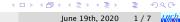
fast exp and tanh for simpleNN

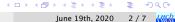
Manuel Schiller

University of Glasgow

June 19th, 2020



- last time, introduced simpleNN for fast NN evaluations
- today:
 - after (auto-)vectorizing matrix multiplication, it's time to turn to activation functions which contain $\exp(x)$ and $\tanh(x)$
 - micro-bechmark time and accuracy behaviour for long vectors of floating point numbers



benchmarking method

- time benchmarking:
 - time the following code fragment for cacheline-aligned source and destination

```
template <typename T, typename FN>
[[gnu::noinline]] void bench(const T * first, const T * last, T * dest, FN&& fn) noexcept
{
    for (; last != first; **first, **dest) *dest = fn(*first);
}
```

- time from std::chrono::high_resolution_clock::now()
- array size 65536, take minimum time of 128 trials
- subtract time it takes to copy source to destination
- accuracy benchmarking
 - ULP (Unit in the Last Place): x = 1.mmm...mmmu $\times 2^{\text{exp}}$ (where m, ..., u: mantissa bits)
- compiler flags: -std=c++14 -02 -march=native -ftree-vectorize
 (for Raspberry Pi 3B: -std=c++14 -02 -march=armv8-a -mfloat-abi=hard
 -mfpu=neon-fp-armv8)



computation method

generalities:

- to vectorize well, control flow must not depend on input data
- therefore: branchless if, abs, copysign, ...

```
// almost (cond ? a : b)
float sel(bool cond, float a, float b)
{
  int32_t mask = -int32_t(cond);
  return bit_cast<float>((bit_cast<int32_t>(x) & mask) | (bit_cast<int32_t>(b) & ~mask))
}
```

- $= \exp(x)$
 - use $\exp(x) = \exp(n/\ln(2) + r) = 2^n P(r)/Q(r)$
- \blacksquare tanh(x)
 - use x' = x/8 to reduce argument magnitude
 - use $tanh(x') \sim P(x')/Q(x')$
 - use argument doubling formula $tanh(2x) = \frac{2 \tanh(x)}{1 + (\tanh(x))^2}$ three times to undo argument reduction



results for float

time:

[ns]	Core i7-2640	Core i7-2640	Ryzen 7 2700U	Ryzen 7 2700U	ARM v8a (BCM2837)	ARM v8a (BCM2837)
	gcc 10	clang 10	gcc 8	clang 10	gcc 8	clang 9
std::exp	6.84	6.87	2.41	2.39	49.42	50.08
vdt::fast_exp	10.55	2.48	4.14	1.09	77.22	76.65
my::exp	2.16	1.47	1.25	0.96	76.37	70.77
std::tanh	30.34	30.35	15.97	15.72	185.02	184.74
vdt::fast_tanh	22.77	5.30	4.16	1.14	83.07	83.29
my::tanh	5.44	5.31	1.07	0.84	83.07	79.99

accuracy (compare against std::..., argument uniformly distributed over range for which finite result is expected)

[ULP]	min.	max.	RMS
vdt::fast_exp	-1	4.9×10 ⁶	185
my::exp	-2	2	0.35
vdt::fast_tanh	-6	6	0.83
my::tanh	-6	7	0.83

- VDT uses branchy version of my tanh code
- vdt::exp has an overflow problem for large x where it gets quite inaccurate
- need to understand what's happening with exp on ARM...



results for double

time:

[ns]	Core i7-2640	Core i7-2640	Ryzen 7 2700U	Ryzen 7 2700U	ARM v8a (BCM2837)	ARM v8a (BCM2837)
	gcc 10	clang 10	gcc 8	clang 10	gcc 8	clang 9
std::exp	15.87	15.91	8.58	8.22	94.77	93.25
vdt::fast_exp	11.66	4.42	5.13	3.25	99.29	85.50
my::exp	5.91	4.71	3.46	2.83	100.98	85.35
std::tanh	32.47	32.38	16.99	16.83	209.20	208.29
vdt::fast_tanh	35.22	16.98	5.86	3.21	134.38	128.81
my::tanh	17.27	16.99	3.16	2.70	137.79	125.30

accuracy (compare against std::..., argument uniformly distributed over range for which finite result is expected)

[ULP]	min. max.		RMS	
vdt::fast_exp	-6.7×10 ¹⁵	3.0×10 ¹⁶	0.37	
my::exp	-2	2	0.37	
vdt::fast_tanh	-6	6	0.60	
my::tanh	-6	6	0.60	

- VDT uses branchy version of my tanh code
- vdt::exp has an overflow problem for large x where it gets quite inaccurate
- need to understand what's happening with exp on ARM...



next steps

- nice speedup possible for activation functions (although still need to understand some things)
- put this into simpleNN, and measure impact
- maybe also look into Intel/AMD performance (math) libraries (but not portable)
- clean up code, and put onto gitlab
- possibly: feed that back into some library

