browser-based multilingual translation

bergamot

Horizon 2020 Research and Innovation Action
Grant Agreement No. 825303

https://browser.mt

Deliverable 5.1:
Translation software with initial CPU optimizations

Lead author(s):	Kenneth Heafield (UEDIN)
Contributing author(s):	Roman Grundkiewicz (UEDIN), Nikolay Bogoychev (UEDIN), Alham Fikri Aji (UEDIN)
Internal Reviewer(s):	Ulrich Germann (UEDIN)

Work Package:	5
Type of Deliverable:	Other
Due Date:	31 August 2019
Date of Submission:	31 August 2019
Current Version:	1.0

Document History

Version	Date	Changes
1.0	31 August 2019	Original Submission

Executive Summary

The Bergamot project will deploy machine translation client-side on desktop computers as an extension to Firefox. Translation speed and quality are crucial components of the overall user experience and success of the project. The primary component of this Deliverable 5.1 are contributions to the code base of the open-source Marian machine translation system (available at https://github.com/marian-nmt/marian-dev), and a CPU matrix library, *intgemm* (available at https://github.com/kpu/intgemm/), both of which include initial optimizations for matrix computations on CPUs. The Workshop on Neural Generation and Translation conducted an efficiency shared task under identical conditions in 2018 and 2019. Marian submissions dominated the 2018 shared task. Marian's 2019 CPU submissions are 5–8 times as fast as the 2018 submissions with higher quality. On a single core, CPU translation speed ranges from 15 sentences per second at 28.52 BLEU to 153 sentences per second at 26.27 BLEU. Model sizes are as small as 19 MB.

Contents

1 Introduction

The Bergamot project aims to offer high-quality machine translation without compromising user privacy. By running machine translation locally, rather than in the cloud, users retain control over the text they translate. This is particularly useful for government and corporate intranets, where confidentiality requirements often effectively prohibit the use of cloud translation services. Moving from a cloud server to an end-user desktop entails supporting a wide range of CPUs with relatively small amounts of RAM and often no computationally useful GPU. Translation should nonetheless be usably fast and competitive with top systems in terms of quality. This deliverable is concerned with optimizing the translation software for CPUs.

The translation software is Marian https://marian-nmt.github.io/, an efficient machine translation system written in C++. Code is hosted at https://github.com/marian-nmt/marian-dev which includes the `intgemm` library https://github.com/kpu/intgemm for matrix multiplication and low-level functions on CPUs. Many of the optimizations are now included in the `master` branch while very recent changes appear in the `4bitdecode` and `intgemm-ssru` branches for 4-bit quantization and recurrent units computed in integers, respectively.

Machine translation systems can be optimized at several levels. Implementing the Simple Recurrent Unit (Lei et al., 2018) substantially increased speed and parallelism by removing unnecessary complexity in the model structure. Vectorizing operations improves performance on CPUs by making better use of Single-Instruction Multiple-Data (SIMD) instructions. Transcendental functions can be approximated to make them easier to vectorize and to improve speed. Integer operations generally outperform floating-point operations, so there is an ongoing effort to make Marian run completely in integers; this release increases usage of integers. The integers can be quantized to 8 bits (Junczys-Dowmunt et al., 2018) with no loss in quality (Quinn and Ballesteros, 2018). We are going further by compressing models to 4 bits, which reduces the download size of a model to 19 MB. Support for loading compressed models is publicly available in the `4bitdecode` branch but uses larger values at runtime; native 4-bit operation is being investigated.

2 Evaluation and Shared Task

It is relatively easy to make a machine translation system faster simply by configuring a smaller model, but this comes at the expense of quality. There is no single best system, but a Pareto frontier of optimal systems in the trade-off between speed and quality. The existence of a frontier also raises the standard of evidence for claims to have optimized a system: there should be a Pareto-comparable baseline (i.e. one slower and with lower quality) and that baseline should have previously been on the frontier. However, papers have been published without Pareto-comparable baselines (Gu et al., 2018).

The Workshop on Neural Generation and Translation (previously the Workshop on Neural Machine Translation) runs a shared task on efficiency[1] and constructs a Pareto frontier. The shared task settings were identical in 2018 and 2019, with the same training data, test sets, and hardware. The hardware for the CPU track is an `m5.large` AWS instance which has one hardware core of a Xeon Platinum 8175M CPU while the GPU track used one `p3.2xlarge` instance with a V100 GPU.

As selected by the shared task organizers, the training data and test sets are restricted to the WMT 2014 English–German task (Bojar et al., 2014) with peculiar preprocessing. Due to the smaller data sets allowed in WMT 2014 (ParaCrawl did not exist at the time), BLEU scores are below the state of the art, but we do not believe this impacts the internal comparison of systems and many of the optimizations are deployed widely with Marian.

Figure 1 shows the Pareto frontier of Marian's submissions to the efficiency shared task in 2018 and 2019. The Figure also shows others' submissions from 2018; results of the 2019 evaluation were not available at press time. The CPU results are also shown in Table 1.

	Time (s)	**BLEU**
2019 Marian	178.0	28.5
	52.9	28.0
	23.9	27.0
	17.8	26.3
2018 Marian	1537.8	28.1
	273.2	27.5
	94.1	26.0
2018 Competitors	1168.6	27.4
	470.7	25.8
	76.8	23.1

Table 1: Time to translate the 2737 sentences in newstest2014 and quality on this set approximated by BLEU. Time was measured on one core of an Intel Xeon Platinum 8175M CPU.

We can pair each 2018 Marian submission with a 2019 submission of slightly higher quality to obtain Pareto-comparable points. For example, the 2018 submission that took 94.1 seconds to achieve 26.0 BLEU can be compared with the 2019 submission that took 17.9 seconds to achieve 26.3 BLEU. Hence Marian sped up at least 5.28x in this case. Applying the same logic to each submission yields a range of 5.16–8.63x as summarized in the abstract.

Optimizing Marian is a collaborative project with contributions from Microsoft and Intel (which wrote letters of support for Bergamot), Samsung,

[1] https://sites.google.com/view/wngt19/efficiency-task

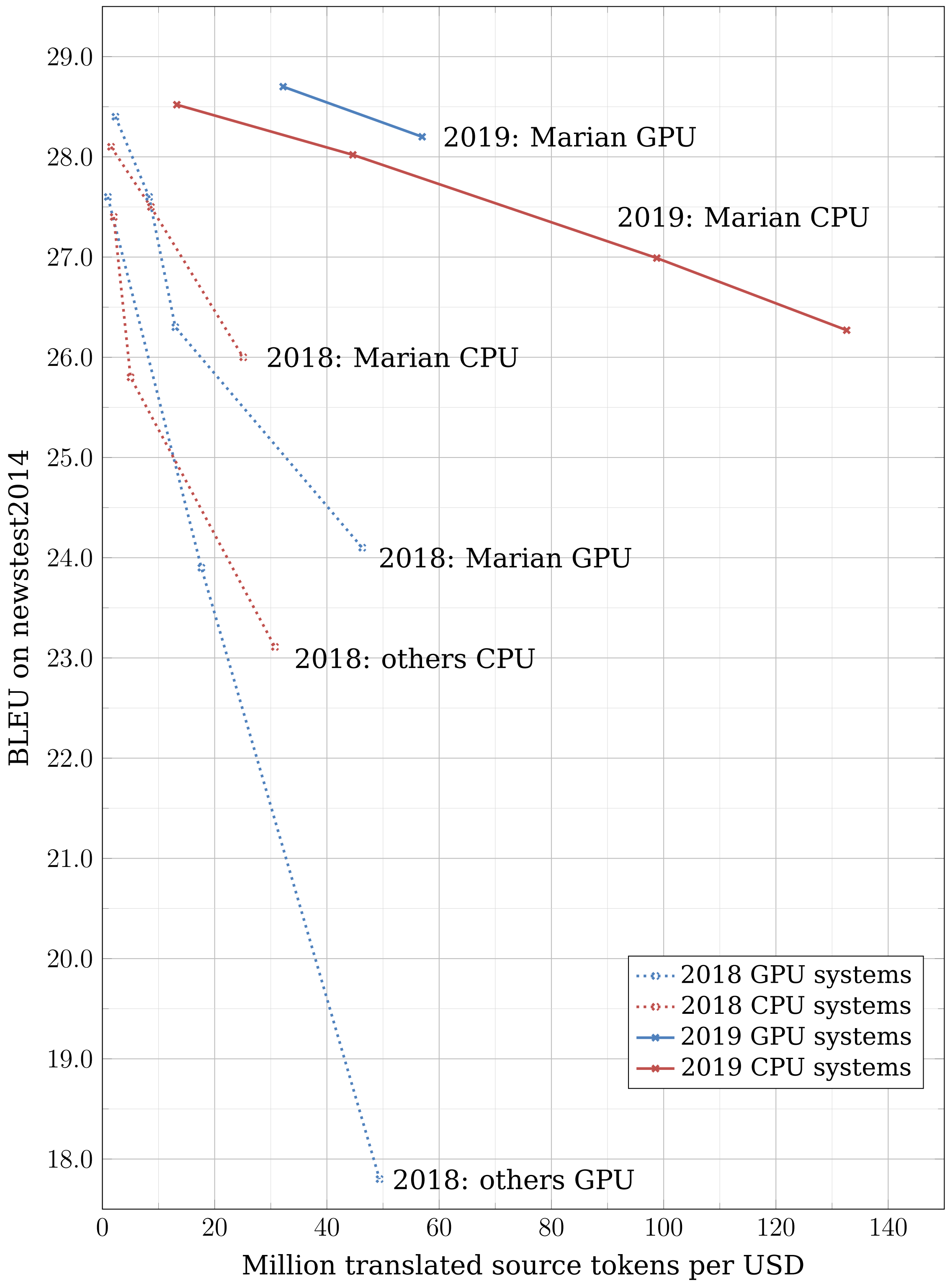

Figure 1: Performance on the 2019 Workshop on Neural Generation and Translation shared task for the `newstest2014` test set. The x-axis costs are based on prices charged by Amazon: $3.06/hr for a V100 GPU and $0.096/hr for one Skylake Xeon CPU core.

and other open-source contributors. The following sections highlight efforts undertaken at the University of Edinburgh.

3 4-bit Quantization

3.1 Introduction

Model quantization (MQ) is the quantization of a model's parameters so that they can be stored in fewer bits than the 32 or even 64 bits required per parameter when they are stored as single or double-precision floating point numbers. Research in MQ aims to develop schemes that reduce storage requirements for models and allow for faster inference during deployment of the model while minimizing the loss in model accuracy.

In order to understand MQ, it helps to recapitulate how non-integer numbers are stored or approximated in floating point representation. A floating point number has two fixed-point components: a signed string of digits of a fixed length representing a number with a fixed precision (*mantissa* or *significand*), and a scale parameter of a fixed length (*exponent*) expressing the magnitude of the number. The value of the number is then calculated as *mantissa*$\times 2^{exponent}$. A 32-bit IEEE 754-2008 binary floating point number, for example, reserves 24 bit for the mantissa (including 1 sign bit), and 8 bits for the exponent (representing magnitudes from 2^{-126} to 2^{127}), leading to a range of representable numbers from $\pm 1.18 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$. In quantization schemes that also aim to facilitate computation,[2] typically only one of the two aforementioned components is stored for each number. In linear quantization it is the mantissa, in logarithmic quantization the exponent (plus one bit for the number's sign). The respective other component is shared among all numbers in the respective quantization scheme. So for successful MQ, we thus need to find a scale (for linear quantization) or factor (for logarithmic quantization) that can be shared across all numbers represented in the respective quantization scheme. In the case of neural models, which involve many weight matrices, it makes sense to establish these numbers on a per-matrix basis.

3.2 Related Work

A considerable amount of research on MQ has been done in the area of computer vision with convolutional neural networks; research on MQ in the area of neural machine translation is much more limited. In this section,

[2] An alternative is the use of code books, where the index into an arbitrary, fixed-length list of numbers is stored. For example, Han et al. (2015) use **k-means clustering** to cluster the model weights for each layer in a neural model and store the cluster index instead of the number, with all numbers in a cluster sharing the same value. This reduces storage requirements but does not facilitate computation.

we will therefore also refer to work on neural models for image processing where appropriate.

Lin et al. (2016), Hubara et al. (2016), Hubara et al. (2017), Jacob et al. (2018), and Junczys-Dowmunt et al. (2018) all use linear quantization. Lin et al. (2016) and Hubara et al. (2016) use a fixed scale parameter prior to model training; Junczys-Dowmunt et al. (2018) and Jacob et al. (2018) base it on the maximum tensor values for each matrix observed in the trained models.

Observing that their parameters are highly concentrated near 0 (see also Lin et al., 2016; See et al., 2016), Miyashita et al. (2016) opt for logarithmic quantization. They report an improvement in preserving model accuracy over linear quantization while achieving the same model compression rate.

Hubara et al. (2017) compress an LSTM-based architecture for language modeling to 4-bit without any quality degradation (while increase the unit size by a factor of 3). See et al. (2016) pruned an NMT model by removing any weight values that are lower than a certain threshold. They achieve 80% model sparsity without any quality degradation.

The most relevant work with respect to our purposes is the submission of Junczys-Dowmunt et al. (2018) to the Shared Task on Efficient Neural Machine Translation in 2018 (cf. Sec. 2). This submission applied an 8-bit linear quantization for NMT models without any noticeable deterioration in translation quality. Similarly, Quinn and Ballesteros (2018) proposed to use an 8-bit matrix multiplication to speedup an NMT system.

3.3 Beyond the State of the Art

This deliverable provides the following improvements over the state of the art.[3]

- We present a new logarithmic quantization formula that reduces the loss in precision due to logarithmic quantization.
- We also develop a new method of empirically determining an optimal scaling parameter for logarithmic quantization that minimizes the loss in precision with respect to the original matrix.
- We show that we can reduce the loss in translation quality due to quantization by further training the quantized model.

3.4 Improved Model Quantization

Lin et al. (2016) and See et al. (2016) report that parameters in deep learning models are normally distributed, and most of them are small values.

[3] Work performed by Alham Fikri Aji under the supervision of Kenneth Heafield.

Therefore, we adopt a logaritmic quantization, where each center is defined as 2^p, similar to Miyashita et al. (2016). This allows for more centers for smaller values, giving us more precision of representation where parameter value density is the highest.

When compressing the model to B-bit, a single bit will be used for the sign, hence we are left with $B-1$ for representing the values. p is an integer defined as $0 \geq p > (\frac{B}{2})$. We use a symmetric quantization; we apply the compression in absolute function, then put back the sign after compression. Therefore, our quantization centers (in absolute value) will be $1, 0.5, 0.25, ...$ up to $2^{\frac{B}{2}-1}$.

However, we find that models might not have the same magnitude as the quantization centers. To solve this issue, we also scale the the model values temporarily before quantizing, then re-scale it back to the original magnitude. This approach is different to that of Miyashita et al. (2016), where quantization centers are not-scaled, thus letting every layers to have the same centers.

Miyashita et al. (2016) quantize a value by rounding the logarithmic value to the closest integer. However, we found that this does not always quantize a value to the closest 2^k. For example, this approach will quantize 5.8 to 2^3 instead of 2^2, because $round(log_2(5.7))$ is 3. Instead, we always round up the logarithmic value after we divide it by 1.5, as shown in Eq.1 below.

$$
\begin{aligned}
\vec{v} &= |\vec{v}|/S \\
\vec{v} &= clip(\vec{v}, [1, 2^{\frac{B}{2}-1}]) \\
\vec{q} &= ceil(\log_2(\frac{2}{3}\vec{v})) \\
\vec{a} &= sign(\vec{v}) * 2^{\vec{q}} \\
\vec{v'} &= \vec{a} * S
\end{aligned}
\tag{1}
$$

$$
S = max(|\vec{V}|) \tag{2}
$$

Junczys-Dowmunt et al. (2018) and Jacob et al. (2018) scale the model based on its maximum value (Equation 2), which might be very unstable. Alternatively, Lin et al. (2016) and Hubara et al. (2016) use a pre-defined step-size in their fixed point quantization. Our objective is to select a scaling factor S such that the quantized parameter is as close to the original as possible. Therefore, we optimize S such that it minimizes the squared error between the original and the compressed parameter.

We propose a method to fit S with Expectation-Maximization. We first start with an initial scale S based on parameters' maximum value. For given S, we apply our quantization routine described in Equation 1, resulting in a center assignment $\vec{a}$. For a given assignment $\vec{a}$, we fit a new scale S' such that:

$$S = \arg\min_S \sum_i (v'_i - v_i)^2 \tag{3}$$

Substituting v' within Eq. 3 with the result of the last line in Eq. 1, we have:

$$S = \arg\min_S \sum_i (a_i * S - v_i)^2 \tag{4}$$

To optimize the given objective, we take the first derivative of Equation 4 such that:

$$\begin{aligned}
\frac{d}{dS}\sum_i (a_i * S - v_i)^2 &= 0 \\
2\sum_i (a_i * (a_i * S - v_i)) &= 0 \\
\sum_i (a_i^2 * S) - \sum_i (a_i * v_i) &= 0 \\
S\sum_i a_i^2 &= \sum_i (a_i * v_i) \\
S &= \frac{\sum_i (a_i * v_i)}{\sum_i a_i^2}
\end{aligned} \tag{5}$$

We optimize S for each tensor independently.

3.5 Retraining

Unlike Junczys-Dowmunt et al. (2018), we retrain the model after initial quantization to allow it to recover some of the quality loss. In the retraining phase, we compute the gradients normally with full precision. We then re-quantize the model after every update to the parameters, including fitting scaling factors. The re-quantization error is preserved in a residual valriable and added to the next step's parameter (Seide et al., 2014). This error feedback mechanism was introduced in gradient compression techniques to reduce the impact of compression errors by preserving compression errors as stale gradient updates for the next batch (Aji and Heafield, 2017; Lin et al., 2017). We see reapplying quantization after parameter updates as a form of gradient compression, hence we explore the usage of an error feedback mechanism to potentially improve the final model's quality.

3.6 Handling Biases

We do not quantize bias values in the model. We found that they do not follow the same distribution as other parameters, and attempting to log-quantize them used only a fraction of the available quantization points. In any case, bias values do not take up a lot of memory relative to other parameters. In our Transformer architecture, they account for only 0.2% of the parameter values.

3.7 Low-Precision Matrix Multiplication

Our end goal is to run a log-quantized model without decompressing it. Activations coming into a matrix multiplication are quantized on the fly; intermediate activations are not quantized.

We use the same log-based quantization procedure described in Section 3.4. However, we only attempt a max-based scale. Running the slower EM approach to optimize the scale before every dot product would not be fast enough for inference applications.

The derivatives of ceiling and sign functions are zero almost everywhere and undefined in some places. For retraining purposes, we apply a straight through estimator (Bengio et al., 2013) to the ceiling function. For the sign function, we treat the quantization function differently for each individual value in $\vec{v}$, based on their sign. Therefore, we now compute a_i as:

$$a_i = \begin{cases} 2^q_i, & \text{if } v_i > 0 \\ -1 * 2^q_i, & \text{otherwise} \end{cases} \tag{6}$$

Since we multiply by a constant, the derivative will be either multiplied by 1 or -1. In latter case, the derivative of $|\vec{v}|$ will be -1, which returns the derivative's sign back to positive. Hence, the derivative of our quantization function is:

$$\frac{d\vec{v}}{d\vec{v'}} = \begin{cases} 1, & \text{if} 1 \geq \frac{|\vec{v}|}{S} \geq 2^{\frac{B}{2}-1} \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

3.8 Experiments

3.8.1 Experiment Setup

We use systems for the WMT 2017 English to German news translation task for our experiment; these differ from the WNGT shared task setting previously reported. We use back-translated monolingual corpora (Sennrich et al., 2016a) and byte-pair encoding (Sennrich et al., 2016b) to preprocess

Method	**Scaling**					
	Fixed	**Max**	**Optim**	**Fixed**	**Max**	**Optim**
Baseline	35.66					
	Without retraining			**With retraining**		
+ Model Quantization	25.2	28.08	33.33	34.92	34.81	35.26
+ No Bias Quantization	34.16	34.29	34.31	35.09	35.25	35.47

Table 2: 4-bit Transformer quantization performance for English to German translation, measured in BLEU score. We explore different method to find the scaling factor, as well as skipping bias quantization and retraining.

the corpus. Quality is measured with BLEU (Papineni et al., 2002) score using sacreBLEU script (Post, 2018).

We first pre-train baseline models with both Transformer and RNN architecture. Our Transformer model consists of six encoder and six decoder layers with tied embedding. Our deep RNN model consists of eight layers of bidirectional LSTM. Models were trained synchronously with a dynamic batch size of 40 GB per-batch using the Marian toolkit (Junczys-Dowmunt et al., 2018). The models are trained for 8 epochs. Models are optimized with Adam (Kingma and Ba, 2014). The rest of the hyperparameter settings on both models are following the suggested configurations (Vaswani et al., 2017; Sennrich et al., 2017).

3.8.2 4-bit Transformer Model

In this first experiment we explore different ways to scale the quantization centers, the significance of quantizing biases, and the significance of retraining. We use pretrained Transformer model as our baseline, and apply our quantization algorithm on top of that.

Table 2 summarizes the results. Using a simple, albeit unstable max-based scaling has shown to perform better compared to the fixed quantization scale. However, fitting the scaling factor to minimize the quantization squared-error produces the best quality. Interestingly, the BLEU score differences between quantization centers are diminished after retraining.

We can also see improvements by not quantizing biases, especially without retraining. Without any retraining involved, we reached the highest BLEU score of 35.47 by using optimized scale, on top of uncompressed biases. Without biases quantization, we obtained 7.9x compression ratio (instead of 8x) with a 4-bit quantization. Based on this trade-off, we argue that it is more beneficial to keep the biases in full-precision.

Retraining has shown to improve the quality in general. After retraining, the quality differences between various scaling and biases quantization configurations are minimal. These results suggest that retraining helps the model to fine-tune under a new quantized parameter space.

To show the improvement of our method, we compare several com-

Method	**Transformer**	**RNN**
Baseline	35.66	34.28
Reduced Dimension	29.03 (-6.63)	30.88 (-3.40)
Fixed-Point Quantization	34.61 (-1.05)	34.05 (-0.23)
Ours	35.47 (-0.19)	34.22 (-0.06)

Table 3: Model performance (in BLEU) of various quantization approaches, on both Transformer and RNN architecture.

pression approaches to our 4-bit quantization method with retraining and without bias quantization. One of arguably naive way to reduce model size is use smaller unit size. For Transformer, we set the Transformer feed-forward dimension to 512 (from 2048), and the embedding size to 128 (from 512). For RNN, we set the RNN dimension to 320 (from 1024) and the embedding size to 160 (from 512). This way, the model size for both architecture is relatively equal to the 4-bit compressed models.

We also introduce the fixed-point quantization approach as comparison, based on Junczys-Dowmunt et al. (2018). We made few modifications; Firstly We apply retraining, which is absent in their implementation. We also skip biases quantization. Finally, we optimize the scaling factor, instead of the suggested max-based scale.

Table 3 summarizes the result. It shown that reducing the model size by simply reducing the dimension performed worst. Logarithmic based quantization has shown to perform better compared to fixed-point quantization on both architecture.

RNN model seems to be more robust towards the compression. RNN models have lesser quality degradation in all compression scenarios. Our hypothesis is that the gradients computed with a highly compressed model is very noisy, thus resulting in noisy parameter updates. Our finding is in line with prior research which state that Transformer is more sensitive towards noisy training condition (Chen et al., 2018; Aji and Heafield, 2019).

3.9 Quantized Dot-Product

We now apply logarithmic quantization for all dot-product inputs. We use the same quantization procedure as the parameter, however we do not fit the scaling factor as it is very inefficient. Hence, we try using max-scale and fixed-scale. For the parameter quantization, We use optimized scale with uncompressed biases, based on the previous experiment.

Table 4 shows the quality result of the experiment. Generally we see a quality degradation compared to a full-precision dot-product. There is no significant difference between using max-scale or fixed-scale. Therefore, using fixed-scale might be beneficial, as we avoid extra computation cost to determine the scale for every dot-product operations.

Experimental CPU implementations of a 4-bit quantized dot product

Method	**Transformer**	**RNN**
Baseline	35.66	34.28
+ Model Quantization	35.47 (-0.19)	34.22 (-0.06)
+ Dot Product Quantization	35.05 (-0.61)	()

Table 4: Model performance (in BLEU) of model quantization with dot-product quantization, on both Transformer and RNN architecture.

Bit	**Transformer**		**RNN**	
	Size (rate)	**BLEU(Δ)**	**Size (rate)**	**BLEU(Δ)**
32	251 MB	35.66	361 MB	34.28
4	32 MB (7.88x)	35.47 (-0.19)	46 MB (7.90x)	34.22 (-0.06)
3	24 MB (10.45x)	34.95 (-0.71)	34 MB (10.49x)	34.11 (-0.17)
2	16 MB (15.50x)	33.40 (-2.26)	23 MB (15.59x)	32.78 (-1.50)
1	8 MB (30.00x)	29.43 (-6.23)	12 MB (30.35x)	31.71 (-2.51)

Table 5: Compression rate and performance of both Transformer and RNN with various bit-widths. The compression rate between Transformer and RNN is not equal, as they have different bias to parameter size ratio.

are in `intgemm` at https://github.com/kpu/intgemm/blob/working/log4/log4.h. The faster implementation currently takes 8.7 clocks per dot product over 128 4-bit values, compared to 4.5 clocks per dot product over 64 8-bit fixed-point values. This is a slight speed gain despite the fact that there is no native support for 4-bit values (and there is native support for 8-bit values).

3.9.1 Beyond 4-bit precision

With 4-bit quantization and uncompressed biases, we obtain 7.9x compression rate. Bit-width can be set below 4 bit to obtain an even better compression rate, albeit introducing more compression error. To explore this, we sweep several bit-width. We skip bias quantization and optimize the scaling factor.

Training an NMT system below 4-bit precision is still a challenge. As shown in Table 5, model performance degrades with less bit used. While this result might still be acceptable, we argue that the result can be improved. One idea that might be interesting to try is to increase the unit-size in extreme low-precision setting. We shown that 4-bit precision performs better compared to full-precision model with (near) 8x compression rate. In addition, Han et al. (2015) has shown that 2-bit precision image classification can be achieved by scaling the parameter size. Alternative approach is to have different bit-width for each layers (Hwang and Sung, 2014; Anwar et al., 2015).

We can also see RNN robustness over Transformer in this experiment, as RNN models degrade less compared to the Transformer counterpart. RNN model outperforms Transformer when compressing at binary precision.

3.10 Conclusion

We compress the model size in neural machine translation to approximately 7.9x smaller than 32-bit floats by using a 4-bit logarithmic quantization. Bias terms behave different and can be left uncompressed without affecting the compression rate significantly. We also find that retraining after quantization is necessary to restore the model's performance.

4 CPU support

CPUs can process multiple values with one instruction. Exploiting this functionality is crucial to performance. However, due to the introduction of new instructions and wider widths over time, there are over 13 different versions of these instructions supported by various processors. Particularly important to Bergamot are 16-bit and 8-bit integer multiplication instructions, so the project supports different code paths for the SSE2, SSSE3, AVX2, and AVX512 instruction sets. SSE2 was introduced with the Pentium 4 in 2000, so Bergamot supports a wide range of processors. For AVX512, the instruction set is further subdivided so our code assumes support for AVX512F, AVX512BW, and AVX512DQ, AVX512VL instructions that exist on all AVX512 processors except the Xeon Phi processors that are rare on desktops. AVX512VNNI optimization for even newer CPUs is slated for addition once we can acquire access to a machine.

Because the Bergamot software will run on end-user desktops without recompiling code, our goal is that code will detect the CPU it is running on and choose the fastest code path at runtime. This implies the code must be compiled for multiple architecture versions and assembled into one binary. While this is meant to be supported, we found bugs in both GCC[4] and clang.[5] Our code in `intgemm` works around these unfortunate bugs by creating separate functions for each architecture with largely identical code using macros or multiple-inclusion.

5 Integers Beyond Matrix Multiplication

Marian was not vectorizing many transcendental functions due to the difficulty of computing them with vector instructions. However, neural networks are tolerant to approximation. For example, we approximated the sigmoid function $\sigma(x)$ by quantizing x to an integer then indexing a lookup

[4] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=89929
[5] https://bugs.llvm.org/show_bug.cgi?id=41482

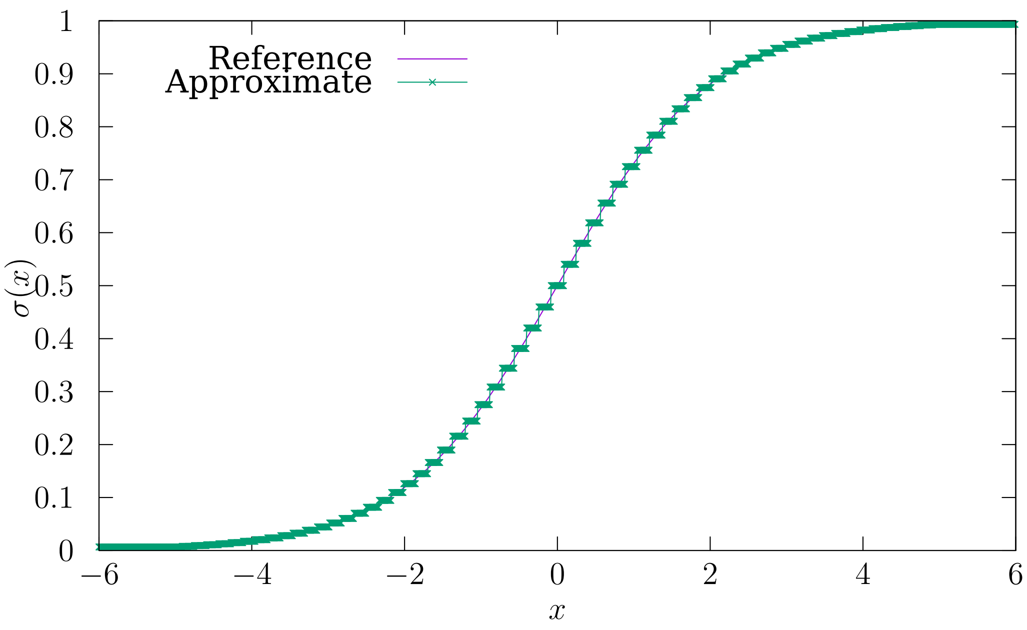

Figure 2: Sigmoid function approximation with 5-bit integer lookup table. There are 64 quantization points with a 5-bit table by exploiting the symmetry property of sigmoid.

table. Due to the symmetry $\sigma(-x) = 1 - \sigma(x)$, a 32-entry lookup table could service 64 quantization points. The `vpermi2d` instruction implements a 32-entry lookup table in one instruction for 16 lookups in parallel. Quantizing sigmoid alone, along with integrating it into the highway function to avoid writing temporaries to memory, increased speed by 3%. The impact on quality was negligible: 0.02% BLEU gain on newstest2014. Work is underway for the softmax function, which is more complicated to vectorize but consumes far more time.

These functions that take in floating-point arguments are a steppingstone to a full-integer implementation. The 32-entry lookup table suggests that 5- to 6-bit integers are sufficient to represent inputs and outputs of transcendental functions in neural networks, even without retraining. We hypothesize that 4 bits will be sufficient with retraining. In the `intgemm-ssru` branch, the entire simplified recurrent unit is being rewritten in integers.

6 Reducing Memory Traffic

Marian typically performs matrix multiplication, writes the results to memory, reads from memory, adds a bias term, writes to memory, reads from memory, applies a non-linear function, writes to memory, and uses the values somewhere else. These memory accesses are often the most expensive operation, especially for desktops that typically have lower memory bandwidth than servers. By fusing functions, we aim to reduce memory traffic and therefore increase speed. The `intgemm` library supports fused multiplication and bias addition along with improved use of the 8-bit multiply instruction to avoid sign bit manipulation in critical loops. This results in a ≈10% improvement in overall translation speed relative to existing 8-bit multiplication. Code is included in the master code of `intgemm`.

In our larger project to replace the recurrent unit with pure integer operations, we are fusing the recurrent unit together to the extent possible. Intermediate values stay in registers unless they are needed for a matrix multiplication (where they will need to be accessed at different times). This code appears in the `intgemm-ssru` branch.

7 Summary

Marian is 5–8 times as fast on CPUs as it was in the 2018 evaluation. Quantizing models to 4 bits yields substantially smaller models for users to download. As this deliverable is "initial" CPU optimization, work is continuing in several directions: native 4-bit arithmetic, optimizing for bleeding-edge CPUs, pervasive integers, and reduced memory traffic.

References

Aji, Alham Fikri and Kenneth Heafield. 2017. "Sparse communication for distributed gradient descent." *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, 440–445.

Aji, Alham Fikri and Kenneth Heafield. 2019. "Making asynchronous stochastic gradient descent work for transformers." *arXiv preprint arXiv:1906.03496*.

Anwar, Sajid, Kyuyeon Hwang, and Wonyong Sung. 2015. "Fixed point optimization of deep convolutional neural networks for object recognition." *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 1131–1135.

Bengio, Yoshua, Nicholas Léonard, and Aaron C. Courville. 2013. "Estimating or propagating gradients through stochastic neurons for conditional computation." *CoRR*, abs/1308.3432.

Bojar, Ondrej, Christian Buck, Christian Federmann, Barry Haddow, Philipp Koehn, Johannes Leveling, Christof Monz, Pavel Pecina, Matt Post, Herve Saint-Amand, Radu Soricut, Lucia Specia, and Aleš Tamchyna. 2014. "Findings of the 2014 workshop on statistical machine translation." *Proceedings of the Ninth Workshop on Statistical Machine Translation*, 12–58. Baltimore, Maryland, USA.

Chen, Mia Xu, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Niki Parmar, Mike Schuster, Zhifeng Chen, et al. 2018. "The best of both worlds: Combining recent advances in neural machine translation." *arXiv preprint arXiv:1804.09849*.

Gu, Jiatao, James Bradbury, Caiming Xiong, Victor O.K. Li, and Richard Socher. 2018. "Non-autoregressive neural machine translation." *International Conference on Learning Representations*.

Han, Song, Huizi Mao, and William J Dally. 2015. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." *arXiv preprint arXiv:1510.00149*.

Hubara, Itay, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. "Binarized neural networks." *Advances in neural information processing systems*, 4107–4115.

Hubara, Itay, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. "Quantized neural networks: Training neural networks with low precision weights and activations." *The Journal of Machine Learning Research*, 18(1):6869–6898.

Hwang, Kyuyeon and Wonyong Sung. 2014. "Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1." *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 1–6.

Jacob, Benoit, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. "Quantization and training of neural networks for efficient integer-arithmetic-only inference." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2704–2713.

Junczys-Dowmunt, Marcin, Kenneth Heafield, Hieu Hoang, Roman Grundkiewicz, and Anthony Aue. 2018. "Marian: Cost-effective high-quality neural machine translation in C++." *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation*, 129–135. Melbourne, Australia.

Kingma, Diederik P and Jimmy Ba. 2014. "Adam: A method for stochastic optimization." *arXiv preprint arXiv:1412.6980*.

Lei, Tao, Yu Zhang, Sida I. Wang, Hui Dai, and Yoav Artzi. 2018. "Simple recurrent units for highly parallelizable recurrence." *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 4470–4481. Brussels, Belgium.

Lin, Darryl, Sachin Talathi, and Sreekanth Annapureddy. 2016. "Fixed point quantization of deep convolutional networks." *International Conference on Machine Learning*, 2849–2858.

Lin, Yujun, Song Han, Huizi Mao, Yu Wang, and William J Dally. 2017. “Deep gradient compression: Reducing the communication bandwidth for distributed training.” *arXiv preprint arXiv:1712.01887*.

Miyashita, Daisuke, Edward H Lee, and Boris Murmann. 2016. “Convolutional neural networks using logarithmic data representation.” *arXiv preprint arXiv:1603.01025*.

Papineni, Kishore, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. “Bleu: a method for automatic evaluation of machine translation.” *Proceedings of the 40th annual meeting on association for computational linguistics*, 311–318.

Post, Matt. 2018. “A call for clarity in reporting bleu scores.” *Proceedings of the Third Conference on Machine Translation: Research Papers*, 186–191.

Quinn, Jerry and Miguel Ballesteros. 2018. “Pieces of eight: 8-bit neural machine translation.” *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*, 114–120.

See, Abigail, Minh-Thang Luong, and Christopher D Manning. 2016. “Compression of neural machine translation models via pruning.” *arXiv preprint arXiv:1606.09274*.

Seide, Frank, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. “1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns.” *Fifteenth Annual Conference of the International Speech Communication Association*.

Sennrich, Rico, Alexandra Birch, Anna Currey, Ulrich Germann, Barry Haddow, Kenneth Heafield, Antonio Valerio Miceli Barone, and Philip Williams. 2017. “The University of Edinburgh’s neural mt systems for WMT17.” *Proceedings of the Second Conference on Machine Translation*, 389–399.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016a. “Improving neural machine translation models with monolingual data.” 86–96.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016b. “Neural machine translation of rare words with subword units.” *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 1715–1725.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. “Attention is all you need.” *Advances in Neural Information Processing Systems*, 5998–6008.