OpenAI

Block-based GPU Programming with Triton

PHIL TILLET
OPENAI

WHY TRITON ?

CUDA: FLEXIBILITY AT THE COST OF SIMPLICITY

- Pros: developers can do whatever the heck they want:
  - can squeeze the last bits of performance
  - can use whatever data-structure you want

- Cons: developers can do whatever the heck they want:
  - performance optimization is cumbersome and time-consuming
  - codebases are complex and hard to maintain
  - algorithms are opaque to researchers

GRAPH COMPILERS: SIMPLICITY AT THE COST OF FLEXIBILITY

- Pros: very fast iteration speed for researchers:
    - can prototype certain types of ideas *very* quickly

- Cons:
    - can’t represent certain types of ideas:
        - custom data-structures (e.g., linked lists, trees, etc.)
        - in-operator control flow (e.g., loops, branches, early exit, etc.)
    - code generation is a difficult problem:
        - heavy use of templates and pattern-matching
        - lots of performance cliffs

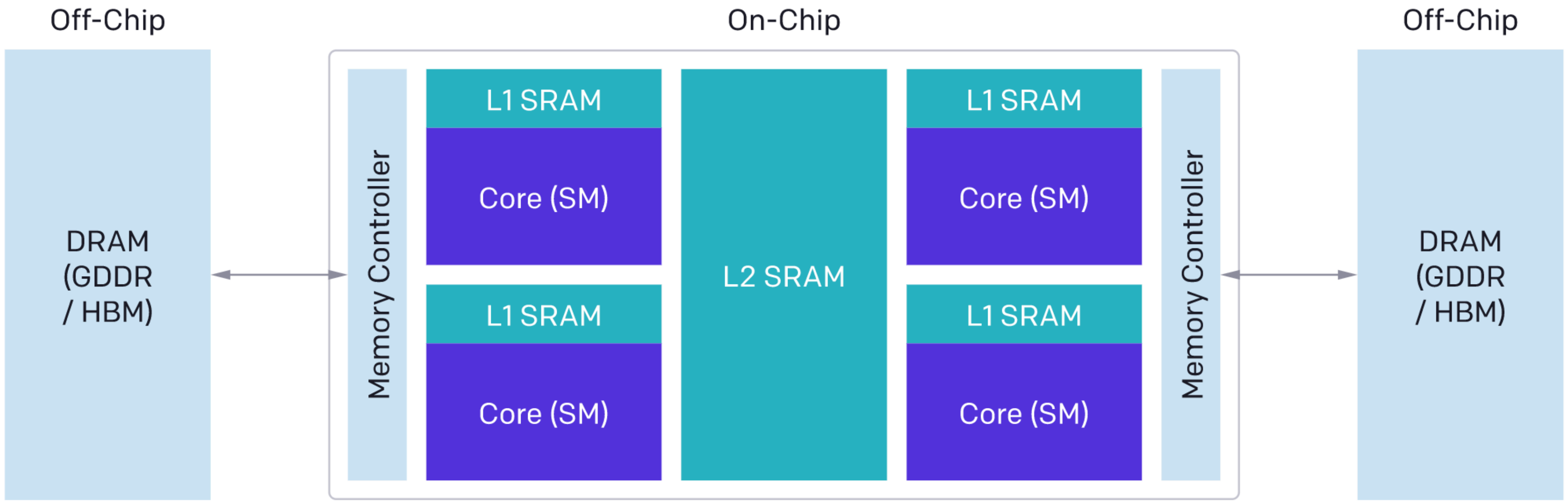
TRITON: MEET ME IN THE MIDDLE ?

- Pro: simpler than CUDA; more expressive than graph compilers:
    - do in ~days what would take ~weeks (or months~) in CUDA
    - can write algorithms out-of-scope of graph compilers (e.g., radix sort, tree walk, etc.)
    - code can still be understood/modified/written by researchers
    - performance-portable across and within different vendors
    - code will be at least as fast as the best graph compilers

- Con: less expressive than CUDA; more complicated than graph compilers;
    - code will be at most as fast as the best CUDA
    - you will iterate slower than if you use a graph compiler when applicable

WHAT IS TRITON ?

MACHINE MODEL



PROGRAMMING MODEL

Users define tensors (i.e., *blocks* of data) in **SRAM**, and modify them using **torch-like operators**

Embedded in Python

Like in Numba, kernels are defined in Python using the triton.jit decorator

Pointer arithmetics

Users can construct tensors of pointers and dereference them element-wise

Shape Constraints

Triton tensors must have power-of-two number of elements along each dimension

VECTOR ADDITION

VECTOR ADDITION

Let us start with a element-wise kernel,
on a single core, without bounds-checking

```
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, 1024)

    # create 1024 pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load 1024 elements of X, Y, Z
    x = tl.load(x_ptrs)
    y = tl.load(y_ptrs)
    # do computations
    z = x + y
    # write-back 1024 elements of X, Y, Z
    tl.store(z_ptrs, z)

N = 1024
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = (1, )
_add[grid](z, x, y, N)
```

VECTOR ADDITION (REVISITED)

A few lines of code can be added to handle bounds-checking and parallelization.

Different instances of _add are mapped to different thread blocks

```
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N):
    # same as torch.arange
    offsets = tl.arange(0, 1024)
    offsets += tl.program_id(0)*1024
    # create 1024 pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load 1024 elements of X, Y, Z
    x = tl.load(x_ptrs, mask=offset<N)
    y = tl.load(y_ptrs, mask=offset<N)
    # do computations
    z = x + y
    # write-back 1024 elements of X, Y, Z
    tl.store(z_ptrs, z, mask=offset<N)

N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = (triton.cdiv(N, 1024), )
_add[grid](z, x, y, N)
```

VECTOR ADDITION (RE-REVISITED)

A few lines of code can be added to parameterize block sizes

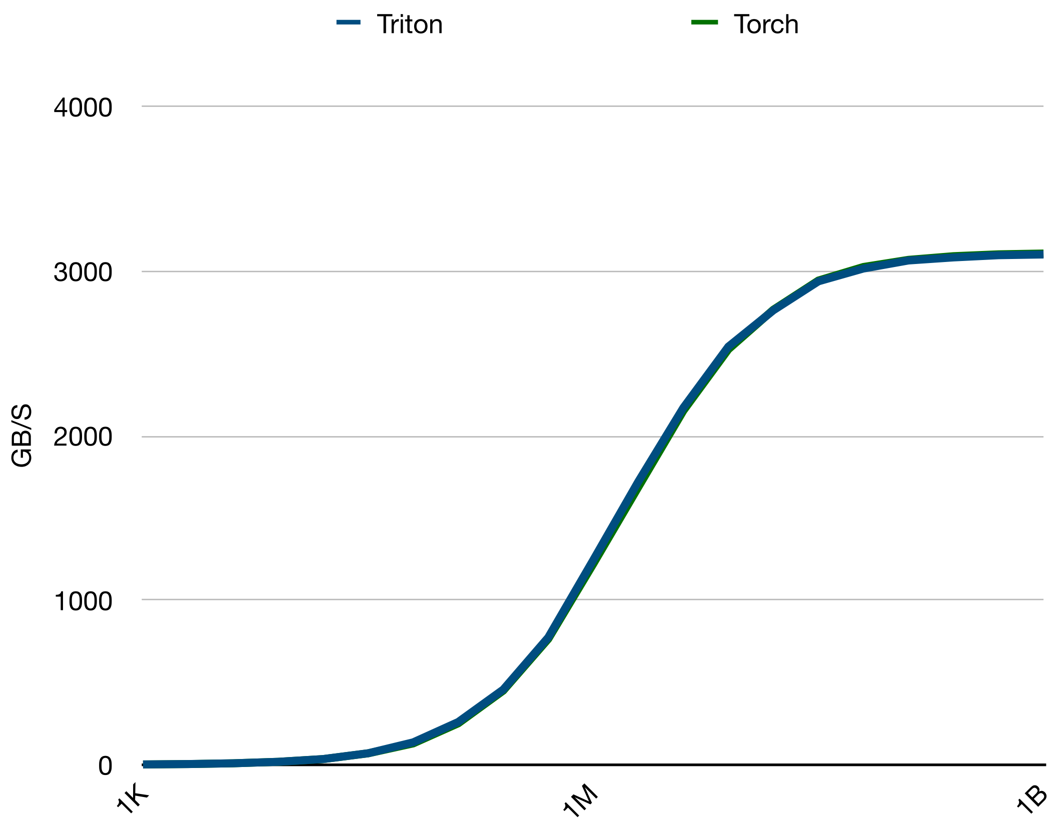
The grid can be a function of the kernel meta-parameters (e.g., tile size)

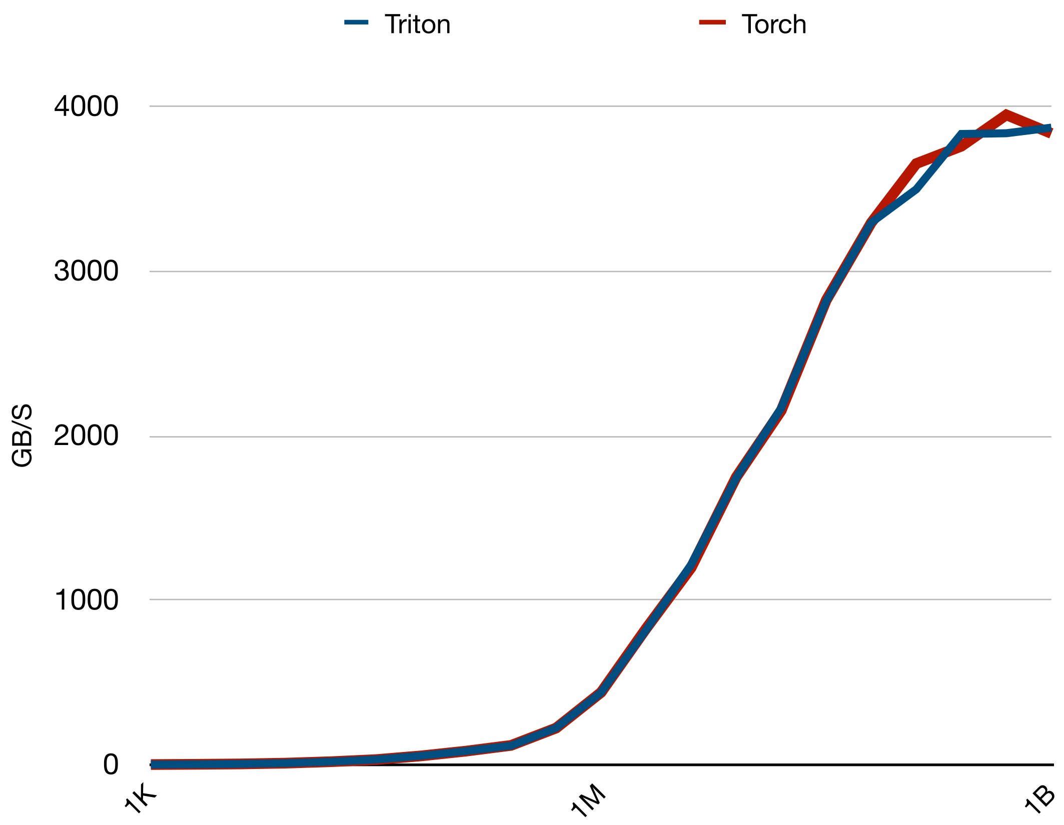
```
import triton.language as tl
import triton

@triton.jit
def _add(z_ptr, x_ptr, y_ptr, N, BLOCK: tl.constexpr):
    # same as torch.arange
    offsets = tl.arange(0, BLOCK)
    offsets += tl.program_id(0)*BLOCK
    # create 1024 pointers to X, Y, Z
    x_ptrs = x_ptr + offsets
    y_ptrs = y_ptr + offsets
    z_ptrs = z_ptr + offsets
    # load 1024 elements of X, Y, Z
    x = tl.load(x_ptrs, mask=offset<N)
    y = tl.load(y_ptrs, mask=offset<N)
    # do computations
    z = x + y
    # write-back 1024 elements of X, Y, Z
    tl.store(z_ptrs, z, mask=offset<N)

N = 192311
x = torch.randn(N, device='cuda')
y = torch.randn(N, device='cuda')
z = torch.randn(N, device='cuda')
grid = lambda args: (triton.cdiv(N, args['BLOCK']), )
_add[grid](z, x, y, N, TILE=1024)
```

VECTOR ADDITION PERFORMANCE

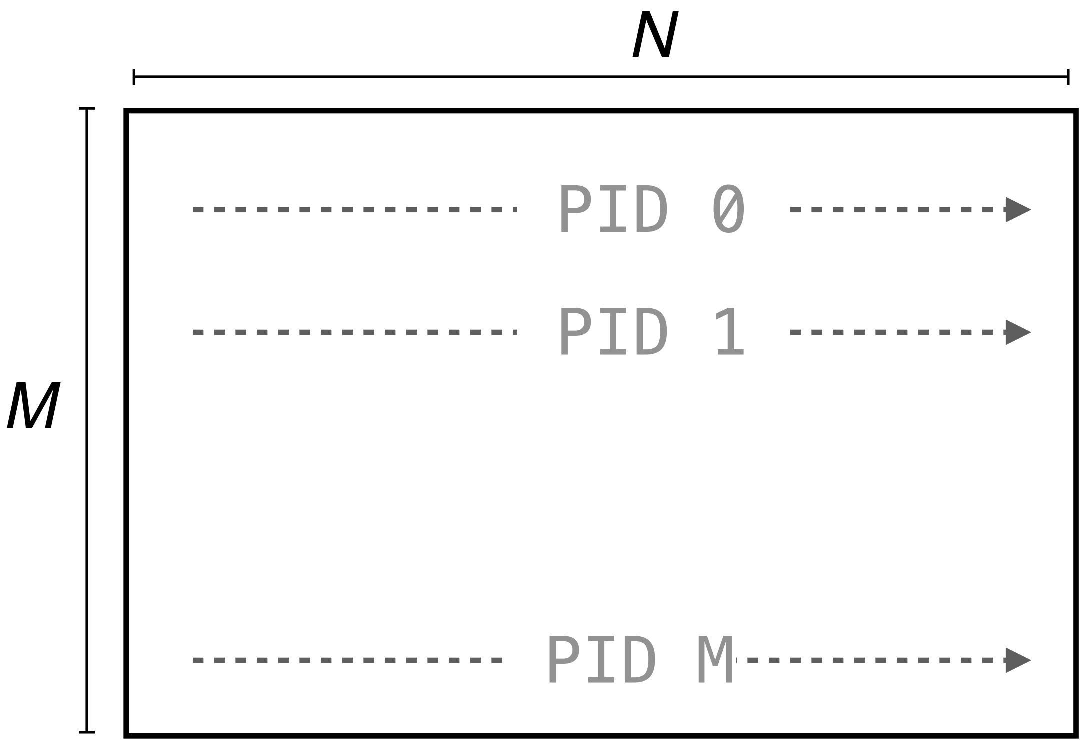




SOFTMAX

PARALLELIZATION STRATEGY

Each Triton program instance normalizes a different row of the `MxN` input matrix



Only works when each row of the matrix can fit in the GPU's SRAM

SOFTMAX

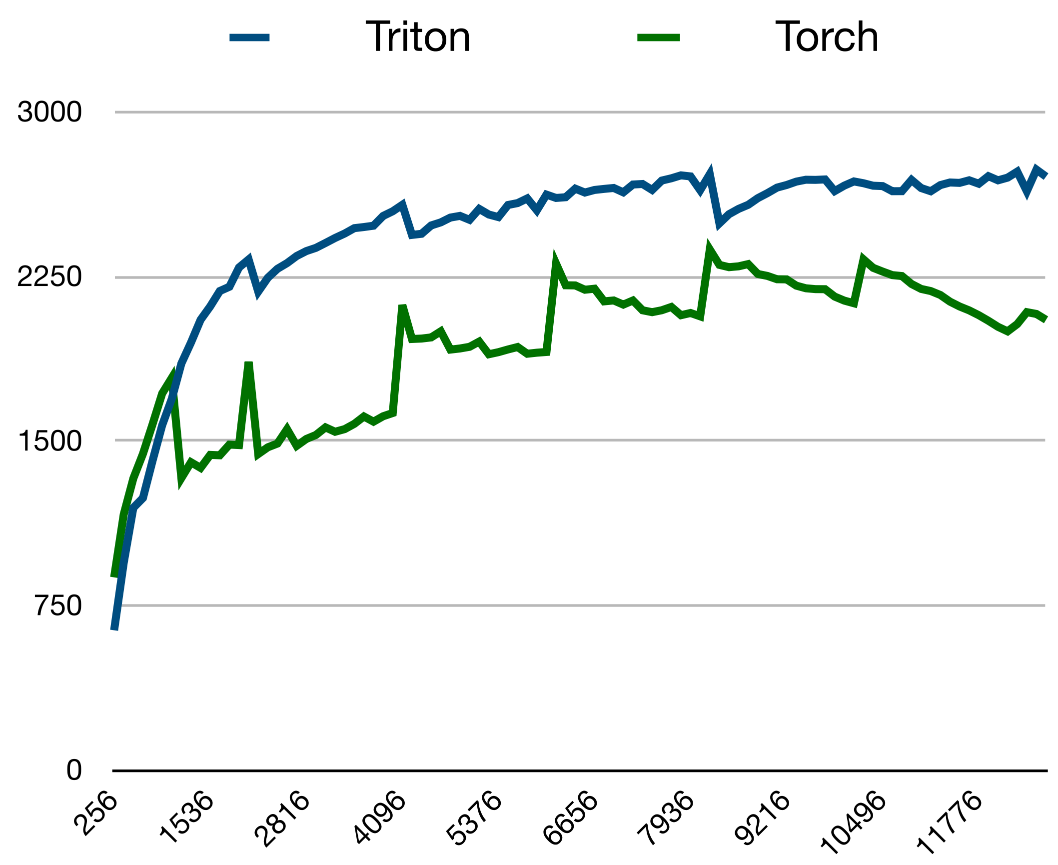
Entirely fused kernels for softmax can be written in less than 10 lines for tensors whose rows fit in SRAM

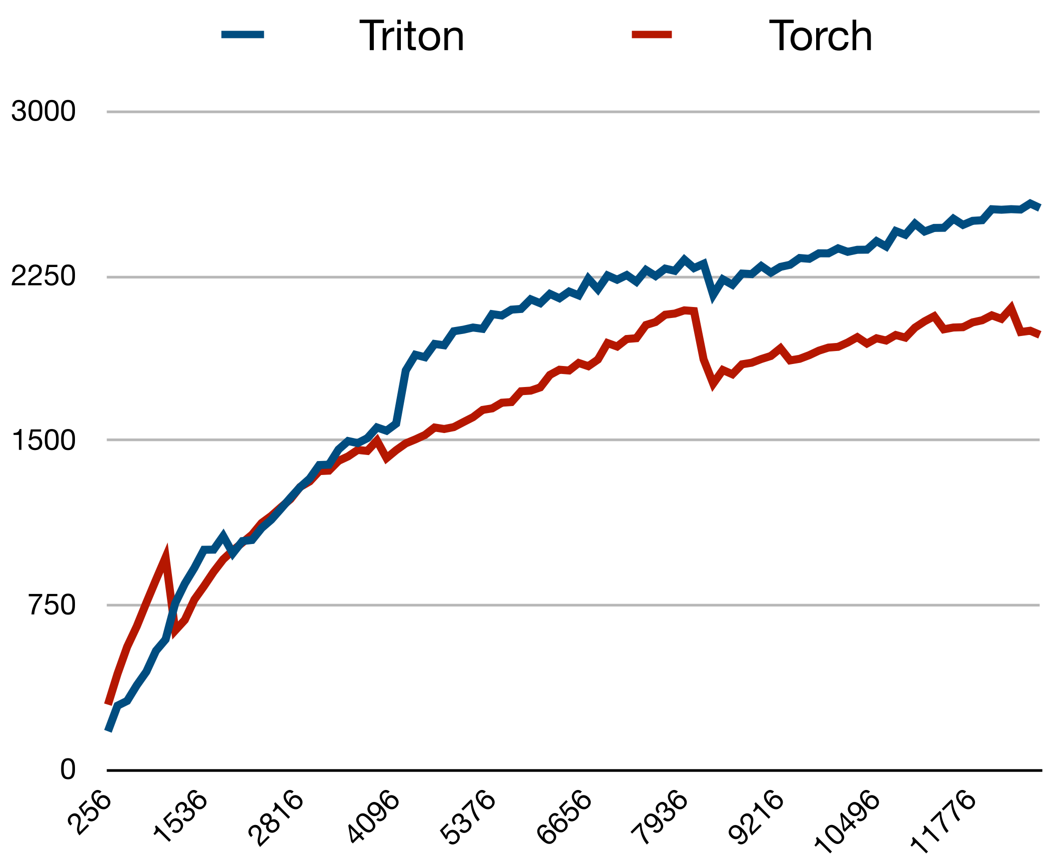
Triton tensors must have power-of-two number of elements. Appropriate padding is required in `tl.load`

```
import torch
import triton.language as tl
import triton

@triton.jit
def _softmax(z_ptr, x_ptr, stride, N,
             BLOCK: tl.constexpr):
    # Each program instance normalizes
    # all columns in a different row of X
    row = tl.program_id(0)
    cols = tl.arange(0, BLOCK)
    # Load a row of row-major X to SRAM
    x_ptrs = x_ptr + row*stride + cols
    x = tl.load(x_ptrs, mask = cols < N,
                other = float(‘-inf’))
    # Normalization in SRAM, in FP32
    x = x.to(tl.float32)
    x = x - tl.max(x, axis=0)
    num = tl.exp(x)
    den = tl.sum(num, axis=0)
    z = num / den;
    # Write-back to DRAM into row-major Z
    tl.store(z_ptr + row*stride + cols, z,
             mask = cols < N)
```

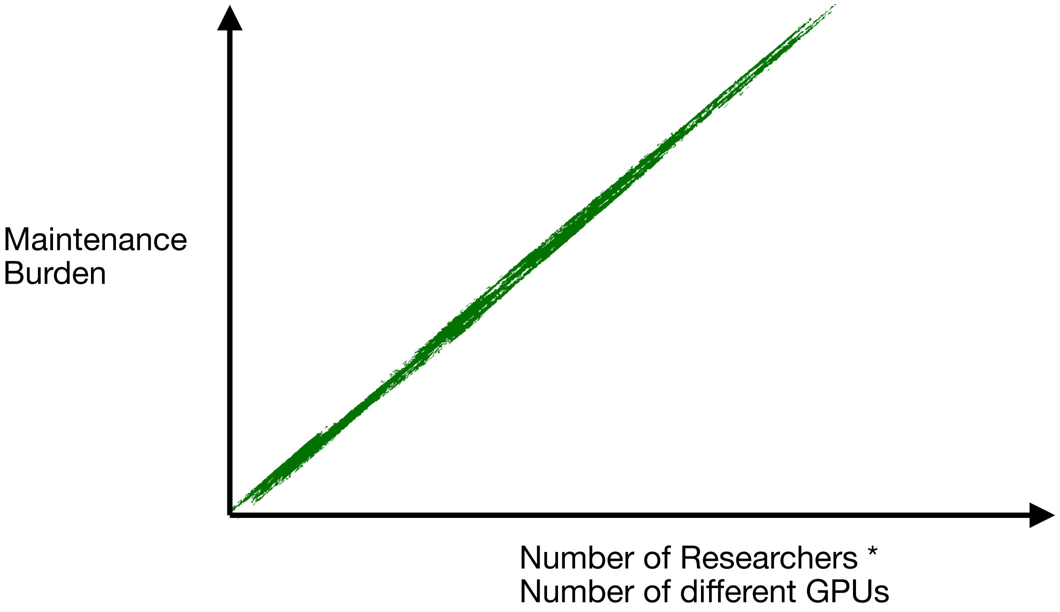
SOFTMAX PERFORMANCE

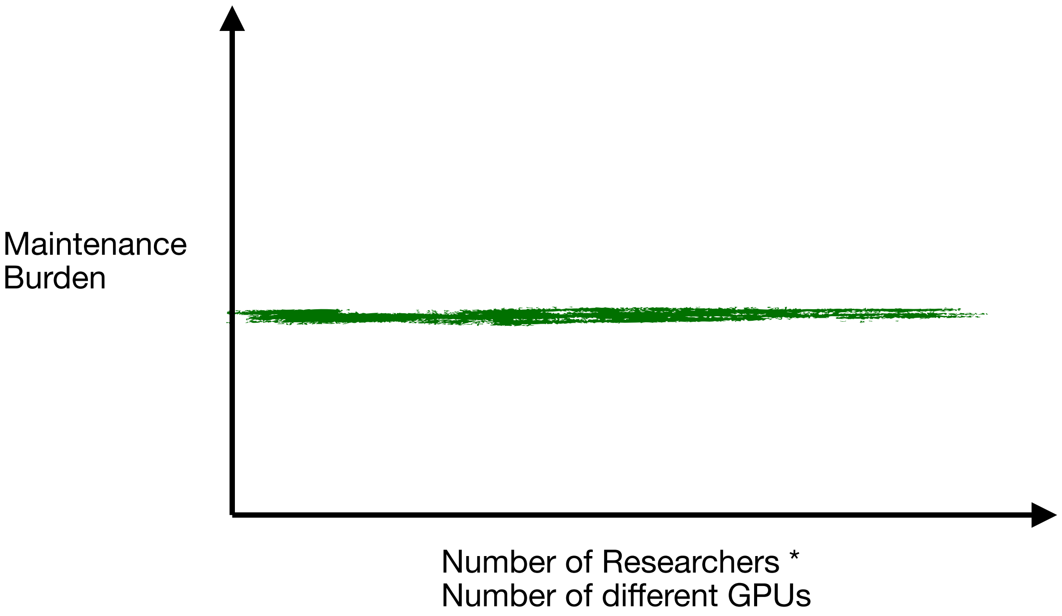




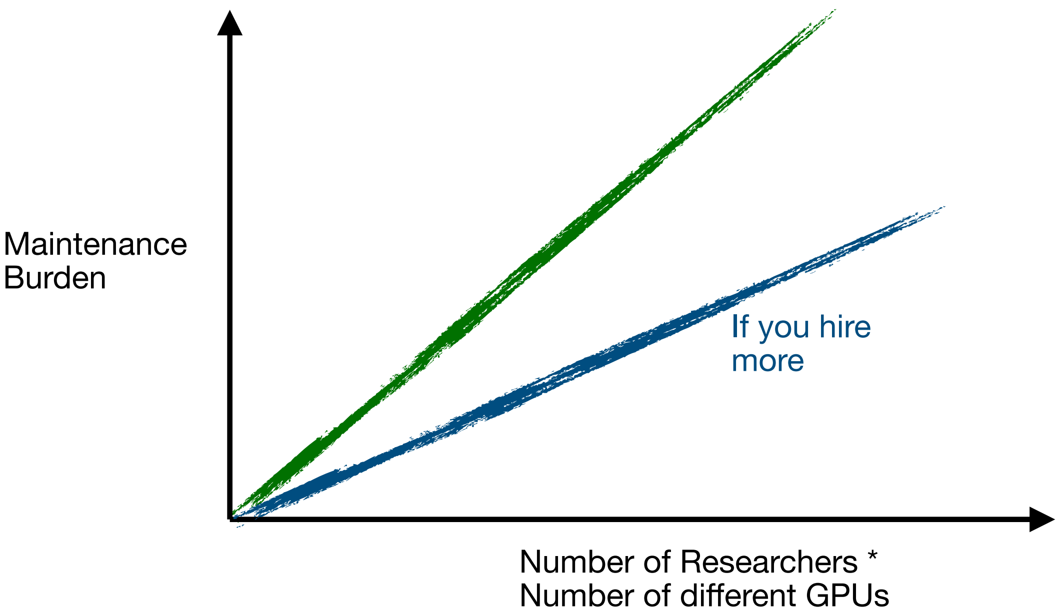
WHY IS IT POWERFUL ?

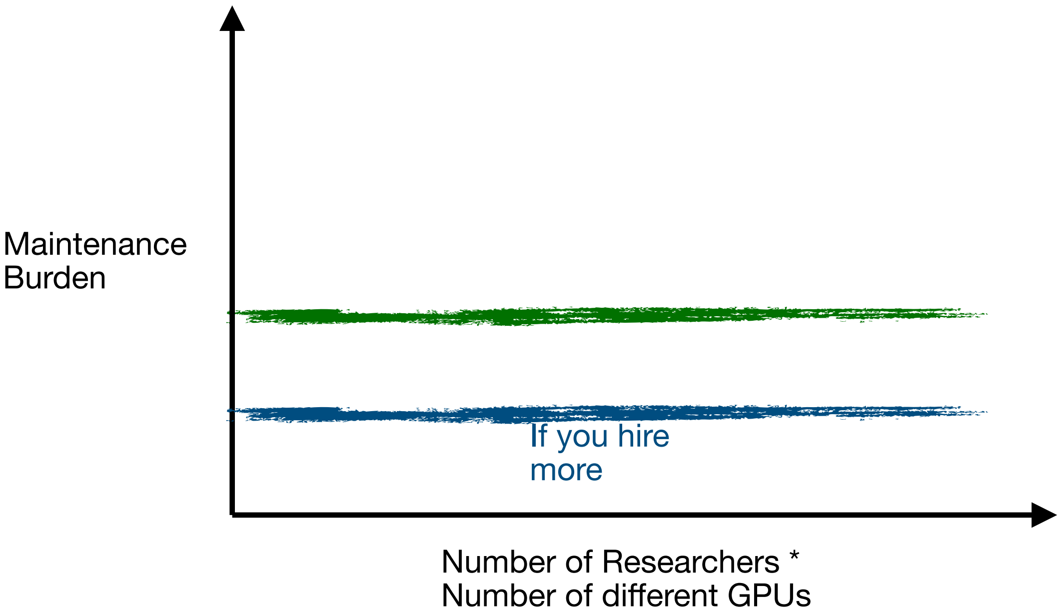
OFFLOADING KERNEL DEVELOPMENT TO RESEARCHERS





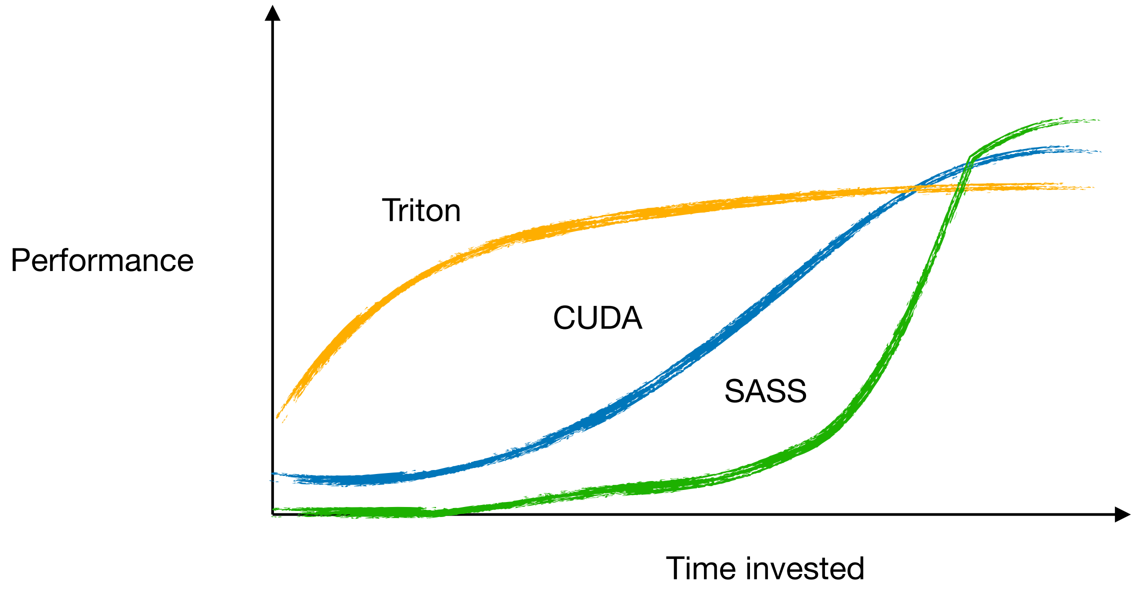
OFFLOADING KERNEL DEVELOPMENT TO RESEARCHERS





REDUCING TECHNICAL DEBT

GETTING GREAT PERFORMANCE QUICKLY



THANK YOU FOR YOUR ATTENTION