Optimization of NumPy transcendental functions for Arm SVE

Fuyuka YAMADA, Kentaro KAWAKAMI, Kouji KURIHARA, Kazuhito MATSUDA, and Tsuguchika TABARU

Computing Laboratory, Fujitsu Limited

SVE instruction set | Background

Supercomputer Fugaku

Fujitsu A64FX

AWS Graviton3

- **SVE (Scalable Vector Extension) instruction set**
  - ARM's new vector instruction
  - SIMD register size
    - Specified as a multiple of 128, up to 2,048.
    - Can be selected by CPU manufacturer.
- CPU supporting SVE
  - **Fujitsu A64FX**
    - Used in supercomputer Fugaku
    - 512 bits vector length
  - **AWS graviton3**
    - 256 bits vector length

https://www.fujitsu.com/jp/about/resources/publications/technicalreview/2020-03/
https://www.fujitsu.com/global/products/computing/servers/supercomputer/a64fx/
https://www.itmedia.co.jp/news/articles/2112/01/news088.html

NumPy | Background

- NumPy is the primary Python math library.
  - Used in various applications such as scientific domain and machine learning.
  - Time consuming:
    - Such Python applications leave most of processing to NumPy.
    - NumPy occupies 75% of processing time in LiNGAM.
- NumPy is essential to speeding-up of Python applications.

Applications using NumPy

Analysis tool

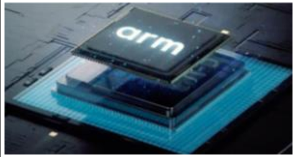
Quantum computing framework

LiNGAM

Causal inference

Issue

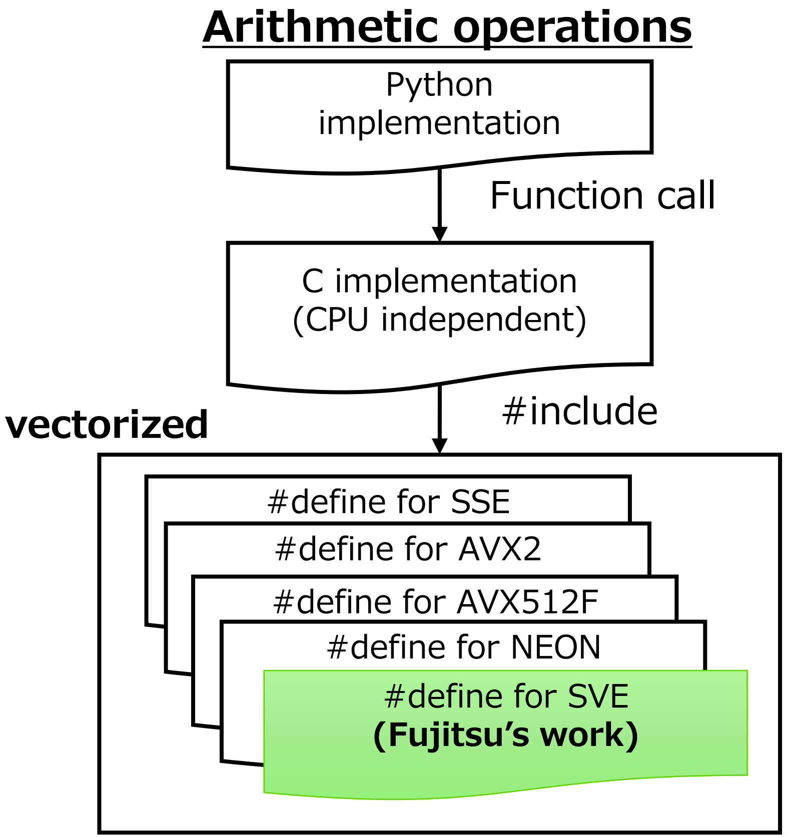
- NumPy currently does not support SVE.
  - Python applications using NumPy do not run fast as expected.

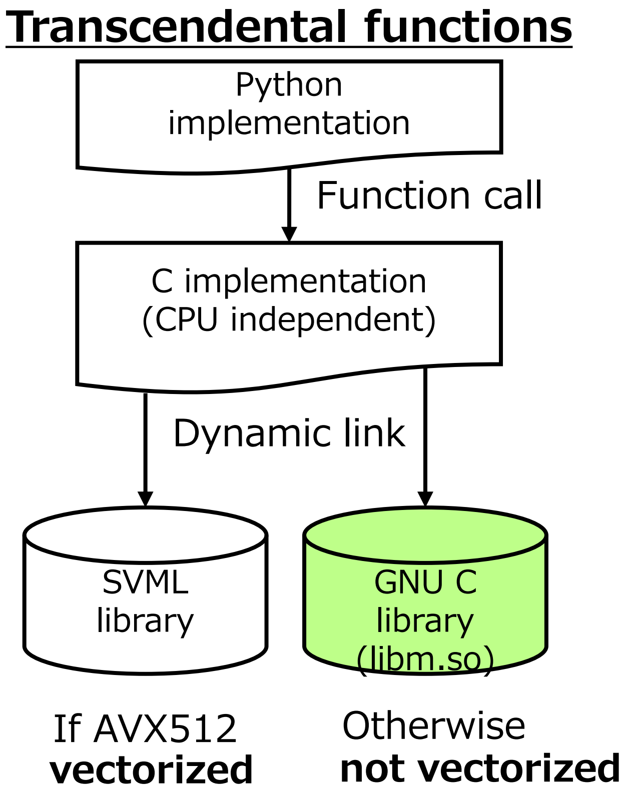
	Not supported	NumPy				
Supported Architecture	**SVE**	ASIMD (NEON)	SSE	AVX2	AVX 512	VEC
Hardware	Arm CPUs		Intel CPUs			PowerPC CPUs
	 https://www.sammobile.com/news/galaxy-s23-arm-cortex-x3-a715-a510-cpu-cores-improved-performance-efficiency/		https://www.intel.co.jp/content/www/jp/ja/products/docs/processors/core/13th-gen-processors.html			https://www.debian.org/ports/powerpc/index.ja.html

Purpose of this work

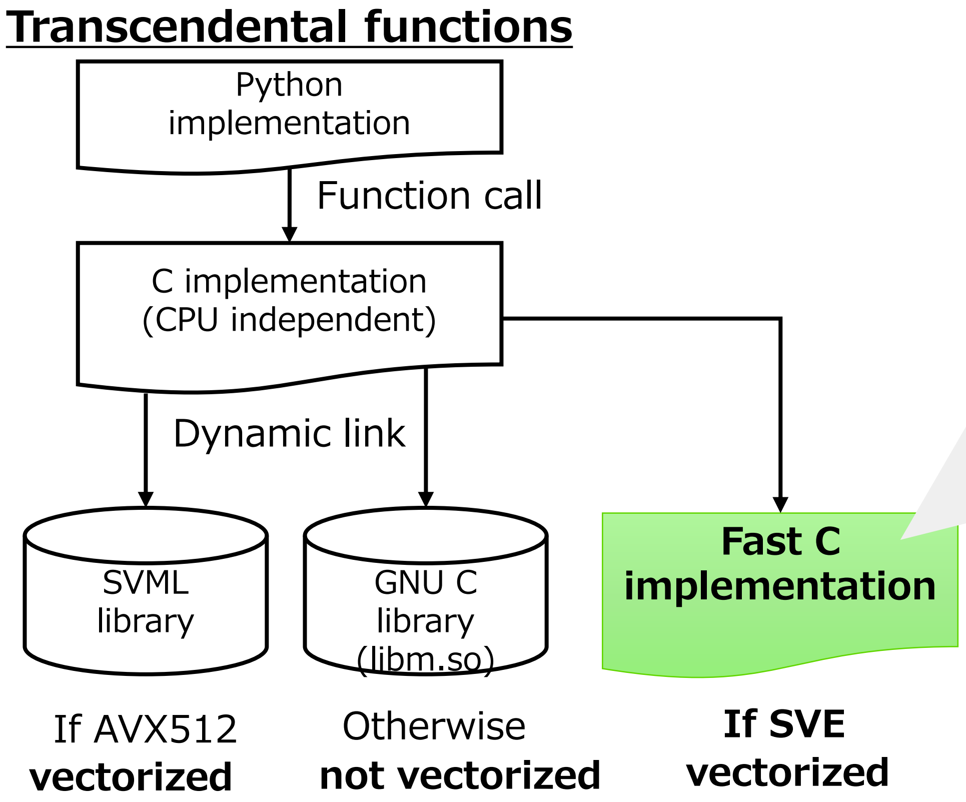
- Support of vectorized operation using SVE for speed-up
- Current situation
  - Arithmetic operations (addition, multiplication, etc.)
    - Fujitsu’s effort (but not accepted so far)
      - See “Implementation of NumPy AArch64 SVE support and its performance evaluation” in IPSJ SIG Technical Reports,2022-HPC-187(17),1-8 (2022-11-24) , 2188-8841
    - ASIMD (NEON) version is available, but only 128-bit length.
  - **Transcendental functions** (sin, log, exp, etc.)
    - Not supported, scalar operation using libm only.
- Contribution
  - Implementation of vectorized transcendental functions using SVE in NumPy

Source Code Structure (current)





Source code structure (new)



We implemented three schemes

1. **SIMD-style operation**
2. **Inlining**
   To reduce function calls.
3. **Loop-unrolling**
   To efficiently use operation units

1. SIMD-style implementation

FUJITSU

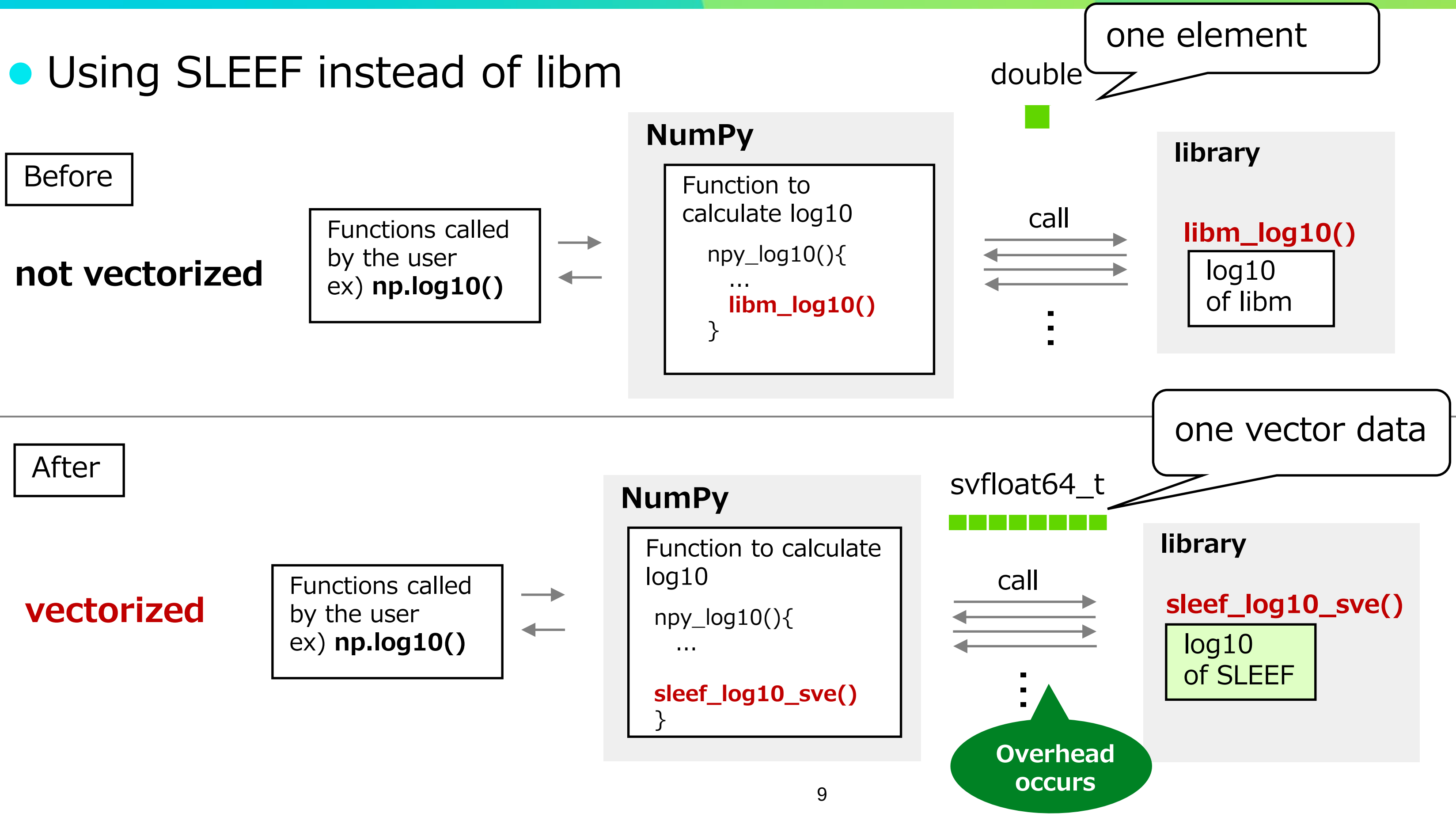
- Issue: Coding difficultly
  - Each transcendental function has a large number (up to 250) of lines.
    - Note: arithmetic function typically has 3 lines.
  - Manual coding is impractical.

- **Proposal: Use of the SIMD operations library SLEEF**
  - SLEEF supports various instruction sets such as NEON and SVE.
    - Implemented for SVE by ACLE (SVE built-in functions)
  - Boost Software License 1.0 (BSL -1.0)

SLEEF
SIMD Library for
Evaluating Elementary Functions

1. Example: log10()

- Using SLEEF instead of libm

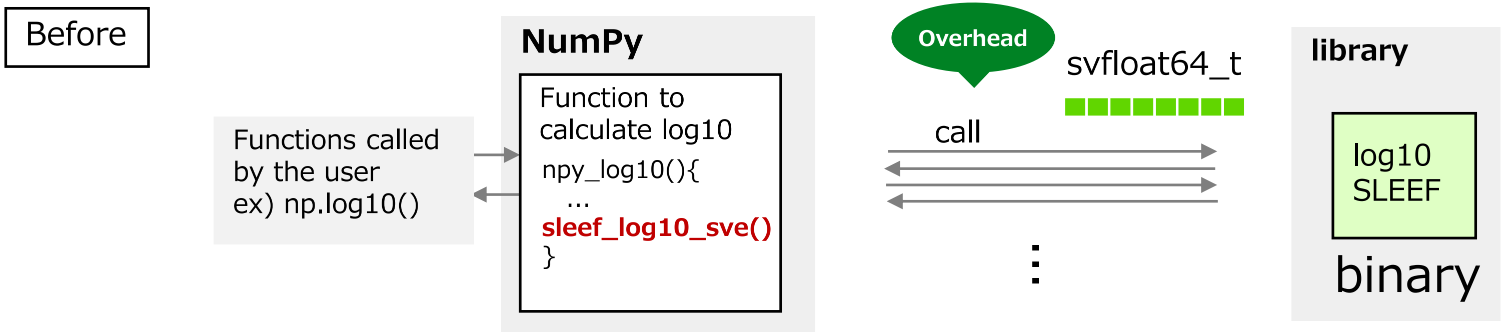


Further speed up | 2. Inlining

- Issue of using SLEEF: overhead of function call.
- Proposal: inlined SLEEF
  - The number of times of coefficient load and register save/return is reduced.

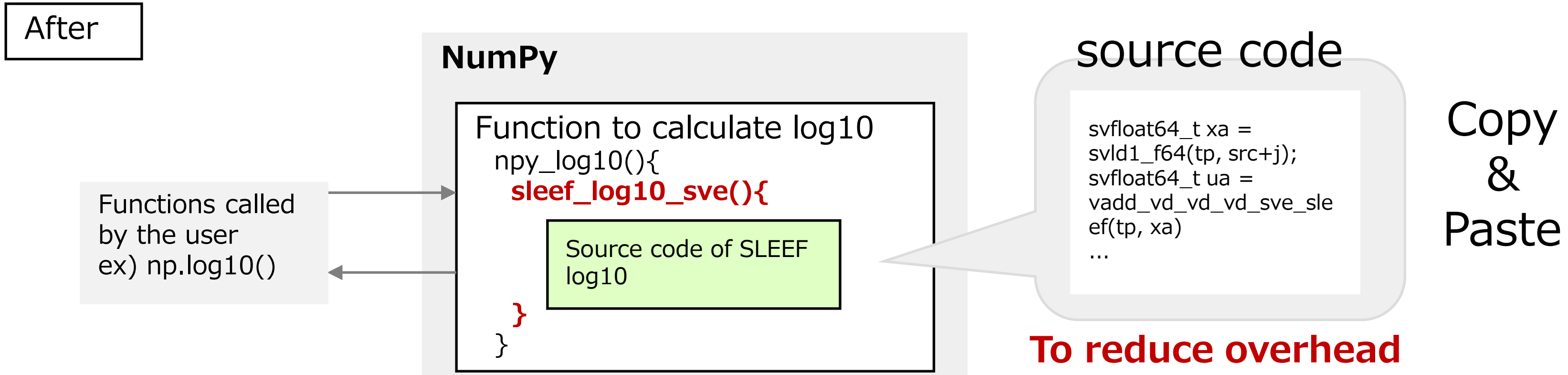
Our method to speed-up
1. SIMD by SLEEF
2. **Inlining**
3. Loop-unrolling

2. Inlining



Before
Functions called by the user ex) np.log10()
NumPy
Function to calculate log10
Overhead
svfloat64_t
call
library
log10 SLEEF
binary

```
npy_log10(){
  …
sleef_log10_sve()
}
```



After
Functions called by the user ex) np.log10()
NumPy
Function to calculate log10
Source code of SLEEF log10
source code
Copy & Paste

```
npy_log10(){
  sleef_log10_sve(){
    Source code of SLEEF log10
  }
}
```

```
svfloat64_t xa =
svld1_f64(tp, src+j);
svfloat64_t ua =
vadd_vd_vd_vd_sve_sle
ef(tp, xa)
…
```

To reduce overhead

Further speed up | 3. loop-unrolling

- Proposal: **Loop-unrolling**
  - Improve computing efficiency and reduce processing time by performing instruction sequence in parallel.

Our method to speed-up

1. SLEEF
2. Inline
3. **Loop-unrolling**

Example of loop unrolling

- We implemented loop-unrolling source code.

Before
without loop-unrolling

```
const int n = 1024;
const int l = 1000000;
double src[n];
double dst[n];
int vstep = 8; /* num of SIMD lanes */
const svbool_t tp = svptrue_b64();

for (int k=0; k<l; k++){
  int j = 0;
  for (int i=n; i > vstep; i -= vstep, j += vstep) {

    svfloat64_t x = svld1_f64(tp, src+j);
    ...
    svfloat64_t ua = vadd_vd_vd_vd_sve_sleef(tp, xa)
    ...
    svst1_f64(tp, dst+j, x);
}}
```

Expanding of SLEEF log10 function

After
with loop-unrolling 2

Unfold the processing of 2 loops into 1 loop

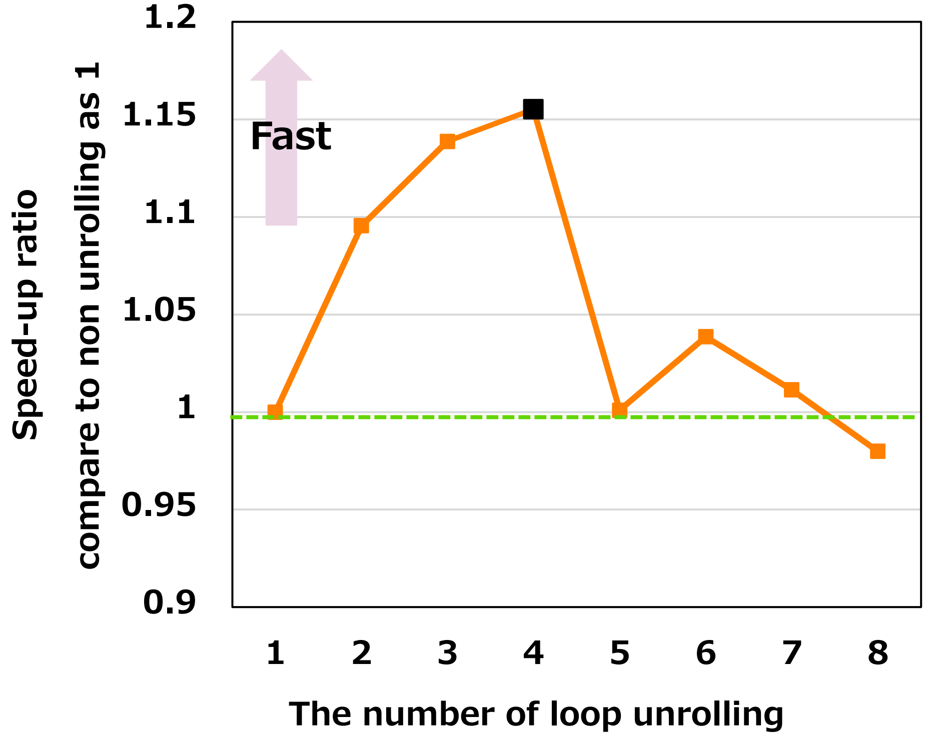
```
const int n = 1024;
const int l = 1000000;
double src[n];
double dst[n];
int vstep = 8; /* num of SIMD lanes */
const svbool_t tp = svptrue_b64();

for (int k=0; k<l; k++){
  int j = 0;
  for (int i=n; i > 2*vstep; i -= 2*vstep, j += 2*vstep) {

    svfloat64_t xa = svld1_f64(tp, src+j);
    svfloat64_t xb = svld1_f64(tp, src+j+vstep);
    ...
    svfloat64_t ua = vadd_vd_vd_vd_sve_sleef(tp, xa)
    svfloat64_t ub = vadd_vd_vd_vd_sve_sleef(tp, xb)
    ...
    svst1_f64(tp, dst+j, xa);
    svst1_f64(tp, dst+j+vstep, xb);

}}
```

Two vector processed

Each sub-process of two vectors are performed alternately.

※The method using pragma did not produce the intended assembler code.

Optimal number of unrollings | Results



- We analyze the effect of changing the number of unrolls.
- The speed-up ratio saturates at 4 for Log10.
- Optimal number is dependent to the operation.
  - Since each transcendental function uses different number of registers.

Experimental environment | 1. SIMD+2. inlined

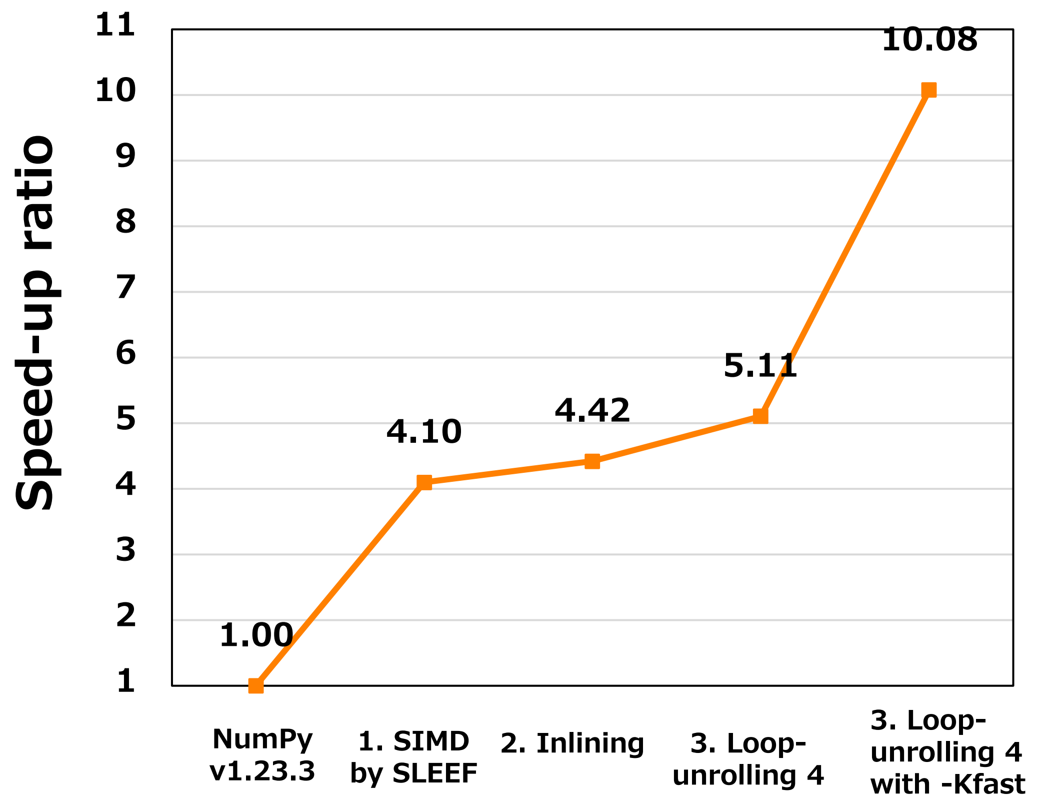
Machine	PRIMEHPC FX700
CPU	Fujitsu A64FX
Num of used core	1 core
OS	CentOS Linux 8
Compiler	Fujitsu compiler (trad mode)
Optimization option	-O3, -Kfast
NumPy	v1.23.3
Tested function	log10

```
import numpy as np

num = 1024
step = 1000000

a = np.random.rand(num)

for i in range(step):
    b = np.log10(a)
```

Python source code for log10

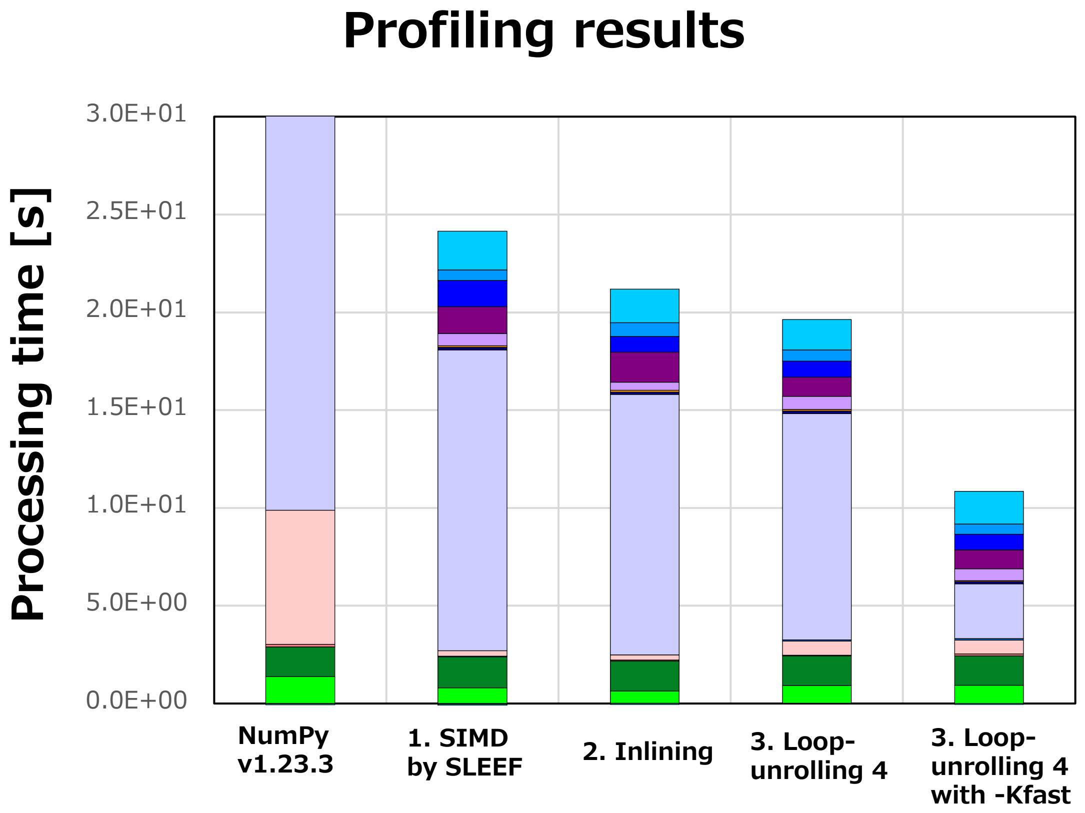
- Array Size: 1024
- Iteration: 1 million times

Summary of speed-up rate | Results



- 1. SIMD by SLEEF
  - log10: 4.1 times faster
- 2. Inline
  - log10: 0.3 times faster
- 3. Loop-unrolling
  - log10: 0.7 times faster(unroll 4)
- **1+2+3**
  - **log10: 5.11 times faster**
- (1+2+3+ **-Kfast option)**
  - log10: 10.8 times faster
  - Fujitsu compiler option
  - Stronger optimization than -O3
  - With the **-Kfast**, it does not pass the NumPy test.

Profiling | Results



1	2	3	4
🟪	🟦	🟦	🟦

- Instruction commit time
  - One to four instruction committed per clock.

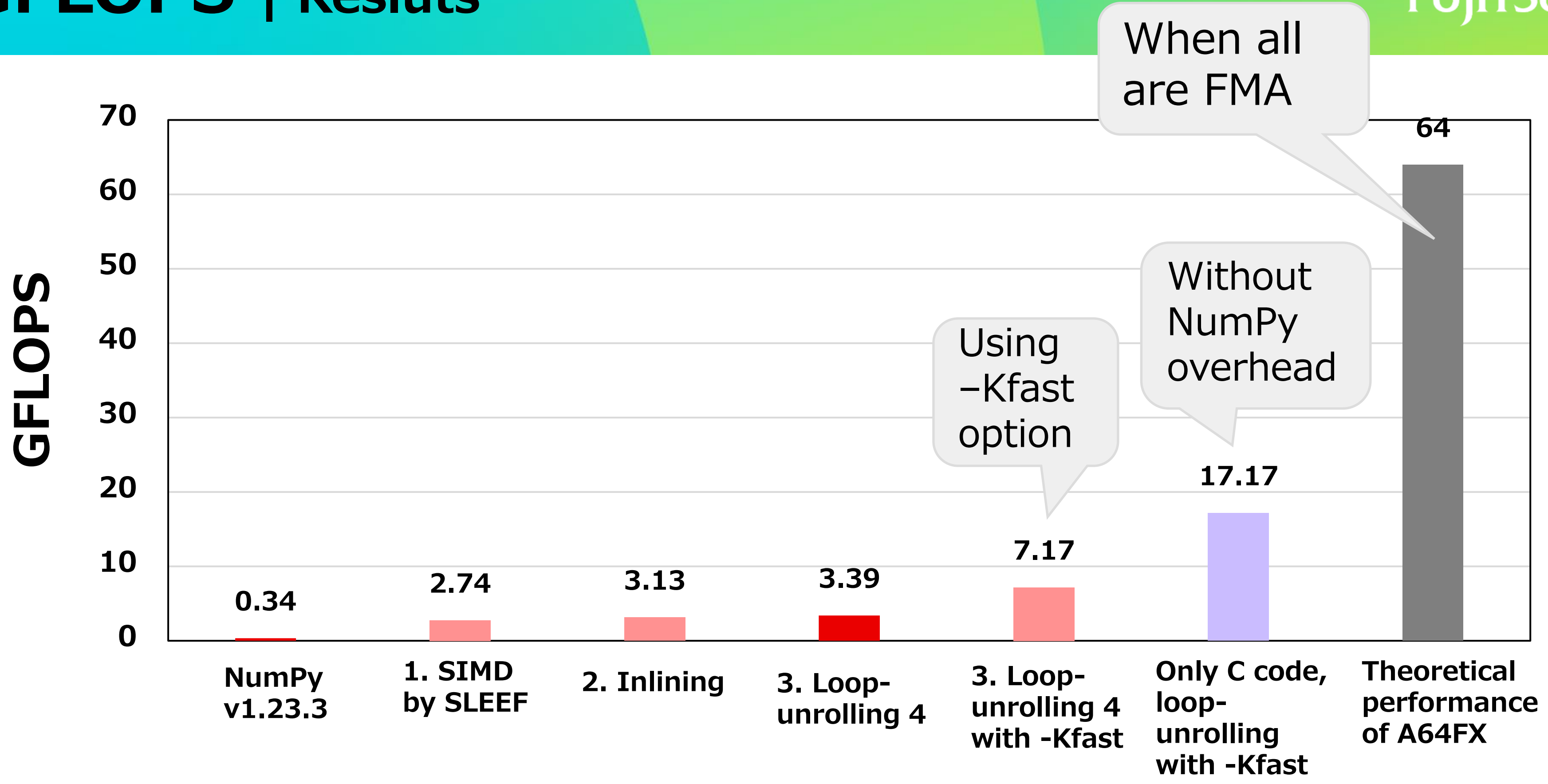
▢

- Operation wait time
  - The performance drops if this time becomes long.

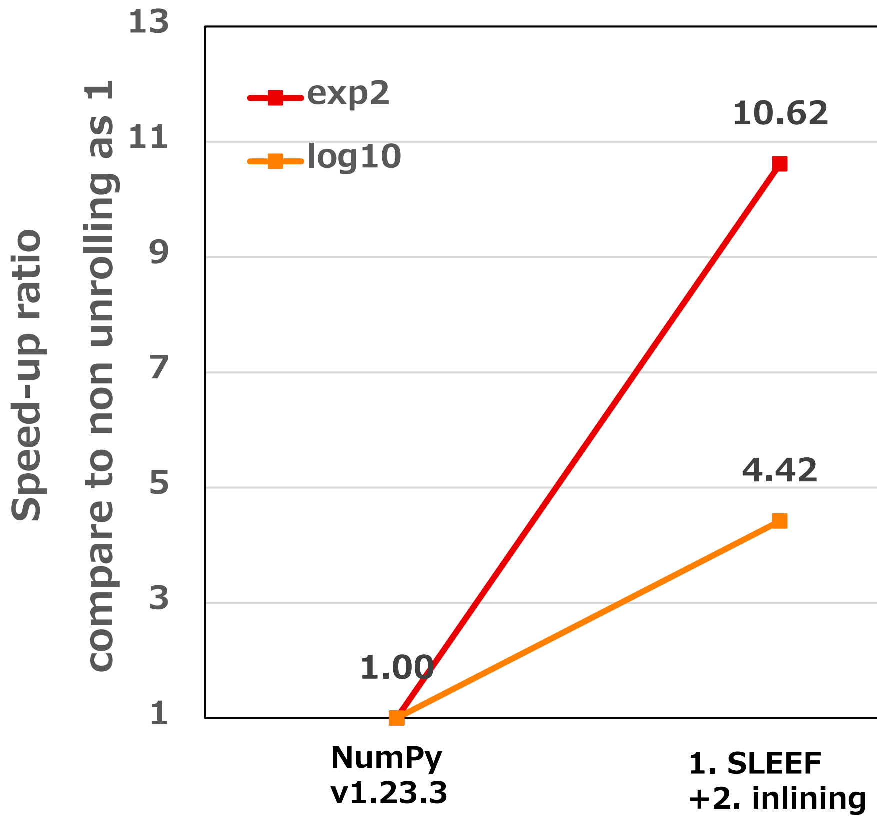
🟥 🟩 🟩

- Memory/cache access wait
  - The performance drops if this time becomes long.
  - In loop-unrolling, register spills occurs.

GFLOPS | Resluts



Speed-up in other functions



- Ongoing work
- Speed-up by 1. SLEEF, 2. inlining.
  - log10 (previously mentioned)
    - 4.42 times faster
  - exp2
    - 10.62 times faster

Conclusion

Goal

- To optimize vectorized transcendental functions of NumPy for CPU with SVE

Methods

1. SIMD by SLEEF
2. Inlining
3. Loop-unrolling

Results (speed-up)

- **Finally**
  - log10: 5.1 times speed-up (fcc, -O3)

Supplement

- NumPy with scheme 1 “SIMD by SLEEF (ASIMD version)” is already available.
  - Repository: https://github.com/yamadafuyuka/numpy/
  - Branch: add_SLEEF
  - Support functions: log2, log10, exp2.
  - SLEEF installation is needed for use.
  - We haven’t released the SVE version yet.

Thank you

yamada.fuyuka@fujitsu.com