Glow: Graph Lowering Compiler Techniques for Neural Networks

발표: 김정호

구성

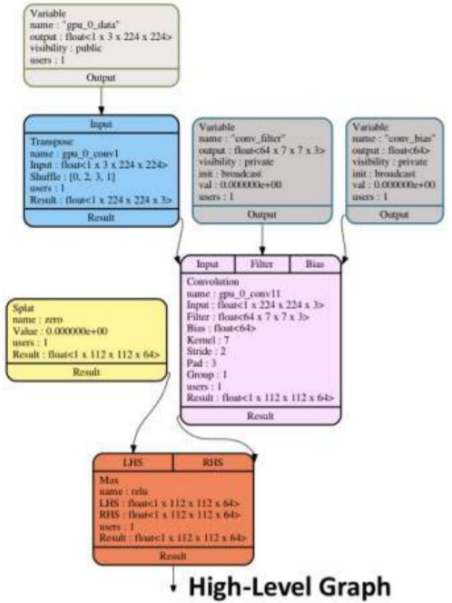
1. Intro
2. Glow Compiler
    a. Overall flow
    b. High-level IR
    c. Node lowering
    d. Low-level IR
    e. Optimization lists
3. Glow Runtime
4. Evaluation

1. Intro

- Graph Lowering
  - 다양한 NN Op을 Linear Algebra Op으로 분해하여 Backend의 NN Op 구현부담을 줄이자
- Facebook
  - PyTorch의 backend
- 주로 서버를 타겟팅
  - NXP에서는 MCU,DSP를 target하며 적극적으로 사용 중
- Offline SDK
  - AOT, JIT 모두 지원
  - ONNX, Caffe2
  - post-training quantization
  - target-Independent/dependent optimizations
  - CPU, GPU, DSP, NPU code generations
- Runtime
  - single interface로 여러 디바이스(카드)와 Host를 관리

2. Glow IRs

- High-level IR => Low-level IR => Backend CodeGen




```
declare {
 %input = weight float<8 x 28 x 28 x 1>, broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>, xavier, 25.0
 %filter0 = weight float<16>, broadcast, 0.100
 %weights = weight float<10 x 144>, xavier, 144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo, @in %input,
     @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in %allo0,
     @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

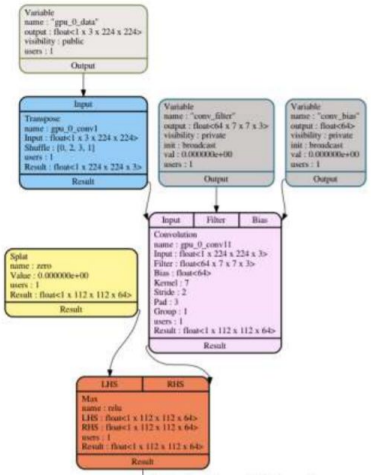
Low-Level IR

```
LBB14_1:
  vmovaps 3211264(%rcx,%rax,4), %ymm1
  vmovaps 3211296(%rcx,%rax,4), %ymm2
  vmovaps 3211328(%rcx,%rax,4), %ymm3
  vaddps 6422528(%rcx,%rax,4), %ymm1, %ymm1
  vaddps 6422560(%rcx,%rax,4), %ymm2, %ymm2
  vmovaps 3211360(%rcx,%rax,4), %ymm4
  vaddps 6422592(%rcx,%rax,4), %ymm3, %ymm3
  vaddps 6422624(%rcx,%rax,4), %ymm4, %ymm4
  vmaxps %ymm0, %ymm1, %ymm1
  vmaxps %ymm0, %ymm2, %ymm2
  vmaxps %ymm0, %ymm3, %ymm3
  vmovaps %ymm1, 6422528(%rcx,%rax,4)
  vmovaps %ymm2, 6422560(%rcx,%rax,4)
  vmaxps %ymm0, %ymm4, %ymm1
  vmovaps %ymm3, 6422592(%rcx,%rax,4)
  vmovaps %ymm1, 6422624(%rcx,%rax,4)
  addq $32, %rax
```

Machine Code

2. Glow Compiler

1. Graph Loading
=> High-level IR construction



High-Level Graph

2. High-level IR 최적화
Target independent/dependent

3. Node Lowering
NN Op => Linear algebra ops

4. Additional optimizations in high-level IR

5. IRGen:
High-level IR => Low-level IR

```
declare {
 %input = weight float<8 x 28 x 28 x 1>, broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>, xavier, 25.0
 %filter0 = weight float<16>, broadcast, 0.100
 %weights = weight float<10 x 144>, xavier, 144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo, @in %input,
     @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in %allo0,
     @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

Low-Level IR

6. Memory-related optimizations

```
LBB14_1:
  vmovaps 3211264(%rcx,%rax,4), %ymm1
  vmovaps 3211296(%rcx,%rax,4), %ymm2
  vmovaps 3211328(%rcx,%rax,4), %ymm3
  vaddps 6422528(%rcx,%rax,4), %ymm1, %ymm1
  vaddps 6422560(%rcx,%rax,4), %ymm2, %ymm2
  vmovaps 3211360(%rcx,%rax,4), %ymm4
  vaddps 6422592(%rcx,%rax,4), %ymm3, %ymm3
  vaddps 6422624(%rcx,%rax,4), %ymm4, %ymm4
  vmaxps %ymm0, %ymm1, %ymm1
  vmaxps %ymm0, %ymm2, %ymm2
  vmaxps %ymm0, %ymm3, %ymm3
  vmovaps %ymm1, 6422528(%rcx,%rax,4)
  vmovaps %ymm2, 6422560(%rcx,%rax,4)
  vmaxps %ymm0, %ymm4, %ymm1
  vmovaps %ymm3, 6422592(%rcx,%rax,4)
  vmovaps %ymm1, 6422624(%rcx,%rax,4)
  addq $32, %rax
```

Machine Code

7. Code generation

2. Glow Compiler

- *High-level IR*
  - 일반적인 node-based graph 형태
  - Graph-level target-independent/dependent 최적화
  - Node lowering

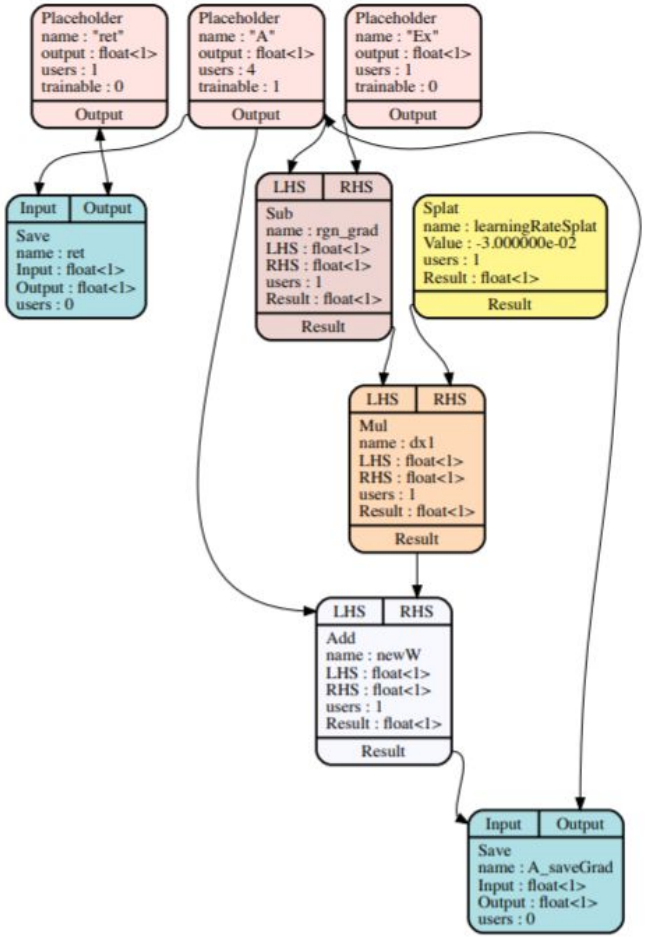


Figure 2: A lowered compute graph in Glow's high-level IR, representing a regression of A, automatically differentiated by Glow.

2. Glow Compiler

- *Node Lowering* (in high-level IR)
  i. 프레임워크에서는 똑같은 op(e.g. ReLU, SGD)을 GPU, CPU, DSP 등 backend 마다 구현해야하는 부담이 있다.
  ii. Glow에서는 backend마다 op를 따로 구현하는 대신 high-level op을 바로 실행시키지 않고 "Node Lowering"으로 NN op node를 linear algebra op nodes로 변환한다.
    1. e.g. Fully-Connected Layer ⇒ MatMul + Add
  iii. Lowering이 low-level IR이 아닌 high-level graph에서 이뤄지는 이유
    1. newly-lowered graph에 graph-level 최적화
    2. instruction scheduling
    3. lowered graph에 대해 target-specific 최적화

2. Glow Compiler

- *IRGen*: Lowered high-level IR -> Low-level IR
- *Low-level IR*
  - instruction-based representation
    - with memory(address)
  - optimizations related to memory
    - static memory allocation
    - scheduling for memory op. latency hiding
    - copy elimination
    - ...

```
declare {
 %input = weight float<8 x 28 x 28 x 1>,
     broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>,
     xavier, 25.0
 %filter0 = weight float<16>, broadcast,
     0.100
 %weights = weight float<10 x 144>, xavier,
     144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo,
     @in %input, @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in
     %allo0, @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

Figure 3: Unoptimized low-level Glow IR.

2. Glow Compiler

- function은 *declare* section과 *program* section으로 구성
  - *declare*
    - lifetime이 프로그램 전체로 지정되는 memory region을 선언 (global variable in C 같은 부분)
  - *program*
    - locally allocated memory regions
    - qualifiers
      - copy elimination이나 buffer sharing같은 optimization 시 optimizer가 참조
      - @in: “read from”
      - @out: “written into”
      - @inout: “free to use”

```
declare {
 %input = weight float<8 x 28 x 28 x 1>,
     broadcast, 0.0
 %filter = weight float<16 x 5 x 5 x 1>,
     xavier, 25.0
 %filter0 = weight float<16>, broadcast,
     0.100
 %weights = weight float<10 x 144>, xavier,
     144.0
 %bias = weight float<10>, broadcast, 0.100
 %selected = weight index<8 x 1>
 ...
 %result = weight float<8 x 10>
}

program {
 %allo = alloc float<8 x 28 x 28 x 16>
 %conv = convolution [5 1 2 16] @out %allo,
     @in %input, @in %filter3, @in %bias0
 %allo0 = alloc float<8 x 28 x 28 x 16>
 %relu = max0 @out %allo0, @in %allo
 %allo1 = alloc index<8 x 9 x 9 x 16 x 2>
 %allo2 = alloc float<8 x 9 x 9 x 16>
 %pool = pool max [3 3 0] @out %allo2, @in
     %allo0, @inout %allo1
 ...
 %deal6 = dealloc @out %allo6
 %deal7 = dealloc @out %allo7
 %deal8 = dealloc @out %allo8
 %deal9 = dealloc @out %allo9
}
```

Figure 3: Unoptimized low-level Glow IR.

2. Glow Compiler

- Graph optimization lists
  - DCE/CSE
  - Transpose/Concat nodes
    - fusion, folding, elimination
  - Pooling nodes
    - ReLU와 Pool의 순서를 바꿔서 Pooling의 입력이 작아지도록
  - Regression nodes
    - inference 시 bypass

2. Glow Compiler

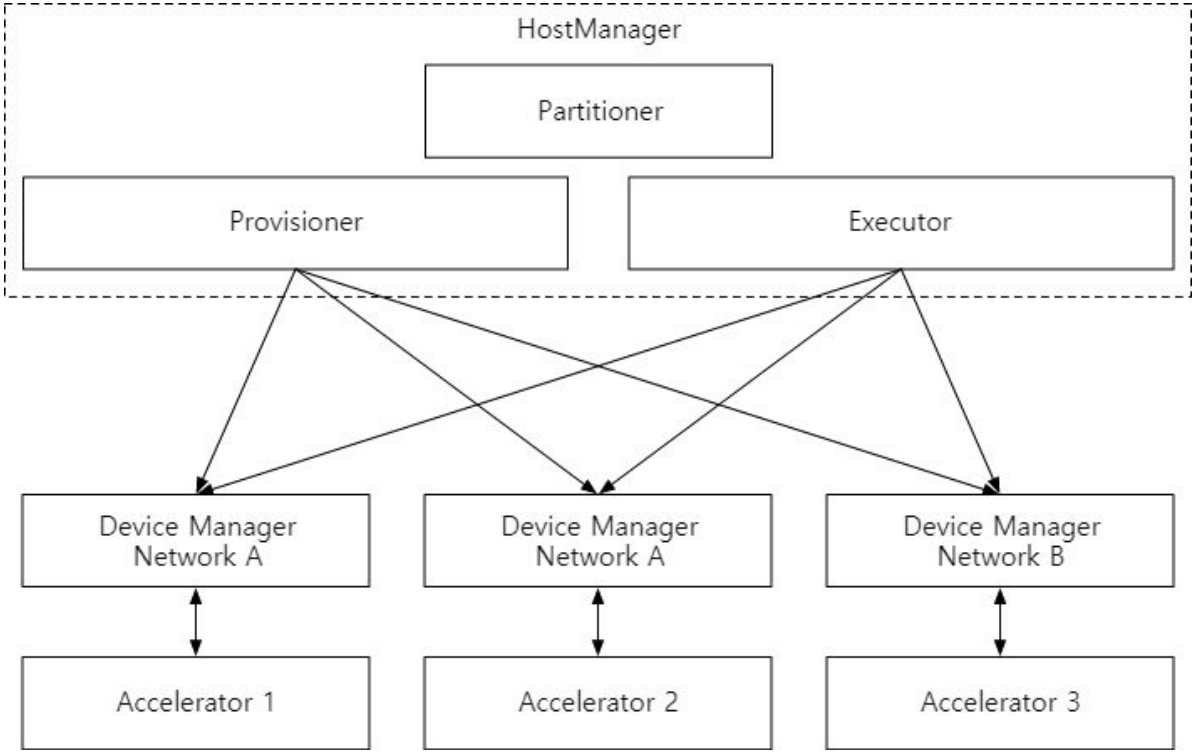
- Quantization specific optimization lists
  - folding, remove operations
    - *Quantize*(*Dequantize*(X)) -> *RescaleQuantized*(X)
    - remove *Dequantize*(*Quantize*(X))
    - fold *RescaleQuantized* operator up into the operation
      - Add, Mul, Div, Conv…
    - fold arithmetic op into BatchNorm

2. Glow Compiler

- Memory-related optimizations
  - DSE
  - Deallocations hoisting
  - Buffer sharing
  - Operator Stacking
    - e.g. sequence of element-wise operations(e.g. ReLU, Add, Sub) => single kernel
  - ...

3. Glow Runtime

- Partitioner
- Provisioner
- DeviceManager
- Executor



3. Glow Runtime

- Partitioner
  - slice a graph into multiple sub-graphs depending on the backend
    - memory constraints, cost model, communications between sub-models
- Provisioner
- DeviceManager
- Executor

3. Glow Runtime

- Partitioner
- Provisioner
  - Sub-graph들을 device(card)에 매핑
  - JIT compilation
  - Data-level, model-level을 모두 고려
- DeviceManager
- Executor

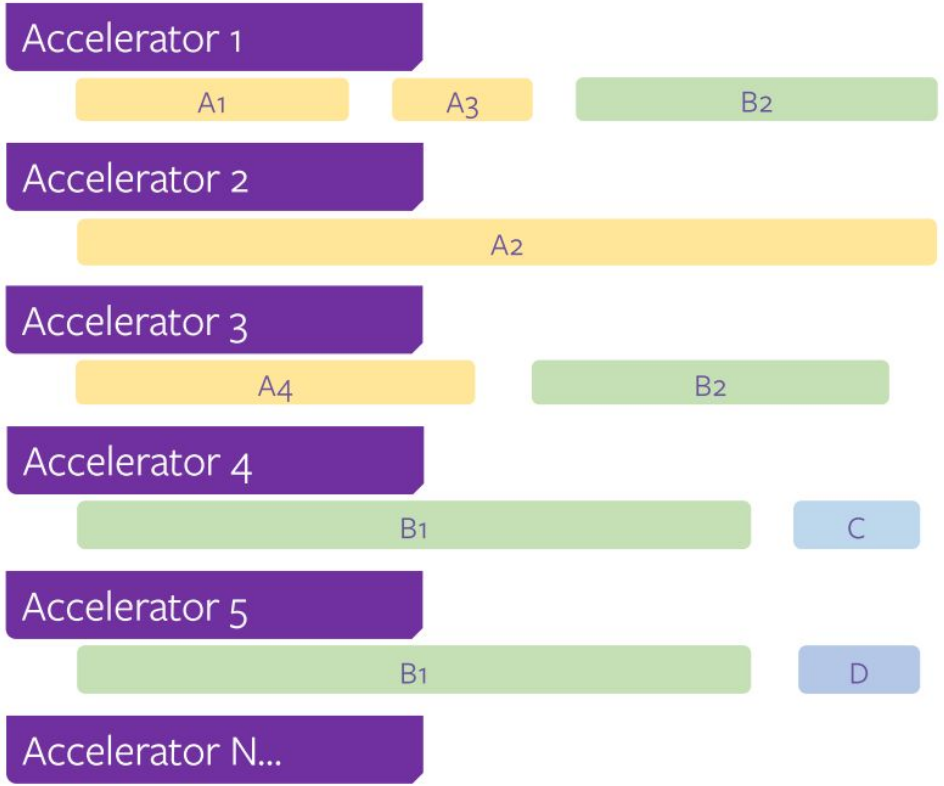


Figure 10: Four networks (A, B, C, D) have been assigned to five accelerators. A and B have been partitioned, and B has been duplicated on multiple accelerators.

3. Glow Runtime

- Partitioner
- Provisioner
- DeviceManager
  - HW abstraction
  - handles model loading, memory transfer
  - tracks HW state
- Executor

3. Glow Runtime

- Partitioner
- Provisioner
- DeviceManager
- Executor
  - handles the execution of a network
  - 각 sub-network의 실행 상태 (asynchronous)를 관리하고 입출력을 전파한다.
  - 각 sub-network의 input이 준비되는지 모니터링하고 trigger

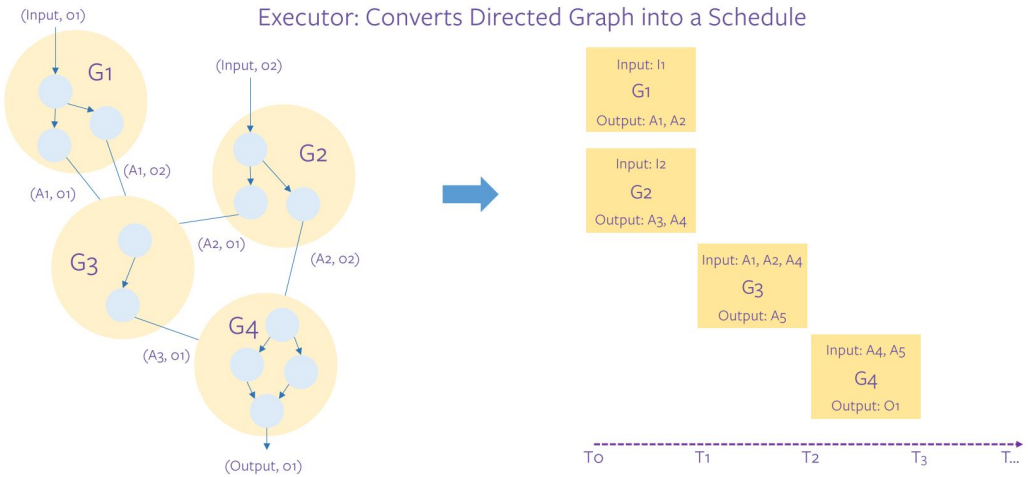


Figure 9: A simple example showing a graph partitioned into multiple sub-graphs, themselves making up a directed graph, and then converted into a schedule by the executor.

4. Evaluation

- Models: Resnet50, VGG19
- Kaby Lake i7-7600U에서 Glow, TensorFlow-1.7, TVM 을 비교
  - TF with XLA enabled
  - TVM with LLVM 6.0
    - auto-tuning disabled
- TF는 Eigen을 이용하는데 Eigen은 im2col과 matmul을 이용하여 Conv를 동작시키지만, Glow는 바로 Conv를 컴파일하기 때문에 im2col 오버헤드가 없고 shape-aware code-gen을 했기 때문에 2.7배 FPS를 보였다.
- TVM보다는 30% 정도 빨랐는데, TVM도 im2col overhead는 없지만 다른 auto-tuning같은 최적화를 적용하지 않았기 때문인 것으로 보인다.

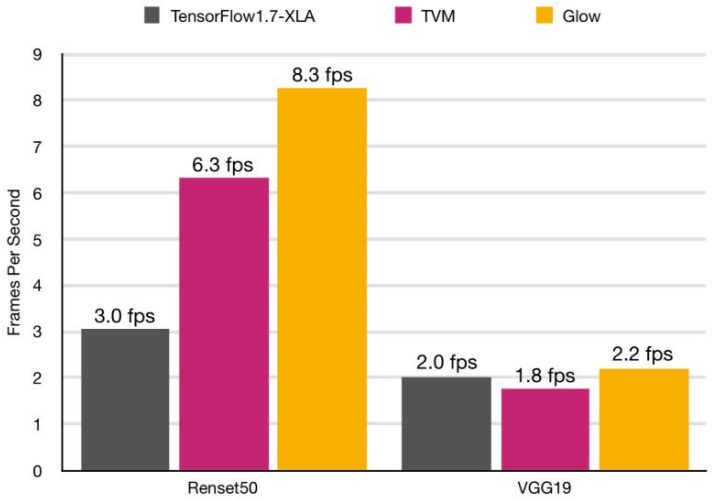


Figure 11: Glow vs. TensorFlow-1.7 and TVM on an Intel® Core i7-7600U; frames per second on a single thread.

4. Evaluation

1. Auto-tuning 기반의 TVM과는 다른 접근방법
2. 생각보다 큰 규모
3. API 형태로 구성
4. 단순한 컴파일을 넘어 서비스를 위한 Runtime을 많이 고려한 것으로 보임

Reference

1. https://github.com/pytorch/glow
2. https://www.slideshare.net/YiHsiuHsu/glow-introduction
3. https://arxiv.org/pdf/1805.00907.pdf

감사합니다.