Fakultät für
Mathematik und Informatik

Analysis and optimization of data transfer in Multi-GPU Python application

FernUniversität in Hagen

Why Python?

- HPC /Parallel Applications are often written in C, C++ or FORTRAN..
- People today learn of Python or R
- Data Science, but also in Physics, Biology and Engineering
- Often with pre-compiled libraries are used (often written in C, C++ or Fortran)
- Real Python is slower than „bare metal C"
  - Interpreted Language
  - Objects and limited memory management capabilities

SciPy

NumPy

SymPy

pandas

matplotlib

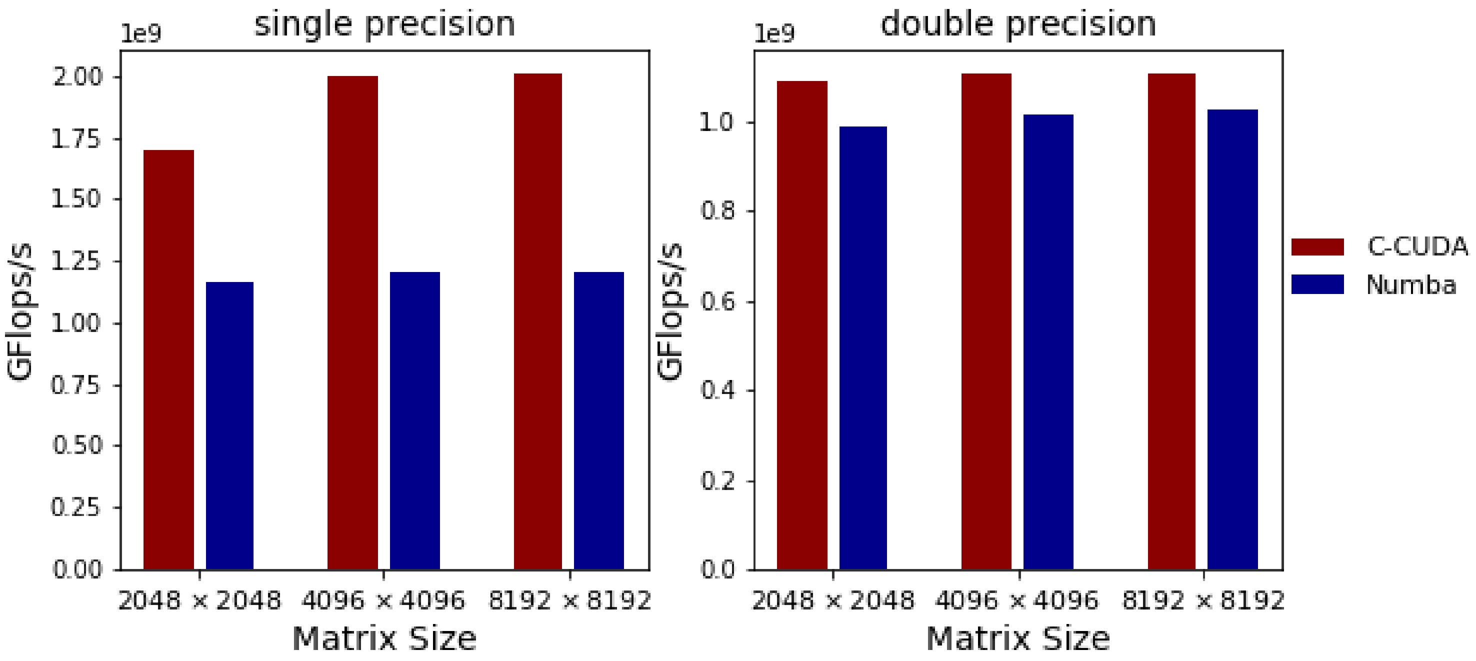
Numba (for GPUs)

```
@cuda.jit
def fast_matmul(A, B, C):

    sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    bx_offset, ay_offset = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y

    tmp = 0.0
    for i in range(0, cuda.gridDim.x, 1):
        by_offset = ty + i *TPB
        ax_offset = tx + i *TPB
        sA[ty,tx] = A[ay_offset, ax_offset]
        sB[ty,tx] = B[by_offset,bx_offset]
        cuda.syncthreads()
        for j in range(TPB):
              tmp += sA[ty,j] * sB[j,tx]

        cuda.syncthreads()
    C[ay_offset,bx_offset] = tmp
```

- Just-in-time compiler for Python
- Fast and parallel code in *Python Style*
- Not a “wrapper” for C like cython
- Not a library with optimized functions
- Functions are marked and compiled during execution
- Support for GPUs with CUDA-like programming (Nvidia GPU)

Numba: Performance compared to C-CUDA

- Matrix-Matrix Multiplication
  - Optimized algorithm with sh memory
  - Block wise algorithm
  - The same implementation
- Ignore JIT compile-overhead (exclude first iteration)
- Ignore Data Transfer (pure GPU-I Time)
- Performance is much worse



NVIDIA TeslaV100 GPU
IBM POWER9 processors(8 cores per core).
CUDA 10.1.105

Optimizations

```python
@cuda.jit
def fast_matmul(A, B, C):

    sA = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    sB = cuda.shared.array(shape=(TPB, TPB), dtype=float32)
    bx_offset, ay_offset = cuda.grid(2)

    tx = cuda.threadIdx.x
    ty = cuda.threadIdx.y

    tmp = 0.0
    for i in range(0, cuda.gridDim.x, 1):
        by_offset = ty + i *TPB
        ax_offset = tx + i *TPB
        sA[ty,tx] = A[ay_offset, ax_offset]
        sB[ty,tx] = B[by_offset,bx_offset]
        cuda.syncthreads()
        for j in range(TPB):
              tmp += sA[ty,j] * sB[j,tx]

        cuda.syncthreads()
    C[ay_offset,bx_offset] = tmp
```

- Use float32 declaration for tmp (if singe precision)

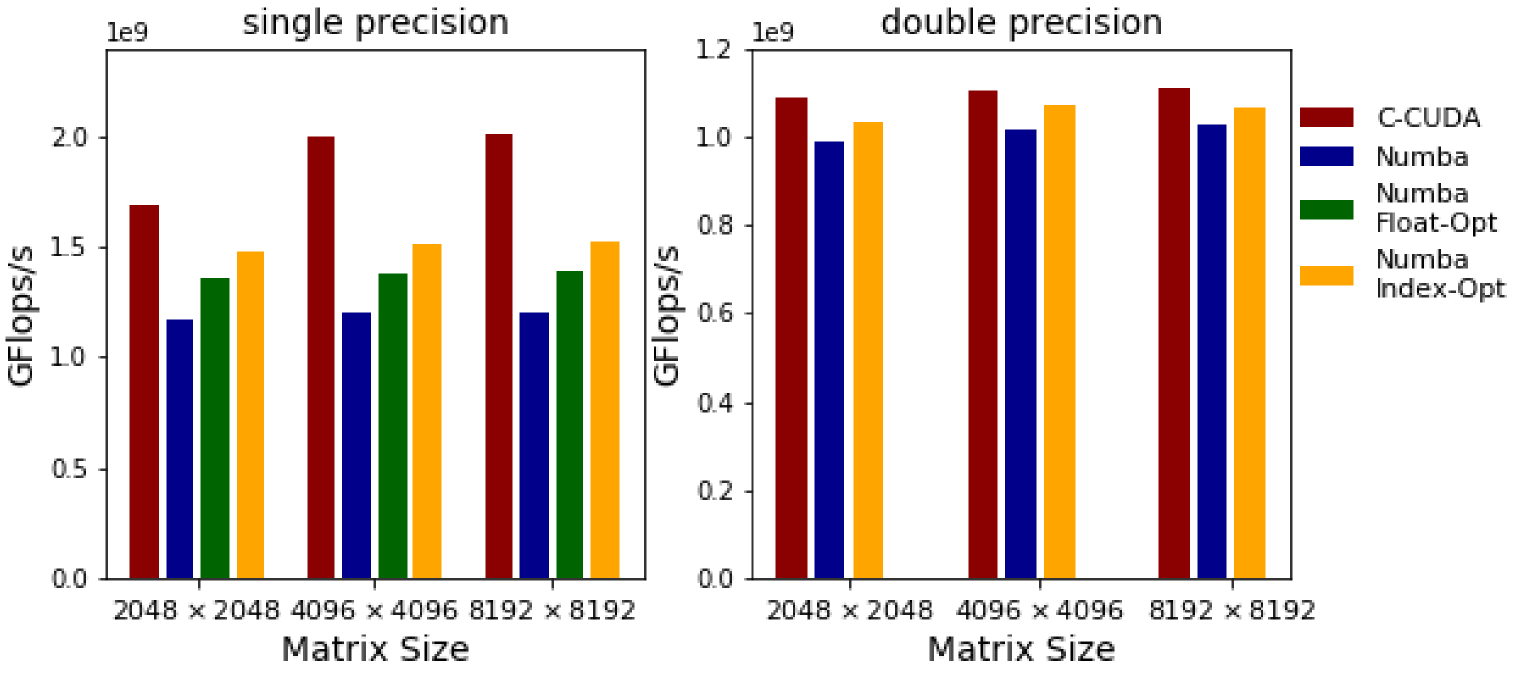
```python
tmp = float32(0.0)
```

- Further analysis of PTX Code

```
shr.s64 %r1, %rd0, 63;
and.b64 %r2, %r1, rd22;
rd22=A.shape[1]
add.s64 %r3, %r2, %rd0;
```

- Python allows negative indices – Additional Integer operations
- Change in numba
  - theadId, blockId to positive integers

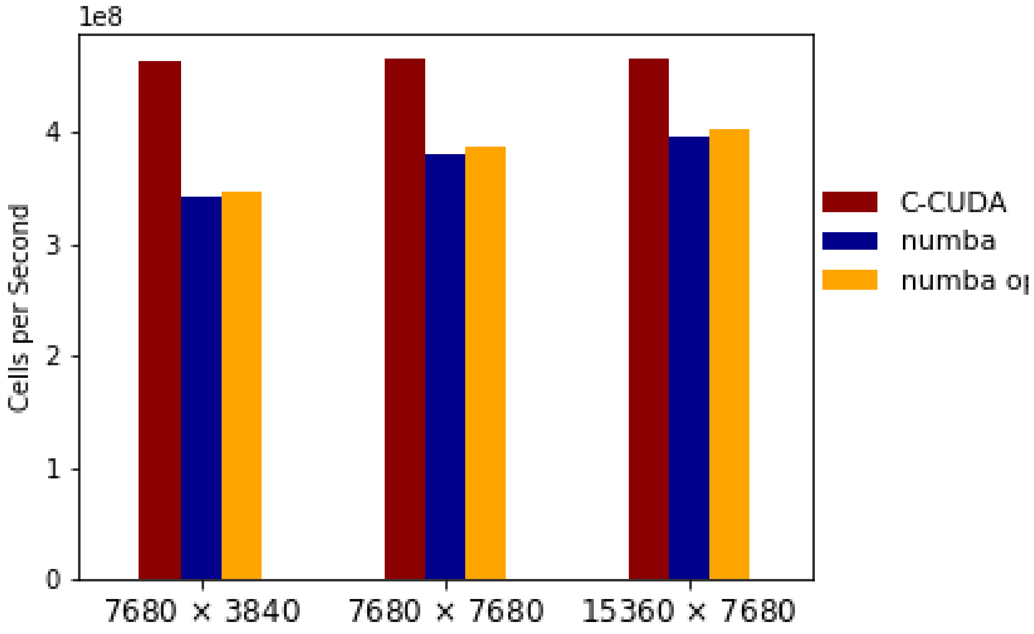
Optimization Results

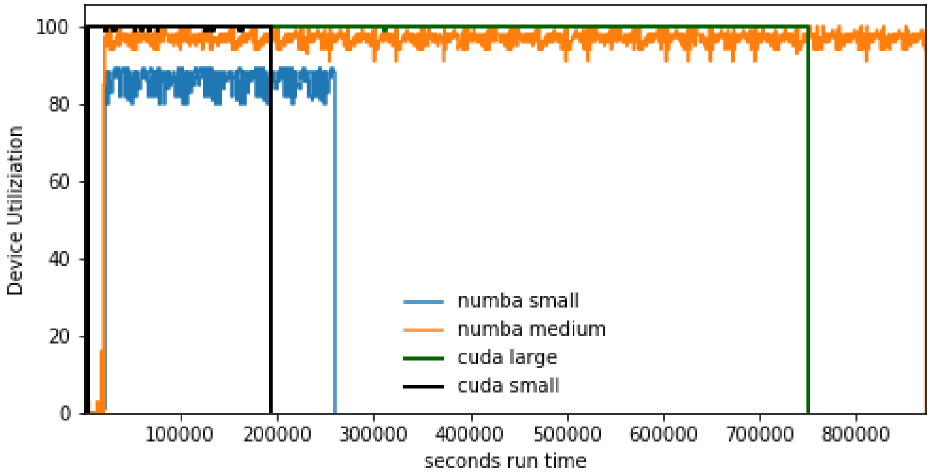


- Performance closer to the C Variants
- For single-precision, types are important
- Still lower performance

Application-like Benchmark: CloverLeaf

- For „real“ applications, the performance difference: 75% for small problem size, 86% for larger problems
- GPU-Utilization: For C-Version 100%, much lower for Python-Version
  - Python Overhead „between“





CuPy

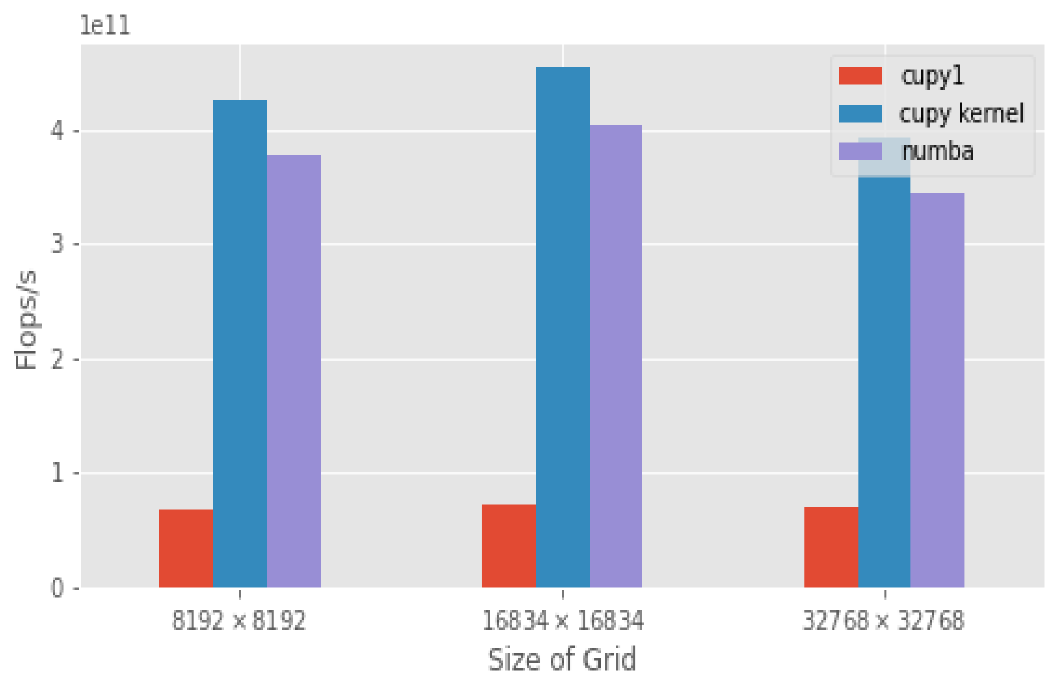
- Cupy allows the use of CUDA-Device Arrays like numpy arrays
- Allows an easy adaption of NUMPY-Applications for GPUs
- Support for CUPS and other high-performance libraries
- Using pre-compiled libraries -> high performance
- Compatible with numba (arrays can be used)

```
import cupy as cp
A = cp.random.rand(8192,8192)
B = cp.random.rand(8192,8192)

C = A @ B
```

CuPy 2

- Also allows the definition of custom kernels
- Sometimes more efficient than using the numpy-style Coding
- Use JIT-Compilation
- Different Approach than Numba:
  - Direct Source-to-Source compilation, without LLVM
- Current interface less "Pythonic"
- But: Good performance

```
data_b[1:-1,1:-1]= 0.25 *
```

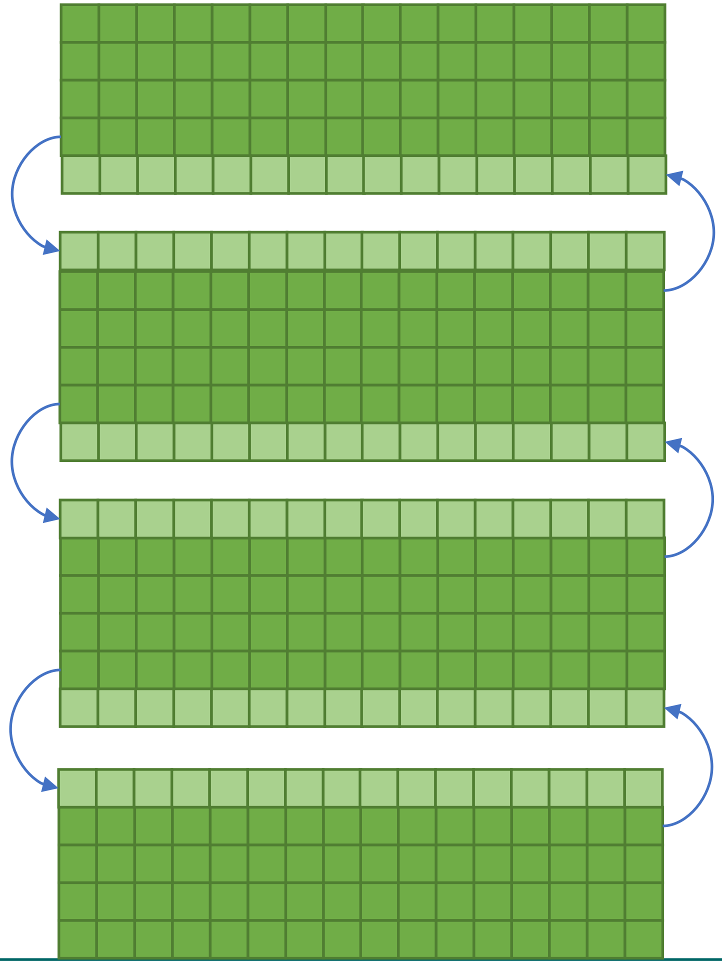



```
tx, ty = cuda.grid(2)
if tx>1 and tx<a.shape[0]-1 and ty> 1 and ty<a.shape[1]- 1:
  b[ty][tx] = 0.25 *
       (a[ty][tx + 1] + a[ty][tx-1] +
        a[ty + 1][tx] + a[ty-1][tx])
```

But what is about multiple GPUs?

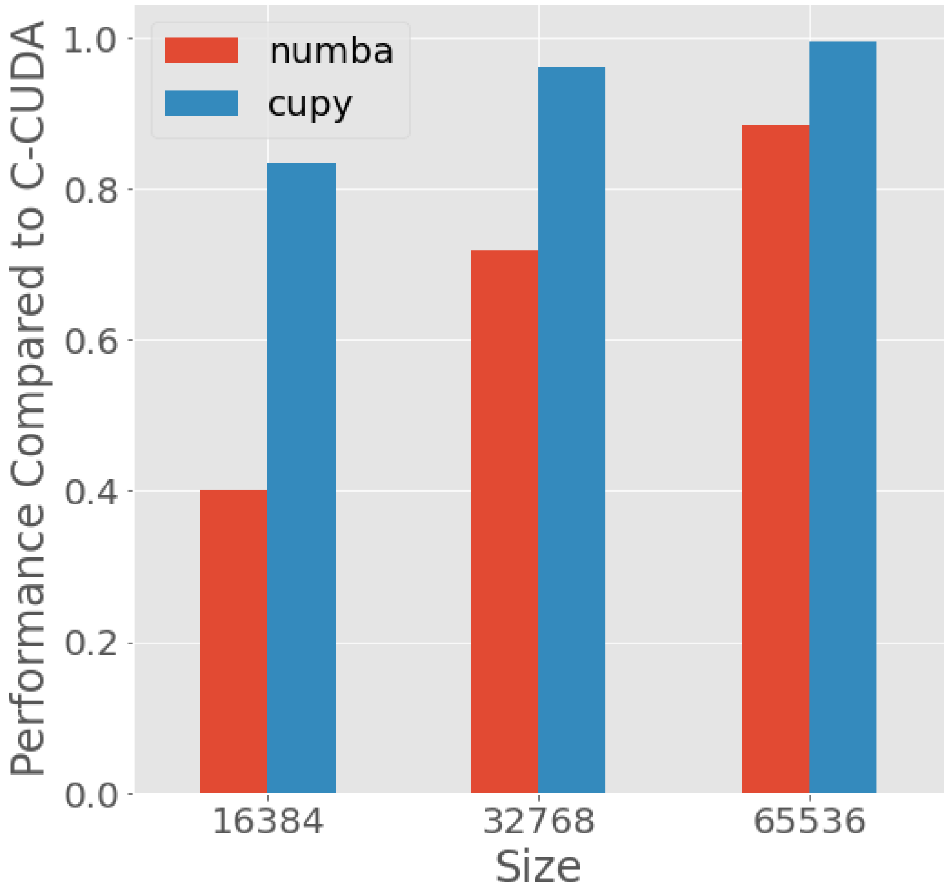
- MPI is the defacto standard for communication in HPC-Systems
- CUDA-Aware MPI allow communication of GPU memory
- Highly Optimized GPU-Data Transfer
- mpi4py is a wrapper around MPI for Python
- mpi4py is also CUDA-Aware

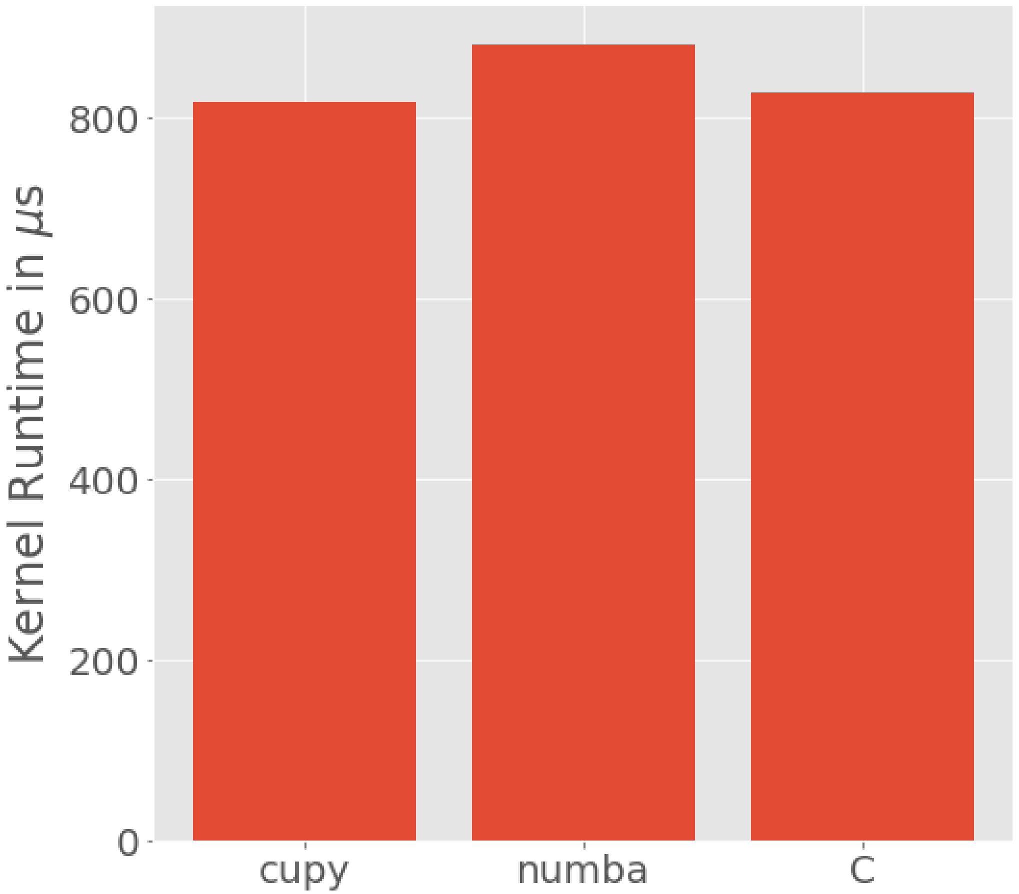
Jacobi-Kernel



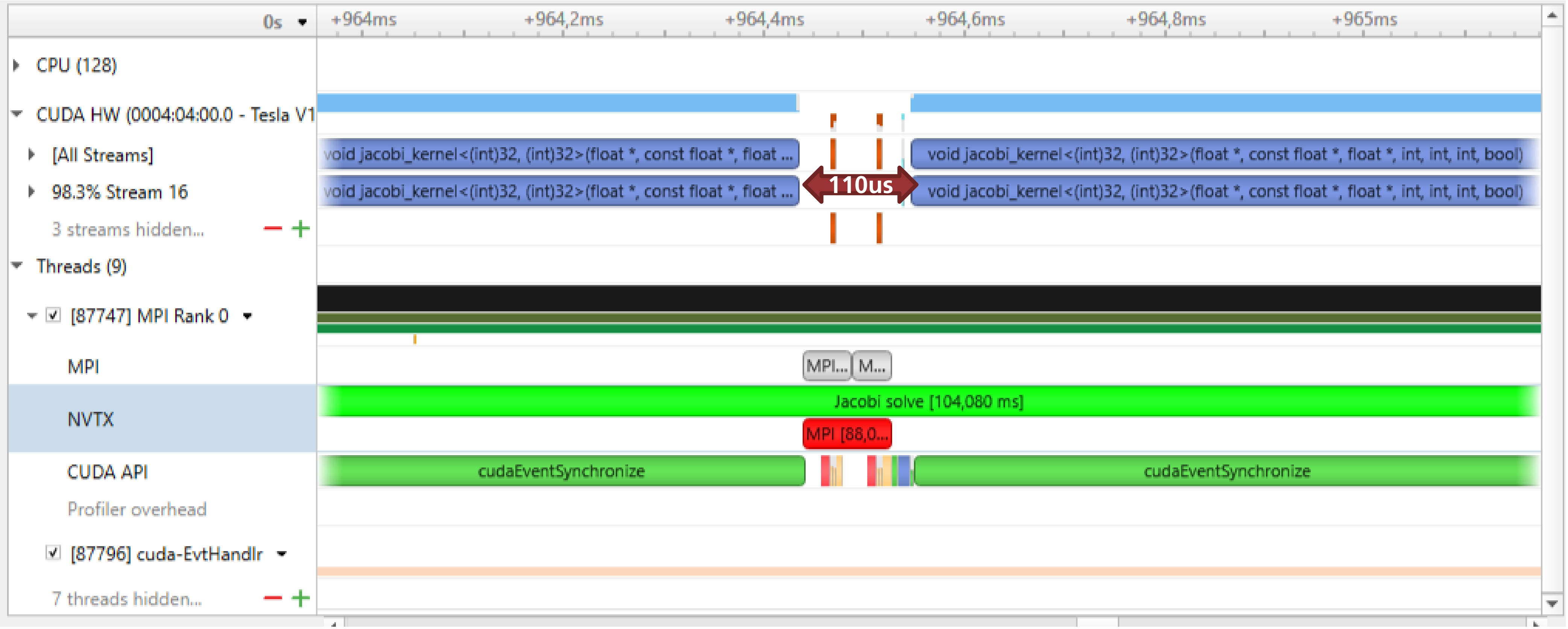
```
while(norm >1e-5 and max_iter<iter){
   compute_kernel<<<..,..,..>>>
   cudaDeviceSynchronize();
   exchange_neighbors();
   norm =compute_norm()
   iter ++
}
```

Performance: C vs. Python with 4 GPUs, MPI

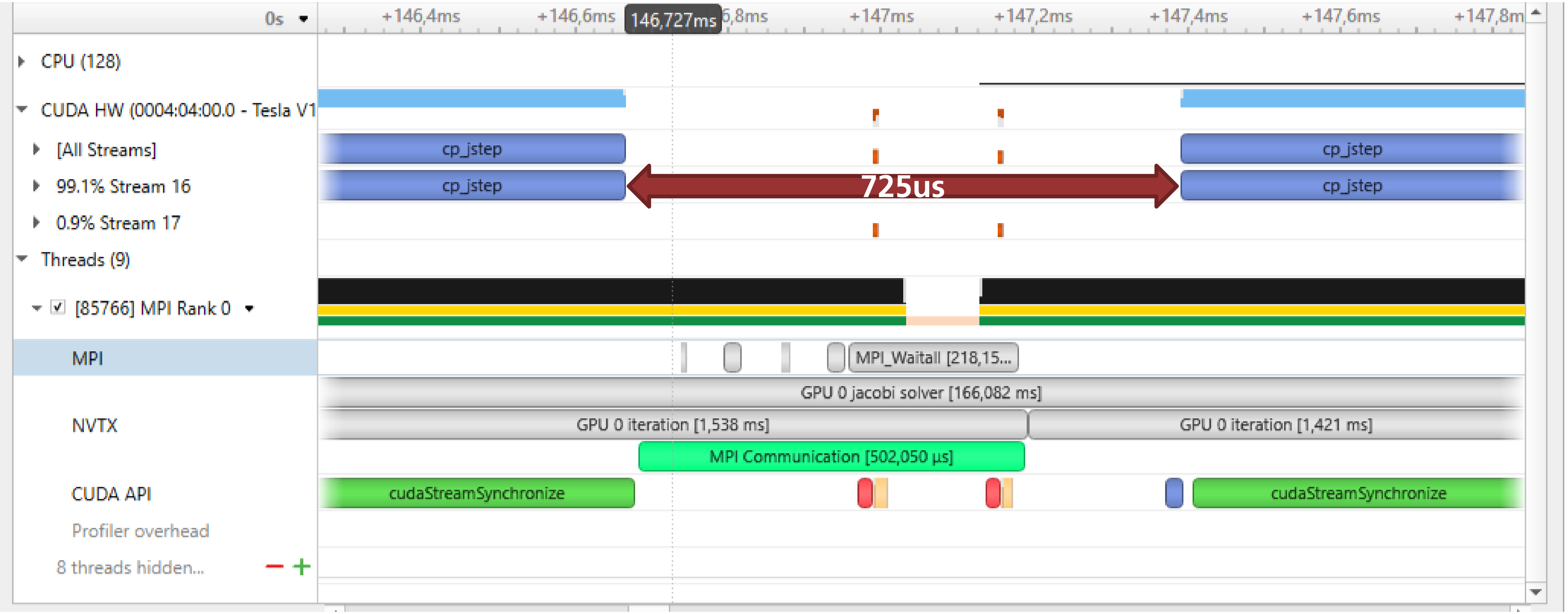




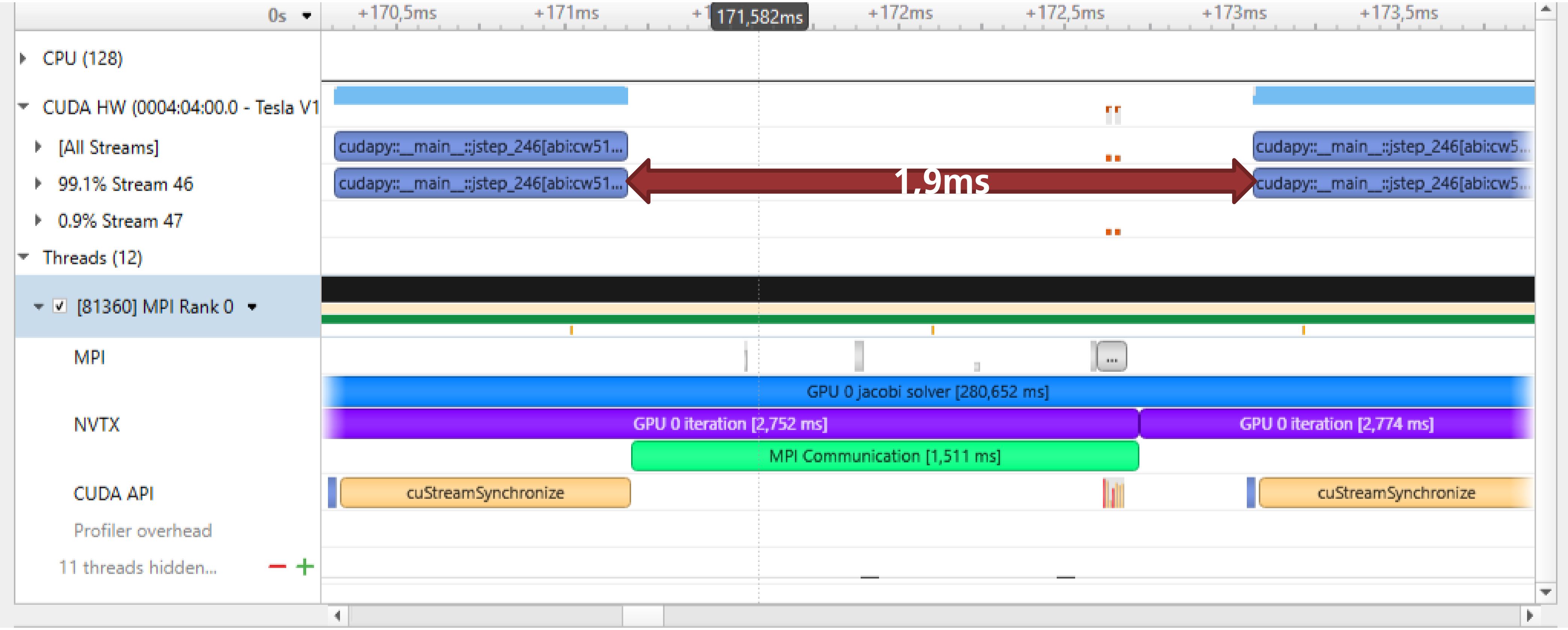
Tracing of C-CUDA+MPI



Tracing of CUPY+MPI

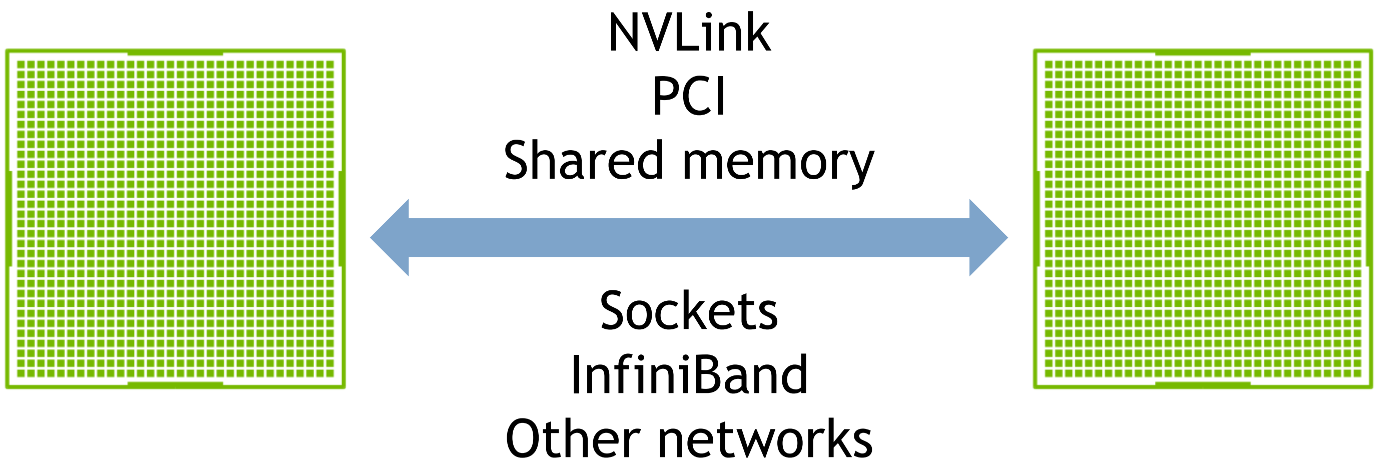


Tracing of Numba+MPI

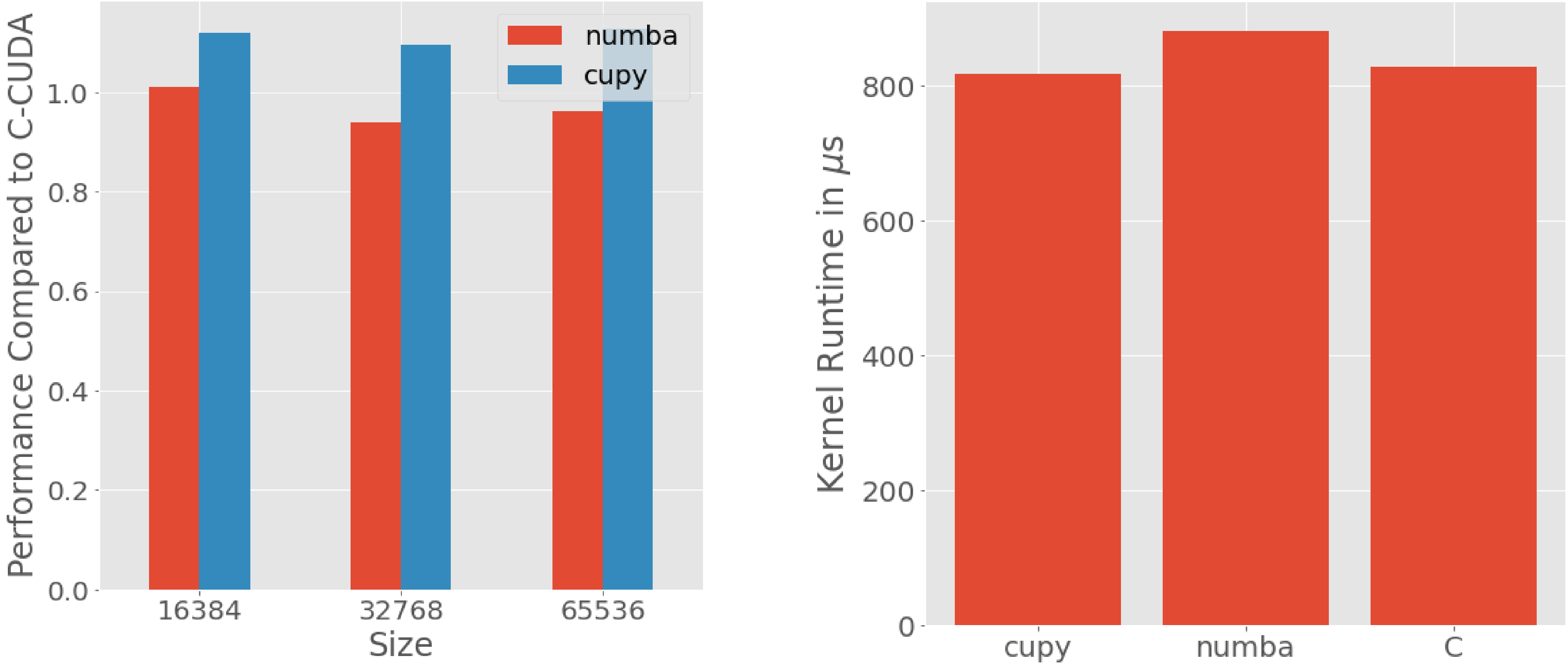


NCCL-A GPU-Aware Communication

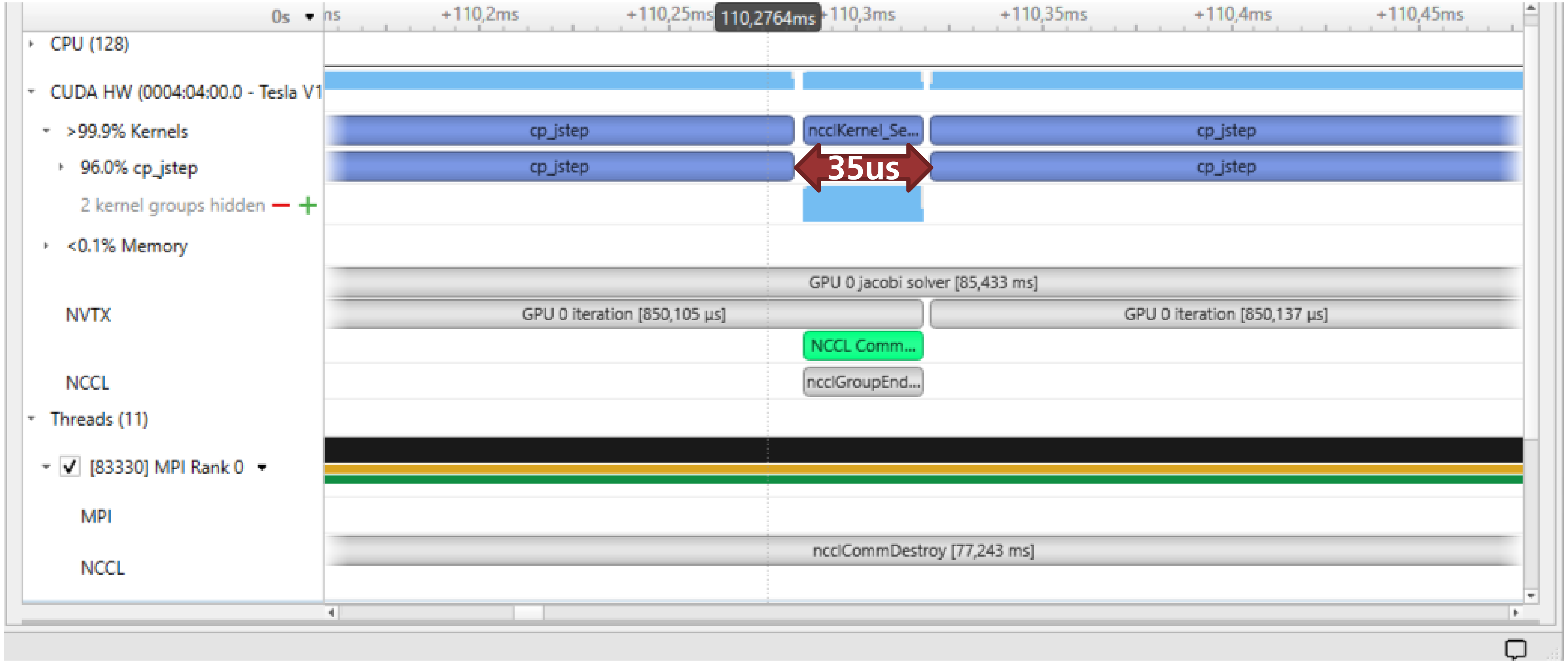
- Library for efficient communication with GPUs
- First: Collective Operations (e.g. Allreduce), as they are required for DeepLearning
- Since 2.8: Support for Send/Recv between GPUs
- Library running on GPU: Communication calls are translated to GPU a kernel (running on a stream)



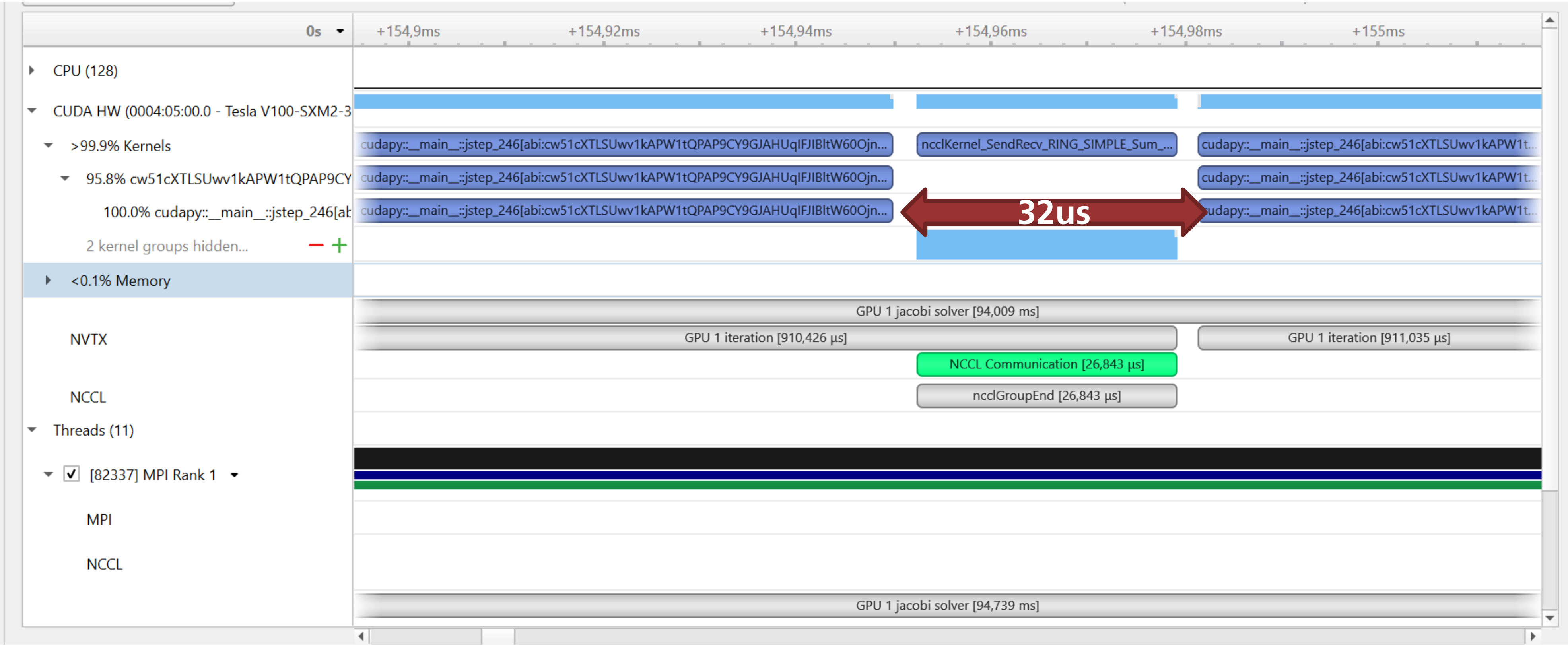
Performance: Python vs. C, 4 GPUs, NCCL



Tracing of CUPY+NCCL



Tracing of Numba+NCCL



Summary

– Python CAN be powerful for GPUs, if they are used in the "correct way"
– Best Performance is reached, if the Interpreter is "out of the loop"
– NCCL can help to do this for multi-GPU applications

Next Steps

– Better understand the problem with MPI and Numba
– Adapt the concept of "Streams" to other areas of high-performance computing with Python
– Adapt the Idea to CPUs