Marian: Cost-effective High-Quality Neural Machine Translation in C++

Marcin Junczys-Dowmunt[†] **Kenneth Heafield**[‡]
Hieu Hoang[‡] **Roman Grundkiewicz**[‡] **Anthony Aue**[†]

[†]Microsoft Translator
1 Microsoft Way
Redmond, WA 98121, USA

[‡]University of Edinburgh
10 Crichton Street
Edinburgh, Scotland, EU

Abstract

This paper describes the submissions of the "Marian" team to the WNMT 2018 shared task. We investigate combinations of teacher-student training, low-precision matrix products, auto-tuning and other methods to optimize the Transformer model on GPU and CPU. By further integrating these methods with the new averaging attention networks, a recently introduced faster Transformer variant, we create a number of high-quality, high-performance models on the GPU and CPU, dominating the Pareto frontier for this shared task.

1 Introduction

This paper describes the submissions of the "Marian" team to the Workshop on Neural Machine Translation and Generation (WNMT 2018) shared task (Birch et al., 2018). The goal of the task is to build NMT systems on GPUs and CPUs placed on the Pareto Frontier of efficiency in accuracy.[1]

Marian (Junczys-Dowmunt et al., 2018) is an efficient neural machine translation (NMT) toolkit written in pure C++ based on dynamic computation graphs.[2] One of the goals of the toolkit is to provide 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. Ideally this should be accomplished within a single execution engine that does not require specialized, inference-only decoders.

The CPU back-end in Marian is a very recent addition and we use the shared-task as a testing ground for various improvements. The GPU-bound computations in Marian are already highly optimized and we mostly concentrate on modeling aspects and beam-search hyper-parameters.

The weak baselines (at 16.9 BLEU on newstest2014 at least 12 BLEU points below the state-of-the-art) could promote approaches that happily sacrifice quality for speed. We choose a quality cut-off of around 26 BLEU for the first test set (newstest2014) and do not spend much time on systems below that threshold.[3] This threshold was chosen based on the semi-official Sockeye (Hieber et al., 2017) baseline (27.6 BLEU on newstest2014) referenced on the shared task page.[4]

We believe our CPU implementation of the Transformer model (Vaswani et al., 2017) and attention averaging networks (Zhang et al., 2018) to be the fastest reported so far. This is achieved by integer matrix multiplication with auto-tuning. We also show that these models respond very well to sequence-level knowledge-distillation methods (Kim and Rush, 2016).

2 Teacher-student training

2.1 State-of-the-art teacher

Based on Kim and Rush (2016), we first build four strong teacher models following the procedure for the Transformer-big model (model size 1024, filter size 4096, file size 813 MiB) from Vaswani et al. (2017) for ensembling. We use 36,000 BPE joint subwords (Sennrich et al., 2016) and a joint vocabulary with tied source, target, and output embeddings. One model is trained until convergence for eight days on four P40 GPUs. See tables 3 and 4 for BLEU scores of an overview of BLEU scores for models trained in this work.

[1]See the shared task description: https://sites.google.com/site/wnmt18/shared-task

[2]https://marian-nmt.github.io

[3]We added smaller post-submission systems to demonstrate that our approach outperforms systems by other participants when we take part in the race to the quality bottom.

[4]https://github.com/awslabs/sockeye/tree/wnmt18/wnmt18

Proceedings of the 2nd Workshop on Neural Machine Translation and Generation, pages 129–135
Melbourne, Australia, July 20, 2018. ©2018 Association for Computational Linguistics

Model	Emb.	FFN	MiB
Transformer-big	1024	4096	813
Transformer-base	512	2048	238
Transformer-small	256	2048	101
Transformer-tiny-256*	256	1536	84
Transformer-tiny-192*	192	1536	60

Table 1: Transformer students dimensions. Post-submission models marked with *.

2.2 Interpolated sequence-level knowledge-distillation

As described by Kim and Rush (2016), we re-translate the full training corpus source data with the teacher ensemble as an 8-best list. Among the eight hypotheses per sentence we choose the translation with the highest sentence-level BLEU score with regard to the original target corpus. Kim and Rush (2016) refer to this method as interpolated sequence-level knowledge-distillation. Next, we train our student models exclusively on the newly generated and selected output.

2.3 Decoding with small beams

Whenever we use beam size 1, we skip softmax evaluation and simply select the output word with highest activation. The input sentences are sorted by source length, then decoded in batches of approximately equal length. We batch based on number of words. For CPU decoding we use a batch size of at least 384 words (ca. 15 sentences), for the GPU at least 8192 words (ca. 300 sentences).

3 Student architectures

3.1 Transformer students

For our Transformer student models we follow the Transformer-big and Transformer-base configurations from Vaswani et al. (2017). Additionally we investigate a Transformer-small and post-submission two Transformer-tiny variants on the CPU. We also use six blocks of self-attention, source-attention, and FFN layers with varying embedding (model) and FNN sizes, see Table 1.

Transformer-big is initialized with one of the original teachers and fine-tuned on the teacher-generated data until development set BLEU stops improving for beam-size 1. The remaining student models are trained from scratch on teacher-generated data until development set BLEU stalls for 20 validation steps when using beam-size 1.

3.2 Averaging attention networks

Very recently, Zhang et al. (2018) suggested averaging attention networks (AAN), a modification of the original Transformer model that addresses a decode-time inefficiency, apparently without loss of quality. During translation, the self-attention layers in the Transformer decoder look back at their entire history, introducing quadratic complexity with respect to output length. Zhang et al. (2018) replace the decoder self-attention layer with a cumulative uniform averaging operation across the previous layer. During decoding, this operation can be computed based on the single last step. Decoding is then linear with respect to output length. Zhang et al. (2018) also add a feed-forward network and a gate to the block. We choose a smaller FFN size than Zhang et al. (2018) (corresponding to embeddings size instead of FFN size in table 1) and experiment with removing the FFN and gate.

3.3 RNN-based students

Our focus lies on efficient CPU-bound Transformer implementations. However, Marian and its predecessor Amun (Junczys-Dowmunt et al., 2016) were first implemented as fast GPU-bound implementations of Nematus-style (Sennrich et al., 2017b) RNN-based translation models. We use these models to cover the lower end of the quality spectrum in the task. We train a standard shallow GRU model (RNN-Nematus, embedding size 512, state size 1024), a small version (RNN-small, embedding size 256, state size 512) and a deep version with 4 stacked GRU blocks in the encoder and 8 stacked GRU blocks in the decoder (RNN-deep, embedding size 512, states size 1024). This model corresponds to the University of Edinburgh submission to WMT 2017 (Sennrich et al., 2017a).

4 Optimizing for the CPU

Most of our effort was concentrated on improving CPU computation in Marian. Apart from improvements from code profiling and bottleneck identification, we worked towards integrating integer-based matrix products into Marian's computation graphs.

4.1 Shortlist

A simple way to improve CPU-bound NMT efficiency is to restrict the final output matrix multiplication to a small subset of translation candidates. We use a shortlist created with fastalign (Dyer et al., 2013). For every mini-batch we restrict the output

vocabulary to the union of the 100 most frequent target words and the 100 most probable translations for every source word in a batch. All CPU results are computed with a shortlist.

4.2 Quantization and integer products

Previously, Marian tensors would only work with 32-bit floating point numbers. We now support tensors with underlying types corresponding to the standard numerical types in C++. We focus on integer tensors.

Some of our submissions replaced 32-bit floating-point matrix multiplication with 16-bit or 8-bit signed integers. For 16-bit integers, we follow Devlin (2017) in simply multiplying parameters and inputs by 2^{10} before rounding to signed integers. This does not use the full range of values of a 16-bit integer so as to prevent overflow when accumulating 32-bit sums; there is no AVX512F instruction for 32-bit add with saturation.

For 8-bit integers, we swept quantization multipliers and found that 29 was optimal, but quality was still poor. Instead, we retrained the model with matrix product inputs (activations and parameters but not outputs) clipped to a range. We tried $[-3, 3]$, $[-2, 2]$, and $[-1, 1]$ then settled on $[-2, 2]$ because it had slightly better BLEU.[5] Values were then scaled linearly to $[-127, 127]$ and rounded to integers. We accumulated in 16-bit integers with saturation because this was faster, observing a 0.05% BLEU drop relative to 32-bit accumulation.

The test CPU is a Xeon Platinum 8175M with support for AVX512. We used these instructions to implement matrix multiplication over 32 16-bit integers or 64 8-bit integers at a time.[6]

4.3 Memoization

To ensure contiguous memory access, the integer matrix product $\mathrm{dot}'_{\mathrm{int}}(A, B)$ calculates AB^{T} instead of AB. It also expects its inputs A and B to be correctly quantized integer tensors. Therefore, we have to compute $\mathrm{dot}'_{\mathrm{int}}(\mathrm{quant}_{\mathrm{int}}(A), \mathrm{quant}_{\mathrm{int}}(B^{\mathrm{T}}))$ to use the quantized integer product as a replacement for the floating point matrix product.

In most cases, B is a parameter, while A contains activations. Repeating the quantization and transposition operations for every decoder parameter at every step would incur a significant performance penalty. To counter this, we introduce memoization into Marian's computation graphs. Memoization caches the values of constant nodes that will not change during the lifetime of the graph.

[5]This might however have been an artifact of the posterior clipping process rather than an effect of quantization.

[6]The only packed 8-bit multiplication instruction is `vpmaddubsw`, which requires AVX512BW. Interestingly, Amazon's hypervisor hides support for AVX512BW from CPUID but the instruction works as expected so we used it.

Model	1s	384w	BLEU
Transf.-base-AAN	1018.8	397.5	27.5
+shortlist	758.1	293.7	27.5
+int16	2703.2	491.4	27.5
+memoization	572.9	294.3	27.5
+auto-tuning	574.8	273.2	27.5
Transformer-big	4797.0	1537.8	28.1
+clip=2 (+mem.)	5006.9	1737.1	27.7
+int8 (+mem.)	1772.6	1169.9	27.5

Table 2: Time to translate newstest2014 with batch-size equal to 1 sentence (1s) and around 384 words (384w) using integer multiplication variants vs 32-bit float matrix multiplication.

During inference, parameter nodes are constant. Apart from that any node with only constant children is constant and can be memoized. In our example, B is constant as a parameter, B^{T} is constant because its only child is constant, so is $\mathrm{quant}_{\mathrm{int}}(B^{\mathrm{T}})$. $\mathrm{dot}'_{\mathrm{int}}(\mathrm{quant}_{\mathrm{int}}(A), \mathrm{quant}_{\mathrm{int}}(B^{\mathrm{T}}))$ itself is not constant, as the activations A can change. Values for constant nodes are calculated only once during the first forward step in which they appear; subsequent calls will use cached versions.

4.4 Auto-tuning

At this point, the float32 (Intel's MKL) product and our int16 matrix product can be used interchangeably for small and mid-sized models (we see overflow for the large Transformer model). While trying to choose one implementation, we noticed that both algorithms will outperform the respective other in different contexts. In the face of many different matrix sizes and access patterns it is difficult to determine reliable performance profiles. Instead, we implemented an auto-tuner.

We hash tensor shapes and algorithm IDs and annotate each node in an alternative subgraph with a timer. We collect the total execution time across 100 traversals of each alternate subgraph. Once this limit has been reached, usually within a few sentences, the auto-tuner stops measurements and selects the fastest alternative for all subsequent calls.

4.5 Optimization results

Table 2 illustrates the effects of the optimizations introduced in this section for sentence-by-sentence and batched translation. Adding a shortlist improves translation speed significantly. Enabling int16 multiplication without memoization hurts performance; with memoization we see improvements for single-sentence translation and similar performance to MKL for batched translation. With auto-tuning, single-sentence translation achieves the same performance as before and batched translation improves. In both cases the auto-tuning algorithm was able to choose a good solution. In the single-sentence case we would always use the int16 product. In the batched case a mix performs better than a hard choice.

We also see respectable improvements for the Transformer-big model with int8 multiplication. Most of the loss in BLEU is due to the fine-tuning process with clipping during training.

5 Results and cost-effective decoding

In tables 3 and 4, we summarize our experiments with GPU and CPU models. Bold rows contain results for our task submissions. We report model sizes in MiB, translation time without initialization and BLEU scores for newstest2014. Time has been measured on AWS p3.x2large instances (NVidia V100) and AWS m5.large instances, the official evaluation platforms of the shared task.

All our student models outperform the baselines in terms of translation speed and quality, but as stated before, we are mostly interested in models above a 26 BLEU threshold. It seems that the new AAN architecture is a promising modification of the Transformer with minimal or no quality loss in comparison to its standard equivalent. We also see that teacher-student methods can be successfully used to create high-performance and high-quality Transformer systems with greedy decoding.

We compare our systems on a common cost-effectiveness scale expressed as the number of source tokens translated per US Dollar $\left[\frac{\text{w}}{\text{USD}}\right]$. Given the hourly price for a dedicated AWS GPU (p3.x2large, 3.259 USD/h) or CPU (m5.large, 0.102 USD/h) instance[7] and the time to translate newstest2014 consisting of 62,954 source tokens with a chosen model and instance, we calculate:

$$\frac{62,954\ [\text{w}]}{\text{Translation time [s]}} \cdot \frac{3,600\ \ [\text{s/h}]}{\text{Instance price}\ \ [\$/\text{h}]}.$$

[7] The same instance types were used for the shared task.

This representation has multiple advantages:

- Systems deployed on different hardware can be compared directly;
- The linear mappings into the common space are scale-preserving and correctly represent relative speed differences between systems on the same hardware;
- We can relate three important categories — speed, quality, and cost — to each other in a single visualization.

Figures 1 and 2 illustrate cost-effectiveness of our models, the baselines and submissions by other participants versus translation quality on newstest2014. Figure 1 contains all models with a cost-effectiveness log-scale. This reflects a trend that speed gains are exponential in quality loss. Based on Figure 1, it seems that our models dominate the Pareto-frontier for high-quality models for CPU and GPU models compared to the baselines and other participants.

We added post-submission systems (23i) and (24i) on the CPU to demonstrate that we can outperform the results of other participants for speed and quality when lowering our quality threshold.

In Figure 2 with a linear cost-effectiveness scale, we emphasize models around and above the quality threshold of 26 BLEU which were our main focus in this work. It is interesting to see that similar Marian models have surprisingly similar cost-effectiveness across different hardware types.

6 Conclusions

We demonstrated that Marian can serve as an integrated research and deployment platform with highly efficient decoding algorithms on the GPU and CPU. Transformer architectures can be efficiently trained in teacher-student settings and then used with small beams or with greedy decoding. To our knowledge, this is also the first work to integrate Transformer architectures with low-precision matrix multiplication. By combining these methods with the new averaging attention networks, we created a number of high-quality, high-performance models on the GPU and CPU, dominating the Pareto frontier for this shared task.

Acknowledgments

We thank Jon Clark who suggested the graphing in USD. Partly supported by a Mozilla Research Grant.

No	Model	MiB	Time	BLEU
(1)	Baseline GPU	–	51.6	16.8
(2)	Sockeye GPU (Transformer-base b=5)	–	231.9	27.6
(3)	Teacher - Transformer-big b=8	813	109.7	28.1
(4)	Teacher - Transformer-big×4 b=8	3252	410.8	29.0
(5)	Transformer-big b=4	813	52.0	28.4
(6)	**Transformer-big b=2**	**813**	**31.9**	**28.4**
(7)	Transformer-big	813	19.9	28.2
(8)	Transformer-base b=4	238	40.5	27.8
(9)	Transformer-base b=2	238	22.9	27.8
(10)	Transformer-base	238	12.8	27.6
(11)	Transformer-base-AAN b=4	220	15.9	27.7
(12)	**Transformer-base-AAN b=2**	**220**	**8.9**	**27.7**
(13)	Transformer-base-AAN	220	7.2	27.6
(14)	Transformer-small	101	10.8	26.4
(15)	Transformer-small-AAN	100	5.9	25.8
(16)	**Transformer-small-AAN -ffn**	**98**	**5.7**	**26.2**
(17)	Transformer-small-AAN -ffn -gate	95	5.6	25.8
(18)	**RNN-small-Amun**	**88**	**1.6**	**24.1**
(19)	RNN-Nematus-Amun	199	2.2	24.8
(20)	RNN-small	88	1.8	24.1
(21)	RNN-Nematus	199	2.5	24.8
(22)	RNN-Deep	323	2.9	25.7

Table 3: Results on newstest2014 - GPU systems. Submitted systems in bold. All student systems have been used with beam-size 1 unless stated differently (b=n).

No	Model	MiB	Time	BLEU
(1)	Baseline CPU	–	4492.2	16.8
(2)	Sockeye CPU (Transformer-base b=5)	–	1168.6	27.4
(7)	**Transformer-big**	**813**	**1537.8**	**28.1**
(7i)	**Transformer-big-int8**	**813**	**1169.9**	**27.5**
(10)	Transformer-base	238	393.1	27.4
(10i)	Transformer-base-int16	238	400.2	27.4
(13)	Transformer-base-AAN	220	288.7	27.5
(13i)	**Transformer-base-AAN-int16**	**220**	**273.2**	**27.5**
(14)	Transformer-small	101	134.1	26.5
(14i)	Transformer-small-int16	101	133.2	26.5
(15i)	Transformer-small-AAN-int16	100	108.8	25.8
(16i)	Transformer-small-AAN-int16 -ffn	98	108.3	26.2
(17)	Transformer-small-AAN -ffn -gate	95	100.6	26.0
(17i)	**Transformer-small-AAN-int16 -ffn -gate**	**95**	**94.1**	**26.0**
(23i)	Transformer-tiny-256-AAN-int16 -ffn -gate*	84	79.7	25.3
(24i)	Transformer-tiny-192-AAN-int16 -ffn -gate*	60	61.1	24.4

Table 4: Results on newstest2014 - CPU systems. Submitted systems in bold. Post-submission systems marked with *. All student systems have been used with beam-size 1 unless stated differently (b=n).

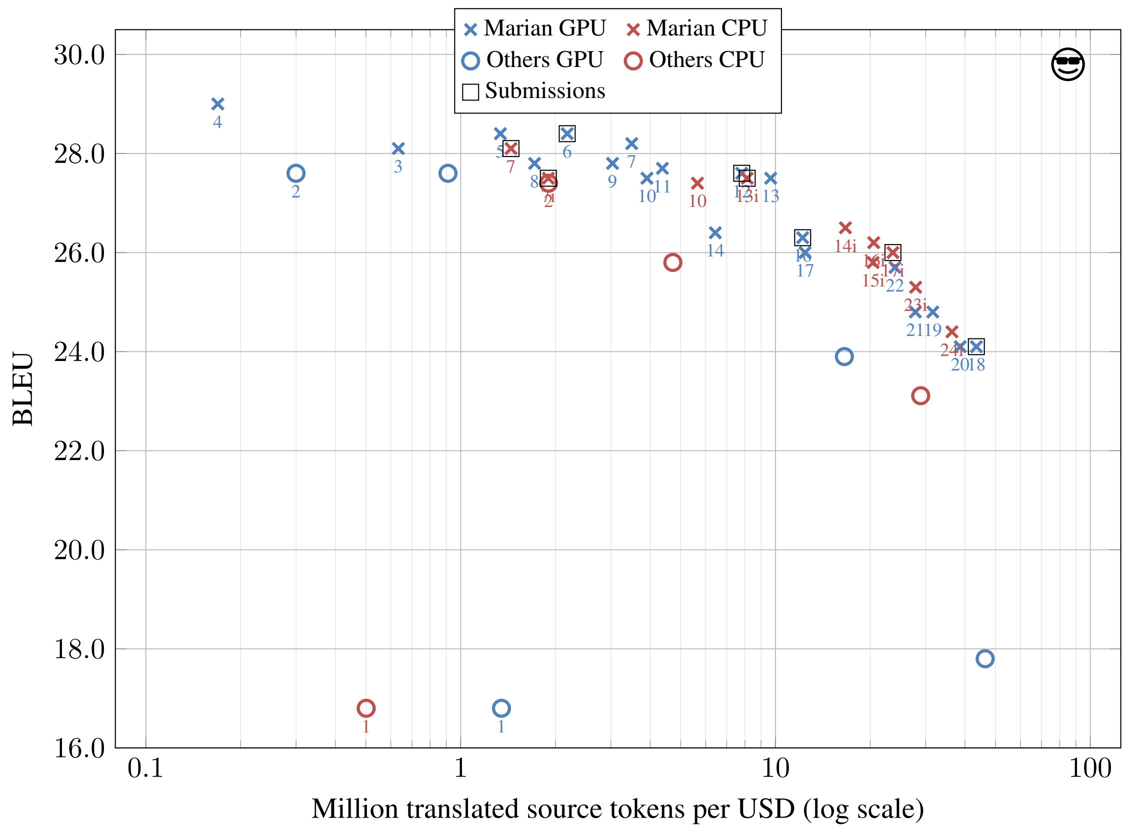

Figure 1: Cost-effectiveness (logarithmic scale) vs BLEU for all systems and baselines.

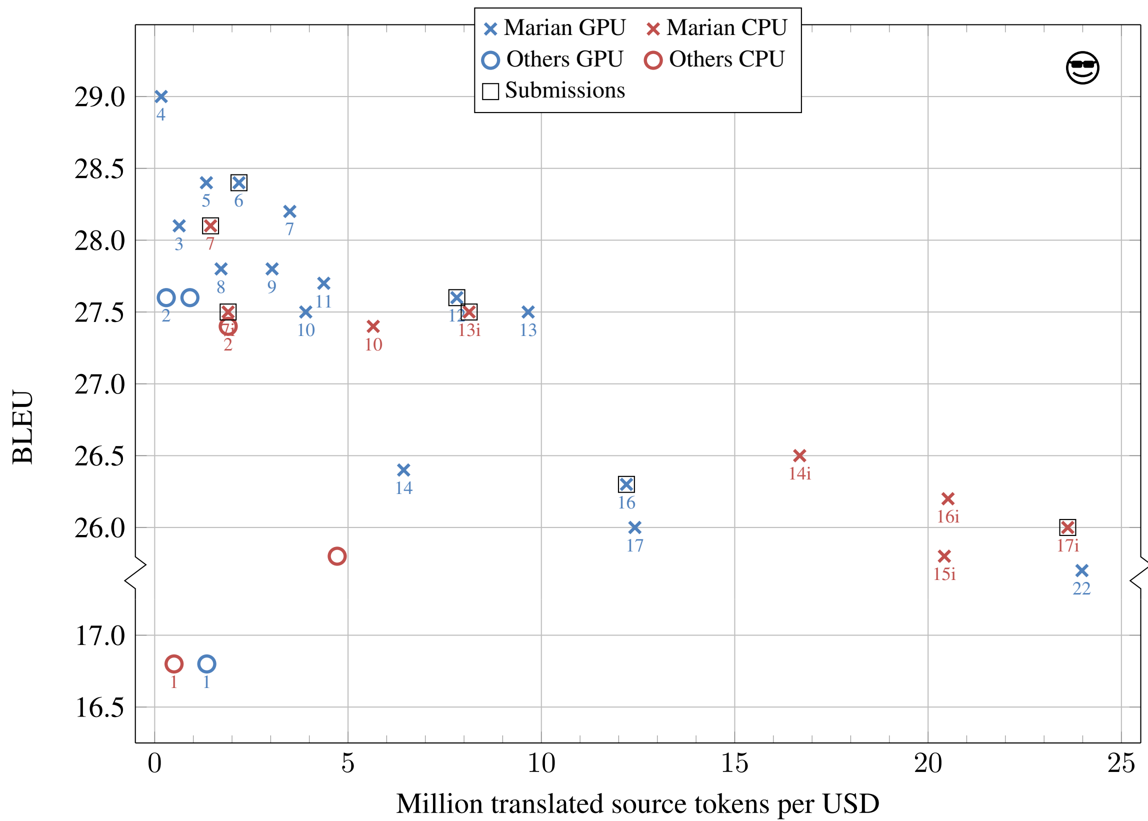

Figure 2: Cost-effectiveness (linear scale) vs BLEU for systems around and above 26 BLEU and baselines.

References

Alexandra Birch, Andrew Finch, Minh-Thang Luong, Graham Neubig, and Yusuke Oda. 2018. Findings of the Second Workshop on Neural Machine Translation and Generation. In The Second Workshop on Neural Machine Translation and Generation.

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.

Chris Dyer, Victor Chahuneau, and Noah A. Smith. 2013. A simple, fast, and effective reparameterization of IBM model 2. In HLT-NAACL, pages 644–648. The Association for Computational Linguistics.

Felix Hieber, Tobias Domhan, Michael Denkowski, David Vilar, Artem Sokolov, Ann Clifton, and Matt Post. 2017. Sockeye: A toolkit for neural machine translation. CoRR, abs/1712.05690.

Marcin Junczys-Dowmunt, Tomasz Dwojak, and Hieu Hoang. 2016. Is neural machine translation ready for deployment? A case study on 30 translation directions. In Program of the 13th International Workshop on Spoken Language Translation (IWSLT 2016).

Marcin Junczys-Dowmunt, Roman Grundkiewicz, Tomasz Dwojak, Hieu Hoang, Kenneth Heafield, Tom Neckermann, Frank Seide, Ulrich Germann, Alham Fikri Aji, Nikolay Bogoychev, André F. T. Martins, and Alexandra Birch. 2018. Marian: Fast neural machine translation in C++. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, Melbourne, Australia.

Yoon Kim and Alexander M. Rush. 2016. Sequence-level knowledge distillation. In Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing, EMNLP 2016, Austin, Texas, USA, November 1-4, 2016, pages 1317–1327.

Rico Sennrich, Alexandra Birch, Anna Currey, Ulrich Germann, Barry Haddow, Kenneth Heafield, Antonio Valerio Miceli Barone, and Philip Williams. 2017a. The University of Edinburgh's neural MT systems for WMT17. In Proceedings of the Second Conference on Machine Translation, Volume 2: Shared Task Papers, pages 389–399, Copenhagen, Denmark. Association for Computational Linguistics.

Rico Sennrich, Orhan Firat, Kyunghyun Cho, Alexandra Birch, Barry Haddow, Julian Hitschler, Marcin Junczys-Dowmunt, Samuel Läubli, Antonio Valerio Miceli Barone, Jozef Mokry, and Maria Nadejde. 2017b. Nematus: a toolkit for neural machine translation. In Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics, pages 65–68, Valencia, Spain. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1715–1725, Berlin, Germany. 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 I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, Advances in Neural Information Processing Systems 30, pages 5998–6008. Curran Associates, Inc.

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), Berlin, Germany. Association for Computational Linguistics.