Fast Neural Machine Translation Implementation

Hieu Hoang[†] **Tomasz Dwojak**[*] **Rihards Krislauks**[‡]
Daniel Torregrosa[¶] **Kenneth Heafield**[†]

[†]University of Edinburgh [*]Adam Mickiewicz University
[‡]Tilde [¶]Universitat d'Alacant

Abstract

This paper describes the submissions to the efficiency track for GPUs at the Workshop for Neural Machine Translation and Generation by members of the University of Edinburgh, Adam Mickiewicz University, Tilde and University of Alicante. We focus on efficient implementation of the recurrent deep-learning model as implemented in Amun, the fast inference engine for neural machine translation. We improve the performance with an efficient mini-batching algorithm, and by fusing the softmax operation with the k-best extraction algorithm. Submissions using Amun were first, second and third fastest in the GPU efficiency track.

1 Introduction

As neural machine translation (NMT) models have become the new state-of-the-art, the challenge is to make their deployment efficient and economical. This is the challenge that this shared task (Birch et al., 2018) is shining a spotlight on.

One approach is to use an off-the-shelf deep-learning toolkit to complete the shared task where the novelty comes from selecting the appropriate models and tuning parameters within the toolkit for optimal performance.

We take an opposing approach by eschewing model selection and parameter tuning in favour of efficient implementation. We use and enhanced a custom inference engine, Amun (Junczys-Dowmunt et al., 2016), which we developed on the premise that fast deep-learning inference is an issue that deserves dedicated tools that are not compromised by competing objectives such as training or support for multiple models. As well as delivering on the practical goal of fast inference, it can serve as a test-bed for novel ideas on neural network inference, and it is useful as a means to explore the upper bound of the possible speed for a particular model and hardware. That is, Amun is an inference-only engine that supports a limited number of NMT models that put fast inference on modern GPU above all other considerations.

We submitted two systems to this year's shared task for the efficient translation on GPU. Our first submission was tailored to be as fast as possible while being above the baseline BLEU score. Our second submission trades some of the speed of the first submission to return better quality translations.

2 Improvements

We describe the main enhancements to Amun since the original 2016 publication that has improved translation speed.

2.1 Batching

The use of mini-batching is critical for fast model inference. The size of the batch is determined by the number of inputs sentences to the encoder in an encoder-decoder model. However, the number of batches during decoding can vary as some sentences have completed translating or the beam search add more hypotheses to the batch.

It is tempting to ignore these considerations, for example, by always decoding with a constant batch and beam size and ignoring hypotheses which are not needed. Figure 1 illustrates a naïve mini-batching with a constant size batch. The downside to this algorithm is lower translation speed due to wasteful processing.

Amun implements an efficient batching algorithm that takes into account the actual number of hypotheses that need to be decoded at each decoding step, Figure 2.

Proceedings of the 2nd Workshop on Neural Machine Translation and Generation, pages 116–121
Melbourne, Australia, July 20, 2018. ©2018 Association for Computational Linguistics

Algorithm 1 Naïve mini-batching

```
procedure BATCHING(encoded sentences i)
    Create batch b from i
    while hypo h ≠ EOS, ∀h ∈ b do
        Decode(b)
    end while
end procedure
```

Algorithm 2 Mini-batching

```
procedure BATCHING(encoded sentences i)
    Create batch b from i
    while b ≠ ∅ do
        Decode(b)
        for all hypo h ∈ b do
            if h = EOS then
                Remove h from b
            end if
        end for
    end while
end procedure
```

We will compare the effect of the two implementations in the Section 4.

2.2 Softmax and K-Best Fusion

Most NMT models predict a large number of classes in their output layer, corresponding to the number of words or subword units in their target language. For example, Sennrich et al. (2016) experimented with target vocabulary sizes of 60,000 and 90,000 sub-word units.

The output layer of most deep learning models consist of the following steps

1. multiplication of the weight matrix with the input vector $p = wx$
2. addition of a bias term to the resulting scores $p = p + b$
3. applying the activation function, most commonly softmax $p_i = \exp(p_i)/\sum \exp(p_i)$
4. a search for the best (or k-best) output classes $\mathrm{argmax}_i\, p_i$

Figure 1 shows the amount of time spent in each step during translation. Clearly, the output layer of NMT models are very computationally expensive, accounting for over 60% of the translation time.

We focus on the last three steps; their outline is shown in Algorithm 3. For brevity, we show the algorithm for 1-best, a k-best search is a simple extension of this.

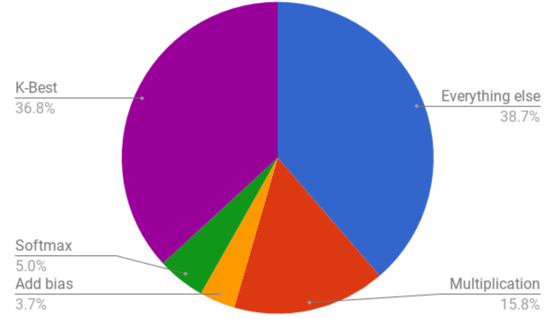

Figure 1: Proportion of time spent during translation

Algorithm 3 Original softmax and k-best algorithm

```
procedure ADDBIAS(vector p, bias vector b)
    for all p_i in p do
        p_i ← p_i + b_i
    end for
end procedure

procedure SOFTMAX(vector p)
                    ▷ calculate max for softmax stability
    max ← −∞
    for all p_i in p do
        if p_i > max then
            max ← p_i
        end if
    end for
                    ▷ calculate denominator
    sum ← 0
    for all p_i in p do
        sum ← sum + exp(p_i − max)
    end for
                    ▷ calculate softmax
    for all p_i in p do
        p_i ← exp(p_i − max) / sum
    end for
end procedure

procedure FIND-BEST(vector p)
    max ← −∞
    for all p_i in p do
        if p_i > max then
            max ← p_i
            best ← i
        end if
    end for
    return max, best
end procedure
```

As can be seen, the vector p is iterated over five times - once to add the bias, three times to calculate the softmax, and once to search for the best classes. We propose fusing the three functions into one kernel, a popular optimization technique (Guevara et al., 2009), making use of the following observations.

Firstly, softmax and exp are monotonic functions, therefore, we can move the search for the best class from FIND-BEST to SOFTMAX, at the start of the kernel.

Secondly, we are only interested in the probabilities of the best classes during inference, not of all classes. Since they are now known at the start of the softmax kernel, we compute softmax only for those classes.

Algorithm 4 Fused softmax and k-best

```
procedure FUSED-KERNEL(vector p, bias vector b)
    max ← −∞
    sum ← 0
    for all p_i in p do
        p'_i ← p_i + b_i
        if p'_i > max then
            Δ ← max − p'_i
            sum ← Δ × sum + 1
            max ← p'_i
            best ← i
        else
            sum ← sum + exp(p'_i − max)
        end if
    end for
     return 1/sum, best
end procedure
```

Thirdly, the calculation of max and sum can be accomplished in one loop by adjusting sum whenever a higher max is found during the looping:

$$\begin{aligned} sum &= e^{x_t - max_b} + \sum_{i=0...t-1} e^{x_i - max_b} \\ &= e^{x_t - max_b} + \sum_{i=0...t-1} e^{x_i - max_a + \Delta} \\ &= e^{x_t - max_b} + e^{\Delta} \times \sum_{i=0...t-1} e^{x_i - max_a} \end{aligned}$$

where max_a is the previous maximum value, max_b is the now higher maximum value, i.e., $max_b > max_a$, and $\Delta = max_a - max_b$. The outline of our function is shown in Algorithm 4.

In fact, a well known optimization is to skip softmax altogether and calculate the argmax over the input vector, Algorithm 5. This is only possible for beam size 1 and when we are not interested in returning the softmax probabilities.

Algorithm 5 Find 1-best only

```
procedure FUSED-KERNEL-1-BEST(vector p, bias vector b)
    max ← −∞
    for all p_i in p do
        if p_i + b_i > max then
            max ← p_i + b_i
            best ← i
        end if
    end for
     return best
end procedure
```

Since we are working on GPU optimization, it is essential to make full use of the many GPU cores available. This is accomplished by well-known parallelization methods which multi-thread the algorithms. For example, Algorithm 5 is parallelized by sharding the vector p and calculating $best$ and max on each shard in parallel. The ultimate $best$ is found in the following reduction step, Algorithm 6.

2.3 Half-Precision

Reducing the number of bits needed to store floating point values from 32-bits to 16-bits promises to increase translation speed through faster calculations and reduced bandwidth usage. 16-bit floating point operations are supported by the GPU hardware and software available in the shared task.

In practise, however, efficiently using half-precision value requires a comprehensive redevelopment of the GPU code. We therefore make do with using the GPU's Tensor Core[1] fast matrix multiplication routines which transparently converts 32-bit float point input matrices to 16-bit values and output a 32-bit float point product of the inputs.

3 Experimental Setup

Both of our submitted systems use a sequence-to-sequence model similar to that described in Bahdanau et al. (2014), containing a bidirectional

[1]https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/

Algorithm 6 Parallel find 1-best only

```
procedure FUSED-KERNEL-1-BEST(vector p, bias vector b)
                                              ▷ parallelize
    Create shards p^1...p^n from p
    parfor p^j ∈ p^1...p^n do
        max^j ← −∞
        for all p_i^j in p^j do
            if p_i^j + b_i > max then
                max^j ← p_i^j + b_i
                best^j ← i
            end if
        end for
    end parfor
                                              ▷ reduce
    max ← −∞
    for all max^j ∈ max^1...max^n do
        if max^j > max then
            max ← max^j
            best ← best^j
        end if
    end for
    return best
end procedure
```

RNN in the encoder and a two-layer RNN in the decoder. We use byte pair encoding (Sennrich et al., 2016) to adjust the vocabulary size.

We used a variety of GPUs to train the models but all testing was done on an Nvidia V100. Translation quality was measured using BLEU, specifically multi-bleu as found in the Moses toolkit[2]. The validation and test sets provided by the shared task organisers were used to measure translation quality, but a 50,000 sentence subset of the training data was used to measure translation speed to obtain longer, more accurate measurements.

3.1 GRU-based system

Our first submitted system uses gated recurred units (GRU) throughout. It was trained using Marian (Junczys-Dowmunt et al., 2018), but Amun was chosen as inference engine.

We experimented with varying the vocabulary size and the RNN state size before settling for a vocabulary size of 30,000 (for both source and target language) and 256 for the state size, Table 1.

After further experimentation, we decided to use sentence length normalization and NVidia's Tensor Core matrix multiplication which increased translation quality as well as translation speed. The beam was kept at 1 throughout for the fastest possible inference.

[2]https://github.com/moses-smt/mosesdecoder

	State dim		
Vocab size	256	512	1024
1.000	12.23		12.77
5,000	16.79		17.16
10,000	18.00		18.19
20,000	-		19.52
30,000	18.51	19.17	19.64

Table 1: Validation set BLEU (newstest2014) for GRU-based model

	Vocab size	
Beam size	40,000	50,000
1	23.45	23.32
2	24.15	24.04
3	24.48	
4	24.42	
5	24.48	

Table 2: Validation set BLEU for mLSTM-based model

3.2 mLSTM-based system

Our second system uses multiplicative-LSTM (Krause et al., 2017) in the encoder and the first layer of a decder, and a GRU in the second layer, trained with an extension of the Nematus (Sennrich et al., 2017) toolkit which supports such models; multiplicative-LSTM's suitability for use in NMT models has been previously demonstrated by Pinnis et al. (2017). As with our first submission, Amun is used as inference engine. We trained 2 systems with differing vocabulary sizes and varied the beam sizes, and chose the configuration that produced the best results for translation quality on the validation set, Table 2.

4 Result

4.1 Batching

The efficiency of Amun's batching algorithm can be seen by observing the time taken for each decoding step in a batch of sentences, Figure 2. Amun's decoding becomes faster as sentences are completely translated. This contrasts with the Marian inference engine, which uses a naïve

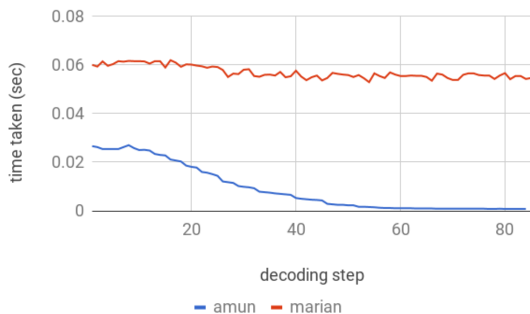

Figure 2: Time taken for each decoding step for a batch of 1280 sentences

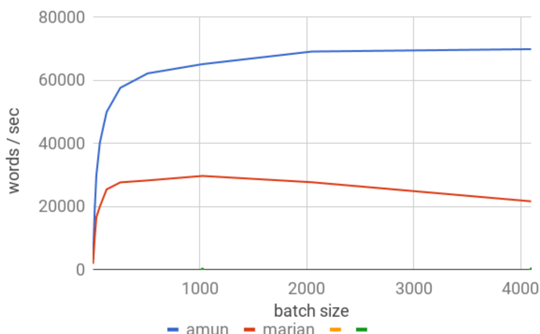

Figure 3: Speed v. batch size

batching algorithm, where the speed stays relatively constant throughout the decoding of the batch.

Using batching can increase the translation speed by over 20 times in Amun, Figure 3. Just as important, it doesn't suffer degradation with large batch sizes, unlike the naïve algorithm which slows down when batch sizes over 1000 is used. This scalability issue is likely to become more relevant as newer GPUs with ever increasing core counts are released.

4.2 Softmax and K-Best Fusion

Fusing the bias and softmax operations in the output layer with the beam search results in a speed improvement by 25%, Figure 4. Its relative improvement decreases marginally as the beam size increases.

Further insight can be gained by examining the time taken for each step in the output layer and beam search, Table 3. The fused operation only has to loop through the large cost matrix once, therefore, for low beam sizes its is comparable in speed to the simple kernel to add the bias. For higher beam sizes, the cost of maintaining the n-

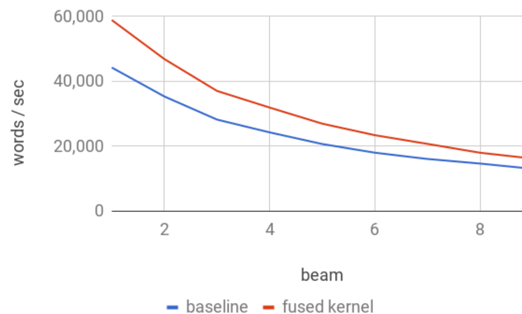

Figure 4: Using fused operation

best list is begins to impact on speed.

	Baseline	Fused
Beam size 1		
Multiplication	5.39	5.38 (+0%)
Add bias	1.26	
Softmax	1.69	2.07 (-86.6%)
K-best extr.	12.53	
Beam size 3		
Multiplication	14.18	14.16 (+0%)
Add bias	3.76	
Softmax	4.75	3.43 (-87.1%)
K-best extr.	18.23	
Beam size 9		
Multiplication	38.35	38.42 (+0%)
Add bias	11.64	
Softmax	14.4	17.5 (-72.1%)
K-best extr.	36.7	

Table 3: Time taken (sec) breakdown

4.3 Tensor Cores

By taking advantage of the GPU's hardware accelerated matrix multiplication, we can gain up to 20% in speed, Table 4.

Beam size	Baseline	Tensor Cores
1	39.97	34.54 (-13.6%)
9	145.8	116.8 (-20.0%)

Table 4: Time taken (sec) using Tensor Cores

5 Conclusion and Future Work

We have presented some of the improvement to Amun which are focused on improving NMT inference.

We are also working to make deep-learning faster using more specialised hardware such as FPGAs. It would be interesting as future work to bring our focused approach to fast deep-learning inference to a more general toolkit.

References

Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

Birch, A., Finch, A., Luong, M.-T., Neubig, G., and Oda, Y. (2018). Findings of the second workshop on neural machine translation and generation. In *The Second Workshop on Neural Machine Translation and Generation*.

Guevara, M., Gregg, C., Hazelwood, K. M., and Skadron, K. (2009). Enabling task parallelism in the cuda scheduler.

Junczys-Dowmunt, M., Dwojak, T., and Hoang, H. (2016). Is neural machine translation ready for deployment? a case study on 30 translation directions. In *Proceedings of the 9th International Workshop on Spoken Language Translation (IWSLT)*, Seattle, WA.

Junczys-Dowmunt, M., Grundkiewicz, R., Dwojak, T., Hoang, H., Heafield, K., Neckermann, T., Seide, F., Germann, U., Aji, A. F., Bogoychev, N., Martins, A. F. T., and Birch, A. (2018). Marian: Fast Neural Machine Translation in C++. *ArXiv e-prints*.

Krause, B., Murray, I., Renals, S., and Lu, L. (2017). Multiplicative LSTM for sequence modelling. *ICLR Workshop track*.

Pinnis, M., Krišlauks, R., Miks, T., Deksne, D., and Šics, V. (2017). Tilde's Machine Translation Systems for WMT 2017. In *Proceedings of the Second Conference on Machine Translation (WMT 2017), Volume 2: Shared Task Papers*, pages 374–381, Copenhagen, Denmark. Association for Computational Linguistics.

Sennrich, R., Firat, O., Cho, K., Birch, A., Haddow, B., Hitschler, J., Junczys-Dowmunt, M., Läubli, S., Miceli Barone, A. V., Mokry, J., and Nadejde, M. (2017). 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.

Sennrich, R., Haddow, B., and Birch, A. (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.