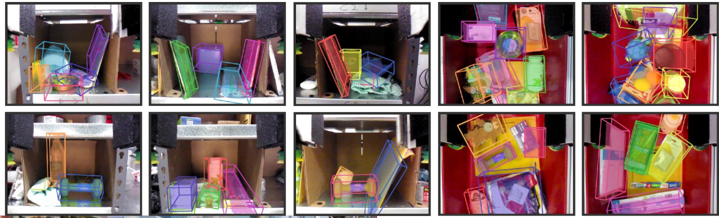
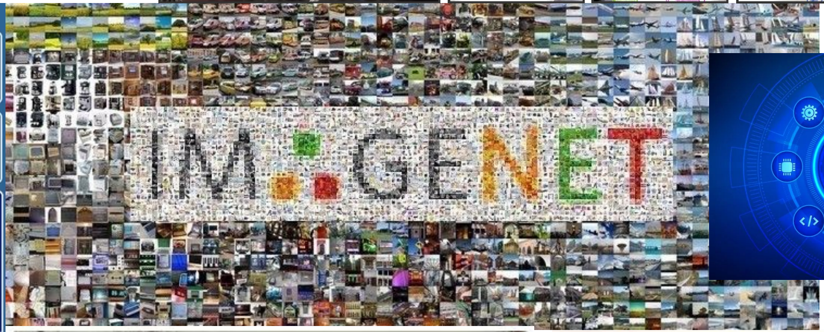
Accelerating Graph Convolutions

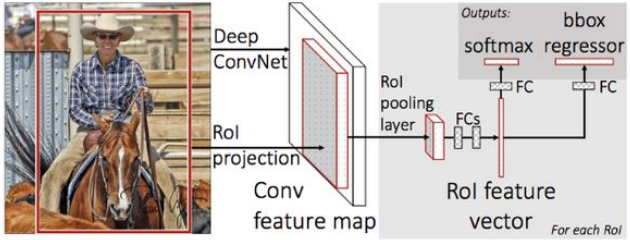
A Simple Python Runtime System Built for CS263

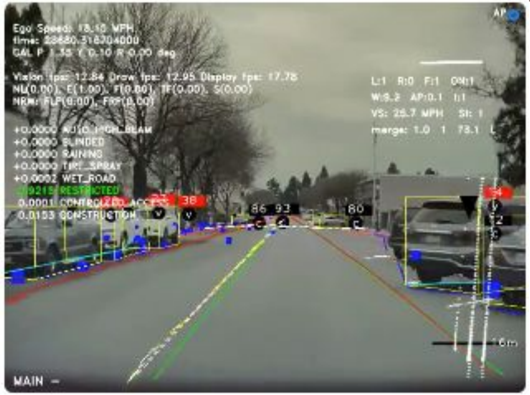
Yuke Wang and Sirius Zhang

So Called AI Revolution









Our Project / Presentation Milestones

- Looked at runtime system’s role in this “AI” revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN)
- Profiled performance difference and runtime bottleneck.

Our Project / Presentation Milestones

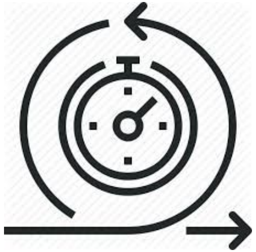
- Looked at runtime system’s role in this “AI” revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
- Profiled performance difference and runtime bottleneck.

What Data Scientists / ML Researchers Need?

- Fast iteration in algorithm development.
- Matrix, tensor (“matrix” > 2D), and linear algebra are the workhorse of all math-heavy, data-hungry algorithms.
- Fast matrix / tensor calculation, potentially accelerated through parallel computing hardware.

$$\begin{bmatrix} a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 \\ a_7 & a_8 & a_9 \end{bmatrix} \begin{bmatrix} b_1 & b_2 & b_3 \\ b_4 & b_5 & b_6 \\ b_7 & b_8 & b_9 \end{bmatrix} = \begin{bmatrix} c_1 & c_2 & c_3 \\ c_4 & c_5 & c_6 \\ c_7 & c_8 & c_9 \end{bmatrix}$$

NVIDIA



How Runtime System Helped in this Revolution?

- Use High level language such as Python to cover up low-level machine programming details.
- For performance, bringing in CUDA and C/C++, and use them under the hood for performance critical places.

Our Project / Presentation Milestones

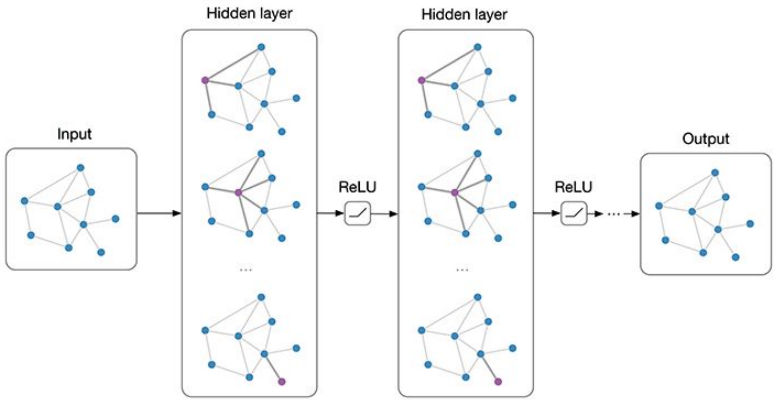
- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
    - Overall Goal
    - Intro to GCN
    - Implemented two variants of Graph convolutional kernels used a mixed of CUDA/C++/Python code.
        - Scatter & Gather (SAG) kernel
        - Sparse Matrix and Matrix Multiplication (SpMM) kernel
    - Python Wrapper
        - Different Variants
        - Our Choice

Our Project / Presentation Milestones

- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
  - Overall Goal
  - Intro to GCN
  - Implemented two variants of Graph convolutional kernels used a mixed of CUDA/C++/Python code.
    - Scatter & Gather (SAG) kernel
    - Sparse Matrix and Matrix Multiplication (SpMM) kernel
  - Python Wrapper
    - Different Variants
    - Our Choice

Overall Goal

- Implement a runtime GCN library that looks at input graph's characteristic to make decisions on what specific kernels to call.
- Specifically, switching between S&G kernel and SpMM kernel based on input graph's at runtime.

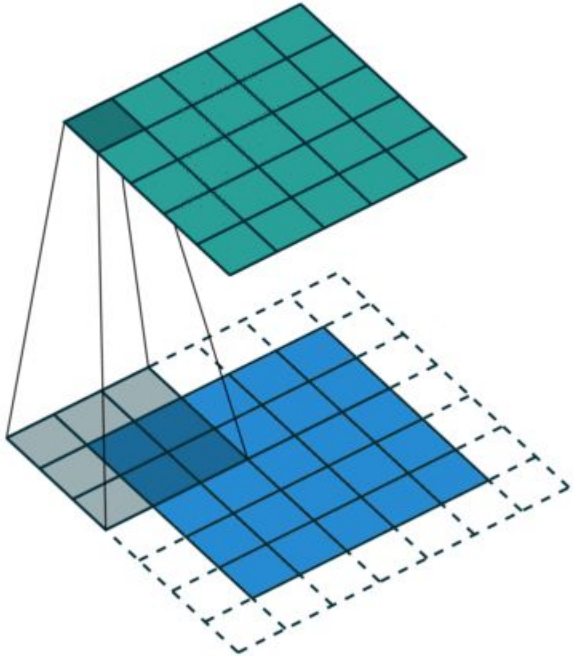


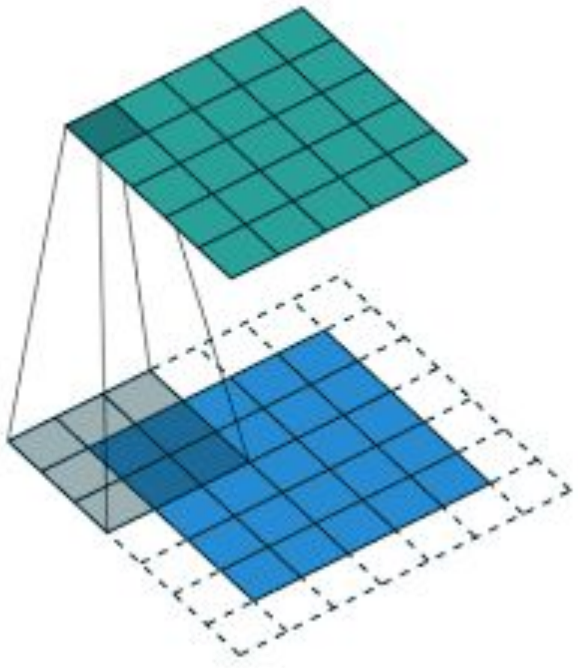
Multi-layer Graph Convolutional Network (GCN) with first-order filters.

Our Project / Presentation Milestones

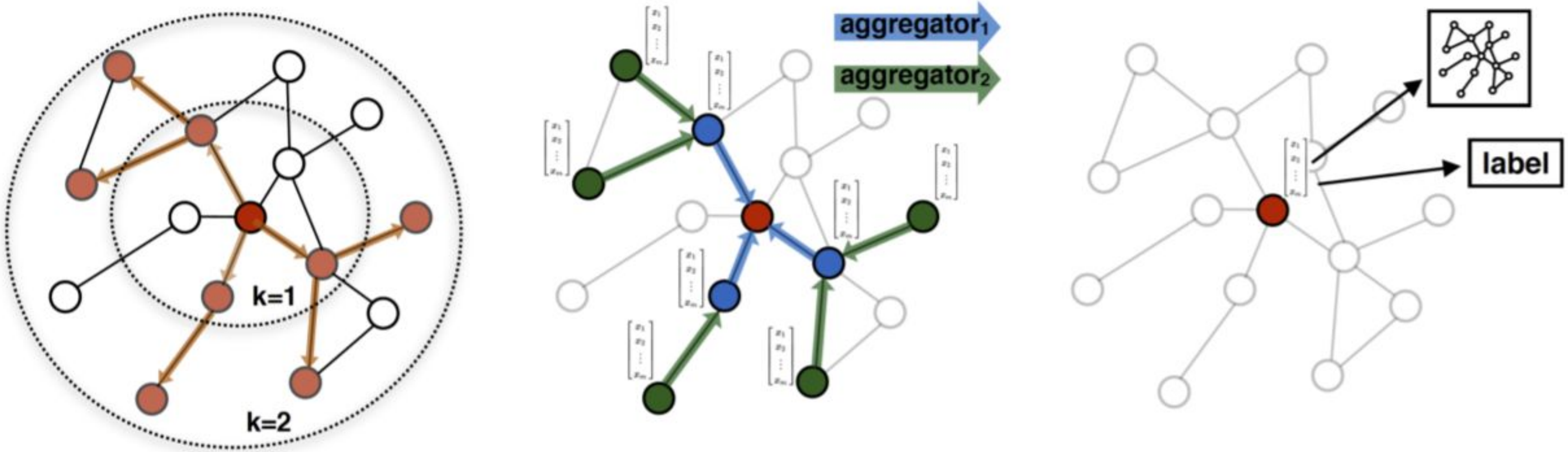
- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
  - Overall Goal
  - Intro to GCN
  - Implemented two variants of Graph convolutional kernels used a mixed of CUDA/C++/Python code.
    - Scatter & Gather (SAG) kernel
    - Sparse Matrix and Matrix Multiplication (SpMM) kernel
  - Python Wrapper
    - Different Variants
    - Our Choice

2D Convolution Versus Graph Convolution





2D Convolution Versus Graph Convolution



1. Sample neighborhood

2. Aggregate feature information from neighbors

3. Predict graph context and label using aggregated information

Mathematical Formulation of GCN

- SpMM (1)(2)
- SAG (3)

- Every neural network layer can then be written as a non-linear function

$$H^{(l+1)} = f(H^{(l)}, A)\,,$$

with $H^{(0)} = X$ and $H^{(L)} = Z$ (or z for graph-level outputs), L being the number of layers. The specific models then differ only in how $f(\cdot, \cdot)$ is chosen and parameterized.

- As an example, let's consider the following very simple form of a layer-wise propagation rule:

$$f(H^{(l)}, A) = \sigma\left(AH^{(l)}W^{(l)}\right)\,,$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\cdot)$ is a non-linear activation function like the ReLU. Despite its simplicity this model is already quite powerful (we'll come to that in a moment).

- $$a_v^{k+1} = Aggregate^{k+1}(h_u^{k+1} | u \in Neighbor(v))$$
$$h_v^{k+1} = Update^{k+1}(a_v^{k+1}, h_v^k)$$

https://tkipf.github.io/graph-convolutional-networks/

Our Project / Presentation Milestones

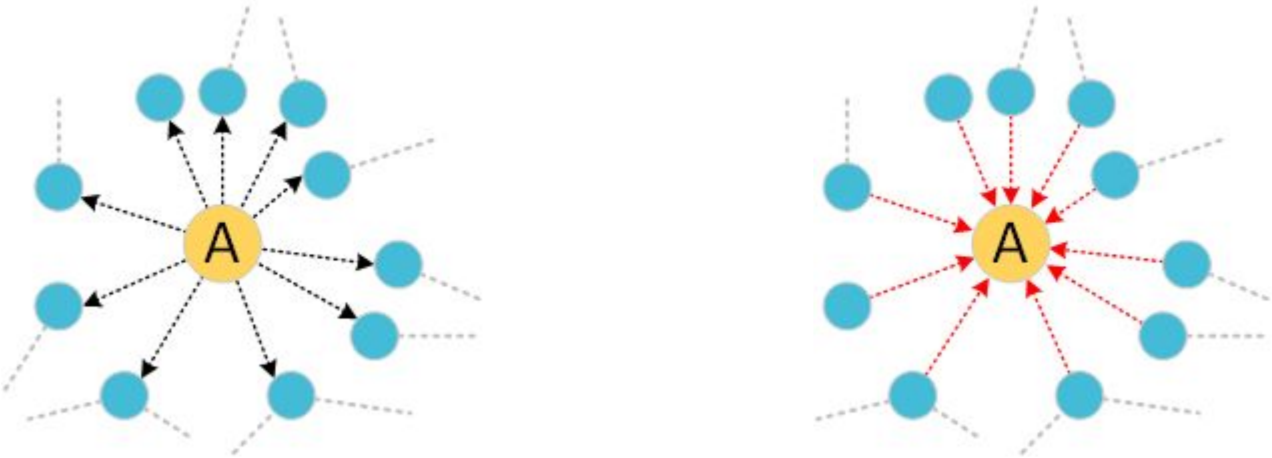
- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
  - Overall goal
  - Intro to GCN
  - Implemented two variants of Graph convolutional kernels used a mixed of CUDA/C++/Python code.
    - Scatter & Gather (SAG) kernel
    - Sparse Matrix and Matrix Multiplication (SpMM) kernel
  - Python Wrapper
    - Different Variants
    - Our Choice

Our Project / Presentation Milestones

- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
    - Intro to GCN
    - Implemented two variants of Graph convolutional kernels used a mixed of CUDA/C++/Python code.
        - Scatter & Gather (SAG) kernel
        - Sparse Matrix and Matrix Multiplication (SpMM) kernel
    - Python Wrapper
        - Different Variants
        - Our Choice

Scatter & Gather (SAG)

- Originated from graph processing, such as, PageRank, BFS.
- Push (conflicting-write) Vs. Pull (irregular-read)

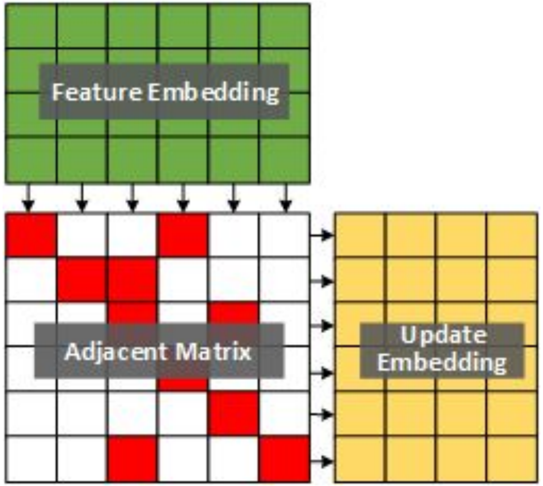


Our Project / Presentation Milestones

- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
    - Intro to GCN
    - Implemented two variants of Graph convolutional kernels used a mixed of CUDA/C++/Python code.
        - Scatter & Gather (SAG) kernel
        - Sparse Matrix and Matrix Multiplication (SpMM) kernel
    - Python Wrapper
        - Different Variants
        - Our Choice

Sparse Matrix-Matrix Multiplication (SpMM)

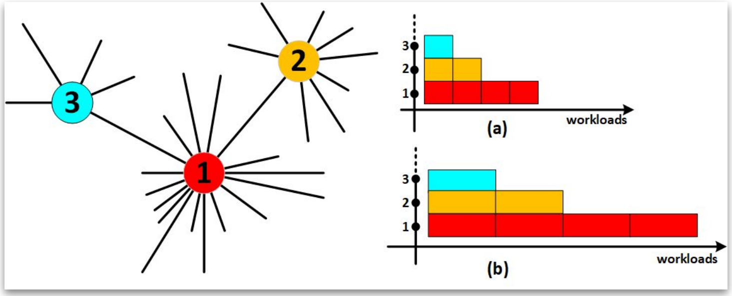
- Adapted from GraphBLAS-SpMV[1]
- Used for SUM-based Aggregation.

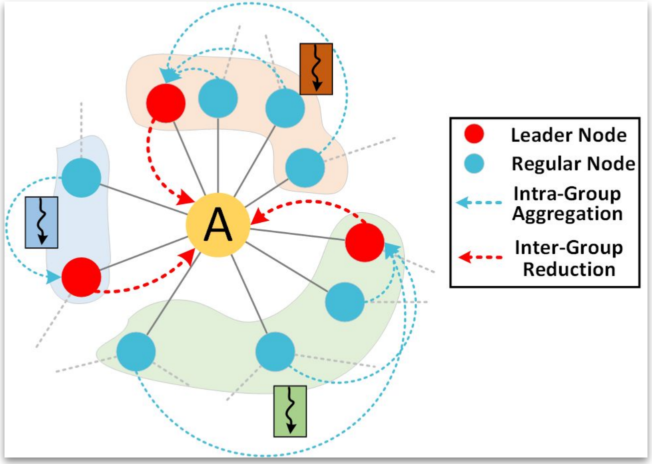


[1] http://graphblas.org/index.php?title=Graph_BLAS_Forum

Kernel Optimization

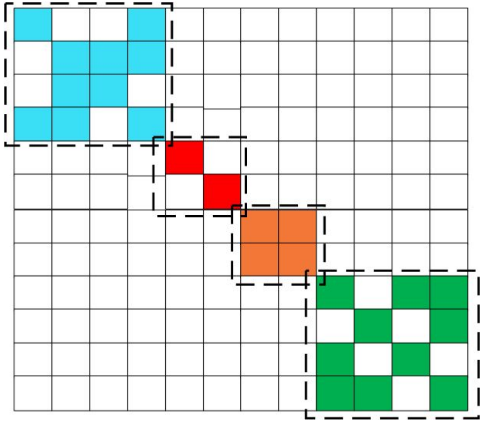
Group-based Neighbor Partitioning

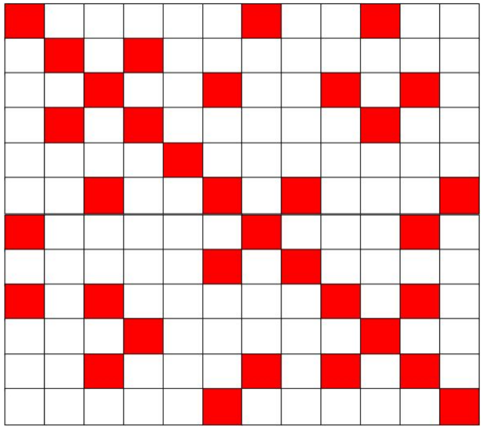




Kernel Choice

- Block-diagonal graph-- SpMM (top right).
- Random irregular graph -- SAG (bottom right).





Our Project / Presentation Milestones

- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
  - Intro to GCN
  - Implemented two variants of Graph convolutional kernels used a mixed of CUDA/C++/Python code.
    - Scatter & Gather (SAG) kernel
    - Sparse Matrix and Matrix Multiplication (SpMM) kernel
  - Python Wrapper
    - Different Variants
    - Our Choice

Our Project / Presentation Milestones

- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
    - Intro to GCN
    - Implemented two variants of Graph convolutional kernels used a mixed of CUDA/C++/Python code.
        - Scatter & Gather (SAG) kernel
        - Sparse Matrix and Matrix Multiplication (SpMM) kernel
    - Python Wrapper
        - Different Variants
        - Our Choice

Implementation Details, list of wrapper choices

- Numba
- PyCuda
- scikit-CUDA
- SWIG
- Pytorch
- etc…..

Implementation Details, list of wrapper choices

- Numba (runtime compiler specifically optimize for math numpy code)
- PyCuda (Full CUDA API in Python, GC, Try-Catch CUDA errors, etc)
- scikit-CUDA (similar to PyCuda, only PyCuda is backed by Nvidia)
- SWIG (Simple Wrapper Interface Generator, statically compiled .o shared library)
- Pytorch (Augo-grad)
- etc…..

Python Wrapper Variant -- PyTorch

- Allow easy integrations of mixed CUDA, C++, Python code.

```python
from setuptools import setup, Extension
from torch.utils import cpp_extension

setup(name='lltm_cpp',
      ext_modules=[cpp_extension.CppExtension('lltm_cpp', ['lltm.cpp'])],
      cmdclass={'build_ext': cpp_extension.BuildExtension})
```

```cpp
PYBIND11_MODULE(TORCH_EXTENSION_NAME, m) {
  m.def("forward", &lltm_forward, "LLTM forward");
  m.def("backward", &lltm_backward, "LLTM backward");
}
```

```python
class LLTMFunction(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input, weights, bias, old_h, old_cell):
        outputs = lltm_cpp.forward(input, weights, bias, old_h, old_cell)
        new_h, new_cell = outputs[:2]
        variables = outputs[1:] + [weights]
        ctx.save_for_backward(*variables)

        return new_h, new_cell
```

Python layer

```cpp
std::vector<torch::Tensor> lltm_cuda_forward(
    torch::Tensor input,
    torch::Tensor weights,
    torch::Tensor bias,
    torch::Tensor old_h,
    torch::Tensor old_cell) {
  auto X = torch::cat({old_h, input}, /*dim=*/1);
  auto gates = torch::addmm(bias, X, weights.transpose(0, 1));

  const auto batch_size = old_cell.size(0);
  const auto state_size = old_cell.size(1);

  auto new_h = torch::zeros_like(old_cell);
  auto new_cell = torch::zeros_like(old_cell);
  auto input_gate = torch::zeros_like(old_cell);
  auto output_gate = torch::zeros_like(old_cell);
  auto candidate_cell = torch::zeros_like(old_cell);

  const int threads = 1024;
  const dim3 blocks((state_size + threads - 1) / threads, batch_size);

  AT_DISPATCH_FLOATING_TYPES(gates.type(), "lltm_forward_cuda", ([&] {
    lltm_cuda_forward_kernel<scalar_t><<<blocks, threads>>>(
        gates.data<scalar_t>(),
        old_cell.data<scalar_t>(),
        new_h.data<scalar_t>(),
        new_cell.data<scalar_t>(),
        input_gate.data<scalar_t>(),
        output_gate.data<scalar_t>(),
        candidate_cell.data<scalar_t>(),
        state_size);
  }));

  return {new_h, new_cell, input_gate, output_gate, candidate_cell, X, gates};
}
```

C++ layer

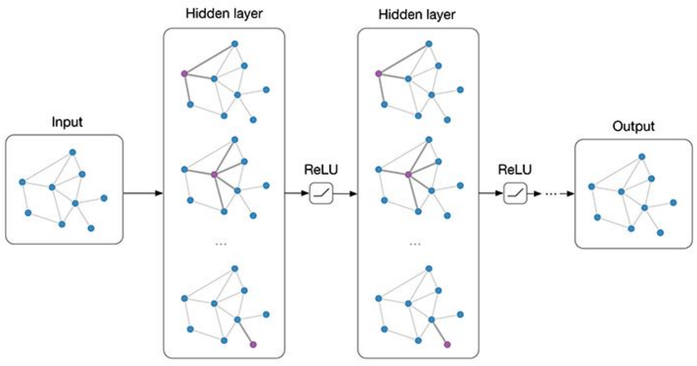
```cpp
template <typename scalar_t>
__global__ void lltm_cuda_forward_kernel(
    const scalar_t* __restrict__ gates,
    const scalar_t* __restrict__ old_cell,
    scalar_t* __restrict__ new_h,
    scalar_t* __restrict__ new_cell,
    scalar_t* __restrict__ input_gate,
    scalar_t* __restrict__ output_gate,
    scalar_t* __restrict__ candidate_cell,
    size_t state_size) {
  const int column = blockIdx.x * blockDim.x + threadIdx.x;
  const int index = blockIdx.y * state_size + column;
  const int gates_row = blockIdx.y * (state_size * 3);
  if (column < state_size) {
    input_gate[index] = sigmoid(gates[gates_row + column]);
    output_gate[index] = sigmoid(gates[gates_row + state_size + column]);
    candidate_cell[index] = elu(gates[gates_row + 2 * state_size + column]);
    new_cell[index] =
        old_cell[index] + candidate_cell[index] * input_gate[index];
    new_h[index] = tanh(new_cell[index]) * output_gate[index];
  }
}
```

CUDA Layer

https://pytorch.org/tutorials/advanced/cpp_extension.html

Our Project Recap and Our Choice of Wrapper

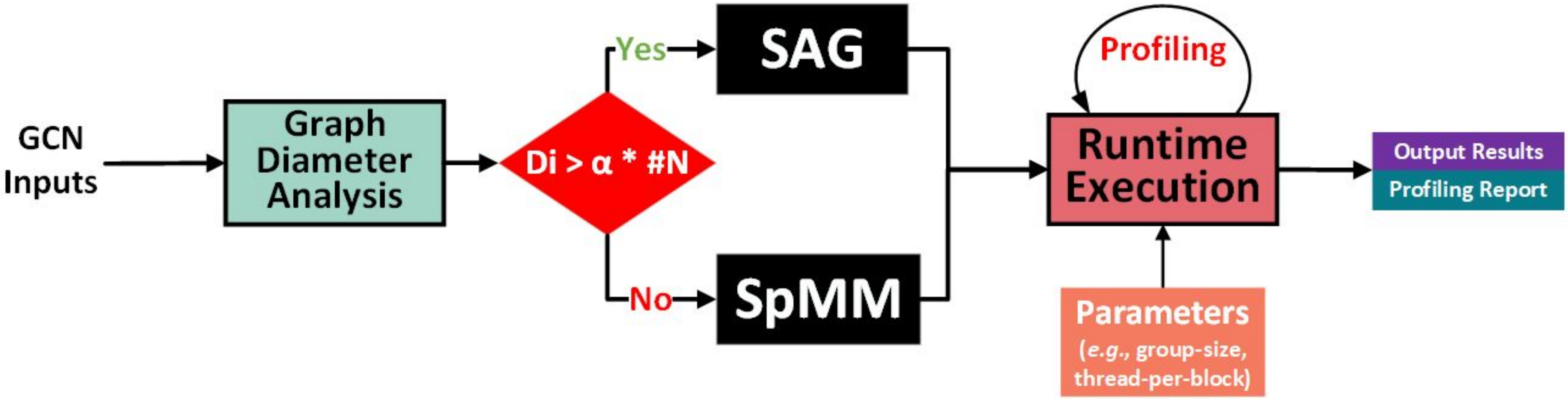
- Implement a runtime GCN library that looks at input graph's characteristic to make decisions on what specific kernels to call.
- Specifically, switching between S&G kernel and SpMM kernel based on input graph's at runtime.
- PyTorch wrapper.



Multi-layer Graph Convolutional Network (GCN) with first-order filters.

PyTorch

Overall Architecture



Our Project

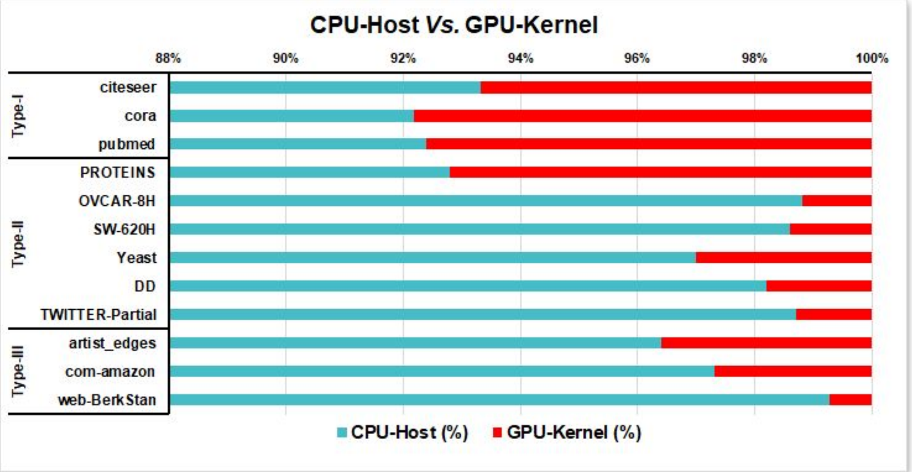
- Looked at runtime system's role in this "AI" revolution.
- Implemented a runtime system for Graph Convolutional Networks (GCN).
- **Profiled performance difference and runtime bottleneck.**

Tools & Datasets

- **[Host]** Xeon 4110 Silver 8 core, 64GB DDR4 RAM.
- **[GPU]** Nvidia Quadro P6000 (24GB GDDR5X).
- **`perf_counter()`** for CPU runtime profiling.
- **`NVProf`** for GPU kernel runtime profiling.

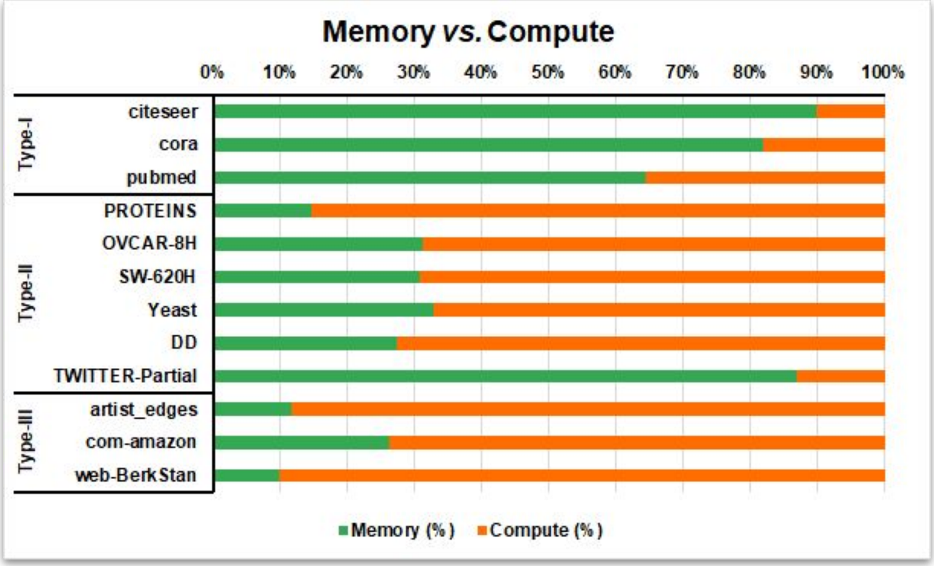
	Dataset	Node	Edges	Dim	Classes	Avg Degree
Type I	Citeseer	3,327	9,464	3703	6	2.84
	Cora	2,708	10,858	1433	7	4.01
	Pubmed	19,717	88,676	500	3	4.50
	PPI	56,944	818,716	50	121	14.38
Type II	PROTEINS_full	43,471	162,088	29	2	3.73
	OVCAR-8H	1,890,931	3,946,402	66	2	2.09
	Yeast	1,714,644	3,636,546	74	2	2.12
	DD	334,925	1,686,092	89	2	5.03
	TWITTER-Partial	580,768	1,435,116	1323	2	2.47
	SW-620H	1,889,971	3,944,206	66	2	2.09
Type III	amazon0505	410,236	4,878,875	96	22	11.89
	artist	50,515	1,638,396	100	12	32.43
	com-amazon	334,863	1,851,744	96	22	5.53
	soc-BlogCatalog	88,784	2,093,195	128	39	23.58
	amazon0601	403,394	3,387,388	96	22	8.40

Runtime Decomposition

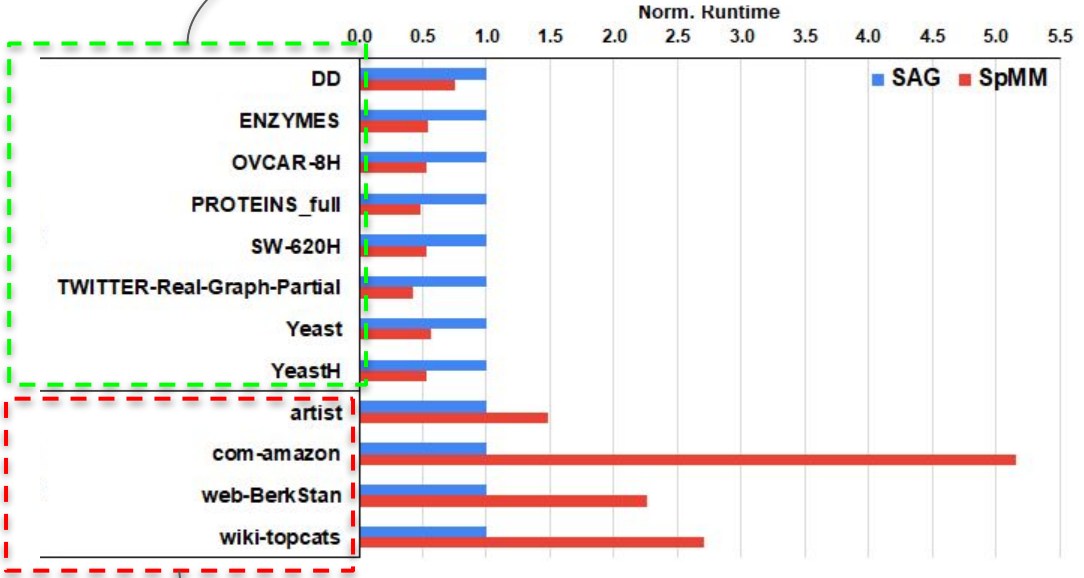


- Intelligent Kernel Switch
  + Graph Preprocessing Cost
- GPU Kernel Cost

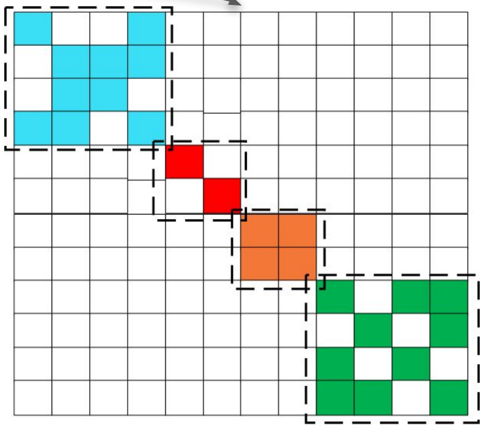
- **Memory**
  - **Allocation**
  - **CPU Host <-> GPU**

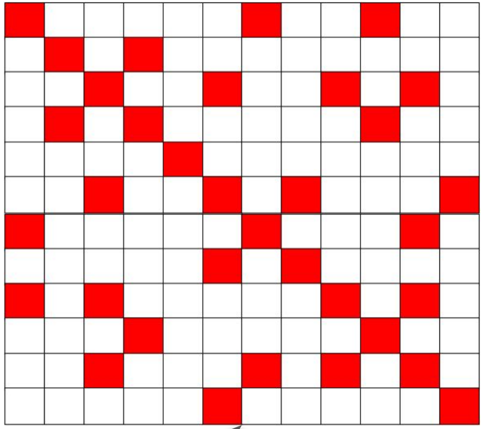


SpMM vs SAG

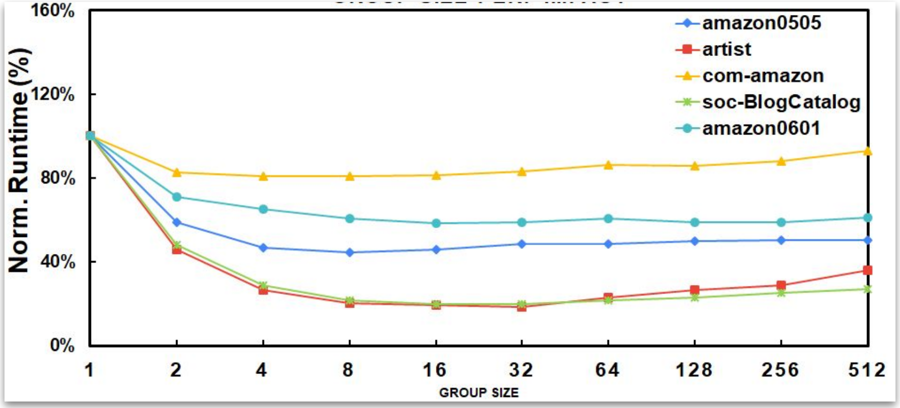


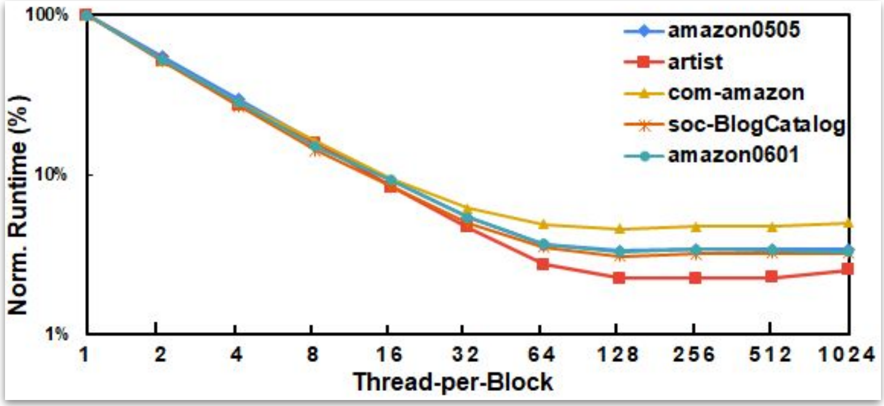
Note: we normalize runtime of SpMM *w.r.t.* SAG

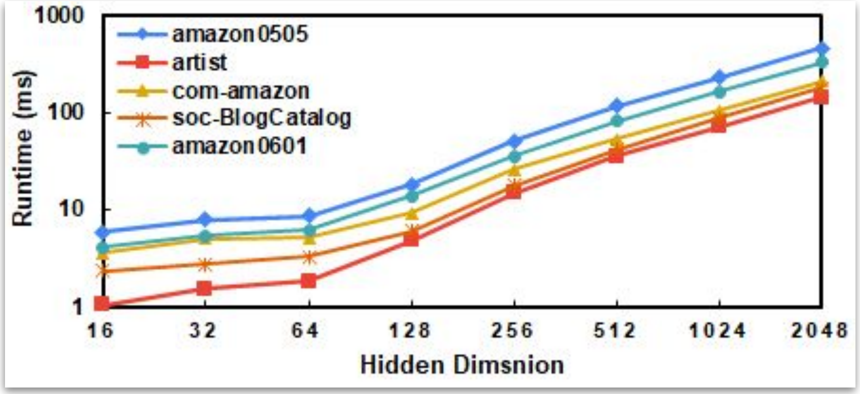




Case Studies







Conclusion

- Surveyed different runtime systems / python wrappers popularized from the recent “AI” wave.

- Built a GCN runtime on GPUs.
  - Scatter and Gather (SAG) Kernel
  - Sparse matrix multiplication (SpMM) kernel
  - Pytorch mixed CUDA/C++/Python wrapper
  - Runtime decision making on graph characteristics. **AUTO** and **Manual** Kernel Selection.
  - Performance tuning options. **Group-size** and **thread-per-block**.

- A complete profiling toolset (Profiler + Report Generator) for CPU host and GPU kernel evaluation.

Thank You

Q&A

Data Science Workhorse -- NumPy

- All-in-one solution for data storage, matrix/tensor math.

```
>>> A = np.array( [[1,1],
...             [0,1]] )
>>> B = np.array( [[2,0],
...             [3,4]] )
>>> A * B                       # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B                       # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)                    # another matrix product
array([[5, 4],
       [3, 4]])
```

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                            # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                            # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                         # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

How Runtime System Helped in this Revolution?

- Adding specific runtime compiling supports for these people. Study their programming habits / code usage pattern, and design specific runtime optimization strategies accordingly. Numba is a good example.

Python Wrapper Variant -- Numba

- A Just in Time compiler that works best on python code that uses NumPy arrays, functions, and loops.
- Use python decorators as hint to Numba compiler.
- Unlike Pytorch, TensorFlow, etc, no support for autograd.

```
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

https://numba.pydata.org/numba-doc/latest/user/5minguide.html

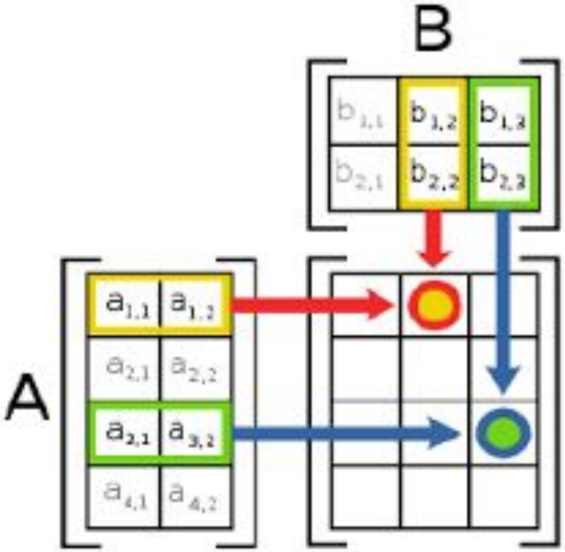
Matrix Multiplication Optimization As An Example

- Naive iterative matrix multiplication algorithm involves two big for loops. Compute for one C_ij at one time step.

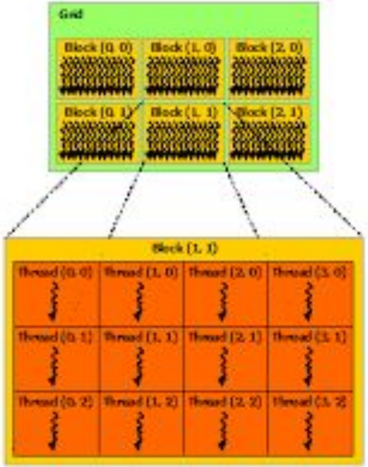
$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}.$$

- Input: matrices A and B
- Let C be a new matrix of the appropriate size
- For i from 1 to n:
    - For j from 1 to p:
        - Let $\mathrm{sum} = 0$
        - For k from 1 to m:
            - Set $\mathrm{sum} \leftarrow \mathrm{sum} + A_{ik} \times B_{kj}$
        - Set $C_{ij} \leftarrow \mathrm{sum}$
- Return C

Matrix Multiplication Optimization As An Example



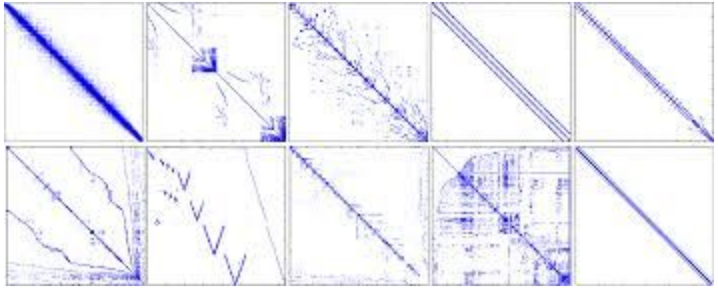
- Can be accelerated using CUDA. Compute all C_ij at the same time. Put A and B in shared GPU memory. Each thread writes into one C_ij.

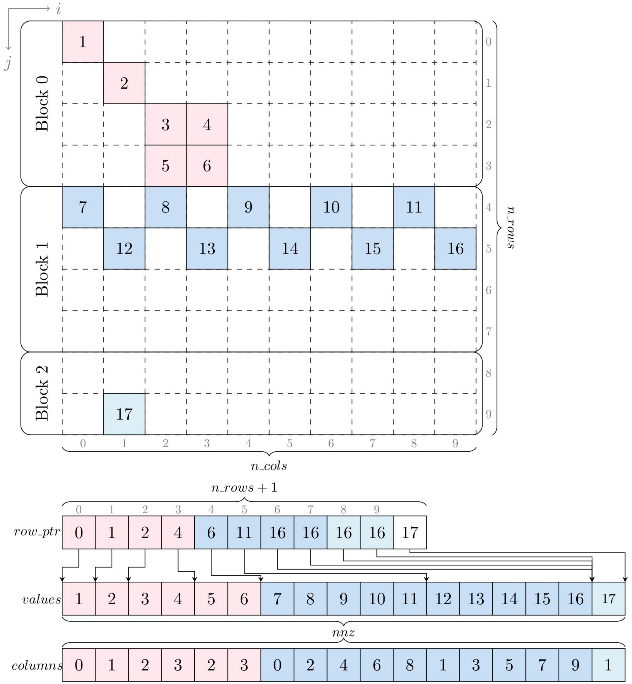


Matrix Multiplication Optimization As An Example

- Can also introduce parallelism by using block matrix multiplication and exploiting sparsity of a matrix, etc.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$





Mathematical Formulation of GCN

- Every neural network layer can then be written as a non-linear function

$$H^{(l+1)} = f(H^{(l)}, A)\,,$$

with $H^{(0)} = X$ and $H^{(L)} = Z$ (or z for graph-level outputs), L being the number of layers. The specific models then differ only in how $f(\cdot, \cdot)$ is chosen and parameterized.

- As an example, let's consider the following very simple form of a layer-wise propagation rule:

$$f(H^{(l)}, A) = \sigma\left(AH^{(l)}W^{(l)}\right)\,,$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\cdot)$ is a non-linear activation function like the ReLU. Despite its simplicity this model is already quite powerful (we'll come to that in a moment).

Think of A as an Adjacency Matrix describing your Graph. W is the trainable weight.

https://tkipf.github.io/graph-convolutional-networks/

Python Wrapper Variant -- PyCuda

- Full NVIDIA CUDA API in Python.
- Work with Numpy
- Garbage collector support. Object cleanup tied to object’s lifetime.
- Automatic Error Checking. Python exceptions. More Traceable Errors.

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

https://documen.tician.de/pycuda/

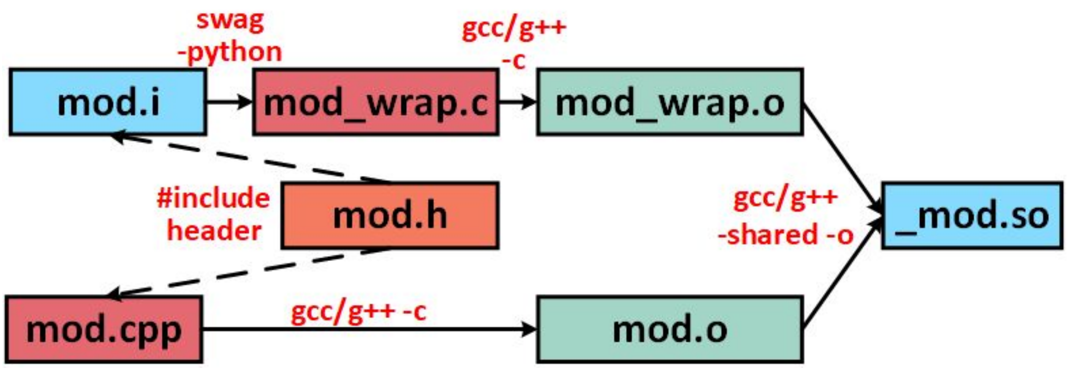
Python Wrapper Variant -- scikit-CUDA

- Similar to PyCUDA (backed by Nvidia).
- Less developed.
- With scikit-learn, SciPy open source community.

https://scikit-cuda.readthedocs.io/en/latest/

Python Wrapper Variant -- SWIG

- Generic Wrapper. Statically Compiled as a .so library that you can use at runtime.



https://intermediate-and-advanced-software-carpentry.readthedocs.io/en/latest/c++-wrapping.html

Python Wrapper Variant -- PyTorch

- One of the most popular Auto-grad Frameworks among Machine Learning researchers.
- Torch tensor objects wrap around Numpy objects with Auto-grad support.

```
>>> torch.tensor([[1., -1.], [1., -1.]])
tensor([[ 1.0000, -1.0000],
        [ 1.0000, -1.0000]])
>>> torch.tensor(np.array([[1, 2, 3], [4, 5, 6]]))
tensor([[ 1,  2,  3],
        [ 4,  5,  6]])
```

```
>>> cuda0 = torch.device('cuda:0')
>>> torch.ones([2, 4], dtype=torch.float64, device=cuda0)
```

```
>>> x = torch.tensor([[1., -1.], [1., 1.]], requires_grad=True)
>>> out = x.pow(2).sum()
>>> out.backward()
>>> x.grad
tensor([[ 2.0000, -2.0000],
        [ 2.0000,  2.0000]])
```

d(x^2)/dx = 2x

https://pytorch.org/docs/stable/tensors.html

Implementation Details, list of wrapper choices

- Numba (runtime compiler specifically optimize for Numpy math code, GPU)
- PyCuda
- scikit-CUDA
- SWIG
- Pytorch
- etc…..

Implementation Details, list of wrapper choices

- Numba (runtime compiler specifically optimize for math numpy code)
- PyCuda (Full CUDA API in Python, GC, Try-Catch CUDA errors, etc)
- scikit-CUDA
- SWIG
- Pytorch
- etc…..

Implementation Details, list of wrapper choices

- Numba (runtime compiler specifically optimize for math numpy code)
- PyCuda (Full CUDA API in Python, GC, Try-Catch CUDA errors, etc)
- scikit-CUDA (similar to PyCuda, only PyCuda is backed by Nvidia)
- SWIG
- Pytorch
- etc…..

Implementation Details, list of wrapper choices

- Numba (runtime compiler specifically optimize for math numpy code)
- PyCuda (Full CUDA API in Python, GC, Try-Catch CUDA errors, etc)
- scikit-CUDA (similar to PyCuda, only PyCuda is backed by Nvidia)
- SWIG (Simple Wrapper Interface Generator, statically compiled .o shared library)
- Pytorch
- etc…..