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

Nikolay Bogoychev

University of Edinburgh
N.Bogoych@ed.ac.uk

Machine translation is heavy

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?

Machine translation is heavy

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?

Model size

- How do we speed up the machine translation?
- It’s simple, just use smaller models.

Model size

- 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!

Computational cost

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.

Computational cost

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.

Computational cost

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.

Deep encoder shallow decoder

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.

Decoders

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.

Matrix sizes

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.

Pruning

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

Output layer

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

- Reduce vocabulary size. Bad

Output layer

The output layer matrix has size $DIM_{emb} * |N|_{vocab}$ 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.

Batching

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 $INT_8 \times INT_8 = INT_{32}$.
  (Not really true for a lot of the hardware)

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.
  $A_i = \frac{127 * A_i}{MAX(|A|)}$, $B_i = \frac{127 * B_i}{MAX(|B|)}$

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.
  $A_i = \frac{127 * A_i}{MAX(|A|)}$, $B_i = \frac{127 * B_i}{MAX(|B|)}$
- After multiplication, perform de-quantisation and re-quantisation for the next activation:

$$A_{fp32} * B_{fp32} \approx A_{int8} * B_{int8} * \frac{MAX(A) * MAX(B)}{127^2}$$

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 $INT_8 \times INT_8 = INT_{32}$.
  (Not really true for a lot of the hardware)
- Quantise Activation and parameter matrices to 8-bit.
  $A_i = \frac{127 * A_i}{MAX(|A|)}$, $B_i = \frac{127 * B_i}{MAX(|B|)}$
- After multiplication, perform de-quantisation and re-quantisation for the next activation:

$$A_{fp32} * B_{fp32} \approx A_{int8} * B_{int8} * \frac{MAX(A) * MAX(B)}{127^2}$$

- Parameters are converted to 8-bit in advance, activations at runtime, always before multiplication.
- Quantisation Multipliers are computed in advance.

Speed results

CPU

Applying out full bag of tricks:

CPU, 16 threads, 3000 SENTENCES	TIME	BLEU
Teacher latency	4597s	36.5
Teacher batched	652s	36.5
Student latency	84s	35.2
Student batched	11s	35.2
Student batched shortlisted	8s	35.2
Student batched quantised shortlisted	7.1s	35.0

CPU

Applying out full bag of tricks:

CPU, 16 threads, 3000 SENTENCES	TIME	BLEU
Teacher latency	4597s	36.5
Teacher batched	652s	36.5
Student latency	84s	35.2
Student batched	11s	35.2
Student batched shortlisted	8s	35.2
Student batched quantised shortlisted	7.1s	35.0

1 Thread, to make it more granular

CPU, 1 thread, 3000 SENTENCES	TIME	BLEU
Student latency	189s	35.2
Student batched	38s	35.2
Student batched shortlisted	27s	35.2
Student batched quantised shortlisted	21s	35.0

GPU

Applying out full bag of tricks for the GPU... Maybe it's better to not use more all tricks.

GPU, 100k SENTENCES	TIME	BLEU
Teacher latency	13539s	36.5
Teacher batched 64	1763s	36.5
Student latency	2784s	35.2
Student batched 64	218s	35.2
Student batched 64 shortlisted	220s	35.2
Student batched 64 fp16	197s	35.2
Student batched 64 fp16 + software optimisation	124s	35.2
Student batched 1132 fp16 + software optimisation	36s	35.2
Student batched 1132 8-bit + software optimisation	40s	35.0

Cost

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.

GPU vs CPU

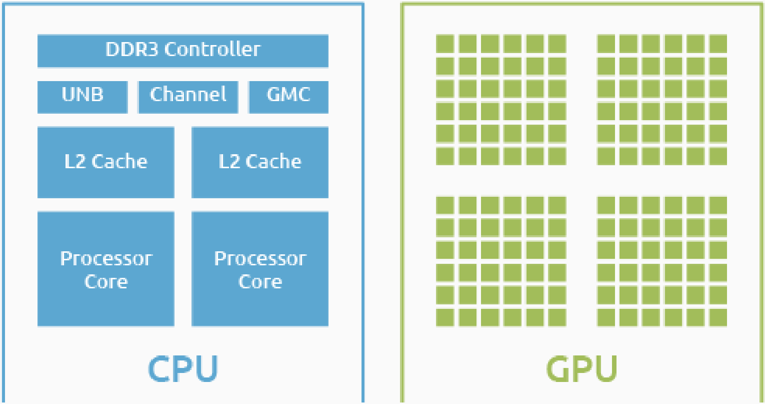

Figure 1: Taken from *https://www.adlinktech.com/en/gpu-computing*

Memory Latency

Different memory models:
CPU is transparent
GPU memory management is very explicit.

CPU		GPU	
Memory type	Latency	Memory type	Latency
Register	0	Register	0
L1 cache	4	Shared	4–8
L2 cache	10	Global GPU	200–800
L3 cache	40	CPU	10K+
Remote L3*	80	Remote GPU	22K+
DRAM	330+		

GPU Decoding

Purpose

Running GPUs is expensive in terms of cloud credits.

- Batch translation
- Back/Forward translation
- Seldom used in production

Optimising for GPU

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.

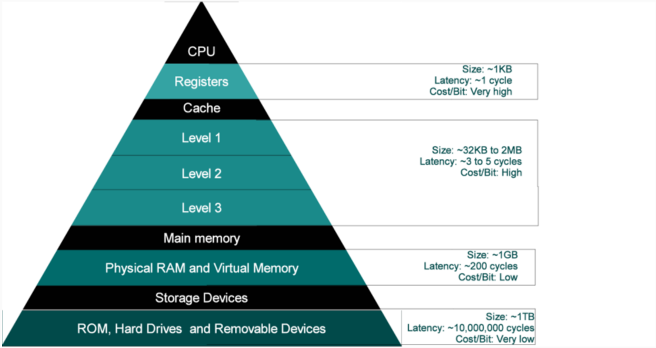

Figure 2: Source: Source: Andalam et al. (2013)

Memory access

It's all about memory, really.

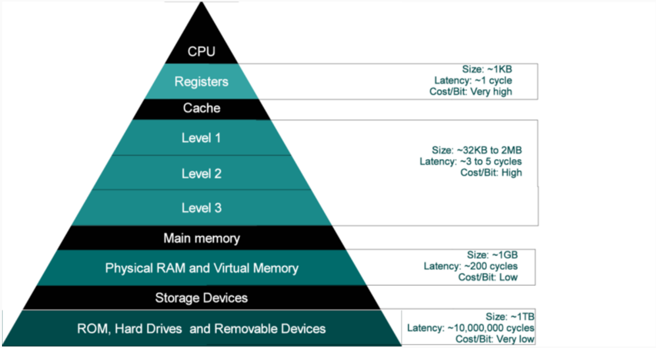

Figure 2: Source: Source: Andalam et al. (2013)

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

Crucial bits

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.

Instruction	Paramters	Cycles
_mm256_fmadd_ps	8	4
_mm256_dpbusd_epi32	32	5

And of course MEMORY.

Quantisation complications #1

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

Quantisation complications #1

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

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

Quantisation complications #1

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

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

$$\alpha = max(|A|) \tag{1}$$

$$\beta = max(|B|) \tag{2}$$

$$A_{fp32} * B_{fp32} \approx \tag{3}$$

$$\frac{\alpha\beta}{127^2}(\frac{A * 127}{\alpha} + [127]) * \frac{B * 127}{\beta} = \tag{4}$$

$$= \frac{\alpha\beta}{127^2}(\frac{AB * 127^2}{\alpha\beta} + \frac{[127]B * 127}{\beta}) = \tag{5}$$

$$= AB + \frac{[127]B * \alpha}{127} = \tag{6}$$

$$= AB + [1]B * \alpha \tag{7}$$

Quantisation complications #2

Different architectures support a disjoint set of instructions

- ARM: $INT_8 * INT_8 = INT_{32}$
- x86 non-server pre 2019: $UINT_8 * INT_8 = INT_{16}$
- x86 server after 2019: $UINT_8 * INT_8 = INT_{32}$
- x86 2023?: $INT_8 * INT_8 = INT_{32}$

Library takes care of abstractions, but...

Quantisation complications #2

Different architectures support a disjoint set of instructions

- ARM: $INT_8 * INT_8 = INT_{32}$
- x86 non-server pre 2019: $UINT_8 * INT_8 = INT_{16}$
- x86 server after 2019: $UINT_8 * INT_8 = INT_{32}$
- x86 2023?: $INT_8 * INT_8 = INT_{32}$

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: $INT_8 * INT_8 = INT_{32}$
- x86 non-server pre 2019: $UINT_8 * INT_8 = INT_{16}$
- x86 server after 2019: $UINT_8 * INT_8 = INT_{32}$
- x86 2023?: $INT_8 * INT_8 = INT_{32}$

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

Profiling

Profiling

Fire up the profiler and see what doesn't add up.

Profiling

Fire up the profiler and see what doesn't add up.

	A	B
1	Function	CPU Time
2	[Loop@0x7f38980a01a8 in jit_avx512_core_amx_gemm_kern]	293.680011
3	[Loop@0x7f38980a21a8 in jit_avx512_core_amx_gemm_kern]	173.910006
4	[Loop at line 213 in marian::cpu::integer::PrepareBNodeOp<(marian::Type)77825>::forwardOps(void)::{lambda()#1}::operat	139.320005
5	[Loop@0x7f38980a2324 in jit_avx512_core_amx_gemm_kern]	98.230004
6	[Loop@0x7f38980a0320 in jit_avx512_core_amx_gemm_kern]	95.200003
7	_Z13_mm512_max_psDv16_fS_	87.360003
8	[Loop@0x7f3897c550c0 in jit_avx512_core_amx_copy_kern]	70.960003
9	func@0x1a6884	61.020002
10	[Loop@0x7f389653a090 in jit_avx512_core_amx_copy_kern]	57.090002
11	_Z13_mm512_mul_psDv16_fS_	42.430002
12	[Loop at line 507 in dnnl::impl::cpu::x64::gemm_kernel<signed char, signed char, int>]	41.670001
13	[Loop@0x7f3894d411c1 in inner_product_utils::jit_pp_kernel_t]	40.970001
14	[Loop@0x7f3894d811c1 in inner_product_utils::jit_pp_kernel_t]	40.950001
15	[Loop@0x7f3894f811c1 in inner_product_utils::jit_pp_kernel_t]	40.840001
16	[Loop@0x7f3894f411c1 in inner_product_utils::jit_pp_kernel_t]	40.640001
17	[Loop at line 224 in marian::cpu::Transpose0213<(bool)0>]	39.140001
18	func@0x7abc4	38.410001
19	[Loop@0x280d4e1 in [MKL BLAS]@avx512_xsgemv]	32.810001
20	std::partial_sort<__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int>>>, marian::NthElementCPU::getNBe	27.500001
21	[Loop at line 33 in marian::cpu::E<(unsigned long)3>::element<(unsigned long)3, marian::functional::Assign<marian::functio	27.240001
22	[Outside any known module]	26.790001
23	_Z13_mm512_and_psDv16_fS_	25.090001

Profiling

Fire up the profiler and see what doesn't add up.

	A	B
1	Function	CPU Time
2	[Loop@0x7f38980a01a8 in jit_avx512_core_amx_gemm_kern]	293.680011
3	[Loop@0x7f38980a21a8 in jit_avx512_core_amx_gemm_kern]	173.910006
4	[Loop at line 213 in marian::cpu::integer::PrepareBNodeOp<(marian::Type)77825>::forwardOps(void)::{lambda()#1}::operat	139.320005
5	[Loop@0x7f38980a2324 in jit_avx512_core_amx_gemm_kern]	98.230004
6	[Loop@0x7f38980a0320 in jit_avx512_core_amx_gemm_kern]	95.200003
7	_Z13_mm512_max_psDv16_fS_	87.360003
8	[Loop@0x7f3897c550c0 in jit_avx512_core_amx_copy_kern]	70.960003
9	func@0x1a6884	61.020002
10	[Loop@0x7f389653a090 in jit_avx512_core_amx_copy_kern]	57.090002
11	_Z13_mm512_mul_psDv16_fS_	42.430002
12	[Loop at line 507 in dnnl::impl::cpu::x64::gemm_kernel<signed char, signed char, int>]	41.670001
13	[Loop@0x7f3894d411c1 in inner_product_utils::jit_pp_kernel_t]	40.970001
14	[Loop@0x7f3894d811c1 in inner_product_utils::jit_pp_kernel_t]	40.950001
15	[Loop@0x7f3894f811c1 in inner_product_utils::jit_pp_kernel_t]	40.840001
16	[Loop@0x7f3894f411c1 in inner_product_utils::jit_pp_kernel_t]	40.640001
17	[Loop at line 224 in marian::cpu::Transpose0213<(bool)0>]	39.140001
18	func@0x7abc4	38.410001
19	[Loop@0x280d4e1 in [MKL BLAS]@avx512_xsgemv]	32.810001
20	std::partial_sort<__gnu_cxx::__normal_iterator<int*, std::vector<int, std::allocator<int>>>, marian::NthElementCPU::getNBe	27.500001
21	[Loop at line 33 in marian::cpu::E<(unsigned long)3>::element<(unsigned long)3, marian::functional::Assign<marian::functio	27.240001
22	[Outside any known module]	26.790001
23	_Z13_mm512_and_psDv16_fS_	25.090001

What is ntd_element doing there with beam size of 1?

Max element

Understand, Optimise, Overcome!

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

Table 1: Performance of *max element* on various different compilers on Intel Cascade lake. For more information check `https://github.com/XapaJIaMnu/maxelem_test`.

Quality

Shortlisting

- IBM models don't capture idioms
- KNN shortlisting requires specific model configuration

Quantisation

Quantisation drops quality

- Some can be recovered with fine tuning
- Combining different methods together further drops quality

Evaluation

Use multiple metrics.

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

Evaluation

Use multiple metrics.

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

	BLEU	COMET	time
baseline:	27.5	0.45	35s
baseline + A	27.4	0.44	18s
baseline + B	27.2	0.43	15s
baseline + C	27.2	0.37	10s

Evaluation

Use multiple metrics.

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

	BLEU	COMET	time
baseline:	27.5	0.45	35s
baseline + A	27.4	0.44	18s
baseline + B	27.2	0.43	15s
baseline + C	27.2	0.37	10s

- 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

References

- 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
- On Efficiency shared task: Kim et al, 2019, Bogoychev et al, 2020, Behnke et al, 2021, Heafield et al 2019-2021
- IBdecoder (Zhang et al, 2020)
- Tutorial on efficient MT: *https://nbogoychev.com/efficient-machine-translation/*