ACROBAT:在编译时优化动态深度学习的自动批处理

Pratik Fegade [1] Tianqi Chen [1 2] Phillip B. Gibbons [1] Todd C. Mowry [1]

摘要

动态控制流是一种重要的技术，通常用于设计具有表现力和高效的深度学习计算，用于文本解析、机器翻译、早期退出深度模型等应用程序。动态控制流导致的控制流发散使得批处理(batching)这一能够实现高吞吐量和硬件利用率的重要优化难以手动执行。在本文中，我们介绍了ACROBAT，这是一个通过执行混合静态+动态编译器优化和端到端张量代码生成来实现动态深度学习计算高效自动批处理的框架。在Nvidia GeForce GPU上，ACROBAT的性能比最先进的自动批处理框架DyNet高出8.5倍。

.

1 介绍

近年来，深度学习(DL)在广泛的应用中发挥着越来越大的作用。随着它们的应用变得越来越复杂，深度学习模型本身的大小和复杂度也在增加。对于推理服务以及训练，这些模型对今天的深度学习系统和硬件提出了极端的要求。

深度学习计算复杂性的一个重要来源是使用动态控制流作为执行的一部分。与静态前馈模型计算不同，使用动态控制流或动态计算的*计算*执行可能会因模型的不同输入而有所不同。在其他应用中，这一属性已被有效地用于(1)建模结构化数据，如解析树(Socher et al.， 2013a; 2012)和图像(Shuai et al.， 2015)，(2)通过使用束搜索来执行更好质量的机器翻译和文本解析(Wiseman & Rush, 2016;科恩,2004;Buckman等人，2016)，以及(3)提前退出卷积(Kaya & Dumitras, 2018;Teerapittayanon等人，2017)和transformer (Xin等人，2020;Elbayad et al.， 2019)模型，以降低推理延迟。因此，动态控制流提供的适应性在各种情况下都是有用的。

批处理是一种重要的优化，可以在训练和内部提高吞吐量和硬件利用率深度学习模型的推理。虽然对于静态DL计算很直接，但动态计算中控制流发散的存在使得手动批处理困难且容易出错。因此，过去有大量的工作在执行自动批处理，或*自动批处理*，用于动态DL计算。为了处理编译过程中动态计算执行知识的缺乏，过去的工作通常要么(1)严重依赖动态分析，使它们能够处理一般的动态控制流(Neubig et al.， 2017b;Looks et al.， 2017)，或(2)专门用于特定的控制流模式或模型，从而更多地依赖于静态分析(Xu et al.， 2018;Fegade et al.， 2021)。前一种框架往往*会因动态分析而招致较高的执行开销*，而后一种框架*缺乏通用性*，无法支持DL计算中*广泛存在*的和未来的控制流模式。

此外，过去的工作往往*严重依赖供应商库*，如cuDNN (Chetlur et al.， 2014)和oneAPI (Intel, 2022)。然而，由于实现供应商库是一个密集的过程，他们通常只实现常用的标准张量运算符。此外，由于这些核是单独优化的，没有任何关于它们所使用的更大应用程序的上下文知识，因此像核融合这样的重要优化无法再执行。

为了克服过去工作的这些限制，我们提出了ACROBAT[1]，一种用于动态DL计算的自动批处理框架，它依赖于新颖的*混合静态+动态优化*和*端到端张量核编译*。我们在设计acrobat1时的主要见解是，尽管在动态模型编译期间缺乏完美的执行知识，但编译器可以

[1]Carnegie Mellon University, Pittsburgh, USA [2]OctoAI. Correspondence to: Pratik Fegade <pratikfegade@gmail.com>.

Proceedings of the 5 th *MLSys Conference*, Santa Clara, CA, USA, 2024.

[1]Automated Compiler and Runtime-optimized Batching

表1。ACROBAT与其他自动批处理动态深度学习计算方案的比较。与ACROBAT的混合分析不同，纯静态或动态方法可能过于保守，或者分别有很高的开销。

框架	PyTorch	DyNet	皮质	TFFold	杂技演员
Auto-batch支持	没有	是的	是的	是的	是的
Auto-batch分析	-	Dyn.只	静态只	Dyn.只	混合动力
供应商库使用	高	高	没有一个	高	没有一个
普遍性	高	高	低	中期	高
用户impl。努力	低	低	高	低	低
表演	低	低	高	低	高

经常执行静态分析和优化以辅助动态分析。这减少了执行开销，同时有效地利用了输入计算中的并行性。ACROBAT依赖于传统的编译器技术，如上下文敏感性(Aho et al.， 2007)和污染分析，以及最少的用户注释来实现这种静态分析。此外，ACROBAT的端到端张量核生成使它能够自动生成核优化和专门用于更大的计算再次使用静态分析来识别和利用数据重用的机会(如我们在§5中看到)。ACROBAT的通用性允许使用简单的高级语言表达各种控制流模式，从简单的条件语句到复杂的递归计算。表1提供了ACROBAT与相关工作的定性比较。

简而言之，本文做出了以下贡献:

1. 本文调查和描述了在不同深度学习计算中发现的动态控制流结构。
2. 采用新颖的静态+动态混合优化和自动化端到端内核代码生成，设计了用于动态计算的自动批处理框架ACROBAT。这种设计使我们能够减少执行开销，并生成有效利用数据重用机会的高效张量核。在开发这些优化时，我们严重依赖于传统的编译技术。
3. 我们对ACROBAT进行原型设计，并根据最先进的深度学习框架对其进行评估(Xu et al.， 2018;Neubig等人，2017a;Paszke et al.， 2019)，并报告了在Nvidia gpu上的显著性能提升。

2 背景

2.1 DL计算中的动态控制流

在本节中，我们将在自动批处理问题的背景下，看看在各种DL计算中存在的不同类型的控制流动态性。这将告知我们如何设计一个系统，在动态深度学习计算的批处理执行中利用张量算子的并行性。

请注意，给定一个涉及控制流的计算，通常有多种方法来实现它。我们考虑

表2。在DL计算中发现的控制流属性。图例:ITE:迭代控制流，REC:递归控制流，TDC:模型显示张量依赖的控制流(其中控制流决策基于中间张量上的值)，IP:计算显示高度实例并行性，ICF:模型推理显示控制流，TCF:模型训练显示控制流。

深度学习计算	Ite	rec	TDC	IP	icf	TCF
RNN (Rumelhart 等 ，1986)、LSTM (Hochreiter & Schmidhuber, 1997)、GRU (Cho等，2014)、GraphRNN (You等，2018)	✓				✓	✓
transformer的推测解码(利维坦等人，2023)	✓		✓	✓	✓	
DIORA (Drozdov et al.， 2019)，中文分割(Chen等，2015)	✓			✓	✓	✓
DAG-RNN(帅 等 ，2015)，TreeL-STM (Socher 等 ，2013)，MV-RNN (Socher等，2012)		✓		✓	✓	✓
StackLSTM (Dyer等人，2015)	✓		✓		✓	✓
使用LSTM进行光束搜索(Wiseman & Rush, 2016)	✓		✓	✓	✓	
混合专家(Shazeer等人，2017;Ma等人，2018;Fedus等，2021年)			✓		✓	✓
早期退出模型(Kaya & Dumitras, 2018;Teerapittayanon等人，2017;艾尔巴亚德等人，2019年)			✓		✓	
树 对 树 NN(Chen 等 人，2018b)，双 递 归 NN(Alvarez-Melis & Jaakkola, 2017)		✓	✓	✓	✓	✓
R-CNN (Girshick等人，2013)，Fast R-CNN (Girshick, 2015)	✓		✓	✓	✓	✓

实现给定计算的最自然的方式。例如，自顶向下的树遍历可以实现为宽度优先遍历(BFS)或深度优先遍历(DFS)。虽然BFS遍历可能更有效，但基于dfs的遍历实现起来更自然。下面的讨论也在表2中进行了总结。

控制流围绕静态子图:我们观察到，对于大多数显示控制流动态的深度学习计算，动态控制流*围绕*张量计算。考虑由清单1中所示的@rnn函数实现的简单顺序RNN模型。在这里，我们看到顺序控制流围绕着第5行和第6行上的RNN单元，这是张量计算的静态子图，没有中间的控制流。

依赖张量的控制流:控制流的决定通常依赖于DL计算中中间张量的值。此类模型和计算的示例包括机器翻译中的束搜索、stack - stm (Dyer等人，2015)、树对树神经网络(T2TNN) (Chen等人，2018b)、早期退出模型(Kaya & Dumitras, 2018;Teerapittayanon等人，2017;辛等人，2020;Elbayad等人，2019)和mix -of- experts (Shazeer等人，2017;Ma et al.， 2018;Fedus et al.， 2021)。同时，在TreeLSTM (Socher等人，2013a)、DAG-RNN、顺序rnn及其变体等模型中，控制流仅依赖于输入，而不依赖于中间张量。

重复控制流:如果一个模型可以表示为迭代或递归计算，我们就说它具有重复的控制流。这包括这样的迭代模型

如rnn及其变体(例如LSTM和GRU (Cho等人，2014))和stacklstm，以及TreeLSTM、树对树神经网络和dag - rnn等递归模型(Shuai等人，2015)。另一方面，混合专家模型和早期退出模型不会表现出重复的控制流。这样的模型包含在一个静态前馈网络中的条件执行。重复的控制流通常也可以嵌套。例如，GraphRNN模型执行两个rnn，一个嵌套在另一个内部。类似地，用于自上而下递归生成树的DRNN模型，涉及对给定树节点迭代生成子节点。

递归的存在，与迭代控制流相反，往往可以使静态分析复杂化，因为并行性更容易与后者开发。我们在§4.2中看到，如何在运行时利用跨递归调用的并行性，例如，可以要求多个并发执行上下文，类似于fork-join并行范式(McCool et al.， 2012)。

训练和推理中的控制流:我们在表2中看到，许多模型的计算都涉及训练和推理期间的动态控制流。然而，这并不是具有早期退出的模型的情况，在训练期间，我们通常希望训练所有的退出分支，而不是评估一个，就像在推理期间的情况一样。此外，搜索过程(如束搜索)通常仅在推理期间使用，因此底层模型在训练期间可能不会表现出动态性(除非模型计算本身涉及动态性，例如在RNN模型的情况下)。

控制流并行性:动态控制流可以导致DL计算中的并行性。这样的计算可能会表现出(1)*Batch并行性*，它存在于mini-batch中的不同输入实例之间，和/或(2)*Instance Parallelism*，它指的是由于动态控制流依赖而产生的并行性，例如递归并行性。这种并行性的数量在不同的计算中差异很大。递归模型，通常(尽管不总是)在不同的递归调用之间具有显著的并行性。相应地，迭代计算可能包含可以并发执行的循环。一个例子是清单1中RNN实现中对@map函数调用的调用。

2.2 动态配料

ACROBAT基于动态批处理(Looks et al.， 2017;Neubig et al.， 2017b)，一种在动态控制流存在的情况下执行自动批处理的先验技术。给定一小批输入实例，动态批处理涉及在后台为每个实例构建张量操作符的数据流图(DFGs)的同时，延迟执行每个输入实例的模型计算。这些DFGs的执行是在请求特定张量的值时触发的(例如，当模型包含与张量相关的控制流时)。在执行过程中，运行时可以识别DFGs中的批处理机会，并适当地启动批处理内核。

3 ACROBAT:概述和API

控制流动态性要求依赖潜在昂贵的运行时分析来实现自动批处理。在ACROBAT中，我们观察到积极的静态分析通常提供足够的信息来减少这种分析的开销。这样的分析进一步使我们能够以端到端的方式生成专门的、更有效的张量核。

```
def @rnn(inps, state, bias, i_wt, h_wt) {
  match(inps) {
    Nil => Nil,
    Cons(inp, tail) => {
      let inp_linear = bias + nn.dense(inp, i_wt);
      let new_state = sigmoid(inp_linear + nn.dense(state, h_w));
      Cons(new_state, @rnn(tail, new_state, bias, i_w, h_w))
    }}}

def @main(rnn_bias: Tensor[(1, 256)], rnn_i_wt: Tensor[(256, 256)],
          rnn_h_wt: Tensor[(256, 256)], rnn_init: Tensor[(1, 256)],
          c_wt: Tensor[(16, 512)], cbias: Tensor[(1, 16)],
          inps: List[Tensor[(1, 256)]]) {
  (* Recursive computation stage (program phase 1) *)
  let rnn_res =
    @rnn(inps, rnn_init, rnn_bias, rnn_i_wt, rnn_h_wt);
  (* Output transformations stage (program phase 2) *)
  @map(fn(p: Tensor[(1, 256)]) {
         nn.relu(cbias + nn.dense(p, c_wt))
       }, rnn_res) }
```

清单1。用函数式语言表达的简单RNN模型(这里使用Relay (Roesch et al.， 2019)作为说明)作为ACROBAT的输入。

现在，我们将看看利用上述见解的ACROBAT的编译和执行工作流(如图1所示)。ACROBAT被设计成以简单的图灵完备函数语言表示的非批处理深度学习计算作为输入。这使得ACROBAT用户能够轻松地用动态控制流表示模型，例如§2.1中讨论的那些。例如，清单1演示了一个简单的RNN模型，ACROBAT可以将其作为输入。

给定一个输入计算1,acrobat中的编译从批处理的内核第2代开始。在这里，AC-ROBAT执行新颖的静态分析(§5.1)以识别数据重用机会，并相应地生成实现输入程序中使用的张量运算符的批核3。进一步，gather operator fusion(§5.2)使我们能够生成最小化数据移动的专用核。这些未优化的内核然后由自动调度器4进行优化。一旦优化，就可以为批处理的内核生成目标代码10，如CUDA c++。同时，输入程序进一步优化和以提前(AOT)方式编译以生成c++代码。作为此编译的一部分，ACROBAT gen-

Erates代码(1)通过我们的

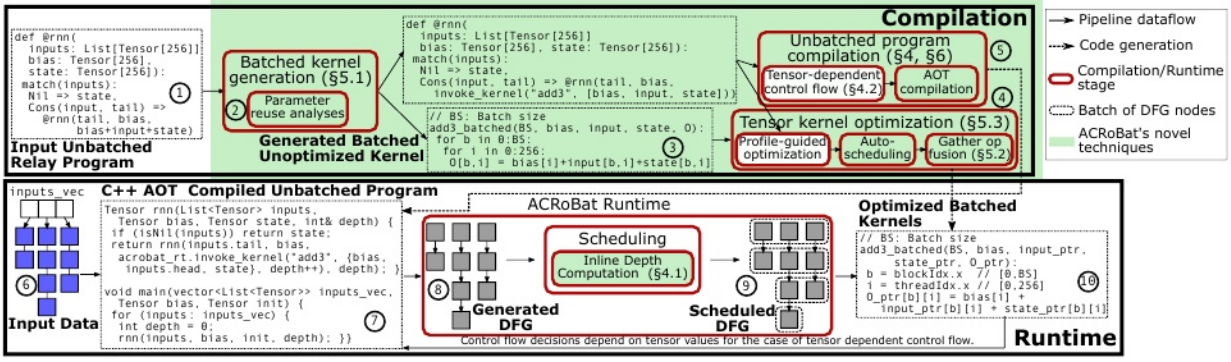

图1所示。ACROBAT工作流程概述。附录中的图7显示了之前完全动态方法DyNet的相应概述。请注意ACROBAT如何在编译时执行重要的新分析和代码生成，以减少运行时开销。

内联深度计算方法，以及(2)在存在张量依赖的控制流时自动启用并发执行(§4.2)。

在运行时，ACROBAT惰性地执行编译的AOT
在一个小批量的输入6上输入程序7，并构造DFGs 8。然后，ACROBAT运行时库将
调度这些DFGs(使用上面提到的内联深度计算)9，同时寻找批处理机会。然后，它将为每个已识别的批DFG节点调用优化的批处理内核10。如果输入程序显示张量相关的控制流，则执行循环返回到AOT编译程序，该程序将进一步执行并创建更多的DFGs。

现在我们来看看§4中的ACROBAT的混合优化和§5中的张量核生成。

4 混合静态+动态优化

动态控制流往往排除了静态程序转换。因此，ACROBAT采用混合方法，通过(1)为动态分析提供提示(§4.1)或(2)生成代码，为动态分析提供更大的并行性自由(§4.2)来利用静态程序知识。进一步，静态分析还使我们能够执行优化，如内核融合，这对高性能很重要(§7.4)。下面，我们提供了更多关于我们的混合分析的细节。

4.1 内联深度计算

正如过去的工作(Fegade et al.， 2021)所指出的，之前的完全动态方法会产生重大的调度开销。例如，正如我们将在表5中所示，DyNet的调度开销支配了TreeLSTM模型的张量计算时间。相反，如下所述，AC-ROBAT设计了一个方案来执行调度，因为它构建了DFGs，从而大大降低了调度开销(§7)。

DFG调度算法有两个目标:
G.1正确性:调度任务时要尊重任务之间的依赖关系。

G.2性能:识别和利用并行性。

给定一个DFG(s)，我们可以通过按其拓扑深度[2]的递增顺序执行DFG节点(每个节点代表一个张量算子)来满足这两个目标，从而同时执行相同深度的节点(Neubig等，2017a; Looks et al.， 2017)。为了在DFG构造期间计算这些深度，我们做了以下两个观察:

0.1未批处理程序调用张量运算符的顺序，即节点添加到DFGs的顺序，是一个有效的依赖顺序。

关于实例并行性的信息(例如，表2中所示的TreeLSTM模型中的递归并行性)通常在编译期间可用。

```
1  List<Tensor> rnn(List<Tensor> inps, Tensor state, Tensor bias,
2                   Tensor i_wt, Tensor h_wt, int& depth) {
3    if (inps == ListNil()) return ListNil();
4    auto inp_linear = AcrobatRT.InvokeKernel("bias_dense",
5                        0, {bias, i_wt, inps.head});
6    auto new_state = AcrobatRT.InvokeKernel("sigmoid_add_dense",
7                        , {inp_linear, h_wt, state});
8    return ListCons(new_state, rnn(inps.tail, state, bias, i_wt,
9                                   h_wt, )); }
10
11 vector<Tensor> main(Tensor rnn_bias, Tensor rnn_i_wt,
12                     Tensor rnn_h_wt, Tensor rnn_init, Tensor c_wt,
13                     Tensor cbias, vector<List<Tensor>> inps_vec) {
14   vector<Tensor> res;
15   for (auto inps: inps_vec) {
16
17     /* Recursive computation stage (program phase 1) */
18     auto rnn_res = rnn(inps, rnn_init, rnn_bias, rnn_i_wt,
19                        rnn_h_wt, );
20     /* Output transformations stage (program phase 2) */
21
22     res.push_back(map([&](Tensor p) { AcrobatRT.InvokeKernel(
23       "relu_bias_dense", , {cbias, c_wt, p}); }, rnn_res)); }
24   return res; }
```

清单2。AOT为清单1中的RNN模型编译了输出，突出显示了内联深度计算代码。

基于这些观察，我们设置了一个opero -的深度

[2]如果P(n)表示一个节点n消耗的所有张量的所有生产者的所有集合，那么它的深度d_n由$d_n = 1 + \max_{p \in P(n)} d_p$如果P(n) 6=∅，否则为0。

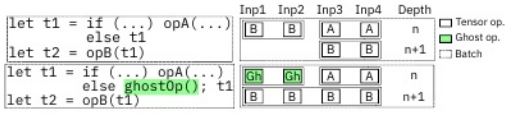

图3。幽灵操作员可以实现更好的批处理。

ator等于其在执行未批处理程序引起的依赖顺序中的位置，从而满足目标G.1。然后，我们依靠上面的观察0.2，通过使用以下技术来发现和利用并行性的机会:

实例并行性:我们注意到实例并行性通常源于递归或在独立项列表上使用@map函数(观察0.2)。我们len - 确保这样的并发操作符在执行未批处理的程序时被分配相同的深度。我们

```
funA() {
 concurrent {
  funA(); funA();
 }
 funC();
}
```

图2。并发调用注释。

依赖简单的用户注释来获取递归并行性的信息m3。图2展示了一个例子，对funA的两次递归调用被标注为并发。还要注意，过去的工作自动并行化(Hogen et al., 1992;Aleen & Clark, 2009)可能会用来代替这样的注释。清单2显示了为清单1中的RNN模型生成的AOT编译代码。我们看到，在第23行，@map函数内部所有对relu偏差稠密核的调用都被分配了相同的深度。

对抗深度调度的渴望:正如在过去的工作中所指出的(Neubig等人，2017b)，基于深度的调度方案，如ACROBAT使用的方案，在执行张量算子时往往过于渴望，导致利用并行性的次优量。过去的工作依赖于*基于议程的调度*(Neubig等人，2017b)，这是一种更昂贵的调度方案，作为基于深度的方案的替代方案来缓解这个问题。相反，ACROBAT依赖于编译时分析，如下所述。

幽灵操作:在有条件if语句的情况下，急于批处理会导致次优批处理，如图3上方窗格所示。我们看到，由于输入Inp1和Inp2的操作B实例被急切地批处理，更重要的是，与输入Inp3和Inp4的操作B实例分开，立即批处理会导致次优的批处理时间表。在这种情况下，ACROBAT可以静态地插入*幽灵操作*，从而延迟某些操作的调度和执行，如图下方窗格所示。注意，幽灵操作仅仅影响ACROBAT的调度行为，在张量内核执行期间被忽略。

[3]Users can mark a set of function calls as concurrent in the input code. Of the seven models we evaluate in § 7, four required one such annotation each, while the rest did not require any.

程序阶段:另一方面，当重复(递归或迭代)控制流存在时，我们依赖*程序阶段*(Sherwood et al., 2003)来对抗上述调度的次优性。给定了这些程序阶段的知识，ACROBAT等待调度和执行一个阶段中的操作符，直到所有先前阶段的操作符都被调度和执行。我们发现，将输入深度学习计算的单个语义阶段考虑为单个阶段是将计算划分为阶段的一个很好的启发式。ACROBAT还为用户提供了一种通过手动注释程序阶段来覆盖这种启发式的方法，尽管在我们的评估中，我们不需要这样的注释。我们在§A中提供了有关程序阶段和幽灵操作的更多细节和解释。附录3。

此外，ACROBAT还能够静态提升操作员，我们在§A中更详细地描述。附录1。例如，在清单2中，第5行对内核bias dense的调用被赋值为静态计算的深度0，这在运行时有效地将内核调用提升出递归。

4.2 张量依赖的控制流

ACROBAT惰性地执行非批处理程序，为批处理中的每个输入实例创建DFGs。在没有张量相关控制流的情况下，我们可以先对每个实例顺序执行未批处理程序，然后一次触发所有DFGs的批处理和执行。然而，在依赖张量的控制流存在的情况下，这样的顺序执行将不允许我们利用任何批处理并行性，因为我们需要在依赖于中间张量值的控制流决策处触发执行。虽然以前的工作将重构输入计算的负担放在用户身上以减轻这个问题，但ACROBAT通过使用*fiber*[4]*自动生成代码,对每个输入实例并发地*执行未批处理程序。通过这种方式，可以为每个实例执行未批处理的程序，在不触发DFG评估的情况下，任何程序都无法进行。在这一点上，可以执行评估，并在图4所示的情况下恢复并发执行。相应地，为了在依赖张量的控制流存在的情况下利用实例并行性，ACROBAT启动并发纤维，类似于并行的fork-join模型(McCool et al., 2012)。因此，ACROBAT将并行性的静态知识与动态并发执行相结合，作为其混合分析的一部分，以有效地利用依赖张量控制流存在的并行性。

[4]Fibers (Boost, 2022) allow multiple execution stacks to be cooperatively scheduled on a single process.

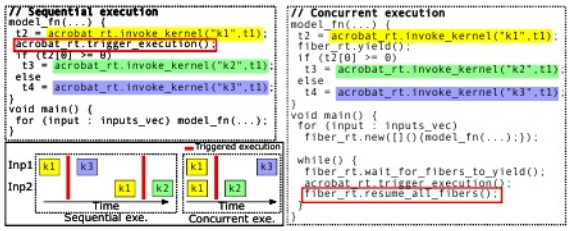

图4。在张量依赖控制流存在的情况下，并发执行未批处理的程序。

表3。评估中使用的模型和数据集。

Model	Description	Dataset
TreeLSTM	TreeLSTM	Stanford sentiment treebank (Socher et al., 2013b)
MV-RNN	MV-RNN	Stanford sentiment treebank
BiRNN	Bidirectional RNNs	XNLI (Conneau et al., 2018)
NestedRNN	An RNN loop nested inside a GRU loop	GRU/RNN loops iterate for a random number of iterations in [20, 40].
DRNN	Doubly recurrent neural networks for top-down tree generation	Randomly generated tensors.
Berxit	Early exit for BERT inference (Xin et al., 2021). All layers share weights.	Sequence length 128.
StackRNN	StackLSTM parser with LSTM cells replaced by RNN cells.	XNLI

5 端到端张量核生成

正如我们上面提到的，ACROBAT通过避免使用供应商库来实现端到端的、统一的和自动的张量内核代码生成。这允许ACRO-BAT支持更大的操作符集，而无需额外的编译器开发工作。下面提供了关于ACRO-BAT张量核生成的更多细节。

5.1 利用参数重用

给定输入的非批处理计算，ACROBAT需要生成批处理核来实现计算中使用的张量算子。生成这些核并不简单，因为一些输入张量(通常是模型参数)可能会在对运算符的调用之间共享。例如，在对图1中输入计算1中使用的元素级加法运算符add3的多次调用中，偏差参数将被共享(因为它是一个模型参数)，因此应该在参数输入和状态的所有值中重用。这可以在图1中对应的批处理内核(3和10)中看到。

完全动态的自动批处理方法，例如DyNet中使用的方法，无法准确识别此类参数重用，而是依赖于启发式，这可能是脆弱的，导致次优性能(§7.3)。另一方面，ACROBAT使用1上下文敏感的污染分析来识别张量操作符的共享参数。这里使用静态分析允许ACROBAT获得关于参数重用模式的准确知识。

除了上述分析之外，ACROBAT还通过采用§B.1中描述的代码复制和水平融合进一步探索了数据重用的机会。

[5]Context sensitivity is a static analysis technique that allows the compiler to reason about a function in the different contexts it may be called under leading to increased analysis precision. For the DL computations we worked with, we found that a 1-context sensitive analysis was sufficient. Deeper contexts might be useful, however, for more complex computations.

5.2 融合内存聚集操作

由于ACROBAT动态地识别跨DFG的批处理机会，因此批处理中所有DFG节点的输入张量可能不会连续地布置在加速器的内存中。在这种情况下，之前的工作在对张量进行操作之前(通过调用供应商库内核)执行内存收集，从而导致重要的数据移动(§7.4)。相反，ACROBAT生成专门的批处理内核来直接操作分散在内存中的张量，从而有效地将昂贵的收集操作与批处理内核融合在一起。图1中生成的批处理内核10说明了这一点。如§7所示，这种融合可以带来显著的性能提升。

6 实现细节

我们的ACROBAT原型是基于TVM (Chen et al., 2018a) v0.9.dev0、DL框架和张量编译器构建的。因此，它接受以Relay表示的输入计算。我们的原型，ACROBAT也执行粒度粗化优化(Zha等人，2019;Xu et al.， 2018;Fegade等，2021年;高等，2018;Silfa et al., 2020)，详见§A。附录2。

如§7.2所示，我们发现在控制流动态性存在的情况下，使用解释型虚拟机(VM)执行未批处理的程序会导致显著的VM开销。因此，ACROBAT以AOT方式将输入计算编译为c++(如附录§C中所讨论的)。此外，由于TVM不支持训练，我们评估ACROBAT对深度学习计算的(批处理)推理。其他实现细节，包括ACROBAT使用TVM自动调度器的细节，可以在§C的附录中找到。

7 评价

我们现在将ACROBAT与Nvidia GPU上的Cortex和DyNet进行比较。Cortex和DyNet都是用于深度学习计算的最先进的自动批处理框架，分别显示递归和一般不受限制的控制流。它们已被证明比通用框架更快

表4。中继虚拟机与ACROBAT的AOT编译:以毫秒为单位的推理延迟。

隐藏的大小	批量大小	TreeLSTM		MV-RNN		BiRNN	
		虚拟机	AOT	虚拟机	AOT	虚拟机	AOT
小	8	30.68	2.66	4.0	0.55	29.88	2.23
小	64	28.94	9.47	3.91	1.63	28.88	5.47
大	8	31.64	3.85	4.34	1.06	32.04	4.82
大	64	29.49	15.9	4.36	4.6	30.43	13.72

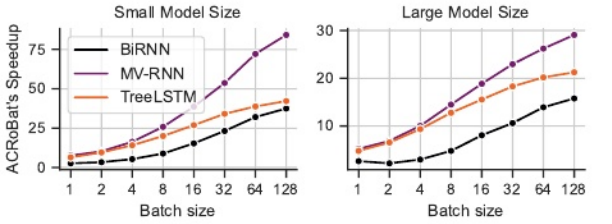

图5。在PyTorch上为TreeLSTM、MV-RNN和BiRNN模型获得的加速。

如 PyTorch 和 TensorFlow (Neubig 等， 2017a;b;Fegade et al.， 2021)。我们还比较了ACROBAT与PyTorch(§??)的性能。

7.1 实验装置

模型:我们使用表3中列出的模型进行评估。对于每个模型，我们看两种模型尺寸——小模型和大模型。对于MV-RNN模型，我们对小型和大型模型尺寸使用隐藏尺寸64和128，而对于Berxit模型，小型模型使用与$\text{BERT}_{\text{BASE}}$模型相同的超参数(Devlin等人，2018)，而大型模型使用与$\text{BERT}_{\text{LARGE}}$模型相同的超参数(Devlin等人，2018)，只是我们在这种情况下使用18层而不是24层。对于剩余的模型，小模型和大模型尺寸分别使用256和512的隐藏尺寸。

实验环境:我们在带有AMD Ryzen Threadripper 3970X CPU(64个具有2-way超线程的逻辑核心)和Nvidia RTX 3070 GPU的Linux工作站上运行实验。机器运行Ubuntu 20.04, CUDA 11.1和cuDNN 8.0.5。我们将其与DyNet的commit 3e1b48c7(2022年3月)进行比较，后者使用了Eigen库(v3.3.90)。

7.2 AOT编译的好处

我们首先看看AOT编译的好处(§6)。TreeLSTM、MV-RNN和BiRNN模型[6]使用Relay VM和ACROBAT的AOT编译器(粒度粗化、集合算子融合和程序阶段优化打开)执行时的性能如表4所示。我们看到，与针对这些模型的AOT编译的本机代码相比，这些开销显著降低了执行速度(最高降低了13.45倍)。因此，在本节的其余部分，我们将在打开AOT编译时评估ACROBAT的性能。

7.3 整体性能

在本节中，我们将ACROBAT的性能与PyTorch、DyNet和Cortex进行比较。

[6]ACROBAT's prototype implementation does not currently support the execution of the remaining models in Table 3 using the Relay VM.

与PyTorch的性能比较

图5比较了ACROBAT与PyTorch (v1.9.0a0+gitf096245)在TreeLSTM、MV-RNN和BiRNN模型下的性能[7]。PyTorch不执行自动批处理，因此无法在计算中利用任何可用的实例或批处理并行性。此外，相对于PyTorch, ACROBAT的内核融合和其他静态优化也提高了它的性能。与较大的模型大小相比，小模型大小的加速比更高，因为开发实例和批处理并行性对于大模型大小的相对重要性较低，这是因为单个张量算子的并行性增加了。与其他两个模型相比，ACROBAT在BiRNN模型上的相对较差的性能可归因于BiRNN中缺乏实例并行性，导致ACROBAT可以利用的并行性较低。同样，由于与MV-RNN相比，TreeLSTM表现出更高的静态和张量并行性，因此利用实例和批处理并行性的相对重要性较低，导致性能低于MV-RNN。

与DyNet的性能比较

现在我们将ACROBAT的性能与DyNet进行比较。正如§6中提到的，TVM不支持DL模型的训练。因此，由于缺乏对训练模型参数的访问，我们使用伪随机性来模拟NestedRNN、DRNN、Berxit和StackRNN模型中的张量相关控制流，作为我们评估的一部分。通过使用预先确定的随机种子进行公平比较，我们确保伪随机性在ACROBAT和DyNet实现中是一致的。当执行内联深度计算时，DRNN模型是一个例外。在这种情况下，ACROBAT利用DRNN的递归实例并行性使用纤维(§4.2)导致随机控制流决策的变化。我们通过呈现50个不同的随机种子的平均执行时间来解释这一点。

DyNet和ACROBAT的执行延迟为

[7]We use TorchScript only for the BiRNN model as it does not currently support recursive data types (PyTorch Community, 2020), such as the parse trees the TreeLSTM and MV-RNN models operate on.

表5所示。DyNet与ACROBAT:推理延迟(DyNet/ACROBAT)，单位为毫秒和加速。由于批量大小为64的内存不足错误，Berxit模型的DyNet实现被终止。

隐藏的大小	批量大小	TreeLSTM		MV-RNN		BiRNN		NestedRNN		DRNN		Berxit		StackRNN	
		时间	加速	时间	加速	时间	加速	时间	加速	时间	加速	时间	加速	时间	加速
小	8	4.31/1.48	2.93	2.11/0.54	3.96	3.13/2.16	1.45	29.38/31.01	0.95	6.7/1.74	3.87	63.54/38.49	1.66	47.78/22.69	2.11
小	64	26.18/5.81	4.51	12.45/1.48	8.47	12.04/4.86	2.49	84.55/65.73	1.29	25.3/5.24	4.84	—–/ 204.54	-	213.98/39.06	5.48
大	8	4.58/2.4	1.92	2.27/1.04	2.19	3.95/4.43	0.9	46.03/35.61	1.3	8.44/2.45	3.45	113.18/64.49	1.76	64.67/43.75	1.48
大	64	26.53/11.44	2.33	13.89/4.46	3.13	12.11/13.11	0.93	94.97/100.17	0.95	26.5/9.99	2.66	—–/ 335.3	-	230.74/86.82	2.66

表6所示。对于批大小为64的动力网和ACROBAT，在各种活动中花费的时间(毫秒)[1]。

活动	TreeLSTM，小		BiRNN，大	
	动力网	ACROBAT	动力网	ACROBAT
DFG构造	8.8	1.5	4.5	1.0
调度	9.7	0.4	3.3	0.4
内存复制时间	3.1	0.1	2.3	0.2
GPU内核时间[2]	6.1	4.0	6.6	11.2
#内核调用	1653	183	580	380
CUDA API时间[3]	16.5	3.9	12.0	11.1

[1]报告的时间对应于多次运行，并使用Nvidia Nsight系统通过手动插装和profiling获得。由于分析开销，执行时间可能与表5中的不匹配。

[2]包括内存复制内核。

包括调用cudaMemcpy, cudaMemcpyAsync和所有内核。

如表5-[8]。由于许多原因，ACROBAT在大多数情况下比DyNet性能更好。表6列出了框架为TreeLSTM模型的不同运行时活动所花费的时间。我们看到，ACROBAT的静态核融合和粒度粗化等优化减少了调用张量核的数量，从而显著降低了DFG的构建和调度开销。此外，内联深度计算允许acrobat以较低的开销利用可用的并行性。静态内核融合和聚集算子融合等优化使ACROBAT能够启动更少的GPU内核，进一步减少在CUDA API上花费的时间。我们将在§7.4中更详细地了解每个ACROBAT优化的好处。

虽然总体而言，ACROBAT在所有模型配置上的性能比DyNet好2.3倍，但在BiRNN和NestedRNN模型的某些配置上，DyNet的性能略好于ACROBAT。对于前者，表6显示，虽然ACROBAT在DFG构建、调度和内存传输方面的运行时开销较低，但与DyNet相比，它在内核执行上花费的时间更多。我们相信，更好的张量核优化可以帮助缩小这种性能差距。

除了上述原因之外，ACROBAT在特定基准测试中表现更好，原因如下所述。

准确的参数重用推理和自动化的批处理内核生成:如§5.1所述，AC-ROBAT使用静态分析来推断参数重用，使其具有准确的知识来静态生成适当的批处理内核。另一方面，DyNet基于启发式的方法无法批量导入

表7所示。对TreeLSTM、MV-RNN和DRNN模型进行§7.3中描述的改进后的模型执行时间(以毫秒为单位)。DN、DN++和AB分别代表DyNet、DyNet改进版和ACROBAT。

模型尺寸	批量大小	TreeLSTM			MV-RNN			DRNN		
		DN	DN	AB	DN	DN	AB	DN	DN	AB
小	8	4.31	3.8	1.48	2.11	1.05	0.54	6.7	3.29	1.74
小	64	26.18	22.69	5.81	12.45	3.15	1.48 - 25.3		18.51	5.24
大	8	4.58	4.14	2.4	2.27	1.83	1.04 - 8.44		3.82	2.45
大	64	26.53	24.09	11.44 - 13.89		10.47	4.46 - 26.5		18.86	9.99

某些操作符的姿态，强制顺序非批量执行，导致性能低下。例如，只有当所有实例的第一个参数是同一个张量时，DyNet才会启发式地批处理矩阵乘法运算符的多个实例。这通常是有效的，因为第一个参数通常是一个模型参数，通常作为线性变换的一部分。然而，我们的MV-RNN模型的DyNet实现将两个中间张量激活叠加在一起，因此DyNet无法批量处理此操作符的实例，从而强制顺序非批量执行。当我们修改DyNet的启发式矩阵乘法时，其性能显著提高，如表7所示。

此外，如§5所述，与依赖供应商库的DyNet等方法相比，ACROBAT的端到端内核生成导致支持批处理的张量运算符的更广泛覆盖。因此，DyNet不支持某些操作的批处理，这再次导致顺序执行和低性能。具体来说，DyNet不支持argmax操作符的批处理执行，StackRNN模型使用argmax操作符，以便根据嵌入的RNN单元的结果确定每次迭代中的下一个解析器操作。类似地，DRNN模型中使用的元素乘运算符，在需要执行广播时以非批处理的方式执行。另一方面，acroc - bat自动生成这些张量运算符的优化的批处理实现。我们还发现DyNet无法批量调用构造常数张量的运算符。我们使用这个操作符初始化TreeLSTM模型中树叶的隐藏状态。另一方面，ACROBAT静态地识别常数张量可以被重用，因此只创建一次张量。当我们在DyNet中手动利用这种重用时，TreeLSTM模型的性能得到了改善，如表7所示。

[8]We consider the best of the two scheduling schemes DyNet implements (Neubig et al., 2017b) for each model configuration.

表8所示。注意，与ACROBAT不同，Cortex仅限于递归计算，并且不支持表3中的其他模型。此外，Cortex通过依赖手工的内核优化，给用户带来了很高的开发负担。

隐藏的批处理小		TreeLSTM		MV-RNN		BiRNN	
		皮质ACROBAT	皮质ACROBAT	皮质ACROBAT	皮质ACROBAT	皮质ACROBAT	皮质
小	8	0.79	1.48	1.14	0.54	1.28	2.16
小	64	3.62	5.81	6.92	1.48	3.48	4.86
大	8	1.84	2.4	5.3	1.04	2.47	4.43
大	64	10.23	11.44	41.15	4.46	10.74	13.11

用于处理张量相关控制流的自动代码生成:DRNN模型以自上而下递归的方式从输入向量表示构建树。它既展示了张量依赖的控制流，也展示了实例并行性(可以同时生成多个子树)。我们在§4.2中看到了ACROBAT如何在张量相关的控制流中使用纤维自动利用实例并行性。另一方面，DyNet无法利用这种并行性，因此ACROBAT在该模型上的性能明显优于DyNet。表7还显示了在DyNet中为DRNN模型获得的性能改进，当模型计算所显示的实例并行性像上面详细介绍的那样被手动利用时。

与Cortex的性能比较

表8比较了ACROBAT与Cortex在TreeLSTM、MV-RNN和BiRNN模型下的性能。请注意，这不是一个苹果与苹果的比较，因为专门用于递归计算的Cortex不支持一般的控制流(如表3中的其他模型所示)，不像表1中提到的ACROBAT。此外，与ACROBAT的自动内核生成不同，Cortex给需要为特定硬件手动优化和调整模型的用户带来了很高的开发负担。类似地，虽然ACROBAT可以在TreeLSTM和BiRNN模型的递归计算中自动提升输入线性变换(如§A.1所述)，但在Cortex的情况下，它们需要手动提升和卸载到cuBLAS。

由于高度专门用于递归计算，Cortex能够利用积极的内核融合、模型持久性和低内核调用开销，因此在TreeLSTM和BiRNN模型上的性能比ACROBAT高1.87倍。然而，请注意，Cortex在MV-RNN模型上的表现比ACROBAT差得多。这是因为Cortex的限制性API需要输入解析树的叶子的嵌入向量的额外副本，这是ACROBAT可以避免的，因为它更灵活的交互脸。总的来说，ACROBAT提供了与Cortex相当的性能，同时支持更广泛的深度学习计算，开发人员的工作量要小得多。

7.4 优化的好处

现在我们评估ACROBAT执行的不同优化的相对好处。图6显示了我们逐步执行优化时，表3中模型的执行时间(批量大小为64)。标准的核融合(即不包括§5.2中讨论的聚集算子融合的核融合)为所有模型提供了显著的好处[10]。粒度粗化和内联深度计算，两者都减少了调度开销，对于TreeLSTM和MV-RNN等控制流相对较高的模型最有利。此外，在DRNN模型的情况下，内联深度计算还使ACROBAT能够利用计算中固有的实例并行性(§4.2)，从而降低执行时间。与token分类一样，BiRNN模型涉及到每个token输出的线性算子。在这里，程序阶段允许ACROBAT将所有这些操作符批处理在一起，如§4.1所述。StackRNN模型根据当前解析器动作执行不同的张量运算符，这涉及到一个条件语句。因此，Ghost算子能够更优化地利用并行性，从而带来更好的性能。

聚集算子融合对某些基准是有利的，但对其他基准则不是。这样的融合会导致间接的内存访问，从而导致内核执行速度变慢。虽然ACROBAT在适当的时候会将这些负载从循环中提升出来，但这并不总是可能的，这取决于自动调度程序生成的调度。此外，聚集操作符融合会导致速度变慢，主要是在具有迭代执行和很少实例并行的模型中。与DyNet一样，当收集算子融合关闭时，ACROBAT仅在输入张量在内存中不是连续的情况下执行显式内存收集。这种情况在这样的迭代模型中更有可能出现，从而钝化了聚集算子融合的优势。此外，在Berxit等模型中，粗化静态块的相对较高的张量计算成本进一步降低了集合算子融合可能提供的任何好处。

总的来说，具有相对较少的控制流或较高的张量计算量的模型(如Berxit或NestedRNN)或具有大尺寸的模型从减少调度开销的优化中获益较少。

[9]For example, implementing the MV-RNN model in Cortex requires 325 LoC in Python, as compared to the 79 LoC of Relay and 108 LoC of Python in ACROBAT.

[10]The kernels used in the implementations with and without standard kernel fusion were auto-scheduled for the same number of auto-scheduler iterations.

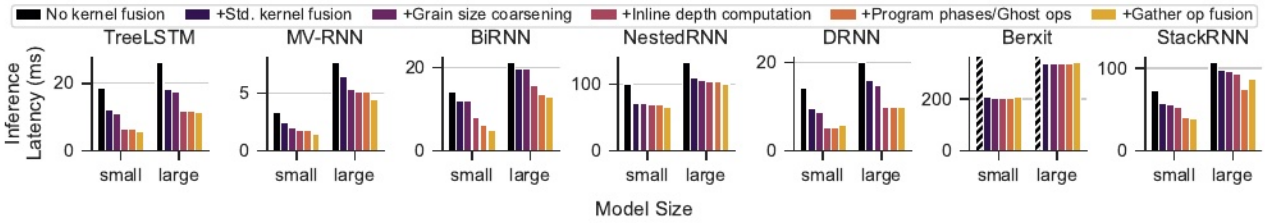

图6。的好处　　不同的优化。Berxit的未融合执行由于内存不足错误而被杀死。

8 相关工作

用于动态控制流的自动批处理:关于动态计算的自动批处理技术已经有了大量的工作。除了动态批处理(在DyNet、TensorFlow Fold、Cavs、Cortex和ByteTransformer (Zhai et al.， 2023)中以各种形式用于变压器模型)之外，静态程序转换(Bradbury & Fu, 2018;阿格沃尔,2019;阿加瓦尔&加尼切夫，2019;Frostig等人，2018;Radul等人，2020)也探索了自动批处理。如(Radul等人，2020)所述，此类技术通常无法充分利用程序中所有可用的并行性。ACRO-BAT以这些过去的技术为基础，有效地使用静态和动态分析，从而在利用所有可用的并行性的同时实现更低的运行时开销。用于低延迟RNN推理的在线批处理方法，如BatchMaker (Gao等人，2018)和E-BATCH (Silfa等人，2020)是对ACRO-BAT的补充。(Qiao & Taura, 2019)提出了对反向传播的动态批处理技术的改进，而ED-Batch (Chen et al.， 2023)提出了动态批处理的调度和内存规划的有效方法。这些可以通过ACROBAT的混合优化进一步改进。此外，虽然粒度粗化在过去的工作中进行了探索，但我们在通用自动批处理框架的背景下静态地使用它。

优化动态DL计算:除了自动批处理，还有大量关于优化动态DL计算执行的工作。过去的工作(Jeong等人，2019;Kim等人，2021年;Suhan et al.， 2021)探索了可以优化以加速动态模型的dgs的惰性创建。也有工作(Durvasula et al.， 2024;Zheng et al.， 2023)在更好的调度和低开销执行张量核方面进行了研究，以优化动态DL计算的动态执行模式。进一步SoD^2(Niu et al.， 2024)开发了优化动态计算的技术，包括那些具有动态形状的计算。这些技术不执行批处理，是对ACROBAT技术的补充。虽然ACROBAT是基于TVM构建的，但我们的技术可以在其他常用的编译器框架中实现——具有表达性表示(PyTorch, 2020;Lattner et al.， 2020)以一种直接的方式。

收集算子融合优化类似于CUTLASS库中对稀疏GEMM执行的收集和散射融合(CUTLASS, 2022)，尽管我们将此优化作为编译的一部分自动执行。如§C所述。1、ACROBAT借用了DietCode中的一些技术来高效地生成代码。DietCode的技术是对我们技术的补充，它可以完全集成到ACROBAT中以获得更好的内核性能。

传统的编译技术:ACROBAT使用编译技术编写通用语言的程序。这些技术包括上下文敏感(Aho et al.， 2007)，被广泛用于安全目的的污点分析(Tripp et al.， 2009;Huang等人，2015)，轮廓引导优化(Chen等人，2006;Gupta et al.， 2002)(如§C所述。附录1)和程序阶段，它们已被用于自适应优化程序的不同部分的系统，以获得最佳性能(Huang等人，2001;Barnes et al.， 2002)。ACROBAT的内联深度计算和DFG调度更一般地类似于流水线和标量处理器的静态和动态指令调度(Smith, 1989;Ponomarev等人，2001;费舍尔,1981;吉本斯和穆奇尼克，1986)。然而，ACROBAT在DL框架的上下文中应用了这些技术。

9 结论

本文介绍了ACROBAT，它是一个编译器和运行时框架，用于自动批处理动态DL计算。ACROBAT采用混合静态+动态分析，以低运行时开销实现有效的批处理，并使用端到端代码生成来生成高度优化的张量核，以实现高效执行。虽然我们仅在批推理的情况下评估了这些技术，但我们相信它们也适用于DL训练。在动态深度学习计算日益重要的背景下，我们认为ACROBAT是迈向深度学习框架的各个组件(如张量编译器、高级语言编译器和运行时)之间更多协作关系的重要一步。

致谢

这项工作得到了美国国家科学基金会(CNS-2211882)、Oracle、IBM、高通、DARPA(实时机器学习、

或RTML项目)和并行数据实验室(PDL)联盟(亚马逊，Facebook，谷歌，惠普企业，日立，IBM，英特尔，微软，NetApp, Oracle, Pure Storage, Salesforce，三星，希捷，TwoSigma和西部数据)。我们要感谢Saman Amarasinghe、Dominic Chen、Siyuan Chen、Stephen Chou、Chris Fallin、Graham Neubig、Olatunji Ruwase和卡内基梅隆大学催化剂研究小组对我们的工作提出的宝贵建议和反馈。

参考文献

Agarwal, A. Static automatic batching in TensorFlow. In Chaudhuri, K. and Salakhutdinov, R. (eds.), *Pro-ceedings of the 36th International Conference on Ma-chine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 92–101. PMLR, 09–15 Jun 2019. URL https://proceedings.mlr.press/v97/agarwal19a.html.

Agarwal, A. and Ganichev, I. Auto-vectorizing tensorflow graphs: Jacobians, auto-batching and beyond. *CoRR*, abs/1903.04243, 2019. URL http://arxiv.org/abs/1903.04243.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compil-ers: principles, techniques, & tools*. Pearson Education India, 2007.

Aleen, F. and Clark, N. Commutativity analysis for soft-ware parallelization: Letting program transformations see the big picture. *SIGARCH Comput. Archit. News*, 37(1):241–252, mar 2009. ISSN 0163-5964. doi: 10.1145/2528521.1508273. URL https://doi.org/10.1145/2528521.1508273.

Alvarez-Melis, D. and Jaakkola, T. Tree-structured decoding with doubly-recurrent neural networks. In *ICLR*, 2017.

Barnes, R., Nystrom, E., Merten, M., and Hwu, W. Vacuum packing: extracting hardware-detected program phases for post-link optimization. In *35th Annual IEEE/ACM International Symposium on Microarchitecture, 2002. (MICRO-35). Proceedings.*, pp. 233–244, 2002. doi: 10.1109/MICRO.2002.1176253.

Boost. Boost.Fiber, 2022. URL https: //www.boost.org/doc/libs/1_79_0/libs/fiber/doc/html/index.html. Last accessed July 1, 2022.

Bradbury, J. and Fu, C. Automatic batching as a compiler pass in pytorch. In *Workshop on Systems for ML*, 2018.

Buckman, J., Ballesteros, M., and Dyer, C. Transition-based dependency parsing with heuristic backtracking. In *Proceedings of the 2016 Conference on Empirical Meth-ods in Natural Language Processing*, pp. 2313–2318, Austin, Texas, November 2016. Association for Com-putational Linguistics. URL https://aclweb.org/anthology/D16-1254.

Chen, S., Fegade, P., Chen, T., Gibbons, P. B., and Mowry, T. C. ED-batch: efficient automatic batching of dynamic neural networks via learned finite state machines. In *Proceedings of the 40th International Conference on Ma-chine Learning*, ICML'23. JMLR.org, 2023.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, Carlsbad, CA, October 2018a. USENIX Association. ISBN 978-1-939133-08-3. URL https://www.usenix.org/conference/osdi18/presentation/chen.

Chen, W.-k., Bhansali, S., Chilimbi, T., Gao, X., and Chuang, W. Profile-guided proactive garbage collection for locality optimization. *SIGPLAN Not.*, 41(6):332–340, jun 2006. ISSN 0362-1340. doi: 10.1145/1133255.1134021. URL https://doi.org/10.1145/1133255.1134021.

Chen, X., Qiu, X., Zhu, C., and Huang, X. Gated re-cursive neural network for Chinese word segmentation. In *Proceedings of the 53rd Annual Meeting of the As-sociation for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 1744–1753, Beijing, China, July 2015. Association for Computa-tional Linguistics. doi: 10.3115/v1/P15-1168. URL https://aclanthology.org/P15-1168.

Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. *CoRR*, abs/1802.03691, 2018b. URL http://arxiv.org/abs/1802.03691.

Chetlur, S., Woolley, C., Vandermersch, P., Cohen, J., Tran, J., Catanzaro, B., and Shelhamer, E. cuDNN: Efficient primitives for deep learning. *CoRR*, abs/1410.0759, 2014. URL http://arxiv.org/abs/1410.0759.

Cho, K., van Merrienboer, B., G¨ulc¸ehre, C¸., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase represen-tations using RNN encoder-decoder for statistical ma-chine translation. *CoRR*, abs/1406.1078, 2014. URL http://arxiv.org/abs/1406.1078.

Conneau, A., Rinott, R., Lample, G., Williams, A., Bowman, S. R., Schwenk, H., and Stoyanov, V. XNLI: Evaluating cross-lingual sentence representations. In *Proceedings*

of the 2018 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2018.

CUTLASS. Gather and Scatter Fusion, 2022. URL https://github. com/NVIDIA/cutlass/tree/master/examples/

36 gather scatter fusion. Łast accessed July 25, 2022.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for lan-guage understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Drozdov, A., Verga, P., Yadav, M., Iyyer, M., and McCal-lum, A. Unsupervised latent tree induction with deep inside-outside recursive autoencoders. In *North Ameri-can Association for Computational Linguistics*, 2019.

Durvasula, S., Zhao, A., Kiguru, R., Guan, Y., Chen, Z., and Vijaykumar, N. Acs: Concurrent kernel execution on irregular, input-dependent computational graphs, 2024.

Dyer, C., Ballesteros, M., Ling, W., Matthews, A., and Smith, N. A. Transition-based dependency parsing with stack long short-term memory. *CoRR*, abs/1505.08075, 2015. URL http://arxiv.org/abs/1505.08075.

Elbayad, M., Gu, J., Grave, E., and Auli, M. Depth-adaptive transformer. *CoRR*, abs/1910.10073, 2019. URL http: //arxiv.org/abs/1910.10073.

Fedus, W., Zoph, B., and Shazeer, N. Switch transform-ers: Scaling to trillion parameter models with simple and efficient sparsity. *CoRR*, abs/2101.03961, 2021. URL https://arxiv.org/abs/2101.03961.

Fegade, P., Chen, T., Gibbons, P., and Mowry, T. Cortex: A compiler for recursive deep learning models. In Smola, A., Dimakis, A., and Stoica, I. (eds.), *Proceedings of Machine Learning and Systems*, volume 3, pp. 38–54, 2021. URL https://proceedings.mlsys.org/paper/2021/file/182be0c5cdcd5072bb1864cdee4d3d6e-Paper.pdf.

Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7): 478–490, 1981. doi: 10.1109/TC.1981.1675827.

Frostig, R., Johnson, M., and Leary, C. Compiling machine learning programs via high-level tracing. 2018. URL https://mlsys.org/Conferences/doc/2018/146.pdf.

Gao, P., Yu, L., Wu, Y., and Li, J. Low latency rnn inference with cellular batching. In *Proceed-ings of the Thirteenth EuroSys Conference*, EuroSys ’18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450355841. doi: 10.1145/3190508.3190541. URL https://doi.org/10.1145/3190508.3190541.

Gibbons, P. B. and Muchnick, S. S. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pp. 11–16, 1986.

Girshick, R. B. Fast R-CNN. *CoRR*, abs/1504.08083, 2015. URL http://arxiv.org/abs/1504.08083.

Girshick, R. B., Donahue, J., Darrell, T., and Malik, J. Rich feature hierarchies for accurate object detection and se-mantic segmentation. *CoRR*, abs/1311.2524, 2013. URL http://arxiv.org/abs/1311.2524.

Gupta, R., Mehofer, E., and Zhang, Y. Profile guided com-piler optimizations, 2002.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Novem-ber 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

Hogen, G., Kindler, A., and Loogen, R. Automatic parallelization of lazy functional programs. In *Symposium Proceedings on 4th European Symposium on Program-ming*, ESOP ’92, pp. 254–268, Berlin, Heidelberg, 1992. Springer-Verlag. ISBN 0387552537.

Huang, M., Renau, J., and Torrellas, J. Profile-based energy reduction in high-performance processors. In *4th Work-shop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, 2001.

Huang, W., Dong, Y., Milanova, A., and Dolby, J. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 106–117, 2015.

Intel. Intel oneAPI Deep Neural Network Library, 2022. URL https://www.intel.com/content/www/us/en/developer/tools/oneapi/

onednn.html#gs.4je6v8. Last accessed July 1, 2022.

Jeong, E., Cho, S., Yu, G.-I., Jeong, J. S., Shin, D.-J., and Chun, B.-G. JANUS: Fast and flexible deep learning via symbolic graph execution of imperative programs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pp. 453–468, Boston,

MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/jeong.

Kaya, Y. and Dumitras, T. How to stop off-the-shelf deep neural networks from overthinking. *CoRR*, abs/1810.07052, 2018. URL http://arxiv.org/abs/1810.07052.

Kim, T., Jeong, E., Kim, G.-W., Koo, Y., Kim, S., Yu, G., and Chun, B.-G. Terra: Imperative-symbolic co-execution of imperative deep learning programs. In Ranzato, M., Beygelzimer, A., Dauphin, Y., Liang, P., and Vaughan, J. W. (eds.), *Advances in Neural Information Processing Systems*, volume 34, pp. 1468–1480. Curran Associates, Inc., 2021. URL https://proceedings.neurips.cc/paper/2021/file/

0b32f1a9efe5edf3dd2f38b0c0052bfe-

Paper.pdf.

Koehn, P. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*, pp. 115–124. Springer, 2004.

Lattner, C., Amini, M., Bondhugula, U., Cohen, A., Davis, A., Pienaar, J., Riddle, R., Shpeisman, T., Vasilache, N., and Zinenko, O. MLIR: A compiler infrastruc-ture for the end of Moore's law, 2020. URL https: //arxiv.org/abs/2002.11054.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding, 2023.

Looks, M., Herreshoff, M., Hutchins, D., and Norvig, P. Deep learning with dynamic computation graphs. *CoRR*, abs/1702.02181, 2017. URL http://arxiv.org/abs/1702.02181.

Ma, J., Zhao, Z., Yi, X., Chen, J., Hong, L., and Chi, E. H. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 1930–1939, 2018.

McCool, M., Robison, A. D., and Reinders, J. Chapter 8 - fork–join. In McCool, M., Robison, A. D., and Reinders, J. (eds.), *Structured Parallel Programming*, pp. 209–251. Morgan Kaufmann, Boston, 2012. ISBN 978-0-12-415993-8. doi: https: //doi.org/10.1016/B978-0-12-415993-8.00008-6. URL https://www. sciencedirect. com/science/article/pii/B9780124159938000086.

Neubig, G., Dyer, C., Goldberg, Y., Matthews, A., Ammar, W., Anastasopoulos, A., Ballesteros, M., Chiang,

D., Clothiaux, D., Cohn, T., Duh, K., Faruqui, M., Gan, C., Garrette, D., Ji, Y., Kong, L., Kuncoro, A., Kumar, G., Malaviya, C., Michel, P., Oda, Y., Richardson, M., Saphra, N., Swayamdipta, S., and Yin, P. Dynet: The dynamic neural network toolkit, 2017a.

Neubig, G., Goldberg, Y., and Dyer, C. On-the-fly operation batching in dynamic computation graphs, 2017b.

Niu, W., Agrawal, G., and Ren, B. Sod2: Statically optimizing dynamic deep neural network. *arXiv preprint arXiv:2403.00176*, 2024.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alch´e-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL http://papers. neurips. cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Ponomarev, D., Kucuk, G., and Ghose, K. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proceedings. 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pp. 90–101, 2001. doi: 10.1109/MICRO.2001.991108.

PyTorch. TorchScript, 2020. URL https:// pytorch.org/docs/stable/jit.html. Last ac-cessed Sept 09, 2021.

PyTorch Community. Github issue number 42487: Support recursive data type in TorchScript, 2020. URL https://github.com/pytorch/pytorch/issues/42487. Last accessed July 25, 2022.

Qiao, Y. and Taura, K. An automatic operation batching strategy for the backward propagation of neural networks having dynamic computation graphs, 2019. URL https: //openreview.net/forum?id=SkxXwo0qYm.

Radul, A., Patton, B., Maclaurin, D., Hoffman, M., and A. Saurous, R. Automatically batching control-intensive programs for modern accelerators. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 390–399. 2020. URL https://proceedings.mlsys.org/paper/2020/file/140f6969d5213fd0ece03148e62e461e-Paper.pdf.

Roesch, J., Lyubomirsky, S., Kirisame, M., Weber, L., Pollock, J., Vega, L., Jiang, Z., Chen, T., Moreau, T., and Tatlock, Z. Relay: A high-level compiler for deep learning, 2019. URL https://arxiv.org/abs/1904.08368.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learn-ing representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

Schuster, M. and Paliwal, K. K. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q. V., Hinton, G. E., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *CoRR*, abs/1701.06538, 2017. URL http://arxiv.org/abs/1701.06538.

Sherwood, T., Sair, S., and Calder, B. Phase tracking and prediction. *ACM SIGARCH Computer Architecture News*, 31(2):336–349, 2003.

Shuai, B., Zuo, Z., Wang, G., and Wang, B. Dag-recurrent neural networks for scene labeling. *CoRR*, abs/1509.00552, 2015. URL http://arxiv.org/abs/1509.00552.

Silfa, F., Arnau, J., and Gonz´alez, A. E-BATCH: energy-efficient and high-throughput RNN batching. *CoRR*, abs/2009.10656, 2020. URL https://arxiv.org/abs/2009.10656.

Smith, J. Dynamic instruction scheduling and the astronautics zs-1. *Computer*, 22(7):21–35, 1989. doi: 10.1109/2.30730.

Socher, R., Huval, B., Manning, C. D., and Ng, A. Y. Seman-tic Compositionality Through Recursive Matrix-Vector Spaces. In *Proceedings of the 2012 Conference on Empir-ical Methods in Natural Language Processing (EMNLP)*, 2012.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., and Potts, C. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 1631– 1642, Seattle, Washington, USA, October 2013a. As-sociation for Computational Linguistics. URL https://www.aclweb.org/anthology/D13-1170.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. Recursive deep models for semantic composition ality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pp. 1631–1642, 2013b.

Suhan, A., Libenzi, D., Zhang, A., Schuh, P., Saeta, B., Sohn, J. Y., and Shabalin, D. Lazytensor: combining eager execution with domain-specific compilers. *CoRR*, abs/2102.13267, 2021. URL https://arxiv.org/abs/2102.13267.

Teerapittayanon, S., McDanel, B., and Kung, H. T. Branchynet: Fast inference via early exiting from deep neural networks. *CoRR*, abs/1709.01686, 2017. URL http://arxiv.org/abs/1709.01686.

Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weis-man, O. Taj: Effective taint analysis of web applica-tions. *SIGPLAN Not.*, 44(6):87–97, jun 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542486. URL https: //doi.org/10.1145/1543135.1542486.

Wiseman, S. and Rush, A. M. Sequence-to-sequence learning as beam-search optimiza-tion. *CoRR*, abs/1606.02960, 2016. URL http://arxiv.org/abs/1606.02960.

Xin, J., Tang, R., Lee, J., Yu, Y., and Lin, J. Dee-bert: Dynamic early exiting for accelerating BERT inference. *CoRR*, abs/2004.12993, 2020. URL https: //arxiv.org/abs/2004.12993.

Xin, J., Tang, R., Yu, Y., and Lin, J. Berxit: Early exiting for bert with better fine-tuning and extension to regression. In *Proceedings of the 16th conference of the European chapter of the association for computational linguistics: Main Volume*, pp. 91–104, 2021.

Xu, S., Zhang, H., Neubig, G., Dai, W., Kim, J. K., Deng, Z., Ho, Q., Yang, G., and Xing, E. P. Cavs: An efficient runtime system for dynamic neural net-works. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pp. 937–950, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www. usenix. org/conference/atc18/presentation/xu-shizen.

You, J., Ying, R., Ren, X., Hamilton, W. L., and Leskovec, J. Graphrnn: A deep generative model for graphs. *CoRR*, abs/1802.08773, 2018. URL http://arxiv.org/abs/1802.08773.

Zha, S., Jiang, Z., Lin, H., and Zhang, Z. Just-in-time dynamic-batching. *CoRR*, abs/1904.07421, 2019. URL http://arxiv.org/abs/1904.07421.

Zhai, Y., Jiang, C., Wang, L., Jia, X., Zhang, S., Chen, Z., Liu, X., and Zhu, Y. Bytetransformer: A high-performance transformer boosted for variable-length in-puts, 2023.

Zheng, B., Jiang, Z., Yu, C. H., Shen, H., Fromm, J., Liu, Y., Wang, Y., Ceze, L., Chen, T., and Pekhimenko, G. Dietcode: Automatic optimization for dynamic tensor programs. In Marculescu, D., Chi, Y., and Wu, C. (eds.), *Proceedings of Machine Learning and Systems*, volume 4, pp. 848–863, 2022. URL https:// proceedings.mlsys.org/paper/ 2022/file/ fa7cdfad1a5aaf8370ebeda47a1ff1c3-Paper.pdf.

Zheng, B., Yu, C. H., Wang, J., Ding, Y., Liu, Y., Wang, Y., and Pekhimenko, G. Grape: Practical and efficient graphed execution for dynamic deep neural networks on gpus. In *Proceedings of the 56th Annual IEEE/ACM Inter-national Symposium on Microarchitecture*, MICRO ’23, pp. 1364–1380, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400703294. doi: 10.1145/3613424.3614248. URL https://doi.org/10.1145/3613424.3614248.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., Gonzalez, J. E., and Stoica, I. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/zheng.

更多关于混合静态+动态优化的细节

A.1 运营商提升

给定一个递归计算，例如清单1中的@rnn函数，通常某些张量运算符不是递归诱导的顺序依赖的一部分。例如，可以将清单1中第5行输入的线性变换提升到递归之外。ACROBAT不再像过去那样依赖于运行时调度算法来识别这些操作符，而是静态地发现可以提升的操作符。我们通过依赖1-context敏感的污点分析来静态计算此类操作符的深度来实现这一点。我们看到，在清单2中，如何将第5行的核偏置密集调用赋值为静态计算深度0。在运行时，这样的运算符因此被有效地提升出递归。对于RNN示例，这允许我们将所有输入词嵌入的线性变换批量处理在一起，而不是一次执行一个。

A.2 粒度粗化

通常，调度是在单个张量算子的粒度上执行的，即DFG中的每个节点对应一个张量内核调用。我们在§2.1中看到，DL计算如何经常包含嵌入在动态控制流中的较大静态子图。因此，AC-ROBAT在较粗粒度的静态子图上执行调度，从而减少了调度开销。由于这些块不包含任何控制流，以这种方式粗化粒度不会导致被利用的并行性的损失。这种优化在过去的工作中也被探索过(Zha等人，2019;Xu et al.， 2018;Fegade等，2021年;高等，2018;Silfa et al.， 2020)，如图8所示。

A.3 对抗深度调度的急迫性

我们在§4.1中看到了ACROBAT是如何依靠虚操作和程序阶段来对抗深度调度的渴望的。下面，我们提供了同样的更详细的解释。

*虚操作符:*在图3的上方窗格中，我们看到，在存在条件语句的情况下，提前批处理会导致次优的批处理计划。具体来说，输入Inp1和Inp2的运算符B的实例被急切地批处理，更重要的是，与输入Inp3和Inp4的运算符B的实例是分开的。在下方的窗格中，我们插入了对幽灵操作符的调用，从而导致最优的时间表。ACROBAT静态地识别这些情况，并根据需要插入幽灵操作符。请注意，幽灵操作符仅影响调度，在内核执行期间被忽略。

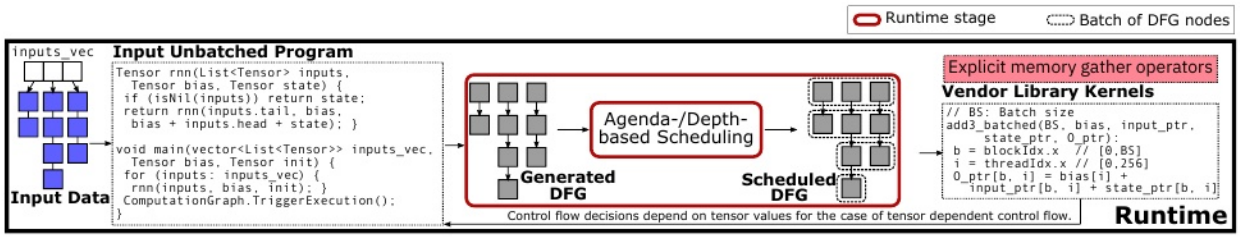

图7。DyNet运行时管道概述。请注意，DyNet缺乏任何静态或编译时分析，以及如何依赖显式内存收集操作，导致高数据移动成本，如我们在§7中所示。

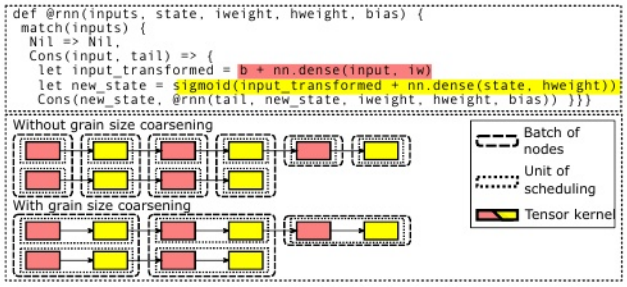

图8。清单1中@rnn函数的粒度粗化。

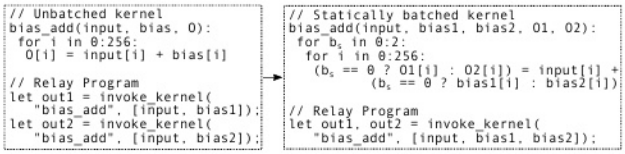

图9。水平融合促进参数重用。

程序阶段:对于清单1中的RNN示例，为了利用第19行输出操作符的最大并行性，应该等到@rnn函数中调用的所有操作符都为所有输入实例执行完毕。这样，所有输入实例中所有单词对应的所有输出操作符都可以作为一个批处理的内核调用执行。这将要求所有这些输出操作符被分配相同的深度。然而，情况可能并非如此，因为每个输入句子的长度可能会有所不同。在语义上，我们可以将RNN计算分为两个语义阶段——初始递归计算和随后的输出转换。给定这样的程序阶段，ACROBAT在进入下一个阶段之前，先在一个阶段调度和执行操作符。这样，在转到输出操作符之前，acroo - bat确保对所有输入实例执行所有RNN函数。

关于ACROBAT张量核生成的更多细节

B.1 利用数据重用

为了更好的数据重用而进行的代码复制:输入程序中的代码重用通常会禁止上面提到的参数重用。考虑以下代码清单，其中，与清单1中实现的RNN模型类似，我们实现了一个双向RNN (BiRNN) (Schuster & Paliwal, 1997)计算。在这里，我们用不同的模型参数调用相同的@rnn函数来实现前向和后向rnn。在这种情况下，@rnn函数调用的张量算子将不会被静态地确定为具有任何跨参数常量

多次调用，从而排除了模型参数的数据重用。为了解决这个问题，在生成批处理内核之前，ACROBAT识别出这种数据重用情况(同样使用上下文敏感的污染分析)，并传递地复制必要的函数，以便在以后生成批处理内核时启用数据重用[11]。例如，在BiRNN示例中，ACROBAT将传递复制@rnn函数(包括它调用的张量运算符)，并在下面的清单中对每个向前和向后调用使用@rnn函数的不同副本。

```
1  (* Type annotations are omitted in the listing for simplicity. *)
2  def @main(f_rnn_bias, f_rnn_i_wt, f_rnn_h_wt, f_rnn_init,
3            b_rnn_bias, b_rnn_i_wt, b_rnn_h_wt, b_rnn_init,
4            inps_list) {
5    let rinps_list = @reverse_list(inps_list);
6    let forward_res = @rnn(inps_list, f_rnn_init,
7                           f_rnn_bias, f_rnn_i_wt, f_rnn_h_wt);
8    let backward_res = @rnn(rinps_list, b_rnn_init,
9                            b_rnn_bias, b_rnn_i_wt, b_rnn_h_wt);
10 }
```

在静态块内重用:给定一个张量运算符，上面讨论的分析考虑了mini-batch中不同输入实例所做的调用之间共享的参数。这通常适用于模型参数，因为它们是跨多个输入实例共享的。然而，通常情况下，同一个静态块内对同一个张量运算符的多个调用共享一个参数。例如，在常用的LSTM单元中就是这种情况，其中四个门的计算都涉及同一个输入向量的并发线性变换。在这种情况下，ACROBAT水平地融合这些调用，以便利用参数数据重用。这在图9中进行了说明。

[11]Simply inlining the @rnn function will not work here as it is a recursive function.

C更多实现细节

C.1 张量核优化

下面，我们将讨论ACROBAT如何依赖于TVM的自动调度程序(Zheng等人，2020)来自动生成输入程序中使用的(可能融合的)张量算子的批处理版本的优化实现。

自动调度器操作符优先级:给定一个由多个张量操作符组成的DL计算，自动调度器根据张量操作符的相对估计执行成本来优先优化张量操作符。在其他因素中，这个估计的成本与输入程序执行期间调用操作符的次数成正比。为了在存在控制流(如重复或条件控制流)的情况下准确估计给定操作员的执行频率，AC-ROBAT依赖于轮廓引导优化(PGO)。当不可能实现PGO时，ACROBAT还提供了一个简单的静态分析，根据递归中操作符调用的嵌套深度，启发式地执行此估计。

处理可变循环范围:由于ACROBAT调度的动态特性，在生成的未优化的批处理内核中，批处理维度对应的循环具有可变范围(例如图1中的内核3)。为了优化这些内核，acrobat为批处理维度自动调度了一个对应的具有静态循环扩展的内核，并自动将生成的调度应用于具有可变扩展的原始内核。更进一步，当为具有可变区段的循环生成代码时，我们经常需要插入条件检查，以避免越界访问。我们依靠DietCode (Zheng等人，2022)中提出的局部填充和局部分区技术来消除这些条件检查，因为它们可能严重损害性能

C.2 提前编译

我们在§6中看到，ACROBAT将输入Relay计算提前编译为c++。作为编译的一部分，ACROBAT将所有动态控制流以及不规则数据结构降低到本地c++控制流和类。Relay通过将标量建模为零维张量来处理标量。ACROBAT的AOT编译器也将这种零维张量及其上的普通算术运算符降低为原生c++标量。我们在§7.2中看到，这种AOT编译显著地减少了动态控制流的执行开销。

C.3 其他细节

正如正文第6节所讨论的，我们通过扩展TVM来原型化AC-ROBAT。我们发现TVM的算子融合通有限，往往无法融合张量重塑、拼接、转置等内存复制算子。因此，在我们对DL计算的实现中，我们手动向编译器提供融合提示，以迫使此类算子与它们的消费者进行融合。此外，我们目前的原型只支持Relay的功能子集。具体来说，目前不支持通过可变引用产生的副作用。ACROBAT的运行时系统进行了大量优化，以减少运行时开销。我们使用竞技场分配(在CPU和GPU上)和GPU上的异步执行。我们还批处理CPU和GPU之间的内存传输操作，尽可能减少CUDA API开销。

表9所示。NestedRNN(小，批处理大小为8)执行时间(不使用/使用PGO)，说明在自动调度期间使用PGO调用频率的好处。

Auto-scheduler iter。	100	250	500	750	1000
执行时间(ms)	41.08/42.49	34.58/30.88	31.61/24.4	27.33/23.72	25.63/24.34

D补充评估和额外的细节

D.1 PGO在张量核自动调度中的优势

我们在§C中提到过。1 ACROBAT使用调用频率(通过PGO获得)在自动调度过程中优先考虑张量算子优化。为了评估这种优化的好处，我们看看NestedRNN在优化和没有优化的情况下的性能。这个基准计算平均执行内部RNN循环的30次迭代，每次迭代外部GRU循环。因此，在RNN循环中调用的运算符比在GRU循环中调用的运算符对基准性能的影响要大得多。表9显示了使用PGO和不使用PGO的基准测试对不同迭代auto-scheduler[12]的执行时间，显示了AC-ROBAT如何在启用PGO的情况下更好地为RNN算子的自动调度确定优先级。

[12]Due to the inherent randomness in the auto-scheduling process, the given execution times are averaged over 10 runs of the auto-scheduler each.