Data Subsetting: A Data-Centric Approach to Approximate Computing

Younghoon Kim*, Swagath Venkataramani†, Nitin Chandrachoodan‡, Anand Raghunathan*
*School of Electrical and Computer Engineering, Purdue University
†IBM T. J. Watson Research Center
‡Indian Institute of Technology, Madras
*{kim1606, raghunathan}@purdue.edu †swagath.venkataramani@ibm.com ‡nitin@ee.iitm.ac.in

***Abstract*—Approximate Computing (AxC), which leverages the intrinsic resilience of applications to approximations in their underlying computations, has emerged as a promising approach to improving computing system efficiency. Most prior efforts in AxC take a compute-centric approach and approximate arithmetic or other compute operations through design techniques at different levels of abstraction. However, emerging workloads such as machine learning, search and data analytics process large amounts of data and are significantly limited by the memory subsystems of modern computing platforms.**

In this work, we shift the focus of approximations from computations to data, and propose a data-centric approach to AxC, which can boost the performance of memory-subsystem-limited applications. The key idea is to modulate the application's data-accesses in a manner that reduces off-chip memory traffic. Specifically, we propose a data-access approximation technique called *data subsetting*, in which all accesses to a data structure are redirected to a subset of its elements so that the overall footprint of memory accesses is decreased. We realize data subsetting in a manner that is transparent to hardware and requires only minimal changes to application software. Recognizing that most applications of interest represent and process data as multi-dimensional arrays or tensors, we develop a templated data structure called `SubsettableTensor` that embodies mechanisms to define the accessible subset and to suitably redirect accesses to elements outside the subset. As a further optimization, we observe that data subsetting may cause some computations to become redundant and propose a mechanism for application software to identify and eliminate such computations. We implement `SubsettableTensor` as a C++ class and evaluate it using parallel software implementations of 7 machine learning applications on a 48-core AMD Opteron server. Our experiments indicate that data subsetting enables 1.33×–4.44× performance improvement with <0.5% loss in application-level quality, underscoring its promise as a new approach to approximate computing.

I. Introduction

Emerging workloads related to machine learning and data analytics occupy a significant fraction of execution cycles across the entire spectrum of computing platforms from mobile devices to the cloud. These workloads process large amounts of data and thereby place immense demands on the memory sub-systems of modern computing platforms. With the growing processor-memory performance gap (exacerbated by many-core processors and hardware accelerators with thousands of cores), the ability of the memory sub-system to feed data to the processing cores has become the key determinant of overall system performance.

An interesting attribute of many emerging application domains is their *intrinsic resilience i.e.,* the ability to produce results of acceptable quality despite a certain degree of inaccuracy in their underlying computations. *Approximate Computing* (AxC) is an emerging design paradigm that leverages intrinsic resilience to improve efficiency [1], [2]. Over the years, AxC techniques spanning various levels of design abstraction — software, architecture and circuits — have demonstrated significant improvements in performance and energy across a broad range of applications. A vast majority of prior efforts in AxC have focused only on approximating *computations* [3]–[14]. However, recent work suggests that applying it to other system components has a potential to result in additional improvements [15]–[20]. In this work, we explore AxC as an approach to alleviating the memory bandwidth bottleneck.

We propose a *data-centric approach* to approximate computing that can be used to improve the performance of software on off-the-shelf computing platforms (without any hardware changes). The key idea is to *modulate accesses to data structures so as to shape the memory traffic* such that the overall memory bandwidth requirement is reduced. Specifically, we propose a technique called *data subsetting*, wherein accesses to the data structure are restricted to a subset of its elements. Constraining the data structure accesses to lie within a smaller foot-print renders the resulting memory traffic more cache-friendly, thereby enhancing performance.

We identify two key challenges with data subsetting. First, we need to identify a subset of elements that are representative of the entire data structure. The size of the subset and the elements constituting the subset need to be selected based on application and data characteristics. These choices could vary from program-to-program, across data structures within the same program, and even for the same data structure over the course of execution of a program. The second challenge pertains to how the data accesses are approximated. When the application program attempts to access data that falls outside the chosen subset, the access needs to be redirected to a location within the subset. To address both these challenges in a manner that is transparent to hardware while requiring only minimal changes to the application program, we define a new templated data structure called `SubsettableTensor`. A `SubsettableTensor` is associated with an Access Redirection Function (ARF) through which the application program specifies the portion of the data structure that can be accessed and how accesses to other parts of the data structure are handled.

To extract maximum performance benefits from data subsetting on off-the-shelf computing platforms, we incorporate the following optimizations. First, to exploit spatial locality, caches are accessed at the granularity of cache lines where each cache line could store multiple elements from the data structure. If all elements present within the cache line are not part of the accessible subset, then the benefits of data

This work was supported in part by the NSF Award #1423290 and by C-BRIC, one of six centers in JUMP, an SRC program sponsored by DARPA.

978-3-9819263-2-3/DATE19/©2019 EDAA

subsetting are not fully realized, as some bandwidth is wasted in fetching elements that will never be accessed. To avoid this, we propose a *Subset Buffer* into which all the elements in the accessible subset are copied. During execution, accesses to the data structure are translated to a location within the subset buffer. Also, since data subsetting redirects many accesses to the same location in memory, some of the computations could be rendered redundant. We provide a mechanism for the application software to identify and eliminate these redundant computations to further enhance performance benefits.

In summary, the key contributions of our work are:

- We propose a data-centric approach to approximate computing, which shifts the focus of approximations from computations to data by modulating accesses to data structures so as to reduce memory bandwidth. We present an approximation technique called data subsetting, wherein accesses are limited to a subset of the data structure, which enhances performance by making the memory traffic more cache-friendly.
- We realize data subsetting in a manner that is minimally intrusive to application software through extensions to data structures. Specifically, we develop a templated data structure called `SubsettableTensor`. This approach enables the application program to fully control which elements of the data structure are part of the accessible subset and how accesses to other elements are handled.
- We improve the performance of data subsetting on off-the-shelf platforms through the use of a *subset buffer*, which enhances spatial locality among elements in the accessible subset. We also enable application software to eliminate computations that are rendered redundant as a result of data subsetting.

We apply data subsetting to parallel software implementations of 7 machine learning applications. On a 48-core AMD Opteron server, we demonstrate 1.33×-4.44× improvement in performance with negligible loss in output quality.

II. Related Work

Approximating computing is a vibrant area of research and prior efforts have proposed approximation techniques spanning the different layers of the computing stack [1], [2]. We present an overview of such efforts and highlight the key distinguishing features of our work.

Compute Approximations. Most efforts in AxC can be broadly classified as compute-centric approaches. From the bottom-up, at the circuit-level, research efforts focus on developing approximate arithmetic circuits that are highly efficient (low power, critical path, and/or leakage). This ranges from manual approximate designs of adders and multipliers [3], [4] to automatic methodologies capable of approximating arbitrary logic [5]. Next, at the architecture level, research efforts have explored approximate architectures for both general-purpose and domain-specific processors [7], [8], with suitable programming support [6]. Finally, software approximations allow applications to achieve better performance on off-the-shelf platforms. The most commonly used software approximation technique is computation skipping, wherein selected instructions or loops of the program are skipped [9]–[11]. Other software approximations include relaxing data dependencies or synchronization primitives [9], [12], precision tuning [13], and replacing functions with look-up tables or neural networks [14].

In contrast to the above approaches, we propose a data-centric approach to AxC. As opposed to approximating computations, we target modulating data structure accesses in a manner that reduces memory bandwidth, thereby speeding-up applications executed on memory-bound platforms.

Memory Approximations. Recent work has begun to recognize the merits of applying approximations to other system components [20]. Specifically, a few recent efforts have applied AxC to the memory subsystem. These include reducing DRAM refresh rate [15], [21], storing/accessing data in a compressed format [17], and speculating on the results of loads [18], [19], among others. Data subsetting is qualitatively different from the above techniques, which approximate the value of the data accessed (through skipped refreshes, compression, *etc.*) as opposed to approximating the memory location that is being accessed. Further, all the above methods require changes in hardware to carry out approximations. On the other hand, we realize data subsetting using off-the-shelf hardware and with minimal changes to application software.

III. Data Subsetting

The goal of our work is to improve the performance of memory-bound applications on off-the-shelf computing platforms using approximate computing. The key idea is to use a data-centric approach, wherein we shape the memory traffic by modulating data accesses in a manner that renders them more cache friendly. We achieve this by proposing a new approximation technique called data subsetting. In this section, we begin by describing the key concepts behind data subsetting and outline the key challenges in its realization. Finally, we describe optimizations that enhance the benefits of data subsetting on off-the-self computing platforms.

A. Concept

Figure 1 illustrates the concept behind data subsetting. The key idea is to *approximate the accesses* to the elements of a given data structure. Consider a data structure D, which is a set of elements stored in memory. Let D_{subset} be a subset of elements in the data structure. When the application attempts to access (read or write) an element of D, data subsetting approximates the access so as to ensure that it falls within D_{subset}. If the element to be accessed is already within D_{subset}, then no approximation occurs. If the element lies outside D_{subset}, then the access is *redirected* to a different element that falls within D_{subset}. In the case of a read request, an incorrect value is returned, whereas in the case of a write, an incorrect memory location is written.

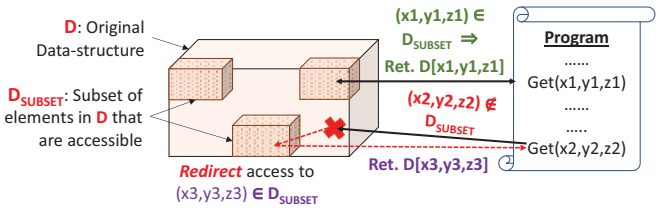

Fig. 1: Data subsetting: Concept

Benefits. By limiting accesses to a subset of elements in the data structure, data subsetting results in the following benefits:

(i) Since data subsetting limits the foot-print of data accesses, it reduces the size of the application's working set. This leads to fewer accesses to lower levels of the memory hierarchy, and in the extreme, could result in the entire working set fitting in on-chip cache, leading to substantial performance improvements. (ii) As accesses are redirected to fewer memory locations, it fundamentally increases locality and the FLOPs/Byte ratio of the application and reduces its total bandwidth requirement. (iii) Finally, since data subsetting returns the same value for many different reads and/or overwrites the same location for many different writes, some of the computations in the application are rendered redundant. Such computations can be identified and eliminated to further boost the overall performance (Section III-C2).

B. Challenges

We now describe the challenges in realizing data subsetting in practical applications.

1) Subset Selection: The first challenge lies in identifying the subset of data that is representative of the entire data structure. Naturally, subset selection is a strong function of the type of data processed by the application and the context in which they are used. For example, for data elements that are spatially correlated (*e.g.*, image pixels) or temporally correlated (*e.g.*, audio samples, video frames), a subset identified by *periodic selection* of elements often works well as it takes advantage of the locality in values. Different data strctures may exhibit spatial/temporal correlations at different granularities, which influences the subset selection. Figure 2(a) illustrates subset selection at the granularity of pixels (darker pixels are in the subset), exploiting the fact that computation results on neighboring pixels are similar. In contrast, Figure 2(b) shows a coarser-grained selection of a subset of images from a set of reference images (images with darker lines are in the subset), exploiting similarity across images. In the case of data structures with unordered data elements, as shown in Figure2(c), random subset selection may be desirable since it preserves the statistics of the data.

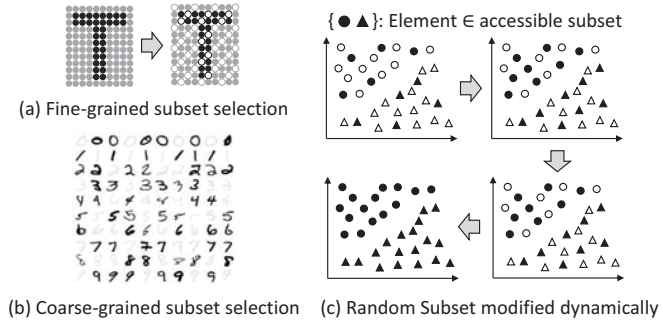

Fig. 2: Different forms of subset selection

In some applications, the choice of the subset for a given data structure may vary dynamically through the course of program execution. For instance, iterative algorithms (K-Means clustering, Stochastic Gradient Descent for Deep Neural Network (DNN) training, *etc.*) compute the output by iterating over the same data multiple times, producing a more refined output in each iteration. In such cases, the initial iterations could be executed on a small subset of the data to compute an approximate value of the output. As the iterations progress, the size of the subset could be gradually increased, allowing the algorithm to refine the final value of the output.

2) Access Approximation through Redirection: Once the subset is identified, the second challenge lies in identifying how accesses to other parts of the data structure are approximated to fall within the subset. In this case, we define an *Access Redirection Function* (ARF), a many-to-one function that maps an index to an element in the data structure to an approximate index that falls within the subset. Figure 3 illustrates ARF in the context of a 1D-tensor (D), where the subset (D_{subset}) is constructed by periodically sampling D at an interval K. We approximate accesses such that all accesses between indices $i*K$ to $i*(K+1)-1$, where $i \in \{0, 1, \ldots, |D|/K\}$, are redirected to index $i*K$. In this case, the ARF is a staircase function with K being the height of each step. This can be mathematically expressed as $ARF(i) = K * \lfloor i/K \rfloor$.

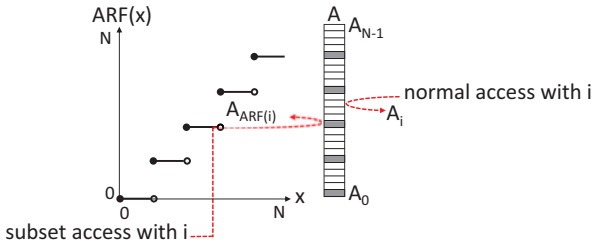

Fig. 3: Access Redirection Function (ARF) example

Our objective is to realize data subsetting with no changes to the hardware, while minimizing changes to the application software. We achieve this by building a templated data structure called `SubsettableTensor`, which is described in Section IV-A. A `SubsettableTensor` provides in-built implementations of commonly used ARFs, but also allows application software to specify any arbitrary ARF. The ARF may be modified over the course of program execution.

C. Optimizations

To maximize the performance benefits from data subsetting on off-the-shelf computing platforms, we propose 2 additional optimizations *viz. subset buffer* and *redundant computation elimination*, which are described in the following subsections.

1) Subset Buffer: In general-purpose systems, to exploit spatial locality in accesses, the cache hierarchy is accessed at the granularity of cache lines. It is possible for each cache line to store multiple elements of a data structure. This adversely impacts data subsetting when all elements present in a given cache line are not part of the accessible subset. This is because an access request to a subset element fetches the entire cache line (including out-of-subset elements) from lower in the memory hierarchy, amounting to wasted memory bandwidth. For example, as shown in Figure 4, consider a 1D-tensor with 8B data elements. Suppose the subset is constructed by periodically selecting one in every four elements. If each cache line is comprised of 8 data elements (64B), accessing a subset element would fetch all the non-subset elements surrounding it, which results in *no* bandwidth reduction.

To address this issue, once the subset elements are identified, we allocate a different space in memory called the *subset buffer* and fill it with elements that are only part of the subset. Then, instead of redirecting accesses to subset elements within the data structure, we redirect them to a location within the subset buffer. For example, the ARF for periodic

selection is modified as: $ARF_{sb}(i) = ARF(i)/K = \lfloor i/K \rfloor$. The cache lines that store the subset buffer are comprised only of subset elements, and no bandwidth is wasted. The performance overhead of populating the subset buffer for the first time is negligible, as the cost is amortized over several accesses to each location within the subset buffer.

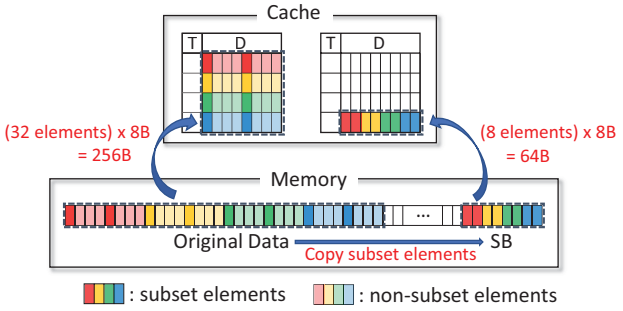

Fig. 4: Memory bandwidth reduction using subset buffers

Besides boosting performance, the use of subset buffers has other indirect benefits. Since subset elements are copied to a different location in memory, their values can be modified to better represent the data structure. For instance, consider a *neighborhood average* subsetting scheme, wherein the subset selection is periodic (period = K), but instead of picking every K^{th} element, we use the average value of the data elements within that period. In this case, *both* the access as well as the value are approximated together. Also, the use of subset buffers makes it easier to utilize optimized libraries (*e.g.*, BLAS) that expect their inputs to be laid out contiguously, by simply passing them a pointer to the subset buffer.

2) Eliminating Redundant Computations: In data subsetting, since accesses to non-subset elements are redirected to subset elements, the computations that use the non-subset elements could be rendered redundant. For example, if the application computes the maximum value of all elements in the data structure, under data subsetting, iterating over the entire data structure (with access redirection) *vs.* iterating only over the subset elements would produce identical results. In many other cases, iterating just over the subset elements may produce an incorrect answer, but can be easily compensated to compute the correct result. For instance, computing the sum of all elements in the data structure under periodic data subsetting can be achieved by computing the sum over the subset and multiplying the result by the subsetting period.

One broad approach to enable applications to skip computations pertaining to non-subset elements is to provide a mechanism that iterates *only* through the data elements in the subset. To this end, we develop a *custom iterator* for the `SubsettableTensor` data structure, which loops over the data elements presently in the subset. The application can use the iterator to compute only on the subset elements (with/without compensation at the end), further improving performance.

IV. Realizing Data Subsetting in Software

Recognizing that applications in the domains of interest ubiquitously utilize multi-dimensional arrays or tensors to represent data, we develop `SubsettableTensor`, a new templated data structure that application programs can use to realize data subsetting. In this section, we describe the `SubsettableTensor` data structure and illustrate its use through a sample application.

Code 1: SubsettableTensor data structure

```
1  template <class T> class SubsettableTensor {
2     T *original_; vec<int> shape;
3     T *SB_;
4     // Type of ARF:  (i) Periodic, (ii) Random,
5     // (iii) Neighbor-average, (iv) User-defined
6     string ARFtype; vec<int> ARFparam;
7     int *userARF(int); // Func.ptr to custom ARF
8
9     SubsettableTensor(T* data, vec<int> shape);
10    Subset(ARFtype, ARFparam, int(&usrARF)(int));
11    int ARF(int index)
12     { /* Execute pre/user-defined ARF */ }
13    T& operator[] (const int index)
14     { return SB_[ARF(index)]; }
15 }
16 #define for_subset(ST, INDEX) \
17   /* iterate through subset indexes */
```

A. SubsettableTensor data structure

Code 1 describes the `SubsettableTensor` data structure. It contains a pointer to the original data (*original_*) and a vector describing the shape of the tensor (line 2). It also contains a pointer to the subset buffer (*SB_*), which is populated when a subset is constructed. The data structure allows the application program to choose one of the pre-defined ARFs (*ARFtype* and *ARFparam*), or provide a custom ARF through a function pointer (userARF) (lines 4-7). Our implementation supports 3 pre-defined ARFs *viz. periodic*, *random* and *neighbor-average* that were described in Section III. The selection of ARF is achieved through a member function called *Subset* (line 10), which also allocates and fills in the subset buffer. Every access to a given index of the data structure is first passed through the ARF (lines 11-12), which computes an approximated index. The subset buffer is accessed using the approximated index and the value is returned (lines 13-14). Finally, to enable redundant computation skipping, `SubsettableTensor` provides an iterator (*for_subset*), which loops through the subset indices.

B. Illustration: K-means Clustering

We now illustrate how `SubsettableTensor` can be used in a practical application by using K-Means clustering as an example. In K-Means, given a set of points (represented as vectors), the objective is to group them into K clusters. K-Means is an iterative algorithm, where in each iteration, the distances between all points and the centroids of the K clusters are computed. Each point is assigned to the cluster to whose centroid it is closest. Each cluster's centroid is recomputed by averaging the points assigned to it. This process is repeated until convergence (*e.g.*, no point changes clusters).

Code 2 shows the pseudo-code for K-Means clustering with data subsetting (changes made for data subsetting are highlighted in boldface). In this case, we define the input points (*in*) as a `SubsettableTensor` – *insub* (line 2). First, a subset is constructed by randomly selecting a fraction (*initSubsetRatio*) of the input points (line 3). In each iteration, we process the chosen subset of points (line 6), compute the distances with each cluster centroid (lines 7-9), and assign each point to the cluster whose centroid is closest to (line 10). We then compute the fraction of inputs whose clusters were reassigned (*fracReassigned* in line 11), based on which we compute a new subset ratio (*newSubRatio* in line 12). When

Code 2: Data subsetting applied to K-Means clustering

```
1  int in[VEC][FEA], cent[K][FEA], assign[VEC];
2  SubsettableTensor<int> insub(in,{VEC,FEA});
3  insub.Subset(Random,{initSubsetRatio,1},NULL);
4  do {
5    int assign_old [VEC] = assign;
6    for_subset(insub, int i) {
7      int dist[K] = 0;
8      for(int j=0; j<K; j++) {
9        dist[j]=CalcDist(in[i,:], cent[j,:]);
10     int closest=FindClosestCluster(dist, j); }
11   fracReassigned = diff(assign_old, assign)/VEC;
12   newSubRatio = getSubRatio(fracReassigned)
13   insub.Subset(Random,{newSubRatio,1},NULL);
14   CalculateCentroid(in, cent, assign);
15 } while (fracReassigned > 0 && newSubRatio == 1)
```

only a small fraction of inputs are reassigned, we ascertain that the centroids have stabilized and therefore we increase the size of the subset to consider more points. This is achieved by calling the *Subset* function on *insub* with *newSubRatio* (line 13). The algorithm terminates when there are no reassignments and the subset ratio is 1 *i.e.,* clusters have been assigned to all input points (line 15).

Thus, data subsetting can be realized using `SubsettableTensor` with minimal changes to the original program.

V. Experimental Setup

Benchmarks. To evaluate data subsetting, we consider a benchmark suite comprising of 7 machine learning applications listed in Table I. The benchmarks use 5 different classification and clustering algorithms. In the case of the Deep Neural Network (DNN) benchmarks, we applied data subsetting only to the largest convolutional and fully connected layer *viz. conv*2 and $fc6$ for AlexNet, and *conv*1_2 and $fc6$ for VGG16, respectively. For the classification benchmarks, we used classification accuracy *i.e.,* fraction of inputs classified correctly, as our quality metric. In the case of K-Means clustering, quality is measured as the average distance between all data points and their corresponding cluster centroids.

TABLE I: Machine learning benchmark applications

Algorithm	Application	Dataset	# Inputs	Data Size (MB)
GLVQ	Eye detection (EYE)	Image set from NEC labs	1465	64
KNN	Digit classification (DGT)	MNIST	1000	180
	Digit classification (DGT2)	Gissete	1000	115
SVM	Text classification (TXT)	Reuters	598	34
DNN	AlexNet	ImageNet	1000	3.9 (conv2)
				146 (fc6)
	VGG16	ImageNet	1000	110 (conv1_2)
				397 (fc6)
KMEANS	K-means clustering (KMS)	Volcanoes	128	512

Performance evaluation. All the benchmarks were implemented in C++ and parallelized using OpenMP. We also developed data-subsetted versions of the benchmarks by using the `SubsettableTensor` data structure. Our experiments were conducted on a 48-core server platform, whose parameters are shown in Table II.

TABLE II: System configuration used in experiments

CPU	AMD Opteron, 48 cores, 2.3GHz
CACHE	L1 64KB, L2 512KB, L3 20MB, 64B lines
BUS	42.7 GB/s peak bandwidth
MEM	190GB, DDR3-1333MHz, 8 channels

VI. Results

This section evaluates the effectiveness of data subsetting using various experiments.

A. Performance Benefits

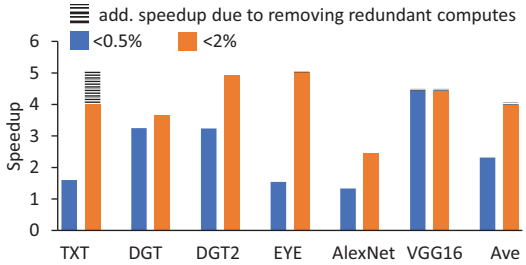

Fig. 5: Speedup obtained within different quality constraints

Figure 5 shows the speedup achieved with data subsetting for different application-level quality constraints. Across all benchmarks, the speedup ranges between 1.33×-4.44× (2.31× on average) for a negligible loss (<0.5%) in quality. When the quality constraint is relaxed to <2%, the speedup increases to 2.47×-5.06× (4.05× on average). Figure 5 also highlights the performance gain due to skipping computations rendered redundant by data subsetting. Since the applications were primarily memory-bound, a significant fraction of the benefits stem from bandwidth reduction as opposed to computation skipping, which was effective only for `TXT` and on average yielded 7% additional performance gain.

B. Comparison with Loop Perforation

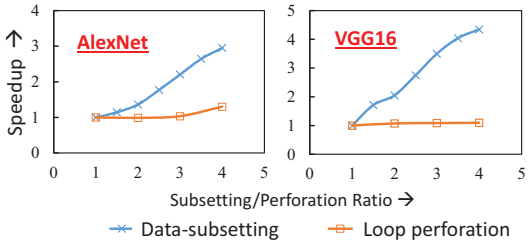

Fig. 6: Data subsetting *vs.* loop perforation

We compare the performance of data subsetting with a well-known compute-centric approximation technique called *loop perforation* [10], wherein iterations of loops are periodically or randomly skipped from execution. Figure 6 shows the speedup achieved with increasing subsetting/perforation ratio (which varies inversely with the subset size or loop iterations executed). Computations on non-subset elements were eliminated using the for_subset iterator for data subsetting. First, in the case of data subsetting, as the subsetting ratio is increased, we achieve a steady improvement in performance. In some cases, the performance improvement is super-linear (*e.g.* subsetting ratio of 3 yields 3.5× performance benefits), as data structures begin to fit within levels of the memory hierarchy.

In the case of loop perforation, the speed-ups are significantly lower. Although loop perforation eliminates both computations and data accesses present in the skipped loop iterations, it does not necessarily translate into memory bandwidth reduction since the adjacent data elements in the same cache line are being fetched despite being unused. Moreover, skipping computations reduces the time the processor works on a cache line, shortening the interval between cache line

fetches and thus putting even more pressure on the memory subsystem. Since data subsetting utilizes a subset buffer, out-of-subset data elements are not fetched. This result underscores the effectiveness of data-centric approaches in the context of memory-bound applications.

C. Choice of Access Redirection Function

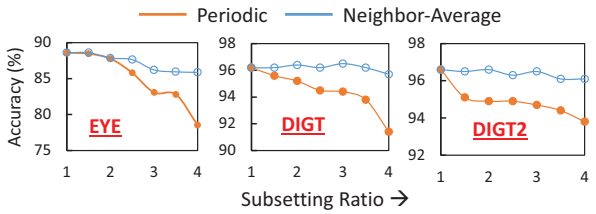

Fig. 7: Impact of ARF choice on application quality

Next, we study the impact of ARF choice on application quality (classification accuracy). To this end, we consider 2 ARFs – periodic *vs.* neighborhood-average. In periodic subsetting, every K^{th} is selected to be part of the subset. Neighborhood-average also picks one in every K elements, but places the average of all elements in each interval of K in the subset buffer. Figure 7 shows the degradation in output quality as the degree of subsetting is increased from 1 to 4. In all cases, we observe a degradation in quality as the subsetting period is increased. However, neighborhood-average incurs very little quality degradation compared to periodic ARF since it accounts for out-of-subset elements. Therefore, it is useful for datasets that exhibit lower spatial locality.

D. Dynamic Modulation of Subset Size

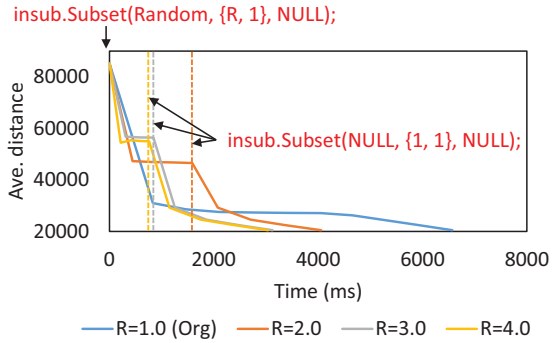

Fig. 8: Data subsetting applied to K-Means clustering

A desirable feature of `SubsettableTensor` is the ability to modulate the subset size during execution. This can be leveraged in the context of iterative algorithms such as K-Means clustering (Section IV-B). We demonstrate this in action in Figure 8, which plots how the quality metric (average distance between data points and their cluster centroids) converges over time when we begin the iterations with different sizes of subsets. We find that the original algorithm (blue line: $R = 1$) takes the longest time to converge since each iteration takes longer. When we start clustering on only half of the data elements which are randomly picked (orange line: $R = 2$), we find that the algorithm converges quicker, but converges to a sub-optimal solution at a higher average distance. At this point, we increase the subset size to include all elements in the data structure, which enables it to converge to the same average distance as the original algorithm, while achieving 1.62× performance improvement. We observe a similar behavior when we begin with even smaller subset sizes ($R = 3, 4$), but beyond a point we do not see further performance improvements, as the subset sizes are too small to move the cluster centroids in the correct direction.

VII. Conclusion

Optimizing memory bandwidth is key to the efficiency of emerging data-intensive applications on modern computing platforms. We address this challenge using approximate computing, wherein we adopt a data-centric approach. Specifically, we propose an approximation technique called data subsetting, which approximates accesses to a data structure by redirecting them to fall within a pre-defined subset of elements. This enhances performance as the foot-print of the application's memory accesses is significantly reduced. We realize data subsetting through a templated data structure called `SubsettableTensor`, which allows application software to specify and dynamically modify which elements constitute the accessible subset, and how accesses to other parts of the data structure are redirected. Additionally, we propose optimizations such as the use of a subset buffer, and identifying and skipping redundant computations to enhance the benefits of data subsetting. We evaluate data subsetting on parallel software implementations of 7 machine learning applications and demonstrate 1.33×-4.44× improvement in performance.

References

[1] S. T. Chakradhar and A. Raghunathan, "Best-effort computing: Re-thinking parallel software and hardware," in *Proc. DAC*, June 2010.

[2] S. Venkataramani *et al.*, "Approximate computing and the quest for computing efficiency," in *Proc. DAC*, 2015, pp. 120:1–120:6.

[3] V. Gupta *et al.*, "IMPACT: Imprecise adders for low-power approximate computing," in *Proc. ISLPED*, Aug 2011, pp. 409–414.

[4] P. Kulkarni *et al.*, "Trading accuracy for power with an underdesigned multiplier architecture," in *Proc. VLSID*, Jan 2011, pp. 346–351.

[5] S. Venkataramani *et al.*, "SALSA: Systematic logic synthesis of approximate circuits," in *Proc. DAC*, June 2012, pp. 796–801.

[6] A. Sampson *et al.*, "EnerJ: Approximate data types for safe and general low-power computation," in *Proc. PLDI*, 2011, pp. 164–174.

[7] H. Esmaeilzadeh *et al.*, "Architecture support for disciplined approximate programming," in *Proc. ASPLOS*, 2012, pp. 301–312.

[8] S. Venkataramani *et al.*, "Quality programmable vector processors for approximate computing," in *Proc. MICRO*, 2013, pp. 1–12.

[9] J. Meng *et al.*, "Best-effort parallel execution framework for recognition and mining applications," in *Proc. IPDPS*, 2009, pp. 1–12.

[10] S. Sidiroglou-Douskos *et al.*, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. ESEC/FSE*, 2011.

[11] M. Samadi *et al.*, "SAGE: Self-tuning approximation for graphics engines," in *Proc. MICRO*, 2013, pp. 13–24.

[12] R. Akram *et al.*, "Approximate lock: Trading off accuracy for performance by skipping critical sections," in *Proc. ISSRE*, Oct 2016.

[13] C. Rubio-González *et al.*, "Precimonious: Tuning assistant for floating-point precision," in *Proc. SC*, Nov 2013, pp. 1–12.

[14] H. Esmaeilzadeh *et al.*, "Neural acceleration for general-purpose approximate programs," in *Proc. MICRO*, 2012, pp. 449–460.

[15] S. Liu *et al.*, "Flikker: Saving DRAM refresh-power through critical data partitioning," in *Proc. ASPLOS*, 2011, pp. 213–224.

[16] A. Raha *et al.*, "Quality-aware data allocation in approximate DRAM," in *Proc. CASES*, 2015.

[17] A. Ranjan *et al.*, "Approximate memory compression for energy-efficiency," in *Proc. ISLPED*, July 2017, pp. 1–6.

[18] J. S. Miguel *et al.*, "Load value approximation," in *Proc. MICRO*, Dec 2014, pp. 127–139.

[19] A. Yazdanbakhsh *et al.*, "RFVP: Rollback-free value prediction with safe-to-approximate loads," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 62:1–62:26, Jan. 2016.

[20] A. Raha and V. Raghunathan, "Towards full-system energy-accuracy tradeoffs: A case study of an approximate smart camera system," in *Proc. DAC*, 2017, pp. 74:1–74:6.

[21] A. Raha *et al.*, "Quality configurable approximate DRAM," *IEEE Transactions on Computers*, vol. 66, no. 7, pp. 1172–1187, July 2017.