From Research to Production and Back: Ludicrously Fast Neural Machine Translation

Young Jin Kim[†*] **Marcin Junczys-Dowmunt**[†*]
Hany Hassan[†] **Alham Fikri Aji**[‡] **Kenneth Heafield**[†‡]
Roman Grundkiewicz[†‡] **Nikolay Bogoychev**[‡]

[†]Microsoft, 1 Microsoft Way, Redmond, WA 98121, USA
{youki,marcinjd,hanyh}@microsoft.com

[‡]University of Edinburgh, Edinburgh, Scotland, EU
{a.fikri,kheafiel,rgrundkie,n.bogoych}@ed.ac.uk

Abstract

This paper describes the submissions of the "Marian" team to the WNGT 2019 efficiency shared task. Taking our dominating submissions to the previous edition of the shared task as a starting point, we develop improved teacher-student training via multi-agent dual-learning and noisy backward-forward translation for Transformer-based student models. For efficient CPU-based decoding, we propose pre-packed 8-bit matrix products, improved batched decoding, cache-friendly student architectures with parameter sharing and light-weight RNN-based decoder architectures. GPU-based decoding benefits from the same architecture changes, from pervasive 16-bit inference and concurrent streams. These modifications together with profiler-based C++ code optimization allow us to push the Pareto frontier established during the 2018 edition towards 24x (CPU) and 14x (GPU) faster models at comparable or higher BLEU values. Our fastest CPU model is more than 4x faster than last year's fastest submission at more than 3 points higher BLEU. Our fastest GPU model at 1.5 seconds translation time is slightly faster than last year's fastest RNN-based submissions, but outperforms them by more than 4 BLEU and 10 BLEU points respectively.

1 Introduction

This paper describes the submissions of the "Marian" team to the Workshop on Neural Generation and Translation (WNGT 2019) efficiency shared task (Hayashi et al., 2019). The goal of the task is to build NMT systems on CPUs and GPUs placed on the Pareto Frontier of efficiency and accuracy.

Marian (Junczys-Dowmunt et al., 2018a) is an efficient neural machine translation (NMT) toolkit written in pure C++ based on dynamic computational graphs.[1] Marian is a research tool which can be used to define state-of-the-art systems that at the same time can produce truly deployment-ready models across different devices. This is accomplished within a single execution engine that does not require specialized, inference-only decoders. Our submissions to last year's edition of the same shared task defined the Pareto frontiers for translation quality versus CPU-based and GPU-based decoding speed (Junczys-Dowmunt et al., 2018b).

The title of this paper refers to beneficial co-development of our shared task submissions and our in-productions systems at Microsoft. The improvements from our submission to last year's edition of the shared task (Junczys-Dowmunt et al., 2018b) enabled fast CPU-based decoding with light-weight Transformer models and were a first step towards deploying them in Microsoft's online translation services. Subsequent improvements resulted in a successful launch of Marian as the Microsoft Translator training and inference tool (Microsoft-Translator). Our submissions to this year's edition start out with the currently deployed student model architectures as they are used for Microsoft online-translation systems and explore better teacher-student training and faster CPU-bound inference for the needs of the shared task. These innovations are finding their way back into our production systems at the time of writing.

We improve all aspects of our submissions from last year. Better teachers trained via multi-agent dual learning provide higher quality training data for student models. Better teacher-student training via noisy backward-forward translation minimizes the gap between teacher and student and allows to strongly reduce student size via parameter sharing and fewer decoder layers. At the same time, we are able to shift the need for smaller architectures to decoding with low-precision inference (8-bit on the CPU, 16-bit on the GPU). Similar to last year, we do not let the BLEU score drop below 26 points.

[*]First authors with equal contribution.
[1]https://github.com/marian-nmt/marian

Proceedings of the 3rd Workshop on Neural Generation and Translation (WNGT 2019), pages 280–288
Hong Kong, China, November 4, 2019.
www.aclweb.org/anthology/D19-56%2d

2 Better teacher-student training

Extending our submission from last year (Junczys-Dowmunt et al., 2018b), we train four forward (en-de) and four inverse (de-en) teacher models according to the Transformer-big configuration (model size 1024, filter size 4096, 6 blocks, file size 813 MiB) from Vaswani et al. (2017). We think of a teacher as the set of all models that have been used to create the artificial training data.

Unless stated differently, our student is a single model that follows the Transformer-base configuration (model size 512, filter size 2048, 6 blocks) with modifications. See Section 3 for details. For all models, we use the same vocabulary of 32,000 subwords, computed with SentencePiece (Kudo and Richardson, 2018). The training data is provided by the shared task organizers and restricted to about 4 Million sentences from the WMT news translation task for English-German. Use of other data is not permitted.

We again implement the interpolated sequence-level knowledge distillation method proposed by Kim and Rush (2016): The teacher ensemble is used to forward-translate the training data and collect 8-best lists for each sentence. Choosing the best translation for each sentence based on sentence-level BLEU compared to the original target, we create a synthetic target data. The student is trained on the original source and this synthetic forward translated target.

Table 1 contains BLEU scores of the teacher ensemble (T) and a student model distilled from this teacher (Student ← T). The gap is 2.4 BLEU.

2.1 Knowledge distillation with noisy backward-forward translation

In our experience, student training benefits from forward-translated data that was not seen during teacher training. Since we do not have access to additional monolingual source data, we generate noisy back-translated sentences (Edunov et al., 2018), one set per inverse teacher model. Noisy sentences are generated by sampling from the output softmax distribution via added Gumbel noise. We then use the forward (en-de) teacher ensemble to translate the sampled English sentences into German and choose the best output from the 8-best list measured against the original target. This increases the training corpus 5-fold. Training on this new data reduces the gap to the teacher to 1.3 BLEU; a single teacher model is only 0.4 BLEU better.

System	BLEU
Teacher (T)	28.9
Single teacher model	28.0
Student without teacher	25.9
Student ← T	26.5
Student ← T with 4×NBFT	27.6
Teacher with MADL (T-MADL)	29.8
Single teacher model	29.2
Student ← T-MADL	26.9
Student ← T-MADL with 4×NBFT	28.3

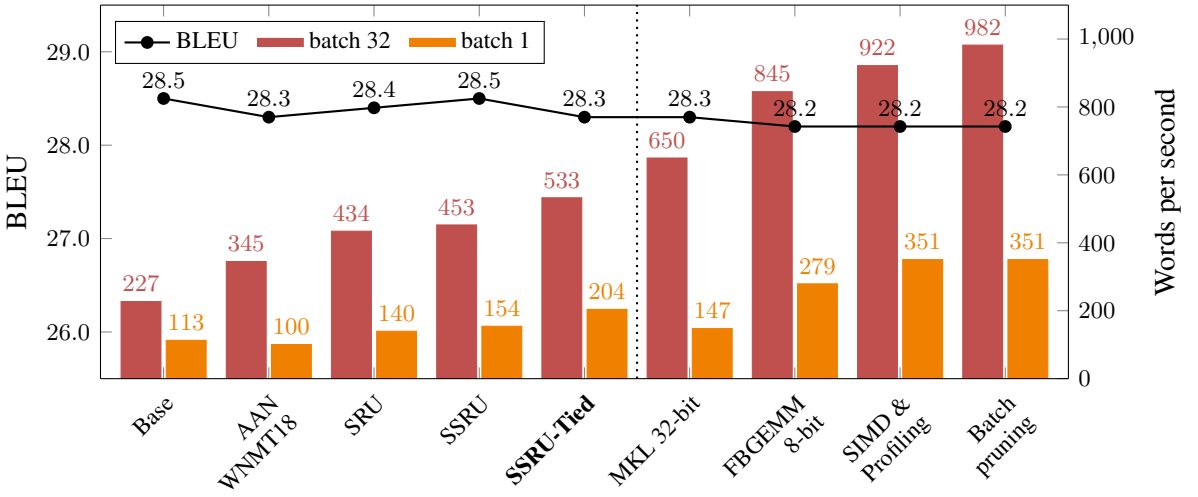
Table 1: Effects of noisy backward-forward translation (NBFT) and Multi-Agent Dual Learning on teacher-student training (newstest2014)

It seems unusual to feed our student with degraded training data, but the goal is to closely mimic the teacher. Since the forward translations are correct outputs of the teacher over noised inputs, the space of probed translations that would not be available to the student otherwise is increased. The role of choosing the best translation from the 8-best list should be investigated in the future.

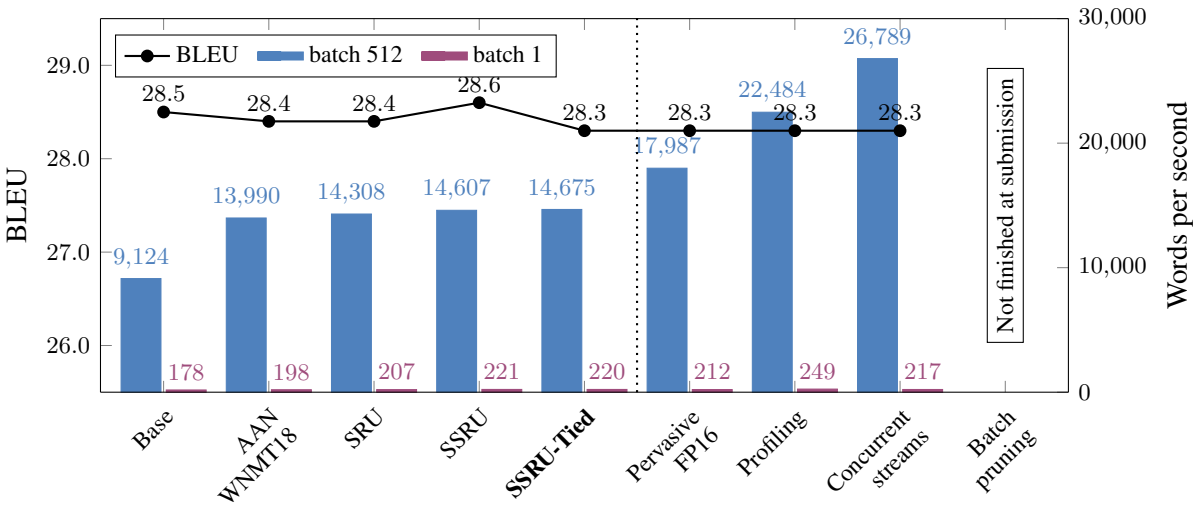
2.2 Multi-Agent Dual Learning

Apart from closing the gap between teacher and student, we can try to improve the teacher and hope the student follows. We adapt Multi-Agent Dual Learning (MADL) by Wang et al. (2019) for this purpose. MADL requires additional monolingual data which we cannot supply. Instead, using the teacher-ensembles for each direction, we generate synthetic German and English corpora from the training data (without noise). We again select the best translations from the generated 8-best lists and join (original English - original German), (original English - synthetic German) and (synthetic English - original German) data sets into new training data for four new teacher models.

Individual teacher models improve by about 1.2 BLEU and an ensemble of four new teachers by 0.9 BLEU (Table 1). We repeat the interpolated knowledge-distillation procedure with the new teacher. The student model (T ← T-MADL) improves only slightly when trained without the noisy input data (+0.4 BLEU), but by a large margin with noisy forward-backward translation. The gap between the new teacher and its student remains at 1.5 BLEU, but a student outperforms a single teacher without MADL (28.3 vs 28.0).



(a) Performance on a single CPU core and thread for newstest2014 on AWS m5.large, dedicated instance



(b) Performance on a NVidia Volta 100 GPU for newstest2014 on AWS p3.x2large

Figure 1: BLEU scores versus words per second with different student architectures and optimizations on CPU and GPU. Results left of the dotted black line are for different architectures with Marian v1.7 as it was released before the shared task. Results right of the dotted black line are recently implemented runtime optimizations applied to "SSRU-Tied". These optimizations will be available with Marian v1.9.

3 Faster student architectures

As mentioned before, our student architecture is a variant of the Transformer-base configuration from Vaswani et al. (2017) with a model size of 512, filter size of 2048 and six blocks of layers in encoder and decoder. Our encoder is always a Transformer encoder with self-attention, our decoder differs in choice of auto-regression mechanisms and parameter tying. In this section, we do not change dimensions or number of blocks. Other dimensions and model depths are discussed in Section 6.

Figure 1 provides an overview about the evolution of student architectures explored for the previous shared task, as Microsoft in-production models and as candidate submissions for the current edition of the shared task. All student variants have been trained with the best teacher-student procedure from the previous section; the example model used there was SSRU-Tied (bold in Figure. 1) which is also the Microsoft Translator in-production model.

We discuss the influence of self-regression mechanisms in Section 3.1 and parameter tying in Section 3.2. Architecture-indepedent but device-specific optimizations for the CPU are detailed in Section 4 and for the GPU in Section 5. More general optimizations are outlined in Section 4.2. Performance has been measured with Marian v1.7, measurements are self-reported by Marian.

3.1 SSRU instead of self-attention or AAN

In previous work (Junczys-Dowmunt et al., 2018b) and later experiments, we found that replacing the self-attention mechanims in Transformer decoders with an Average Attention Network (Zhang et al., 2018) or modern RNN variants does not affect student quality while resulting in faster decoding on GPU and CPU. This is mainly caused by reducing the decoder complexity from $O(n^2)$ to $O(n)$ over the number of output tokens n. In Figure 1 we see how switching from a vanilla Transformer-base variant to a student with AAN improves speed on both devices, but more so on the GPU.

While we had good results for AANs in Junczys-Dowmunt et al. (2018b), we feel somewhat uneasy about the flat element-wise average used to accumulate over inputs. RNNs share the linear computational complexity of the AAN over input size during decoding[2], but can learn a more complex accumulation function. A particularly interesting RNN variant is the SRU (Simple Recurrent Unit) proposed earlier than AANs by Lei et al. (2017)[3]. This RNN variant has no matrix multiplication in its recurrent step and is (at decode-time) surprisingly similar to the AAN if we think of the forget-gate as an adaptive exponential average:

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_t\mathbf{x}_t + \mathbf{b}_f) \\
\mathbf{r}_t &= \sigma(\mathbf{W}_r\mathbf{x}_t + \mathbf{b}_r) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + (1 - \mathbf{f}_t) \odot \mathbf{W}\mathbf{x}_t \\
\mathbf{h}_t &= \mathbf{r}_t \odot \tanh(\mathbf{c}_t) + (1 - \mathbf{r}_t) \odot \mathbf{x}_t
\end{aligned}$$

where the cell-state $\mathbf{c}_{t-1}$ is elementwise-interpolated via forget-gate $\mathbf{f}_t$ with its transformed input $\tilde{\mathbf{x}}_t$ to form the new cell-state $\mathbf{c}_t$. The original formulation adds an output reset-gate $\mathbf{r}_t$ to act as a learnable skip connection for the input $\mathbf{x}_t$ and $\tanh(\mathbf{c}_t)$.

In the Transformer, every block $g(\cdot)$ is followed by an additive skip connection and a layer-normalization operation (Ba et al., 2016):

$$\mathbf{h}_t = \boldsymbol{\alpha} \odot \mathrm{LN}(g(\mathbf{x}_t) + \mathbf{x}_t) + \boldsymbol{\beta}$$

where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are trainable scale and bias vectors.

Given that this construction fulfills a similar role to the reset-gate in the original SRU formulation, we drop the reset-gate $\mathbf{r}_t$ and replace the $\tanh$ non-linearity with a ReLU operation[4] to arrive at our Simpler Simple Recurrent Unit (SSRU):

$$\begin{aligned}
\mathbf{f}_t &= \sigma(\mathbf{W}_t\mathbf{x}_t + \mathbf{b}_f) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + (1 - \mathbf{f}_t) \odot \mathbf{W}\mathbf{x}_t \\
\mathbf{h}_t &= \boldsymbol{\alpha} \odot \mathrm{LN}(\mathrm{ReLU}(\mathbf{c}_t) + \mathbf{x}_t) + \boldsymbol{\beta}
\end{aligned}$$

which replaces the self-attention block in the transformer decoder. This variant saves another matrix multiplication and does not seem to suffer from any performance degradation for student models[5] compared to self-attention, AANs, traditional RNN variants, or the original SRU. In Figure 1a we can see how switching from Base to AAN to SRU and finally to SSRU does not affect BLEU much (within 0.2 per cent) and is actually highest for the SSRU. Speed on CPU increases significantly, both for batch size 32 and 1. On the GPU (Figure 1b) there are small improvements.

3.2 Tied decoder layers

Work on the Universal Transformer (Dehghani et al., 2019) has shown the effectiveness of depth-wise recurrence in Transformer models — i.e. the same parameters are being reused for every corresponding layer across Transformer blocks without quality loss. This is remarkable by itself, but there is also a potential for efficient CPU-bound computations. Although the repeated application of the same parameter set across different layers does not reduce the total number of floating point operations[6] that need to be performed compared to using different parameters per layer, it reduces the parameter size of the decoder layers six-fold and improves CPU cache-locality.

We tie all layers in the decoder (not the encoder) and probably manage to keep the entire set of decoder parameters in L2-cache during translation. Moving from SSRU to SSRU-Tied in Figure 1a, we see a 1.39x speed up and a small drop of 0.2 BLEU. The GPU is largely unaffected since cache-locality is less of an issue here.

[2] AANs parallelize better during training since the average can be computed non-recurrently, however for the small student models the increase in training time is negligible.

[3] We are describing the SRU based on V1 of the preprint on Arxiv from September 2017. Subsequent updates and publications seem to have changed the implementation, resulting in more complex variants, e.g. Lei et al. (2018). We implemented the SRU at time of publication of V1 and missed the updates, but our variant seems to work just fine.

[4] The transformer uses ReLU non-linearities everywhere.

[5] It seems however that student-sized models trained from scratch behave worse when using either SRU or SSRU compared to all the alternatives. There is however no difference between the SRU and SSRU which seems to confirm that the reset-gate $\mathbf{r_t}$ can be dropped when the additive skip-connection is already present.

[6] It does when projections of the encoder into decoder space for purpose of applying cross-attention can be cached.

Optimization	Batch 1	Batch 32
mixed 32/16-bit	1.38 (1.00)	0.82 (1.00)
8-bit FBGEMM	1.89 (1.36)	1.30 (1.58)
SIMD & Profiling	2.39 (1.72)	1.42 (1.73)
Batch pruning	2.39 (1.72)	1.51 (1.84)

Table 2: Relative speed-up for new CPU-bound optimizations compared to float32 MKL baseline and WNMT2018 mixed precision inference (in parentheses) for same SSRU-Tied student model.

4 Optimizing for the CPU

All CPU-bound results in Figure 1a have been computed with a setup from our WNMT2018 submission (Junczys-Dowmunt et al., 2018b). On batch-level, a shortlist selects the 75 most common target words and up to 75 most probable translations per input-batch word. This set of words is used to create an output vocabulary matrix over a couple of hundred words instead of 32,000 which reduces the computational load with no loss in quality (compare with GPU-bound BLEU scores in Figure 1b).

For systems left of the dotted black line, matrix multiplication is executed with mixed 32-bit (Intel's MKL library) and 16-bit (own implementation based on Devlin (2017)) kernels. All systems right of the dotted line, are the same model as "SSRU-Tied" without re-training, but executed with different runtime optimizations. In this section we discuss new runtime optimizations which will be available in Marian v1.9.

4.1 8-bit matrix multiplication with packing

The AWS m5.large target platform for CPU-bound decoding is equipped with an Intel Xeon Platinum 8175 CPU. This CPU supports 8-bit integer instructions with AVX-512 (Advanced Vector eXtensions-512) which can be used to accelerate deep neural network models (Wu et al., 2016; Rodriguez et al., 2018; Bhandare et al., 2019). With the open source FBGEMM library, we integrated 8-bit quantization and matrix multiplication routines with AVX-512 SIMD instructions into our code.

To fully benefit from the faster computation of matrix products in 8-bit, we chose to pre-quantize and pre-pack all parameter matrices offline, except for the embeddings matrix, then save them to a model file. Activations computed during inference are quantized on-the-fly. Matrix products with the short-listed output layer or with non-parameter matrices are executed in 32-bit with MKL.

Quantization. Among the quantization methods offered by FBGEMM, we see the best results when quantizing each column of the weight matrix separately with different scales and offsets per column which have to be provided to the FBGEMM API.

As reported by Lin et al. (2016); Bhandare et al. (2019) and confirmed by our own observations, the distribution of floating point values per column in a weight matrix seems to follow a normal distribution. We compute the average $\bar{x}_j$ and standard deviation σ_j per column j and quantize with saturation into the range $(\bar{x}_j - 7\sigma_j, \bar{x}_j + 7\sigma_j)$. We determined the factor 7 empirically, testing BLEU for values from 1 to 10. This prevents outliers in the weight matrix from distorting the resolution of the quantized values. We seem to lose no more than 0.3 BLEU due to quantization for some models, and only 0.1 BLEU for SSRU-Tied in a base configuration. By comparison, when quantizing to minimum-maximum values in columns, we lose up to 4.0 BLEU for certain models. See the FBGEMM blog (FBGEMM) for more details on quantization.

Packing. We mentioned in Section 3.2 how the repeated application of the same parameters across layers helps L2-cache locality. Packing allows us to also benefit from the CPU's L1-cache and vector registers by changing the layout of the input matrices for a GEMM operation. The FBGEMM API explicitly exposes the packing operation as well as matrix multiplication on pre-quantized and pre-packed matrices.

In Table 2 we see a respectable speed-up against a pure MKL float32 version and our mixed 32/16-bit inference (in parentheses). The speed-up is more impressive in the case of batch-size 1 which is our deployment scenario, but the large batch which we use for the shared task benefits as well.

4.2 Other optimizations

Speed improvements in one part of the code often expose bottlenecks in other places, as these now take up a larger fraction of the time during profiling. We found that element-wise kernels did not vectorize properly and fixed that; we replaced expensive C++ shared-pointers with light-weight non-locking smart-pointers for all small objects and changed the access pattern to model configuration options; we improved our beam-search algorithm to remove finished batch-entries dynamically. The combined speed-up (Table 2) from these optimizations further improves on top of the "fancier" methods.

5 Optimizing for the GPU

We use the same models for our GPU-bound experiments as for CPU decoding. Different than for our experiments on the CPU, we see in Figure 1b that the most influential architecture change is the replacement of decoder self-attention with complexity $O(n^2)$ with any other auto-regressive layer with complexity $O(n)$. There are small speed improvements as we move to layers with smaller amounts of FLOPS, but the parallelization inside the GPU nearly hides these changes. As mentioned before, layer-tying barely affects the speed while there was significant improvement on the CPU. We gain a lot more from the model-independent runtime optimizations described next.

Pervasive FP16 inference. On NVidia Volta 100 GPUs with at least CUDA 9.2 it is straightforward to activate FP16 matrix multiplication with very small modifications to the code. All other operations are executed in 32-bit floating point while inputs to the matrix product are rounded on-the-fly. Outputs are accumulated in 32-bit floats. We used this for our GPU submissions last year.

This year we extended Marian to perform pervasive FP16 inference, i.e. parameters are directly stored in 16-bit floats and all operations stay in 16-bit. The matrix product does not need to convert to 16-bit before or to 32-bit after execution. Improvements in speed stem from operations other than the matrix product and from faster memory access and copying. In Figure 1b, we see a respectable speed improvement for large batches and no loss in BLEU. Interestingly, there is no speed-up for batch-size 1.

Profiling. GPU decoding also benefits strongly from the profiler-based optimizations mentioned in Section 4.2. In a situation where the translation of a full WMT test sets can be performed in one or two second, the time spent during the construction or destruction of millions of book-keeper objects like hypotheses or beams starts to matter a lot. The gain here is larger than for the pervasive FP16 inference. Unfortunately we did not finish the GPU version of the batch-pruning algorithm in time for the shared-task or this description.[7] With the large batches we could expect additional improvements.

[7]The beam search algorithm in the FP16 branch had diverged from the CPU branch and there was no good way to quickly apply the new beam search version to GPU decoding. This will be done in Marian v1.9.

Concurrent streams. We found that the powerful Volta 100 GPU was not fully saturated even when decoding with large batch sizes. Hence, we send multiple batches at once to the same GPU using two CPU threads. The CUDA scheduler assigns different default streams to each CPU thread and we get similar benefits from streams as if these were assigned explicitly. Going beyond two threads does not seem to help. In the case of decoding with batch size 1 it's actually detrimental. We hypothesize that our GPU decoding is not as efficient as it could be and unnecessarily exchanges information between GPU and CPU. This happens in shorter and more frequent intervals for batch-size 1.

6 Submissions and discussion

6.1 Submissions

We submit four CPU students and three GPU systems, summarized in Table 3. We report model configurations, architectures, dimensions, depth, number of parameters, file sizes in MiB for CPU and GPU, translation speed in words per second and BLEU for newstest2014, omitting newstest2015. Time has been measured by the shared-task organizers on AWS m5.large (CPU) and p3.x2large (GPU) instances.

Until this moment, we kept model dimensions and decoder depth constant while optimizing a configuration that corresponds to the Microsoft in-production models (bold row in Table 3). For the final shared task submissions, we vary model dimensions and — similar to Senellart et al. (2018) — decoder depth in order to explore the trade-offs between quality and speed on the Pareto frontier.

We train a shallower base-configuration "(1) Base" with two tied decoder layers and small loss in BLEU compared to the 6-layer version. The speed-up is significant on both device types. To cover higher-quality models, we add a "(2) Large" configuration with improved BLEU but slower translation speed. As in the previous year, we do not submit models below 26 BLEU, but due to the improved teacher-student training, we can cut down model size drastically before that threshold is reached. We are seeing respectable BLEU scores for our "(3) Small" and "(4) Tiny" configurations at impressive word-per-second rates, 2,668 and 3,597 respectively. Compared to last year's fastest CPU-bound submission (Senellart et al., 2018), these are more than three and four times faster at over 3 points higher BLEU.

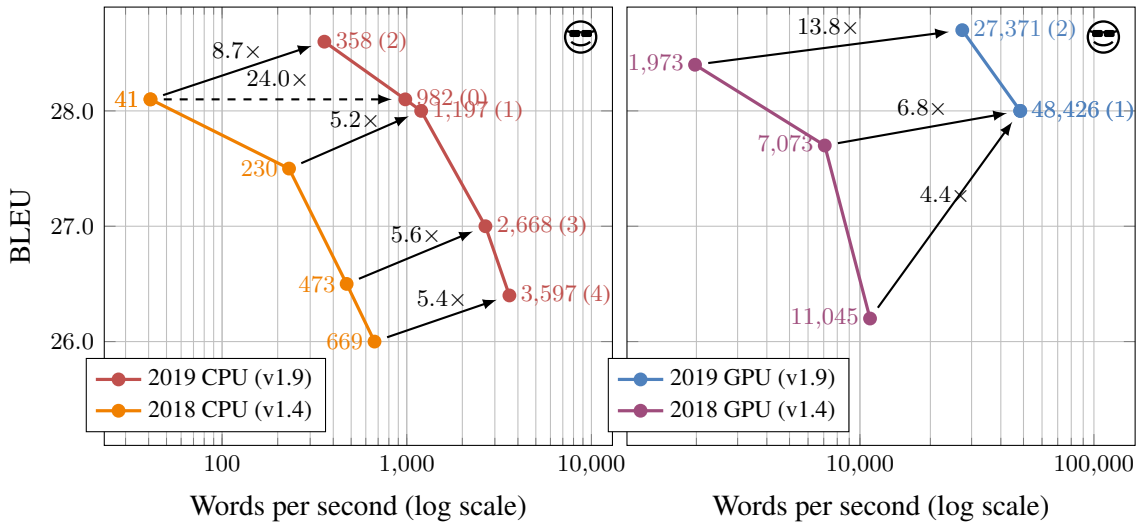


Figure 2: Relative speed improvements for fastest Marian models of comparable or better quality than submissions to WNMT2018 on newstest2014. Numbers in parentheses next to words-per-second values correspond to numbered submissions in Table 3. We also include our unsubmitted in-production model (0).

Configuration	Auto-reg.	Emb.	FFN	Depth	Params.	File size [MiB] CPU	File size [MiB] GPU	Words per second CPU	Words per second GPU	BLEU
Teacher×8	Self-Att.	1024	4096	6	1673.3 M	–		–		29.8
Base	Self-Att.	512	2048	6	60.6 M	–		–		28.5
Base	SSRU	512	2048	6	57.4 M	–		–		28.5
(0) Base	**SSRU**	**512**	**2048**	**6 tied**	**39.0 M**	–		**982**	**26,789**	**28.2**
(1) Base†‡	SSRU	512	2048	2 tied	39.0 M	85	75	1,197	48,246	28.0
(2) Large†‡	SSRU	1024	3072	6 tied	108.4 M	199	207	358	27,371	28.6
(3) Small†	SSRU	256	2048	3 tied	17.6 M	41	34	2,668	–	27.0
(4) Tiny†	SSRU	256	1536	1	15.7 M	39	31	3,597	–	26.4
(5) Base 4-bit‡	SSRU	512	2048	2 tied	39.0 M	–	19	–	23,532	27.5

Table 3: Configuration of student models and submissions. Models marked with † were submitted to the CPU track, with ‡ to the GPU track. Speed and BLEU for submissions as reported by the shared-task organizers.

The file sizes reported in Table 3 refer to prepacked 8-bit models with 32-bit embeddings for the CPU, and to models stored in FP16 for the GPU. As a separate experiment, we also applied 4-bit logarithmic quantization to further reduce the model size (Aji and Heafield, 2019) for a GPU model: "(5) Base 4-bit". This model is quantized in the form of $s \cdot 2^k$ where s is an optimized scale factor. We do not quantize biases. After initial quantization, we finetune the model to recover from the loss of quality. However, compression is only done on the model level. Therefore in this experiment, we only aim to improve the efficiency in terms of model size. We compressed the model size 8x smaller compared to 32-bit floating-point model, with a 0.5 drop to 27.5 BLEU. By quantizing the base model, we gained smaller model size (19 MiB) and better BLEU compared to the Tiny model (31 MiB).

6.2 Results and discussion

Unfortunately, in this edition of the efficiency shared task, we competed mostly against ourselves; one other team participated in the GPU track (see Figure 3), no other in the CPU track. Hence, we concentrate on improvements against our own historic results. Based on Figure 2, we can claim that the Marian 2019 team has left the Marian 2018 team in the dust. We connected each of our past submissions with one of our current submissions via a speed-up arrow if the current submission is the fastest model to have at least equal BLEU. The connected models are not necessarily similar in terms of size or architecture. By combining improved knowledge distillation methods with more efficient models and more optimized code, we were able to push the Pareto frontier towards 4.4 to 24.0 times faster translation systems at improved BLEU.

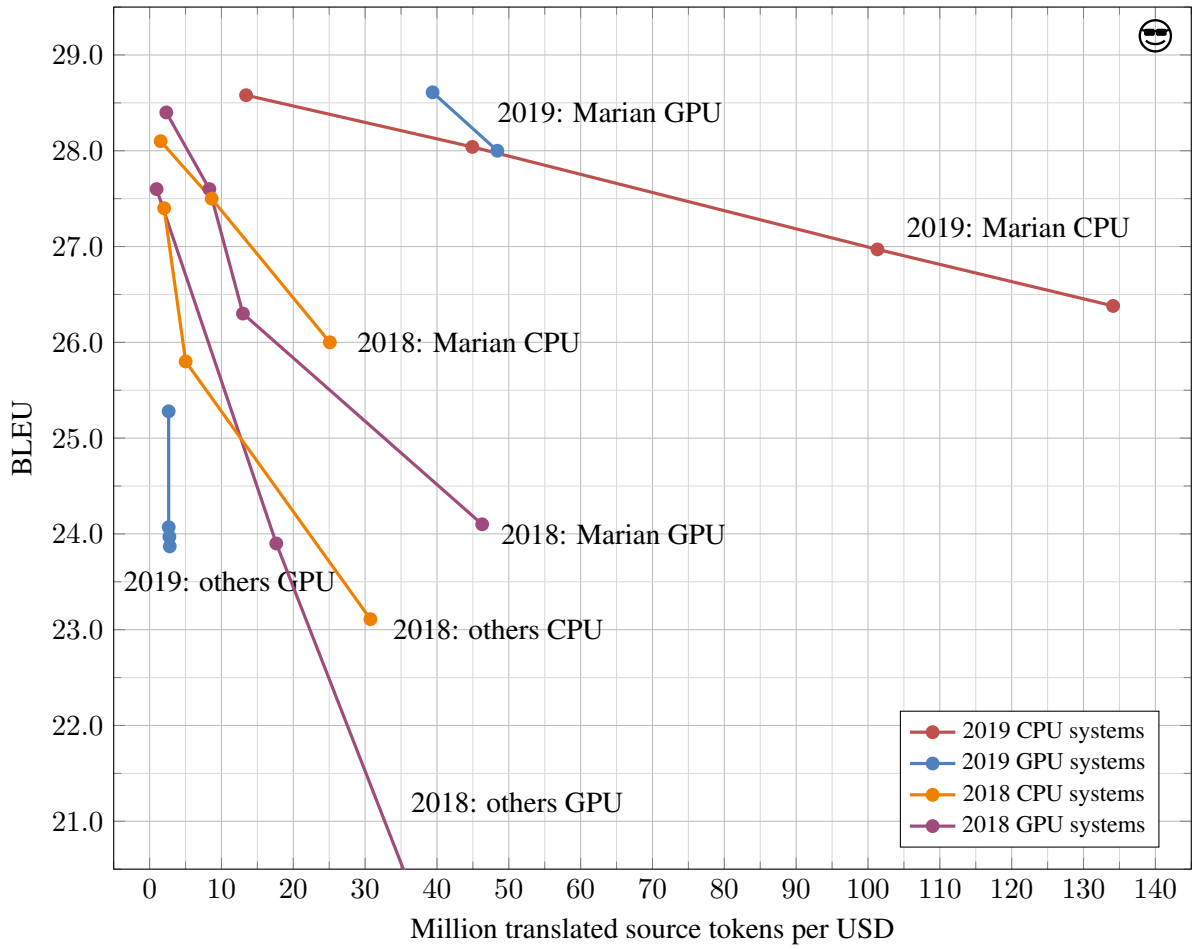


Figure 3: Pareto frontier for cost-effectiveness vs BLEU for all submissions (ours and other participants) from 2018 and 2019 on newstest2014 as reported by the organizers. We omit the weak baselines.

In Junczys-Dowmunt et al. (2018b), we compared the cost-effectiveness of GPU and CPU decoding in terms of millions of words translated per USD based on AWS instance costs. We updated the costs to reflect current prices, 0.096 USD and 3.06 USD per hour for CPU and GPU instances respectively, and visualized the results for all participants from both shared tasks — WNMT2018 and WNGT2019 — in Figure 3. Compared to last year where CPU and GPU decoding were similarly cost-effective at similar BLEU, we are starting to see a trend that highly-efficient CPU decoding is about to out-compete GPU-bound decoding in terms of cost-effectiveness according to the AWS price model.

If run on the GPU, the smaller models from our fastest CPU-track submissions would not improve speed-wise over our fastest GPU-track submissions; they would just achieve lower BLEU scores at similar speed. Our mid-sized student model already translates a WMT test set in ca. 1.5 seconds, the smaller models cannot improve over that for these short test sets. Furthermore, these are cost-effectiveness scores reported within settings of the shared task which favors (maybe unrealistically) bulk and batched translation. At Microsoft Translator, our preferred scenario is translation with batch-size 1 for low latency.

Going back to Figure 1 and comparing speed for batch-size 1 alone, we are seeing that a single CPU core with our highly optimized CPU models is faster than a Volta 100 GPU with the same models. This may of course be an artifact of under-optimized GPU performance in Marian, but on the other hand, we do not see any other participant in the shared task with more efficient GPU decoding algorithms. There is also the unexplored question of multi-core CPU decoding, where the current shared-task setup — again somewhat unrealistically — allows only single-thread CPU-bound submissions. Improvements here might go a long way in term of better cost-effectiveness on the CPU compared to the GPU.

Acknowledgments

The authors would like to thank Shufang Xie from Microsoft Research Asia for his help with the MADL training procedure. Co-authors from the University of Edinburgh would like to acknowledge:

This work was supported by funding from the European Union's Horizon 2020 research and innovation programme under grant agreements No 825303 (Bergamot).

It was performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (http://www.csd3.cam.ac.uk/), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk).

References

Alham Fikri Aji and Kenneth Heafield. 2019. Neural machine translation with 4-bit precision and beyond.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer Normalization. In NIPS 2016 Deep Learning Symposium, Barcelona, Spain.

Aishwarya Bhandare, Vamsi Sripathi, Deepthi Karkada, Vivek Menon, Sun Choi, Kushal Datta, and Vikram Saletore. 2019. Efficient 8-bit quantization of transformer neural machine language translation model. arXiv preprint arXiv:1906.00532.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. 2019. Universal transformers. In 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019.

Jacob Devlin. 2017. Sharp models on dull hardware: Fast and accurate neural machine translation decoding on the CPU. In Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, pages 2820–2825, Copenhagen, Denmark. Association for Computational Linguistics.

Sergey Edunov, Myle Ott, Michael Auli, and David Grangier. 2018. Understanding back-translation at scale. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pages 489–500, Brussels, Belgium. Association for Computational Linguistics.

FBGEMM. Open-sourcing FBGEMM for state-of-the-art server-side inference [online].

Hiroaki Hayashi, Yusuke Oda, Alexandra Birch, Ioannis Constas, Andrew Finch, Minh-Thang Luong, Graham Neubig, and Katsuhito Sudoh. 2019. Findings of the third workshop on neural generation and translation. In Proceedings of the Third Workshop on Neural Generation and Translation.

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, et al. 2018a. Marian: Fast neural machine translation in C++. In Proceedings of ACL 2018, System Demonstrations, pages 116–121.

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

Yoon Kim and Alexander M Rush. 2016. Sequence-level knowledge distillation. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, pages 1317–1327.

Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pages 66–71.

Tao Lei, Yu Zhang, and Yoav Artzi. 2017. Training RNNs as fast as CNNs. CoRR, abs/1709.02755.

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

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

Microsoft-Translator. Neural machine translation enabling human parity innovations in the cloud [online].

Andres Rodriguez, Eden Segal, Etay Meiri, Evarist Fomenko, Young Jin Kim, Haihao Shen, and Barukh Ziv. 2018. Lower numerical precision deep learning inference and training. Intel White Paper.

Jean Senellart, Dakun Zhang, Bo Wang, Guillaume Klein, Jean-Pierre Ramatchandirin, Josep Crego, and Alexander Rush. 2018. OpenNMT system description for WNMT 2018: 800 words/sec on a single-core CPU. In Proceedings of the 2nd Workshop on Neural Machine Translation and Generation, pages 122–128, Melbourne, Australia. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In Advances in neural information processing systems, pages 5998–6008.

Yiren Wang, Yingce Xia, Tianyu He, Fei Tian, Tao Qin, ChengXiang Zhai, and Tie-Yan Liu. 2019. Multi-agent dual learning. In International Conference on Learning Representations.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv preprint arXiv:1609.08144.

Biao Zhang, Deyi Xiong, and Jinsong Su. 2018. Accelerating neural transformer via an average attention network. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1789–1798, Melbourne, Australia. Association for Computational Linguistics.