Efficient machine translation. How to get the bestest and fastest models

Nikolay Bogoychev

University of Edinburgh
N.Bogoych@ed.ac.uk
We all love winning WMT with huge machine translation systems.

- 4x transformer big - 800M parameters
- Wider transformers - 2B parameters
- NLLB - 50B parameters
- What’s next?
We all love winning WMT with huge machine translation systems.

- 4x transformer big - 800M parameters
- Wider transformers - 2B parameters
- NLLB - 50B parameters
- What’s next?

How do we actually do MT in production?
Models
Model size

- How do we speed up the machine translation?
• How do we speed up the machine translation?
• It’s simple, just use smaller models.
• How do we speed up the machine translation?
• It’s simple, just use smaller models.
• But we also want translation quality.
Knowledge distillation

- We want to learn a small model, but it has bad quality.
- Instead learn a big model (transformer-big ensemble)
- Translate your training set with your big model.
- Train your small model on the artificial data.
Training the student

- Overfit the student to the teacher distribution (all training tricks that you know apply).
Training the student

- Overfit the student to the teacher distribution (all training tricks that you know apply).
- Evaluate the student on the dev set TRANSLATED by the teacher. You expect to approach 100 BLEU.
Training the student

- Overfit the student to the teacher distribution (all training tricks that you know apply).
- Evaluate the student on the dev set TRANSLATED by the teacher. You expect to approach 100 BLEU.
- Training will take a while...
- Student can run with a beam size of 1!
Shrinking the model always reduces the computational costs, but not all parameters are born equally computationally heavy.

- Encoder runs once, decoder runs for every word.
Shrinking the model always reduces the computational costs, but not all parameters are born equally computationally heavy.

- Encoder runs once, decoder runs for every word.
- Self-attention is really expensive, especially in the decoder.
Shrinking the model always reduces the computational costs, but not all parameters are born equally computationally heavy.

- Encoder runs once, decoder runs for every word.
- Self-attention is really expensive, especially in the decoder.
- The output layer is usually the largest matrix in the model.
Encoder is much cheaper computationally than the decoder. Don’t use 6-6 configuration but explore:

- 12-1?
- 6-2?

Evaluate the speed/quality tradeoff and make a decision.
We want to reduce the computational cost of decoders of our model.

- Reduce their depth: Use only 1 or 2 layer decoder.
- Replace expensive components: Replace attention with AAN or SSRU

Evaluate the speed/quality tradeoff and make a decision.
Reducing the dimensions of the matrices is the easiest way to scale down the model

- Reduce embedding layer size: 512 -> 256 -> 128
- Reduce FFNN layer size: 2048x2048 -> 1024x1024
- Reduce the dimension of attention heads.

Evaluate the speed/quality tradeoff and make a decision.
Models have a lot of built in redundancy. Prune parameters that have little affect on the overall computation.

- Identify non important parameters during training.
- Set them to zero
- Remove them from the model

Less parameters *should* reduce the computational workload.
Decoding time tricks
The output layer matrix has size $\text{DIM}_{\text{emb}} \times |N|_{\text{vocab}}$ and is the single largest computation in the model. Speed it up by:

- Reduce vocabulary size. Bad
The output layer matrix has size $DIM_{emb} \times |N|_{vocab}$ and is the single largest computation in the model. Speed it up by:

- Reduce vocabulary size. Bad
- Use lexical shortlist.
- Use KNN clustering

Evaluate the speed/quality tradeoff and make a decision.
Our hardware is faster when multiplying larger matrices

- Group similarly sized sentences together
- Higher throughput and higher latency
Quantisation

CPUs and GPUs have 8-bit integer multiplication instructions that allow for much faster matrix multiplication than what is possible in FP32.

- Hardware allows us to do $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$. (Not really true for a lot of the hardware)
CPUs and GPUs have 8-bit integer multiplication instructions that allow for much faster matrix multiplication than what is possible in FP32.

- Hardware allows us to do $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$.
  (Not really true for a lot of the hardware)
- Quantise Activation and parameter matrices to 8-bit.
  
  \[
  A_i = \frac{127 \times A_i}{\text{MAX}(|A|)}, \quad B_i = \frac{127 \times B_i}{\text{MAX}(|B|)}
  \]
Quantisation

CPUs and GPUs have 8-bit integer multiplication instructions that allow for much faster matrix multiplication than what is possible in FP32.

- Hardware allows us to do $INT_8 \times INT_8 = INT_{32}$.
  (Not really true for a lot of the hardware)
- Quantise Activation and parameter matrices to 8-bit.
  \[ A_i = \frac{127 \times A_i}{\text{MAX}(|A|)} , \quad B_i = \frac{127 \times B_i}{\text{MAX}(|B|)} \]
- After multiplication, perform de-quantisation and re-quantisation for the next activation:
  \[ A_{fp32} \times B_{fp32} \approx A_{int8} \times B_{int8} \times \frac{\text{MAX}(A) \times \text{MAX}(B)}{127^2} \]
Quantisation

CPUs and GPUs have 8-bit integer multiplication instructions that allow for much faster matrix multiplication than what is possible in FP32.

- Hardware allows us to do $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$. (Not really true for a lot of the hardware)
- Quantise Activation and parameter matrices to 8-bit. 
  $$A_i = \frac{127 \times A_i \max(|A|)}{\max(|A|)}, \quad B_i = \frac{127 \times B_i \max(|B|)}{\max(|B|)}$$
- After multiplication, perform de-quantisation and re-quantisation for the next activation: 
  $$A_{fp32} \times B_{fp32} \approx A_{int8} \times B_{int8} \times \frac{\max(A) \times \max(B)}{127^2}$$
- Parameters are converted to 8-bit in advance, activations at runtime, always before multiplication.
- Quantisation Multipliers are computed in advance.
Speed results
Applying out full bag of tricks:

<table>
<thead>
<tr>
<th></th>
<th>TIME</th>
<th>BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU, 16 threads, 3000 SENTENCES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Teacher latency</td>
<td>4597s</td>
<td>36.5</td>
</tr>
<tr>
<td>Teacher batched</td>
<td>652s</td>
<td>36.5</td>
</tr>
<tr>
<td>Student latency</td>
<td>84s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched</td>
<td>11s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched shortlisted</td>
<td>8s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched quantised shortlisted</td>
<td>7.1s</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Applying out full bag of tricks:

<table>
<thead>
<tr>
<th>CPU, 16 threads, 3000 SENTENCES</th>
<th>TIME</th>
<th>BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teacher latency</td>
<td>4597s</td>
<td>36.5</td>
</tr>
<tr>
<td>Teacher batched</td>
<td>652s</td>
<td>36.5</td>
</tr>
<tr>
<td>Student latency</td>
<td>84s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched</td>
<td>11s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched shortlisted</td>
<td>8s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched quantised shortlisted</td>
<td>7.1s</td>
<td>35.0</td>
</tr>
</tbody>
</table>

1 Thread, to make it more granular

<table>
<thead>
<tr>
<th>CPU, 1 thread, 3000 SENTENCES</th>
<th>TIME</th>
<th>BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student latency</td>
<td>189s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched</td>
<td>38s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched shortlisted</td>
<td>27s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched quantised shortlisted</td>
<td>21s</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Applying out full bag of tricks for the GPU... Maybe it’s better to not use more all tricks.

<table>
<thead>
<tr>
<th>GPU, 100k SENTENCES</th>
<th>TIME</th>
<th>BLEU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teacher latency</td>
<td>13539s</td>
<td>36.5</td>
</tr>
<tr>
<td>Teacher batched 64</td>
<td>1763s</td>
<td>36.5</td>
</tr>
<tr>
<td>Student latency</td>
<td>2784s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched 64</td>
<td>218s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched 64 shortlisted</td>
<td>220s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched 64 fp16</td>
<td>197s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched 64 fp16 + software optimisation</td>
<td>124s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched 1132 fp16 + software optimisation</td>
<td>36s</td>
<td>35.2</td>
</tr>
<tr>
<td>Student batched 1132 8-bit + software optimisation</td>
<td>40s</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Cloud cost to translate 1M characters.

- Google: $20
- Amazon: $15
- Microsoft: $10
- Efficient submissions: $0.001

Cloud MT providers running pretty hefty profit margins
Hardware Aware optimisation
Tune to your hardware

CPUs and GPUs have fundamentally different properties and optimising for them differs a bit.

GPUs:

- Don’t mind larger matrices all that much.
- Limited gains from quantisation
- Good for throughput, not so much for latency.
Tune to your hardware

CPUs and GPUs have fundamentally different properties and optimising for them differs a bit.

GPUs:

- Don’t mind larger matrices all that much.
- Limited gains from quantisation
- Good for throughput, not so much for latency.

CPUs:

- Really want smaller matrices.
- Huge gains from quantisation.
- Cache is extremely important.
- Good for latency, not so much for throughput.
- Cheaper to decode in most production cases than the GPU.
Figure 1: Taken from
Different memory models:
CPU is transparent
GPU memory management is very explicit.

<table>
<thead>
<tr>
<th>Memory type</th>
<th>Latency</th>
<th>Memory type</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>0</td>
<td>Register</td>
<td>0</td>
</tr>
<tr>
<td>L1 cache</td>
<td>4</td>
<td>Shared</td>
<td>4–8</td>
</tr>
<tr>
<td>L2 cache</td>
<td>10</td>
<td>Global GPU</td>
<td>200–800</td>
</tr>
<tr>
<td>L3 cache</td>
<td>40</td>
<td>CPU</td>
<td>10K+</td>
</tr>
<tr>
<td>Remote L3*</td>
<td>80</td>
<td>Remote GPU</td>
<td>22K+</td>
</tr>
<tr>
<td>DRAM</td>
<td>330+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CPU decoding

GPU Decoding
Running GPUs is expensive in terms of cloud credits.

- Batch translation
- Back/Forward translation
- Seldom used in production
GPUs care mostly about big matrices. Diminishing returns for smaller models.

- Shortlisting doesn’t help unless your vocabulary size is > 100000
- \(fp16\) decoding works well
- Quantisation to 8-bit doesn’t help in most cases
- Sparsity helps in limited cases.
CPU Decoding
Memory access

It’s all about memory, really.

Figure 2: Source: Andalam et al. (2013)
Memory access

It’s all about memory, really.

Modern systems have 40-80MB of L3 Cache. What is the most intensive part of decoding?

Figure 2: Source: Andalam et al. (2013)
Accessing L3 cache is 10X faster than accessing main memory. Idea: Fit the most computationally intensive parts in the cache.

- Decoder (Deep encoder - Shallow decoder/ tied decoder)
- Output Layer/Embeddings (Shortlisting techniques)

Further size reductions have diminishing returns when it comes to speed.
Quantisation

Why does quantisation help? It’s all about SIMD.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameters</th>
<th>Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>_mm256_fmadd_ps</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>_mm256_dpbusd_epi32</td>
<td>32</td>
<td>5</td>
</tr>
</tbody>
</table>

And of course MEMORY.
Quantisation complications #1

x86 has no $\text{INT}_8 \times \text{INT}_8$, unlike ARM. It only has $\text{UINT}_8 \times \text{INT}_8$. 
x86 has no $INT_8 \times INT_8$, unlike ARM. It only has $UINT_8 \times INT_8$.

- Shift the sign bit onto the parameter. Slow ;(
- Add 127 to the activations
Quantisation complications #1

x86 has no $INT_8 \times INT_8$, unlike ARM. It only has $UINT_8 \times INT_8$.

- Shift the sign bit onto the parameter. Slow ;(
- Add 127 to the activations

\[
\alpha = \max(|A|) \quad (1)
\]
\[
\beta = \max(|B|) \quad (2)
\]
\[
A_{fp32} \times B_{fp32} \approx \quad (3)
\]
\[
\frac{\alpha \beta}{127^2} \left( \frac{A \times 127}{\alpha} + [127] \right) \times \frac{B \times 127}{\beta} = \quad (4)
\]
\[
= \frac{\alpha \beta}{127^2} \left( \frac{AB \times 127^2}{\alpha \beta} + [127]B \times 127 \right) = \quad (5)
\]
\[
AB + \frac{[127]B \times \alpha}{127} = \quad (6)
\]
\[
= AB + [1]B \times \alpha \quad (7)
\]
Quantisation complications #2

Different architectures support a disjoint set of instructions

- ARM: $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$
- x86 non-server pre 2019: $\text{UINT}_8 \times \text{INT}_8 = \text{INT}_{16}$
- x86 server after 2019: $\text{UINT}_8 \times \text{INT}_8 = \text{INT}_{32}$
- x86 2023?: $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$

Library takes care of abstractions, but...
Different architectures support a disjoint set of instructions

- ARM: $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$
- x86 non-server pre 2019: $\text{UINT}_8 \times \text{INT}_8 = \text{INT}_{16}$
- x86 server after 2019: $\text{UINT}_8 \times \text{INT}_8 = \text{INT}_{32}$
- x86 2023?: $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$

Library takes care of abstractions, but... it doesn’t take advantage of streaming memory.

- Execute a single _mm256_dpbusd_epi32
- Apply de-quantisation
- Apply activation functions
- Then write to memory
Quantisation complications #2

Different architectures support a disjoint set of instructions

- ARM: $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$
- x86 non-server pre 2019: $\text{UINT}_8 \times \text{INT}_8 = \text{INT}_{16}$
- x86 server after 2019: $\text{UINT}_8 \times \text{INT}_8 = \text{INT}_{32}$
- x86 2023?: $\text{INT}_8 \times \text{INT}_8 = \text{INT}_{32}$

Library takes care of abstractions, but... it doesn’t take advantage of streaming memory.

- Execute a single _mm256_dpbusd_epi32
- Apply de-quantisation
- Apply activation functions
- Then write to memory

Existing libraries (oneDNN/MKL/FBGEMM) don’t quite do that (oneDNN almost does it). On the GPU side, nvidia’s CUTLASS does it. Hence, write your own GEMM implementation: https://github.com/kpu/intgemm
Pruning complications

Pruning can drastically decrease the number of parameters

• Up to 70% Sparsity with minimal loss of BLEU
• Hardware doesn’t like it as much
• Minimal loss of BLEU doesn’t mean minimal loss in quality
Fire up the profiler and see what doesn’t add up.
Fire up the profiler and see what doesn’t add up.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Loop@0x7f38980a01a8 in jit_avx512_core_amx_gemm_kern]</td>
<td>293.680011</td>
</tr>
<tr>
<td>[Loop@0x7f38980a21a8 in jit_avx512_core_amx_gemm_kern]</td>
<td>173.910006</td>
</tr>
<tr>
<td>[Loop at line 213 in marian::cpu::integer::PrepareBNodeOp<a href="">marian::Type</a>77825::&lt;forwardOps(void)::{lambda()#1}::opar]</td>
<td>139.320005</td>
</tr>
<tr>
<td>[Loop@0x7f38980a2324 in jit_avx512_core_amx_gemm_kern]</td>
<td>98.230004</td>
</tr>
<tr>
<td>[Loop@0x7f38980a0320 in jit_avx512_core_amx_gemm_kern]</td>
<td>95.200003</td>
</tr>
<tr>
<td><em>Z13_mm512_max_psDv16_fs</em></td>
<td>87.360003</td>
</tr>
<tr>
<td>[Loop@0x7f3897c550c0 in jit_avx512_core_amx_copy_kern]</td>
<td>70.960003</td>
</tr>
<tr>
<td>func@0x1a6884</td>
<td>61.020002</td>
</tr>
<tr>
<td>[Loop@0x7f389653a90 in jit_avx512_core_amx_copy_kern]</td>
<td>57.090002</td>
</tr>
<tr>
<td><em>Z13_mm512_mul_psDv16_fs</em></td>
<td>42.430002</td>
</tr>
<tr>
<td>[Loop at line 507 in dnn::impl::cpu::x64::gemm_kernel&lt;signed char, signed char, int&gt;]</td>
<td>41.670001</td>
</tr>
<tr>
<td>[Loop@0x7f3894d411c1 in inner_product_utils::jit_pp_kernel_t]</td>
<td>40.970001</td>
</tr>
<tr>
<td>[Loop@0x7f3894d811c1 in inner_product_utils::jit_pp_kernel_t]</td>
<td>40.950001</td>
</tr>
<tr>
<td>[Loop@0x7f3894f11c1 in inner_product_utils::jit_pp_kernel_t]</td>
<td>40.840001</td>
</tr>
<tr>
<td>[Loop@0x7f3894f411c1 in inner_product_utils::jit_pp_kernel_t]</td>
<td>40.640001</td>
</tr>
<tr>
<td>[Loop at line 224 in marian::cpu::Transpose0213&lt;(bool)0&gt;]</td>
<td>39.140001</td>
</tr>
<tr>
<td>func@0x7abc4</td>
<td>38.410001</td>
</tr>
<tr>
<td>[Loop@0x280d4e1 in [MKL BLAS]@avx512_xsgemv]</td>
<td>32.810001</td>
</tr>
<tr>
<td>std::partial_sort&lt;gnu_cxx::normal_iterator&lt;int*&gt;, std::vector&lt;int, std::allocator&lt;int&gt;&gt;, marian::NthElementCPU::getNBE&gt;</td>
<td>27.500001</td>
</tr>
<tr>
<td>[Loop at line 33 in marian::cpu::E&lt;{unsigned long}3&gt;::element&lt;{unsigned long}3, marian::functional::Assign&lt;marian::function::getNBE&lt;marian::cpu::E&lt;marian::cpu::NthElementCPU::getNBE&lt;marian::cpu::Transpose0213&lt;(bool)0&gt;)}}&gt;)]</td>
<td>27.240001</td>
</tr>
<tr>
<td>[Outside any known module]</td>
<td>26.790001</td>
</tr>
<tr>
<td><em>Z13_mm512_and_psDv16_fs</em></td>
<td>25.090001</td>
</tr>
</tbody>
</table>
Fire up the profiler and see what doesn’t add up.

<table>
<thead>
<tr>
<th>Function</th>
<th>CPU Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop@0x7f38980a01a8 in jit_avx512_core_amx_gemm_kern</td>
<td>293.680011</td>
</tr>
<tr>
<td>Loop@0x7f38980a21a8 in jit_avx512_core_amx_gemm_kern</td>
<td>179.910006</td>
</tr>
<tr>
<td><a href="mari">Loop at line 213 in marian::cpu::integer::PrepareBNodeOp<a href="">marian::Type77825</a>::forwardOps(void)::{lambda0}#1::operator+</a></td>
<td>139.320005</td>
</tr>
<tr>
<td>Loop@0x7f38980a324 in jit_avx512_core_amx_gemm_kern</td>
<td>98.230004</td>
</tr>
<tr>
<td>Loop@0x7f38980a0320 in jit_avx512_core_amx_gemm_kern</td>
<td>95.200003</td>
</tr>
<tr>
<td>_Z13_mm512_max_psDv16_fs</td>
<td>87.360003</td>
</tr>
<tr>
<td>Loop@0x7f3897c55c0 in jit_avx512_core_amx_copy_kern</td>
<td>70.960003</td>
</tr>
<tr>
<td>func@0x1a6884</td>
<td>61.020002</td>
</tr>
<tr>
<td>Loop@0x7f389653a090 in jit_avx512_core_amx_copy_kern</td>
<td>57.090002</td>
</tr>
<tr>
<td>_Z13_mm512_mul_psDv16_fs</td>
<td>42.430002</td>
</tr>
<tr>
<td><a href="marian">Loop at line 507 in dnn!::impl::cpu::x64::gemm_kernel&lt;signed char, signed char, int&gt;</a></td>
<td>41.670001</td>
</tr>
<tr>
<td>Loop@0x7f3894d411c1 in inner_product Utils::jit_pp_kernel_t</td>
<td>40.970001</td>
</tr>
<tr>
<td>Loop@0x7f3894d811c1 in inner_product Utils::jit_pp_kernel_t</td>
<td>40.950001</td>
</tr>
<tr>
<td>Loop@0x7f3894f811c1 in inner_product Utils::jit_pp_kernel_t</td>
<td>40.840001</td>
</tr>
<tr>
<td>Loop@0x7f3894f411c1 in inner_product Utils::jit_pp_kernel_t</td>
<td>40.640001</td>
</tr>
<tr>
<td>Loop at line 224 in marian::cpu::Transpose0213&lt;boo&gt;0&gt;</td>
<td>39.140001</td>
</tr>
<tr>
<td>func@0x7abc4</td>
<td>38.410001</td>
</tr>
<tr>
<td>Loop@0x280d4e1 in [MKL BLAS]@avx512_xsgemv</td>
<td>32.810001</td>
</tr>
<tr>
<td>std::partial_sort&lt; gnu_cxx::normal_iterator&lt;int*, std::vector&lt;int, std::allocator&lt;int&gt;&gt;&gt;, marian::NthElementCPU::getNBEϾched&lt;marian::fuc</td>
<td>27.500001</td>
</tr>
<tr>
<td>[Loop at line 33 in marian::cpu::E&lt;unsigned long3&gt;::element&lt;unsigned long3&gt;, marian::functional::Assign&lt;marian::functiona</td>
<td>27.240001</td>
</tr>
<tr>
<td>[Outside any known module]</td>
<td>26.790001</td>
</tr>
<tr>
<td>_Z13_mm512_and_psDv16_fs</td>
<td>25.090001</td>
</tr>
</tbody>
</table>

What is ntd_element doing there with beam size of 1?
Understand, Optimise, Overcome!

<table>
<thead>
<tr>
<th></th>
<th>GCC 11.2</th>
<th>clang 14</th>
<th>icc 2022</th>
</tr>
</thead>
<tbody>
<tr>
<td>std::max_element</td>
<td>2.6696s</td>
<td>0.4221s</td>
<td>0.4662s</td>
</tr>
<tr>
<td>sequential</td>
<td>1.0831s</td>
<td>1.1924s</td>
<td>1.1472s</td>
</tr>
<tr>
<td>AVX512 max + max_reduce</td>
<td>0.2412s</td>
<td>0.2152s</td>
<td>0.2142s</td>
</tr>
<tr>
<td>AVX512 max_reduce only</td>
<td>0.2570s</td>
<td>0.2629s</td>
<td>0.2325s</td>
</tr>
<tr>
<td>AVX512 cmp_ps_mask</td>
<td><strong>0.1884s</strong></td>
<td><strong>0.1826s</strong></td>
<td><strong>0.1833s</strong></td>
</tr>
<tr>
<td>AVX512 ^+ vectorized overhang</td>
<td>0.2097s</td>
<td>0.2089s</td>
<td>0.2072s</td>
</tr>
<tr>
<td>AVX cmp_ps + movemask</td>
<td>0.2181s</td>
<td>0.1697s</td>
<td>0.1702s</td>
</tr>
<tr>
<td>SSE cmplt_psp + movemask</td>
<td>0.2692s</td>
<td>0.2051s</td>
<td>0.2221s</td>
</tr>
</tbody>
</table>

**Table 1:** Performance of *max element* on various different compilers on Intel Cascade lake. For more information check [https://github.com/XapaJIaMnu/maxelem_test](https://github.com/XapaJIaMnu/maxelem_test).
Quality
Shortlisting

• IBM models don’t capture idioms
• KNN shortlisting requires specific model configuration
Quantisation drops quality

- Some can be recovered with fine tuning
- Combining different methods together further drops quality
Use multiple metrics.

- Monitor changes in BLEU and COMET
- Beware of sudden drops in either
Use multiple metrics.

- Monitor changes in BLEU and COMET
- Beware of sudden drops in either

<table>
<thead>
<tr>
<th></th>
<th>BLEU</th>
<th>COMET</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline:</td>
<td>27.5</td>
<td>0.45</td>
<td>35s</td>
</tr>
<tr>
<td>baseline + A</td>
<td>27.4</td>
<td>0.44</td>
<td>18s</td>
</tr>
<tr>
<td>baseline + B</td>
<td>27.2</td>
<td>0.43</td>
<td>15s</td>
</tr>
<tr>
<td>baseline + C</td>
<td>27.2</td>
<td>0.37</td>
<td>10s</td>
</tr>
</tbody>
</table>
Use multiple metrics.

- Monitor changes in BLEU and COMET
- Beware of sudden drops in either

<table>
<thead>
<tr>
<th></th>
<th>BLEU</th>
<th>COMET</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline:</td>
<td>27.5</td>
<td>0.45</td>
<td>35s</td>
</tr>
<tr>
<td>baseline + A</td>
<td>27.4</td>
<td>0.44</td>
<td>18s</td>
</tr>
<tr>
<td>baseline + B</td>
<td>27.2</td>
<td>0.43</td>
<td>15s</td>
</tr>
<tr>
<td>baseline + C</td>
<td>27.2</td>
<td>0.37</td>
<td>10s</td>
</tr>
</tbody>
</table>

- Something is wrong, look at our data!

*In der Nähe der Nähe der Nähe der Nähe*
MT in your pocket

Let’s see, what did we do?
MT in your pocket

Let’s see, what did we do?

https://translatelocally.com
https://translatelocally.com/web/
Alternatives
What else is out there

Many methods exist

- IBdecoder (Zhang et al, 2020)
- Non-autoregressive MT (Choose your pickings)
- Semi-non autoregressive
- Something else?
What else is out there

Many methods exist

• IBdecoder (Zhang et al, 2020)
• Non-autoregressive MT (Choose your pickings)
• Semi-non autoregressive
• Something else?

Student transformers still work the best
What else is out there

Many methods exist

- IBdecoder (Zhang et al, 2020)
- Non-autoregressive MT (Choose your pickings)
- Semi-non autoregressive
- Something else?

Student transformers still work the best

Thank you for your time
• Sequence-Level Knowledge Distillation by Kim and Rush, 2016
• Deep Encoders and Shallow decoders by Kasai et al, 2020
• KNN shortlisting by Shi et al, 2018
• Pruning Behnke and Heafield, 2021
• IBdecoder (Zhang et al, 2020)
• Tutorial on efficient MT: https://nbogoychev.com/efficient-machine-translation/