|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство образования и науки Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ: Информатика и системы управления (ИУ)

КАФЕДРА: Системы обработки информации и управления (ИУ5)

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К МЕТОДОЛОГИИ МАШИННОГО ОБУЧЕНИЯ***

***НА ТЕМУ:***

**Циклические инвариантные алгоритмы внешнего подъема для операторов**

Студент \_\_\_ИУ5И-31М\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** \_\_**\_\_**Ян Тяньци\_**\_\_\_\_\_**\_

(Группа) (Подпись, дата) (И.О.Фамилия)

Преподаватель **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**  **\_\_\_\_**Ю.Е. Гапанюк**\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Консультант **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

*2022 г.*

**СОДЕРЖАНИЕ**

[**Введение** 2](#_Toc123075242)

[**1. Проектирование и внедрение системы** 3](#_Toc123075243)

[1.1 Архитектура системы 3](#_Toc123075244)

[1.2 Алгоритм внешнего подъема петлеинвариантного кода 4](#_Toc123075245)

[1.3 Нормализация 6](#_Toc123075246)

[1.4 Функция затрат 8](#_Toc123075247)

[**2.Экспериментальные результаты и анализ** 9](#_Toc123075248)

[2.1 Вопросы исследования 9](#_Toc123075249)

[2.2Набор данных 9](#_Toc123075250)

[2.3 Экспериментальные результаты и анализ 11](#_Toc123075251)

[**3.Заключение** 15](#_Toc123075252)

[**Список литературы** 16](#_Toc123075253)

**Введение**

TVM [1] - компилятор глубокого обучения с открытым исходным кодом [2-5], который генерирует многоаппаратный внутренний код, обеспечивая оптимизацию на уровне графов [6] и оптимизацию на уровне операторов. Однако традиционные алгоритмы инвариантного внешнего подъема цикла не могут быть напрямую использованы для оптимизации компиляторов глубокого обучения, поэтому в данной работе предлагается эвристическая стратегия для алгоритмов инвариантного внешнего подъема цикла для расширенных промежуточных представлений компиляторов глубокого обучения.

Во-первых, мы разработали функцию для измерения вычислительных затрат выражений, чтобы определить, следует ли вычислять инварианты цикла, опираясь на модель затрат оптимизации вычисления инвариантов цикла компилятора Halide, и компромисс между устранением избыточных вычислений и распределением регистров простым и легко реализуемым способом. Во-вторых, алгоритм корректирует порядок комбинаций операндов и преобразует форму условных выражений для нормализации выражений перед обнаружением инвариантов цикла, обеспечивая лучшее время для оптимизации инвариантов цикла и других последующих оптимизаций. Наконец, в данной работе объединены особенности TVM IR и компилятора целевой платформы для обеспечения специальной обработки внутренних циклов с относительно небольшим количеством ветвлений и итераций, разрешая оптимизационные конфликты на разных уровнях промежуточного представления.

**1. Проектирование и внедрение системы**

## 1.1 Архитектура системы

Система, разработанная и реализованная в данной работе, основана на версии 0.7 компилятора глубокого обучения TVM с открытым исходным кодом. Оптимизацию TVM [1] можно разделить на два отдельных модуля, оптимизацию на уровне графа и оптимизацию на уровне оператора, которые можно разделить на автоматическую оптимизацию расписания на основе машинного обучения на уровне тензорных выражений и оптимизацию на уровне TVM IR. Настоящий алгоритм оптимизирует TVM IR, генерируемый оператором, и включает в себя модификации модуля оптимизации на уровне оператора. Общая схема архитектуры и ключевой поток компиляции алгоритма циклического инвариантного внешнего подъема в TVM показаны на рисунке 1.

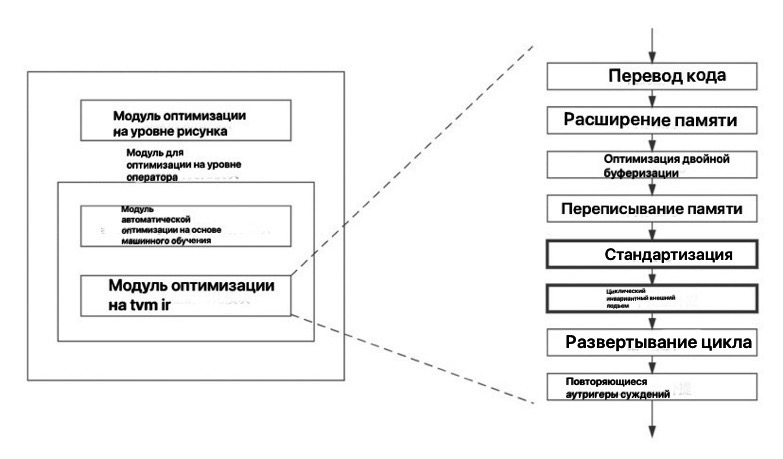


Fig.1 The architecture of Loop Invariant Code Motion algorithm in TVM

Первая - это предварительная часть нормализации, которая нормализует выражения так, чтобы измененные выражения были более пригодны для идентификации циклических инвариантов; вторая - это часть исключения инвариантов, которая идентифицирует и исключает выражения и утверждения, которые остаются инвариантными в цикле. TVM организует серию оптимизационных обходов на Мы регистрируем оптимизационные итерации этого алгоритма в менеджере оптимизационных итераций TVM и располагаем дополнительные итерации в соответствующих местах в последовательности собственных оптимизационных итераций TVM (как показано жирным шрифтом на рисунке 1). Собственные итерации оптимизации TVM также используются для снятия операторов ветвления с инвариантами цикла, а собственные итерации оптимизации упрощения TVM модифицированы для отключения прямой подстановки переменных присваивания целочисленных типов.

## 1.2 Алгоритм внешнего подъема петлеинвариантного кода

Типичными примерами инвариантов цикла в компиляторах глубокого обучения являются вычисление адресных выражений, обращающихся к элементам массива, и вычисление условных выражений, содержащих переменные итерации цикла, параметры программы и непосредственные числа.

Определение инвариантных выражений в цикле требует только выделения, тех выражений, для которых результат вычисления может измениться в течение нескольких итераций цикла. Для определения и подъема инварианта петли можно выполнить следующие шаги расчета.

1) пометьте переменную итерации цикла как "не поднимаемую".

2) пометить все переменные с фиксированным значением, достигаемые переменной итерации цикла, как

"неподъемный".

3) отметьте все выражения, которые не помечены как "небустируемые", и замените их новой переменной.

4) фальсифицировать присвоение новой переменной в 3) перед циклом.

Кроме того, порядок операндов в выражении имеет решающее значение для идентификации инвариантов цикла в выражении; поэтому перед поиском инвариантов цикла необходимо нормализовать выражение, используя операторный закон объединения для выполнения таких преобразований, как рекомбинация выражений.

Метод 1 Алгоритм циклического инвариантного внешнего подъема

Исходная процедура T

Выход: оптимизированная программа

|  |  |
| --- | --- |
| 1: | **procedure** LICM(*T* ) |
| 2: | *R ← ϕ* |
| 3: | *T ←* Normalize(*T* ) |
| 4: | **for** each statement *S* ∈ *T* **do** |
| 5: | **if** *S* : *x* = *e* ∧ *Promote*[*e*] = ⊥**then** |
| 6: | *Promote*[*x*] *←* ⊥ |
| 7: | **if** *S* : *loop*(*x, e*) **then** |
| 8: | *Promote*[*x*] *←* ⊥ |
| 9: | **for** sub-expression *e* ∈ *S* **do** |
| 10: | **if** *Promote*[*e*] ≠ ⊥**then** |
| 11: | *rewrite*(*z, e, S*) |
| 12: | *R*∪= *{z ← e}* |
| 13: | **for** each invariant *I* : *z ← e* ∈ *R* **do** |
| 14: | **if** *cost*(*e*) *≥ K* ∧ *e* no side effects **then** |
| 15: | lift invariant *z ← e* |
| 16: | **else** |
| 17: | abort lifting *I* |
| 18: | **return** *T* |

Этот алгоритм использует преимущества ключевых особенностей компиляторов глубокого обучения для реализации инвариантного к циклу внешнего подъема в их высокоуровневых промежуточных представлениях, компенсируя недостатки традиционных алгоритмов инвариантного к циклу внешнего подъема. Кроме того, алгоритм отличается от аналогичных алгоритмов с круговыми инвариантами тем, что он использует новый подход к предварительной обработке для преобразования выражений, а не просто рекомбинирует аддитивные и субтрактивные выражения, как это делает Halide; это позволяет алгоритму более полно устранять избыточность, избегая дополнительной работы по устранению общих подвыражений из внешних инвариантов и делая компилятор целевой платформы более Простота.

## 1.3 Нормализация

Нормализация - это сохраняющее семантику преобразование утверждений и выражений с целью сделать преобразованную программу более пригодной для последующих инвариантных упоминаний цикла вне инварианта или для оптимизации компилятором целевой платформы. Корректировка порядка следования операндов выражения может привести к значительным улучшениям в оптимизации инвариантов цикла. Например, предположим, что из двух приведенных ниже выражений только i является переменной итерации цикла, а j - переменной итерации внешнего цикла.



Fig.2 The ast of exprerssion（1）Рисунок 2 Абстрактное синтаксическое дерево для выражения (1)

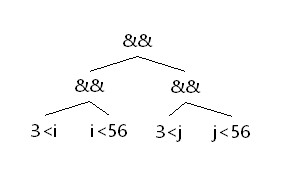


Fig.3 The ast of exprerssion（2）Рисунок 3 Абстрактное синтаксическое дерево для выражения (2)

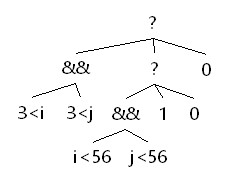


Fig.4 The ast of exprerssion（3）Рисунок 4 Абстрактное синтаксическое дерево для выражения (3)

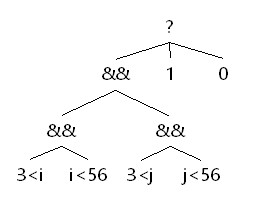


Fig.5 The ast of exprerssion（4）Рисунок 5 Абстрактное синтаксическое дерево для выражения (1)

Идея изменения порядка объединения операндов выражения заключается в следующем: для операции, удовлетворяющей закону объединения, операнды объединяются в возрастающем или убывающем порядке в соответствии с глубиной вложенности итерационных переменных цикла, соответствующих входящему в них циклу; такая операция может объединять разрозненные инварианты цикла вместе и облегчать нахождение инвариантных выражений цикла. Исходя из вышеперечисленных особенностей, циклические инвариантные выражения могут быть объединены в следующие шаги.

1) Привяжите целочисленный ранг к каждому выражению e.

2) если выражение e является константой, привяжите для него ранг к 0.

3) привязать ранг к глубине вложенного цикла, если выражение e является итерационной переменной цикла, иначе

a) связать максимальное значение ранга операнда, определяющего переменную.

b) Для операций, удовлетворяющих закону связывания, каждое выражение сортируется по возрастанию ранга, а затем рекомбинируется в новое выражение.

Метод 2 Алгоритмы нормализации экспрессии

Исходная процедура T

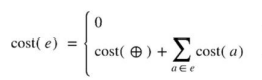
Выход: оптимизированная программа

|  |  |
| --- | --- |
| 1: | **procedure** NORMALIZE(*T* ) |
| 2: | **for** each expression *e* ∈ *T* **do** |
| 3: | **if** *e*: if(*e*1, *e*2, *e*3) **then** |
| 4: | *e ←* condCollapse(*e*) |
| 5: | **for** each var *x* ∈ *e* **do** |
| 6: | *rank*(*x*) *←* loopDepth(*x*) |
| 7: | **for** sub-expression *a* ∈ *e* **do** |
| 8: | *rank*(*a*) *←* max {rank(*x*) |(*x* ∈ *a*)} |
| 9: | **for** each commutative operator **do** |
| 10: | *L ←* deconstruct(*e*, ) |
| 11: | *L ←* stableSort(*L*) |
| 12: | *L ←* groupExpressions(*L*, ) |
| 13: | *E ←* construct(*L*, ) |
| 14: | *T ←* ifconcat(*T*) |
| 15: | **return** *T* |

Алгоритмы нормализации могут в большей степени фокусироваться на конкретных преобразованиях подскриптов и условных выражений, чтобы облегчить реализацию инвариантной к циклам оптимизации выбросов и оптимизации потока управления.

## 1.4 Функция затрат

Суррогатная функция cost(e) принимает на вход выражение e, вычисляет и возвращает суррогатное значение e для аут-рейза. Компилятор считает, что выигрыш достаточен, и выводит выражение e тогда и только тогда, когда возвращаемое значение функции больше или равно заданному пороговому значению K. Для того чтобы взвесить конфликт между устранением избыточных вычислений и оптимизацией распределения регистров, а также избежать взаимного влияния двух различных уровней оптимизации, была разработана функция стоимости cost(e) для расчета стоимости выражения и направления алгоритма на вычисление циклического инварианта. Значение подстановки выражения представляет собой накладные расходы, необходимые для вычисления выражения, т.е. выгоду от внешнего поднятия выражения, так что алгоритм будет поднимать выражение только тогда, когда современное значение больше определенного порога K. Функция стоимости cost(e) для выражения e определяется как



Когда e - постоянная или переменная величина, ее стоимость определяется как 0; когда e является составным выражением, его стоимость определяется рекурсивно по его подвыражению a плюс стоимость его оператора ⊕.

Эта базовая функция стоимости cost(e) может быть модифицирована для различных аппаратных платформ.

**2.Экспериментальные результаты и анализ**

## 2.1 Вопросы исследования

Эксперимент был направлен на ответ на следующие два исследовательских вопроса.

1) Производительность. Улучшает ли новый циклический инвариантный алгоритм внешнего подъема, предложенный в этой статье, производительность фактического оператора, генерирующего TVM? Приводит ли это к снижению производительности в некоторых конкретных случаях и для каких операторов? Если да, то каковы причины деградации?

2) Корректность. Оказывает ли изменение порядка следования операндов в данной работе какое-либо влияние на корректность (т.е. вычислительную точность) оптимизированных генерируемых операторов?

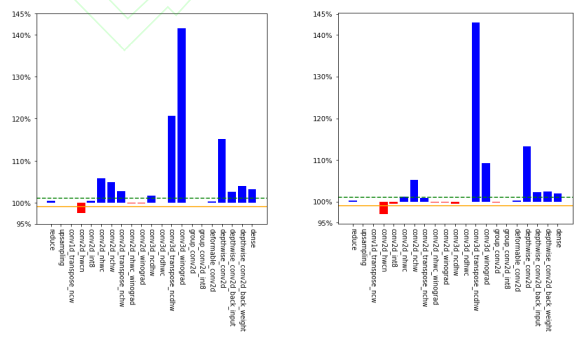
В экспериментах использовалась версия 10.0 компилятора NVCC от NVIDIA в качестве компилятора для платформы GPU.

## 2.2Набор данных

Реализации отдельных операторов из библиотеки операторов TOPI компании TVM при различных масштабах ввода были выбраны в качестве эталонного тестового набора для этого эксперимента, который включал 27 операторов и 511 тестовых случаев. В таблице 2 перечислены все операторы и их количество случаев, выбранных для данной реализации. Эти операторы широко используются во фреймворках глубокого обучения и ТВМ, и в экспериментах приводится большое количество примеров использования каждого оператора при различных масштабах входного сигнала в обычных нейронных сетях, и экспериментальные результаты на этих примерах являются в некоторой степени репрезентативными.

Table 2 Benchmark

|  |  |  |
| --- | --- | --- |
| Имя оператора | Количество столбцов | Количество допустимых столбцов |
| 1 batch\_matmul | 6 | 0 |
| 2 batch\_to\_space\_nd | 4 | 0 |
| 3 clip | 3 | 0 |
| 4 reduce | 16 | 4 |
| 5 relu | 3 | 0 |
| 6 upsampling | 30 | 3 |
| 7 conv1d | 22 | 0 |
| 8 conv1d\_transpose\_ncw | 18 | 5 |
| 9 conv2d\_hwcn | 9 | 9 |
| 10 conv2d\_int8 | 78 | 46 |
| 11 conv2d\_nhwc3 | 13 | 13 |
| 12 conv2d\_nchw | 86 | 68 |
| 13 conv2d\_transpose\_nchw | 13 | 6 |
| 14 conv2d\_nhwc\_winograd | 12 | 10 |
| 15 conv2d\_winograd | 27 | 25 |
| 16 conv3d\_ncdhw | 18 | 15 |
| 17 conv3d\_ndhwc | 11 | 4 |
| 18 conv3d\_transpose\_ncdhw | 12 | 3 |
| 19 conv3d\_winograd | 17 | 17 |
| 20 group\_conv2d | 13 | 10 |
| 21 group\_conv2d\_int8 | 13 | 10 |
| 22 deformable\_conv2d | 3 | 2 |
| 23 depthwise\_conv2d | 14 | 4 |
| 24 depthwise\_conv2d\_back\_ input | 16 | 5 |
| 25 depthwise\_conv2d\_back\_ weight | 16 | 16 |
| 26 dense | 12 | 6 |
| 27 image | 25 | 0 |



(a) Arithmetic average speedups (b) Total time speedups

Fig.6 Operator speedups

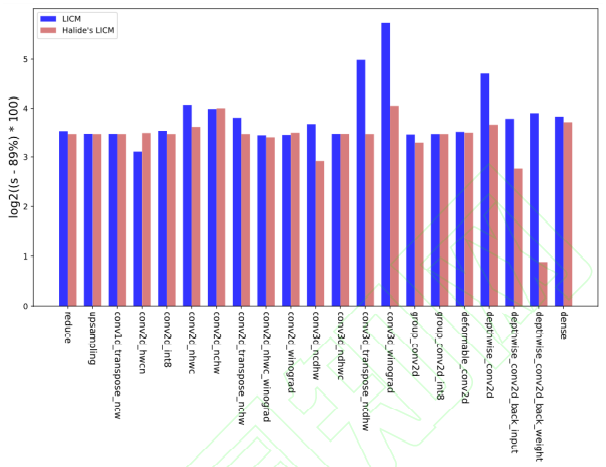
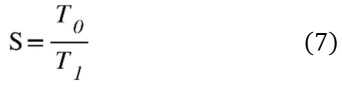


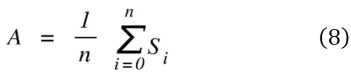
Fig.7 The comparison between our algorithm and Halide's LICM

## 2.3 Экспериментальные результаты и анализ

Чтобы уменьшить экспериментальную ошибку при исследовании влияния алгоритма на производительность, мы считаем те случаи, которые генерируют один и тот же код до и после оптимизации и имеют время работы менее 50 микросекунд, как недопустимые случаи, и не рассматриваем случаи с колебаниями времени работы менее 1%. Количество допустимых случаев для каждого оператора приведено в таблице 2, и мы называем оператор допустимым, если количество допустимых случаев больше 0. Мы определяем коэффициент выбега оператора S как



где T0 - время работы исходного оператора, сгенерированного TVM, а T1 - время работы оператора, сгенерированного прототипной системой в данном эксперименте (т.е. время работы оператора после циклической инвариантной оптимизации outpropagation). Мы определяем средний коэффициент ускорения оператора A как среднее арифметическое коэффициентов ускорения всех измеренных случаев.



Средний коэффициент ускорения эффективного оператора приведен на рисунке 6(a), а коэффициент ускорения времени выполнения эффективного оператора - на рисунке 6(b). Зеленая пунктирная линия на рисунке представляет коэффициент ускорения 101%, а сплошная оранжевая линия - коэффициент ускорения 99%. С точки зрения среднего коэффициента ускорения операторов, 10 операторов, составленных алгоритмом в данной работе, имеют улучшенную производительность по сравнению с операторами, составленными родным TVM, что составляет 47,6% действительных операторов, а оператором с наиболее значительным улучшением производительности является conv3d\_winograd с коэффициентом ускорения 141,46%. С точки зрения общего коэффициента ускорения времени операторов, по сравнению с операторами, составленными с помощью родного TVM, улучшена производительность 10 операторов, составленных с помощью данного алгоритма, что составляет 47,6% от действительных операторов, а общий коэффициент ускорения времени всех операторов составляет 122,7%.

Для того чтобы дополнительно продемонстрировать эффективность улучшения алгоритма, для сравнения с алгоритмом данной работы была выбрана инвариантная к циклу оптимизация внешнего подъема компилятора Halide, а итерации инвариантной к циклу оптимизации внешнего подъема компилятора Halide были адаптированы и перенесены на TVM. В дополнение к встроенным итерациям оптимизации была добавлена поддержка внешнего подъема условных выражений и других операций, соответствующих закону объединения операторов. Фактические результаты показаны на рисунке 7. Минимальный коэффициент ускорения в этом эксперименте составляет 90%, поэтому коэффициент ускорения вычитается из 89% и записывается в 2. Вышеописанная операция выполняется для визуального сравнения двух алгоритмов, и каждая вертикальная координата вычисляется как



где s - коэффициент ускорения оператора. Красные столбики указывают на оптимизирующий эффект алгоритма инвариантного внешнего подъема цикла компилятора Halide. Действительные измерения оператора для этого эксперимента игнорируются при колебаниях производительности менее 1%. Результаты показывают, что данный алгоритм превосходит другой сравнительный алгоритм на 5 операторов, что составляет 23,81% от всех проверенных операторов. Остальные 71,42% протестированных операторов показали незначительную разницу в улучшении производительности. Общее влияние алгоритма на производительность арифметики явно обусловлено алгоритмом сравнения, что еще раз демонстрирует эффективность предложенного в данной работе улучшения инвариантного к циклу алгоритма внешнего подъема.

Вторая проблема, изучаемая в данном эксперименте, - это точность оператора. Точность оператора в основном измеряется точностью результатов вычислений оператора. Основными факторами, влияющими на точность вычисления оператора, являются точность самого аппаратного вычислительного блока и порядок вычисления данных. Точность аппаратного вычислительного блока тесно связана с экспериментальной аппаратной платформой, и для данного эксперимента был выбран графический процессор NVidia Tesla P4. Порядок вычисления данных в основном зависит от внешнего планирования, кроме того, оптимизационный обход также изменит порядок вычислений для преобразования промежуточного представления, оптимизационный обход, реализованный в данной работе, изменит порядок вычислений только для операций, удовлетворяющих закону сочетания операций, и не изменит результат операции. Чтобы исследовать корректность алгоритма, мы сравниваем результаты всех случаев на GPU с результатами эквивалентной реализации Python на cpu, и устанавливаем допуск на ошибку в 1e-5 из-за ограничения равного тестирования чисел с плавающей точкой. -5, что подтверждает правильность алгоритма в данной работе.

**3.Заключение**

В этой статье мы предлагаем новый циклический инвариантный внеквантовый алгоритм оптимизации подъема для компиляторов глубокого обучения и реализуем прототип системы на базе версии 0.7 TVM с открытым исходным кодом. Эксперименты проведены на 27 операторах и 511 тестовых случаях из библиотеки операторов TOPI в TVM, и результаты показывают, что новый алгоритм оптимизации может эффективно улучшить производительность операторов, обеспечивая при этом корректность. Эта работа представляет собой полезную ссылку для TVM или других компиляторов глубокого обучения для переноса традиционного циклического инвариантного алгоритма внешнего подъема.

**Список литературы**

[1] Chen T, Moreau T, Jiang Z, et al. {TVM}: An automated end-to-end optimizing compiler for deep learning[C]//13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), Carlsbad, 2018. Berkeley: {USENIX} Association, 2018: 578–594.

[2] Wei R, Schwartz L, Adve V. DLVM: A modern compiler infrastructure for deep learning systems[J]. arXiv preprint arXiv:1711.03016, 2017.

[3] Vasilache N, Zinenko O, Theodoridis T, et al. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions[J]. arXiv preprint arXiv:1802.04730, 2018.

[4] Rotem N, Fix J, Abdulrasool S, Catron G, et al. Glow: Graph lowering compiler techniques for neural networks[J]. arXiv preprint arXiv:1805.00907, 2018.

[5] Cyphers S, Bansal A K, Bhiwandiwalla A, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning[J]. arXiv preprint arXiv: 梁佳利 等：面向深度学习算子的循环不变式外提算法 15 1801.08058, 2018.

[6] Roesch J, Lyubomirsky S, Weber L, et al. Relay: A new ir for machine learning frameworks[C]// Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, Philadelphia, 2018. New York: ACM, 2018: 58–68.