NUMBA ON POWER

OPENPOWER ACADEMIC DISUSSION GROUP SC19, DENVER

16 November 2019 | Andreas Herten | Forschungszentrum Jülich *Handout Version*

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Outline

Numba Introduction

Numba

- **Numba**
  Python package for Just-in-Time Compilation (JIT)
  - No need for dedicated compilation step
  - No definition of bindings

→ Accelerate Python code many-fold

- Use LLVM for translation to machine code *(llvmlite)*
- Numpy-aware
- Backends for
  - CPU machine code
  - Parallel Central Processing Unit (CPU) (OpenMP, TBB, …)
  - GPU (NVIDIA. AMD)
- Open Source Software, developed mainly by Anaconda
- Architectures: Linux (`x86`, `ppc64le`, `arm64`), Windows, macOS
- Support for `ppc64le` since July 2018 (partial support before)

Using Numba

- Usually via Python decorators

  Decorators are Python functions with functions as inputs, potentially modifying these input functions; they can be used with `@decor` directly before the function

```python
def reduce(array):
    sum = 0
    for element in array:
        sum += element
    return sum
```

⟶

```python
import numba
@numba.jit
def reduce(array):
    sum = 0
    for element in array:
        sum += element
    return sum
```

- Alternative: Add on top of already defined functions

```python
def reduce(array):
    sum = 0
    for element in array:
        sum += element
    return sum
import numba
j_reduce = numba.jit()(reduce)
```

Using Numba

Options

`@jit(nopython=True)` Compile function in *nopython* mode

- Best performance
- Disable Python interpreter fully for jitted function
- Alternative *Object Mode*: only compile some sub-set of target

`@jit(parallel=True)` Enable automatic parallelization

`@vectorize` Create vectorized function (like it had scalars as inputs) (Numpy `ufunc`)

`@guvectorize` Create general vectorized function (with arrays as inputs, no outputs)

`@jit([float64(float64, float64)])` JIT function with signature

- No signature: *Lazy compilation* – compiled during first invocation (type inference)
- Signature: *Eager compilation* – compiled at first read

More `@stencil`, `@jitclass`, `@overload`, `fastmath=True`, `prange`, environment variables

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

STREAM Example

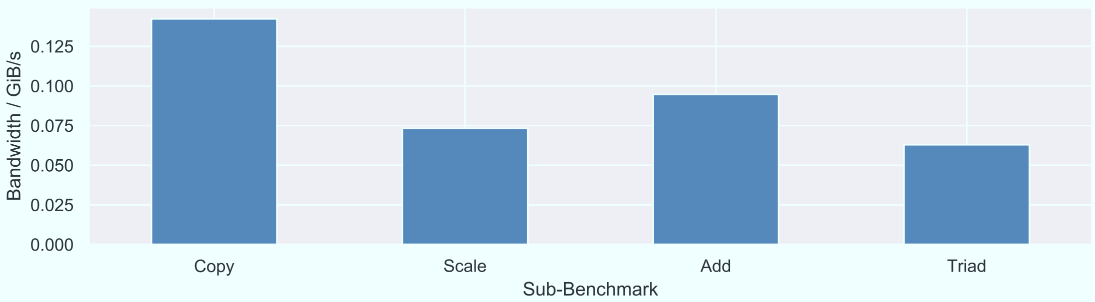
Plain Python

- Everything is better with **examples**!
- Use STREAM benchmark!
- Add Numba on top
- System under test: JURON (Minsky POWER8*NVL* system)
- Parameters
  - Bandwidth for 120 000 000 double-precision arrays
  - Best of 3 of average of 10
  - Time with `timeit`

```python
def copy(lhs, rhs):
    for i in range(len(lhs)):
        lhs[i] = rhs[i]

def scale(lhs, rhs, alpha):
    for i in range(len(lhs)):
        lhs[i] = alpha * rhs[i]

def add(lhs, rhs1, rhs2):
    for i in range(len(lhs)):
        lhs[i] = rhs1[i] + rhs2[i]

def triad(lhs, rhs1, rhs2, alpha):
    for i in range(len(lhs)):
        lhs[i] = rhs1[i] + alpha * rhs2[i]
```

STREAM Example

Plain Python



STREAM Example

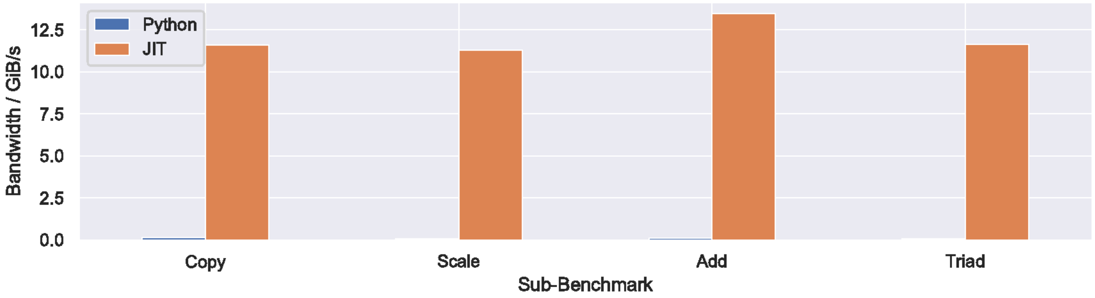
JIT

- Use same functions as before…
- … but now add Numba's JIT compiler

```
from numba import jit
copy  = jit('(float64[:], float64[:])', nopython=True)(copy)
scale = jit('(float64[:], float64[:], float64)', nopython=True)(scale)
add   = jit('(float64[:], float64[:], float64[:])', nopython=True)(add)
triad = jit('(float64[:], float64[:], float64[:], float64)', nopython=True)(triad)
```

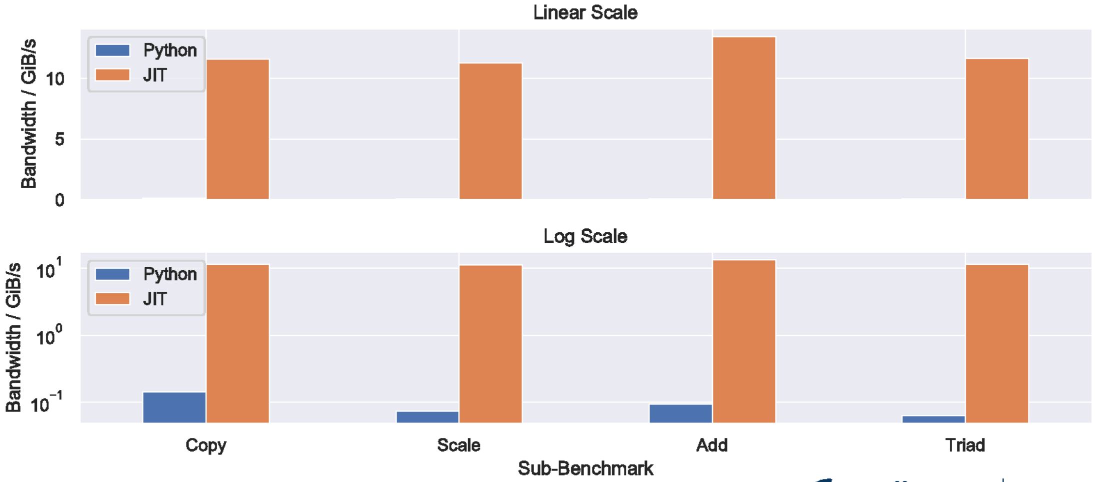
STREAM Example

JIT



STREAM Example

JIT



STREAM Example

guvectorize

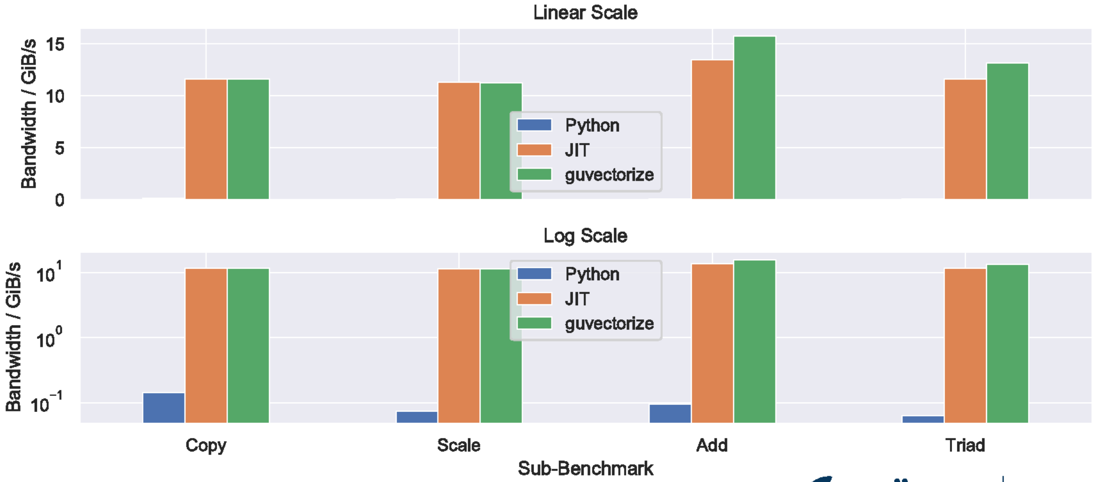
- Can this code benefit from vectorizing and making a general UFunc out of it?
- Additional parentheses in signature: layout of data types involved

$\rightarrow$ Write your own Numpy-like function working on whole arrays!

```python
from numba import guvectorize
copy  = guvectorize('(float64[:], float64[:])', '(n),(n)')(copy)
scale = guvectorize('(float64[:], float64[:], float64)', '(n),(n),()')(scale)
add   = guvectorize('(float64[:], float64[:], float64[:])', '(n),(n),(n)')(add)
triad = guvectorize('(float64[:], float64[:], float64[:], float64)', '(n),(n),(n),()')(triad)
```

STREAM Example

guvectorize



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

STREAM Example

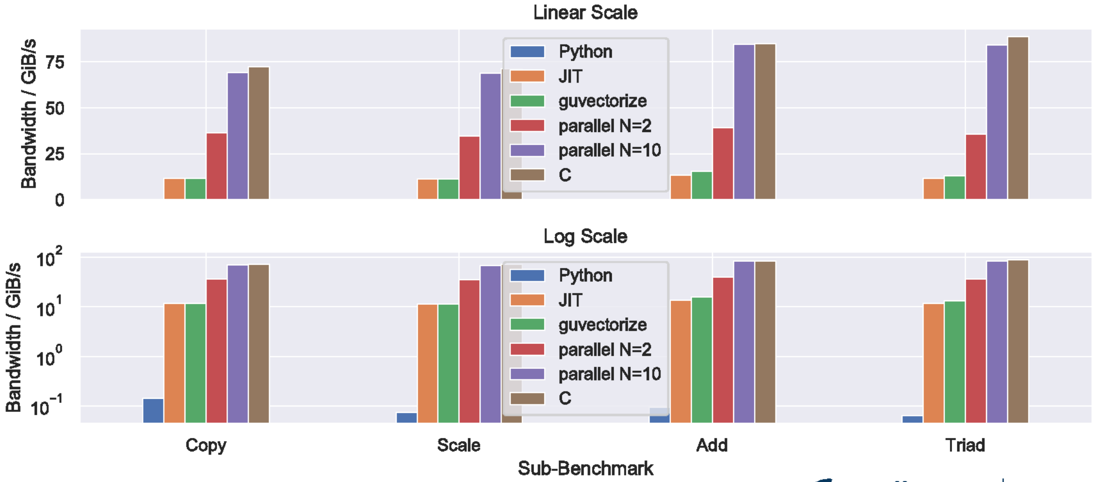
Explicitly Parallel

- Add `parallel=True`
  - Auto-parallelize common Numpy constructs
  - Fuse adjacent parallel operations (cache)
- Shortcut: `njit` is `jit` with `nopython=True`
- Explicit parallelism: `prange()` instead of `range()`
- Parallel backends: TBB, OMP, Workqueue
  - Choose backend: `NUMBA_THREADING_LAYER=omp`
  - Vary threads: `NUMBA_NUM_THREADS=N`

```python
@njit('(float64[:], float64[:])', parallel=True)
def copy(lhs, rhs):
    for i in prange(len(lhs)):
        lhs[i] = rhs[i]

@njit('(float64[:], float64[:], float64)',
↪   parallel=True)
def scale(lhs, rhs, alpha):
    for i in prange(len(lhs)):
        lhs[i] = alpha * rhs[i]

@njit('(float64[:], float64[:], float64[:])',
↪   parallel=True)
def add(lhs, rhs1, rhs2):
    for i in prange(len(lhs)):
        lhs[i] = rhs1[i] + rhs2[i]

# ... same for triad ...
```

STREAM Example

Explicitly Parallel



JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

GPU Support

- Backends for CUDA and ROCm
- Define GPU kernel with `@cuda.jit`
- In kernel, get thread identity with

```
i = cuda.grid(1)  # 1-dim
i = cuda.blockIdx.x *
↪   cuda.blockDim.x +
↪   cuda.threadIdx.x
```

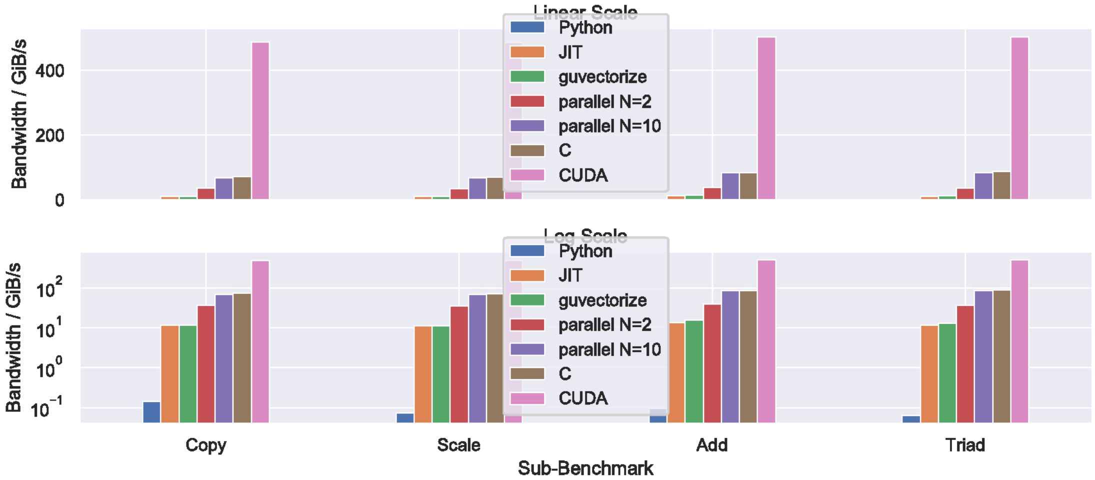
- Launch kernel:

```
kernel[nBlocks, nThreads](args)
```

- CUDA Event API: `numba.cuda.event`
- Support for shared memory, pinned memory
- Uses NVIDIA's NVVM compiler

```
@cuda.jit('void(float64[:], float64[:])')
def copy(lhs, rhs):
  i = cuda.grid(1)
  if i < lhs.shape[0]:
      lhs[i] = rhs[i]

@cuda.jit('void(float64[:], float64[:], float64)')
def scale(lhs, rhs, alpha):
  i = cuda.grid(1)
  if i < lhs.shape[0]:
      lhs[i] = alpha * rhs[i]

@cuda.jit('void(float64[:], float64[:], float64[:])')
def add(lhs, rhs1, rhs2):
  i = cuda.grid(1)
  if i < lhs.shape[0]:
      lhs[i] = rhs1[i] + rhs2[i]

# ... same for triad ...
```

GPU Support



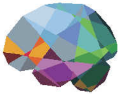
JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

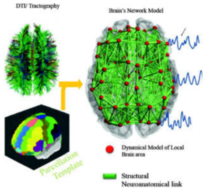
TVB-HPC

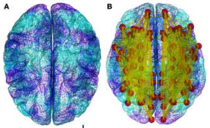
The Virtual Brain

- Framework for simulation of brain **dynamics** on large scale [2]
  - Biologically realistic connectivity matrix (*connectome*)
  - Neural mass models, sparse matrix linear solution (60 - 1000 elements), several free parameters
  - Models built on top of clinical data (fMRI, …)
  - Goal: Help patients with neurological disorders, compare brain
- Current software stack
  - Python simulation core, expendable by Matlab scripts
  - Web-based visual control center
  - Domain-Specific Language to describe brain models (*IDLE*)
- **TVB-HPC**: Use HPC technology for speed-up
  - Code-generation
  - Parallel backends (Numba CPU, Numba GPU, CUDA C/C++)

→ http://www.thevirtualbrain.org/tvb/







TVB-HPC's Kuramoto Model

- Kernel implemented in Numba CUDA by researchers during GPU Hackathon in Jülich
- Based on reference CUDA C/C++ and Numba CPU implementation

→ Study implementation details, programming model features

- Patterns: Simple arithmetic, function calls, unaligned memory accesses (gather, scatter)

```
@cuda.jit
def Kuramoto_and_Network_and_EulerStep_inner(nstep, nnode, ntime, state, input,
↪   param, drift, diffs, obsrv, nnz, delays, row, col, weights, a, i_step_0):
    tcoupling = cuda.threadIdx.x
    tspeed = cuda.blockIdx.x
    sid = cuda.gridDim.x
    idp = tspeed * cuda.blockDim.x + tcoupling
    for i_step in range(0, nstep):
        for i_node in range(0, nnode):
            idx = idp * nnode + i_node
            # ....
            for j_node in range(j_node_lo, j_node_hi):
                acc_j_node = acc_j_node +
                ↪   weights[j_node]*m.sin(obsrv[((idp*ntime+((i_step + i_step_0)
                ↪   % ntime) +
                ↪   -1*delays[tspeed*nnz+j_node])*nnode+col[j_node])*2] +
                ↪   -1*obsrv[((idp*ntime+((i_step + i_step_0) %
                ↪   ntime))*nnode+i_node)*2])
            input[idx] = a[tcoupling]*acc_j_node / nnode
            # ....
            obsrv[((idp*ntime + ((i_step + i_step_0) % ntime))*nnode + i_node)*2
            ↪   + 1] = m.sin(theta)
```

TVB-HPC's Kuramoto Model

```
@cuda.jit
def Kuramoto_and_Network_and_EulerStep_inner(nstep, nnode, ntime, state, input, param, drift,
↪  diffs, obsrv, nnz, delays, row, col, weights, a, i_step_0):
    tcoupling = cuda.threadIdx.x
    tspeed = cuda.blockIdx.x
    sid = cuda.gridDim.x
    idp = tspeed * cuda.blockDim.x + tcoupling
    for i_step in range(0, nstep):
        for i_node in range(0, nnode):
            idx = idp * nnode + i_node
            # ....
            for j_node in range(j_node_lo, j_node_hi):
                acc_j_node = acc_j_node + weights[j_node]*m.sin(obsrv[((idp*ntime+((i_step +
                ↪  i_step_0) % ntime) + -1*delays[tspeed*nnz+j_node])*nnode+col[j_node])*2] +
                ↪  -1*obsrv[((idp*ntime+((i_step + i_step_0) % ntime))*nnode+i_node)*2])
            input[idx] = a[tcoupling]*acc_j_node / nnode
            # ....
            obsrv[((idp*ntime + ((i_step + i_step_0) % ntime))*nnode + i_node)*2 + 1] =
            ↪  m.sin(theta)
```

How does Numba CUDA fare against plain CUDA C/C++?

$\Rightarrow$ Let's find out with TVB-HPC and some Micro-Benchmarks

TVB-HPC Numba
Micro-Benchmarks

STREAM Comparison

- STREAM Micro-Benchmark
  1. Numba CUDA
  2. CUDA C
- Measure execution time with
  1. CUDA Event API
  2. nvprof

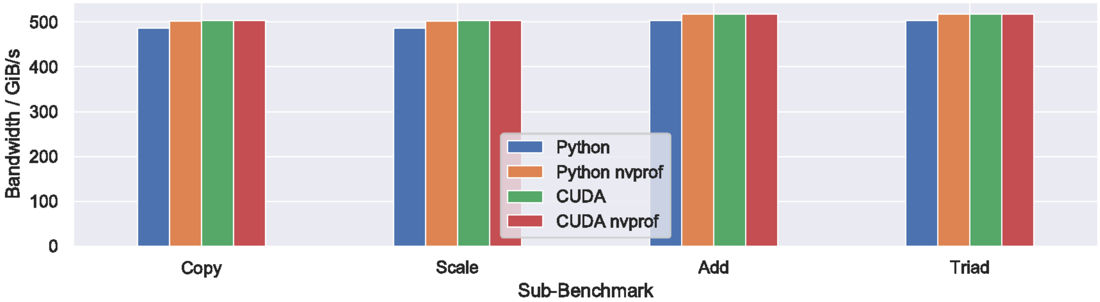
Numba CUDA

```python
@cuda.jit('void(float64[:], float64[:])')
def copy(lhs, rhs):
    i = cuda.grid(1)
    if i < lhs.shape[0]:
        lhs[i] = rhs[i]
```

CUDA C

```c
__global__ void copy(double *
↪   __restrict__ lhs, double *
↪   __restrict__ rhs, const int N) {
    const int i = blockDim.x * blockIdx.x
    ↪   + threadIdx.x;
    if (i < N)
      lhs[i] = rhs[i];
}
```

STREAM Comparison



- Same bandwidth!
- Overhead through Numba CUDA Event API

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Arithmetic: AXPY

- AXPY ($\vec{y} = a * \vec{x} + \vec{y}$)
  1. Numba CUDA
  2. CUDA C++
  3. cuBLAS
- Measure execution time with
  1. CUDA Event API
  2. `nvprof`
- Test precisions
  1. 32 bit
  2. 64 bit

Numba CUDA

```
@cuda.jit('void({dtype}, {dtype}[:],
↪  {dtype}[:])'.format(dtype=np.dtype(real_t).name))
def axpy(a, x, y):
    i = cuda.grid(1)
    if i < y.shape[0]:
        y[i] = a * x[i] + y[i]
```

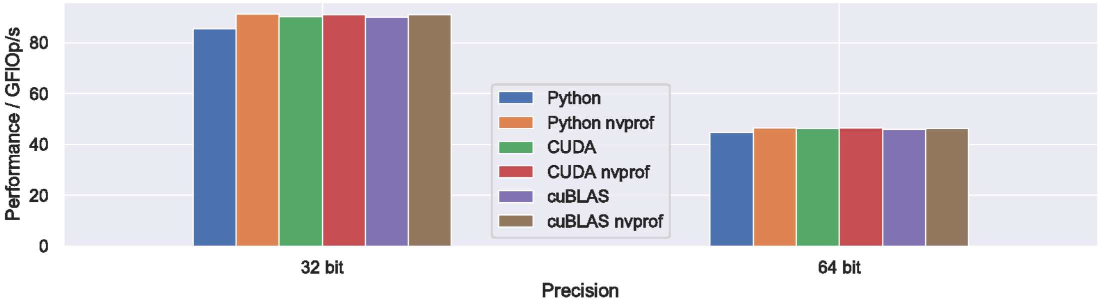
CUDA C

```
template<typename T>
__global__ void axpy(T a, T * __restrict__ x, T * __restrict__ y, const int
↪  N) {
    const int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
        y[i] = y[i] + a * x[i];
}
```

cuBLAS

```
cublasSaxpy(handle, n, alpha, x, incx, y, incy);
cublasDaxpy(handle, n, alpha, x, incx, y, incy);
```

Arithmetic: AXPY



- Same performance!
- Again, overhead through Numba CUDA Event API

Sine Function

- Kuramoto kernel uses `math.sin()`
- Let's compare compilation
- Time given for sin of 100 000 000 double-precision random numbers

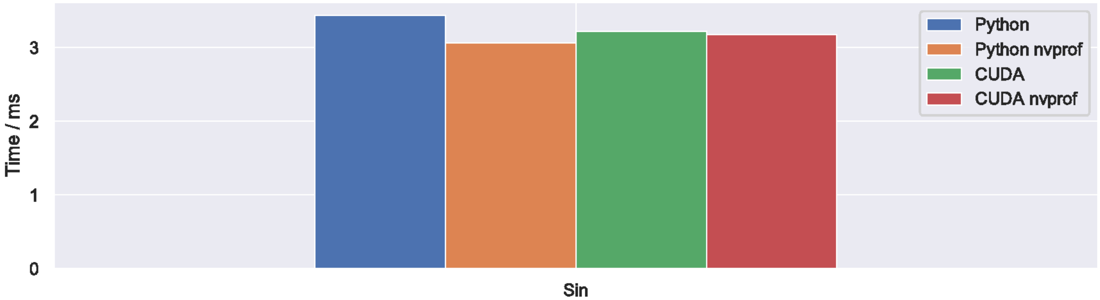
Numba CUDA

```python
@cuda.jit
def cuda_sin(lhs, rhs):
    i = cuda.blockIdx.x * cuda.blockDim.x +
    ↪  cuda.threadIdx.x
    if i < lhs.shape[0]:
        lhs[i] = math.sin(rhs[i])
```

CUDA C

```cpp
template<typename T>
__global__ void cuda_sin(T * __restrict__ lhs, T
↪  * __restrict__ rhs, const int N) {
    const int i = blockDim.x * blockIdx.x +
    ↪  threadIdx.x;
    if (i < N)
        lhs[i] = sin(rhs[i]);
}
```

Sine Function

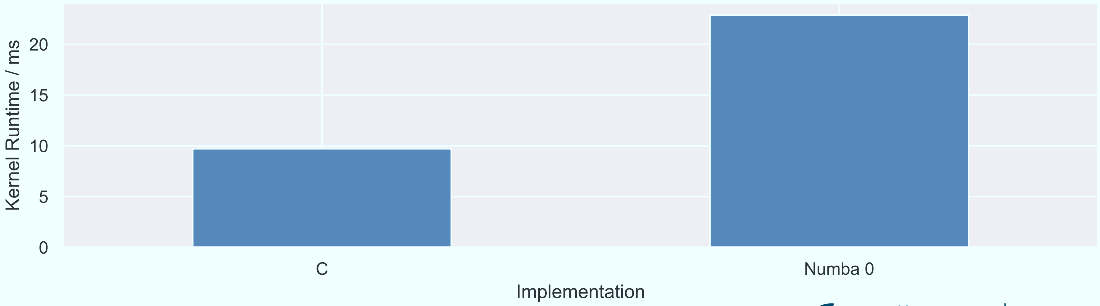


- Again again, some overhead through Numba CUDA Event API (type-checking!)
- Actual Python code seems to be slightly faster than C code
- Notable: First Python invocation takes $\approx$ 60$\times$ longer because of JIT
  Can be omitted if given a signature to `.jit()`
- Both generate lengthy, Taylor-Expansion-based sine PTX code

JÜLICH Forschungszentrum JÜLICH SUPERCOMPUTING CENTRE

Kuramoto Kernel Comparison

- Compare whole Numba CUDA kernel vs. CUDA C kernel
- CUDA C kernel: Compiled *also* just-in-time, but manually by calling `nvcc` from within Python on `.cu` file
- Facilitated via `pycuda.compiler`
  `https://documen.tician.de/pycuda/`



Kuramoto Kernel Zoom

- Arithmetic seems to be identical, external function as well
- Where is time spent in kernel?

→ Study data access pattern

Gather

```
for j_node in range(j_node_lo, j_node_hi):
    acc_j_node = acc_j_node + weights[j_node]
    ↪   * m.sin(obsrv[((idp * ntime +
    ↪   ((i_step + i_step_0) % ntime) -
    ↪   delays[tspeed * nnz + j_node]) *
    ↪   nnode + col[j_node]) * 2] -
    ↪   obsrv[((idp * ntime + ((i_step +
    ↪   i_step_0) % ntime)) * nnode + i_node)
    ↪   * 2])
```

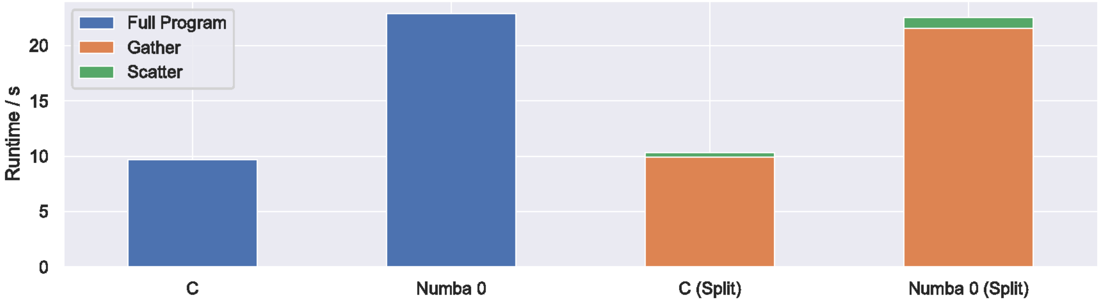
Scatter

```
state[idx] = (state[idx] < 0)*(state[idx] +
↪   6.283185307179586) + ...
theta = state[idx]
obsrv[((idp*ntime + ((i_step + i_step_0) %
↪   ntime))*nnode + i_node)*2 + 1] =
↪   m.sin(theta)
obsrv[((idp*ntime + ((i_step + i_step_0) %
↪   ntime))*nnode + i_node)*2] = theta
```

Kuramoto Kernel Zoom

- Arithmetic seems to be identical, external function as well
- Where is time spent in kernel?

→ Study data access pattern



Gather Refactoring

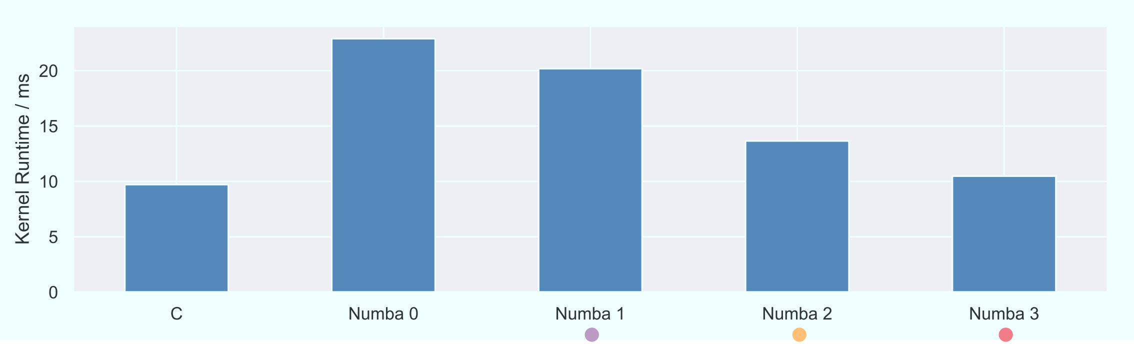
- Gathering data-access: 95 % time

→ Improve gathering performance
  - Not algorithmically
  - But by refactoring

1. Reduce direct global memory writes; add more temporary variables ●
2. Extract loop-constant variables ●
3. Create intermediates → extract more loop-constant variables ●

```
for i_step in range(0, nstep):
  ● step = (idp * ntime + ((i_step + i_step_0) % ntime))
    ↪   * nnode
    for i_node in range(0, nnode):
        # ...
      ● theta_i = obsrv[(step + i_node) * 2]
        for j_node in range(j_node_lo, j_node_hi):
          ● dij   = delays[tspeed*nnz + j_node]*nnode
            column = col[j_node]
            wij = weights[j_node]
            theta_j     = obsrv[(step - dij + column) * 2]
            acc_j_node += wij * m.sin(theta_j - theta_i)
      ● input_tmp = a[tcoupling] * acc_j_node / nnode
```

Gather Refactoring



- Nearly achieve C performance (7 % overhead)…
- …by simple refactoring

⇒ *Promising results for further work!*

Lessons Learned

Tips and Lessons Learned

- Timing from inside Python has overhead! (But `timeit` is very useful)
- The least: Using Numba's bindings to CUDA's Events API (`numba.cuda.event`)
- Use JIT signatures: Danger of running double-precision operations
  `jit("float32")` instead of `jit()`
- Re-defining variables in Python might break things (`a = 1; a=1.`)
- Check data alignment; it might be slow
  `cuda.jit("void(float32)")` made A-aligned data → fix with
  `cuda.jit("void(float32[::1])")`
- Analysis of generated code with `NUMBA_DUMP_ASSEMBLY` and `kernel.inspect_types()` and
  https://numba.pydata.org/numba-doc/latest/reference/envvars.html
- Use `numba -s` for a summary of the Numba installation

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Conclusions

Conclusion

- Numba can accelerate Python code by JIT compilation
- Targets: CPU, Parallel CPU, GPU
- Analysis of example application from TVB-HPC
- Micro-Benchmarks; Part-Benchmarks
- Matches CUDA C performance largely
- Sometimes, only after some refactoring

Thank you
for your attention!
a.herten@fz-juelich.de

- Thanks to Kuramoto authors Sandra Diaz and Marmaduke Woodman!

Appendix

Appendix

Glossary

References

Glossary I

AMD Manufacturer of CPUs and GPUs. 4, 43

API A programmatic interface to software by well-defined functions. Short for application programming interface. 16, 24, 25, 26, 27, 29

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 16, 19, 20, 22, 24, 25, 26, 27, 28, 29, 30, 38

JURON One of the two HBP pilot system in Jülich; name derived from Juelich and Neuron. 7

LLVM An open Source compiler infrastructure, providing, among others, Clang for C. 4

Numba JIT compiler for Python with various backends. 4, 7, 19, 20, 22, 24, 25, 26, 27, 28, 29, 30, 36, 38

Glossary II

NVIDIA US technology company creating GPUs. 4, 16, 41, 42

NVLink NVIDIA's communication protocol connecting CPU $\leftrightarrow$ GPU and GPU $\leftrightarrow$ GPU with high bandwidth. 42

OpenMP Directive-based programming, primarily for multi-threaded machines. 4

POWER CPU architecture from IBM, earlier: PowerPC. See also POWER8. 42

POWER8 Version 8 of IBM's POWER processor, available also within the OpenPOWER Foundation. 42

POWER8*NVL* POWER8 processor generation with NVLink connection between Graphics Processing Unit (GPU) and CPU. 7

Glossary III

ROCm AMD software stack and platform to program AMD GPUs. Short for Radeon Open Compute (*Radeon* is the GPU product line of AMD). 16

TVB-HPC High-Performance Computing sub-project of The Virtual Brain. 19

References I

[2] Paula Sanz Leon et al. “The Virtual Brain: a simulator of primate brain network dynamics”. In: *Frontiers in Neuroinformatics* 7 (2013). DOI: 10.3389/fninf.2013.00010 (page 19).

References: Images, Graphics I

[1] Nick Hillier. *Untitled*. Freely available at Unsplash. URL: https://unsplash.com/photos/yD5rv8_WzxA.

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE