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 so more so in production, utilize every ounce of power

Performance Comparison (Speed)

Pros and Cons(of writing Extensions)

<table>
<thead>
<tr>
<th>Why to Extensions</th>
<th>Why Not to Extensions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed!!</td>
<td>Development is Slower</td>
</tr>
<tr>
<td>Use Optimized C and C++ Lib Functions</td>
<td>Memory Safety Issues</td>
</tr>
<tr>
<td>No GIL</td>
<td>Writing CUDA is Very Tough!!</td>
</tr>
<tr>
<td>More Control/Over Memory</td>
<td></td>
</tr>
</tbody>
</table>

Simple GPU Example

```python
#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 size = sizeof(int) * n * n;
    double *dev_a, *dev_b, *dev_c;
    /\Assign Mem on Device/\n    cudaMalloc((void**)&dev_a, size);...
    cudaMemcpy(dev_a, a, size, 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/\n    cudaMemcpy(dev_c, dev_a, size, cudaMemcpyDeviceToHost);
    /\Free Memory/\n    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

Wrapper Code(wrapper.pyx)
cdef extern from "add_gpu.h":
    void add_gpu(double *, double *, double *, int);
cdef add_gpu(double[:, :, :], double[:, :, :], double[:, :, :], int):
    cudaMalloc((void**)&dev_a, size);...
    cudaMemcpy(dev_a, a, size, 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/\n    cudaMemcpy(dev_c, dev_a, size, cudaMemcpyDeviceToHost);
    /\Free Memory/\n    cudaFree(dev_a); cudaFree(dev_b); cudaFree(dev_c);
}

Setup Script(setup.py)

ext = Extension('goudder',
    sources = ['add_gpu.cu', 'wrapper.pyx'],
    libraries = ['cudart'],
    language = 'c++',
    runtime_library_dirs = [numpy_include],
    include_dirs = [numpy_include],
    extra_compile_args = { 
        'gcc': [],
        'nvcc': [],
        'arch': ["35", '64', '60'],
        '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, Numpy data types for std containers, Performance is not that good while passing buffers
  - Cygwin: Great for simple cases, Pure Python
  - Cuda: Deep learning curve but great returns on Performance critical/Resource Utilization
- More Advanced Examples and Kmeans Cuda Implementation on my github

References

- Tools to Work in Python
  - Extending E-Moves in Python, C++ and CUDA
  - Pybind11
  - Cython
  - nvcc
  - Official Data Types Documentation
  - Phython C Extension Module
  - Pybind11 - Cython
  - An Easy Entry Introduction to CUDA
  - An Efficient Memory Transparent in CUDA C/C++
- Talks
  - Talks

Contact

Krishnakant Singh
akris-singh
#akrissingh