instruction `movl` to explain the workings of the `ldc` and `stc` instructions, though the idea can be extended to other ISAs as well. Consider the following load and store instructions:

```
movl  (%ebx, %esi, 4), %eax ## Load
movl  %eax, (%ebx, %esi, 4) ## Store
```

For traversal of an array, these memory operations use i) base address (`%ebx` in this example), ii) index register (`%esi`), and iii) data size (`4`). Since the base address and index are not known at the compile time, the memory address calculation (`%ebx + %esi * 4`) happens in the AGU at runtime.

Concise memory operations differ from their exact counterparts in the data size field. Here, compiler encodes the number of exponent and mantissa bits as instructed by FSA ($23 \times 8 = 184$ combinations, `8` bits). For example

```
ldc  (%ebx, %esi, #E_#M), %eax
stc  %eax, (%ebx, %esi, #E_#M)
```

In hardware, this precision information is extracted by the instruction decoder and passed on to the CAGU to perform memory address calculations.

Concise Memcopy Function. ACME requires a mechanism to remove the marginal bits from the annotated variables. There are several ways to perform this removal – directly converting the input data into concise format while initializing the approximated variables, or performing removal after the initializations are complete. We take the latter approach because it enables us to carefully evaluate the impact of removing marginal bits on the application speedup.

The ACME compiler adds support for a `cmemcpy` function that can be applied in the application code just after the variable initializations complete. All the variables are in IEEE format just after the initialization. The `cmemcpy` function is a simple loop that makes a pass over the annotated array, creating an in-place (smaller) concise copy of the data using concise store operations. In this way, the annotated input data elements are now stored concisely, fitting more elements in the memory hierarchy. In Section V, we experimentally show that the overhead of applying `cmemcpy` is very small (<1% of application execution time).

D. Format Selection Assistant

ACME uses highly flexible ISA extensions providing a wide spectrum of precision configurations to choose from. Different applications have varying precision requirements, resulting in different number of marginal bits. ACME requires finding out this precision requirement for an application at a specified accuracy target. This requires navigating through a search space of precision configurations consisting of `23` mantissa * `8` exponent = `184` options for each annotated variable.

We introduce a Format Selection Assistant (FSA) to help ACME in quickly finding this suitable precision level. It takes an application, a set of representative inputs, an error metric and an error bound (i.e., the maximum error an application can tolerate). It generates the minimum number of bits (e.g., number of exponent and mantissa bits for floating point numbers) to represent the input.

Tuning Algorithm. This approach leverages the observation that the accuracy of an application in asymmetric storage and compute will typically monotonically increase with length of exponent and mantissa bits. This enables us to leverage a greedy binary-search based approach to reduce the complexity

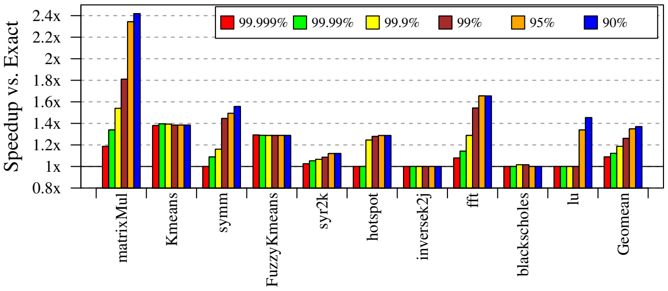


Fig. 9. ACME performance benefits. ACME achieves good speedup for memory-bound applications

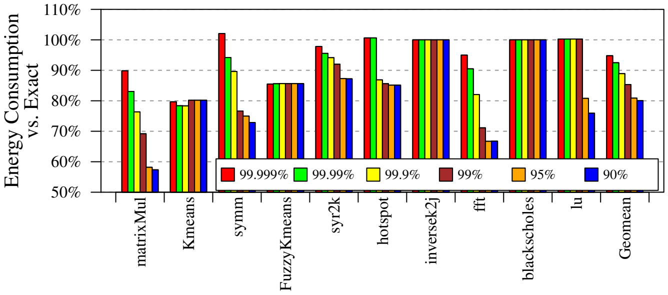


Fig. 10. ACME energy benefits. ACME provides significant energy savings for memory-bound applications

of the accuracy space exploration. The algorithm is greedy because it finds a suitable precision configuration for the first variable while keeping others exact, then it fixes the precision of first variable and moves on to the second variable while keeping the others exact, and so on. In this way, this algorithm finds suitable precision for each variable one-by-one.

We use the intuition that exponent is typically much more important than mantissa for mathematical operations. Thus, we explore the exponent values first, using the maximum number of mantissa bits. For 32-bit IEEE floating point numbers, we start with 4 bits of exponent with 23 bits of mantissa. Once we determine the number of exponent bits using binary search, we again perform binary search over the length of mantissa bits. For each variable, this will require at most $\lceil log_2(8) \rceil + \lceil log_2(23) \rceil = 8$ executions instead of $8 \times 23 = 184$ executions in exhaustive approach. The Format Selection Assistant (FSA) can also be configured to apply a single format to all concise variables in the application, where the search occurs over 1 variable and the formats of all variables are kept in lockstep throughout the tuning algorithm. This reduces the search space significantly at the cost of some reduction in data conciseness, a tradeoff we evaluate in Section V-C.

The final precision configuration at the end of the binary search is used for approximating the application. In addition to accuracy, this exploration also records performance of different precision configurations. In case the approximation results in a performance degradation compared to the exact execution, FSA instructs the compiler to drop the approximation.

V. Evaluation

A. Methodology

Applications. We evaluate ACME across 10 applications. We use matrixMul, symm and syr2k from PolyBench [20], Kmeans, FuzzyKmeans, inversek2j, fft and blackscholes from AxBench [16] and hotspot and lu from Rodinia benchmark suite [21]. These floating point applications are at the core of emerging machine learning and data mining workloads, having a mix of compute-bound and memory-bound applications and thus presenting a wide spectrum of program characteristics for evaluating ACME.

Accuracy Measurement. We use *average relative error* [13, 16, 17] as the error metric for our applications. Average relative error can be calculated using following equation, where v_i is the exact value and v_i^* is the approximated value.

$$AverageRelativeError = \left[\sum_{i=1}^{N} |v_i - v_i^*| / v_i \right] / N$$

Performance and Energy Measurement. We evaluate the performance of ACME on Gem5 simulator [22]. We extend x86 ISA support in Gem5 by adding load-concise and store-concise instructions. We also add functional and timing models of ACME hardware components, CAGU, C2E and E2C units. A penalty of one cycle is added to concise loads and stores to account for conversion latency as detailed in Section V-E.

Processor	8-wide OoO core, 3.0 GHz 192-entry ROB, 72-entry load queue
Private L1 cache	32 KB, 8-way, 2-cycle, 64 B block
Private L2 cache	256 KB, 8-way, 5-cycle, 64 B block
Shared LLC	2 MB, 16-way, 12-cycle, 64 B block
Main memory	1 GB, 200-cycle latency
L1 prefetcher	Tagged prefetcher
L2 and LLC prefetcher	Stride prefetcher

TABLE I. Hardware configuration

Table I lists the specifications of the relevant hardware components that are configured to model an Intel Haswell processor. The applications are simulated for 5 billion instructions or to completion whichever is sooner. For measuring energy, we use McPat [23] and CACTI [24] to calculate the static and dynamic energy of core, caches, DRAM and ACME hardware units.

FSA Testing and Training. We partition the inputs into training and testing sets for all applications. We use the FSA to identify a suitable precision for the application on the training set and then used the same precision on the testing set. We found that the precision obtained from training satisfied the accuracy targets during testing. Unless otherwise noted, our experiments configure the FSA to use a single format for all application variables. The impact of using single- and multi-format configurations is explored in depth in Section V-C.

B. Performance and Energy Benefits

In this section, we evaluate ACME performance and energy tradeoffs for six accuracy targets. For each accuracy target, the number of exponent and mantissa bits is determined by the

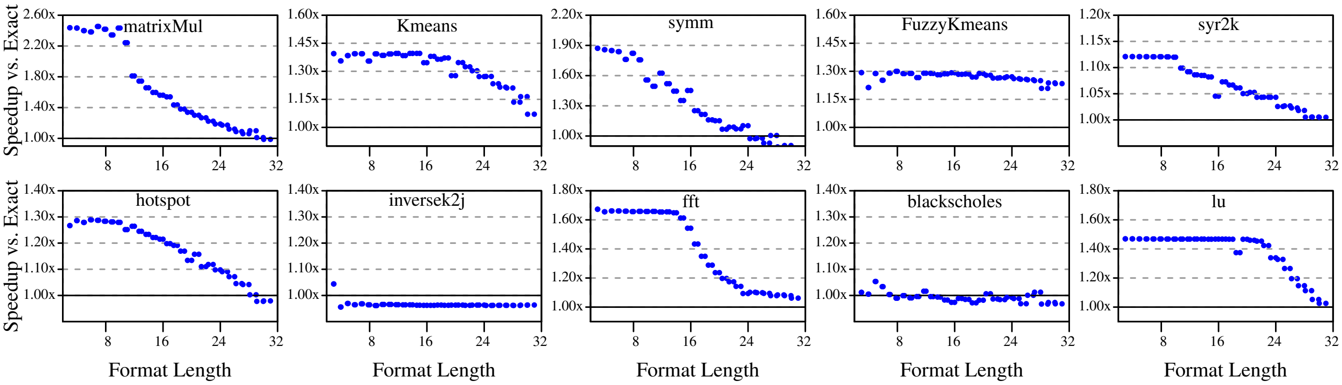


Fig. 11. ACME performance study with varying format length. Smaller length yields less cache and memory pressure, resulting in higher application speedup

FSA. We use this precision configuration to find the speedup and energy savings compared to the exact execution.

Performance Accuracy Tradeoff. The performance-accuracy tradeoffs are shown in Figure 9. The figure shows the speedup of ACME against exact execution carried out on a non-ACME hardware for six accuracy targets. We observe significant speedup for applications that can benefit from larger caches. This occurs because ACME removes marginal bits from the memory subsystem, fitting more data elements into the lower level of memory closer to the processor. As one might expect, speedup goes up with looser accuracy constraints. Nevertheless, ACME gets speedup of 10% while attaining 99.999% accuracy. This is possible because some applications have large number of marginal bits whose contribution to the application accuracy is minute. For compute-bound applications, the FSA chooses exact execution, as reducing the data representation size has minimal impact on performance. For an accuracy target of 99%, ACME achieves a speedup of 1.8x for matMul, with an average of 1.3x for the whole application suite.

Energy Accuracy Tradeoff. Figure 10 presents the energy-accuracy tradeoffs of the same experiment. The figure shows the total energy consumed during the ACME execution compared to exact execution for six accuracy targets. Again, we observe that memory-bound applications consume lower energy compared to the exact execution. There are 2 reasons for this improvement. First, the application finishes sooner, leading to reduced static energy, and second, ACME reduces the number of DRAM requests leading to lower dynamic DRAM energy. ACME hardware components are small and consume minimal amount of energy. For an accuracy target of 99%, ACME reduces the energy consumption to 85% energy of the non-ACME hardware on average.

Impact of Format Length. We next carry out a detailed performance evaluation of ACME with varying number of bits. The experimental setup consists of executing an application with different format lengths (number of bits used to represent a data element). For a particular format length, we can have different configurations of exponent and mantissa bits. The graph presents the one with the highest accuracy. The results of this experiment are presented in Figure 11.

ACME is able to achieve significant speedup for all memory-bound applications with small format lengths. Due to increased effective memory capacity and bandwidth, we observe higher speedup for smaller format lengths. These improvements outweigh the clock-cycle penalty of the C2E unit. With larger format lengths, the benefit of storing data concisely diminishes and extra clock cycle penalty by C2E becomes more prominent. For example for application symm, ACME achieves good speedup for small format lengths that use <16 bits but shows slight performance degradation for larger format lengths >24 bits.

We also observe that a few of the data points do not follow the speedup trend. For example, Kmeans at length = 20 and syr2k at length = 16 . This happens because mapping of data elements to physical cache lines changes with format length. A particular strided-access pattern for a certain format length can cause relatively more conflict misses than the adjacent format lengths. We observe abrupt increase in the number of misses for a certain cache for such format lengths. This is a well-studied cache effect [25].

Finally, as expected ACME does not improve performance for compute-bound applications: blackscholes and inversek2j. Blackscholes has minimal performance degradation because it has good ILP to keep its pipeline busy hiding the cycle penalty induced by the C2E unit. This is not the case in inversek2j, where the C2E penalty delays execution of dependent instructions, resulting in higher degradation. However, we note that the FSA recommends not using concise types for these applications, and thus these applications do not slow down when compiled with ACME compiler.

The experiment demonstrates ACME's ability to improve the memory behavior of applications resulting in significant speedup and energy improvements for applications sensitive to cache and memory performance.

C. Format Selection Assistant

In this section, we show details of FSA-chosen concise format for different accuracy targets, shown in Figure 12. The figure shows the breakdown between the number of exponent

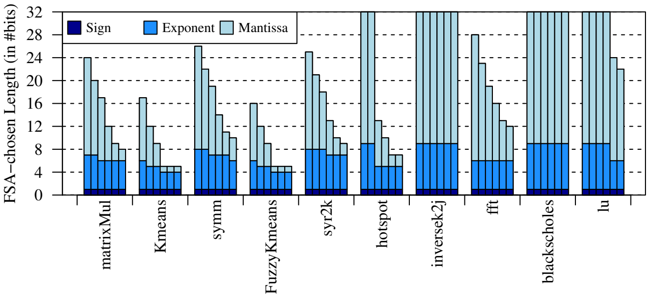


Fig. 12. Breakdown of FSA chosen representation length for six accuracy targets - (left to right) 99.999%, 99,99%, 99.9%, 99%, 95%, and 90%

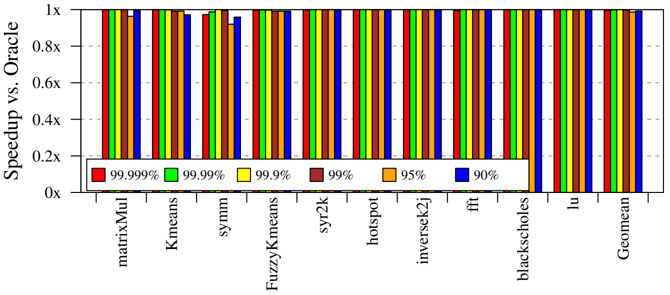


Fig. 13. Comparison of FSA-chosen configuration performance against an oracle format selector. The FSA achieves $> 98\%$ of oracle performance on average

and mantissa bits. We always keep the sign bit in the concise format.

We make 2 key observations from these results. First, the same application has different number of marginal bits for different accuracy targets. For example, matrixMul needs 8 bits for 90% accuracy but 24 bits for 99.999% accuracy. Second, different applications have different number of marginal bits for the same accuracy target. For example, Kmeans achieves 99% accuracy with just 5 bits whereas lu needs all 32 bits to achieve 99% accuracy. The results effectively demonstrates the need of designing a *flexible* approximation approach in order to get the desired accuracy targets.

For compute-bound applications, blackscholes and inverske2j, FSA chooses exact 32-bit representation for all accuracy targets.

Comparison to Oracle. We next compare performance achieved by the FSA configuration against an oracle system that finds the best precision configuration for the application by performing an exhaustive search over all the representation formats. The findings of this experiment are shown in Figure 13. For most of the applications, the accuracy increases and performance decreases with increasing the number of exponent and mantissa bits. Therefore, the greedy binary-search heuristic achieves performance close to the oracle in most cases. But as explained previously, some precision configurations result in relatively more conflict misses which results in sub-optimal performance compared to the oracle.

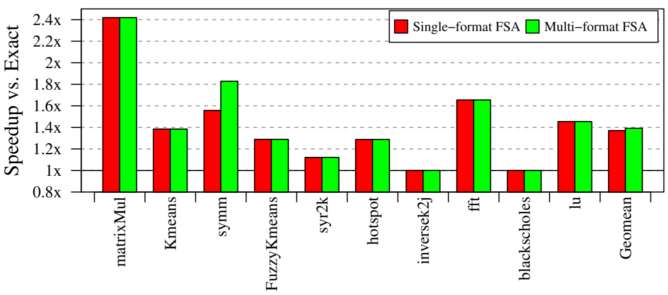


Fig. 14. Comparison of speedup between single-format FSA and multi-format FSA

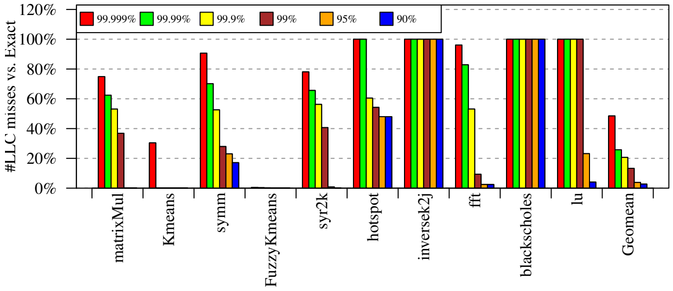


Fig. 15. ACME reduces #off-chip memory requests which is a major source of speedup

Overall, FSA is able to achieve $> 98\%$ of the optimal speedup for all accuracy targets.

Different Formats Across Variables. The ACME hardware and compiler support using different formats among the different variables in an application. However, using different formats increases the complexity of the FSA tuning algorithm and thus increases compilation time. Here we evaluate the impact on performance of using a *multi-format* approach in the FSA. We allow the FSA to select formats among all applications in both multi-format and single-format modes at a 90% accuracy target, presenting our findings in Figure 14.

We observe that multi-format FSA precision settings provide minimal performance benefit on most applications. Kmeans, FuzzyKmeans and lu have only one variable suitable to approximation, and thus do not see any additional performance benefit when using multi-format mode in the FSA. For the compute-bound applications blackscholes and inversek2j, the FSA chooses exact execution in both multi-format and single-format mode. For 4 the remaining 5 applications that have multiple variables and are not compute bound, we observe negligible performance improvements when using the multi-format FSA. This occurs because, while the working set size may be somewhat improved by using the multi-format FSA, it often fails to reduce the footprint by enough to fit the application working set into a closer cache level. The single case where we observe a significant preformance improvement is for symm, where such a reduction occurs.

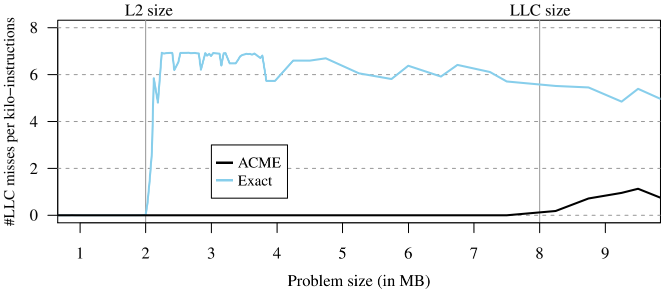


Fig. 16. LLC misses when varying matrixMul working set size with exact and ACME execution

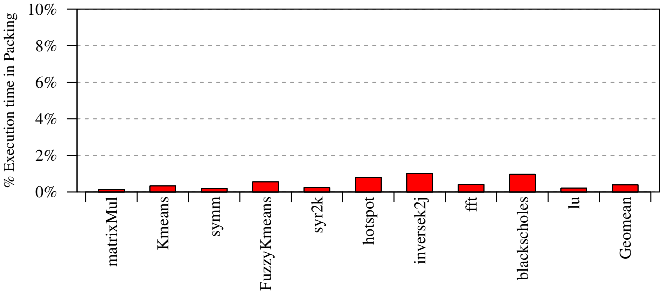


Fig. 17. Overhead of cmemcpy function. The function consumes minute portion of total application execution time

D. Memory Behavior

ACME achieves concise storage by removing the marginal bits throughout the memory subsystem. This results in an increase in effective capacity and bandwidth, improving performance. A major source of speedup comes from reduction in LLC misses. LLC misses are expensive as processor has to wait for DRAM to satisfy the miss. In this section, we perform experiments to understand how ACME impacts memory behavior.

LLC Miss Reduction. We compare the LLC Misses for FSA-chosen configuration for six accuracy targets against exact execution, presented in Figure 15. As expected, ACME brings down the number of LLC misses substantially, which is one of the major causes of performance improvement with ACME. On average, ACME reduces the number of LLC misses by 85% at an accuracy target of 99%.

Impact of Working Set Size. In this experiment, we perform a detailed study on matrixMul with varying working set problem sizes. The experimental setup consists of running exact and ACME version of matrixMul with different problem sizes and then measuring the effect on IPC and LLC misses. The format length chosen for the concise storage is 8 bits which enables us to fit 4 times as many elements in memory-subsystem as compared to exact. The results of this study are shown in Figure 16 where the problem size varies from 1 MB to 9 MB.

When the the problem size is less than 2 MB (the size of our LLC), both the exact and approximate data fits into LLC. Therefore, the number of exact and approximate LLC misses are similar resulting in similar performance for exact and ACME execution. However, as the exact problem size goes beyond 2 MB, we start seeing larger number of exact LLC misses. ACME is still able to fit the data in LLC because it is using only 8 bits to represent the input elements. It is only for configurations larger than 8 MB that ACME begins to introduce increasing numbers of LLC misses.

E. System Overheads

In this section, we discuss the overhead associated with different components of ACME. Note that all these overheads are already included in other parts of the evaluation.

Packing Overhead. ACME compiler adds a `cmemcpy` in the application code to represent the input elements more concisely. Figure 17 shows the portion of application execution time spent in `cmemcpy` function. We see that this overhead is <1% in all the applications. Our hardware implementation removes the marginal bits by performing complex conversions quickly in the hardware, resulting in a minimal overhead. We also implemented a software implementation of store-concise instruction and used it for the `cmemcpy` function. However, we observed as much as 10% overhead with the software implementation, resulting in reduced performance improvements.

Hardware Overhead. In this section, we discuss area, power and frequency numbers for the additional hardware components. We implement CAGU, C2E and E2C unit in Verilog and synthesize it using ARM Artisan IBM SOI 45 nm library. The area, power and frequency of the C2E unit is $0.0034\,\text{mm}^2$, 9.41 mW and 2.78 GHz respectively. Similarly, the numbers are $0.0023\,\text{mm}^2$, 4.23 mW and 2.78 GHz respectively for the E2C unit, and $0.0044\,\text{mm}^2$, 12.7 mW and 2.22 GHz respectively for CAGU. Our baseline is a mainstream core-i7 Haswell processor that operates at a frequency of 3.0 GHz and consumes $177\,\text{mm}^2$ of die area. We see that the additional overhead of ACME units is minimal: 0.0052% area overhead and <0.1% power overhead. By using technology scaling trends [26] to project the frequency for hardware components for 22nm, we find that ACME units can operate at the target frequency of 3 GHz at 22nm. This study shows that additional hardware components are fast and consume minimal area and power.

F. Comparison to Prior Work

In this section, we compare ACME against a state-of-the-art approximate computing cache technique; Doppleganger [13]. Doppleganger increases effective LLC capacity by finding LLC lines that are similar. Approximately similar cache lines are mapped to single line resulting in increase in effective cache size.

Doppleganger finds approximately similar cache lines by encoding the range and average of the values present in the cache lines. This encoding takes form of an N-bit hash map. Two cache lines are treated approximately similar if they produce same map value. Lower the value of N, higher is the compression ratio at the expense of higher application error.

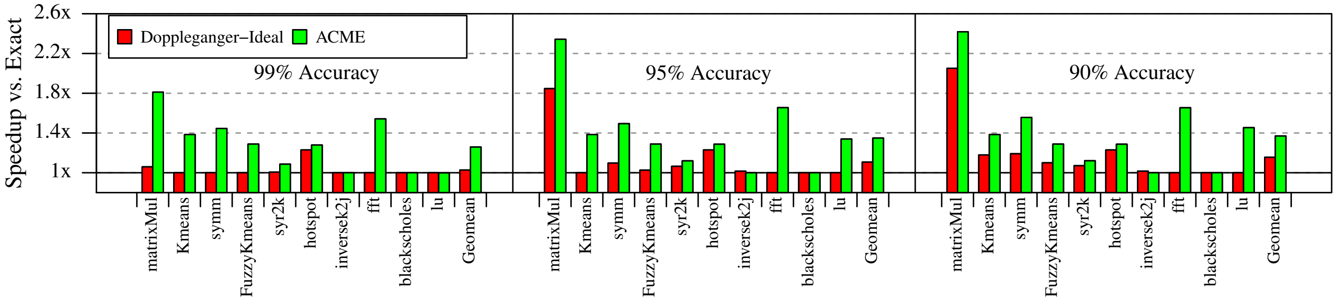


Fig. 18. ACME vs Doppleganger-Ideal; ACME achieves higher concise storage throughout the memory hierarchy resulting in better application speedup

Doppleganger builds this N-bit hash function into the hardware at design time, preventing any accuracy knob. Consequently, Doppleganger might not be able to satisfy an accuracy target with a N-bit hash function. We create an idealized version of Doppleganger, Doppleganger-Ideal, where it is not restricted by a fixed value of N. Instead, it finds the minimum value of this N for each application and accuracy target separately. This lets us measure the approximate similarity in the application which is equivalent to the magnitude by which the effective LLC size is increased. To simulate this effective increase in LLC size for Doppleganger-Ideal, we increase the actual size of LLC as per the measured similarity without increasing the cache latency.

The comparison between ACME and Doppleganger-Ideal is presented in Figure 18. Doppleganger-Ideal shows speedup for some applications for 90% and 95% accuracy but its speedup drops significantly for 99% accuracy. We see that ACME performs better than Doppleganger-Ideal in all the applications, except inversek2j for accuracy target of 90%. There are 2 reasons for this performance difference. First, ACME achieves more concise storage compared to Doppleganger-Ideal. Doppleganger is limited by finding redundancy across cache blocks. ACME, instead, finds the bits that marginally contribute to the accuracy and removes them from the data representation. Second, ACME achieves concise storage throughout the memory hierarchy, compared to Doppleganger-Ideal which operates only on the LLC.

VI. Related Work

One common way to reduce application cache footprint is using cache compression. The majority of cache compression techniques strives to reduce value replication in the memory subsystem [9, 10, 11, 12]. However, these cache compression techniques are limited to integer benchmarks. Prior work shows that floating point data do not show redundancy to the same degree as integer benchmarks [9, 13]. Our work, focusing on floating-point data, is orthogonal and can be applied in conjunction with cache compression.

There have been significant advances in using emerging memory technology as approximate storage to trade-off storage accuracy for performance and energy savings [4, 27, 28, 29]. Our approach is different from these works because we focus on concisely representing the data elements in traditionally designed memory. Doppleganger maps approximately similar cache lines to one physical cache line, resulting in increased effective cache size [13]. Load-value approximation approximates the value of a load on a cache miss [30].

Others have proposed techniques to reduce DRAM energy consumption by adjusting DRAM refresh interval [31, 32, 33]. These techniques are specific to DRAM and focus on energy savings. ACME achieves concise storage throughout the memory hierarchy and reduces DRAM accesses by fitting more elements in the caches.

Recently, research in the field of machine learning has shown that several neural networks require very few bits for storing their input parameters [34, 35, 36]. However, these works are targeted towards deep learning systems. Our work is generic and presents an end-to-end system, tackling challenges that come when converting these memory savings into performance improvements.

There has been research to tune the precision level of an application to tradeoff performance with accuracy. Precimonious [37] and gappa++ [38] provide software precision tuning algorithms to find suitable data types for an application. However, these works are limited to `float` and `double` data types. There has been extensive research in the programming languages field to support approximate computing [5, 6, 15, 19, 39, 40]. Our works uses programmer annotations to identify approximation friendly variables as is done in prior work [15, 16, 17, 18, 19]

VII. Conclusion

This paper introduces a novel asymmetric compute-memory extension to conventional architectures, ACME, that decouples the format of data in the memory hierarchy from the format of data in the compute subsystem. ACME significantly reduces the cost of storing and moving bits throughout the memory hierarchy improving application performance. We add two instructions to the ISA - *concise-loads* and *concise-stores* which are supported in hardware vis three small functional units. Our results show that ACME achieves 1.3× speedup (up to 1.8×) while maintaining 99% accuracy, or 1.1× speedup while maintaining 99.999% accuracy, while incurring negligible area and power overheads; 0.005% area and 0.1% power to a conventional modern architecture.

Acknowledgement

We thank our anonymous reviewers for their feedback and suggestions. We also thank Akshitha Sriraman and Biruk W. Mammo for their help in setting up experimental infrastructure. This research was supported by National Science Foundation under grants NSF-CAREER-1553485, CCF-XPS-1438996, CCF-SHF-1302682 and CNS-CSR-1321047 and by the Swedish Research Council within UPMARC (Uppsala Programming for Multicore Architecture Research Centre).

References

[1] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2009.

[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.

[3] 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.

[4] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *International Symposium on Microarchitecture (MICRO)*, 2013.

[5] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Programming Language Design and Implementation (PLDI)*, 2010.

[6] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[7] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic knobs for responsive power-aware computing," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[8] M. A. Laurenzano, P. Hill, M. Samadi, S. Mahlke, J. Mars, and L. Tang, "Input responsiveness: using canary inputs to dynamically steer approximation," in *Programming Language Design and Implementation (PLDI)*, 2016.

[9] A. Arelakis and P. Stenstrom, "Sc2: A statistical compression cache scheme," in *International Symposium on Computer Architecuture (MICRO)*, 2014.

[10] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[11] S. Sardashti and D. A. Wood, "Decoupled compressed cache: Exploiting spatial locality for energy-optimized compressed caching," in *International Symposium on Microarchitecture (MICRO)*, 2013.

[12] S. Sardashti, A. Seznec, and D. A. Wood, "Skewed compressed caches," in *International Symposium on Microarchitecture (MICRO)*, 2014.

[13] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger, "Doppelganger: A cache for approximate computing," in *International Symposium on Microarchitecture (MICRO)*, 2015.

[14] D. A. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*, 2013.

[15] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "Enerj: Approximate data types for safe and general low-power computation," in *Programming Language Design and Implementation (PLDI)*, 2011.

[16] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *International Symposium on Microarchitecture (MICRO)*, 2012.

[17] A. Yazdanbakhsh, J. Park, H. Sharma, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Neural acceleration for gpu throughput processors," in *International Symposium on Microarchitecture (MICRO)*, 2015.

[18] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Language support for approximate hardware design," in *Design, Automation and Test in Europe (DATE)*, 2015.

[19] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2013.

[20] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to gpu codes," in *Innovative Parallel Computing (InPar)*, 2012.

[21] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *International Symposium on Workload Characterization (ISWC)*, 2009.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," in *SIGARCH Computer Architecture News*, 2011.

[23] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *International Symposium on Microarchitecture (MICRO)*, 2009.

[24] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," *HP Laboratories*, 2009. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.html

[25] G. Rivera and C.-W. Tseng, "Data transformations for eliminating conflict misses," in *Programming language design and implementation (PLDI)*, 1998.

[26] N. M. Ravindra, "International technology roadmap for semiconductors (ITRS) symposium," in *Journal of electronic materials*, 2001.

[27] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *International Symposium on Computer Architecture (ISCA)*, 2009.

[28] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *International Symposium on Computer Architecture (ISCA)*, 2009.

[29] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *International Symposium on Computer Architecture (ISCA)*, 2009.

[30] J. San Miguel, M. Badr, and N. Jerger, "Load value approximation," in *International Symposium on Microarchitecture (MICRO)*, 2014.

[31] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving dram refresh-power through critical data partitioning," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[32] A. Raha, H. Jayakumar, S. Sutar, and V. Raghunathan, "Quality-aware data allocation in approximate dram," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 2015.

[33] R. K. Venkatesan, S. Herr, and E. Rotenberg, "Retention-aware placement in dram (rapid): software methods for quasi-non-volatile dram," in *High-Performance Computer Architecture (HPCA)*, 2006.

[34] M. Courbariaux, Y. Bengio, and J.-P. David, "Low precision arithmetic for deep learning," in *arXiv:1412.7024*, 2014.

[35] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *arXiv:1502.02551*, 2015.

[36] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *arXiv:1506.02626*, 2015.

[37] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning assistant for floating-point precision," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

[38] M. D. Linderman, M. Ho, D. L. Dill, T. H. Meng, and G. P. Nolan, "Towards program optimization through automated analysis of numerical precision," in *Code Generation and Optimization (CGO)*, 2010.

[39] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *Programming Language Design and Implementation (PLDI)*, 2014.

[40] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<T>: A first-order type for uncertain data," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.