Efficient number theoretic transform

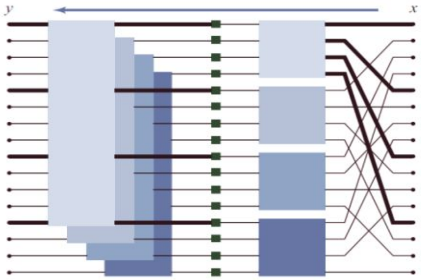
Rudolf Loretan
Artur Melo
Rijad Nuridini
Philippe Panhaleux

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

The NTT algorithm

- **Number-Theoretic Transform (NTT) is a specialization of the DFT**
  - NTT is over ring F = $\mathbb{Z}_P$
  - Replacing $e^{(-2\pi ik)/N}$ with n-th primitive root in matrix.
- **Numeric problem: input: v_1, output: v_2** $\quad v_1, v_2 \in \mathbb{Z}_P^n$
- **Drawback: Same complexity as DFT**
  - Naïve implementation: $O(n^2)$
- **Advantage: DFT is a well known and optimized algorithm: O(n log n)**
  - We can use radix algorithms:

$$\mathrm{DFT}_{16} = \mathrm{DFT}_4 \otimes I_4 \quad T_4^{16} \quad I_4 \otimes \mathrm{DFT}_4 \quad L_4^{16}$$



- **We focused on 2 variants: Radix 2 iterative, Radix 4 recursive**

Design Decisions and Cost Analysis

- **NTT for input sizes 2^N**
- **Work on $\mathbb{Z}_p$ ring of integers modulo p**
  - Chose p ≈ 2^{25} → input numbers must be ≤ 2^{25}
  - Reason: Multiplication can be done in a double w/o overflow
- **Benchmark alternative: FTL-NTTW Library**
- **Cost analysis: int adds, mults, divs, mods - but no index access operations**

$$C_{twiddle} = 25C_{add} + 6C_{mul} + 5C_{mod} \qquad C_{base} = 24C_{add} + 1C_{mul} + 1C_{mod}$$

$$C_{radix4-rec}(n) = \begin{cases} \frac{n}{2}(6C_{add} + 1C_{mul} + 1C_{mod} + (log_4(n) - 1)C_{twiddle} + \frac{(C_{base}+1C_{div})}{2}) & \text{if } 4 \mid n \\ 2C_{radix4-rec}(\frac{n}{2}) + \frac{n}{2}(1C_{mul} + 1C_{mod} + 6C_{add})) & \text{otherwise} \end{cases}$$

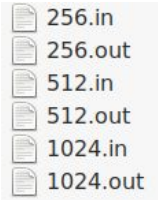
- **Verified using counters embedded in code**

Experimental Setup

- **Intel Core i7 7700 @ 3.6GHz - Skylake Architecture**
  - L2 cache: 256KB
  - L3 cache: 8MB
- **Compiled with g++ 7.5.0**
- **Benchmark: tsc_x86 hardware counters as in homeworks**
  - Performance measurements for warm cache
- **Validation:**
  - Python NTT library:

```
from sympy import ntt
seq = [153, 321, 133, 44]
transform = ntt(seq, prime_no)
```

→



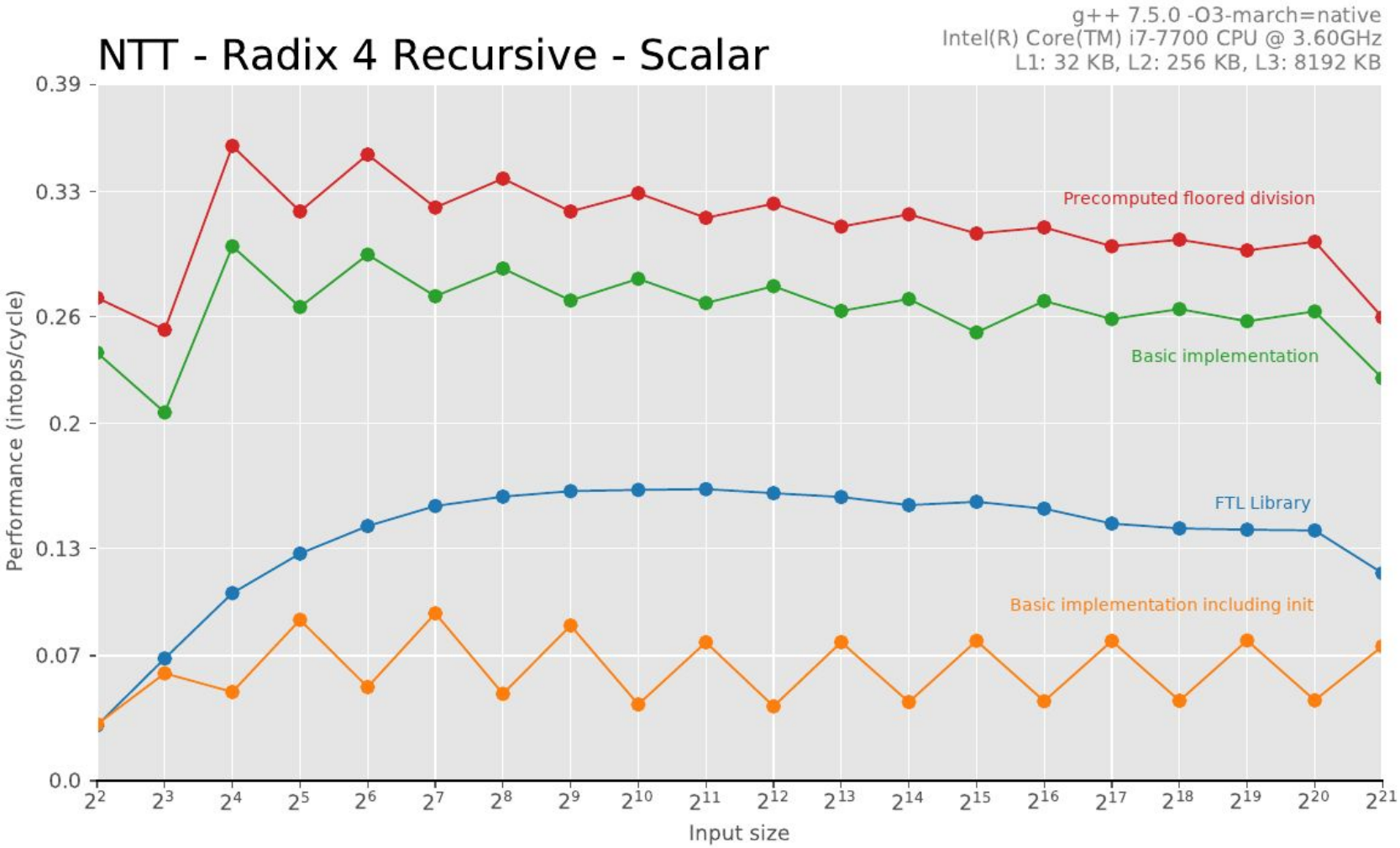
… 2m

- **Profiler:** Intel VTune

Scalar improvements

$$r = a - n\left\lfloor \frac{a}{n} \right\rfloor$$

- We tried replacing divs with shifts, floored division



Profiling

Intel VTune

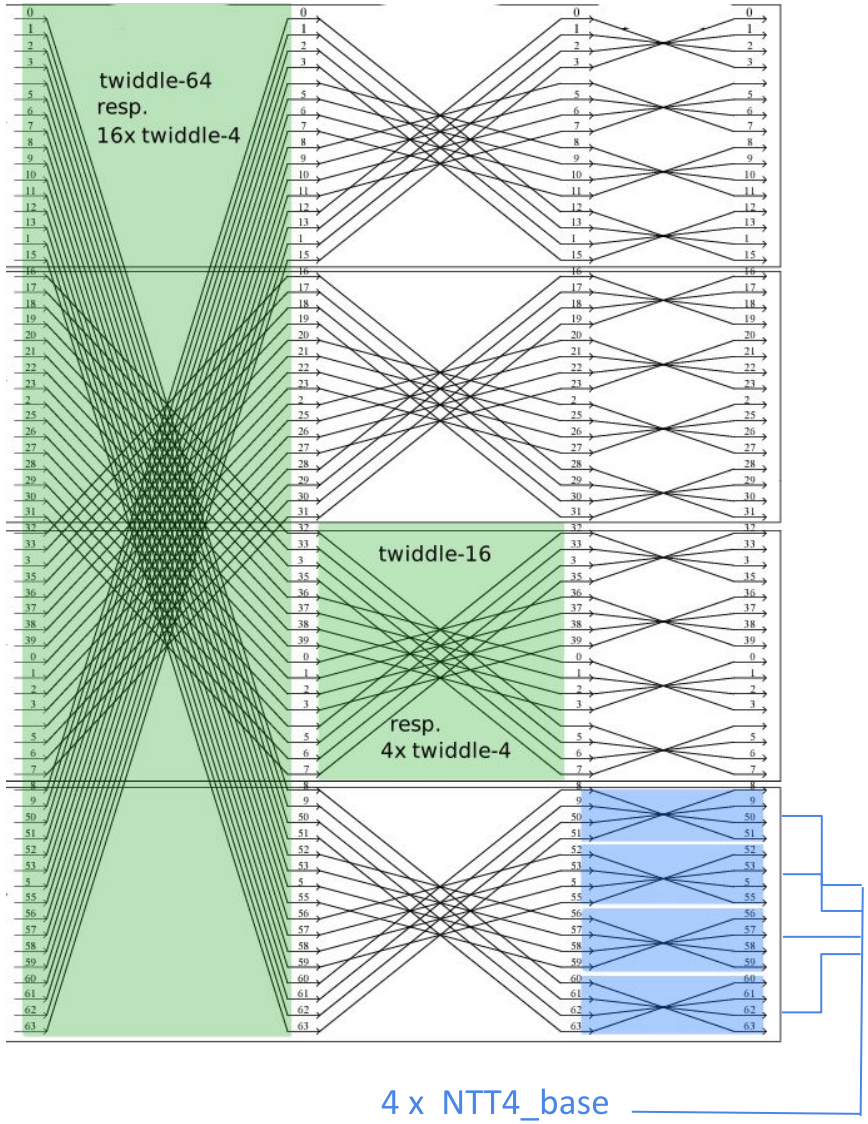
Top Hotspots

This section lists the most active functions in yo
improving overall application performance.

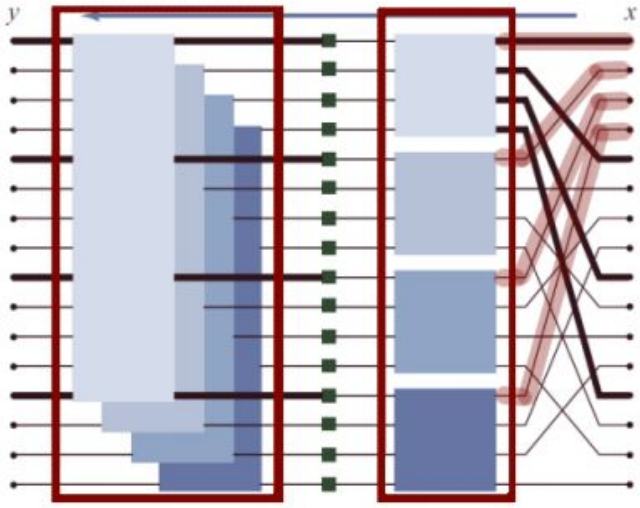
Function	Module	CPU Time
NTT4_twiddle	a.out	45.181s
NTT4_base	a.out	2.510s
do_NTT_radix4	a.out	2.220s
NTT_rec	a.out	0.973s
modpow	a.out	0.612s
[Others]	N/A*	0.204s

*N/A is applied to non-summable metrics.

- **Problems:**
  - Loading data with strides
  - Many overflow checks (before every single add/mult)
- **How can we improve this?**



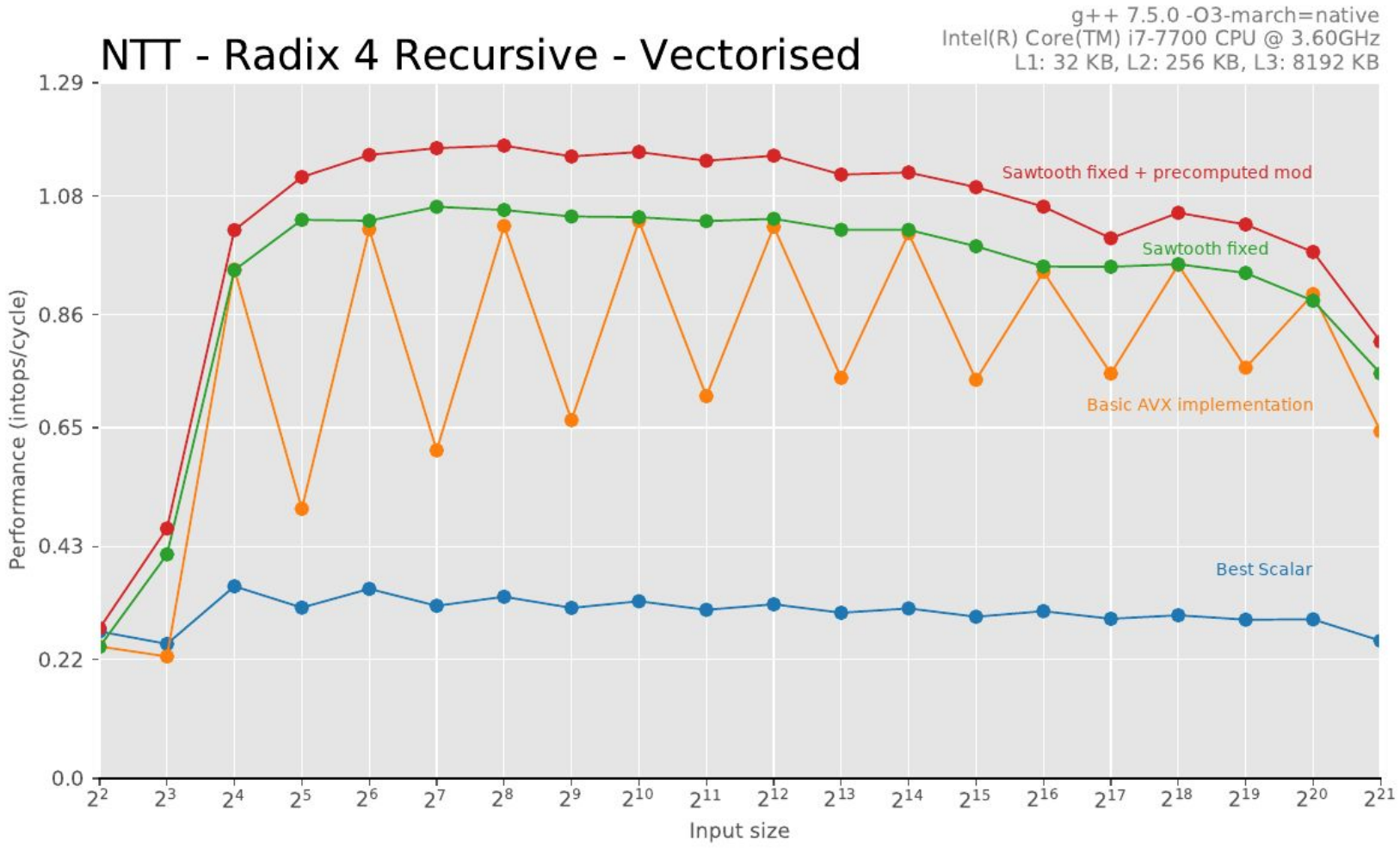
Vectorization – Radix 4



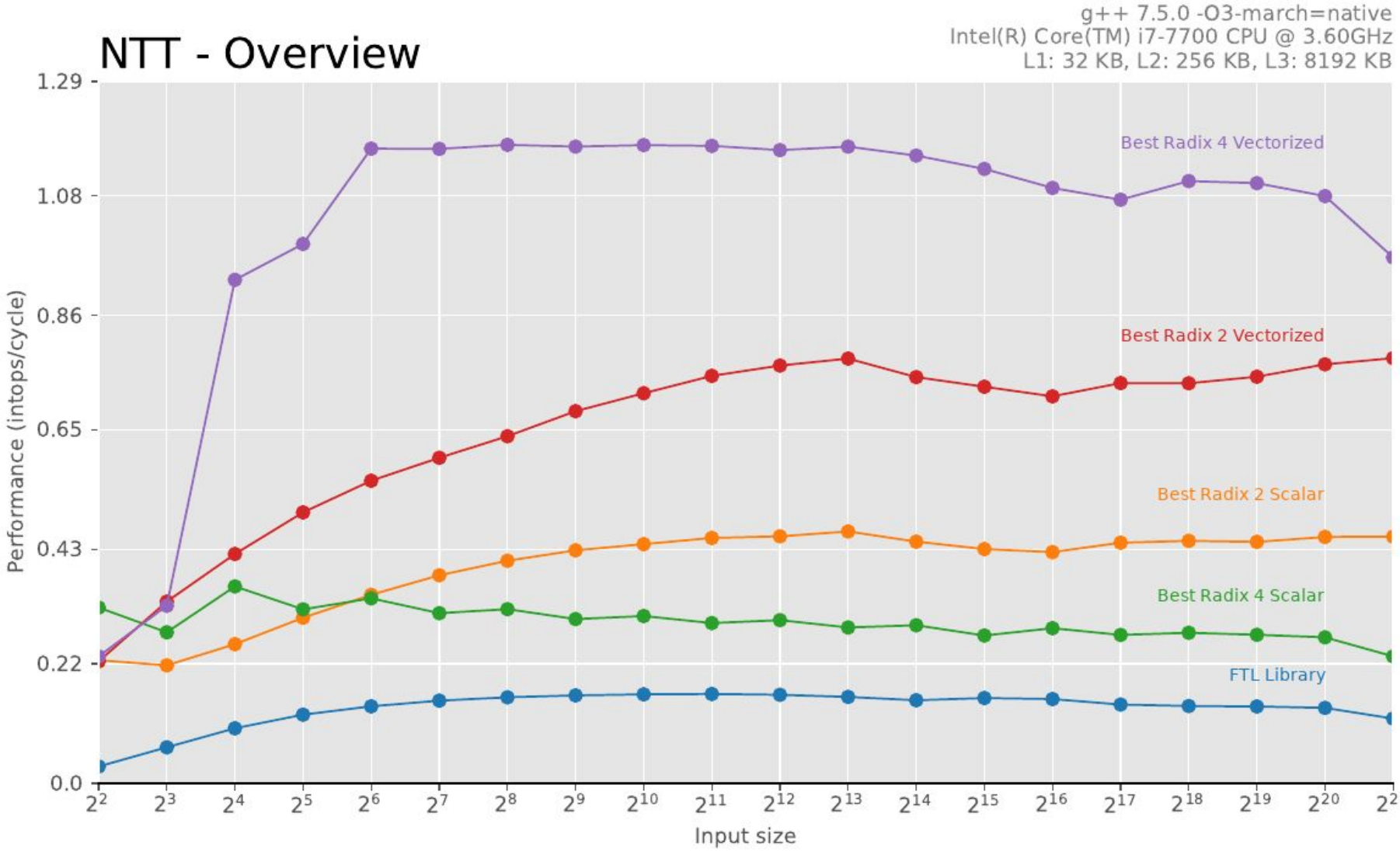
- **Can't vectorize single base4/twiddle4**
- **But: can do 4 at a time**
  - Which is pretty much base16/twiddle16
- **Problem: Integer Intrinsics in AVX2**
  - Extremely small variety of instructions e.g.
    - *multiplication only for epi32 → 16 bit numbers or overflow*
    - *only two comparison instr: (eq and gt)*
    - *Not even plain load/store*
- **Would greatly benefit from AVX-512**

Vectorization – Radix 4

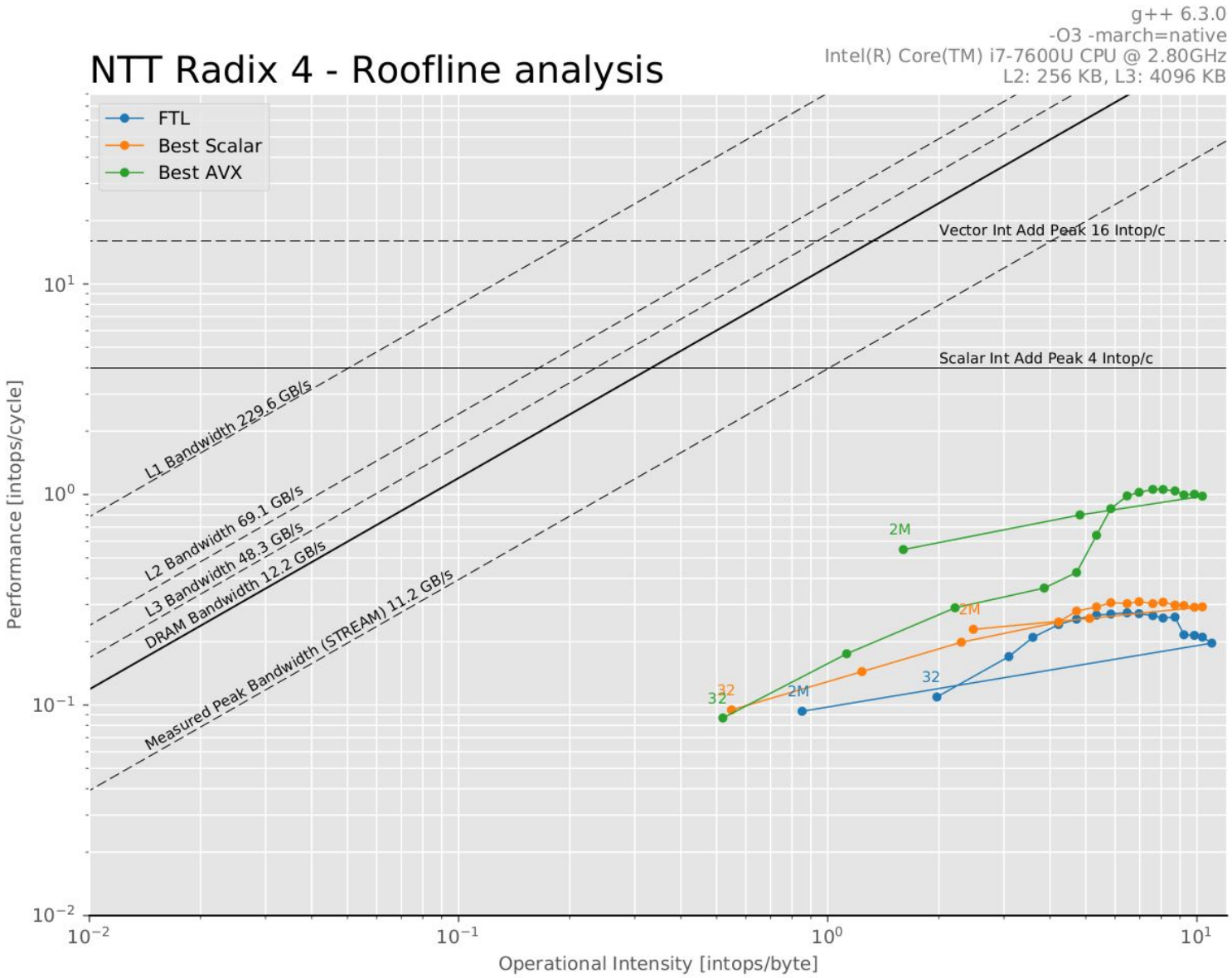
- **Speedup: 7.3x over FTL, 16.5x over scalar base**



Best Implementations



Memory and roofline analysis



Potential future work

- **Maybe we will...**
  - … adapt NFLib to compare our implementation, potentially best library out there
  - … compare AMD vs Intel
    - *Artur has an AMD Ryzen 5 processor - the charts seem different*

- **It could have been interesting, but we won't...**
  - … use code generation for radix algorithms of even higher order (8, 16)
  - … use AVX-512 for a more complete set of int ops
    - *Big potential improvement in speed and code simplicity*