

Высоконагруженные Вычисления

Зинин Владимир

Что такое High Load Computations

- Класс задач, которые требуют работы с большим количеством данных
- Особенности:
 - Часто задачи связаны с большим количеством однотипных расчётов
- Типичные примеры:
 - Симуляции (инженерия, прогнозы погоды)
 - Машинное обучение



С чего начать

- Большинство кода, который мы писали, выполняется на процессоре
- Python в чистом виде достаточно медленный язык
- Хочется сохранить простоту написания и основную логику, получив выигрыш в производительности



Numba

- нумба –библиотека, предназначенная для ускорения высоконагруженных участков кода
- Большую часть её функционала представляют различные декораторы
 - jit
 - vectorize
 - stencil



@jit

- nopython
- cache
- inline
- fastmath
- error_model

```
@jit(nopython=True, fastmath=True, error_model='numpy')
def compute_fast(x):
    out = 0.
    for i in range(x.shape[0]):
        out += np.sqrt(x[i] * x[i] + 1.0) / (x[i] + 1.0)
    return out
```

```
@jit([float64(float64, float64)])
def add(x, y):
    return x + y
```

```
@jit(nopython=True)
def sqdiff(x, y):
    out = np.empty_like(x)
    for i in range(x.shape[0]):
        out[i] = (x[i] - y[i])**2
    return out
```

```
sqdiff.signatures
```

```
[(Array(float32, 1, 'C', False, aligned=True),
  Array(float32, 1, 'C', False, aligned=True)),
 (Array(float64, 1, 'C', False, aligned=True),
  Array(float64, 1, 'C', False, aligned=True))]
```

```
@jit(nopython=True, error_model='numpy')
def frac_diff_nu(x, y):
    out = np.empty_like(x)
    for i in range(x.shape[0]):
        out[i] = 2 * (x[i] - y[i]) / (x[i] + y[i])
    return out
```



Всего выделяют 8 ключевых ступеней:

1. **Analyze bytecode** - numba получает bytecode, создает CFG (control flow graph) и Data flow graph
2. **Generate IR** - Переводит байт-код в более удобное внутреннее представление numba (intermediate representation)
3. **Rewrite IR** - Переписывает IR с учетом некоторых оптимизаций
4. **Infer types** - Статически типизирует переменные где возможно.
Если тип не определяется, вызывается pyobject и происходит откат в object mode
5. **Rewrite typed IR** - Переписывает типизированное промежуточное представление
6. **Perform automatic parallelization** - Автоматическая параллелизация циклов (parfor)
7. **Generate LLVM IR** - Создание LLVM промежуточного представления
8. **Generate machine code** - Генерация нативного машинного кода

Python код → Bytecode → Numba IR → Typed IR → Optimized IR → LLVM IR → Machine Code

```
@jit
def LightFunction(x, y):
    if y == 0:
        return 0
    return x / y
```

```
CFG adjacency lists:
{0: [16, 18], 16: [], 18: []}
CFG dominators:
{0: {0}, 16: {16, 0}, 18: {0, 18}}
CFG post-dominators:
{0: {0}, 16: {16}, 18: {18}}
CFG back edges: []
CFG loops:
{}
CFG node-to-loops:
{0: [], 16: [], 18: []}
CFG backbone:
{0}
```

```
-----IR DUMP: LightFunction-----
label 0:
    x = arg(0, name=x)          ['x']
    y = arg(1, name=y)          ['y']
    $const6.1.1 = const(int, 0)  ['$const6.1.1']
    $8compare_op.2 = y == $const6.1.1 ['$8compare_op.2', '$const6.1.1', 'y']
    bool12 = global(bool: <class 'bool'>)  ['bool12']
    $12pred = call bool12($8compare_op.2, func=bool12, args=(Var($8compare_op.2, 1475666804.py:3),), kws
                                branch $12pred, 16, 18  ['$12pred']

label 16:
    $const16.0 = const(int, 0)  ['$const16.0']
    $16return_const.1 = cast(value=$const16.0) ['$16return_const.1', '$const16.0']
    return $16return_const.1   ['$16return_const.1']

label 18:
    $binop_truediv20.2 = x / y      ['$binop_truediv20.2', 'x', 'y']
    $24return_value.3 = cast(value=$binop_truediv20.2) ['$24return_value.3', '$binop_truediv20.2']
    return $24return_value.3       ['$24return_value.3']
```

Что ещё есть?

```
@vectorize([float64(float64, float64)])  
def add(x, y):  
    return x + y
```

```
@stencil(neighborhood = ((-1, 1), (-1, 1)), standard_indexing="weights",)  
def smooth_2d(a, weights):  
    res = 0.  
    for i in range(-1, 2):  
        for j in range(-1, 2):  
            res += weights[i, j] * a[i, j]  
    return res
```

```
@guvectorize([(int64[:], int64, int64[:])], '(n),()->())'  
def g(x, y, res):  
    acc = 0  
    for i in range(x.shape[0]):  
        acc += x[i] + y  
    res[0] = acc
```

```
@jitclass(spec)  
class Bag(object):  
    def __init__(self, value):  
        self.value = value  
        self.array = np.zeros(value, dtype=np.float32)
```

Как писать быстрый код

- Страйтесь использовать как можно меньше ветвлений, встроенных типов
- Нужно избегать динамической типизации
- Чем код проще, тем он быстрее



Цена ускорения

- Numba переводит python код в более оптимизированный машинный код
- Убирается динамическая типизация
- Применяются оптимизации(parallel for, vectorization, SIMD)
- Но, так как это всё работа LLVM, то программисту нужно писать такой код, что система сможет распознать место к которому можно применить оптимизацию



GPU vs CPU

- Для определенных задач можно использовать GPU
- Для чего они подходят:
 - Матричные операции
 - Обработка изображений
 - Симуляция взаимодействия частиц



GPU

- Видеокарты по своей сути — большое количество ядер, настроенных на выполнение определенного типа задач
- Особенность заключается в том, что эти устройства по-другому работают с памятью. И для максимальной производительности нужно это учитывать.



Распределение потоков

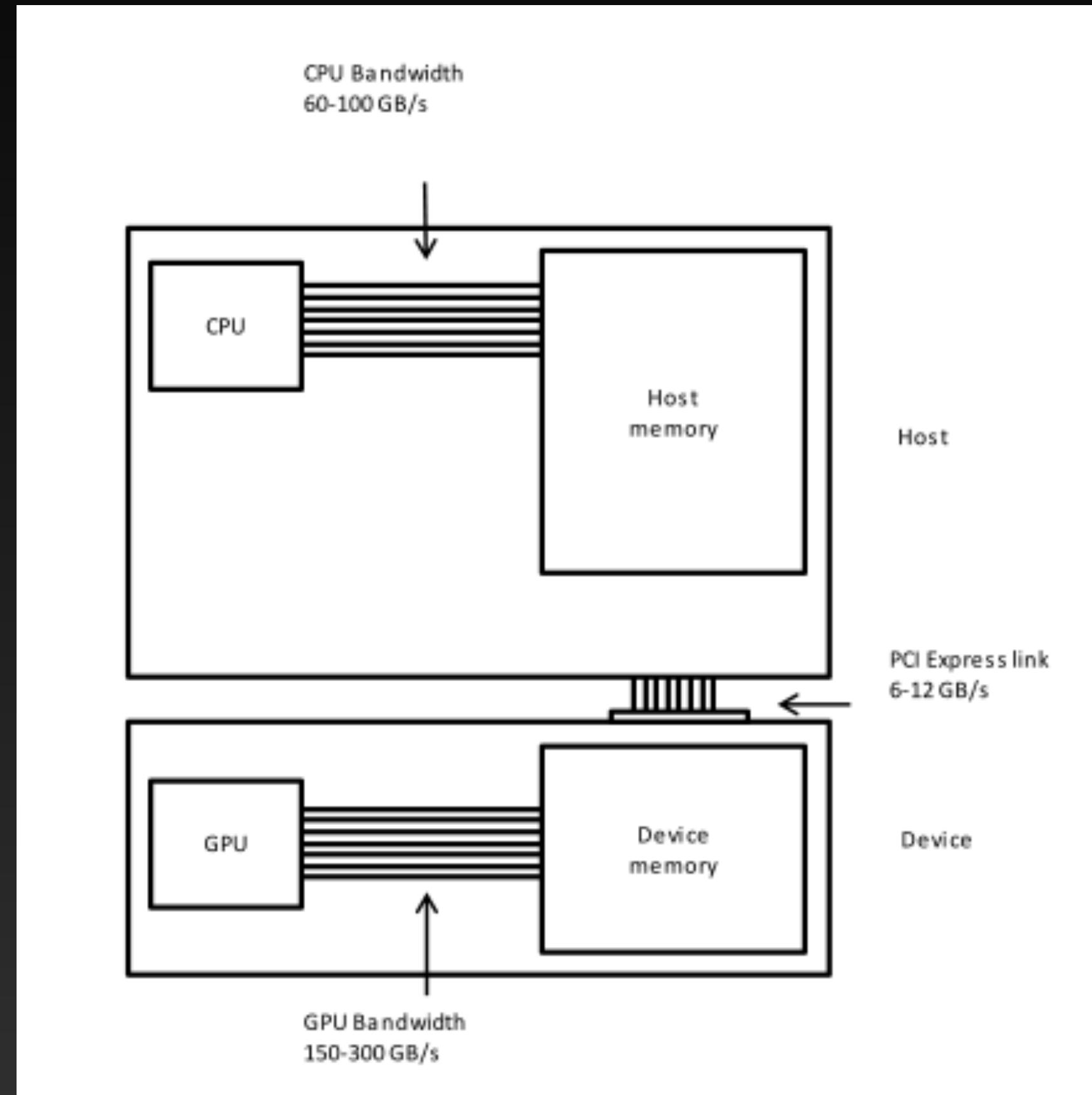
- Каждая задача состоит из ПОТОКОВ
- Потоки объединяются в блоки, каждый блок отправляется на отдельный SM
- Внутри блока команды бьются на warp(блок по 32) — в нём исполняются одновременно

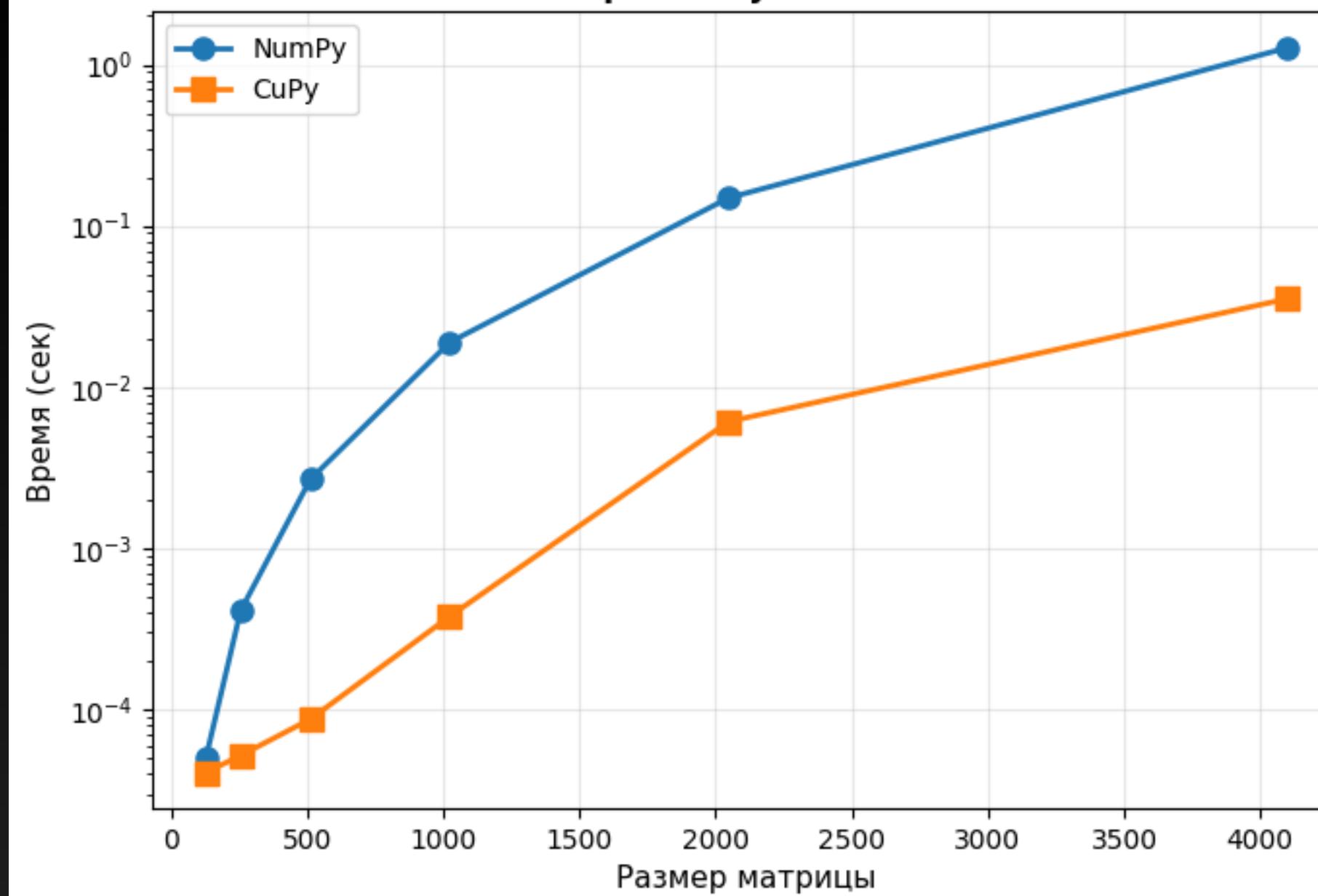
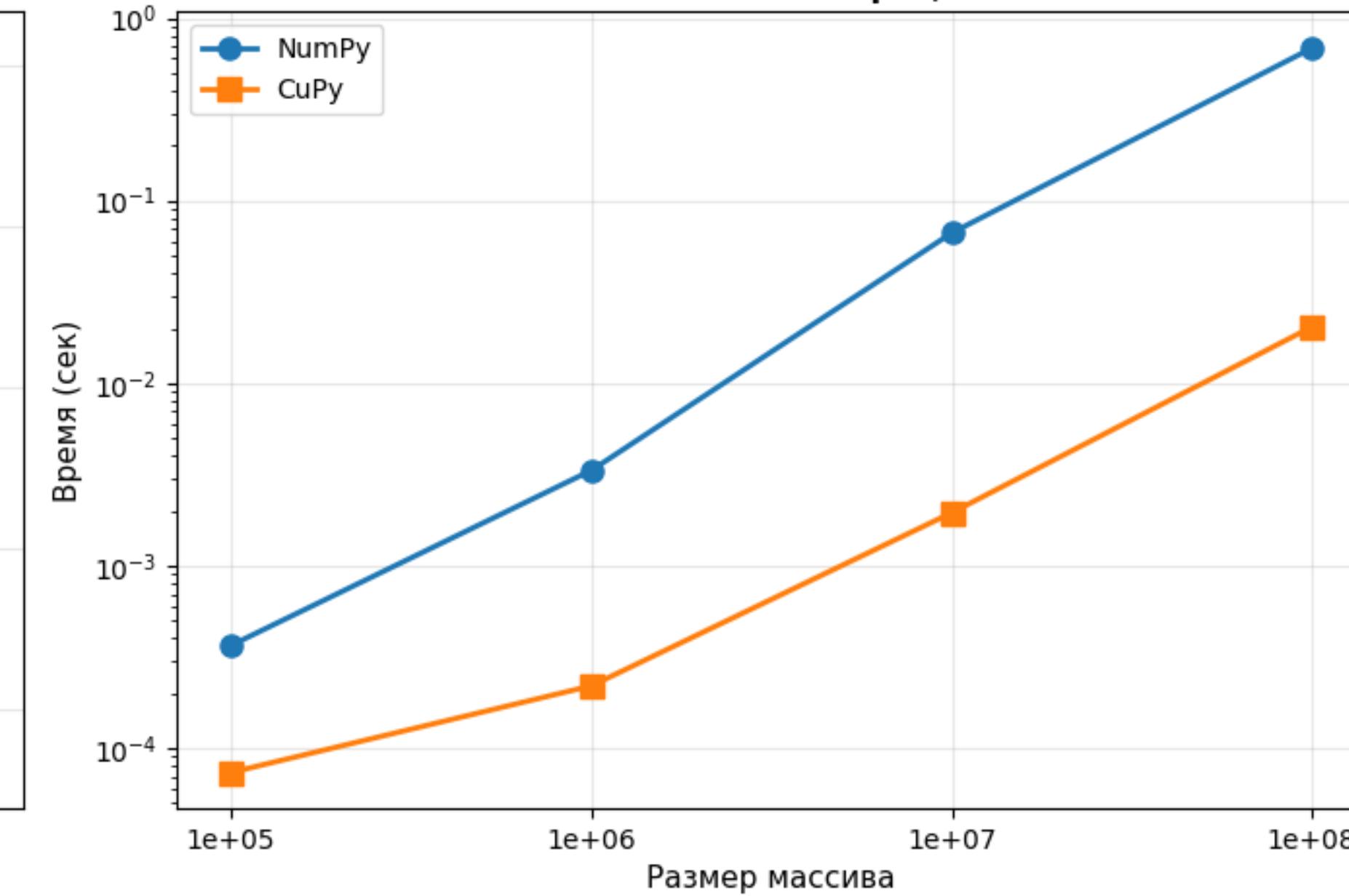
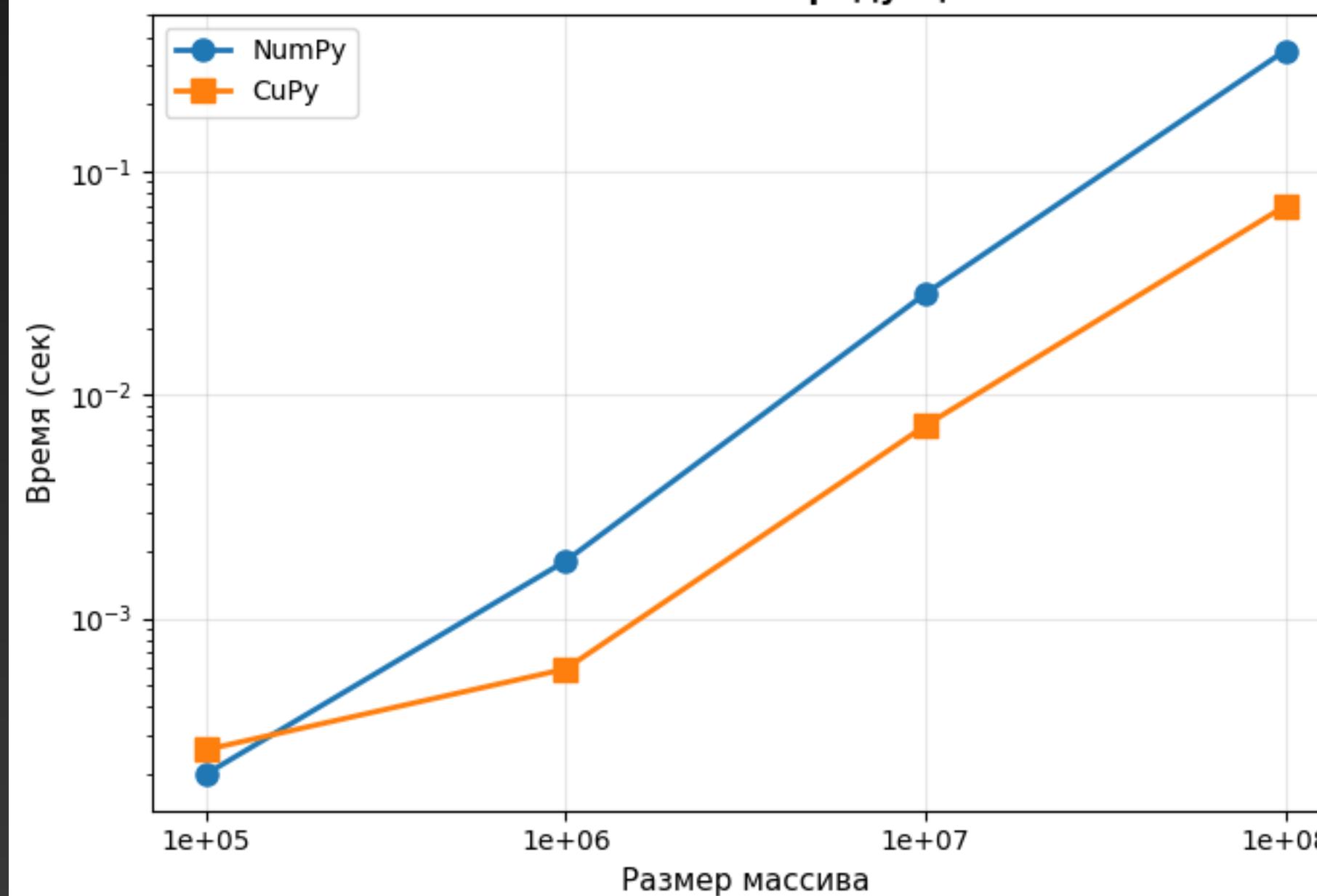
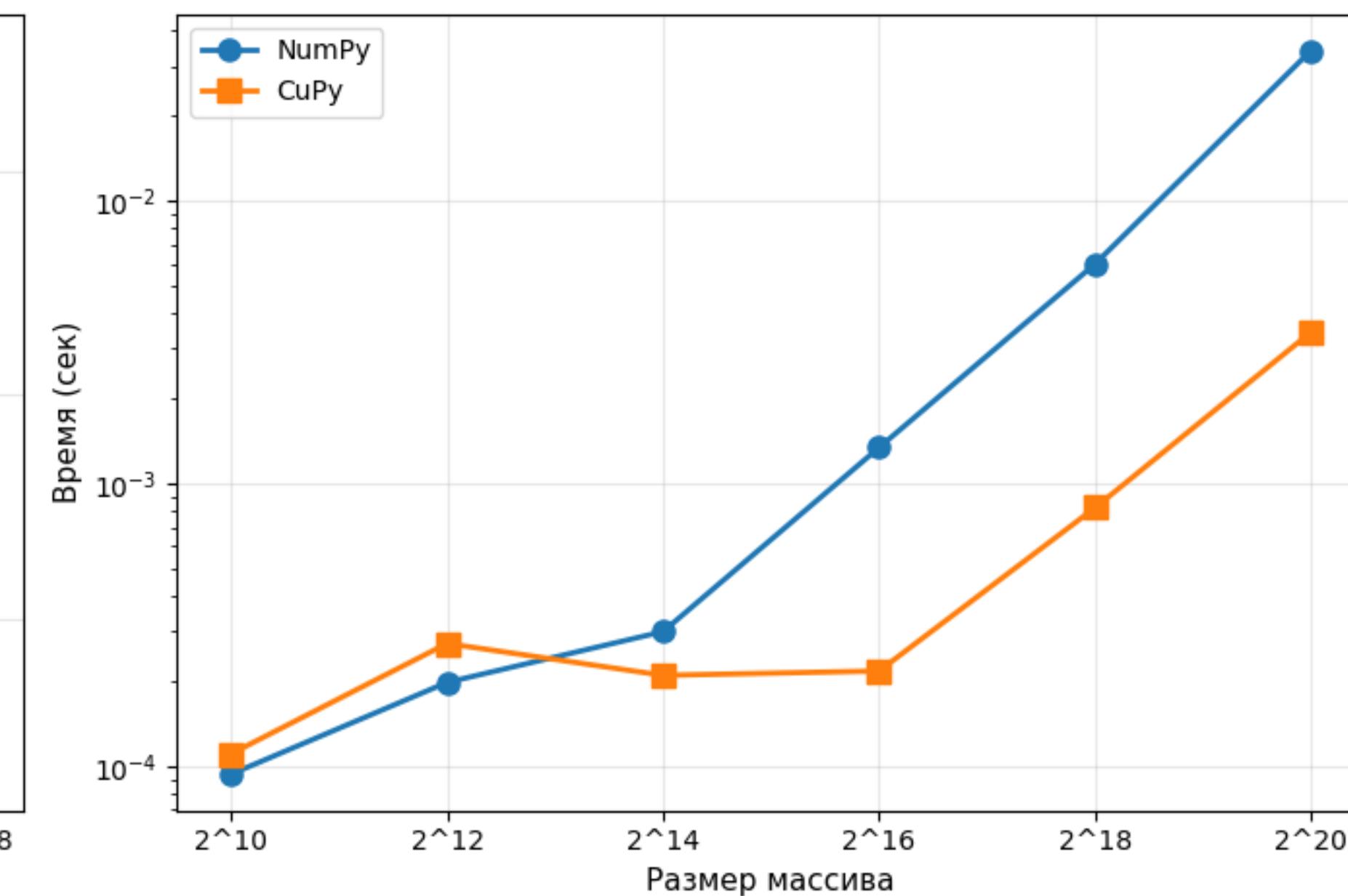


Работа с памятью

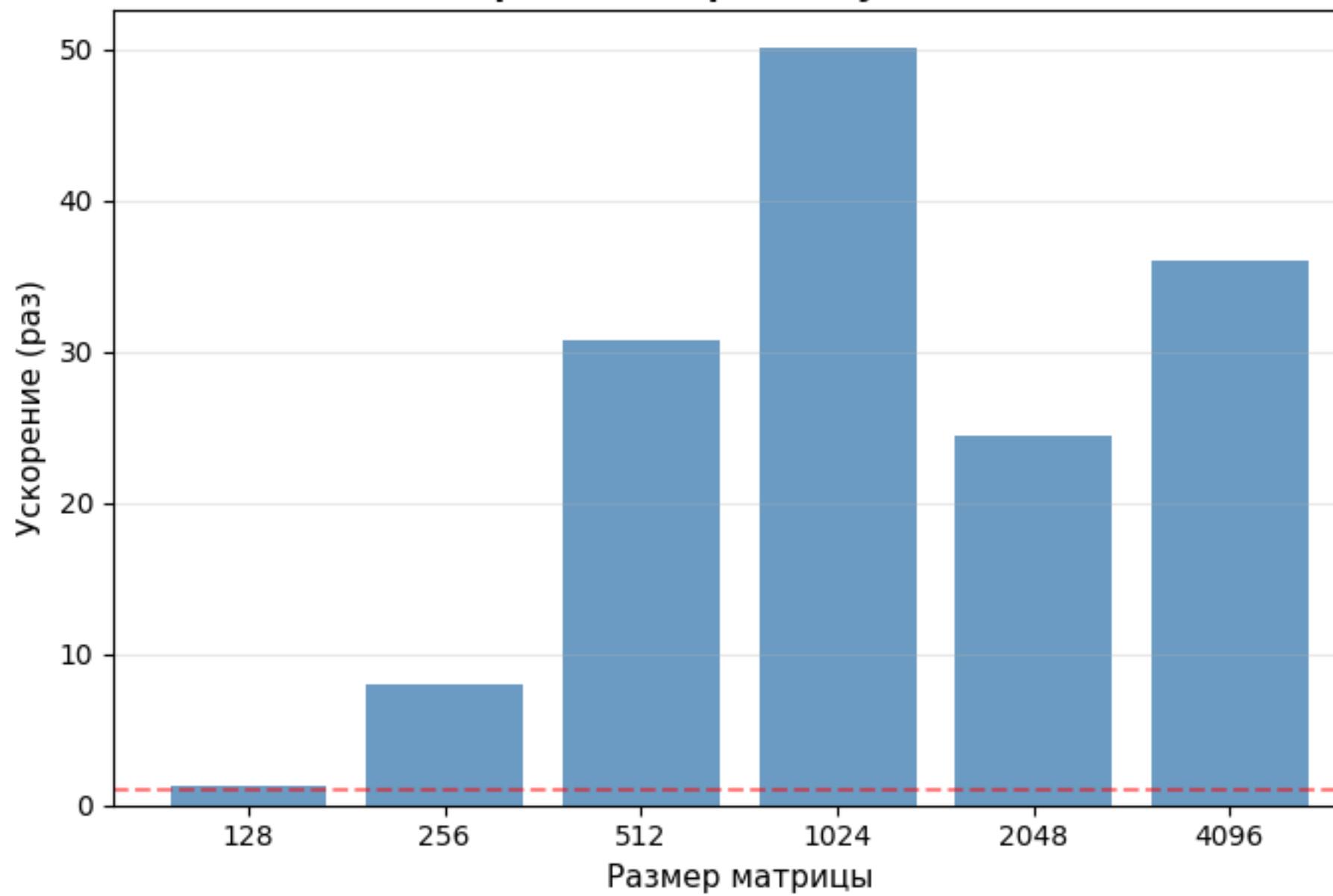
- Нам важно понимать, как устроена работа с warp'ом
- Каждый массив, который мы хотим использовать в вычислениях должен быть специально передан на GPU

Тип памяти	Описание	Время обращения, тактов
DRAM	память компьютера	2000+
Global memory(видео память)	память, общая для всего GPU, её может быть порядка нескольких гигабайт	200-400
L2	служит для обмена L1 с глобальной памятью	100-200
Shared memory(L1)	общая для блока потоков	20-30
Register File	память, выделенная под выполнение конкретного warp	1-2
L0	хранилище команд	1-2

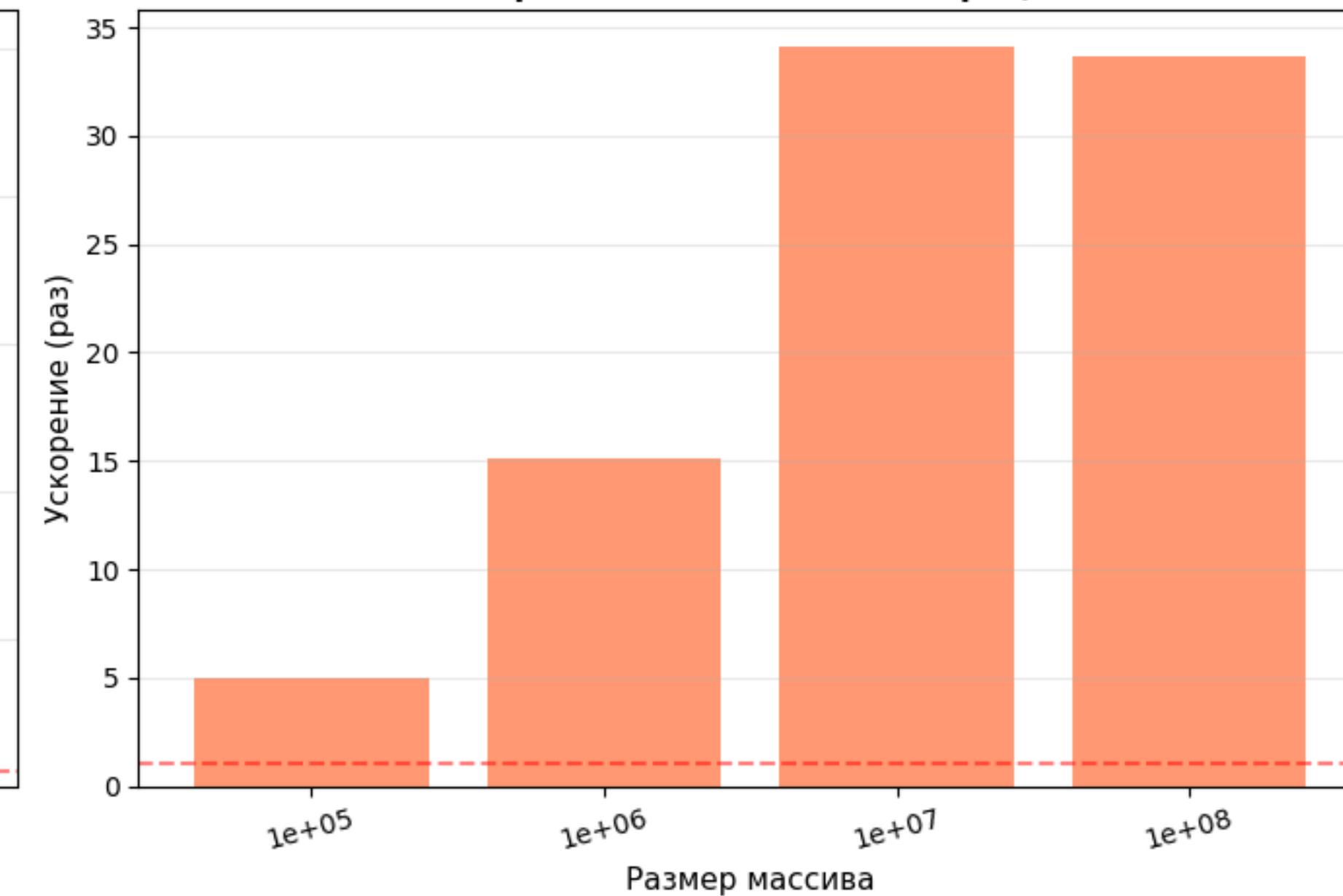


Матричное умножение**Элементные операции****Статистические редукции****FFT**

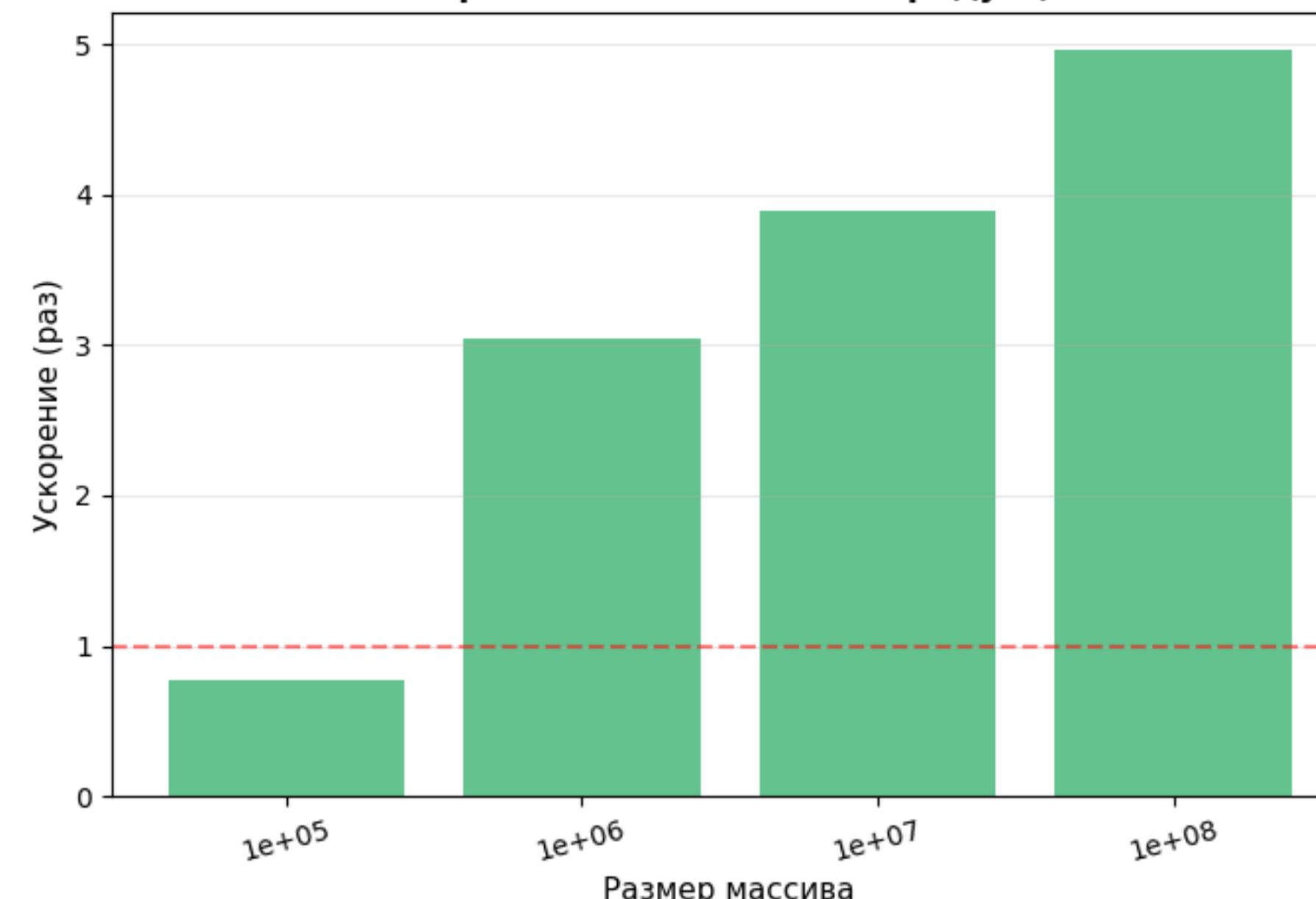
Ускорение: Матричное умножение



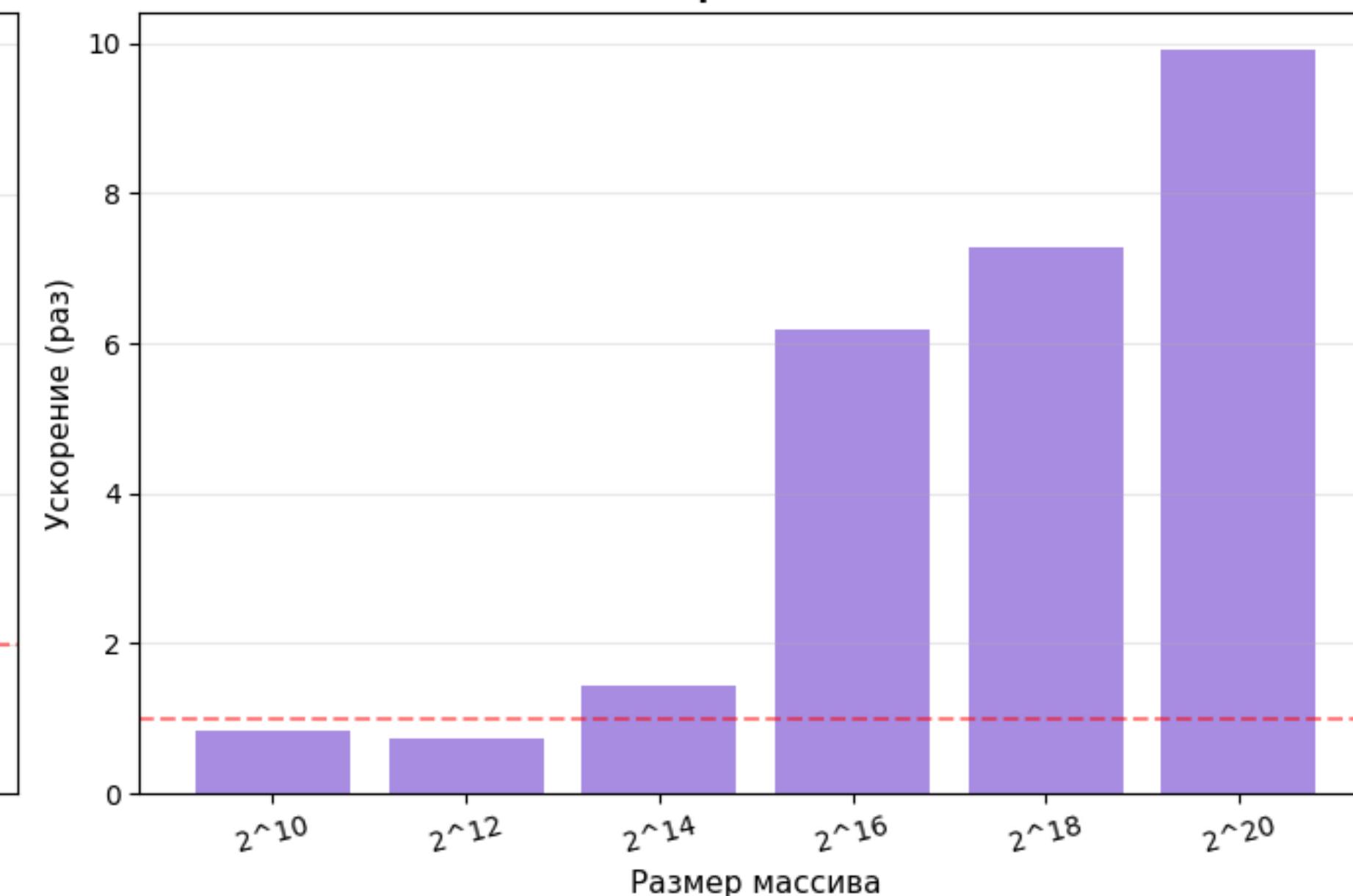
Ускорение: Элементные операции



Ускорение: Статистические редукции



Ускорение: FFT



numba.cuda

- Довольно бесшовный переход от одного исполняющего устройства к другому
- За счёт декораторов позволяет писать python код, который затем работает на GPU



```
from numba import cuda

@cuda.jit
def add_kernel(x, y, out):
    idx = cuda.grid(1)
    if idx < x.size:
        out[idx] = x[idx] + y[idx]
```

```
n = 1000000
x = np.ones(n)
y = np.ones(n) * 2
out = np.zeros(n)

threads_per_block = 256
blocks_per_grid = (n + threads_per_block - 1) // threads_per_block

add_kernel[blocks_per_grid, threads_per_block](x, y, out)
print(out[:10])
```

```
[3. 3. 3. 3. 3. 3. 3. 3. 3.]
```

Как стоит писать код внутри функции

1. Каждый поток должен работать с одним элементом данных
2. Избегайте ветвлений (if/else)
3. Используйте локальные переменные вместо глобальной памяти
4. Минимизируйте обращения к глобальной памяти
5. Используйте shared memory для кэширования данных

Как работать с данными

```
@cuda.jit
def test_func(out):
    idx = cuda.grid(1)
    tid = cuda.threadIdx.x
    bid = cuda.blockIdx.x
    bdim = cuda.blockDim.x
    gdim = cuda.gridDim.x

    if idx < out.size:
        out[idx] = tid * 1000 + bid * 100 + bdim * 10 + gdim

out = np.zeros(10)
test_func[2, 4](out)
print(out)
```

```
42. 1042. 2042. 3042. 142. 1142. 2142. 3142. 0. 0.]
```

- `cuda.grid(1)` - получить глобальный индекс потока (1D)
- `cuda.grid(2)` - получить глобальные индексы (x, y) для 2D
- `cuda.threadIdx.x`, `cuda.threadIdx.y` - индекс потока в блоке
- `cuda.blockIdx.x`, `cuda.blockIdx.y` - индекс блока
- `cuda.blockDim.x`, `cuda.blockDim.y` - размер блока
- `cuda.gridDim.x`, `cuda.gridDim.y` - размер сетки

Работа с памятью

```
shared = cuda.shared.array(256, dtype=np.float32)

x_host = np.ones(1000) # массив в RAM
x_device = cuda.to_device(x_host) # перенесли на GPU

nulls_device = cuda.device_array(1000) # пустой массив в GPU
```

```
arr = np.zeros(5)
func[1, 5](arr)
```

то numba просто рассахаривает код в что-то типа:

```
arr = np.zeros(5)
gpu_zeros = cuda.to_device(arr)
func[1, 5](gpu_zeros)
```

```
@cuda.jit
def kernel(x, y):
    idx = cuda.grid(1)
    if idx < x.size:
        y[idx] = x[idx] * 2

# Создаём два потока для асинхронного выполнения
stream1 = cuda.stream()
stream2 = cuda.stream()

n = 1000000
x1 = cuda.to_device(np.ones(n))
y1 = cuda.device_array(n)
x2 = cuda.to_device(np.ones(n) * 2)
y2 = cuda.device_array(n)

# Запускаем в разных потоках параллельно
threads = 256
blocks = (n + threads - 1) // threads
kernel[blocks, threads, stream1](x1, y1)
kernel[blocks, threads, stream2](x2, y2)

# Синхронизируем оба потока
stream1.synchronize() # Ждём завершения stream1
stream2.synchronize() # Ждём завершения stream2

# Копируем результаты обратно
result1 = y1.copy_to_host()
result2 = y2.copy_to_host()

print("Stream1 result:", result1[:5])
print("Stream2 result:", result2[:5])
```

```
Stream1 result: [2. 2. 2. 2. 2.]
Stream2 result: [4. 4. 4. 4. 4.]
```

ПОТОКИ

- Мы можем использовать потоки, чтобы уменьшить задержки, связанные с загрузкой данных в память устройства

CuPy



CuPy

- Библиотека старается сделать взаимодействие с устройством максимально простым
- Большинство привычных функций питру уже оптимизированы
- Нам не нужно задумываться над распределением потоков

```
# Математические операции
np_x = np.linspace(0, 2*np.pi, 1000000)
cp_x = cp.linspace(0, 2*cp.pi, 1000000)

print('NumPy sin:')
%timeit np.sin(np_x)

print('CuPy sin:')
%timeit cp.sin(cp_x); cp.cuda.Stream.null.synchronize()
```

NumPy sin:
13 ms ± 2.11 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
CuPy sin:
186 µs ± 2.16 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)

Категория	NumPy	CuPy
Создание массивов		
Массив из списка	np.array([1,2,3])	cp.array([1,2,3])
Нули	np.zeros((3,3))	cp.zeros((3,3))
Единицы	np.ones((3,3))	cp.ones((3,3))
Единичная матрица	np.eye(3)	cp.eye(3)
Диапазон	np.arange(10)	cp.arange(10)
Линейное пространство	np.linspace(0,1,100)	cp.linspace(0,1,100)
Случайные числа	np.random.rand(3,3)	cp.random.rand(3,3)
Пустой массив	np.empty((3,3))	cp.empty((3,3))
Заполнение значением	np.full((3,3), 5)	cp.full((3,3), 5)

```
# Без fuse – каждая операция запускает отдельное ядро
def compute_without_fuse(x, y, z):
    return (x * y + z) ** 2

# С fuse – все операции в одном ядре
@cp.fuse()
def compute_with_fuse(x, y, z):
    return (x * y + z) ** 2

x = cp.random.rand(1000000, dtype=cp.float32)
y = cp.random.rand(1000000, dtype=cp.float32)
z = cp.random.rand(1000000, dtype=cp.float32)

print('Без fuse:')
%timeit compute_without_fuse(x, y, z); cp.cuda.Stream.null.synchronize()

print('С fuse:')
%timeit compute_with_fuse(x, y, z); cp.cuda.Stream.null.synchronize()
```

Без fuse:

157 μ s \pm 3.49 μ s per loop (mean \pm std. dev. of 7 runs, 10000 loops each)

С fuse:

82.7 μ s \pm 2.15 μ s per loop (mean \pm std. dev. of 7 runs, 10000 loops each)

Работа с памятью

Для переноса на GPU используем функцию cp.asarray

```
cpu_array = np.ones((1000, 1000), dtype=np.float32)
gpu_from_cpu = cp.asarray(cpu_array)
```

Наоборот cp.asarray или cp.array.get

```
back_to_cpu = cp.asarray(gpu_from_cpu)
```

Kernels

```
# Простая поэлементная операция
multiply_add = cp.ElementwiseKernel(
    'float32 x, float32 y, float32 z', # входные параметры
    'float32 out', # выходной параметр
    'out = x * y + z', # операция
    'multiply_add' # имя ядра
)

x = cp.random.rand(1000000, dtype=cp.float32)
y = cp.random.rand(1000000, dtype=cp.float32)
z = cp.random.rand(1000000, dtype=cp.float32)

result = multiply_add(x, y, z)
print(result[:5])
```

```
[1.0817041 0.22785139 1.1465722 0.89147085 0.5799721 ]
```

```
preamble = r"""
__device__ __forceinline__ float sqrf(float x) {
    return x * x;
}

square = cp.ElementwiseKernel(
    in_params='float32 x, float32 lo, float32 hi',
    out_params='float32 y',
    operation=r"""
        y = sqrf(x);
    """,
    name='clamp_square',
    preamble=preamble,
)
```

```
# сумма квадратов
sum_of_squares = cp.ReductionKernel(
    'float32 x', # входной тип
    'float32 out', # выходной тип
    'x * x', # map: что делаем с каждым элементом
    'a + b', # reduce: как объединяем результаты
    'out = a', # постобработка
    '0', # начальное значение
    'sum_of_squares' # имя
)

x = cp.arange(1000000, dtype=cp.float32)
result = sum_of_squares(x)

print(f'Сумма квадратов: {result}')
print(f'Проверка: {cp.sum(x**2)}')
```

```
Сумма квадратов: 3.333328241594204e+17
Проверка: 3.333328241594204e+17
```

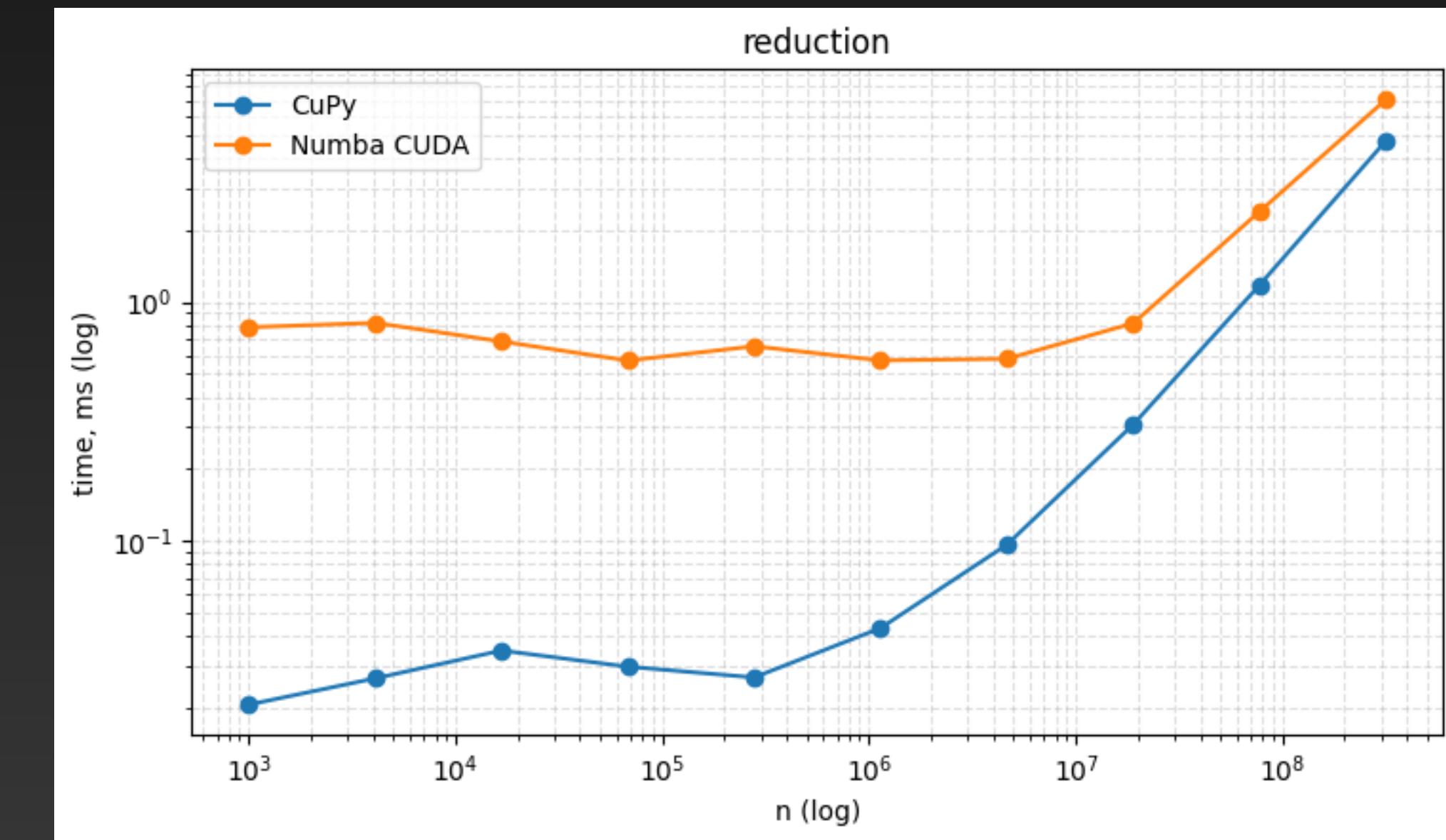
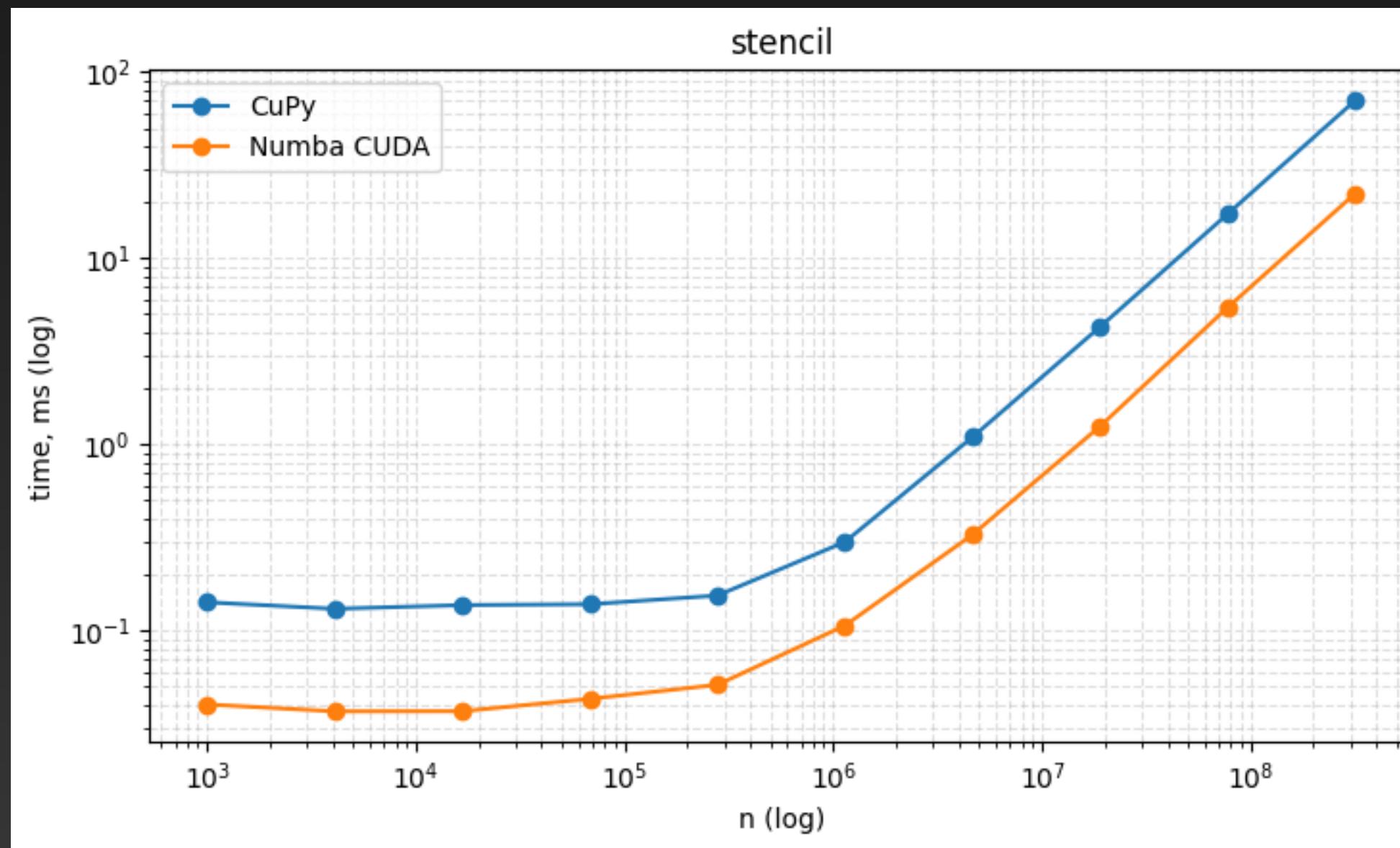
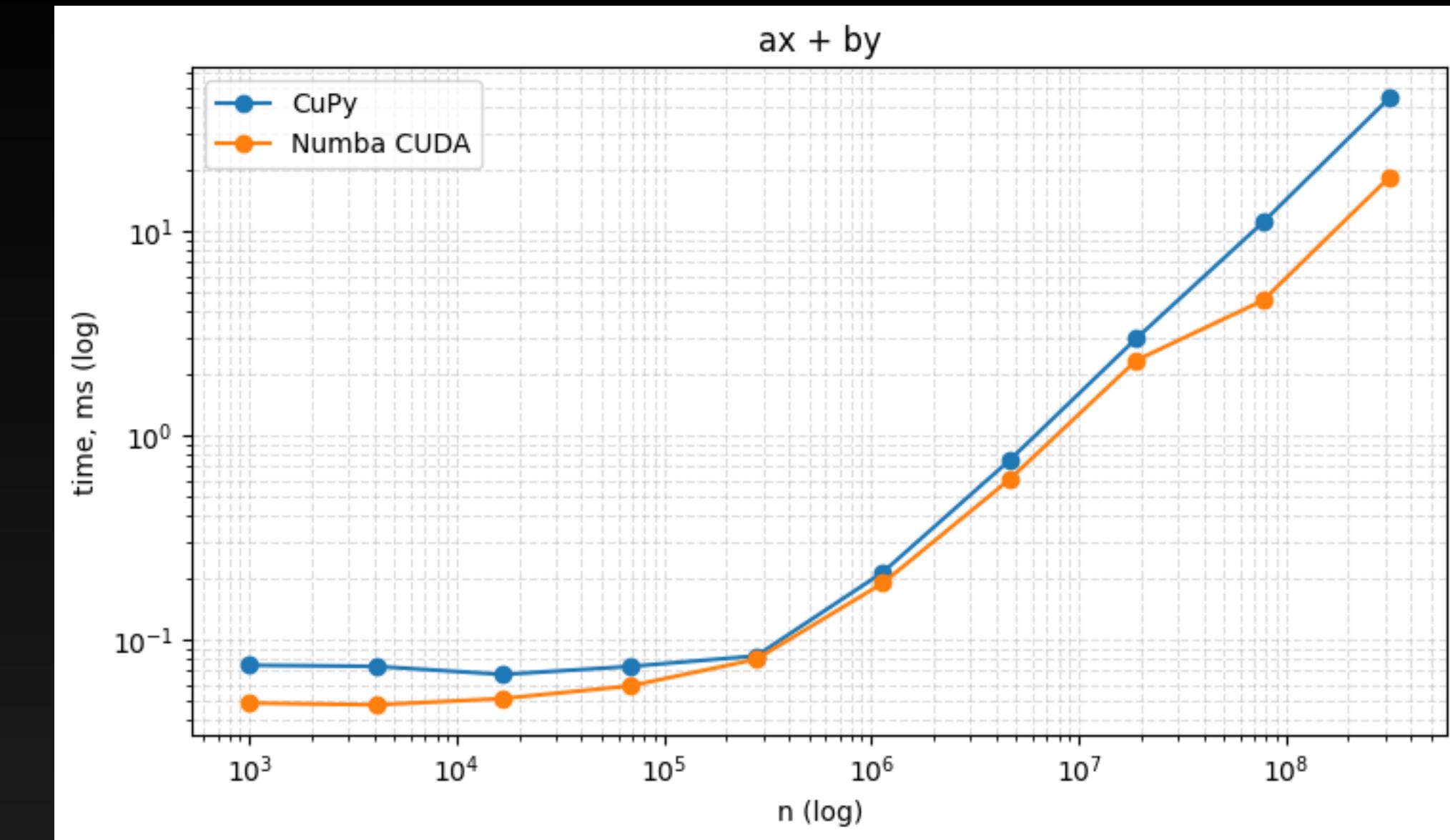
Kernels

- Дают нам возможность использовать CUDA C/C++ код для написания функций
- Работают ближе к железу
- Требуют дополнительных знаний

```
add_kernel = cp.RawKernel(r'''
extern "C" __global__
void my_add(const float* x1, const float* x2, float* y) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    y[tid] = x1[tid] + x2[tid];
}
''', 'my_add')
x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
y = cp.zeros((5, 5), dtype=cp.float32)
add_kernel((5,), (5,), (x1, x2, y)) # grid, block and arguments
y

array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

CuPy vs Numba.cuda



Итоги

- Разобрали основные библиотеки, которые могут помочь с ускорением кода
- Рассмотрели, чем GPU похож на CPU, в чем особенности работы с ним

