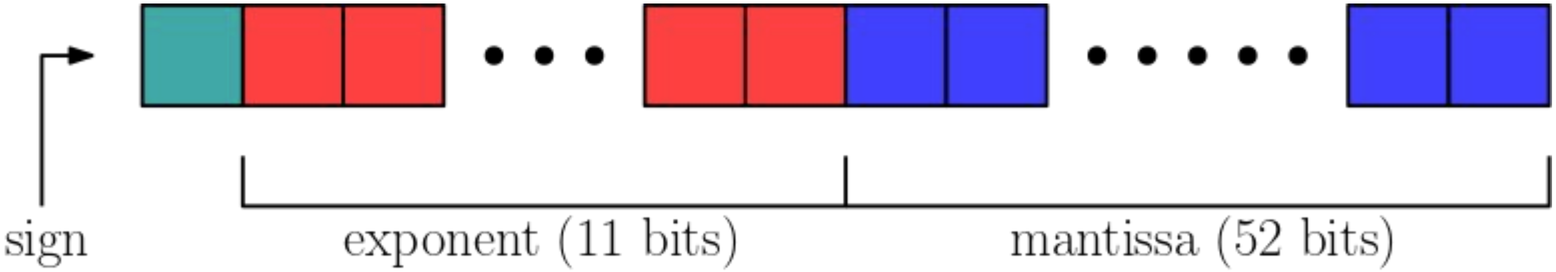
Floating-point expansions

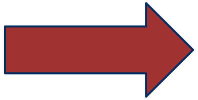
Alec Pauli
Andreas Geiszl
Luca Pinter
Tom Wartmann

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Introduction

double-precision floating-point



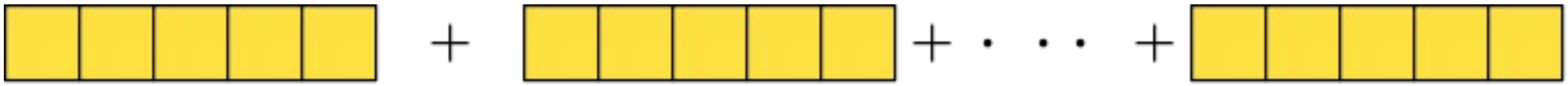


52 bits of precision

Introduction

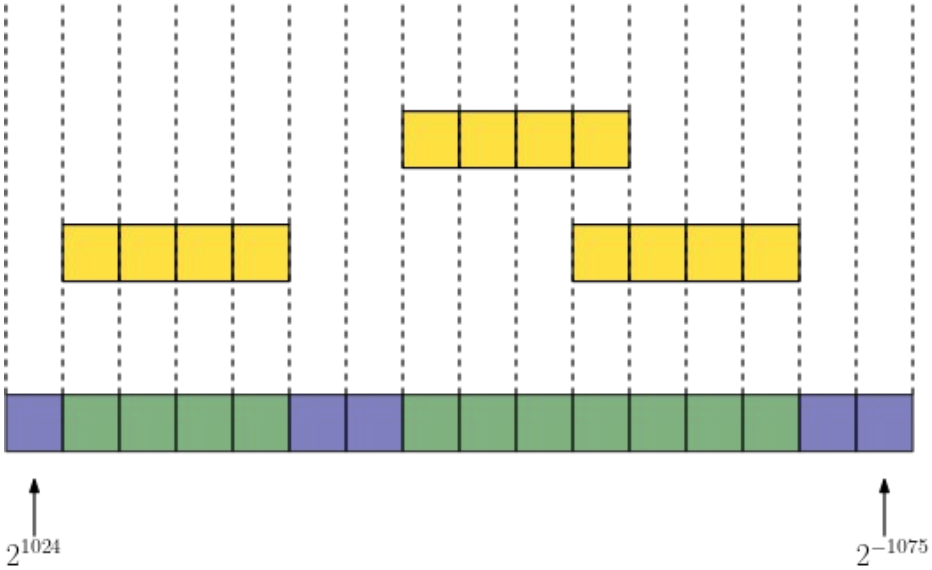
how do we increase precision?

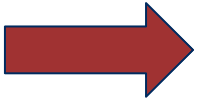
represent numbers as unevaluated sums of double-precision floating-point numbers



Introduction

we use whole range of doubles as precision



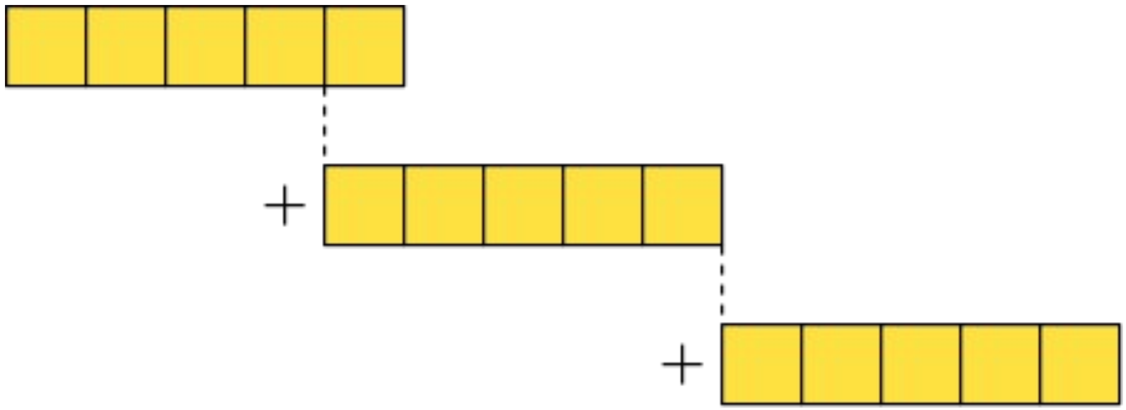


over 2000 bits of precision!

Introduction

Non-overlapping expansions

Definition 2.6. An expansion $u_0, u_1, \ldots, u_{n-1}$ is ulp-*nonoverlapping* if for all $0 < i < n$, we have $|u_i| \leq \mathrm{ulp}(u_{i-1})$.



Cost analysis

- **Cost measure: floating point operations (flops)**
  - ldexp & frexp counted as 1 flop
- **Timing: RDTSC, chrono, QueryPerformanceCounter**

Experimental setup

- **Compiler: MSVC cl.exe on Windows**
  - Also compared to g++ on Linux

- **Optimizations: everything except compiler vectorization; fast floating point operations turned off**
  - **MSVC** flags: /O2 /GL
  - **g++** flags: -O3 -fno-tree-vectorize -ffp-contract=off -march=native

Addition - baseline

```c
void addition(double *a, double *b, double *s, int length_a, int length_b, int length_result)
{
    double *tmp = (double *)alloca((length_a + length_b) * sizeof(double));
    merge(a, b, tmp, length_a, length_b);
    renormalizationalgorithm(tmp, length_a + length_b, s, length_result);

    return;
}
```

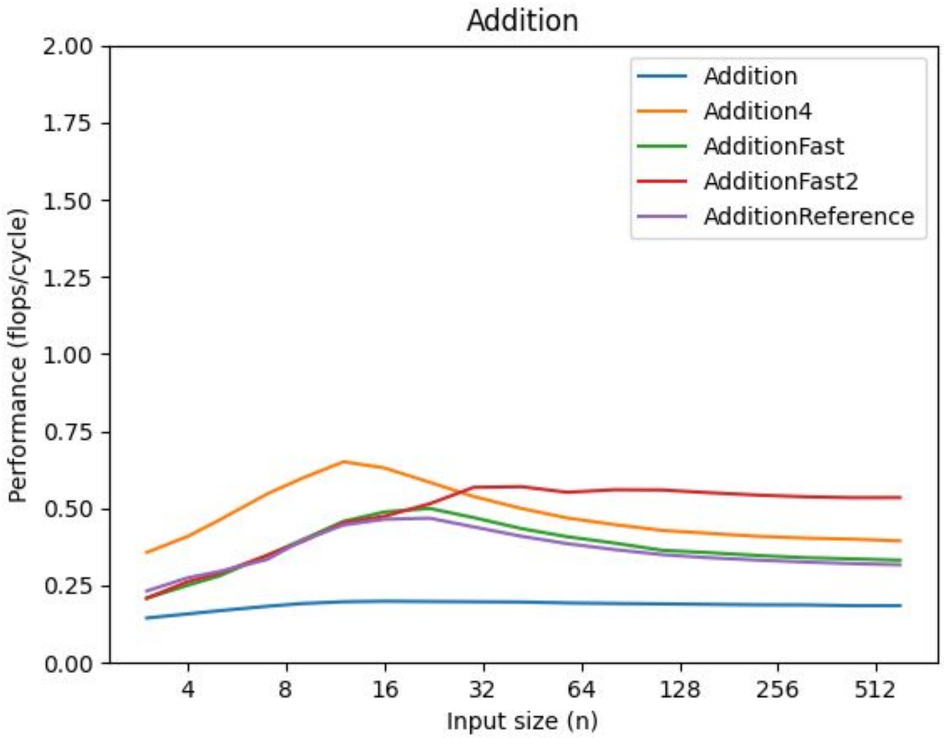
- **Adds two floating point expansions**
- **Components:** Merge + renormalization

Addition - baseline

```c
void renormalizationalgorithm(double* x, int size_of_x, double* f, int m)
{

    double *err = (double *)alloca((size_of_x) * sizeof(double));
    double *f_tmp = (double *)alloca((m + 1) * sizeof(double));
    for (int i = 0; i <= m; i++)
    {
        f_tmp[i] = 0;
    }
    vecSum(x, err, size_of_x);

    vecSumErrBranch(err, size_of_x, m + 1, f_tmp);

    for (int i = 0; i <= (m - 2); i++)
    {

        vecSumErr(&(f_tmp[i]), m - i + 1, &(f_tmp[i]));

        f[i] = f_tmp[i];
    }
    f[m - 1] = f_tmp[m - 1];

    return;
}
```

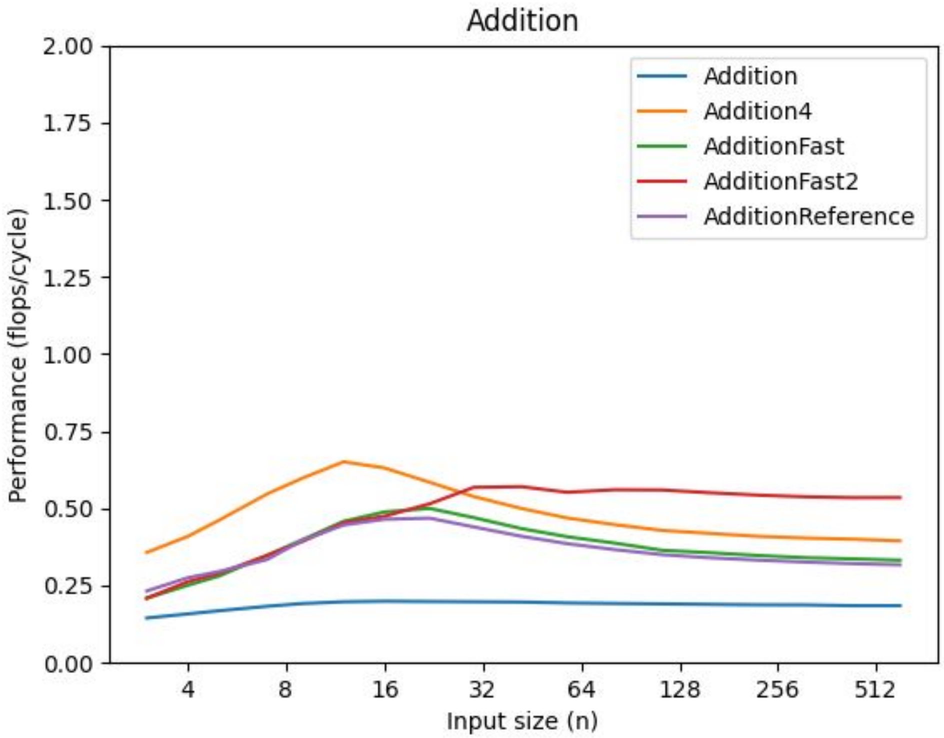
- **Renormalization:** normalizes input vector to ULP-nonoverlapping
- **Components:**
  - VecSum
  - VecSumErrBranch
  - VecSumErr

Addition - Performance



- **Addition:** baseline
- **Addition4**: includes optimized (and vectorized) components
- **AdditionFast:** uses fast two sum (performance is lower, but runtime also)
- **AdditionFast2:** uses vectorized loop of VecSumErr calls

Addition - Performance



- **Many dependencies** between instructions, so difficult to parallelize
- **Many memory operations**

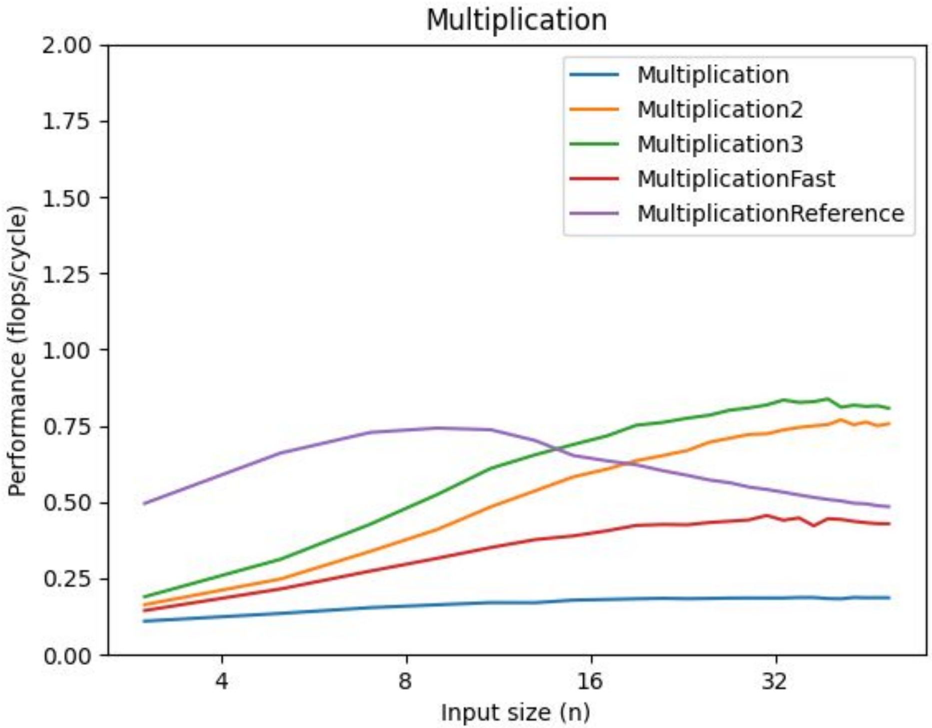
- low performance (< 0.65 flops/cycle)
- ~6.1x overall speedup

Multiplication 1 - baseline

```c
void multiplication(double *a, double *b, double *r, const int sizea, const int sizeb, const int sizer)
{
    ...

    twoMultFMA(a[0], b[0], &(r_ext[0]), &(err[0]));

    for (int n = 1; n <= (k - 1); n++)
    {
        ...

        for (int i = 0; i <= n; i++)
        {
            twoMultFMA(a[i], b[n - i], &(p[i]), &(e_tmp[i]));
        }

        ...

        vecSum(tmp, tmp1, (n * n + n));

        ...
    }

    ...

    renormalizationalgorithm(r_ext, k + 1, r, sizea);
}
```

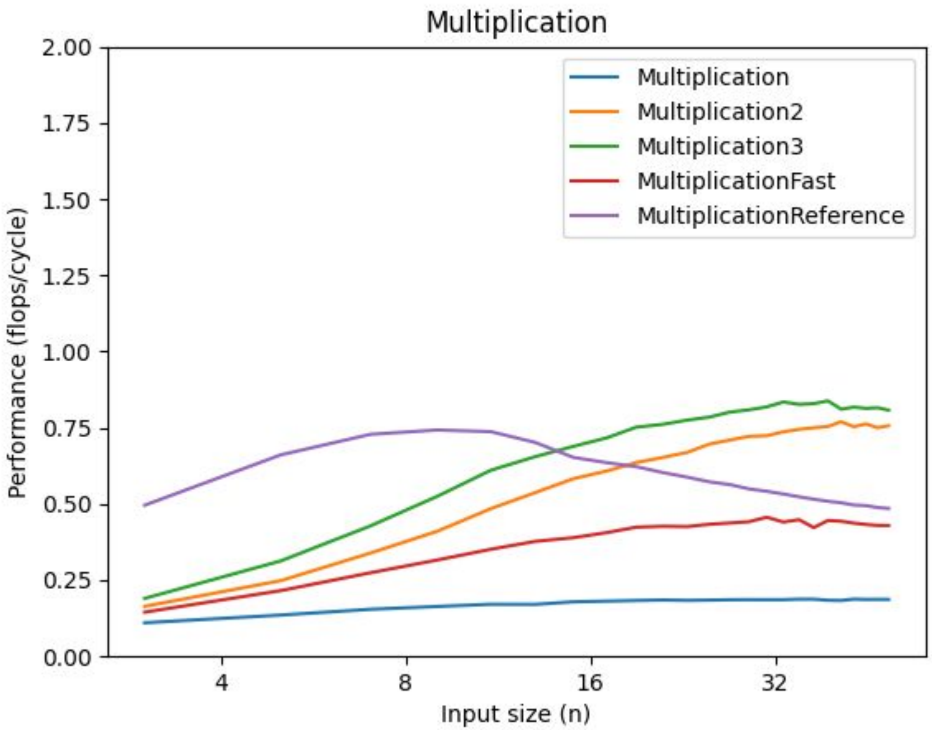
- **Multiplies two floating point expansions**
- **Components:**
  - TwoMultFMA
  - VecSum
  - Renormalization

Multiplication - performance



- **Multiplication:** baseline
- **Multiplication3**: Some vectorizations
- **MultiplicationFast:** uses fast two sum and vectorized loop of VecSumErr calls

Multiplication - performance



- **Even more memory operations** than Addition, so very low performance (< 0.75 f/c)
- ~4.8x speedup

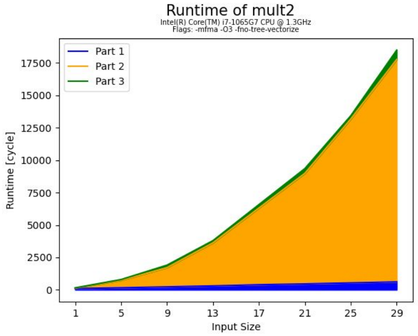
- Created a **specialized implementation** (input size of 4), but have not analyzed it yet

Multiplication Algo 2

Input: ulp-*nonoverlapping* FP expansions $x = x_0 + \ldots + x_{n-1}$; $y = y_0 + \ldots + y_{m-1}$.

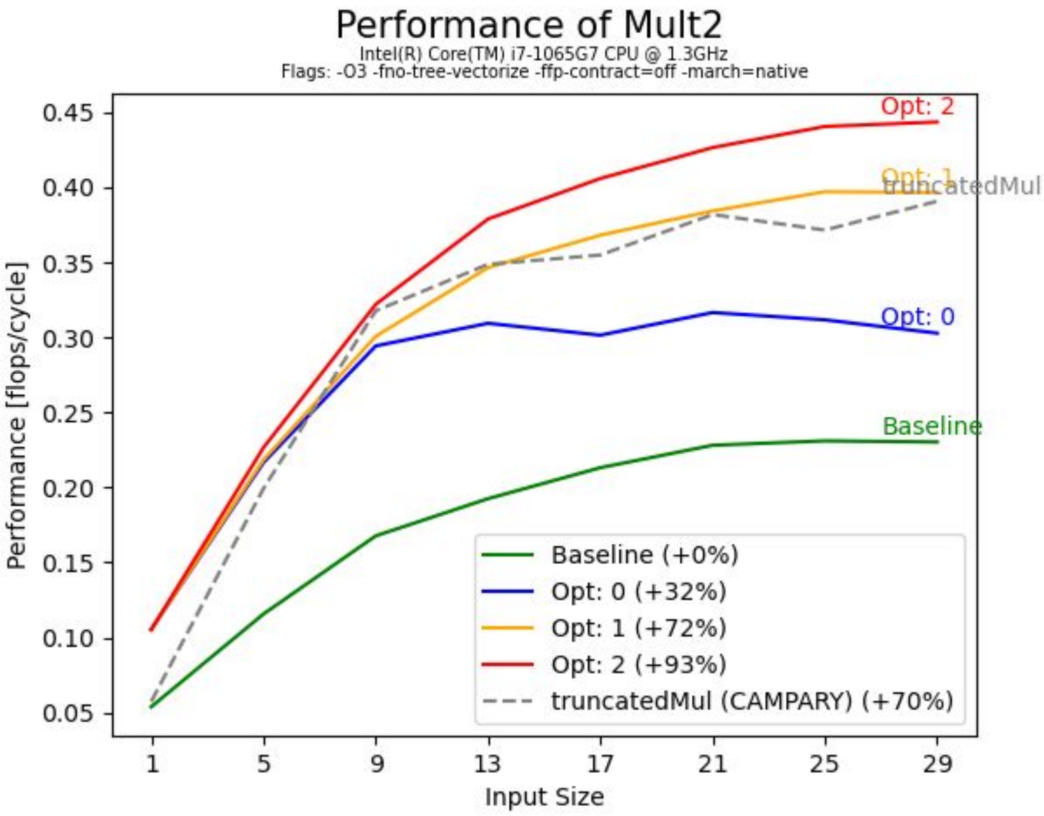
Output: ulp-*nonoverlapping* FP expansion $\pi = \pi_0 + \ldots + \pi_{r-1}$.

1: $e \leftarrow e_{x_0} + e_{y_0}$
2: **for** $i \leftarrow 0$ to $\lfloor r \cdot p/b \rfloor + 1$ **do**
3: $\quad B_i \leftarrow 1.5 \cdot 2^{e-(i+1)b+p-1}$
4: **end for**
5: **for** $i \leftarrow 0$ to $min(n-1, r)$ **do**
6: $\quad$ **for** $j \leftarrow 0$ to $min(m-1, r-1-i)$ **do**
7: $\quad\quad (P, E) \leftarrow \text{2MultFMA}(x_i, y_j)$
8: $\quad\quad \ell \leftarrow e - e_{x_i} - e_{y_j}$
9: $\quad\quad sh \leftarrow \lfloor \ell/b \rfloor$
10: $\quad\quad \ell \leftarrow \ell - sh \cdot b$
11: $\quad\quad B \leftarrow \text{Accumulate}(P, E, B, sh, \ell)$
12: $\quad$ **end for**
13: $\quad$ **if** $j < m-1$ **then**
14: $\quad\quad P \leftarrow x_i \cdot y_j$
15: $\quad\quad \ell \leftarrow e - e_{x_i} - e_{y_j}$
16: $\quad\quad sh \leftarrow \lfloor \ell/b \rfloor$
17: $\quad\quad \ell \leftarrow \ell - sh \cdot b$
18: $\quad\quad B \leftarrow \text{Accumulate}(P, 0., B, sh, \ell)$
19: $\quad$ **end if**
20: **end for**
21: **for** $i \leftarrow 0$ to $\lfloor r \cdot p/b \rfloor + 1$ **do**
22: $\quad B_i \leftarrow B_i - 1.5 \cdot 2^{e-(i+1)b+p-1}$
23: **end for**
24: $\pi[0 : r-1] \leftarrow \text{Renormalize}(B[0 : \lfloor r \cdot p/b \rfloor + 1])$
25: **return** FP expansion $\pi = \pi_0 + \ldots + \pi_{r-1}$.



- (l:1-4) initialize bins (B)
- (l:7-11) multiply each x_i with **each** y_j while *i*, *j* smaller than *r*
  -> add **product** and **overflow** to correct bin
- (l:13-18) multiply each x_i with **last** y_j while *i* smaller than *r*
  -> add only **product** to correct bin
- (l:21-24) make bins non-overlapping

Multiplication Algo 2



Optimizations:

- Opt 0: Precompute
- Opt 1: Opt 0 + Replace complex with simpler functions
- Opt 2: Opt 1 + Remove function calls

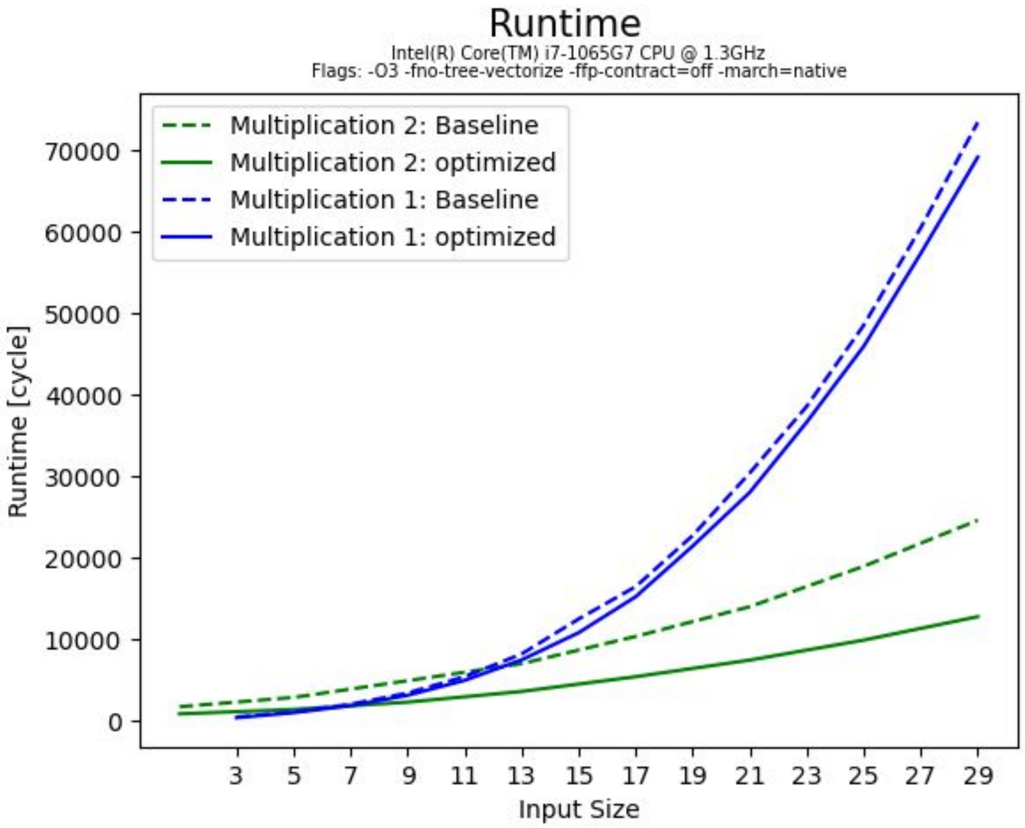
Why has CAMPARY lower Performance?

- Missing optimizations (e.g. divisions by const.)
- has less FLOPS
- probably optimized for GPU

Remarks:

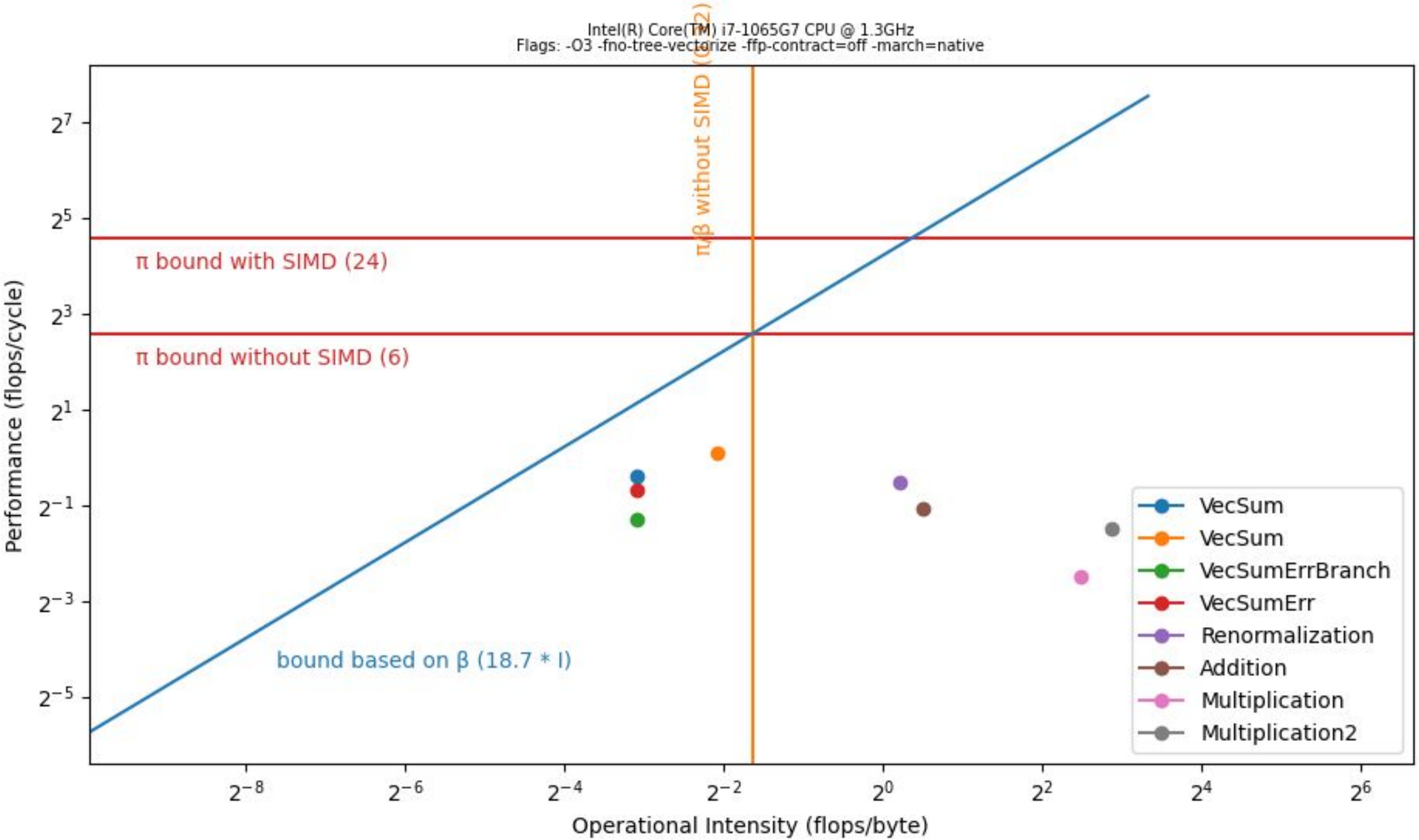
- Loop unrolling didn’t help

Comparison Mult1 & Mult2



- Multiplication 2 is faster for n > 7
- Reasons:
  - Different input requirements
  - Number of flops

Roofline model



References

- **Multiplication & Addition: https://ieeexplore.ieee.org/document/7118139**
- **Multipliation 2: https://ieeexplore.ieee.org/document/7563270**

General Remarks

- Pay attention to the length: e.g., 10 minutes typically means 7–8 slides
- Use proper visuals as much as possible, avoid text-only bullet slides
- Don’t put an overview or organization slide – the talk is too short
- For the very motivated, check out this small guide
  http://people.inf.ethz.ch/markusp/teaching/guides/guide-presentations.pdf

Typical Organization I

- **Algorithm that you consider (maybe 2 slides)**
  - State problem that it solves (input:…, output: …)
  - If possible visualize how it works or show high-level pseudocode
  - State asymptotic runtime
- **Cost analysis (cost measure, exact count)**
- **Baseline implementation (briefly explain), maybe show already performance plot and extract percentage of peak**
- **Optimizations you performed**
  - Briefly discuss major optimizations/code versions
  - Maybe explain the most interesting in a bit greater detail

Typical Organization II

- **Experimental results**
  - Very brief: Experimental setup (platform, compiler)
  - Performance plot over a range of sizes with different code versions
  - Make sure you also push input size to the limit in the experiments
  - Extract overall speedup

- **Every project is different – so adapt as needed**
- **Focus on the most interesting things, don’t explain everything that will be in the final report.**

Try to Make Nice Plots