面向深度学习编译器的循环不变式外提算法

A Loop Invariant Code Motion Technique For Deep Learning Compiler

梁佳利　　华保健 *
中国科学技术大学软件学院
liangjl@mail.ustc.edu.cn　　bjhua@ustc.edu.cn*

摘要— TVM 是一个深度学习编译器，支持将 TVM 的领域专用语言即张量表达式定义的算子编译生成目标平台的代码，并在高级中间表示 TVM IR 上中进行一系列优化。张量表达式对算子执行循环变换，产生与循环迭代变量相关的复杂表达式的计算，在多层嵌套循环内这些计算包含了大量的循环不变式。然而，传统的循环不变量外提技术不能判断不变量外提是否能带来额外收益，无法发现操作数顺序不同的循环不变表达式，不能处理嵌套的条件表达式，并且与目标平台编译器优化存在冲突等。由于这些挑战，传统的循环不变量外提算法无法直接用于深度学习编译器的优化。

本文提出了一种面向深度学习编译器，采用启发式策略的循环不变量外提算法。该算法基于深度学习编译器的高层中间表示，通过调整操作数顺序和简化嵌套条件表达式等方法规范化表达式。为了衡量优化的收益，在结合 TVM IR 和目标平台的特点的基础上，提出了一个新的不变式外提代价指标函数。我们在开源编译器 TVM 0.7 版本上，通过新增优化遍的形式，具体实现了本文所介绍的算法以及代价函数。为评测算法的有效性，并在 Tesla P4 的 GPU 平台上对 TVM topi 的测试算子集中 27 个典型算子不同输入规模的 511 个测例进行了测试。实验结果表明 47.6% 的算子性能得到提升，最大加速比大于 40.0%。

关键词—深度学习编译器, 领域专用语言, 循环不变量外提
中图法分类号—TP311

***Abstract*—TVM is a deep learning compiler, that translates the deep learning operators described by Tensor Expression to TVM IR programs. After a series of operator-level optimizations on TVM IR, TVM generates the target code across diverse hardware back-ends. Tensor Expression, a domain-specific language for tensor computation, performs loop transformation to operators. The result of loop transformation is a number of complicated expressions emerging in nested loop statements, which contain loop invariant code. However, in the context of deep learning applications, the traditional loop invariant code motion algorithm has severe limitations, first, it's difficult to determine the extra benefit of moving certain invariant code out of loops; second, it's difficult to detect loop invariant code which has the different order of operands; third, it cannot process nested condition expressions; furthermore, there are conflicts with target hardware compiler optimizations. The application of loop invariant code motion technique is constrained by the aforementioned problems. In this paper, we propose a new loop invariant code motion algorithm, which takes deep learning application characteristics into consideration in a heuristics way. Our algorithm normalizes the program by manipulating the expression operands and simplifying the nested condition expression. This paper introduces a new cost model, which evaluates the cost of moving certain loop invariant code while the characteristics of TVM IR and target hardware back-ends are fully considered. The algorithm is implemented as a registered TVM pass on open-source compiler TVM version 0.7. To testify the effectiveness and correctness of this algorithm, we conducted experiments on TVM TOPI benchmark with 27 operators and 511 test cases under different input. The experimental results show that this algorithm improves 47.6% of operators' performance, and achieves speedups up to 40.0%.**

***Key words*—Deep learning compilers, domain-specific languages, loop invariant code motion**

通讯作者：华保健（bjhua@ustc.edu.cn）

I. 引言

TVM [1] 是一个开源深度学习编译器 [3]–[7]，通过提供图级别的优化 [2] 和算子级别的优化，它能够为不同的硬件后端生成高效的代码。在实现一个算子时，多种计算方法都能够得到相同的计算结果，但是不同的计

算方法却导致了生成代码的局部性、并行性的差异 [9], [10]。TVM 的领域专用语言—张量表达式支持用简洁的方式定义算子，并提供一系列调度原语对算子代码做循环变换。用张量表达式描述的算子会被 TVM 翻译为 TVM IR–一种树状的、易于描述循环计算的高级中间表示。TVM 在 TVM IR 上实现了一系列分析和变换的优化遍，来对算子的中间表示进行调整和优化，最终生成目标硬件平台上的高效代码。TVM 的模块化设计允许开发者通过新增优化遍的形式，来增加针对目标硬件平台的优化。

深度学习应用 [11]–[14] 中的重要算子，如卷积、矩阵乘和向量加等，都会被 TVM 编译生成 TVM IR 上的多层嵌套循环 [15]，这类程序经过循环变换后会产生与循环迭代变量相关的复杂表达式计算。例如，以下的代码示例是向量加算子的 TVM IR 表示：

```
for(i.outer: int32, 0, 4){
  for(i.inner: int32, 0, 32) {
    C[(i.outer * 32) + i.inner] =
      A[(i.outer * 32) + i.inner] +
      B[(i.outer * 32) + i.inner] } }
```

注意到在内层循环中存在被重复计算的表达式 `i.outer * 32`，它应该被循环不变量外提优化 [23] 提取到第一层和第二层循环之间，从而消除冗余计算，提升算子性能。

然而，在深度学习编译器中实现循环不变式外提优化，存在以下研究挑战：首先，传统编译器中的循环不变量外提优化算法，将计算结果保存在物理寄存器中，避免每次使用时都重复计算 [22]。但是 TVM 等深度学习编译器的高级中间表示无法感知物理寄存器的存在，盲目地外提所有循环不变量会带来不必要的寄存器压力，甚至降低算子的性能 [28]；因此直接将传统编译器中的循环不变式外提算法应用到 TVM 等深度学习编译器中，并不能得到理想的优化效果。

其次，优化算法在对循环不变式的检测过程中，表达式中操作数的顺序会影响不变式的发现 [31]；传统的循环不变式外提算法对深度学习算子中经常出现的复杂嵌套条件表达式的检测能力有限，无法检测合并变换生成的新的循环不变式。

最后，TVM 等深度学习编译器生成的算子代码，还需要经过目标平台编译器的编译优化过程，才能生成目标硬件的可执行代码。目标平台编译器在一个更低级的中间表示上，进行了其他的一些优化 [35]，而在两个不同级别上的优化常常存在冲突 [29]，导致降低了循环不变式外提算法的有效性。Halide 开源编译器 [8] 中实现了一个保守的循环不变式外提算法，但该算法只考虑循环内的表达式，而不对循环内的语句做处理；且该算法在做循环不变式外提之前只对包含加减法的表达式做了重结合，没有对条件表达式做优化；因此，这些简单的处理并不能完全解决上述研究挑战。

在本文中，我们提出了一种面向深度学习编译器高级中间表示的、采用启发式策略的循环不变式外提算法。首先，针对第一个研究挑战，本文借鉴 Halide 的代价模型，设计了代价函数来计算不变式的代价，只外提被认为是有收益的循环不变量，从而以一种简单且易于实现的方式解决了消除冗余计算与寄存器分配之间的矛盾。其次，针对第二个研究挑战，本文算法在检测循环不变量之前，先调整操作数结合顺序和变换条件表达式的形式来对表达式做规范化，为循环不变量外提优化以及后续的其他优化提供更好的时机。最后，针对第三个研究挑战，本文结合 TVM IR 和目标平台编译器的具体特点，对分支和迭代数目比较小的内层循环做了特殊处理，解决了在不同级别的中间表示上的优化冲突。

为验证本文所提算法的有效性，我们在开源编译器 TVM 的 0.7 版本上实现了本文所介绍的算法，并对算法进行了实验和实验结果分析。我们选择 TVM 的 topi 测试集作为算法的测试集合，对该测试集中的 27 个算子对应的 511 个测例进行了测试，这些算子都是 TVM 框架频繁用到的算子，其中包含了多种实现策略的卷积和激活等算子，我们的测例的生成程序都是典型的多层嵌套循环的数组计算，同时一些算子内的补零和边界检查操作在循环内产生了条件表达式和分支。这些算子测例描绘了常见深度学习领域算子特点，并且每一个算子都提供了大量在常见神经网络中不同参数规模的测例，因此具有一定的代表性。实现结果发现，在排除那些优化前后代码没有发生变化的测例后，与未进行循环不变量外提优化的测例相比，有 47.6% 的算子性能得到了显著提升，最大加速比大于 40.0%；所有测例的总时间加速比平均为 22.7%。实验结果证明，本算法对 TVM 生成算子的性能产生了积极影响，加速了含有大量冗余计算的算子的执行，完善了 TVM 的算子优化。

总结起来，本文的主要贡献如下：

1) 本文提出了一个新的适用于深度学习编译器优化的循环不变量外提算法，该算法包括一个新的代码外提代价模型；
2) 本文基于 TVM 开源深度学习编译器（0.7 版本），给出了原型系统设计与实现；
3) 本文选取深度学习领域典型的 27 个算子及其 510 个测例，对原型系统进行了系统测试；实验结果分析表明，该算法对提升 TVM 的性能产生了积极作用。

本文余下内容组织如下：第II小节介绍工作背景与研究动机；第III节讨论循环不变量外提算法的设计、及其在 TVM 深度学习编译器上的实现；第IV 小节给出实验结果并对实验结果进行分析；第V小节介绍相关工作；第VI小节总结全文。

II. 背景与研究动机

本小节介绍关于 TVM 中间表示的相关背景，以给出本文工作的研究动机。

A. 张量表达式和 TVM IR

张量表达式是 TVM 基于 Halide 的算子描述和调度优化分离的思想 [9], [10]，而设计实现的描述张量运算的领域专用语言。该语言定义张量间的输入输出关系，同时提供循环拆分、循环合并和循环分块等一系列调度原语，这些调度原语的组合指定了计算输出张量的具体策略 [16]。

张量表达式在编译过程中被 TVM 翻译到由 TVM IR 表示的模板计算，并在此基础上进行优化。TVM IR 是一种树状的中间表示，TVM IR 中主要有两种类型的节点：1）表达式节点；2）语句节点。表达式节点主要包含：有名字的变量和常数、数组数据的加载、函数调用、算术逻辑表达式、条件表达式等；语句节点包含顺序、循环、分支、赋值等。

下面实例定义了一个描述固定规模矩阵乘法算子的张量表达式。

```
M = 1024; K = 1024; N = 1024; bn = 32
A = te.placeholder((M, K), name="A")
B = te.placeholder((K, N), name="B")
k = te.reduce_axis((0, K), "k")
C = te.compute((M, N),
    lambda x, y: te.sum(A[x, k] * B[k, y], axis=k),
        name="C")
s = te.create_schedule(C.op)
```

变量 A、B 被定义为规模为 $(1024, 1024)$ 的二维矩阵，而 k 被定义为矩阵相乘时，要进行求和归约的那个循环“轴”。矩阵 C 由矩阵 A 和 B 计算得到，C 通过 λ 表达式，来指定矩阵 C 中每一个元素，如何由矩阵 A 和 B 的对应元素计算得到。

上述算子描述经过 TVM 深度学习编译器编译，得到以下 TVM IR 中间表示代码：

```
for (x, 0, 1024){
  for (y, 0, 1024){
    C[x*1024+y] = 0
    for (k, 0, 1024){
      C[x*1024+y] = C[x*1024+y] +
        A[x*1024 + k] * B[k*1024 + y] } } }
```

为了高效实现上述代码中的最内侧循环，TVM 使用了 tile、split 和 reorder 等操作，对矩阵乘法算子进行循环分块、循环拆分和循环重排序：

```
xo, yo, xi, yi =
    s[C].tile(C.op.axis[0], C.op.axis[1], bn, bn)
(k,) = s[C].op.reduce_axis
ko, ki = s[C].split(k, factor=32)
s[C].reorder(xo, yo, ko, ki, xi, yi)
```

上述变换后的算子描述经过 TVM 翻译得到以下 TVM IR 代码：

```
for (xo, 0, 32) {
  for (yo, 0, 32) {
    for (xi.init, 0, 32) {
      for (yi.init, 0, 32) {
        C[xo*32768+xi.init*1024+yo*32+yi.init] = 0 } }
    for (ko, 0, 32) {
      for (ki, 0, 32) {
        for (xi, 0, 32) {
          for (yi, 0, 32) {
            C[xo*32768+xi*1024+yo*32+yi] =
              C[xo*32768+xi*1024+yo*32+yi] +
              A[xo*32768+xi*1024+ko*32+ki] *
              B[ko*32768+ki*1024+yo*32+yi] } } } } } }
```

B. 研究动机

算子经过循环变换后得到的高阶模板计算有更好的并行性和数据局部性，且能够充分利用底层硬件特

性，加速计算 [15]–[18]。但是算子经过循环变换后，数组下标表达式由原来简单的表达式，变为了语义等价但更复杂的表达式，增加了额外不必要的标量运算。例如，在如上矩阵乘算子经过循环变换后得到的 TVM IR 代码中，对于最内层循环，其循环迭代变量是 `yi`，下标表达式 `xo*32768 + xi*1024 + yo*32` 在每次最内层循环迭代时，计算结果都不变，在第一次迭代后的计算都属于冗余计算。

面向深度学习编译器的循环不变量外提技术的设计目标，是通过把循环中多次计算值保持不变的冗余计算移动到循环外，从而减少额外的冗余运算，生成性能更高的算子。另外，TVM 编译生成的算子会经过目标平台编译器的编译优化，才能在常用深度学习硬件，如英伟达的 GPU 和寒武纪的 MLU 硬件上 [36]，高效运行。这类硬件的典型特征是很多部件都为了运行计算密集的程序而设计，具有很高的计算的并行性。为了充分利用这些硬件特性，目标平台编译器会选择展开内层循环，一些计算的延时通常被指令流水线所掩盖，因此一些冗余计算并不影响运行时间，反而会降低物理寄存器压力。而由于硬件更专注于并行计算，对于分支优化处理的任务需要由编译器优化完成 [32], [33]。因此对布尔表达式、条件分支语句进行外提时需要谨慎处理，有时还需要变换条件表达式的形式，使得生成的代码更利于底层编译器的优化。

综合以上讨论，我们面向深度学习编译器，提出了一种新的循环不变量外提算法，该算法考虑了 TVM IR 和目标平台编译器的特点，通过外提循环不变量减少冗余计算，提高 TVM 生成算子的性能。

III. 系统设计与实现

本小节讨论系统的设计与实现。我们在给出系统架构后（第III-A 小节），重点讨论循环不变式外提算法（第III-B小节），并给出规范化和代价函数（第III-C、III-D 小节）。

A. 系统架构

我们设计和实现的系统基于开源深度学习编译器 TVM 的 0.7 版本。TVM 优化 [1] 可以分为图级别优化和算子级别优化两个独立的模块，算子级别的优化又可分为在张量表达式级别的基于机器学习的自动调度优化，和在 TVM IR 级别的优化。本算法针对算子生成的 TVM IR 进行优化，涉及对算子级别优化模块的修改。循环不变量外提算法在 TVM 中的整体架构图和关键编译流程如图1所示。

从整体架构上看，我们设计的循环不变量外提算法由两个关键部分组成，第一个是前处理部分 Normalize，该部分对表达式进行规范化处理，使得变化后的表达式更利于循环不变量的识别；第二个是不变量外提部分 LICM，对循环保持不变的表达式和语句进行识别和外提。TVM 通过优化遍管理器组织一系列优化遍在 TVM IR 上进行优化，我们向 TVM 的优化遍管理器注册本算法的优化遍，并安排新增的优化遍到原生 TVM 优化遍序列的合适的位置（如图 1中加粗边框的遍所示）。同时我们使用了原生 TVM 的 HoistIfThenElse 的优化遍来对分支条件为循环不变量的分支语句做提升；修改原生 TVM 的 Simplify 优化遍，禁止了原本对整数类型的赋值变量进行前向替换的操作。

B. 循环不变代码外提算法

深度学习编译器中的循环不变量实例中，典型的是访问数组元素的地址表达式的计算和条件表达式的计算，表达式中包含了循环迭代变量、程序参数和立即数。TVM IR 中的循环节点包含了循环嵌套关系以及对应的迭代变量，所以我们递归地判断某个表达式是否是循环不变的，只需要构成表达式的值满足以下判定条件：

1) 是常数（不可变的程序参数或立即数）；
2) 该变量的赋值在循环之外，或者；
3) 到达该变量的使用只有一个定值，且该定值的表达式也是循环不变的。

实际上，识别在一个循环中的不变表达式，只需要标记出在循环的多次迭代中计算结果可能发生变化的表达式，那么剩下未标记的表达式都是该循环的不变式。因此，可按照以下计算步骤识别并提升的循环不变式：

1) 标记循环迭代变量为“不可提升”；
2) 标记循环迭代变量到达的定值变量都是“不可提升”；
3) 记录所有没有被标记为“不可提升”的表达式，并用一个新的变量替换该表达式；
4) 将 3) 中新的变量的赋值语句插入循环之前。

虽然单个常数是循环不变的，但是提升它没有意义，所以算法不对单个常数做提升。

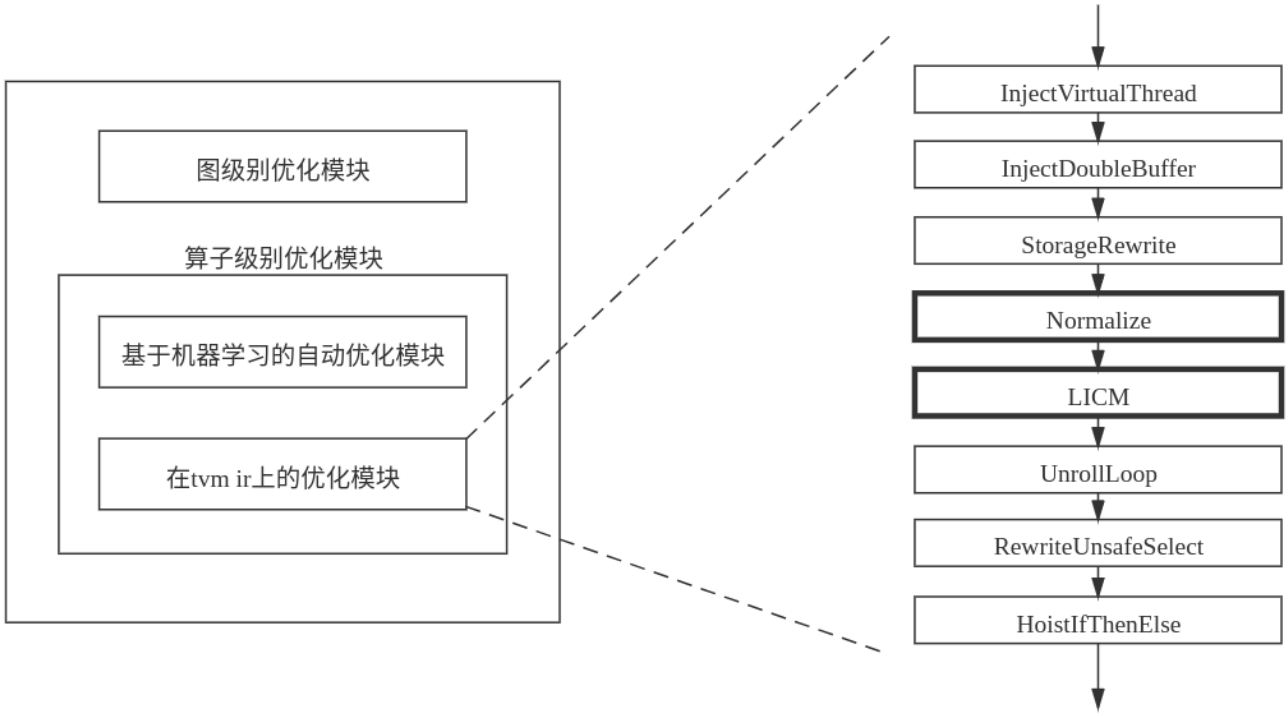


图 1: 循环不变式外提算法在 TVM 中的架构图

另外对于表达式中循环不变式的识别，表达式中操作数的顺序非常关键；不同的操作数顺序不但影响了某些循环不变的表达式能否被发现，还决定了目标编译器中某些对于分支的优化能否被触发。因此，我们在发现循环不变量之前需要先对表达式进行规范化，利用运算符的结合律对表达式进行重结合等 [30] 变换。

算法1 给出了循环不变式外提算法 LICM(T) 的伪代码。该算法接受原始程序 T 做为输入参数，并返回优化后的程序。算法首先调用规范化函数 Normalize(T) 对程序 T 中的语句和表达式做规范化，该规范化函数我们稍后给出。然后算法按照一定的顺序，遍历程序 T 中的语句 S，并按语句 S 的可能语法形式进行分类讨论：若 S 为赋值语句 $x = e$，并且赋值右侧表达式 e 被标记为“不可提升”，那么算法标记被该赋值语句定义的变量 x 为“不可提升”；注意到，算法中使用映射 $Promote[e]$ 来记录给定的表达式 e 是否可提升，特定的常量值 $\perp$ 代表不可提升。若 S 为循环语句 $loop(x, e)$，其中 x 是循环遍历，e 是循环体，则算法标记 S 的循环变量 x 为“不可提升”。对于语句 S 中那些没有被标记为“不可提升”的表达式 e，都可认为是循环不变量，因此，算法生成一个新变量 z，将 S 中的表达式 e 用变量 z 改写，并将识别的循环不变式的候选 $z \leftarrow e$ 添加到循环不变式集合 R 中供后续处理。

算法接着对循环不变式集合 R 中的每个循环不变式 $z \leftarrow e$ 进行分析。算法调用代价函数 $cost(e)$ 计算外提表达式 e 的代价，当且仅当该代价大于等于给定的阈值 K，并且表达式 e 没有副作用时，算法才真正将不变式 $z \leftarrow e$ 外提；否则，算法放弃对不变式 $z \leftarrow e$ 的外提尝试，并恢复被替换的表达式。代价常数 K 的取值与目标平台硬件和编译器密切相关，在通常情况下可取 K 为 1。

传统编译器中的循环不变量外提算法，如 LLVM 中的外提算法 [22]，都是在基于基本块的流图上设计和实现的。这种实现方式主要的缺点是程序在下降的过程中丢失了如循环、作用域以及条件表达式等高层程序信息，所以在执行循环不变量外提算法前这些必要信息都要通过额外的操作来恢复。另外，在编译器低层次的中间表示中，表达式或语句的表示形式发生了改变，这导致原本在高级表示上显而易见的优化时机在后面的优化中很难被发现。而本算法充分利用了深度学习编译

算法 1 循环不变量外提算法

输入： 原始的程序 T

输出： 优化后的程序

procedure LICM(T)
  $R \leftarrow \phi$
  $T \leftarrow$ Normalize(T)
  for each statement $S \in T$ **do**
    if $S : x = e \wedge Promote[e] = \perp$ **then**
      $Promote[x] \leftarrow \perp$
    if $S : loop(x, e)$ **then**
      $Promote[x] \leftarrow \perp$
    for sub-expression $e \in S$ **do**
      if $Promote[e] \neq \perp$ **then**
        $rewrite(z, e, S)$
        $R \cup = \{z \leftarrow e\}$
    for each invariant $I : z \leftarrow e \in R$ **do**
      if $cost(e) \geq K \wedge e$ no side effects **then**
        lift invariant $z \leftarrow e$
      else
        abort lifting I
  return T

器的主要特点，面向其高级中间表示实现循环不变式外提，弥补了传统循环不变量外提算法的不足。另外，该算法和已有的类似循环不变式算法 [25] 也很不相同，如本算法使用的新的前置处理方式对表达式做变换，而不仅仅是像 Halide 那样只对加法减法表达式做重结合；这使得本算法对冗余消除的更为彻底，避免了需要对外提的不变量做公共子表达式消除的额外工作，让目标平台编译器更加简化。

C. 规范化

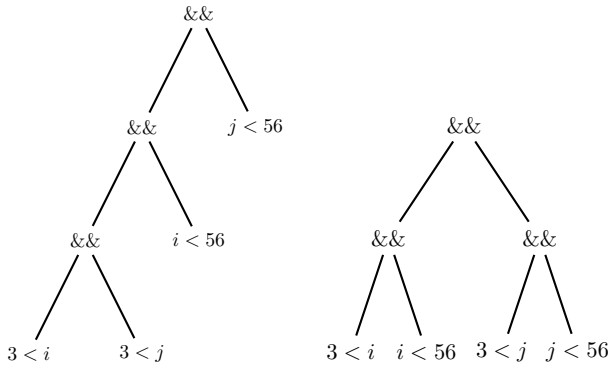
规范化指的是通过对语句和表达式进行保语义的变换，使得变换后的程序更利于后续循环不变量外提或目标平台编译器的优化。规范化最主要的操作包括表达式重结合和对条件表达式和分支的合并操作。

表达式重结合指的是利用特定的代数性质，如结合律、交换律和分配律等，来将表达式划分成常数部分、循环不变部分和变量部分。尽管表达式重结合本身并不能改进性能，但由于某个表达式的子表达式是否为循环不变表达式，很大程度上取决于操作数的运算顺序和结合关系 [31]，因此，表达式重结合能够给循环不变式外提优化带来明显的提升效果。例如，假设下面给定的两个表达式中，只有 i 是循环迭代变量：

$$((((3 < i)\&\&(3 < j))\&\&(i < 56))\&\&(j < 56)) \quad (1)$$

$$(((3 < i)\&\&(i < 56))\&\&((3 < j)\&\&(j < 56))) \quad (2)$$

表达式(1)对应的抽象语法树如图2(a)所示，算法会计算得到两个候选的循环不变式：$(3 < j)$ 和 $(j < 56)$。而表达式(1)经过规范化后得到表达式(2)，其对应的抽象语法树如图2(b)所示，算法得到了更优的候选循环不变表达式 $((3 < j)\&\&(j < 56))$。



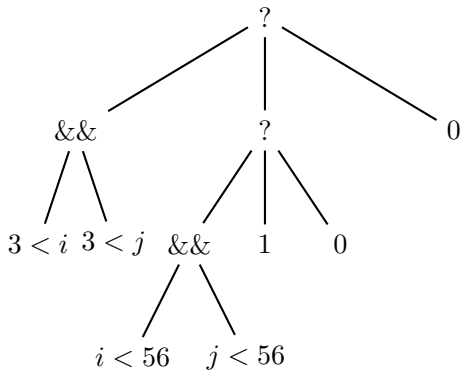
(a) 表达式(1)的抽象语法树　　(b) 表达式(2)的抽象语法树

图 2: 布尔表达式的语法树表示

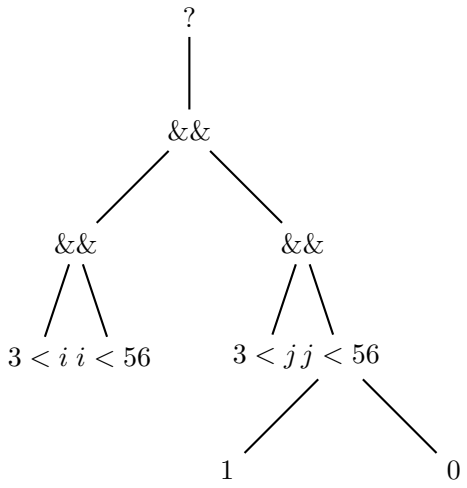
基于表达式的以上特点，可按如下步骤把循环不变的表达式结合到一起：

1) 为每一个表达式 e 绑定一个整数 rank：
   - 若表达式 e 为常数，则为其绑定 rank 为 0；
   - 若表达式 e 为循环迭代变量，则为其绑定 rank 为嵌套循环的深度，否则；
   - 绑定定义该变量的操作数的最大 rank 值。
2) 对于满足结合律的运算，按照 rank 值递增为每个表达式排序后，重新结合为新的表达式。

条件表达式对应判断条件的成立和不成立都有对应的返回值，而在嵌套的条件表达式中，多个判断条件成立与否可能对应于同一个返回值，对此，我们可以合并这些具有相同返回值的判断条件，使得变换后的表达



(a) 表达式(3)的抽象语法树



(b) 表达式(4)的抽象语法树

图 3: 条件表达式的语法树表示

式中的循环不变量更完整。例如，假设下面的表达式中只有 i 是循环迭代变量:

$$((((3 < i)\&\&(3 < j))\ ?((i < 56)\&\&(j < 56))\ ?\ 1 : 0 :\ 0) \tag{3}$$

表达式(3)对应的语法树如图3(a)所示，算法确定的候选循环不变式为 $3 < j$ 和 $j < 56$。表达式(3)经过化简后得到条件表达式:

$$((((3 < i)\&\&(i < 56))\&\&((3 < j)\&\&(j < 56)))\ ?1 : 0) \tag{4}$$

其对应的语法树如图3(b)所示，算法发现了更好的候选循环不变表达式 $((3 < j)\&\&(j < 56))$，并且 $((3 < i)\&\&(i < 56))$ 和 $((3 < j)\&\&(j < 56))$ 这两个表达式的形式更利于实现与布尔表达式相关的窥孔优化。最后，对于相邻的具有相同跳转条件的分支，算法可以合并这些分支语句。

算法 2 表达式规范化算法

输入: 原始的程序 T

输出: 优化后的程序

```
1:  procedure NORMALIZE(T)
2:      for each expression e ∈ T do
3:          if e: if(e1, e2, e3) 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
```

算法2 给出了规范化算法 Normalize(T) 的伪代码实现。该算法接受原始程序 T 作为输入，并输出优化后的程序。算法遍历程序 T 中的表达式 e，并根据表达式 e 的语法形式对其进行处理: 如果表达式 e 为条件表达式 if (e_1, e_2, e_3)，则算法调用函数 condCollapse(e)，对可以进行判断条件合并的表达式 e 进行优化，并返回优化后的结果。接下来，算法按照我们前面讨论的步骤，计算表达式 e 中的所有变量和子表达式的 rank 值。对于完成 rank 值计算的表达式 e，算法遍历其所有满足结合律的操作符 $\oplus$，调用 deconstruct($e, \oplus$) 将由操作符 $\oplus$ 连接的子表达式转换为有序子表达式序列 L；然后，算法调用函数 stableSort(L) 使用稳定排序算法将 L 按照 rank 值从小到大排序并返回排序后的序列 L；最后，算法调用函数 groupExpressions(L,$\oplus$) 将 L 中相同 rank 值的子表达式用操作符 $\oplus$ 连接为新的表达式并返回变换后的序列 L，并调用函数 construct(L,$\oplus$) 用操作符 $\oplus$ 依序将 L 中的所有子表达式连接，并返回变换后的最终结果 e。

算法对程序 T 中的所有表达式遍历结束后，调用函数 ifconcat(T) 合并 T 中相邻的具有相同分支条件的

分支语句，并返回变换后的程序 T。函数 ifconcat(T) 本质上完成控制流优化，通过对分支语句进行合并优化，可有效减少语句和表达式的数量，减少优化算法的执行时间。

传统的标量编译器 [30] 在进行循环不变式外提前，也都会执行代数化简和表达式重结合等优化遍，来完成前置的处理和准备工作。而在深度学习编译器中，和内存读写相关操作的优化应该在更加低级的中间表示上进行，代数化简和常量传播等优化由 TVM 原生的 simplify 优化遍完成，因此，规范化算法可以更专注于对下标和条件表达式进行特定变换，以有利于循环不变量外提优化和控制流优化的实现。

D. 代价函数

代价函数 $cost(e)$ 接受表达式 e 作为输入，计算并返回 e 的外提代价值。当且仅当该函数的返回值大于等于给定的阈值 K 时，编译器才认为有足够的收益，并将该表达式 e 进行外提。需要计算代价函数 $cost(e)$ 的原因是 TVM 生成的高层算子代码，还要经过目标平台编译器的编译，才能生成真正目标硬件上的可执行程序。由于目标平台编译器会在更低层次的中间表示上对程序做一系列优化，要避免两个不同层次的优化相互干扰，因此我们通过代价函数 $cost(e)$ 来计算并选择更有收益的循环不变式进行外提。

对表达式 e 的代价函数 $cost(e)$ 定义为

$$cost(e) = \begin{cases} 0, & e\text{是常数或变量;} \\ cost(\oplus) + \sum_{a \in e} cost(a), & e\text{是复合公式。} \end{cases}$$

当 e 是常数或变量时，其代价定义为 0；当 e 是复合表达式时，其代价递归定义在其子表达式 a 上，另外加上其运算符 $\oplus$ 的代价。下面以一个具体的表达式 `x/1024+y` 为例，详细说明代价值的计算方式。根据定义有 `cost(x/1024+y)=cost(+) + cost(/) + cost(x) + cost(1024) + cost(y)`，不同的运算 $\oplus$ 根据其需要的时钟周期的不同定义了不同的代价，其中其中加法指令的需要的时钟周期较少，我们设置 $cost(+) = 1$，除法指令需要的时钟周期较多，我们设置 $cost(/) = 3$。因此有 `cost(x/1024+y)=1 + 3 + 0 + 0 + 0 = 4`。

这个基础的代价函数 $cost(e)$ 可以修改为适合于不同硬件平台的代价函数，例如，对于 GPU 平台，其平台上的编译器 NVCC 通常选择展开迭代范围很小的循环，并在此基础上执行其他的优化。因此，我们可以设置循环迭代范围很小的循环中的表达式代价值为 0，避免两个不同层次的的优化互相干扰。

IV. 实验与结果分析

本小节详细介绍所进行的实验，并对实验结果进行分析。第IV-A小节给出实验要回答的研究问题；第IV-B给出我们实验平台的各项配置；第 IV-C小节给出实验的数据集；第IV-D给出实验结果，并对不同的试验结果进行分析；第IV-E小节总结本次实验。

A. 研究问题

实验主要回答以下两个研究问题：

1) 性能。本文提出的新的循环不变式外提算法是否能够提升实际 TVM 生成算子的性能？是否以及对哪些算子会造成一些特殊情况性能下降？如果有下降，原因是什么？
2) 正确性。本文算法改变了操作数的顺序，对于优化后生成的算子的正确性（即计算精度）是否会造成影响？

B. 实验平台

实验使用英伟达的 NVCC 编译器的 10.0 版本作为 GPU 平台的编译器。所有实验数据都在表I所示配置的服务器上收集得到。

表 I: 实验环境

	参数配置
CPU	64 个 Intel Xeon® Gold 6130 型号的物理 CPU
GPU	1 个 Tesla P4 型号的物理 GPU
内存	256GB
操作系统	Ubuntu 16.04.6 LTS

C. 数据集

实验选择了 TVM 的 TOPI 算子库中的部分算子在不同输入规模下的实现，作为本次实验的基准测试集，本次测试涉及 27 个算子和 511 个测例。表II列出了本次实现选择的所有算子及其测例数量。这些算子属于深度学习框架和 TVM 常用的算子，并且实验对每一个算子都提供了大量在常见神经网络中不同输入规模的测例，在这些测例上的实验结果具有一定的代表性。

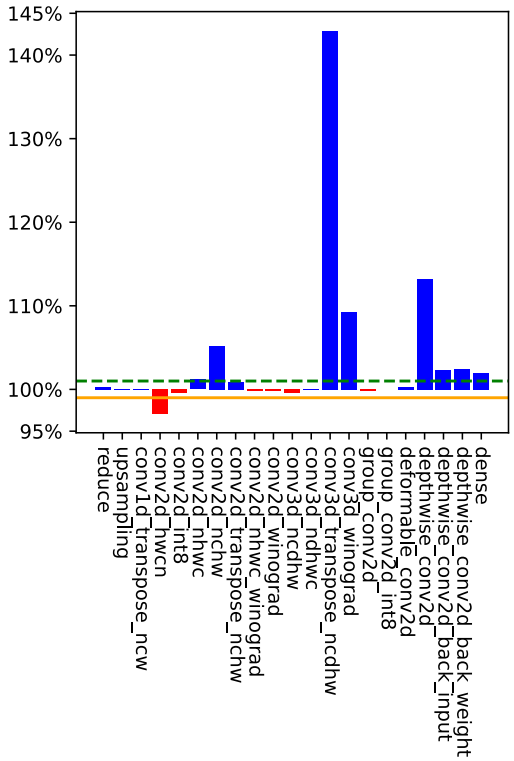


图 5: 算子加速比

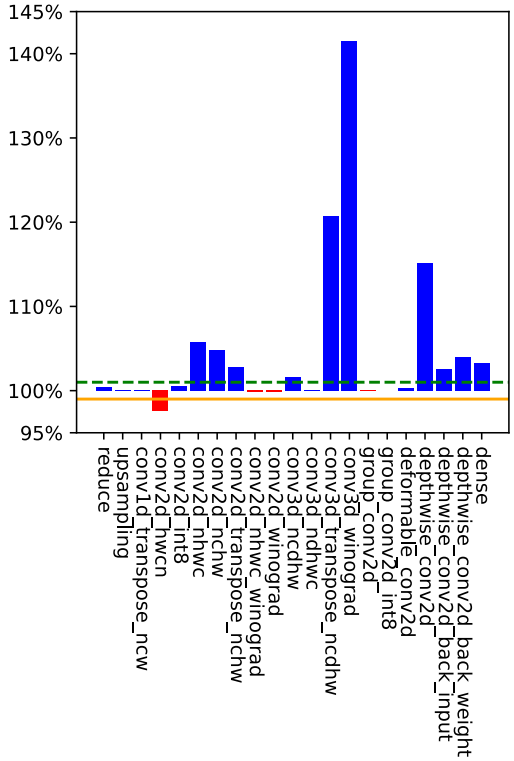


图 6: 算子总时间加速比

D. 实验结果与分析

在研究算法对性能的影响时，为了减少实验误差，我们认为那些优化前后生成代码相同和运行时间小于 50 微秒的测例为无效测例，且不考虑运行时间波动低于 1% 的测例。表 II给出了每个算子的有效测例数量，我们称有效测例数量大于 0 的算子为有效算子。

我们定义算子运行加速比 S 为

$$S = \frac{T_0}{T_1},$$

其中 T_0 为原生 TVM 生成算子的运行时间，T_1 为本次实验的原型系统生成算子的运行时间（即经循环不变式外提优化后算子运行时间）。我们定义算子平均加速比 A 为所有测例的加速比的算术平均值

$$A = \frac{1}{n}\sum_{i=1}^{n} S_i$$

图5给出了有效算子运行时间加速比，图6 给出了有效算子的平均加速比。图中的绿色虚线代表加速比为 101%，橙色实线代表加速比为 99%。从算子平均加速比的角度来看，相比于原生 TVM 编译得到的算子，本文算法编译得到的算子中有 10 个算子性能得到了提升，占有效算子的 47.6%，性能提升最明显的算子为 conv3d_winograd，其加速比达到 141.46%；只有 1 个算子性能下降，占有效算子的 4.7%，该性能下降的算子为 conv2d_hwcn，其加速比为 97.60%；其余算子性能不变。从算子总时间加速比的角度来看，相比于原生 TVM 编译得到的算子，本文算法编译得到的算子中有 10 个算子性能得到了提升，占有效算子的 47.6%，性能提升最明显的算子为 conv3d_transpose_ncdhw，加速比为 142.91%；只有 1 个算子性能下降，占有效算子的 4.7%，该性能下降的算子为 conv2d_hwcn，加速比为 97.10%；其余算子性能不变。所有算子的总时间加速比为 122.7%。

对性能得到提升的测例，我们进一步分析了其优化前后生成的代码，我们发现表达式中的循环不变式被提出了循环外，分支和条件表达式也得到了相应的优化。首先，循环内的指令数目减少了，一部分循环不变的指令被移到了循环外；其次，计算谓词的指令得到了优化，一些有公共操作数的比较指令被减法或其他指令替换；

表 II: 测试算子集

	算子名	测例数量	有效测例数量
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_nhwc	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

再次，分支指令也用谓词执行来替换，减少了分支指令的数量；最后，因为循环内指令减少或者简化，nvcc 编译器对内层循环进行了更大范围的循环展开，同时其他的优化遍对展开后的循环体进行了更好的优化。

对优化后性能下降的一个测例，我们进一步分析发现只有少量的简单表达式被提出了循环外，循环内的控制流结构和指令数基本没有发生变化，这些简单指令的延迟原本就会被流水线隐藏；另外，一些指令和同步指令的相对位置发生了变化，一些原本在同步指令后的计算被移动到了同步指令之前，可能改变后指令调度时的优化效果。需要特别注意的是，尽管这个特定算子的性能出现下降，但其 2.9% 的小幅下降仍然在可接受的范围内。

为研究本文算法的正确性，我们对所有测例在 GPU 上的运行结果与 Python 实现的等价测例在 cpu 上的运行结果进行了比较，由于浮点数的相等测试的约束，本实验设置误差容忍度为 $1e^{-5}$。实验结果表明 100% 的测例实验结果误差小于 $1e^{-5}$，这证实了本文算法的正确性。

E. 结论

本文以 TVM 0.7 版本为基础，实现了添加循环不变式外提优化算法的原型系统，并在 TVM 的 TOPI 算子测试集上针对 27 个算子的 511 个测例进行测试，并收集实验结果。对实验结果的分析表明，本文提出的循环不变式外提算法弥补了闭源编译器 NVCC 在对复杂表达式的冗余删除和控制流优化存在的不足，完善了 TVM 生成算子的优化过程，对算子性能提升产生了积极的作用。

V. 相关工作

作为优化循环内冗余计算的一种有效技术，循环不变式外提已经被广泛研究和应用，很多编译器都使用消除冗余计算的方法优化生成代码性能。Aho 等 [23] 系统讨论了循环不变量外提的概念。TVM [1] 在图级别的中间表示 RELAY 上 [2]，使用公共子表达式消除算法在算子的层级上减少了相同输入规模算子的重复计算；LLVM [21] 在 llvm ir 上使用了循环不变量外提技术 [22]，在基于基本块的控制流图上对循环内不变量指令外提；Steffen 等 [24] 提出了一种代码移动和代数优化的算法；Rosen 等 [20] 使用全局值编号来做冗余消除优化。但是，这些工作的主要不足是这些研究都在底层中间表示、而不是在源代码级别上，对循环内的语句做循环不变式外提优化；因此无法直接应用在深度学习编译器的优化场景中。

不少的研究对于源代码级别的优化做了讨论和实践。Halide 编译器 [8] 在把算子计算描述翻译为 Halide ir 后，针对于循环内的表达式使用了公共子表达式删除和循环不边量外提技术。Rawat 等 [28] 提出了一个寄存器优化框架，在源代码级别对含有多层嵌套循环的高阶模板计算进行了寄存器的优化，提高指令并行性；AKG 框架 [27] 研究了在华为 mindspore 后端的算子自动生成器 akg 上，实现公共子表示消除算法，优化了特定指令的重复计算；Vasilev 等 [26] 提出了一种在递归函数上的循环不变量外提算法；Song 等 [19] 通过展开循环的前几次迭代，来优化循环不变量外提的效果；Bacon 等 [34] 全面总结了在高级语言上的高性能计算的优化方法。但是，考虑到应用于深度学习领域的处理器虽然有很好的指令集并行架构，却不擅长分支的处理，本文

提出的新算法很好地把对分支和条件表达式的优化，整合到了循环不变量外提中。

VI. 结语

本文针对深度学习编译器，提出了一种新的循环不变式量外提优化算法，并基于开源 TVM 的 0.7 版本实现了原型系统。对 TVM 中的 TOPI 算子库中的 27 个算子和 511 个测例进行了实验，实验结果表明该新优化算法在保证正确性的情况下有效提升了算子性能。本文工作为 TVM 或者其他深度学习编译器移植传统循环不变式外提算法提供了有益参考。

参考文献

[1] Chen, Tianqi, et al. "TVM: An automated end-to-end optimizing compiler for deep learning." 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). 2018.

[2] Roesch, Jared, et al. "Relay: A new ir for machine learning frameworks." Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. 2018.

[3] Wei, Richard, Lane Schwartz, and Vikram Adve. "DLVM: A modern compiler infrastructure for deep learning systems." arXiv preprint arXiv:1711.03016 (2017).

[4] C. Leary and T. Wang, "XLA: Tensorflow, compiled,"TensorFlow Dev Summit, 2017.

[5] N. Vasilache et al., "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions,"2018, arXiv: 1802.04730

[6] N. Rotem et al., "Glow: Graph lowering compiler techniques for neural networks,"2018, arXiv: 1805.00907.

[7] S. Cyphers et al., "Intel nGraph: An intermediate representation, compiler, and executor for deep learning,"2018, arXiv: 1801.08058

[8] Ragan-Kelley, Jonathan, et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." Acm Sigplan Notices 48.6 (2013): 519-530.

[9] Ragan-Kelley, Jonathan Millard. Decoupling algorithms from the organization of computation for high performance image processing. Diss. Massachusetts Institute of Technology, 2014.

[10] Ragan-Kelley, Jonathan, et al. "Decoupling algorithms from schedules for easy optimization of image processing pipelines." ACM Transactions on Graphics (TOG) 31.4 (2012): 1-12.

[11] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems 25 (2012): 1097-1105.

[12] Simonyan, Karen, and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556 (2014).

[13] He, Kaiming, et al. "Deep residual learning for image recognition." Proceedings of the IEEE conference on computer vision and pattern recognition. 2016.

[14] Huang, Gao, et al. "Densely connected convolutional networks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[15] K. Datta et al., "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, 2008, pp. 1-12, doi: 10.1109/SC.2008.5222004.

[16] Dursun, Hikmet, et al. "In-Core Optimization of High-Order Stencil Computations." PDPTA. 2009.

[17] Hammouda, Adam, Andrew R. Siegel, and Stephen F. Siegel. "Dynamic Barrier Relaxations for Explicit Stencil Computations." (2014).

[18] Holewinski, Justin, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. "High-performance code generation for stencil computations on GPU architectures." Proceedings of the 26th ACM international conference on Supercomputing. 2012.

[19] Song, Litong, Krishna Kavi, and Ron Cytron. "An unfolding-based loop optimization technique." International Workshop on Software and Compilers for Embedded Systems. Springer, Berlin, Heidelberg, 2003.

[20] Rosen, Barry K., Mark N. Wegman, and F. Kenneth Zadeck. "Global value numbers and redundant computations." Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1988.

[21] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In PLDI '04, pages 75–88, Mar. 2004.

[22] llvm licm source code. https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/LICM.cpp.

[23] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. "Compilers, principles, techniques." Addison wesley 7.8 (1986): 9.

[24] Steffen, Bernhard, Jens Knoop, and Oliver Rüthing. "Efficient code motion and an adaption to strength reduction." International Joint Conference on Theory and Practice of Software Development. Springer, Berlin, Heidelberg, 1991.

[25] halide licm source code. https://github.com/halide/Halide/blob/master/src/LICM.cpp.

[26] Vasilev, V. S., and Alexander I. Legalov. "Loop-invariant Optimization in the Pifagor Language." Automatic Control and Computer Sciences 52.7 (2018): 843-849.

[27] mindspore akg source code. https://github.com/mindspore-ai/akg

[28] Rawat, Prashant Singh, et al. "Register optimizations for stencils on GPUs." Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2018.

[29] Lee, Juneyoung, et al. "Reconciling high-level optimizations and low-level code in LLVM." Proceedings of the ACM on Programming Languages 2.OOPSLA (2018): 1-28.

[30] llvm reassociate source code. https://github.com/llvm/llvm-project/blob/main/llvm/lib/Transforms/Scalar/Reassociate.cpp.

[31] Briggs, Preston, and Keith D. Cooper. "Effective partial redundancy elimination." ACM SIGPLAN Notices 29.6 (1994): 159-170.

[32] Lin D.C., "Compiler support for predicated execution in superscalar processors", M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1992.

[33] Effective Compiler Support for Predicated Execution Using the Hyperblock

[34] Bacon D.F., and Graham S.L., “Compiler transformations for high-performance computing”, ACM Computing Surveys, December 1994, Vol. 26, No. 4, pp.345-420.

[35] Wu, Jingyue, et al. ”gpucc: an open-source GPGPU compiler.” Proceedings of the 2016 International Symposium on Code Generation and Optimization. 2016.

[36] Chen, Tianshi, et al. ”Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning.” ACM SIGARCH Computer Architecture News 42.1 (2014): 269-284.