++

C++로 구현하는 ML Framework

김재우

직접 딥러닝 프레임워크를 개발하게 된 이유

1. 딥러닝의 기본적인 원리에 대한 이해도 증가
   - 연산 커널을 비롯한 알고리즘을 모두 직접 구현

2. 직접 사용할 때 입맛에 따라 변형하고 최적화하기 쉬움
   - 모든 것을 마음대로 커스터마이징 가능

3. Python 대신 C++ 을 사용하여 더 빠른 성능을 구현할 수 있다
   - 최적화의 자유도가 매우 높음

4. 선호하는 사용자 인터페이스 구현이 가능하다

지금까지의 개발 과정

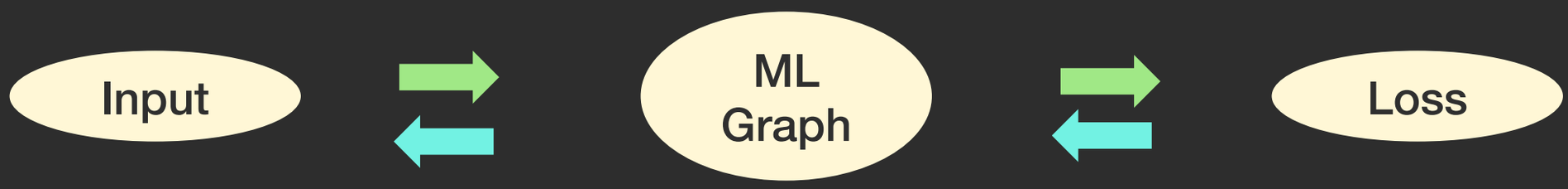
CubbyDNN
- 초기에 가장 많은 시행착오를 겪은 모델
- 여러 번 코드를 갈아엎음...
- 템플릿 기반 vs float 고정
- 최적화 vs 안정성

Takion
- 템플릿 기반으로 다양한 타입 활용 가능
- Compile and run 방식 채택 (모델을 완전히 빌드하고 실행)
- SIMD 명령어를 이용한 최적화 진행

Sapphire
- float 로 타입 고정
- Interpreter 방식으로 실행
- GPU 지원 추가
- Sparse & Dense matrix 사이 선택 가능 (예정)

딥러닝의 진행 과정

1. Training data 를 Load 하고 모델에 알맞게 변형
2. 모델에 변형된 data 를 입력하고 Label과 비교하여 Loss 를 계산
3. Loss 를 이용해 각 가중치(parameter) 의 편미분을 계산 후 업데이트
4. 1~3 단계를 반복



딥러닝 프레임워크가 갖추어야 할 요소

1. **행렬 연산 유닛**
   - 딥러닝에 필요한 연산은 대부분 행렬 연산에 기반함
   - 행렬 연산 능력이 framework 의 성능을 좌우함

2. **딥러닝 연산 유닛**
   - Dense, CNN, Softmax 등등 딥러닝에 필수적인 연산 구현
   - 딥러닝의 원리에 대한 깊은 이해가 요구됨

3. **연산 그래프**
   - 딥러닝에 필요한 연산 유닛을 연결하는 그래프
   - 연산 유닛간 복사, 및 동기화 과정 필요
   - 역전파 연산을 위한 Back tracking 기능

4. **Data Loader 및 변환 도구**
   - 사용자가 쉽게 Data 를 변형하고 취급할 수 있도록 도움
   - Image class, string class 등이 해당됨

Takion

초기 목적

1. C++ 프로그래머들이 쉽고 직관적으로 사용 가능한 프레임워크
2. CPU 벡터 연산을 통해 빠른 속도
3. 다양한 자료형 지원
4. 손쉬운 확장 및 재사용성

Design Decisions

1. Compile and run 방식 채택

- 미리 그래프의 구조를 정의하므로 실행 중 메모리를 동적으로 할당하지 않아도 됨
- 그래프의 구조를 알고 있으므로 프레임워크 내부적으로 연산 유닛 병렬화 가능
- Heuristic 을 통한 연산 그래프 최적화 가능

2. 템플릿 기반으로 제작

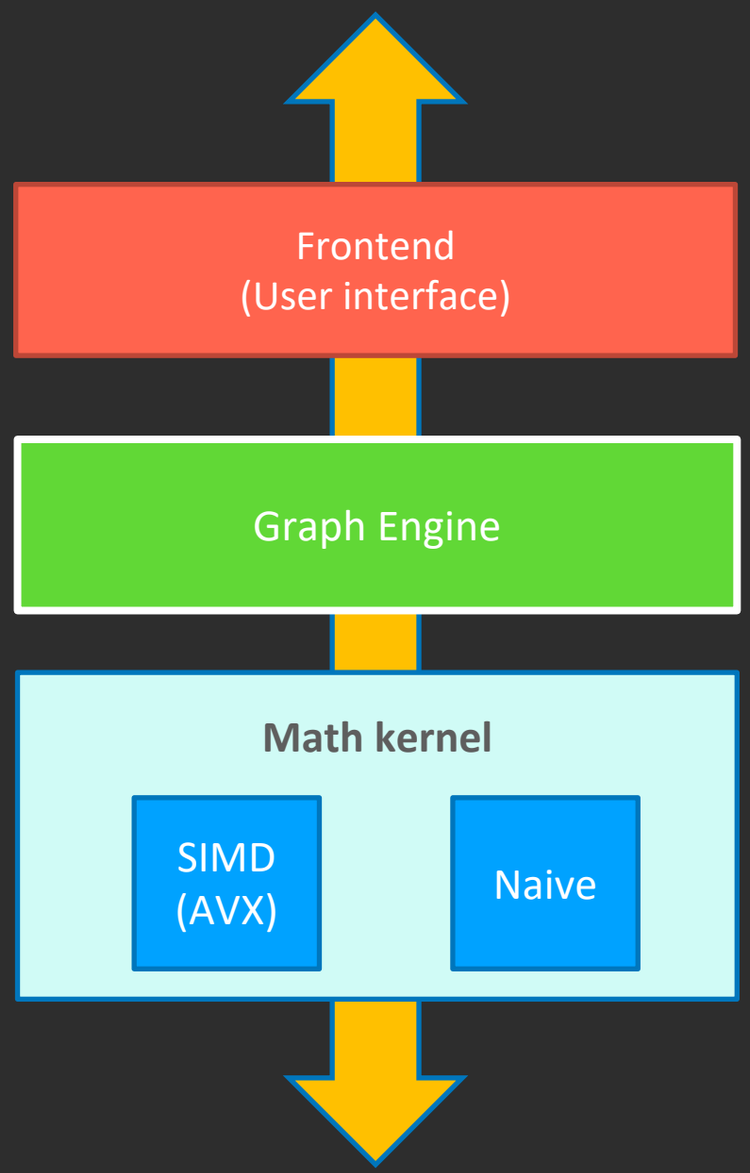
- float 외 다양한 자료형을 지원하기 위해 채택
- 타입을 꼭 정해야 하는 부분에 한하여 타입별로 코드 분리 (SIMD 행렬 연산 등)

Takion 의 구조

- **Frontend**
  - 유저 인터페이스
  - 연산 그래프 제작

- **Graph Engine**
  - 연산 그래프 관리 및 실행
  - 각 노드 사이의 synchronization 및 data copy 담당

- **Math kernel**
  - 벡터 및 행렬 연산 담당
  - Data parallelism을 통한 가속화



User interface – Takion

- 최대한 간소화, 직관적인 디자인을 목표

- 가장 기본적인 요소들만 정의하면 나머지는 Takion 이 알아서 처리

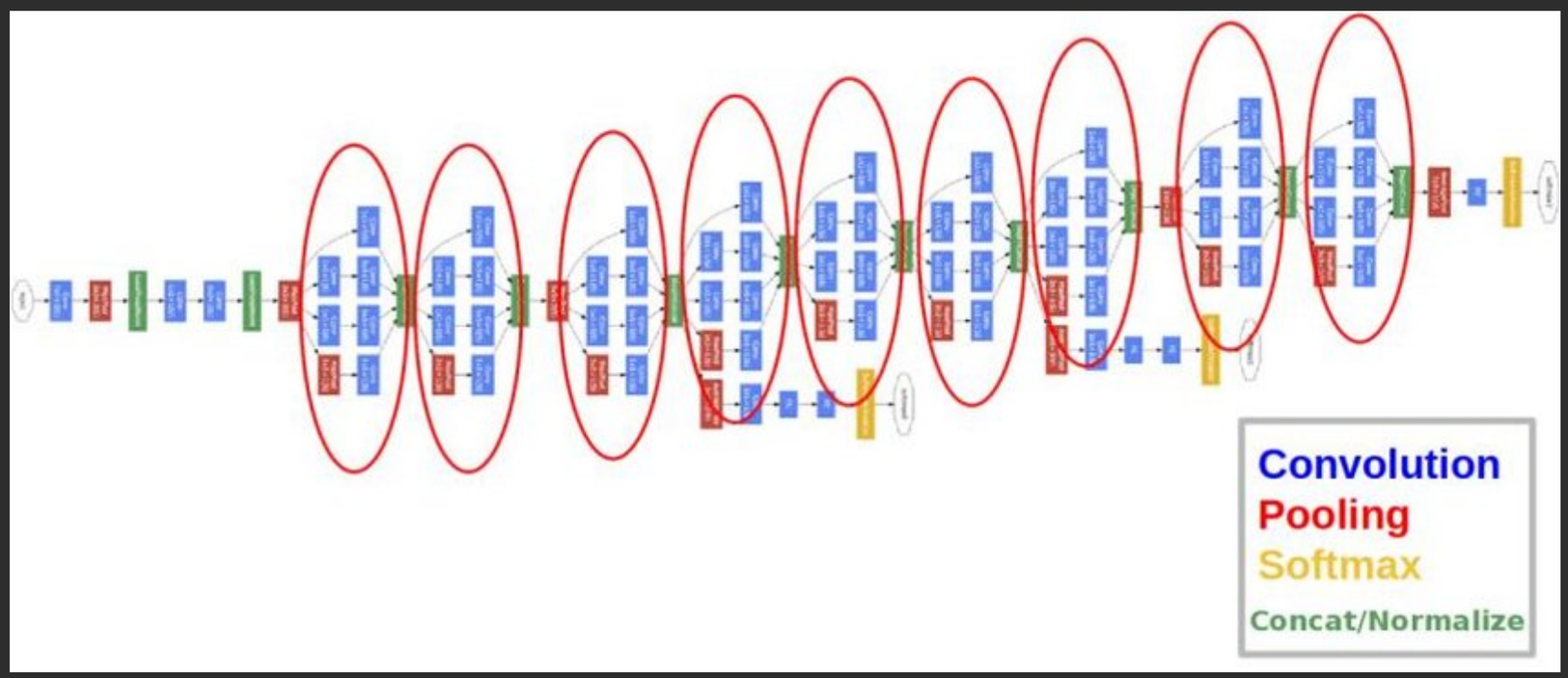
```
Model<float> model(Compute::Device(0, Compute::DeviceType::CPU, "device0"),
                   10);

auto tensor =
    model.Constant(Shape({ 200 }), std::vector<float>(2000, 10), "input");
const auto label = model.Constant(Shape({ 3 }), std::vector<float>(30, 1),
                                  "label");

tensor = model.Dense(tensor, 100);
tensor = model.ReLU(tensor);
tensor = model.Dense(tensor, 25);
tensor = model.ReLU(tensor);
tensor = model.Dense(tensor, 3);
tensor = model.ReLU(tensor);
model.MSE(tensor, label, "MseLoss");

model.Compile("SGD", Parameter({}, { { "epsilon", 0.00001f } }, {}));
model.Fit(10000);
```

Graph structure

Graph 의 요구사항

1. Forward 및 Backward 연산이 가능해야 함 (양방향으로 실행 가능)
2. 동시에 실행가능한 부분을 병렬화
3. 연산 유닛 간 동기화 및 Data Copy
4. 그래프를 컴파일하면 필요한 연산 유닛들이 내부적으로 할당됨

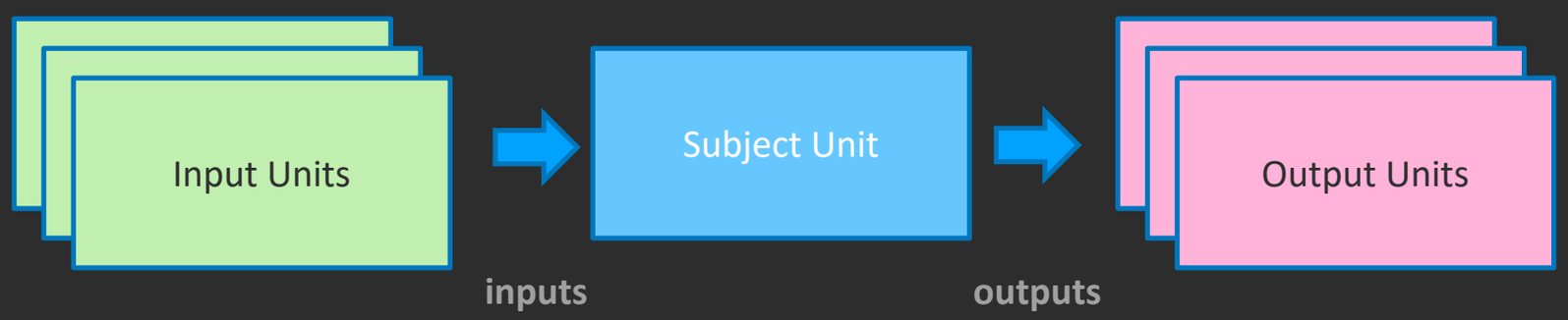
예) GoogleNet 의 경우 모델 중간에 병렬로 실행가능한 유닛들이 존재



Graph

그래프의 실행 과정

1. Subject유닛은 자신과 연결된 유닛들의 상태를 참조하여 모든 input 유닛들이 연산을 마쳤는지 확인
2. Input 유닛의 연산이 끝났다면 Input 유닛의 출력을 Subject 유닛으로 복사
3. Subject 유닛이 연산 완료 후 Ready state 가 되면 결과 data 를 output unit 으로 복사
4. 1 ~ 3 번 과정이 반복됨 (Back propagation의 경우 반대 방향으로 진행)
5. 복사 시 연산 유닛의 형태가 다르다면 알맞은 형태로 data 변환 후 복사
   - SIMD 연산의 경우 alignment 를 지켜야 함



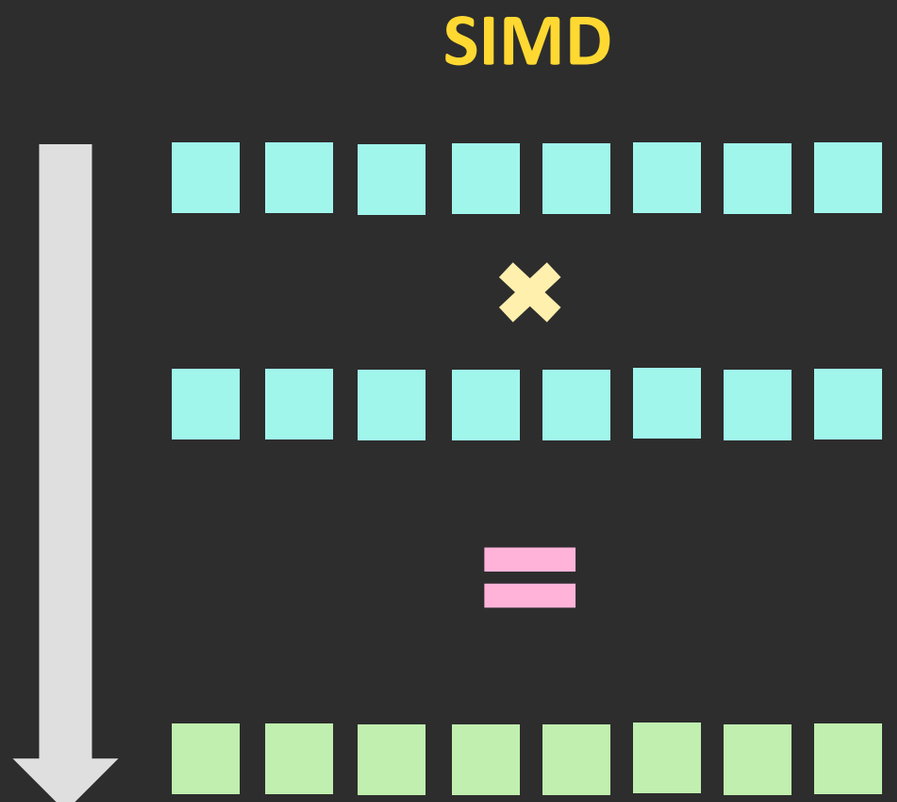
Math Kernel

연산 커널의 요구 사항

1. GEMM 연산이 가능해야 함 (C = A * B + C)
2. 빠르게 실행이 가능해야 함

Math kernel 의 설계

1. SIMD(AVX/AVX2) 명령어 셋을 이용하여 최적화
2. 캐시 메모리 효율을 높이기 위한 Blocking 적용
3. OpenMP 를 이용한 Batch-Level 병렬화



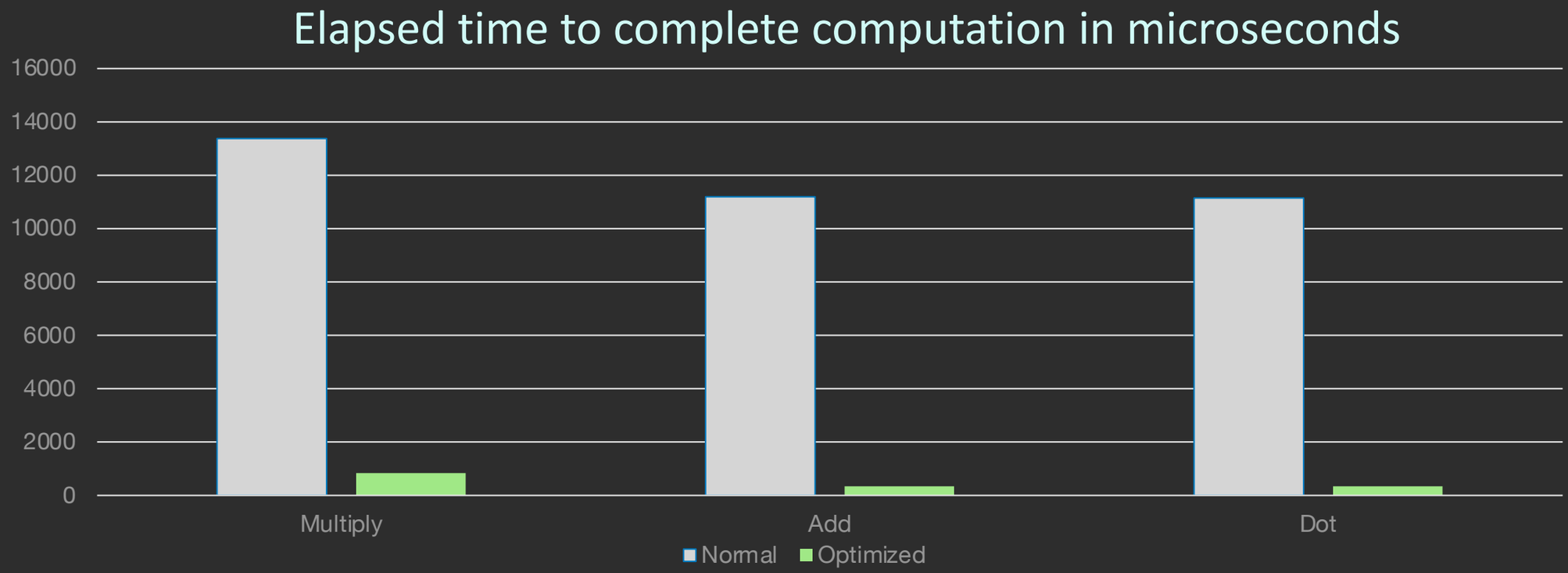
SIMD

- Single Instruction Multiple Data 의 약자
- 명령어 한 번으로 여러 번의 공통된 연산 수행 가능

예) 256bit AVX 연산의 경우 최대 8개의 32bit floating point 연산을 같은 사이클에 실행 가능

Math kernel

행렬 연산 최적화 결과 (Naïve 한 방식과 비교)
수십배에 달하는 성능 향상 달성



Test environment
Ubuntu Linux 20.04
AMD Ryzen9 3900X

Takion 의 문제점

구현 과정에서 느낀 여러가지 문제점들

1. Compile & run 방식 특성상 유연성이 떨어짐
   - 실행 도중에 그래프 변경 불가능
   - 사용자가 Graph 를 변형하는 자유도 저하
2. CUDA 지원 불가
   - CPU 에서만 연산이 가능함
3. Template 구조 때문에 코드가 복잡해짐 (타입별로 다르게 처리해야 함)
   - 전체적으로 코드 양이 늘어나고 Binary 가 커짐

그렇다면..

1. Compile 과정을 생략하여 유연성을 높여보자
2. CUDA 를 default 로 사용할 수 있도록 해 보자
3. Template 을 사용하는 대신 32bit float 으로 고정시켜 보자

-> Sapphire 개발 시작!

Sapphire

Design Purpose (현재 개발 중)

1. 사용자에게 훨씬 더 많은 자유도 부여
   - 커스텀 연산 함수 제작 가능
   - 실행 도중 그래프 구조 변경 가능
   - 다른 코드에 쉽게 이식 가능

2. CUDA 지원
   - CPU, CUDA 에서의 연산을 모두 지원
   - Sparse, Dense matrix를 모두 지원

3. Float 타입 통일
   - 하나의 type 만 지원하고 불필요한 코드 중복 제거

4. Tensor level 에서 더 많은 기능 지원
   - Tensor 의 indexing, slicing, stacking 등 다양한 변형 기능 지원
   - python 에서의 Numpy 와 유사한 기능 제공을 목표로 함

Sapphire 의 Tensor

Tensor 만으로도 다양한 연산 지원

- Host, GPU 사이 공통된 연산 인터페이스 지원
  (Gemm, Add, Dot, 초기화, 및 다양한 활성화 함수)

- SendTo 함수를 통해 쉽게
  Tensor의 위치 변환 가능

```
const Shape shapeA({ M, K });
const Shape shapeB({ K, N });
const Shape shapeC({ M, N });
const Shape shapeOut({ M, N });

const Device cuda(0, "device0");
const Device host("host");

TensorUtil::TensorData A(shapeA, Type::Dense, host, batchSize);
TensorUtil::TensorData B(shapeB, Type::Dense, host, batchSize);
TensorUtil::TensorData C(shapeC, Type::Dense, host, batchSize);
TensorUtil::TensorData out(shapeOut, Type::Dense, host, batchSize);

Compute::Initialize::Normal(A, 10, 5);
Compute::Initialize::Normal(B, 10, 5);
Compute::Initialize::Normal(C, 10, 5);

A.SendTo(cuda);  // Host 메모리에서 GPU 메모리로 데이터 복사
B.SendTo(cuda);
C.SendTo(cuda);
out.SendTo(cuda);

Compute::Initialize::Zeros(out);
Compute::Gemm(out, A, B, C); // GPU 에서 GEMM 실행 (out = A*B + C)

A.SendTo(host); // GPU 메모리에서 Host 메모리로 데이터 복사
B.SendTo(host);
C.SendTo(host);
out.SendTo(host);
out.SendTo(host);

Compute::Initialize::Zeros(out);
Compute::Gemm(out, A, B, C); // Host 에서 GEMM 실행 (out = A*B + C)
```

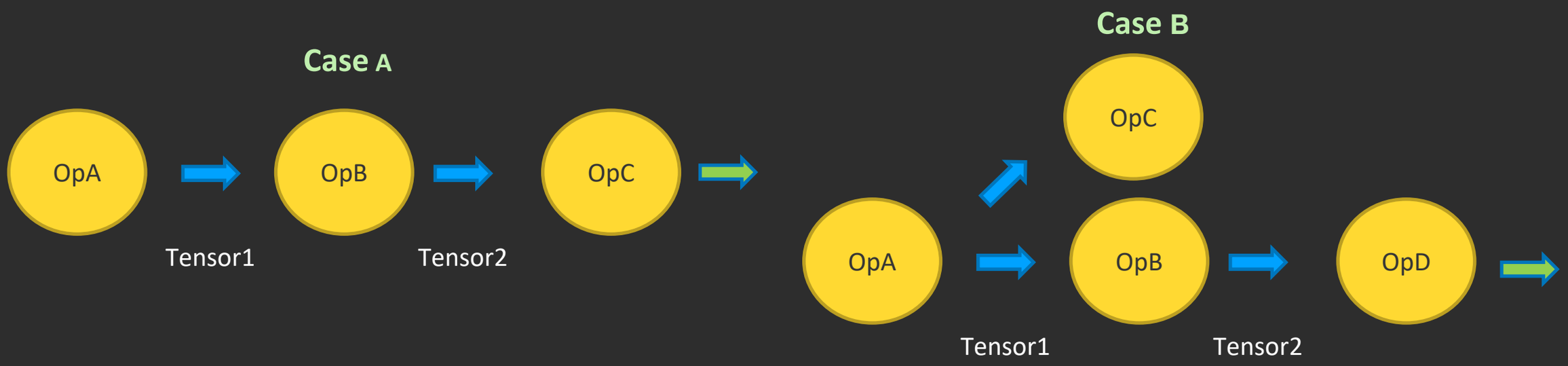
New Graph algrithm

- Interpreter 방식 (실행 즉시 자원을 할당하고 연산 수행)

Tensor1 은 자신이 OpA 의 결과이며, OpB 에서 결과를 전달함
-> Tensor1은 Back propagation 시 OpB 로부터 Gradient 를 받아 OpA 로 전달해야 함

Tensor1은 자신이 OpA 의 결과이며, OpC 및 OpB 에게 결과를 전달함
-> Tensor1 은 Back propagation 시 OpB, OpC 모두에게 Gradient 를 받아야 OpA 로 Gradient 전달 가능

각 Tensor 는 자신이 거쳐 온 Operation 들의 정보를 저장함!
Back propagation 은 Result tensor 에서부터 재귀적으로 실행 가능!



Plans

추가적으로 구현해볼 것들

Sparse matrix 연산 구현
- Sparse matrix 란 0 이 아닌 data 만 표현하는 matrix 표현 방식을 의미
- 메모리를 절약 및 속도 향상 가능

Sapphire 에 AVX 커널 구현
- Takion 의 구현 방식을 기반으로 구현 예정

Visualizer 및 Data loader 를 비롯한 Helper 유틸리티 제작
- Tensorflow, pytorch 의 Tensorboard 에 해당하는 visualizer 제작
- Image 및 String 을 손쉽게 다룰 수 있는 툴 제작