QAPPA: A Framework for Navigating Quality-Energy Tradeoffs with Arbitrary Quantization

Thierry Moreau[†], Felipe Augusto[‡], Patrick Howe[†], Armin Alaghi[†] and Luis Ceze[†]
[†]University of Washington [‡]University of Campinas

ABSTRACT

Quantization naturally exposes knobs in hardware to trade fidelity for efficiency: the more bits that are used to represent the data, the higher the storage and computation overheads. With the emergence of approximate computing research, we set out to answer the following question: how effective is quantization in trading off quality for efficiency, and how does it compare with other approximation techniques? This paper makes the case for quantization as a general approximation technique that exposes fine quality vs. energy tradeoffs and provides practical error guarantees. We assume *arbitrary* quantization levels, and focus on the hardware subsystems that are affected by quantization: memory and computation.

We present QAPPA (Quantization Autotuner for Precision Programmable Accelerators), an autotuner for C/C++ programs that automatically tunes the precision of each arithmetic and memory operation to meet user defined application level quality guarantees. QAPPA integrates energy models of quantization scaling mechanisms to produce bandwidth and energy savings estimates for custom accelerator designs. We use the analysis produced by QAPPA to compare the effectiveness of arbitrary quantization against voltage overscaling and neural approximation. Our analysis shows that when using the right quantization scaling mechanisms in hardware, quantization provides significant energy efficiency benefits over voltage overscaling and comparable energy efficiency gains over neural approximation. Additionally, quantization offers more predictable error degradation and fully tunable error bounds.

1. INTRODUCTION

Energy efficiency is a first-class concern in data centers, embedded systems and sensory nodes. To improve energy efficiency, numerous cross-stack techniques have been proposed to bring hardware and software systems closer to their quality-energy Pareto-optimal design point. Navigating quality-energy tradeoffs is fundamental to digital systems design, and often starts with data representation, i.e. how to map a set of real values to a compact and finite digital representation. This process is called *quantization*, and is essential in keeping computation tractable in digital systems. Quantization offers a natural way to trade quality for energy efficiency by tweaking the number of bits needed to represent data. Using more bits leads to higher fidelity, but also larger compute, data movement and memory overheads.

This paper argues towards adopting arbitrary quantization as a general approximation technique for its effectiveness in delivering smooth quality-energy tradeoffs, and practical error guarantees. Quantization is often overlooked as an effective way to improve quality-energy optimality due to the limited quantization levels available in hardware (e.g. single and double precision floating point), and the large control overheads found in general purpose processors. This paper bypasses those limitations by assuming *arbitrary* quantization, i.e. bit-granular precision tunability, and by targeting hardware accelerators where control overheads are minimal.

We introduce QAPPA (Quantization Autotuner for Precision Programmable Accelerators), a precision auto-tuner for C and C++ programs that finds bit-granular quantization requirements for each program instruction while meeting user-defined application-level quality guarantees (Figure 1). QAPPA leverages ACCEPT [1] in order to guarantee isolation of approximation effects based on lightweight user annotations. We survey a set of hardware precision scaling techniques and evaluate their ability to improve quality-energy optimality using detailed RTL models. We feed those hardware models into QAPPA to identify energy savings opportunities that arise from adopting precision scaling techniques in hardware accelerator designs. QAPPA isolates arithmetic energy savings and memory bandwidth savings, preserving the orthogonality between savings due to specialization and savings due to approximation in hardware accelerators.

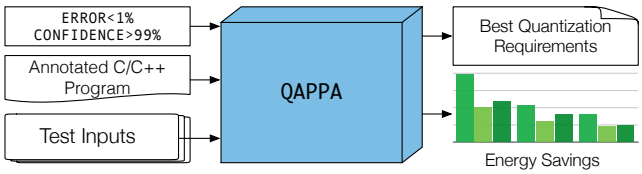

Figure 1: Overview of the QAPPA Framework.

We analyze the PERFECT benchmark suite [2] with QAPPA to unveil significant precision reduction opportunities; about 74%, 57%, and 48% of total precision bits can be dropped to achieve 10%, 1%, and 0.1% average relative error. Respectively, we suggest hardware precision-scaling mechanisms for hardware accelerators that provide 7.7×, 4.8×, and 3.6× energy reduction in arithmetic units, and 4.4×, 3.3×, and 2.8× memory bandwidth reduction.

Finally, we argue that arbitrary quantization compares favorably against other approximation techniques in terms of quality-energy optimality and error guarantees. Our comparative study of approximation techniques includes a SPICE-level characterization of voltage scaling-induced faults, and an analytical evaluation of neural acceleration in terms of hardware resource utilization. Our evaluation reveals that ar-

Technical Report UW-CSE-2017-03-02

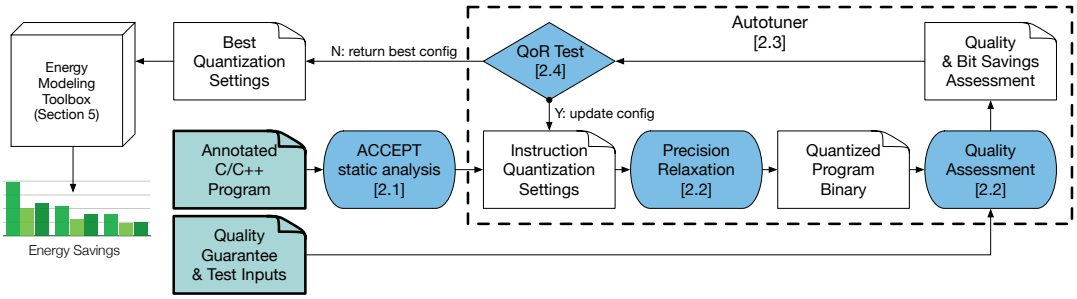

Figure 2: QAPPA Autotuner System Architecture.

bitrary quantization outperforms voltage overscaling in terms of quality-energy optimality, and provides performance that is on par with neural acceleration.

We summarize this paper's contributions below:

- We present QAPPA, an LLVM-based precision tuning framework for C/C++ programs that frees the programmer from the task of fine-tuning program accuracy by inferring instruction granular quantization settings from application-level quality requirements.
- We conduct a detailed survey of hardware precision-scaling mechanisms that can be incorporated in hardware accelerator designs to offer smooth quality vs. efficiency tradeoffs.
- We perform a evaluation that compares arbitrary quantization vs. voltage overscaling and neural approximation, and argue towards the adoption of arbitrary quantization in hardware accelerators as an efficient and predictable approximate optimization.

In Section 2 we describe the QAPPA precision autotuning framework. In Section 3 we quantify how much precision can be trimmed off via quantization in the PERFECT benchmarks. Section 4 evaluates different hardware quantization scaling mechanisms and evaluates the energy and bandwidth reduction achieved by those techniques. Finally, in Section 5, we compare arbitrary quantization to voltage overscaling and neural approximation.

2. QAPPA: A PRECISION AUTOTUNER

QAPPA is a precision autotuning framework built using ACCEPT [1], the LLVM-based approximate compiler for C and C++ programs. In a nutshell, QAPPA takes an annotated C/C++ program and user-specified, high-level quality guarantees to greedily derive quantization requirements for each program instruction. We discuss the design and implementation of QAPPA as illustrated in Figure 2. Section 2.1 describes the annotation model used by QAPPA to identify instructions that are safe to approximate and guarantee program safety. Section 2.2 describes how QAPPA instruments programs to quantify quality loss that results from arbitrary quantization. Section 2.3 describes the autotuner search algorithm and how it is used to find quantization requirements. Section 2.4 describes the quality guarantees that QAPPA provides. The energy modeling toolbox is later discussed in Section 4, where we evaluate different hardware techniques that enable energy scaling.

```
0:  void conv2d (APPROX pix *in, APPROX pix *out, APPROX flt *filter)
1:    for (row)
2:      for (col)
3:        APPROX flt sum = 0
4:        int dstPos = …
5:        for (row_offset)
6:          for (col_offset)
7:            int srcPos = …
8:            int fltPos = …
9:            sum += in[srcPos] * filter[fltPos]
10:       out[dstPos] = sum / normFactor
```

Figure 3: Program annotation with `APPROX` type qualifier. Variables that are safe to approximated are annotated by the user. The compiler then infers the program instructions that can be approximated.

2.1 Annotation Model and Static Analysis

QAPPA leverages ACCEPT [1] to provide type-safety and error isolation guarantees. These isolation guarantees are essential to prevent crashes or catastrophic errors from occurring. QAPPA utilizes the `APPROX` type qualifiers for approximate data used by ACCEPT. Consequently, it is the programmer's responsibility to annotate what variables hold data that is safe to approximate. The compiler then uses flow analysis to infer which instructions are *approximable* from data annotations.

Figure 3 shows how one would annotate a simple convolution kernel. Intuitively, data types such as pixels and filter coefficients can be relaxed, but integer variables that are used to index arrays should remain precise to avoid out-of-array writes. In the convolution example, the compiler infers that the instructions that perform convolution are safe to approximate (instructions from `1.9` and `1.10`). In addition, it identifies that the loads from the image source and the stores to the image destination are also safe to approximate. These approximable instructions will later be used by the autotuner as *knobs* to minimize precision in the target program.

2.2 Assessing Quantization Effects

The QAPPA autotuner relies on a trial-and-error approach to find locally optimal quantization settings that satisfy user-defined accuracy metrics. In order to properly assess quantization effects on a given program execution, QAPPA statically instruments the target program with code that applies arbitrary quantization to individual arithmetic and memory instructions. This can be done in LLVM by replacing all uses of a given static single assignment (SSA) register with its quantized counterpart. In order to perform floating point to fixed point conversion, QAPPA performs an initial dy-

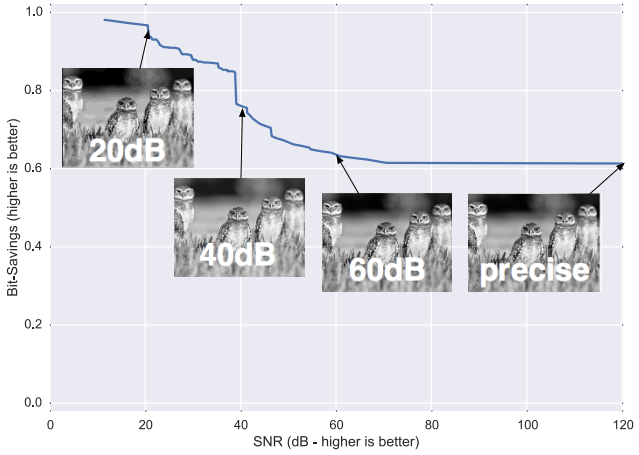

Figure 4: The QAPPA autotuner will seek to maximize bit savings.

namic profiling step on the target program by measuring the value range of each variable.

The degree of quantization and the rounding policy (i.e. up, down, towards zero, away from zero, nearest) are defined for each static instruction in a quantization settings file. The quantization settings dictate how QAPPA applies varying levels of quantization to each instruction in the target program. The instrumented program gets compiled by QAPPA to produce an approximate binary. The approximate binary can then be executed on user-provided input datasets to produce output data on which to quantitatively assess quality degradation with user-defined quality metrics.

2.3 Autotuner Design

The goal of the autotuner is to maximize quantization while satisfying user-specified quality requirements.

Bit Savings.

We define *bit savings* as a hardware-agnostic metric that quantifies how much total precision can be trimmed-off in a program over its execution. QAPPA attempts to maximize bit savings while keeping application accuracy within user-specified margins, as shown in Figure 4.

Bit savings are calculated with the following formula:

$$BitSavings = \sum_{i=1}^{N} \frac{(r_i - q_i)}{r_i} \times \frac{e_i}{\sum_{j=1}^{N} e_j}$$

where r_i and q_i denote the precision in bits of the reference, and quantized instruction i, e_i denotes the number of times instruction i executes, and N denotes the total approximable instructions in the target program. For instance, if a program executes only one single precision floating point instruction, and that QAPPA quantizes that instruction down to 6 bits, the total bit savings will be $\frac{32-6}{32} \times \frac{1}{1} = 81.25\%$.

Autotuner Search Algorithm.

The challenge in the design of an arbitrary quantization autotuner lies in the exponentially large problem search space. Let us consider a program containing m static instructions, where each instruction can be tuned to n different precision levels. In order to find a globally optimal configuration that maximizes bit savings, the autotuner needs to traverse an exponential search space with n^m possible quantization settings, each with different tradeoffs between quality and bit savings. Instead of resorting to a brute-force search to find the optimal configuration, we use a greedy search which finds a local optimum in $O(m^2 * n)$ worst-case time by selecting the path of least quality degradation.

The greedy iterative search algorithm is similar to the approach proposed in Precimonious [3] which uses a trial and error tuning approach to selecting the precision of floating point data. At each step of the search, the QAPPA autotuner identifies the instruction that affects output the least, and relaxes its precision by a single bit. The autotuner repeats the process until it finally reaches a point where decreasing the precision of any instruction violates user-defined quality requirements. We discuss the different quality tests that can be used to guide this search process in Section 2.4. Finally, the autotuner reports locally-optimal instruction quantization settings along with bit savings estimates. Those quantization settings can then be fed into an energy modeling toolbox, which we discuss later in Section 4.

2.4 Quality of Result (QoR) Guarantees

Approximation techniques are only practical if they provide *accuracy guarantees* to the programmer. Guarantees are used as a contract between the tools and the programmer to ensure that the relaxations applied by the tool to the target program will not violate QoR requirements. Guarantees can come under different forms: empirical, statistical and hard guarantees.

Hard guarantees provide the strongest guarantees by assuming worst-case error accumulation. A method to ensure hard guarantees is interval analysis [4], which can be applied to small functions that do not exhibit asymptotic behavior or long chains of operations that could lead to high error accumulation. While hard guarantees are the most desirable to the user, they assume worst-case error accumulation, which are often not representative of real-world inputs. For that reason, QAPPA offers empirical or statistical guarantees.

Empirical Guarantees.

Empirical guarantees provide guarantees that are as good as the datasets provided by the user. This puts more pressure on the programmer to provide satisfactory input coverage, akin to what test engineers do in industry to ensure that code is properly tested, or that learning models are properly trained. This class of guarantees are prevalent in approximate computing literature, due to the complexity involved in providing stricter guarantees [5, 1].

QAPPA provides empirical guarantees by default. The user has to provide a training dataset, and a validation dataset. QAPPA's autotuner traverses the search path of least quality degradation measured on the training input set, but decides when to stop its search when error thresholds are violated on the validation dataset. Having disjoint test and validation sets prevents overfitting issues. QAPPA also provides statistical guarantees, which we discuss next.

Statistical Guarantees.

Statistical guarantees provide a way to reason about unlikely quality violations. Some applications scenarios may tolerate rarely occurring errors if that means achieving significant energy savings. While statistical guarantees make the most sense in the context of non-deterministic approximations [6] and statistical sampling-based approximations [7], they can also be used on deterministic techniques [8]. In the latter case, the rarely occurring quality violation would be the result of a corner case input that would lead to worst case error accumulation.

We augment QAPPA to provide statistical error guarantees in the form of confidence intervals. For example, a confidence interval may imply that the output has an error of at most 10% with a confidence that is equal or greater than 95%. Such statistical guarantees require the user to specify an typical input distribution from which to sample random inputs.

To derive a statistical guarantee, QAPPA measures $N_{violation}$, the number of times the error has exceeded a given error bound δ across N input samples that it has sampled from the user-provided distribution. Then, QAPPA uses the Clopper-Pearson interval [9] to find an upper bound ε of the probability of getting errors that are larger than δ. We have:

$$\varepsilon = \beta(1-\frac{\alpha}{2}; N_{violation}+1, N-N_{violation}) \qquad (1)$$

where β denotes the beta distribution and α is a constant that determines the confidence of the Clopper-Pearson interval. In all our experiments, we set $\alpha = 0.01$. Equation 1 entails

$$Pr[error < \delta] > 1-\varepsilon$$

in which $Pr[*]$ denotes the probability of an event.

3. PERFECT APPLICATION STUDY

We use QAPPA on the PERFECT benchmark [2] kernels to quantify the opportunity for quantization on compute intensive workloads. We answer the following questions: (Section 3.3) How long does the autotuner take to run on the target program? (Section 3.4) How does increasing the strength of guarantees diminish opportunities for precision reduction? (Section 3.5) What dynamic portion of those applications is safe to quantize? (Section 3.6) For the set of instructions that can be relaxed, how much precision can be dropped at different quality constraints? (Section 3.7) How does increasing the strength of guarantees diminish opportunities for precision reduction?

3.1 Benchmark Overview

PERFECT is a benchmark suite composed of compute-intensive application kernels that span image processing, signal processing, compression, and machine learning.

For instance, the Wide Area Motion Imagery (`WAMI`) application represents a typical processing pipeline performed on giga-pixel scale imagery. WAMI comprises an RGB image generation kernel based on the debayer algorithm, an image registration kernel based on the Lucas-Kanade algorithm, and a change detection algorithm based on Gaussian Mixture Models. Table 1 provides an overview of the PERFECT kernels.

3.2 Quality Assessment

For quality assessment, we follow the PERFECT manual guidelines for quality assessment [2], and use a uniform Signal-to-Noise Ratio (SNR) quality metric across all benchmarks to measure quality degradation.

$$SNR_{dB} = 10\log_{10}\left(\frac{\sum_{k=1}^{N}|r_k|^2}{\sum_{k=1}^{N}|r_k-q_k|^2}\right) \qquad (2)$$

The formula used to assess SNR in our benchmarks is provided in Equation 2, where r_k and q_k denote the k^{th} reference and quantized output value. SNR provides an average measure of relative error. It is also worth noting that SNR measures error in a logarithmic scale, i.e. an increase of 20dB corresponds to a $10\times$ relative error reduction. Some kernels do not use an SNR metric by default: `gmm` of the WAMI benchmark measures the number of foreground pixels that have been misclassified. For the sake of uniformity, we convert the classification metric to a logarithmic scale.

3.3 Annotation Effort

The QAPPA framework relies on ACCEPT to apply quantization on program instructions that are deemed to be safe to approximate. The set of approximable instructions are identified via data type annotations by ACCEPT, as discussed in Section 2.1. ACCEPT dictates that approximations must be applied as an opt-in decision, i.e. if the programmer does not annotate any variables in her target program, no precision reduction will take place. This places the burden of expressing to the compiler what data can be affected by approximation on the user. We argue that the burden is necessary to ensure the safety of a program [10]. Thankfully, the code annotations effort is reasonable: we counted the amount of code annotations that we had to insert in each PERFECT kernel, which are enumerated in Table 1 under the "User Annotations" column. Overall, annotations were minimal for each kernel. Most of the time, it came down to annotating all floating point variables and integer variables that hold data (as opposed to an address or index) as approximate.

3.4 Autotuner Runtime

Table 1 summarizes the runtime overhead of the autotuner. The autotuner runtime is dictated by how many steps the autotuner gets to run and how much slower the instrumented approximate program runs at each autotuning step. The goal of the quantization instrumentation step is to faithfully emulate the error resulting from quantization, not to improve performance of the original program. We report an at-most $12.3\times$ slowdown from instrumentation under the "Instrumentation Overheads" column. We report the total number of search steps taken under the "Autotuner Search Step" column where we used a 40dB target.

Table 1 summarizes the total autotuner overhead as a multiple of the original program runtime under the "Autotuner Runtime" column. At worst, the autotuner will take $10{,}000\times$ longer to perform the precision tuning compared to the original runtime, but in the common case it takes about $1000\times$ longer. This runtime overhead isn't too bad considering that we ran the autotuner on microbenchmarks which take less than a second to run, and that this slowdown is compara-

Table 1: PERFECT overview and results summary.

App.	Kernel	Use Case	User Annotation Count	Static Quantized Instruction Count	Dynamic Quantized Instruction Count	Instrumen -tation Overheads	Autotuner Steps (40dB)	Autotuner Runtime
PA1	2D Convolution	Convolutional NNs	6	6	33%	8.9x	26	233x
	DWT	JPEG compression	10	27	44%	3.3x	94	315x
	Histogram Eq.	PDF estimation	12	13	50%	1.5	71	109x
STAP	Outer Product	Covariance Estimation	26	142	81%	10.3x	1143	11762x
	System Solver	Weight Generation	47	77	77%	10.1x	929	9420x
	Inner Product	Adaptive Weighting	41	84	83%	10.5	974	10256x
SAR	Interpolation 1	Radar	25	42	65%	6.4	402	2588x
	Interpolation 2	Radar	21	41	50%	6.5x	528	3437x
	Backprojection	Radar	18	45	82%	6.2x	569	3517x
WAMI	Debayer	Photography	22	124	31%	12.3x	228	2793x
	Lucas-Kanade	Motion Tracking	34	129	51%	4.3x	772	3322x
	Gaussian MMs	Change Detection	25	134	58%	8.1x	107	870x
Required	FFT-1D	Signal Processing	18	43	49%	1.1x	578	642x
	FFT-2D	Signal Processing	18	43	49%	3.1x	1084	3357x
Average			23	68	57%	5x	338	1836x

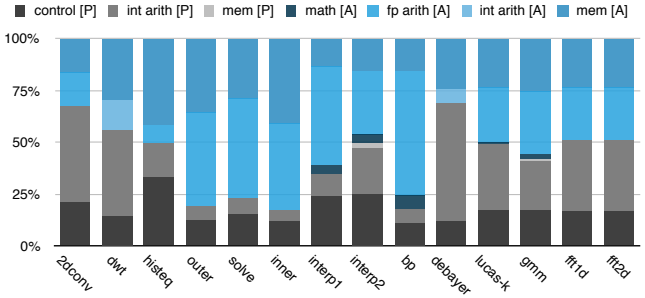

Figure 5: Dynamic instruction category mix of the PERFECT kernels. The approximable instructions are colored in shades of blue, and the precise instructions categories are colored in gray.

ble to the slowdown that many architectural simulators introduce. The QAPPA autotuner was designed to be run once on programs of interests, but we plan to improve its runtime to make it more practical across more challenging applications.

3.5 Approximation Opportunity

Table 1 summarizes application characteristics of the PERPECT kernels derived using QAPPA. The "Static Quantized Instruction Count" column lists the number of static instructions that are safe to approximate according to QAPPA. Each approximable instruction serves as a knob that the autotuner can tune to find a precision-minimal configuration that meets quality requirements. The more precision knobs, the larger the search space for the autotuner.

The "Dynamic Quantized Instruction Ratio" is the ratio of approximable instructions to total instructions, measured over the dynamic execution of the target kernel. The higher the ratio, the larger the opportunity to apply quantization in a given program. Figure 5 shows a detailed instruction category breakdown for each PERFECT kernel. Each category is split between approximable and precise classes, which are respectively colored in blue and gray. The approximate instruction ratio is on average 64% which indicates that the PERFECT benchmark suite is a compelling target for approximate computing.

More importantly, the approximable instructions are for the most part composed of *expensive* operations, such as floating-point arithmetic, loads and stores to memory, and standard C math functions (LLVM IR treats math functions as instructions since back-end architectures may or may not have hardware support for those). Most floating-point and memory operations can be approximated. The kernels mostly access memory to store data, rather than pointers, which are more common in graph applications where pointer-chasing is necessary. The bulk of the precise instructions are composed of control instructions and integer arithmetic used for address computation, neither of which can be approximated without compromising the safety of the program.

3.6 Bit Savings

Figure 6 shows the aggregate bit-savings obtained on approximable instructions that QAPPA was able to obtain on each PERFECT application kernel, on SNR targets from 100dB down to 20dB (0.001% up to 10% average relative error). In general, the lower the quality target, the higher the bit-savings. On average, a 74%, 57%, and 48% average bit-savings can be obtained at 20dB, 40dB and 60dB respectively (10%, 1%, and 0.1% average relative error). We observe that integer benchmarks (`2dconv`, `dwt`, `histeq`, and `debayer`) offer relatively high bit-savings at high SNR requirements (100dB). This is indicative of the common use of wide integer types (e.g. 32-bit) to handle narrow pixel data (e.g. 8-bit) for image processing benchmarks (`PA1`). We also notice that `changedet` provides minimal bit-savings until we lower error to 40dB and 20dB error (1% and 10% misclassification rate). The remaining floating-point kernels all exhibit a smooth tradeoff relationship between bit-savings and quality. We observe that quantization can meet very stringent quality thresholds that are often not achiev-

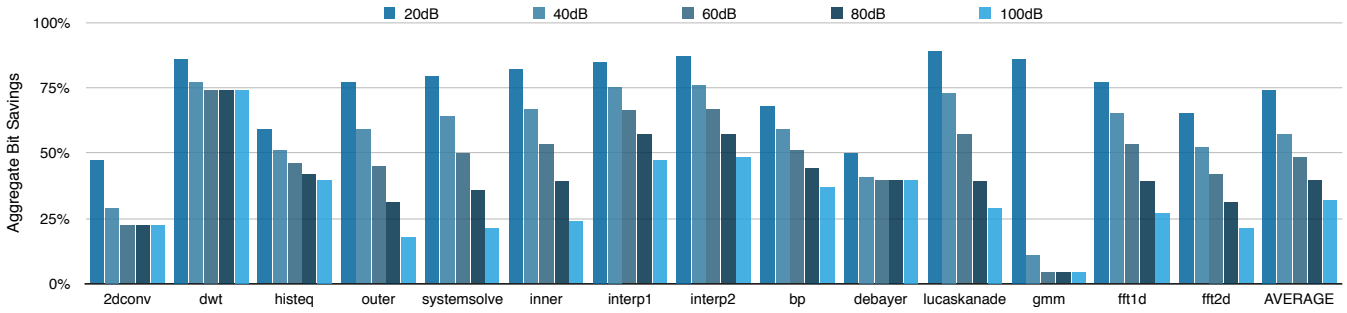

Figure 6: Aggregate bit-savings for 14 PERFECT kernels over a 20dB to 100dB SNR range.

Table 2: Bit-savings loss from using a empirical guarantee to statistical guarantee at 90% and 99% confidence. We vary the quality target at medium (20dB) a high (40dB) settings on the PA1 kernels.

PA1 Kernel	medium quality (20dB)		high quality (40dB)	
	conf>90%	conf>99%	conf>90%	conf>99%
2D Conv.	-4.10%	-13.25%	-4.37%	-8.73%
DWT	-12.50%	-22.47%	-2.51%	-2.73%
Hist. Eq.	-3.02%	-7.35%	-2.91%	-6.76%

able with other approximation techniques. For instance, the PERFECT manual recommends 100dB (0.001% relative error) degradation as a quality target from applying compiler optimizations. We do not know of any approximation techniques that can meet such stringent accuracy guarantees.

3.7 Guarantees

In Section 2.4, we discussed two ways to express QoR guarantees: empirical tests — used so far in this evaluation — and statistical tests, which we discuss in this section. Statistical error guarantees capture the uncertainty that arises from measuring error in a non-exhaustive way. To express a statistical guarantee, the user needs to provide an error threshold δ, and a confidence threshold $1-\varepsilon$. QAPPA then applies the Clopper-Pearson (CP) test to ensure that both δ and $1-\varepsilon$ are satisfied.

Demanding higher confidence leads to more conservative precision relaxations and thus lower bit-savings. We conduct an experiment to quantify the loss in bit-savings when demanding a statistical guarantee at different confidence levels. The baseline bit-savings for this experiment is obtained using empirical error guarantees. We chose the PA1 kernels to conduct our experiment for two reasons: (1) it was straightforward to produce a generative model for image data, and (2) processing each image requires hundred of thousands of kernel invocations which provided enough samples for QAPPA to run the CP test on at high confidence levels.

We conduct our experiment at two quality levels: a medium quality setting at 20dB (10% error) and a high quality setting at 40dB (1% error). Table 2 shows the bit-saving loss at two confidence levels ($1-\varepsilon = \{90\%, 99\%\}$), relative to the bit-savings obtained with empirical guarantees. We evaluate the bit-savings loss using both quality levels, with error thresholds ($\delta = \{10\%, 1\%\}$). Overall, we notice a reduction in bit-savings going from empirical guarantees to statistical guarantees, as the confidence interval increases. These results confirm that stronger statistical guarantees diminish bit-savings returns.

4. DYNAMIC QUANTIZATION SCALING

We survey *dynamic quantization* mechanisms in hardware and discuss the savings in arithmetic energy and memory bandwidth that these mechanisms achieve on hypothetical accelerator designs executing the PERFECT kernels. We isolate the subsystems that are affected by quantization, namely the arithmetic substrate and the memory subsystem. *Arithmetic energy* denotes the fraction of energy that is consumed by arithmetic units in a given hardware design, e.g. ALUs and processing elements. What this study does not focus on are control overheads, which are specific to a given hardware implementation.

The aim of this study is to motivate the adoption of quantization scaling mechanisms in hardware accelerators, where data bandwidth requirements far surpass the instruction bandwidth requirements. General purpose processors spend much of their energy budget in instruction fetching and decoding. Augmenting the ISA of a general processor with bit-granular quantization settings would counteract much of the energy savings that quantization would enable. Thus, this survey targets designs such as vector processors, systolic arrays, or fixed-function accelerators that could incorporate dynamic quantization scaling mechanisms in order to respond to dynamic energy or quality constraints.

4.1 Scaling Quantization in Compute

We evaluate two quantization scaling hardware mechanisms that provide energy reduction on quantized arithmetic operations. The first technique, operand narrowing, aims to minimize power by reducing transistor switching [11] on wide compute units. The second technique, bit slicing (or operator narrowing), utilizes narrow compute unit in parallel to time-multiplex the computation of wider operations, effectively scaling throughput with precision on data-parallel workloads [12]. We compare the energy savings obtained by each technique at different operand quantization levels, over a standard 32 bit arithmetic unit.

Reducing Power with Operand Narrowing.

Operand narrowing is a precision scaling technique that can reduce dynamic switching in standard bit-parallel arithmetic units [11]. The idea is to apply quantization on the

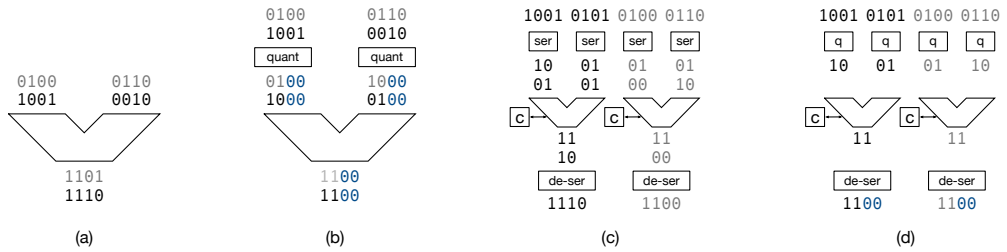

Figure 7: Quantization scaling mechanisms overview. (a) Default wide addition on wide adder. (b) Narrow addition on wide adder. (c) Wide addition on narrow adder (d) Narrow addition on narrow adder.

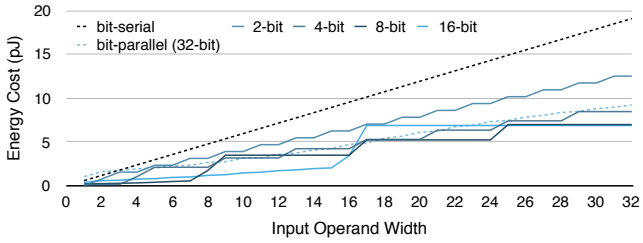

Figure 8: Energy vs. precision relationship for precision-scaled multiplier designs (32 bit baseline).

input operands of the arithmetic units by zeroing the LSBs that correspond to the desired quantization level. This in turns limits the amount of transistor switching in the arithmetic unit's logic, as the lower slices of the datapath remain inactive.

Figure 7.b shows how operand narrowing sets the least significant bits (LSBs) of the input operands to zero, to underutilize the arithmetic unit's lower slice. LSB-zeroing is the precision scaling mechanism proposed in the Quora vector processor [11]. While operand narrowing reduces the amount dynamic power, it does not provide throughput improvements. Next, we discuss a quantization scaling technique that achieves throughput scaling when data parallelism is available.

Increasing Throughput with Bit Slicing.

Bit slicing is a technique used to perform wide arithmetic operations using narrower arithmetic units. The advantage of bit slicing lies in its ability to scale throughput nearly linearly with precision requirements. Given an narrow n bit adder, a wide m bit addition can be done in $O(m/n)$ time, while an m bit multiplication can be done in $O(m/n)$ time on a an $m \times n$ multiplier. Bit slicing reduces arithmetic unit power while increasing computational delay, thus making baseline precision computation on a wide ALU and a bit-sliced ALU roughly equivalent in terms of energy. Bit slicing excels at reducing energy at lower-precisions settings, since lower precision lead to lower computation delays. Bit slicing comes at a cost however, which we will refer to as the *bit-serialization tax*. The bit-serialization tax is attributed to the extra registers needed to time-multiplex a narrow compute unit for wide computation. Bit slicing is best applied in applications that have SIMD parallelism, where bit parallelism can be exchanged for increased SIMD parallelism. This results in designs that have similar area footprint and the ability to dynamically increase throughput as precision requirements go down [12].

4.2 Quantization Scaling Energy Evaluation

Methodology.

We synthesize adder and multiplier designs of varying widths using the Synopsys Design Compiler with the TSMC-65nm library. To model power, we collect switching activities in simulation when adding/multiplying input operands streams of varying widths, from 1 bit to 32 bits. We set a target frequency of 500MHz and perform place and route on each simulated design with ICC. We use PrimeTime PX to accurately model the impact that switching activity has on power.

Multiplier Case Study.

We evaluate the energy cost of performing arithmetic operations on input streams with varying bit widths. The energy per operation vs. input width relationship for a 32-bit multiplier design is shown as a dotted black line in Figure 8. The linear increase in energy reflects an increase in switching activity when the multiplier processes wider input operands.

Next we look at bit slicing: we vary the granularity at which computation is sliced from 1 bit (bit serial) to 32 bits (bit parallel). The relationship between the energy cost and the input width for a 32 bit multiplier is shown as colored lines in Figure 8 for different bit slicing granularities. When the input operand width is narrower than the arithmetic unit width, the energy scales linearly with the input width because of lower switching activity. Conversely, when the input operand width exceeds the width of the serial arithmetic unit width, the energy increases discretely at every n-bit increments, where n denotes the width of the slice. Bit-serial evaluation – i.e. arithmetic unit width of 1 – is a corner case where the relationship between energy and operand width is linear. It is worth noting that no single slice width is produces better results than others across all input widths.

Energy Evaluation on PERFECT.

We use the PERFECT benchmark suite to guide our choice of an energy-optimal precision scaling mechanism at different quality targets from 60dB down to 10dB.

Figure 9 shows energy savings across all PERFECT benchmarks over a standard arithmetic unit executing 32 bit arithmetic operations. Performing operand narrowing exclusively as in Quora [11] on a bit-parallel arithmetic unit results in

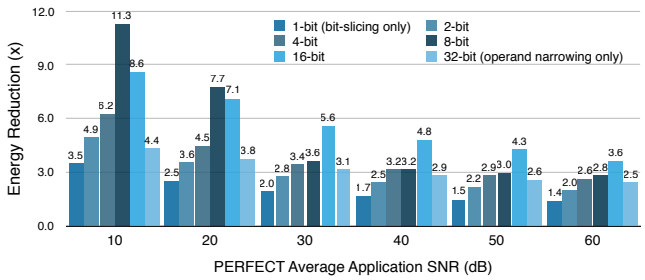

Figure 9: Arithmetic energy reduction on the PERFECT benchmark at different bit slicing granularities and at different SNR targets (higher is better).

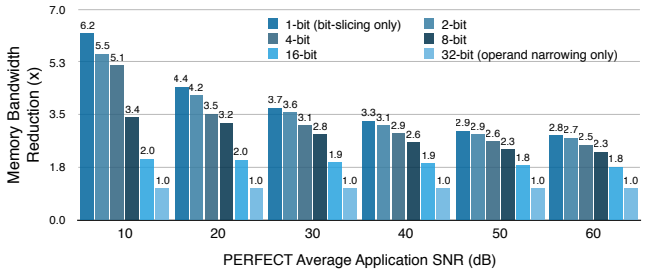

Figure 10: Ideal bandwidth reduction on PERFECT benchmark suite at different data packing granularities and at different SNR targets (higher is better).

significant energy reduction over the precise, non quantization scalable baseline: 3.8×, 2.9× and 2.5× at 20dB, 40dB and 60dB respectively. These energy reductions are improved by combining bit slicing and operand narrowing: a slice width of 16 bits yields optimal energy reductions by 3.6×, 4.8× at 40dB and 60dB while a slice width of 8 bits yields 7.7× energy reduction at 20dB over the baseline arithmetic unit. Finally, we make the observation that applying bit slicing at a 1 bit granularity yields suboptimal energy results at all quality targets.

4.3 Scaling Quantization in Memory

Much of the energy spent in processors and accelerators is associated with data movement to and from memory [13, 14]. Scaling precision in programs can help mitigate memory bandwidth requirements.

Data packing can maximize bandwidth efficiency at arbitrary precision settings. Recent work has proposed hardware packing and unpacking mechanisms to store variable precision weights in neural network accelerators [15]. The idea is to store variable precision data into fixed-width memory, by packing data at a coarse granularity (e.g. an array of coefficients) to mitigate overheads. Figure 11 shows how reduced precision data can be efficiently padded in fixed-width SRAM modules, unpacked for processing, and re-packed before being stored to SRAM again. This results in more effective use of bandwidth and storage, but adds complexity when accessing data. This complexity can be mitigated in hardware accelerators that perform regular data access on large portions of memory, where precision settings can be set on coarse structures. We assume that the data is read and

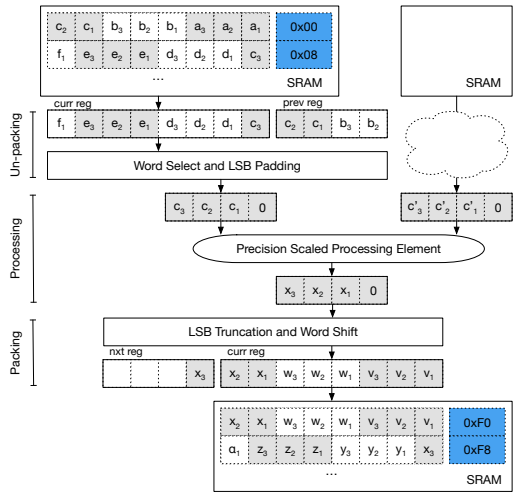

Figure 11: Example of quantization-scalable pipeline: memory packing and unpacking mechanism used in Proteus [15] combined with operand narrowing used in Quora [11]. The input and output data can be loaded in its packed format to save memory bandwidth.

written to DRAM in a dense format, simplifying the on-chip to off-chip storage communication pipeline.

Applying quantization to data can significantly reduce memory bandwidth. Figure 10 shows bandwidth savings on a cache-less accelerator. We vary the data packing granularity from 1 to 32 bits and derive the resulting bandwidth reduction. A data packing granularity of 1 bit can achieve 4.4×, 3.3×, and 2.8× average memory bandwidth reduction on the PERFECT kernels at 20dB, 40dB and 60dB. Data packing at fine granularities can increase both software and hardware overheads for packing and unpacking. A hardware designer might therefore want to align the data packing granularity with the bit slicing width of the precision scalable compute units to minimize control overheads. The optimal data granularity can be determined by the target system energy breakdown between memory, computation, and control which differs for different classes of accelerators and workloads.

5. APPROXIMATION STUDY

In this section, we conduct a comparative evaluation of approximation techniques. We evaluate precision reduction against nondeterministic voltage overscaling [16, 17] and coarse grained neural approximation [18], and compare the quality vs. energy tradeoffs achieved with each technique.

5.1 Voltage Overscaling

We compare the energy savings obtained by quantization against voltage overscaling and contrast the energy savings obtained at different quality targets on the PERFECT benchmark kernels.

Motivation: Determinism vs. Nondeterminism.

Nondeterministic approximations can introduce errors in a random or pseudo-random fashion [16, 17, 19, 20, 21]. While nondeterministic approximations pose a testing and debugging challenge, they can be modeled using probabilistic distributions [6]. We investigate nondeterministic *voltage*

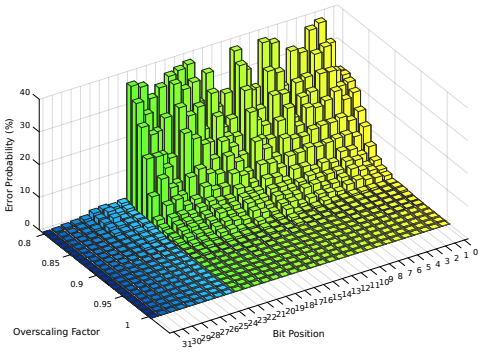

Figure 12: Bit-flip probabilities of each output bit for a single-precision floating point adder at voltage overscaling factors [0.8-1.0]. Sign and exponent bits are in blue, mantissa bits are in green/yellow.

overscaling, a popular approximation technique that reduces compute power at the risk of increasing timing violations. Our evaluation of voltage overscaling relies on (1) characterizing the energy vs. error relationship of voltage overscaling and (2) analyzing how low level timing violations affects application quality.

Characterizing Overscaling Error.

We quantify the effects of voltage overscaling on fixed point and floating point arithmetic designs taken from the Synopsis DesignWare IP library. We simulate those circuits in CustomSim-XA, built on top of FastSpice to perform transistor level power and fault characterization. The circuits are built in Synopsys Design Compiler with a 65nm process and synthesized using a timing constraint of 2GHz. Registers latch the inputs and outputs of the arithmetic units and a synchronizer is used to settle errors caused by metastability. We synthesize a parallel prefix architecture for the fixed point adder and a Booth-encoded Wallace-tree architecture for the fixed point multiplier. We generate 10^5 random input pairs as stimuli to the circuits and profile timing violation errors at three representative voltage overscaling factors (0.95×, 0.90×, and 0.84×), corresponding to 10%, 20%, and 30% power savings respectively. We measure the probability of a timing violation induced bit-flip for each output bits to produce a statistical error model of the voltage overscaled circuit. Figure 12 shows the bit-flip probability distribution for a floating point adder, measured at different voltage overscaling factors, with different color coding to highlight the sign, exponent and mantissa bits.

Comparative Evaluation on PERFECT.

We feed the error models derived above to QAPPA's error injection framework to quantify the effect of voltage overscaling on the application output. We execute each benchmark 100 times on the same input data to obtain an error distribution.

The results of the experimental runs are displayed in Figure 13 and show the effects of voltage overscaling on application quality at 10%, 20% and 30% energy savings. Applying the same voltage overscaling factor to each PERFECT kernel can lead to vastly different errors because of nondeterminism. Integer benchmarks such as `dwt` and `debayer` are

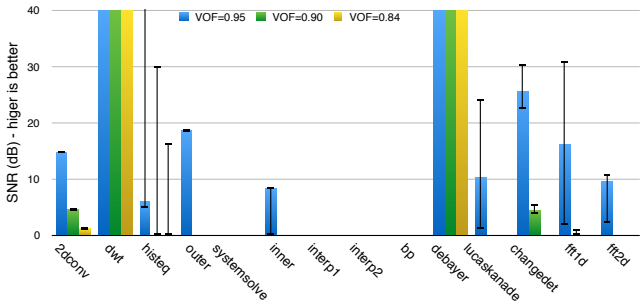

Figure 13: PERFECT kernel SNR at voltage overscaling factors of 0.95, 0.90 and 0.84 corresponding to 10%, 20% and 30% energy savings. SNR is measured collected over 100 runs, values represent median SNR, and error bars represent min and max error.

mostly unaffected by overscaling. The integer circuits have shorter critical paths than their floating point counterparts, and therefore are less affected by voltage overscaling. Other benchmarks including the `SAR` kernels and `systemsolve` produce data that contain erroneous output values (`inf` and `NaN`) which lead to a 0dB SNR. Voltage overscaling does well on simple single-stage functions (`2dconv`), in which errors have localized effects. Multi-stage kernels (`lucaskanade`) on the other hand pose a challenge since errors can propagate and snowball into large output errors.

Discussion.

Quantization provides better energy efficiency at preferable SNR levels for all PERFECT kernels. In addition, the deterministic nature of quantization allows for sounder guarantees and more predictable behavior. We conclude that *is it difficult to justify incorporating voltage overscaling in hardware designs without some form of error correction*. The unbounded errors simply don't justify the energy savings. A hybrid approach of combining fine grained precision requirements with error correction mechanisms proposed in [22] could selectively correct a timing violation error based on what bits are affected, thereby reducing the amount of hardware rollbacks. We reserve the evaluation of such error correction mechanisms for future work.

5.2 Neural Approximation

We discuss how quantization scaling could improve the efficiency and programmability of programmable accelerators and compare the energy benefits of quantization against neural approximation. Neural approximation has limited applicability when it comes to approximating arbitrary functions at arbitrarily low error levels. We evaluate the AxBench [23] benchmark suite at suggested error levels (10% relative) to ground the comparison between quantized acceleration and neural acceleration.

Motivation: Fine vs. Coarse Approximation.

Coarse grained approximation attempts to approximate an entire code region using a regression model (e.g. polynomials, neural networks). Neural acceleration [5, 18] uses neural networks to approximate functions via learning, and utilizes hardware accelerators for efficient execution. Much of the

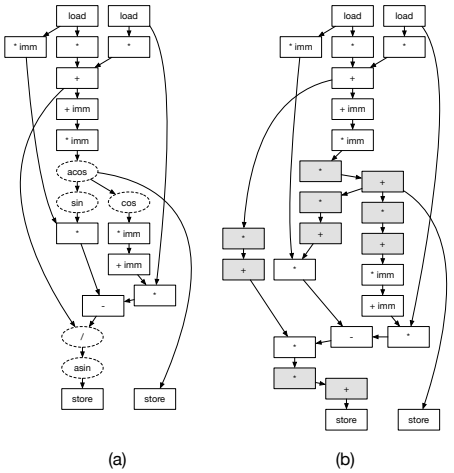

Figure 14: Inverse kinematics kernel: (a) default DFG, (b) optimized fixed point DFG with PWP.

previous studies on neural acceleration have not isolated the efficiency gains attributed to specialization from approximation.

We compare two approximation approaches: (1) fine grained approximation with piecewise polynomial (PWP) approximation of math functions, and (2) neural approximation. In both cases, we assume a hardware accelerator composed of fixed point adders, multipliers and local SRAM storage. We quantify arithmetic energy, and SRAM requirements to draw a cost comparison between the two techniques. We motivate our study with the inverse kinematics (`inversek2j`) function example, which dataflow graph (DFG) is shown in Figure 14.a.

Quantized Acceleration.

We use QAPPA to derive the quantization requirements in each target application at the error rate recommended by AxBench. Quantization provides an opportunity to significantly reduce the cost of standard math function invocations. We leverage QAPPA to derive the accuracy requirements and the input range of standard math functions (e.g. *cos*, *sqrt*, reciprocal etc.) in each target program. We use those requirements to produce piecewise polynomial approximations with a custom math approximation toolbox that we built in Python. The degree of the polynomial dictates computational requirements, while the number of pieces dictates the memory requirements for storing the polynomial coefficients. The DFG of an example quantized program is shown in Figure 14.b. In this example, all nonlinear operators (represented as circles) have been replaced replaced with a piecewise degree-one polynomial approximation.

Neural Acceleration.

Neural acceleration approximates whole functions using neural networks [5, 18, 24, 25]. Neural networks have high internal SIMD parallelism, but are generally computationally and storage demanding [13]. We show the DFG corresponding to a neural network topology used to approximate the `inversek2j` function in Figure 15. It becomes clear that while being highly regular, neural networks consist of many arithmetic operations, and have high memory requirements.

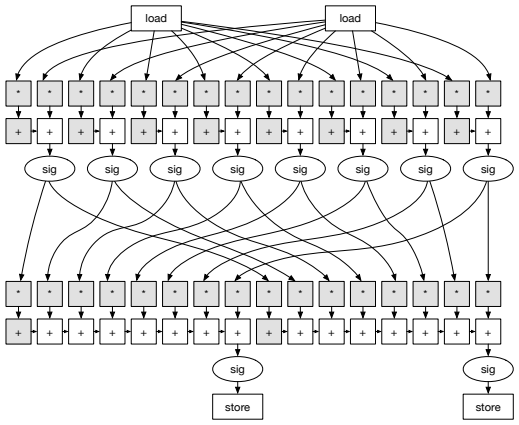

Figure 15: DFG of a neural approximation of the inverse kinematics kernel. Operations that read data from local SRAM are colored in gray.

Comparative Evaluation on AxBench.

We run our study on a set of AxBench [23] benchmarks. We assume a spatially laid-out accelerator design (i.e. each static instruction is mapped to a single processing element, or load/store unit) for each approximation technique and measure hardware efficiency in two key metrics: (1) compute energy and (2) SRAM storage requirements. We use the RTL computation cost models obtained in Section 4 to analytically evaluate energy costs associated with each approximate acceleration technique. We quantitatively measure on-chip SRAM requirements for storing the neural network weights, and piecewise polynomial approximation coefficient tables. Finally we use the neural approximation errors reported in previous literature [18] as quality targets for quantization.

We use two modeling assumptions to estimate the computation and storage costs of neural acceleration. The realistic model based on digital implementations of NPUs [5, 24] assumes 16 bit weights, and a 16-piece linear approximation of the activation function. The optimistic model assumes 8 bit weights, a linear activation function and no quality loss with respect to the realistic model.

We leverage QAPPA to produce a reduced precision quantized program specification for each AxBench kernel. For kernels that have outputs that depend on control-flow, we precompute all branching paths and use predication to resolve the branch outcomes. This approach is tractable in kernels that do not have much control flow divergence. Our compute cost model assumes a quantization scalable 8 bit ALU, that applies a operand narrowing and bit-serial computation depending on the operand width.

We summarize our evaluation of neural approximation vs. reduced precision acceleration in Figure 16. Reduced precision acceleration is more energy-efficient than a neurally approximated acceleration for all of the reviewed AxBench kernels. The storage requirements of the quantized kernels lie between the realistic and optimistic neural network accelerator cost models, except for `blackscholes` where quantized acceleration beats neural acceleration in both cost modeling scenarios.

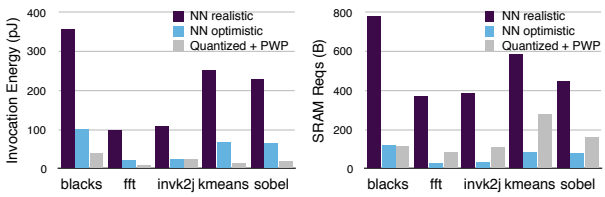

Figure 16: Energy and storage comparison of quantized acceleration vs. neural acceleration on AxBench kernels (lower is better).

Discussion.

While there is not a clear answer as to which technique is more efficient in terms of both energy and storage, we can claim that quantized acceleration offers comparable efficiency benefits to neural acceleration. Neural acceleration provides the benefit of programmability as it requires one hardware accelerator to evaluate any neurally approximated piece of code [24]. However, neural networks have limited success at approximating code at arbitrarily low error levels, as there are no examples in literature that show successful approximations with neural networks below 1% relative error [5, 18, 24, 25]. Quantization and PWP approximation could improve programmability in spatial accelerators by simplifying complex operators such as math functions, down to simple linear operators. The simplified kernel can then be more easily mapped onto a programmable acceleration substrate composed of simple arithmetic functions [26, 27]. Finally, improving the quality guarantees of neurally-approximated programs is the object of much on-going research and remains a challenge for high-dimensional functions [28, 25, 8, 29]. Quantization on the other hand benefits from mature numerical analysis frameworks that provide error analysis and guarantees that programmers are familiar with [30, 31, 4], on top of the empirical and statistical guarantees that QAPPA provides.

6. RELATED WORKS

Approximate Computing.

The emergence of approximate computing research has led to a multitude of hardware and software proposals, along with tools, frameworks and runtimes designed to reason about and mitigate error. Software approximations [32, 33, 34, 35, 36, 37, 38, 7, 39, 40, 41, 42] perform code transformations to trade-off output quality for performance gains on commodity hardware. Hardware approximations [5, 24, 43, 44, 45, 46, 47, 11, 48, 49, 50, 51, 18, 20, 21, 52, 53, 19, 16, 17, 54, 55, 56, 57, 54, 58] on the other hand require alterations at the architecture, microarchitecture, and circuit level to expose quality-efficiency tradeoffs to the software stack. Precision reduction is traditionally considered a software approximation, but requires hardware to support various levels of precision [3, 59, 60]. QAPPA targets arbitrary quantization, which assumes a precision scalable hardware back-end.

Tools and Frameworks.

Precimonious [3] is a dynamic program analysis tool that suggests cheaper floating point type instantiations to improve the performance of floating point functions. QAPPA differs from Precimonious in that it supports approximate type qualifiers to ensure program safety, and that it applies arbitrary quantization to either floating point or integer types. Approxilyzer [61] helps improve hardware resiliency to approximation errors by quantifying the impact of single bit errors on output quality. QAPPA assumes deterministic value truncation or rounding as opposed to random bit-errors. QAPPA is not so much focused on improving resiliency, and rather aims to expose opportunities to reduce energy and bandwidth in hardware accelerators.

Error Guarantees.

Approximate computing has embraced statistical guarantees [8, 6] to provide common-case error bounds. Our work inspires itself from past work to provide statistical error bounds. Numerical analysis exploits interval analysis [30, 31] to reason about quantization and rounding errors in floating point programs. dco/scorpio [4] is a framework that automates significance analysis to identify computation tasks that have high contribution to output quality. QAPPA could be augmented with such frameworks to provide stricter error bounds.

Precision-Scaling Hardware Techniques.

Quora [11] is a precision scalable SIMD architecture that delivers energy precision trade-offs in parallel applications. Stripes and Proteus [12, 15] propose precision scalable compute and storage mechanisms that can improve the energy efficiency of DNN accelerators. QAPPA can be used as a software compiler for such precision scalable architectures, by automatically deriving precision requirements and providing statistical guarantees. Our comparative evaluation of precision-scaling mechanisms aims to motivate more precision scalable architecture proposals like Quora and Stripes.

7. CONCLUSION.

We present QAPPA, a framework that fine-tunes quantization requirements of C/C++ programs, while meeting user defined, application level quality guarantees. We analyze the PERFECT benchmark suite with QAPPA and find that much precision can be discarded at reasonable quality targets. We evaluate hardware mechanisms that can reduce compute energy and memory bandwidth in hardware accelerator designs. We then perform a comparative study of quantization as a viable alternative to voltage overscaling and neural approximation. We show that precision reduction rivals these techniques in terms of energy savings, while exhibiting predictable error and providing practical quality guarantees. We hope that our findings will motivate other researchers to propose quantization scalable architectures to bring systems closer to their quality vs. energy pareto optimal design point.

8. REFERENCES

[1] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A programmer-guided compiler framework for practical approximate computing," Tech. Rep. UW-CSE-15-01-01, U. Washington, 2015.

[2] K. Barker, T. Benson, D. Campbell, D. Ediger, R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Marquez, L. Song, N. Tallent, and

A. Tumeo, *PERFECT (Power Efficiency Revolution For Embedded Computing Technologies) Benchmark Suite Manual*. Pacific Northwest National Laboratory and Georgia Tech Research Institute, December 2013. `http://hpc.pnnl.gov/projects/PERFECT/`.

[3] C. Rubio-Gonzalez, 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 *Int. Conf. High Performance Computing, Networking, Storage and Analysis*, 2013.

[4] V. Vassiliadis, J. Riehme, J. Deussen, K. Parasyris, C. D. Antonopoulos, N. Bellas, S. Lalis, and U. Naumann, "Towards automatic significance analysis for approximate computing," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO 2016, 2016.

[5] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2012.

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

[7] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: Queries with bounded errors and bounded response times on very large data," in *ACM European Conf. Computer Systems (EuroSys)*, 2013.

[8] D. Mahajan, A. Yazdanbakhsh, J. Park, B. Thwaites, and H. Esmaeilzadeh, "Towards statistical guarantees in controlling quality tradeoffs for approximate acceleration," in *Proceedings of the 43rd Annual International Symposium on Computer Architecture*, ISCA '16, 2016.

[9] E. S. P. C. J. Clopper, "The use of confidence or fiducial limits illustrated in the case of the binomial," *Biometrika*, vol. 26, no. 4, pp. 404–413, 1934.

[10] 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 *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2011.

[11] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2013.

[12] P. Judd, J. Albericio, and A. Moshovos, "Stripes: Bit-serial deep neural network computing," *IEEE Computer Architecture Letters*, 2016.

[13] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam, "Dadiannao: A machine-learning supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[14] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[15] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, N. E. Jerger, and A. Moshovos, "Proteus: Exploiting numerical precision variability in deep neural networks," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, 2016.

[16] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *Design, Automation and Test in Europe (DATE)*, 2010.

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

[18] R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *Int. Symp. Computer Architecture (ISCA)*, 2014.

[19] L. N. Chakrapani, P. Korkmaz, B. E. S. Akgul, and K. V. Palem, "Probabilistic system-on-a-chip architectures," in *ACM Transactions Design Automation of Electronic Systems*, 2007.

[20] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *Design, Automation and Test in Europe (DATE)*, 2010.

[21] Y. Yetim, M. Martonosi, and S. Malik, "Extracting useful computation from error-prone processors for streaming applications," in *Design, Automation and Test in Europe (DATE)*, 2013.

[22] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge, "Razor: a low-power pipeline based on circuit-level timing speculation," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2003.

[23] A. Yazdanbakhsh, D. Mahajan, P. Lotfi-Kamran, and H. Esmaeilzadeh, "Axbench: A multi-platform benchmark suite for approximate computing," *IEEE Design and Test, special issue on Computing in the Dark Silicon Era*, 2016.

[24] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin, "SNNAP: Approximate computing on programmable SoCs via neural acceleration," in *Int. Symp. High-Performance Computer Architecture (HPCA)*, 2015.

[25] B. Grigorian, N. Farahpour, and G. Reinman, "BRAINIAC: Bringing reliable accuracy into neurally-implemented approximate computing," in *Int. Symp. High-Performance Computer Architecture (HPCA)*, 2015.

[26] J. Benson, R. Cofell, C. Frericks, C. H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the dyser hardware accelerator into opensparc," in *IEEE International Symposium on High-Performance Comp Architecture*, 2012.

[27] T. Nowatzki and K. Sankaralingam, "Analyzing behavior specialized acceleration," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[28] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Rumba: An online quality management system for approximate computing," in *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, 2015.

[29] D. S. Khudia, B. Zamirai, M. Samadi, and S. Mahlke, "Quality control for approximate accelerators by error prediction," *IEEE Design Test*, vol. 33, Feb 2016.

[30] F. de Dinechin, C. Lauter, and G. Melquiond, "Certifying the floating-point implementation of an elementary function using gappa," *IEEE Trans. Comput.*, vol. 60, Feb. 2011.

[31] E. Darulova and V. Kuncak, "Trustworthy numerical computation in scala," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, 2011.

[32] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: Self-tuning approximation for graphics engines," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2013.

[33] J. Bornholt, E. Torlak, D. Grossman, and L. Ceze, "Optimizing synthesis with metasketches.," in *ACM Symp. Principles of Programming Languages (POPL)*, 2016.

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

[35] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2009.

[36] S. Sidiroglou, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *European Software Engineering Conf. and ACM SIGSOFT Symp. the Foundations of Software Engineering (FSE)*, 2011.

[37] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-based approximation for data parallel applications," in *Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[38] B. Belhadj, A. Joubert, Z. Li, R. Héliot, and O. Temam, "Continuous real-world inputs can open up alternative accelerator designs.," in *Int. Symp. Computer Architecture (ISCA)*, 2013.

[39] L. Renganarayana, S. Vijayalakshmi, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *ACM Work. Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

[40] M. Rinard, "Parallel synchronization-free approximate data structure construction," in *USENIX Work. Hot Topics in Parallelism (HotPar)*, 2013.

[41] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with uncertainty," in *ACM Work. Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

[42] B. Grigorian and G. Reinman, "Accelerating divergent applications on simd architectures using neural networks," in *Int. Conf. on Computer Design (ICCD)*, 2014.

[43] A. K. Mishra, R. Barik, and S. Paul, "iACT: A software-hardware framework for understanding the scope of approximate computing," in *Work. Approximate Computing Across the System Stack*, 2014.

[44] J. S. Miguel, M. Badr, and N. Enright Jerger, "Load value approximation," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2014.

[45] B. Thwaites, G. Pekhimenko, A. Yazdanbakhsh, J. Park, G. Mururu, H. Esmaeilzadeh, O. Mutlu, and T. Mowry, "Rollback-free value prediction with approximate loads," in *Int. Conf. Parallel Architectures and Compilation Techniques (PACT)*, 2014.

[46] M. Sutherland, J. San Miguel, and E. Jerger, "Texture cache approximation on GPUs," 2015.

[47] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions Computers*, vol. 54, pp. 922 – 927, July 2005.

[48] T. Y. Yeh, P. Faloutsos, M. Ercegovac, S. J. Patel, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2007.

[49] S. Venkataramani, K. Roy, and A. Raghunathan, "Substitute-and-simplify: A unified design paradigm for approximate and quality configurable circuits," in *Design, Automation and Test in Europe (DATE)*, 2014.

[50] P. Kulkarni, P. Gupta, and M. Ercegovac, "Trading accuracy for power with an underdesigned multiplier architecture," in *Int. Conf. VLSI Design*, 2011.

[51] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Transactions Very Large Scale Integration Systems (VLSI)*, vol. 8, no. 3, 2000.

[52] Y. Yetim, S. Malik, and M. Martonosi, "CommGuard: Mitigating communication errors in error-prone parallel execution," in *Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

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

[54] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2013.

[55] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *Int. Symp. Computer Architecture (ISCA)*, 2002.

[56] I. J. Chang, D. Mohapatra, and K. Roy, "A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications," *IEEE Trans. Circuits and Systems for Video Technology*, vol. 21, no. 2, pp. 101–112, 2011.

[57] J. San Miguel, J. Albericio, A. Moshovos, and N. Enright Jerger, "Doppelganger: A cache for approximate computing.," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2015.

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

[59] E. Ozre, A. P. Nisbet, and D. Gregg, "A stochastic bitwidth estimation technique for compact and low-power custom processors," in *ACM Transactions Embedded Computing Systems (TECS)*, 2008.

[60] T. M. Aamodt and P. Chow, "Compile-time and instruction-set methods for improving floating- to fixed-point conversion accuracy," *ACM Transactions Embedded Computing Systems (TECS)*, vol. 7, no. 3, 2008.

[61] R. Venkatagiri, A. Mahmoud, S. Kumar Sastry Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2016.