Value Similarity Extensions for Approximate Computing in General-Purpose Processors

Younghoon Kim*, Swagath Venkataramani†, Sanchari Sen†, Anand Raghunathan*
*School of Electrical and Computer Engineering, Purdue University
†IBM T. J. Watson Research Center
*{kim1606, raghunathan}@purdue.edu †{swagath.venkataramani, sanchari.sen}@ibm.com

***Abstract*—Approximate Computing (AxC) is a popular design paradigm wherein selected computations are executed approximately to gain efficiency with minimal impact on application-level quality. Most efforts in AxC target specialized accelerators and domain-specific processors, with relatively limited focus on General-Purpose Processors (GPPs). However, GPPs are still broadly used to execute applications that are amenable to AxC, making AxC for GPPs a critical challenge.**

A key bottleneck in applying AxC to GPPs is that their execution units account for only a small fraction of total energy, requiring a holistic approach targeting compute, memory and control front-ends. This paper proposes such an approach that leverages the application property of *value similarity*, *i.e.*, input operands to computations that occur close-in-time take similar values. Such similar computations are dynamically pre-detected and the fetch-decode-execute of entire instruction sequences are skipped to benefit performance. To this end, we propose a set of lightweight micro-architectural and ISA extensions called VSX that enable: (i) similarity detection amongst values in a cache-line, (ii) skipping of pre-defined instructions and/or loop iterations when similarity is detected, and (iii) substituting outputs of skipped instructions with saved results from previously executed computations. We also develop compiler techniques, guided by user annotations, to benefit from VSX in the context of common Machine Learning (ML) kernels. Our RTL implementation of VSX for a low-power RISC-V processor incurred 2.13% area overhead and yielded 1.19×-3.84× speedup with <0.5% accuracy loss on 6 ML benchmarks.

I. Introduction

In the previous decade, Approximate Computing (AxC) has emerged as a popular design paradigm that leverages the intrinsic ability of applications to tolerate approximations in some of their computations to boost compute efficiency [1], [2]. A significant fraction of prior efforts in AxC have focused on specialized architectures such as domain-specific processors [3], GPUs [4], or custom accelerators [2], while relatively limited effort has been devoted towards General-Purpose Processors (GPPs). However, modern applications that benefit from AxC are still widely executed on GPPs for various reasons, ranging from their easy programmability to stringent cost/area budgets precluding the use of custom accelerators. For example, the majority of Facebook's inference workloads run on CPUs within edge devices [5]. Hence, leveraging AxC to improve the efficiency of GPPs is of significant interest.

AxC in GPPs. A key challenge in approximating GPPs is that their execution units contribute only a small fraction of total energy. Hence, AxC in GPPs has been predominately achieved through software-level approximations such as function approximation, loop skipping, and relaxed synchronization [1], [2], [6]. The few efforts that propose hardware approximations for GPPs employ approximations either within the execution units through memoization [7], [8] and voltage scaling [9], [10], or in the memory sub-system through load value prediction [11]. The benefits from these approaches are limited by the energy expended in the parts that are *not approximated*, such as instruction fetch, decode, control, *etc*.

This work was supported in part by C-BRIC, one of six centers in JUMP, an SRC program sponsored by DARPA.

AxC through Value Similarity. In this paper, we propose a holistic approach to AxC in GPPs that encompasses compute, memory and control front-ends. We leverage the application property of *value similarity*, *i.e.,* input operands to computations that occur close-in-time take similar values, thereby producing results that are similar. Our analysis of six representative Machine Learning (ML) workloads indicates that value similarity is broadly prevalent, impacting as much as 80% of the computations executed. This provides us an opportunity to pre-detect similar computations and skip fetch-decode-execute of entire instruction sequences, while substituting their results with those of previously executed computations, benefiting both performance and energy. To this end, we propose VSX, a set of lightweight micro-architectural and ISA extensions to leverage value similarity in GPPs. We also present compiler techniques that leverage user annotations to benefit from VSX in the context of common ML kernels.

A key trade-off in the design of VSX is balancing the window-of-opportunity *i.e.,* the span of computations across which similarity is exploited *vs.* the complexity of the implementation and overheads incurred. To ease the pressure on the memory-subsystem, GPP applications are structured to maximize access locality to elements within a cache-line. We leverage this observation in VSX and pre-detect similar values among elements of a cache-line when performing a load (the similarity threshold is defined by software). The pre-detected similarity information is passed to the instruction fetch unit, which skips instructions even before they enter the pipeline by appropriately controlling the program counter.

VSX provides maximum flexibility to the compiler to define which instructions are skippable under what conditions. This is critical as the impact of approximations on the overall quality cannot be solely ascertained in hardware. To this end, the micro-architecture contains a programmable *Similarity Based Skip Table* that is configured before entering a code region where value similarity can be exploited. Based on programmer annotations in the code, the compiler generates the skip information for common ML kernels such as GEMM, Conv2D *etc.* As a further optimization, VSX also supports *iteration fast-forwarding*, *i.e.*, when operands to a set of loop iterations are similar to a previous one, the entire iteration set is skipped and replaced by a shorter specialized instruction sequence to produce an approximate output.

In summary, the key contributions of our work are:

- We propose a holistic approach to AxC in GPPs, targeting benefits in compute, memory and control front-ends, by leveraging value similarity in input operands across computations that occur close-in-time.
- We propose a set of lightweight micro-architectural and ISA extensions called VSX that enable dynamic pre-detection of similar computations to conditionally skip the fetch-decode-execute of entire instruction sequences. VSX also includes the ability for iteration fast-forwarding, wherein a set of loop iterations are replaced by a small, specialized sequence of instructions.
- We develop compiler techniques that are guided by user-annotations and transform common application kernels to exploit value similarity and benefit from VSX.

We have implemented VSX within a RISC-V in-order processor in RTL and synthesized it to commercial 45nm technology. With an area overhead of 2.13%, VSX achieves 1.19×-3.84× speedup with <0.5% accuracy loss on 6 ML benchmarks.

II. Related Work

Prior efforts on approximate computing that exploit value similarity as the source of efficiency improvement can be grouped into the following categories.

Approximate memoization. One way of exploiting value similarity is to save computation results for commonly occurring inputs and reuse them upon seeing similar inputs. This idea has been explored in the SW domain [12], the GPP domain [7], [8], [13], and the custom accelerator domain [14], [15]. Our approach differs from these works by being more holistic, *i.e.*, not restricted to the execution units, and by eliminating the need for large memoization buffers.

Approximate load value prediction. Another line of work targeting value similarity focuses on similar values being loaded by GPPs [11] and GPUs [16]. They use the history of loaded values to predict the current load value in case of a cache miss. This approach effectively hides cache miss penalties in memory-bound situations where multiple processor cores fetch data. On the contrary, our approach entirely skips loads as well as computes and other control instructions for data elements that have already been brought up to the L1 cache, thereby being more broadly applicable, including in compute-bound applications.

GPU-specific approximations. Prior efforts that exploit value similarity in GPUs include sharing an adjacent thread's result [17], or dynamically detecting inputs with similarity and computing a representative result for them [4]. Our design differs significantly due to our focus on GPPs. As mentioned earlier, workloads that can benefit from AxC often run on CPUs, especially in resource-constrained platforms where using a GPU is not a feasible option.

Input sampling. Exploiting value similarity among neighboring data elements, prior efforts sample inputs such as pixels to reduce the number of loads [18], or redirect accesses to neighboring elements in order to reduce memory bandwidth [19]. In contrast to these pure SW techniques that apply approximation statically by assuming that neighboring elements are similar, our HW approach detects similar values on-the-fly and approximates them so that the impact on output quality is minimized.

III. Value Similarity: Sources, Opportunities and Challenges

Value similarity refers to the presence of numerically similar values in neighboring elements of a data-structure. The two main sources of value similarity are: i) redundancy in real-world inputs, *e.g.*, homogeneous regions within an image, and ii) nature of the computation itself, *e.g.*, sorting algorithms placing similar values nearby or saturating activation functions in DNNs.

A large number of compute kernels access successive elements of their data-structures (*e.g.*, activation and weight arrays in DNNs) and perform the same operation (*e.g.*, multiply-and-accumulate) on them across multiple loop iterations. The presence of value similarity in these data-structures can thus lead to load instructions loading similar values and feeding them to compute instructions (*e.g.*, addition, multiplication, *etc.*), which in turn produce similar results across consecutive loop iterations. Accordingly, we can skip some of these instructions and approximate their results with that of a previous instruction to achieve execution time savings. However, realizing this approach in a GPP involves overcoming the following key challenges:

- *Identifying potentially skippable instructions.* Dynamically identifying instructions that produce similar results and are thus *skippable*, involves detecting and storing the similarity information of executed instructions. This can lead to a large overhead if done indiscriminately.
- *Detecting similarity before instruction execution.* For maximizing execution time savings, we need to identify the skipped instructions before even fetching them into the pipeline, thereby avoiding the introduction of bubbles. This demands the knowledge of an instruction's similarity information ahead of its execution.
- *Saving & reusing instruction results.* When an instruction is skipped, subsequent instructions using its result should be able to receive an approximated value. Thus, we need a mechanism for saving previous results and reusing them in place of skipped instructions.

We overcome the above challenges through lightweight micro-architectural and ISA extensions to a GPP core that are described in the next section.

IV. VSX: Value Similarity Extensions for GPPs

We propose Value Similarity eXtensions (VSX), a set of low-overhead and minimally intrusive micro-architectural and ISA extensions, for exploiting value similarity in GPPs. This section presents the key ideas of VSX and illustrates in detail how they can be integrated within an in-order processor pipeline.

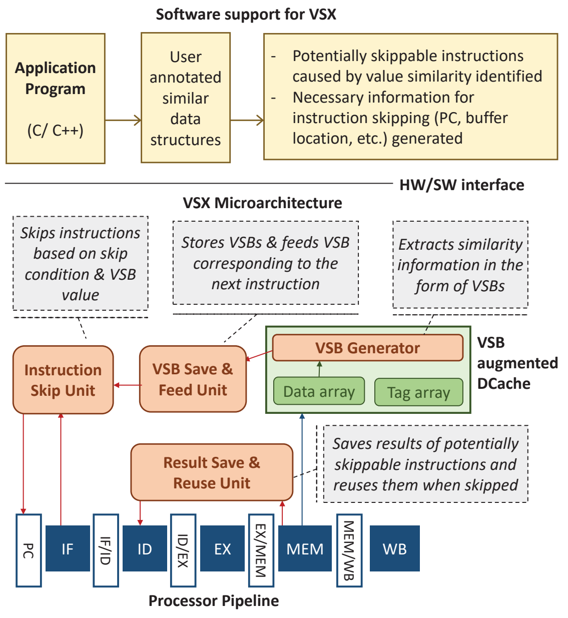

Software support for VSX
Application Program (C/ C++)
User annotated similar data structures
- Potentially skippable instructions caused by value similarity identified
- Necessary information for instruction skipping (PC, buffer location, etc.) generated
HW/SW interface
VSX Microarchitecture
Skips instructions based on skip condition & VSB value
Stores VSBs & feeds VSB corresponding to the next instruction
Extracts similarity information in the form of VSBs
Instruction Skip Unit
VSB Save & Feed Unit
VSB Generator
Data array
Tag array
VSB augmented DCache
Result Save & Reuse Unit
Saves results of potentially skippable instructions and reuses them when skipped
PC
IF
IF/ID
ID
ID/EX
EX
EX/MEM
MEM
MEM/WB
WB
Processor Pipeline

Fig. 1: VSX overview

A. VSX: Overview

Figure 1 presents an overview of the proposed VSX extensions. These extensions help overcome the challenges listed in Section III through HW/SW co-design.

Identifying potentially skippable instructions from user annotations in software. We provide a pragma that allows users to annotate data-structures with value similarity in a C/C++ program. From this annotation, a list of potentially skippable instructions are inferred at compile time to avoid the overhead of analyzing similarity of every instruction. This instruction list, along with their skip conditions, are stored in an *Instruction Skip Unit (ISU)* using a custom instruction. Before fetching every instruction, the *ISU* looks up this information to decide whether it can be skipped, in a similar way to [20].

Extracting similarity information from the data cache. We augment the data cache with a *VSB Generator* that generates bit-flags called *Value Similarity Bits (VSBs)*, indicating whether each element in the cache-line is similar to the first element. When an application kernel traversing an array loads the first data element of a cache-line containing N elements, VSBs for the next $N-1$ elements are produced by the *VSB Generator*. These VSBs are transferred to the core and saved in a *VSB Save & Feed Unit*, which selects the appropriate VSB for a future load instruction and thereby identifies its similarity to a previous load. The similar loads are subsequently skipped by the *ISU*.

Saving & reusing instruction results in hardware. Our instruction skipping strategy requires instructions loading the first data element of each cache-line to complete execution, as VSBs are generated upon execution of those loads. We save the results of such loads and reuse them in place of a skipped load's result using a *Result Save & Reuse Unit (RSRU)*. Thus, compute instructions consuming a skipped load's result are guaranteed to receive a value similar to the actual one.

Code 1: User annotation in SW

```
float *a, *b;
init(a, b, ARRAY_SIZE);
float dp = 0;
#pragma vsx(a,TH1,b,TH2) // by user
for (int i=0; i<ARRAY_SIZE; i++)
    dp += a[i] * b[i];
```

B. Program Annotations and Compiler Techniques for VSX

Code 1 shows how user annotation in SW can be done for a simple loop that computes a dot-product of two vectors. The programmer inserts a pragma right before the kernel of interest — the *for* loop in this example — that specifies the data-structures with value similarity (a, b) and the difference threshold for similarity evaluation (*TH1*, *TH2*). When the *for* loop in Code 1 is compiled to an assembly code shown in Figure 2, instructions *LD1* and *LD2* are identified to be potentially skippable as they load data with value similarity. Compute instruction *MUL* also becomes skippable when *LD1* and *LD2* are both skipped. These skip conditions for different instructions are transferred to the *ISU* using a custom instruction, *SBST-LD*, before executing the application kernel.

C. Instruction Skipping & Result Reuse

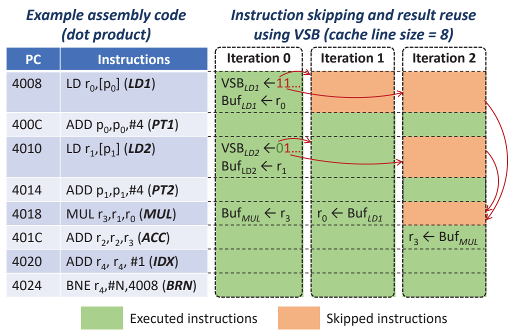
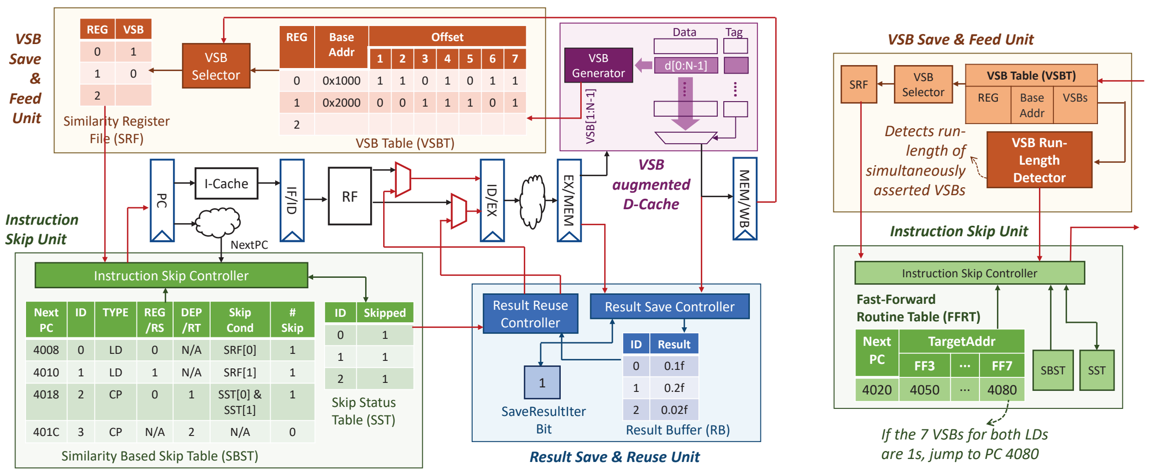

Example assembly code (dot product)
Instruction skipping and result reuse using VSB (cache line size = 8)
PC
Instructions
Iteration 0
Iteration 1
Iteration 2
4008
LD r_0,[p_0] (LD1)
400C
ADD p_0,p_0,#4 (PT1)
4010
LD r_1,[p_1] (LD2)
4014
ADD p_1,p_1,#4 (PT2)
4018
MUL r_3,r_1,r_0 (MUL)
401C
ADD r_2,r_2,r_3 (ACC)
4020
ADD r_4, r_4, #1 (IDX)
4024
BNE r_4,#N,4008 (BRN)
$VSB_{LD1} \leftarrow 11\ldots$
$Buf_{LD1} \leftarrow r_0$
$VSB_{LD2} \leftarrow 01\ldots$
$Buf_{LD2} \leftarrow r_1$
$Buf_{MUL} \leftarrow r_3$
$r_0 \leftarrow Buf_{LD1}$
$r_3 \leftarrow Buf_{MUL}$
Executed instructions
Skipped instructions

Fig. 2: Instruction skip & result reuse example

Figure 2 illustrates instruction skipping & result reuse across multiple iterations of an example dot-product assembly code. As mentioned in Section IV-B, instructions *LD1*, *LD2*, and *MUL* are potentially skippable. In iteration 0, we assume that both *LD1* & *LD2* access the first element of a cache-line and the *VSB Generator* produces 7 VSBs each (VSB_{LD1}/VSB_{LD2}) for the next 7 data elements (assuming a cache-line size of 8). Results of *LD1* & *LD2* are saved in their respective buffer slot (Buf_{LD1}/Buf_{LD2}) within the *RSRU* to be reused whenever *LD1* or *LD2* is skipped in the future. Result of *MUL* is also saved to Buf_{MUL} since it can be skipped as well. In iteration 1, *LD1* is skipped as its corresponding VSB_{LD1} equals 1. When *MUL* requires the result of *LD1* in the same iteration, the saved result Buf_{LD1} is used in place of operand register r_0. In iteration 2, both *LD1* & *LD2* are skipped as both VSB_{LD1} and VSB_{LD2} equals 1. *MUL* is also skipped and *ACC* uses the value of Buf_{MUL} in place of its operand register r_3 as indicated.

D. VSX: Microarchitecture

Figure 3(a) shows the proposed micro-architectural extensions integrated within a 5-stage in-order processor pipeline. The details of each extension are presented below.



REG	VSB
0	1
1	0
2	

REG	Base Addr	Offset 1	2	3	4	5	6	7
0	0x1000	1	1	0	1	0	1	1
1	0x2000	0	0	1	1	1	0	1
2								

Next PC	ID	TYPE	REG /RS	DEP /RT	Skip Cond	# Skip
4008	0	LD	0	N/A	SRF[0]	1
4010	1	LD	1	N/A	SRF[1]	1
4018	2	CP	0	1	SST[0] & SST[1]	1
401C	3	CP	N/A	2	N/A	0

ID	Skipped
0	1
1	1
2	1

ID	Result
0	0.1f
1	0.2f
2	0.02f

SRF	VSB Selector	VSB Table (VSBT) REG	Base Addr	VSBs

Next PC	TargetAddr FF3	⋯	FF7
4020	4050	⋯	4080

(a) (b)

Fig. 3: (a) Overall VSX microarchitecture (b) Modified *ISU* and *VSB Save & Feed Unit* for iteration fast-forwarding

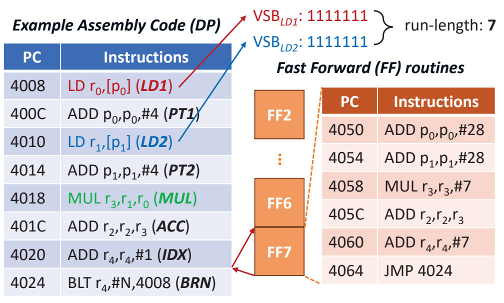
VSB Generator resides inside the data cache and generates VSBs for future load values. It performs a similarity check whenever a potentially skippable load instruction (identified by the *ISU*) accesses the first element of a cache-line. For a cache-line of N data elements (*d[0:N-1]*), $N-1$ VSBs (*VSB[1:N-1]*) indicating each data element's similarity to *d[0]* are generated. We compare the value difference with the user-defined threshold from Section IV-B in order to evaluate similarity.

VSB Save & Feed Unit consists of a *VSB Table (VSBT)*, a *VSB Selector*, and a *Similarity Register File (SRF)*. The *VSBT* is a register file storing the VSBs generated by the *VSB Generator* for different cache-line addresses (*BaseAddr*). It is indexed using pointer registers (*REG*). Whenever a pointer register is updated, the *VSB Selector* uses the updated memory address to select the appropriate VSB from the *VSBT* and stores it in the *SRF*. The *ISU* refers to the *SRF* to obtain the similarity information for the next load instruction.

Instruction Skip Unit (ISU) consists of a *Similarity Based Skip Table (SBST)*, a *Skip Status Table (SST)*, and an *Instruction Skip Controller (ISC)*. The *SBST* stores the information necessary to identify skippable instructions, generated from user annotations described in Section IV-B. The individual entries correspond to different instruction regions, identified using the *NextPC* field, which become skippable whenever the conditions within the *SkipCond* field are met. For every instruction in the *IF* stage of the pipeline, the *SBST* is looked up with the *NextPC* value. If there's a match, the corresponding *SkipCond* is evaluated by the *ISC* and the PC is advanced by the value of the *#Skip* field. For a load instruction (*SBST.TYPE* equals *LD*), *SBST.REG* — index of the pointer register containing its target address — is used to access its corresponding VSB in the *SRF* while evaluating *SkipCond*. The *ISC* further records the result of this skip condition in the *SST* to allow skip condition evaluations of future instructions. For a compute instruction (*SBST.TYPE* equals *CP*), the *SBST.ID* of instructions producing its operands are used to access the *SST* for evaluating the skip condition.

Result Save & Reuse Unit consists of a *Result Buffer (RB)*, a *Result Save Controller (RSC)*, a *Result Reuse Controller (RRC)*, and a *SaveResultIter Bit*. The *RB* saves the results of potentially skipped instructions in different entries indexed using their *SBST.ID*s. The *RSC* in the *MEM* pipeline stage controls the write-enable condition of the *RB*. The result of a load instruction is allowed to be written to the *RB* if it corresponds to the first element of a cache-line. On the other hand, the result of a compute instruction is allowed to be written to the *RB* only if the *SaveResultIter Bit* is set. The *SaveResultIter Bit* is asserted whenever a load saves its result to the *RB*, indicating that a new value will be produced, and is cleared at the end of the loop iteration by the branch instruction (*i.e.*, *BRN* in Figure 2). We ensure that all data-structures associated with potentially skippable loads are aligned to the base address of a cache-line through standard C/C++ library macros. Finally, the *RRC*, in the *ID* stage of the pipeline, allows an instruction's operands to be replaced by the saved values in the *RB*. For any instruction, if the skip record of its operands (*SBST.RS* or *SBST.RT*) is set in the *SST*, the value of the corresponding operand is accessed from the *RB* instead of the register file.

E. Iteration Fast-Forwarding



PC	Instructions
4008	LD r_0,[p_0] (*LD1*)
400C	ADD p_0,p_0,#4 (*PT1*)
4010	LD r_1,[p_1] (*LD2*)
4014	ADD p_1,p_1,#4 (*PT2*)
4018	MUL r_3,r_1,r_0 (*MUL*)
401C	ADD r_2,r_2,r_3 (*ACC*)
4020	ADD r_4,r_4,#1 (*IDX*)
4024	BLT r_4,#N,4008 (*BRN*)

PC	Instructions
4050	ADD p_0,p_0,#28
4054	ADD p_1,p_1,#28
4058	MUL r_3,r_3,#7
405C	ADD r_2,r_2,r_3
4060	ADD r_4,r_4,#7
4064	JMP 4024

Fig. 4: Iteration fast-forwarding example

In addition to the VSX microarchitecture shown in Figure 3(a) for skipping and reusing results of individual instructions, we develop another optimization strategy based on two key observations. First, VSBs provide similarity information for multiple future iterations. For example, assuming that both loads (*LD1, LD2*) in the dot-product loop from Figure 4 have generated VSBs of 1111111, we know that the next 7 loop iterations will produce similar *MUL* results. Second, multiple loop iterations producing similar results can be substituted with a specialized instruction sequence. For example, the next 7 iterations in the previous example can be substituted with a separate instruction sequence *FF7* that produces an identical output. We propose a systematic approach — iteration fast-forwarding — for detecting and substituting loop iterations that produce similar results. First, the run-length of VSBs for all loads is calculated upon each VSB generation. The run-length information is then used to jump to the corresponding specialized sequence called *Fast-Forward Routine (FFR)* that substitutes multiple loop iterations and jumps back to the original instruction stream. Figure 3(b) shows the additional HW units for realizing iteration fast-forwarding. The *VSB Run-Length Detector* attached to the *VSB Save & Feed Unit* from Figure 3(a) detects the run-length of asserted VSBs for all entries in the *VSBT*. The *ISU* is now equipped with an *FFR Table*: A table storing the starting PCs of FFRs for different VSB run-lengths. Finally, the *ISC* modifies the PC to jump to an FFR corresponding to the VSB run-length.

V. Experimental Setup

CPU	Single-core in-order RISC-V w/ FPU at 200MHz clock speed
Cache	L1D: 64KB 2-way SA (w/prefetcher) L1I: 16KB 2-way SA 1-cycle hit latency, 32B lines
Mem	1GB LPDDR3

(a)

Application	Algo.	Dataset	#Inps
EYE - Eye detection	GLVQ	Image set from NEC labs	1465
DGT - Digit classification	KNN	MNIST	1000
DGT2 - Digit classification		Gissete	1000
TXT- Text classification	SVM	Reuters	598
AlexNet	DNN	ImageNet	1000
VGG16			1000

(b)

TABLE I: (a) Benchmarks (b) gem5 system configurations

Evaluation. We implement VSX for a low-power single-core in-order RISC-V processor running at 200MHz. Details of the architecture are listed in Table I(a). We implemented VSX in RTL (Register Transfer Level) using SystemVerilog HDL and synthesized it to a commercial 45nm technology using Synopsys Design Compiler. The cache power and area were obtained using CACTI. Compared to the baseline core and caches, **VSX exhibits 2.13% area and 1.15% power overhead**. The minimal overhead of VSX clearly enables its adoption in resource-constrained systems.

We modeled the proposed VSX microarchitecture using the gem5 [21] cycle-accurate architectural simulator. All benchmarks except DNNs were implemented in C++ and run directly on the gem5 simulator to obtain performance measurements as well as output accuracy. In case of DNNs, Caffe [22] — a widely-used deep learning framework — was coupled with gem5 in a manner that allowed performance measurement using gem5 and accuracy measurement using Caffe. Inputs & weights of the target layer were offloaded from Caffe to gem5 where matrix computations were performed, and their outputs were fed back to Caffe as the input of the next layer. All experiments were run in system-emulation mode.

Benchmarks. To evaluate VSX, we use a benchmark suite consisting of 6 ML applications utilizing 4 different classification algorithms, listed in Table I(b). For DNN benchmarks, we applied VSX to only the largest convolution and fully-connected layers *viz.* $conv2/fc6$ for AlexNet, and $conv1\_2/fc6$ for VGG16, respectively. We used classification accuracy *i.e.,* fraction of inputs classified correctly, as our quality metric.

VI. Results

This section presents the results of our experiments to evaluate the effectiveness of VSX.

A. Speedup-Accuracy Tradeoff

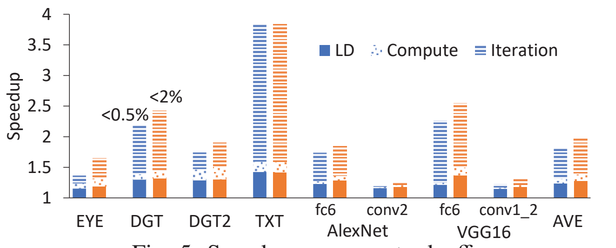


Fig. 5: Speedup-accuracy tradeoff

Figure 5 shows the speedup achieved by VSX for different output quality constraints. The speedup is broken down into different components based on the type of instruction skipped: load, compute, or full iteration. For a negligible loss (<0.5%) in output quality, skipping loads results in a speedup of 1.15×-1.42× (average: 1.24×). Skipping compute instructions corresponding to the skipped loads increases the speedup to 1.17×-1.59× (average: 1.31×). Finally, skipping iterations based on the run-length of asserted VSBs further increases the speedup to 1.19×-3.84× (average: 1.81×). For a relaxed quality constraint of <2%, the achieved speedup increases to 1.19×-1.42× (average: 1.28×) by skipping loads, 1.21×-1.59× (average: 1.39×) by skipping loads and computes, 1.25×-3.84× (average: 1.97×) by skipping iterations as well.

B. Skip Rate Analysis

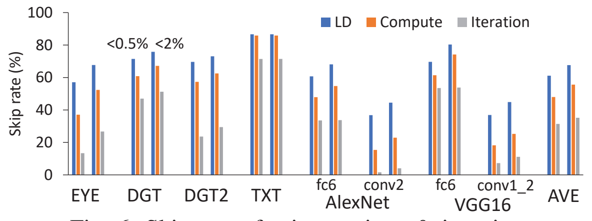


Fig. 6: Skip rates for instructions & iterations

Figure 6 shows the skip rates for load/compute instructions and iterations under different output quality constraints. For a tight quality constraint (<0.5%), skip rates range between 36.8%-86.6% (average: 61.1%) for loads, 15.4%-85.9% (average: 48.0%) for computes, and 1.6%-71.4% (average: 31.4%) for iterations. When the quality constraint is relaxed to <2%, skip rates increase to 44.5%-86.6% (average: 67.6%) for loads, 22.9%-85.9% (average: 55.6%) for computes, and 4.1%-71.5%

(average: 35.2%) for iterations. *TXT* shows the highest skip rate possible for real-world inputs due to its extreme sparsity (>99%), while convolution layers of DNNs show low skip rate due to lack of similarity in their weights.

C. *Speedup Across Different Spatial Distribution of Similarity*

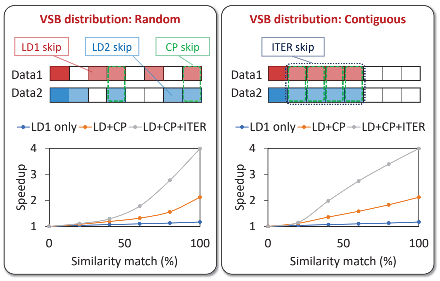


Fig. 7: Speedup *vs.* similarity analysis

Figure 7 illustrates the effect of different spatial distributions of similarity on instruction & iteration skipping. We vary the amount of similarity in the input data-structures of a dot-product kernel and consider two different spatial distributions, *viz., random* and *contiguous*. Contiguous distribution maximizes compute and iteration skipping through the highest possible run-length of VSBs across all loads. The speedups observed for both distributions are similar at low similarity level (low probability of finding VSBs for all loads asserted) and at extremely high similarity level (almost all VSBs are asserted). However, at intermediate similarity levels, the speedup gap between them can be significant (*e.g.*, 1.78× for random vs. 2.74× for contiguous at 60% similarity).

D. *VSX* vs. *Load Value Approximation*

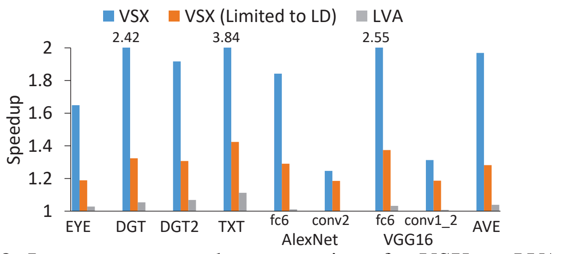


Fig. 8: Iso-accuracy speedup comparison for VSX *vs.* LVA

We compare the speedup benefits of VSX with that of Load Value Approximation [11] (LVA) for <2% quality loss in Figure 8. LVA is an AxC technique that reduces the cache miss latency in GPPs by predicting a load value from previous load values (we use the average of 4 previous values). Every prediction is evaluated afterwards by checking if the predicted value falls within a user-defined *confidence window* of the actual value. This confidence window in turn serves as a knob for modulating performance-accuracy trade-off. Since LVA targets only load instructions, we also show the speedups achieved by VSX when skipping is limited to only load instructions in Figure 8. We observe that the average speedup for VSX in default mode (1.97×) and VSX limited to loads (1.28×) is significantly higher than LVA (1.04×). For a slow single-core processor used in our experimental setup — along with the application kernels showing high spatial locality — cache miss rate is already too low (4.02%) for LVA to optimize. On the contrary, VSX is able to skip a wide range of instructions once the current cache-line is brought up to the L1 data cache, which effectively translates to speedups.

VII. Conclusion

Approximate computing in GPPs requires a holistic approach targeting benefits in compute, memory and the control front-ends. In this work, we propose VSX — a set of lightweight micro-architectural and ISA extensions for GPPs — that exploits value similarity within data-structures for performance improvement. The key idea is to pre-detect similar values in the granularity of a cache-line and use that information to skip fetch-decode-execute of instruction sequence and/or loops, and approximate their result with a previously saved one. We present compiler techniques to transform common ML kernels to use VSX. We evaluate VSX on a low-end in-order RISC-V processor platform and show 1.19×-3.84× speedup across our ML benchmarks at the cost of a 2.13% area overhead, highlighting its applicability to edge platforms.

References

[1] S. T. Chakradhar and A. Raghunathan, “Best-effort computing: Re-thinking parallel software and hardware,” in *Proc. DAC*, June 2010.
[2] S. Mittal, “A survey of techniques for approximate computing,” *ACM Comput. Surv.*, vol. 48, no. 4, Mar. 2016.
[3] S. Venkataramani *et al.*, “Quality programmable vector processors for approximate computing,” in *Proc. MICRO*, 2013, pp. 1–12.
[4] D. Wong *et al.*, “Approximating warps with intra-warp operand value similarity,” in *Proc. HPCA*, 2016, pp. 176–187.
[5] C. Wu *et al.*, “Machine learning at facebook: Understanding inference at the edge,” in *Proc. HPCA*, 2019, pp. 331–344.
[6] S. Sidiroglou-Douskos *et al.*, “Managing performance vs. accuracy trade-offs with loop perforation,” in *Proc. ESEC/FSE*, 2011.
[7] C. Alvarez *et al.*, “Fuzzy memoization for floating-point multimedia applications,” *IEEE Transactions on Computers*, vol. 54, no. 7, pp. 922–927, 2005.
[8] Y. Sato *et al.*, “An approximate computing stack based on computation reuse,” in *Proc. CANDAR*, 2015, pp. 378–384.
[9] S. N. et. al., “Scalable stochastic processors,” in *Proc. DATE 2010*, 2010.
[10] H. Esmaeilzadeh *et al.*, “Architecture support for disciplined approximate programming,” in *Proc. ASPLOS*, 2012, pp. 301–312.
[11] J. S. Miguel *et al.*, “Load value approximation,” in *Proc. MICRO*, Dec 2014, pp. 127–139.
[12] Aurangzeb and R. Eigenmann, “HiPA: History-based piecewise approximation for functions,” in *Proc. ICS*, 2017, pp. 23:1–23:10.
[13] X. He *et al.*, “ACR: Enabling computation reuse for approximate computing,” in *Proc. ASP-DAC*, 2016, pp. 643–648.
[14] A. Raha and V. Raghunathan, “qLUT: Input-aware quantized table lookup for energy-efficient approximate accelerators,” *ACM Trans. Embed. Comput. Syst.*, vol. 16, no. 5s, Sep. 2017.
[15] M. Riera *et al.*, “Computation reuse in DNNs by exploiting input similarity,” in *Proc. ISCA*, 2018, pp. 57–68.
[16] 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.
[17] M. Samadi *et al.*, “SAGE: Self-tuning approximation for graphics engines,” in *Proc. MICRO*, 2013, pp. 13–24.
[18] ——, “Paraprox: Pattern-based approximation for data parallel applications,” in *Proc. ASPLOS*, 2014, pp. 35–50.
[19] Y. Kim *et al.*, “Data subsetting: A data-centric approach to approximate computing,” in *Proc. DATE*, 2019, pp. 576–581.
[20] S. Sen *et al.*, “SparCE: Sparsity aware general-purpose core extensions to accelerate deep neural networks,” *IEEE Trans. Comput.*, vol. 68, no. 6, p. 912–925, Jun. 2019.
[21] N. Binkert *et al.*, “The gem5 simulator,” *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011.
[22] Y. Jia *et al.*, “Caffe: Convolutional architecture for fast feature embedding,” in *Proc. ACM Inter. Conf. on Multimedia*, 2014, p. 675–678.