Writing Extensions and Bindings for GPU

Krishnakant Singh, Rakuten(RIT)

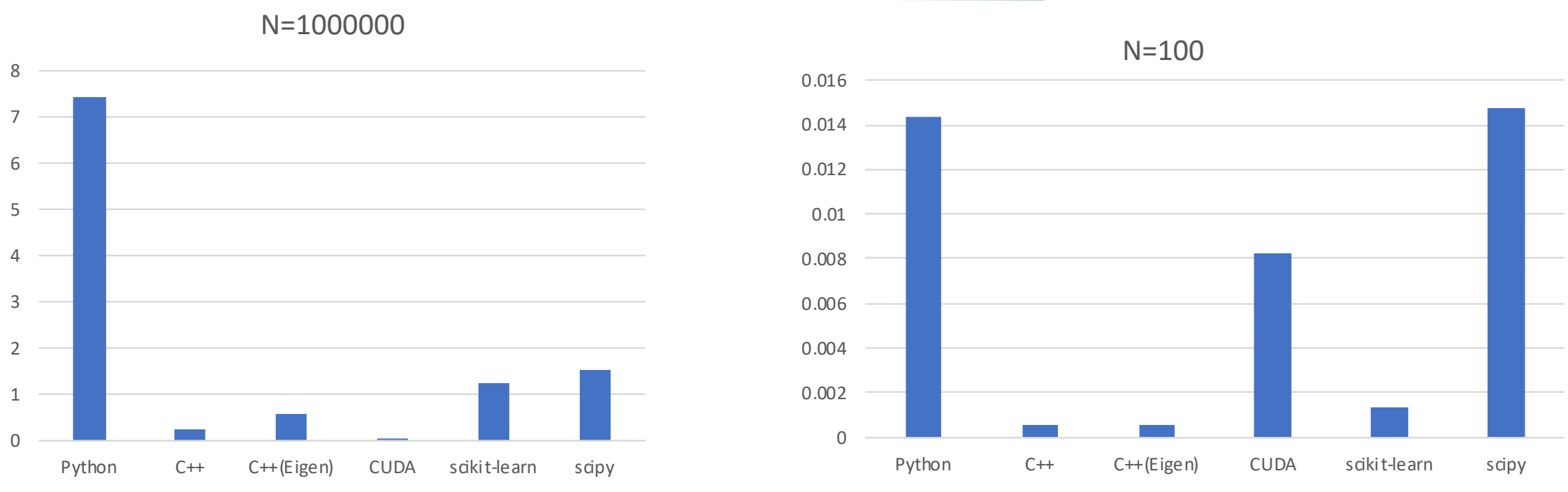
Goals

- Extending Python Code with Native C, C++, Rust etc Code
- Extending Python Code with CUDA Code
- Writing Wrapper Code for CUDA and C Extensions
- Building and Distributing GPU Extensions

Motivation

- Native Python is slow
- Custom Libraries are slow due to wide customizability
- Almost all computation libraries use extension of GPU or CPU eg Pytorch, Numpy, Scipy etc
- GPUs are expensive more so in production, utilize every ounce of power

Performance Comparison (Speed)



Time Comparision for K-Means algorithm of various technologies

Pros and Cons(of writing Extensions)

Why to Extensions	Why Not to Extensions
Speed!!!	Development is Slower
Use Optimized C and C++ Lib Functions	Memory Safety Issues
No GIL	Writing CUDA is Very Tough!!!
More Control Over Memory	

Simple CPU Extension

Native Code (add_cpu.c)

```
#include <stdio.h>
#include <stdlib.h>

void add_cpu(double *a, double *b, double *c,
int n){
    for(int i=0; i < n; i++){
        c[i] = a[i] + b [i];
    }
}
```

Wrapper Code(wrapper.pyx)

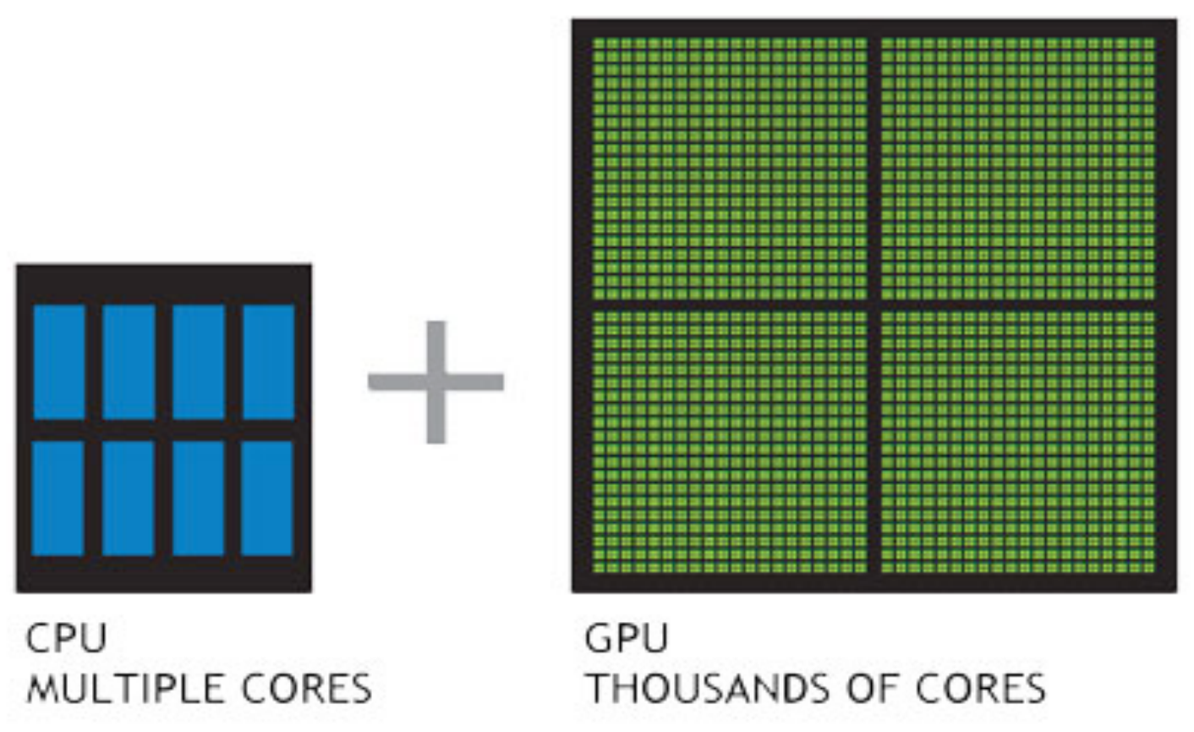
```
# cython: language_level=3
#if defined(NPY_NO_DEPRECATED_API) &&
NPY_NO_DEPRECATED_API >= NPY_1_7_API_VERSION
import numpy as np
cimport numpy as np

cdef extern from "add_cpu.hpp":
    void add_cpu(double* a, double* b, double*
c, int n)

cpdef adder_cpu(double[:] a, double[:] b,
double[:] c):
    cdef int n = a.size
    add_cpu(&a[0], &b[0], &c[0], n)
```

Setup Script(setup.py)

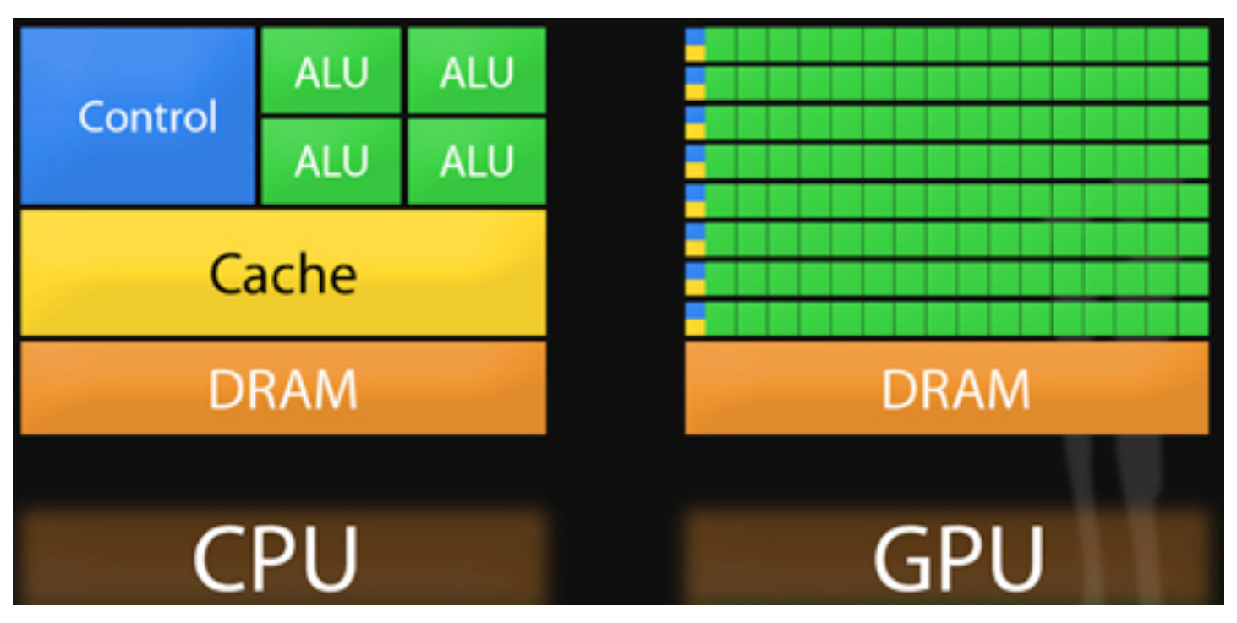
```
#!/usr/bin/env python3
# encoding: utf-8

from distutils.core import setup, Extension

cpu_example = Extension(‘add_cpu', sources =
[‘add_cpu.c'])

setup(name=‘EuroPy2020',
      version='0.1.0',
      description=‘Writing Extension',
      ext_modules=[cpu_example])
```

GPU Architecture

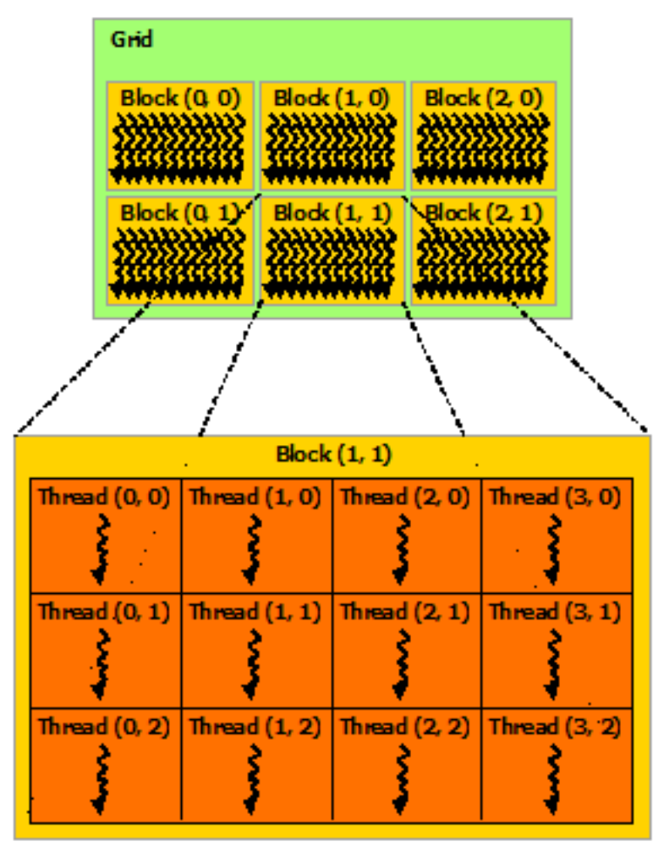
- GPU: CPU with Exponentially More Cores

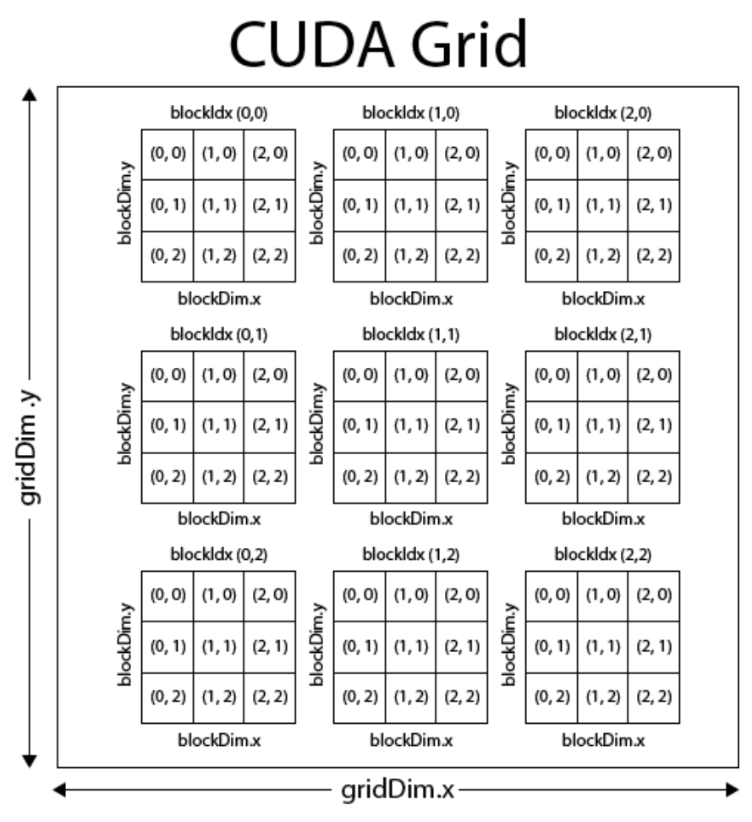


- No Free Lunch: More Cores Less Memory



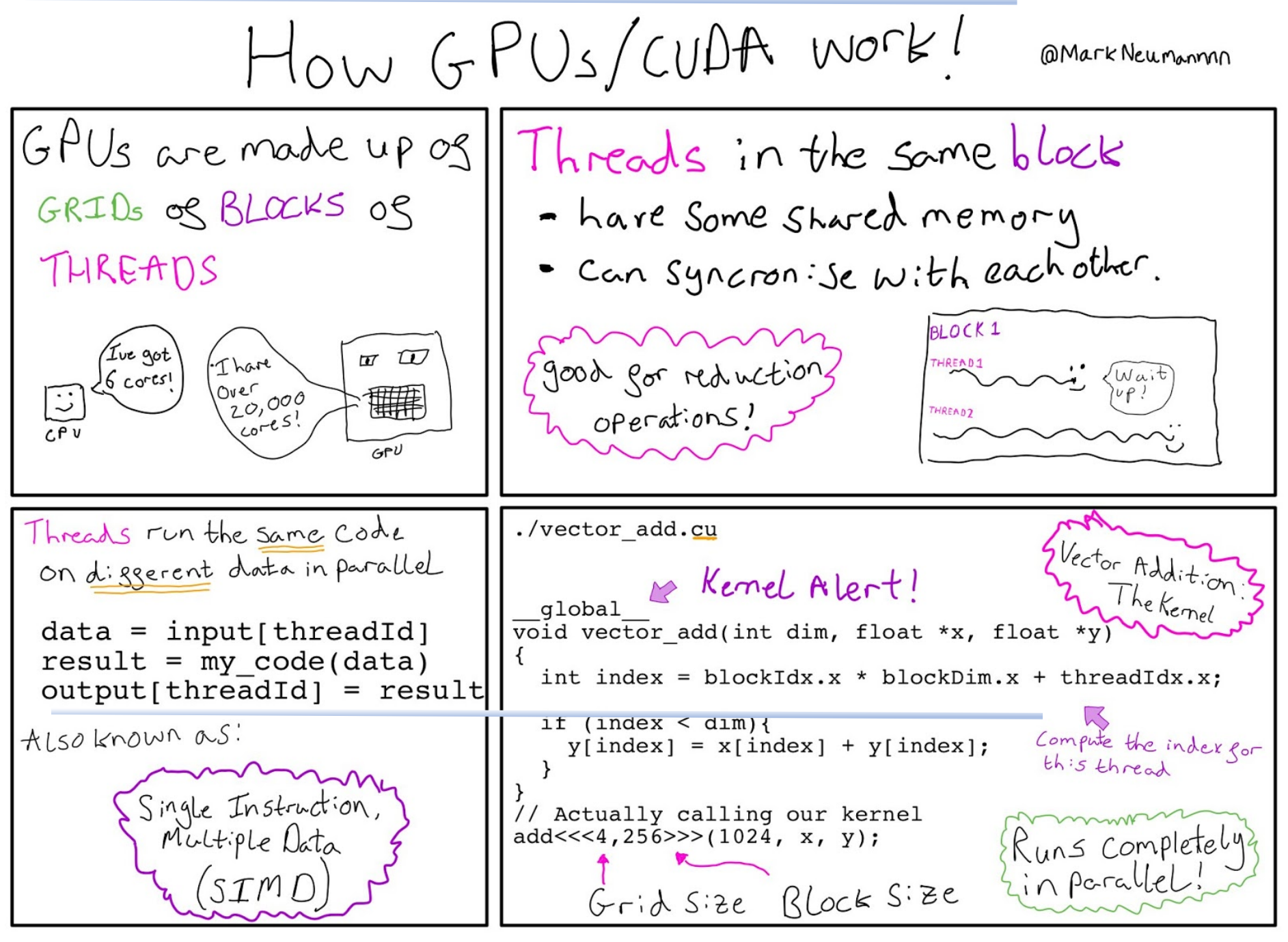
Terminology





- **SM**: GPUs are made of Streaming Multiprocessors
- **Grid**: A SM is divided into blocks of grid.
- **Blocks**: Cooperation is done at block level in GPUS. Blocks can be 1d, 2d or 3d
- **Threads**: Grids are divided into threads, smallest unit of processing in GPUS
- **Warps**: Memory aligned 32 threads. Required for performance optimization

GPU Computing Model



```
data = input[threadId]
result = my_code(data)
output[threadId] = result
```

```
./vector_add.cu

__global__
void vector_add(int dim, float *x, float *y)
{
  int index = blockIdx.x * blockDim.x + threadIdx.x;
  if (index < dim){
    y[index] = x[index] + y[index];
  }
}
// Actually calling our kernel
add<<<4,256>>>(1024, x, y);
```

Cuda In Python

Numba
- Open Source and Active Development
- CPU and GPU support
- Cuda libs like cufft, curand available

PyCuda
- Full Cuda API is available
- RAII: Safe Code

Cupy
- Numpy for GPUs
- Open Source and Supported by Nvidia

Issues
- Harder to Debug
- Some functions might be missing
- Mostly would end up writing CUSTOM kernels in CUDA
- Performance Optimization: Writing CUDA is the only option

Simple GPU Example

- Kernel Code (add_gpu.cu)

```
#include <stdio.h>
#include <cuda.h>
#include "add_gpu.h"


__global__ void add_kernel(double *a, double *b, double
*c, int n)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx < n)
      c[idx] = a[idx] + b[idx];
}
```

Main Function (add_gpu.cu)

```
void add_gpu(double *a, double *b, double *c, int n){
    size_t bytes = sizeof(int) * n * n;
    double *dev_a, *dev_b, *dev_c;
/*Assign Mem on Device*/
    cudaMalloc((void**) &dev_a, sizeof(double)*n*n);...
    /*Copy Host Mem to Device Mem*/
    cudaMemcpy(dev_a, a, bytes, cudaMemcpyHostToDevice);
...

    int num_threads = 256;
    int num_blocks = (n-1) / num_threads + 1;
    add_kernel<<<num_blocks, num_threads>>>(dev_a,
dev_b, dev_c, n);
    /*Copy from Device to Host*/
    cudaMemcpy(c, dev_c, bytes, cudaMemcpyDeviceToHost);
    /*Free Memory*/
    cudaFree(dev_a);cudaFree(dev_b); cudaFree(dev_c);
```

Wrapper Code(wrapper.pyx)

```
cdef extern from "add_gpu.h":
    void add_gpu(double *a, double *b, double *c, int n)

cpdef adder_gpu(double[:] a, double[:] b, double[:] c):
    cdef int n = a.size
    add_gpu(&a[0], &b[0], &c[0], n)
```

Setup Script(setup.py)

```
ext = Extension('gpuadder',
        sources = ['add_gpu.cu', 'wrapper.pyx'],
        library_dirs = [CUDA['lib64']],
        libraries = ['cudart'],
        language = 'c++',
        runtime_library_dirs = [CUDA['lib64']],
        extra_compile_args= {
            'gcc': [],
            'nvcc': [
                '-arch=sm_35', '-c',
                '--compiler-options', "'-fPIC'"
                ]
            },
            include_dirs = [numpy_include,
CUDA['include']]
        )
```

Conclusion

- **Rapid Prototyping**: Use. Numba or Cupy
- **Bindings**
  - **Cython** Learning Curve but great performance and just works
  - **Pybind11:** Easiest to setup, Native Datatypes for std containers, Performance is not that good while passing buffers
  - **Ctypes:** Great for simple cases, Pure Python
- **Cuda** :, Seep learning curve but great returns on Performance critical/Resource Utilization

- More Advanced Examples and Kmeans Cuda Implementation on my github

References

- Tools to Bind to Python
- Exploring K-Means in Python, C++ and CUDA
- Duke Course
- MarkNeuman
- Official Python documentation
- RBuilding a Python C Extension Module
- Python - C++ bindings
- An Even Easier Introduction to CUDA
- An Efficient Matrix Transpose in CUDA C/C++
- Rapids
- Kmean

Contact

Krishnakant Singh
@kris-singh
@krissingh_