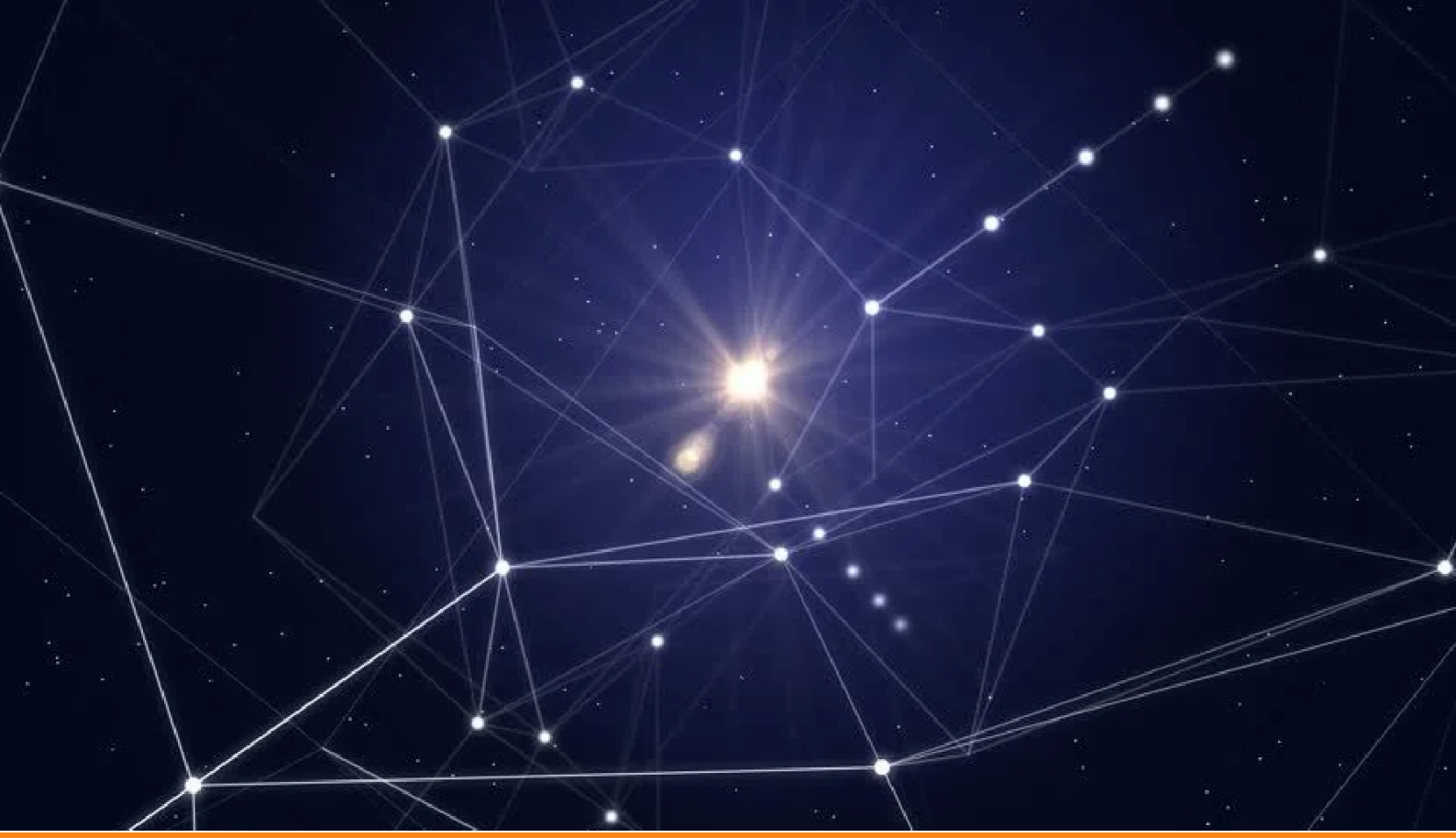


TensorQ 说明文档

一个基于张量网络的大规模量子线路模拟器

作者：赵先和

组织：中国科学技术大学

时间：6 月 1 日, 2023 年

版本：0.1.0

目录

第 1 章 背景介绍

1.1 张量网络算法

张量网络算法是如今在经典计算上模拟量子计算最有效的量子算法之一，许多用于展示量子优越性的专用量子计算机都被张量网络算法攻破。这些量子计算机被用来宣称，解决了某些在经典计算机上不可能完成的特定问题，但后来被经典计算机采用张量网络的方法成功模拟，例如 2019 年，Google 宣称用‘悬铃木’超导量子计算机，以 0.002 的保真度解决了随机量子线路采样问题，而在经典计算机需要 10000 年 [1]。但随后 2021 年，刘勇等人在超级计算机‘太湖之光’上，用张量网络的方法，用 304 秒模拟了‘悬铃木’量子计算机 [11]。在 2022 年，潘峰等人用张量网络算法，在 512 块 GPU 上，用 15 个小时，以保真度 0.0037 完成了同样的问题 [13]。类似地，2020 年中科大用’ 九章’ 光量子计算机计算机，完成了高斯玻色采样问题，宣称这个问题经典计算机很难解决 [17]。2023 年，Changhun Oh 等人宣称，在 144 张 GPU 上完成‘九章’高斯玻色采样实验的模拟 [12]。

可见经典计算上的张量网络算法成为了量子计算机强有力的竞争对手，很多之前被认为是经典计算机难以解决的问题，后来都被张量网络算法解决了。因此发展张量网络算法有两个主要的意义：首先，它可以作为量子优越性的检验器，我们通常需要在经典计算机上找到一个好的算法，来确定量子计算机是否真的无法被经典计算机模拟，而基于张量网络的算法很可能就是复杂度最低的算法。另外，量子计算机的实验结果与量子算法有时候也需要经典模拟进行验证与尝试，而张量网络算法就可以作为模拟量子计算机的一种工具，促进量子计算机和量子算法的发展。

因此，我们设计并开发了 TensorQ 这套基于张量网络的量子计算机模拟工具包，我们将持续跟踪张量网络算法的前沿进展，不断地对这个工具包进行增强与完善。希望广大学者能够通过这个工具包，了解并利用张量网络算法，推动量子计算领域的发展。我们技术有限，不足之处望大家多多包涵，多多指教，欢迎和我们联系与讨论。

1.2 TensorQ 介绍

TensorQ 是一个基于张量网络的量子线路模拟工具包，TensorQ 主要的功能是计算量子线路的振幅，可以选择单振幅、多振幅以及全振幅，可以计算精确振幅也可以计算近似振幅。TensorQ 兼容其他量子线路模拟工具包，例如 Cirq 等，TensorQ 本身并不能生成线路，而是用其他工具包生成的线路，TensorQ 能够将量子线路转化成张量网络。另外 TensorQ 工具包提供低计算复杂度、低空间复杂度的张量网络缩并方案，以及相应的缩并运算得到最后的振幅。

TensorQ 在一定程度上解决了指数墙问题，可以在有限内存的计算资源上模拟比特数目更多的量子线路，例如，Cirq 模拟 35 比特 GHZ 态的量子线路需要 256GiB，而采用 TensorQ 进行模拟所需要的内存远小于 1GiB，具体可以参考4.1小节。

我们的联系方式如下：

- GitHub 地址：https://github.com/Xenon-zhao/TensorQ
- 用户 QQ 群：待建设（建议加群）
- 邮件：648124022@qq.com

1.2.1 安装与使用

在 GitHub 仓库下载 TensorQ 压缩文件’TensorQ-main.zip’，解压压缩文件后。用 win+r 输入 cmd，进入命令行，进入’TensorQ-main’ 所在的地址

```
cd <Yourpath>/TensorQ-main
```

其中，<Yourpath>根据你的文件地址修改。然后用下面的命令行进行安装

```
python setup.py install
```

第 2 章 基础原理

这节将介绍 TensorQ 用到的主要技术，将首先介绍量子模拟中遇到的指数墙问题，然后会介绍张量网络的基本概念，以及 TensorQ 如何解决指数墙问题。

2.1 量子模拟指数墙问题

量子计算系统的希尔伯特空间维度 M 随着量子比特的数量 N 指数增加，$M = 2^N$。在经典计算上模拟大规律量子线路，按照 Cirq、MindQuantum 以及 pyQPanda 等工具包的方法，需要存储 M 维的态矢量，这导致这种模拟方法对计算机内存的需求量通常随着比特个数 N 指数增加，经典计算机所能够模拟的量子线路比特数非常有限，这极大地限制了科研工作者对量子计算和量子算法的研究。

表 2.1: 指数墙问题

qubit 数量	空间维度	空间复杂度 (complex128)	存储设备
10	2^{10}	16 K bytes	
20	2^{20}	16 M bytes	
30	2^{30}	16 G bytes	笔记本
40	2^{40}	16 T bytes	计算集群
50	2^{50}	16 P bytes	超算
53	2^{53}	128 P bytes	所有超算的硬盘

2.2 张量网络基础

这一节将介绍张量网络的基础概念，包括张量是什么？张量的图像表示以及张量的基本运算。

2.2.1 张量的概念

张量的定义如下 [15]：

- N 阶张量又称 **N 维数列**
- 张量阶数即为张量指标的个数
- 向量、矩阵也称为 1 阶张量、2 阶张量
- 每个指标可取的值的个数，被称为指标的维数
- 张量的“直观”定义：多个指标标记下的一堆数

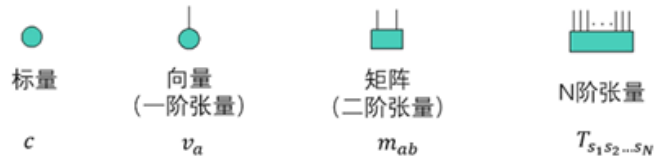
比如张量 $A[n_0][n_1]\cdots[n_{k-1}]$，其指标的个数为 k，称其为 k 阶张量，若指标可取的值的个数为 $n_i = N(0ik-1, N > 0)$，称指标的维数为 N。

一个 qubit 是一个向量，即一阶张量；一个单比特门是一个矩阵，即二阶张量；一个双比特门可以看成一个指标为 4 维的二阶张量也可以看成一个指标维度为 2 的四阶张量，因此量子线路中的所有元素都可以用张量表示。量子线路中的量子比特以及量子纠缠也是天然的可以使用张量网络来描述的 [3].

2.2.2 张量的图像表示

张量用连接着个腿的圆圈或方块表示，为张量的阶数，如下图所示。

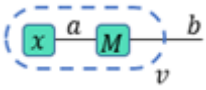
每个腿对应指标，比如一阶张量的腿对应指标 a，二阶张量的腿对应指标 a 和 b。



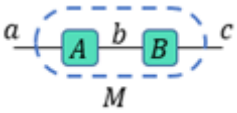
2.2.3 张量的基本运算

1. 向量内积：$c = \sum_a x_a y_a$;

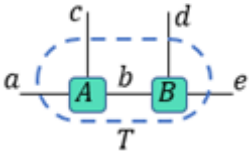
2. 向量乘矩阵：$v_b = \sum_a x_a M_{ab} = xM$;



3. 矩阵乘矩阵：$M_{ac} = \sum_b A_{ab} B_{bc} = AB$;



4. 张量缩并：$T_{acde} = \sum_b A_{acb} B_{bde}$;



2.3 TensorQ 核心技术

这一节将介绍 TensorQ 中的核心技术，包括搜索缩并顺序、多振幅复用以及张量网络切片分解。

2.3.1 优化缩并顺序

上一节已经说明了，量子线路都可以转化成张量网络，而通过张量计算振幅就是要把网络缩并，而将一个张量网络缩并通常有很多种不同的缩并顺序，往往不同的缩并顺序对应着不同的复杂度 [6]，接下来我们通过一个简单的网络缩并来说明这件事。

我们考虑一个如图2.1(a) 所示的张量网络，一共有 5 个张量，用 5 个黑色节点表示。每个节所连接的边表示该张量的指标，两个节点之间相连的边表示两个张量的共有指标，也就是缩并指标。

对于图2.1(a) 所示的张量网络，我们给出两种缩并顺序，把缩并顺序画出树状图的结构，分别如图2.1(b)和2.1(c) 所示，这种图称为缩并树。缩并树中每一个节点表示一个张量，并且用一组字母表示该张量的指标；每一条边表示张量的缩并路径。五个节点按照缩并树从下到上的方向，两两相遇合并为一个新的节点，代表张量两两缩并产生中间张量，并最终缩并为一个标量。

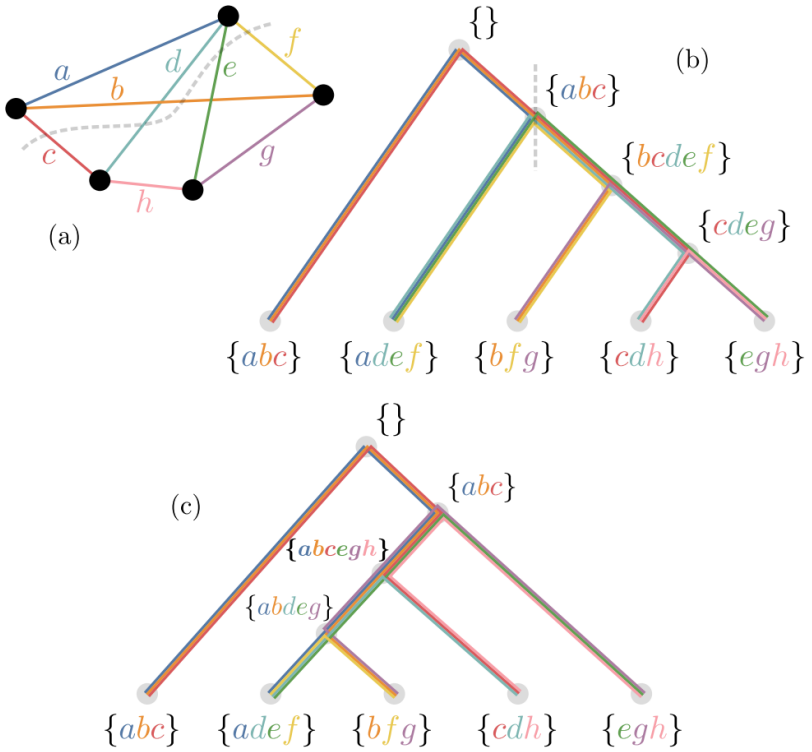


图 2.1: 对于 (a) 所示的图，两个可能的收缩树 (b) 和 (c)，展示了中间张量和缩并。树中的每条边都有一个相对应的张量与子图。张量的大小与指标的数量呈指数关系，沿着边缩并的指标，用独特的颜色表示。树中的每个顶点代表两个张量的成对收缩，以及子图的二分割 (虚线显示了一个例子)。这种两两收缩的计算复杂度通过顶点的指标数量指数怎加。假设每个索引的大小相同，树 (c) 因此具有比树 (b) 更高的最大缩并宽度 (粗体) 和总收缩成本 [6]。

接下来，我们计算两种缩并顺序的复杂度。复杂度分为空间复杂度与时间复杂度，空间复杂度就是将整个张量网络缩并所需要的存储空间，这由整个缩并过程中最大的中间张量的大小决定；时间复杂度就是整个张量网络缩并过程中的浮点运行次数，浮点运行包括加减乘除运算，由于乘法运算是缩并的主要运算开销，一般只计算乘法运算次数。我们假设图2.1(a) 中的每个指标维度相同都为 n，因此缩并顺序 (b) 的最大张量具有 5 个指标 bcdef，空间复杂度为 n^5；缩并顺序 (c) 的最大张量有 6 个指标 abcegh，空间复杂度为 n^6。在时间复杂度计算上，每次缩并的乘法运算的次数与独立指标个数成指数关系，缩并顺序 (b) 一个有 4 次缩并，乘法次数依次为 n^5, n^6, n^6, n^3；缩并顺序 (c) 也有 4 次缩并，乘法次数依次为 n^6, n^7, n^6, n^3。可见，缩并顺序 (b) 的空间复杂度和时间复杂度都更小。

对应一个任意的张量网络，寻找复杂度最低的缩并方案至少是一个#P-hard的问题 [4, 16]。TensorQ 采用 arTensor 工具包进行缩并顺序搜索，arTensor 中采用的是随机算法来进行搜索，通常搜索时间越长得到的缩并顺序复杂度越低，arTensor 可以给出缩并顺序相应的复杂度，根据计算资源的能力得到满足需求的缩并顺序即可。

Cirq、MindQuantum 等现有工具包模拟量子线路的方法称为薛定谔算法，这种方法存储并计算完整的态矢量 [14]。薛定谔算法的缩并顺序如图2.2所示，这种算法可以看作是一种特殊的缩并顺序，对于单振幅或者少振幅的情况，这种顺序显然不是张量网络中复杂度较低的顺序。图2.1中的例子可以启发我们发现，通常先将指标个数较少，共有指标较多的张量先缩并能够产生指标个数较少的中间张量，这样更有利于减少缩并的复杂度。将量子线路转化成张量网络后，搜索低复杂度的缩并顺序，这是 TensorQ 优化大规模量子线路模拟的第一步。

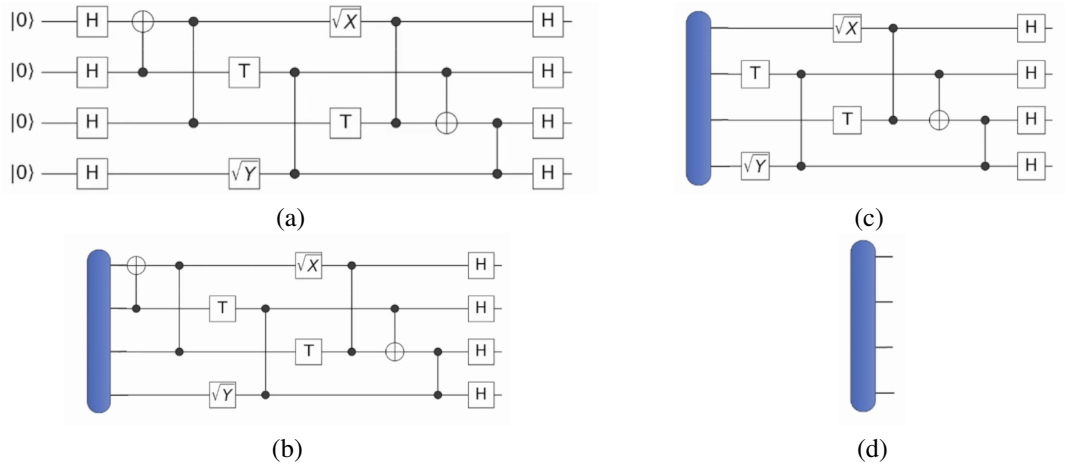


图 2.2: 薛定谔算法将量子线路按照 (a), (b), (c), (d) 的顺序进行缩并。

2.3.2 多振幅复用

对量子线路的模拟通常分为三种，单振幅模拟，多振幅模拟，全振幅模拟，分别表示模拟结果输出末态单个，多个，全部振幅。对于单振幅模拟，张量网络方法比薛定谔算法复杂度低很多，因为线路两端是相互独立的 2N 个二维向量，N 是比特数量，都是一些非常小的一阶张量，一阶张量与其他张量缩并后只会将另外一个张量的阶数也降低，因此张量网络方法很容易找到复杂度比薛定谔算法低的缩并方案。对于全振幅模拟，张量网络方法相比薛定谔算法就没有优势了，因为 N 个比特总共有 2^N 个振幅，当 N 很大时这么多振幅是能难存储的，全振幅模拟的指数墙困难没有办法解决。

不过，在大规模量子线路的研究中，通常关注的都只是一部分振幅，即对于 2^N 个振幅，真正需要计算的只有 m 个振幅，通常 $m << 2^N$，这种情况张量网络方法是可以模拟的。在多振幅的模拟中，又有两种情况，关联多振幅模拟与无关联多振幅模拟，关联多振幅是指，多个振幅对于的比特串的构形是相互关联的，例如 5 个 qubit 中，固定第一、二个 qubit 为 0，那么一共有 8 种比特串，分别是'00000', '00001', '00010', '00011', '00100', '00101', '00110', '00111'。这种情况下，第一、二个比特的末态仍然是二维向量，只有一个指标，而另外三个比特的模拟由于会出现两种情况，因此阶数增加，变成二阶张量。而无关联多振幅的情况更加复杂，它是指多个振幅的比特串构形是任意的，相互之间没有关联，比特任意写成 8 个比特，可以是'01000', '10001', '11010', '00011', '00100', '11101', '10110', '01111'，可以发现这 8 个比特串中，每个 qubit 都既出现了'0' 也出现了'1'。

全振幅、单振幅、关联多振幅、无关联多振幅的模拟，通过下面的图2.3说明了。

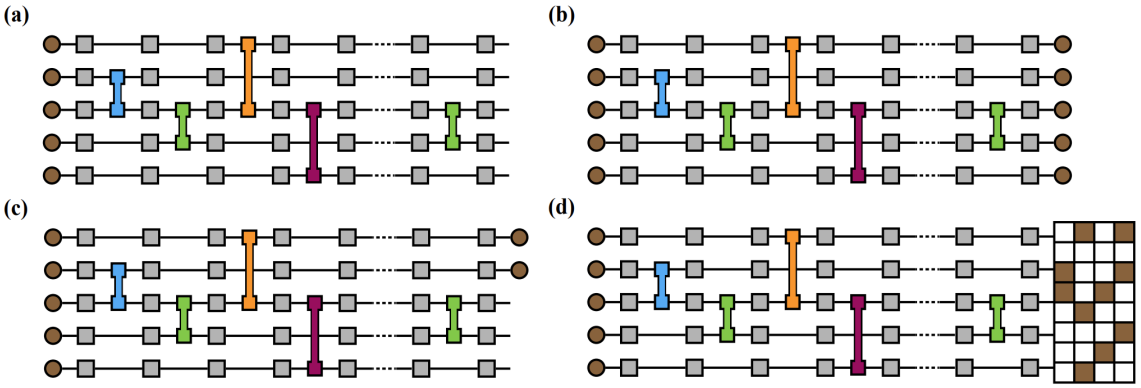


图 2.3: 四种不同的量子线路模拟类型，分别是 (a) 全振幅模拟，(b) 单振幅模拟，(c) 关联多振幅模拟，(d) 无关联多振幅模拟 [13]

无关联多振幅的情况比关联多振幅的情况更加复杂，而且也更加重要，因为关联多振幅只计算了希尔伯特空间中，部分相互关联的振幅，而无关联多振幅才是希尔伯特空间中随机的振幅，更能代表希尔伯特全空间的

性质。但是，无光联振幅中，每个比特都出现了两种情况，是不是要把所有比特的末态都变成二阶张量呢？这是不可行的，如果所有末态比特都变成二阶张量，将会使这个模拟与全振幅模拟一模一样，所以关联多振幅的模拟变成了一个非常重要且有具有挑战的问题。

另外一个缩并顺序搜索的软件包 cotengra[6] 也提供了张量网络的缩并顺序搜索功能，不过它只针对单振幅和全振幅的缩并，并没有优化无关联多振幅的缩并。TensorQ 的缩并模块采用 arTensor 软件包，针对无关联多振幅模拟的情况，采用了多振幅复用的算法，又称为稀疏态模拟算法（sparse-state simulation）[13]，来减低复杂度。多振幅复用算法的原理是，根据需要计算振幅的目标比特串，动态地调整末态比特的指标维度。

如图2.4所示，考虑在 3 比特的线路中计算’111’, ’010’, ’000’ 三个比特串的振幅，在图2.4(a) 中，张量网络初始状态下，三个末态比特都是二阶张量，但是当二、三号比特缩并时，二、三比特共同末态的张量不会计算和保留全部的维度，而是根据二、三号比特的目标比特串有选择地计算和保留部分分量，在这个例子中，二、三比特的目标比特串只需要’11’, ’10’, ’00’ 三个分量的值，而不需要’01’ 分量的值，因此将这个缩并过程中的复杂度降低为多振幅复用前的 3/4。类似的，当下一步缩并，三个比特全部缩并在一起时，也只选择了’111’, ’010’, ’000’ 分量进行计算与存储，这将复杂度降低为了多振幅复用前的 3/8。

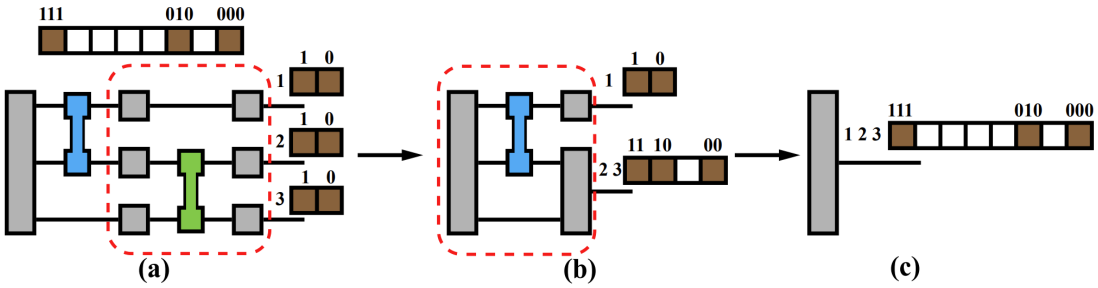


图 2.4: 无关联多振幅复用算法示意图。[13]

因此在一般的无关联多振幅张量网络缩并中，多振幅复用算法将整体的复杂度大大降低，特别是，避免了像全振幅模拟一样需要保存所有的态矢量分量，解决了多振幅模拟中的指数墙问题。

2.3.3 张量网络切片分解

在大规模量子线路模拟的过程中，当比特数目很多，需要计算的振幅数量很多的时候，通过优化缩并顺序以及多振幅复用两种优化算法还是不能很好地解决空间复杂度过大的问题，虽然优化后的空间复杂度会远低于薛定谔算法的复杂度，但往往中间张量仍然需要大量的存储空间。TensorQ 提供了另外一种算法，手动限制张量的空间复杂度，这种算法是张量网络切片分解（slicing）[13]。

切片分解算法的需要用到如图2.5所示的张量网络性质，在一个包含 4 个张量的网络中，我们可以将网络中的指标 k 固定，指标 k 每一种不同的取值都会形成一个不同的子网络，等式左边表示原始张量网络的缩并结果，等式右边表示不同 k 值的子张量网络缩并后再将结果求和。我们可以证明图2.5中的等式是成立的，在等式左边的缩并过程中，选择 k 指标最后进行缩并，先将 i, j, m, l 指标缩并，再将 k 指标缩并，也就是对 k 指标求和；这和等式右边表示的先缩并子网络再求和是完全一样的。

图2.5的等式说明，我们可以保证得到相同结果的条件下，降低张量缩并的空间复杂度。因为，在图2.5等式左边的张量缩并过程中，最大张量是三阶张量（默认每个指标的维度相同，因为在量子线路的模拟过程中，每个指标的维度都是 2，因此空间复杂度只需要考虑张量阶数即可），而等式右边的缩并过程中，最大张量是二阶张量，其中需要注意指标 k 是固定的，不会贡献张量的阶数。推广到更一般的张量网络缩并过程，TensorQ 中通过设置最大张量阶数，限制了最大空间复杂度，在搜索缩并顺序的过程中会自动地尝试将多个指标进行切片，保证缩并过程不破坏最大空间复杂度的限制。将 m 个维度是 2 的指标切片，意味着切片指标有 2^m 种取值，因此原始张量网络会生成 2^m 个子网络，每个子网络的缩并都不会超过空间复杂度上限，将子网络的缩并结果求和就能得到原始张量网络的缩并结果。

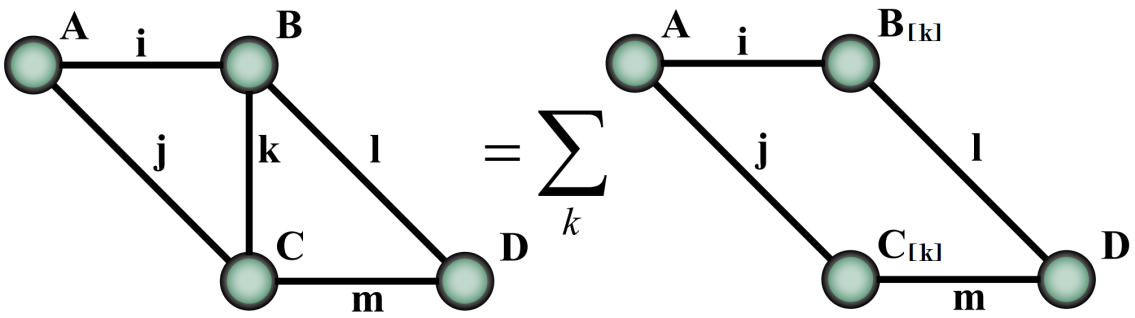


图 2.5: 切片算法示意图。将指标 k 进行切片，也就是将等式左边的原始张量网络的缩并变成了等式右边多个对应不同 k 值的子任务缩并的求和 [13]

另外需要强调的是，切片算法虽然能降低缩并的空间复杂度，但并不能降低整体计算过程的时间复杂度，往往还会导致时间复杂度更高，这称之为切片额外开销（slicing overhead）[2]。我们仍然采用图2.5中的例子进行说明，我们假设所有的指标维度是 2，可以通过枚举证明原始张量网络复杂度最低的缩并顺序是：$[A, B]$, $[AB, C]$, $[ABC, D]$，时间复杂度为：$2^4 + 2^4 + 2^2$；而如果我们采用图2.6所示的，对 i 指标切片，采用复杂度最低的缩并顺序：$[A, C]$, $[AC, B]$, $[ACB, D]$，总空间复杂度为：$2 * (2^3 + 2^3 + 2^2)$；可以发现总的乘法运算增加了 4 次。对于更大的张量网络，切片指标更多的情况，时间复杂度增加得更多。因此切片算法只是空间复杂度超过设备存储范围的权宜之策，通常切片数量越少，时间复杂度越低，建议在 TensorQ 中将空间复杂度限制为设备内存的最大值。

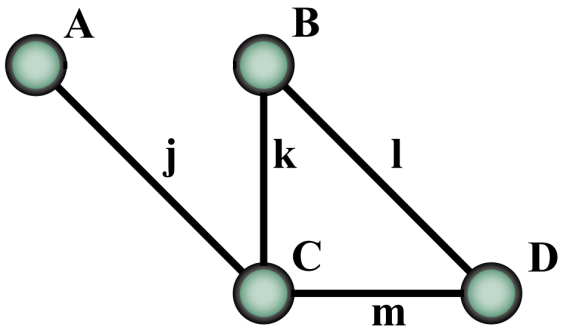


图 2.6: 另外一种切片方案示意图。对指标 i 进行切片，将使整体的时间复杂度更高。

值得一提的是，切片算法还要另外一个作用，在量子纠缠较高的线路中通常纠缠熵较大，纠缠谱是均匀分布的 [7]，类似图2.7中的 $|\phi_{random}\rangle$ 态。

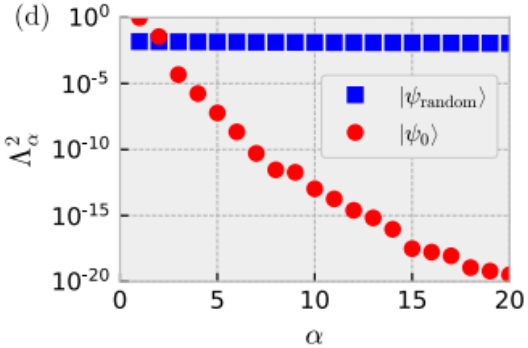


图 2.7: 横场 Ising 模型的两种态的纠缠谱。α 是施密特分解正交项的序号，Λ^2_α 是纠缠谱第 α 项的值，$|\phi_{random}\rangle$ 是自旋随机构形态，$|\psi_0\rangle$ 是基态 [7]。

对于这种态，每一个子网络贡献的结果在统计意义上的等价的，也就是每一个子网络会贡献同样多的保真

度 [13]。如果一个有 K 个子网络，将所有子网络的结果相加可以得到原始网络的精确结果，如果只计算并求和 t($t < K$) 个子网络的结果，将得到原始网络的近似结果，保真度为 $f = t/K$，其中 t 个子网络是任意选择的。这个性质的作用是，当我们不需要模拟量子线路的精确振幅时，可以用这种方法模拟近似振幅，并且时间复杂度相对精确模拟而言降低了 $1/f$ 倍。

因此 TensorQ 通过张量网络切片算法，提供了一个限制空间复杂度方案，虽然这种算法会有用时间复杂度提高的代价，但是它能够模拟切片前会超过设备内存能力的量子线路，完成了之前无法完成的任务。

第 3 章 模块说明与教程

这一章将介绍 TensorQ 中主要的功能模块，包括张量网络的生成，搜索缩并方案以及张量缩并等，同时会提供教程说明如何使用这些模块。

3.1 生成张量网络

3.1.1 使用教程

TensorQ 支持由量子线路自动生成张量网络，在构建量子线路这一步，TensorQ 利用其他软件包完成，这避免了开发者重新学习如何构造量子线路，目前支持'Cirq' 和'MindQuantum' 生成线路。

(a)MindSpore quantum[10] 与 Hiq[9]

(b) Cirq[5]

图 3.1: 目前支持是量子线路生成软件包。

如果你采用 Cirq 工具包来生成量子线路，将线路文件单独放在一个 python 文件中。其中需要注意的是，在量子线路文件中，将量子比特信息赋值给`QUBIT_ORDER`变量，将量子线路信息赋值给`CIRCUIT`变量，TensorQ 将根据这两个变量读取量子线路的信息。

如果你采用 MindQuantum 工具包来生成量子线路，也将线路文件单独放在一个 python 文件中。其中需要注意的是，在量子线路文件中，将量子线路信息赋值给`CIRCUIT`变量，TensorQ 将根据这个变量读取量子线路的信息。同时，在调用 TensorQ 中的 QuantumCircuit 时，输入变量 circuit_package = 'MindQuantum'。

例如图3.2所示的量子线路图可以用下列 python 文件生成。

```
import cirq # 0.7.0版本
import numpy as np

# 比特的网格坐标(QUBIT_ORDER是tensorq需要读取的比特变量)
n = 5
qubits = [cirq.GridQubit(0, i) for i in range(n)]
QUBIT_ORDER = qubits

# 量子线路，注意不需要'测量操作'(CIRCUIT是tensorq需要读取的线路变量)
CIRCUIT = cirq.Circuit(
    moments = [
        cirq.Moment([cirq.H(qubits[0])]),
```

```
        (cirq.Moment([cirq.CNOT(qubits[i], qubits[i+1])]) for i in range(n-1))
    ]
)
```

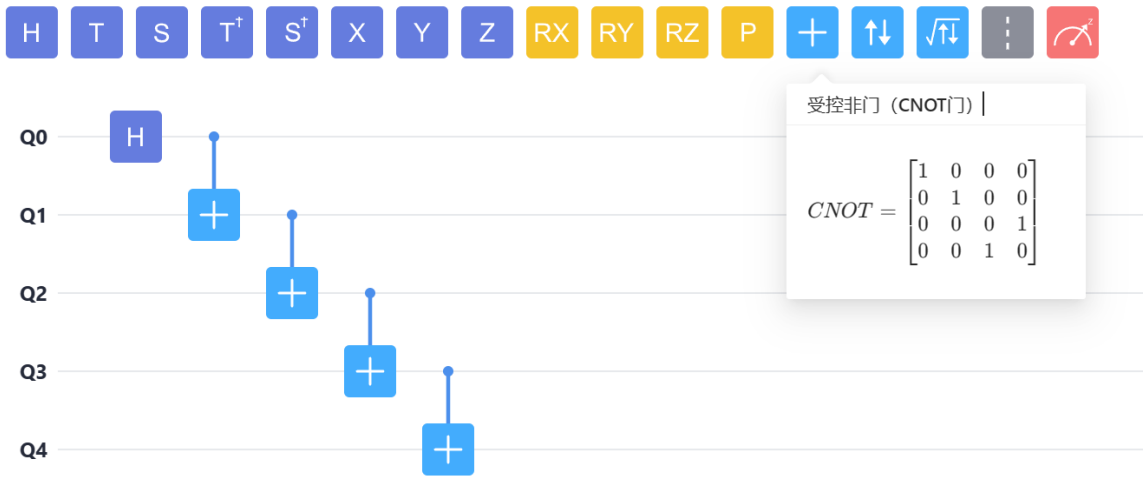


图 3.2: 量子线路示例。采用 HiQ 量子计算云平台 [8] 将量子线路可视化。

将上述量子线路保存到`circuit.py`文件后，用 tensorq 中的 QuantumCircuit 读取量子线路，并将量子子线路转换成张量网络，注意修改量子线路文件后，如果在 Jupyter Notebook 下运行，需要 Restart，tensorq 才能读取到新线路。

```
from tensorq import QuantumCircuit
n = 5
qc_n5 = QuantumCircuit(n = n, twoqubit_simplify = False, fname = 'circuit')
)
```

其中，`n = 5` 表示 5 个比特；`twoqubit_simplify = False`表示不进行两比特门优化，因为该线路中两比特门不必优化；`fname = 'circuit'`表示线路在文件`circuit.py`中。QuantumCircuit 的返回值中将包括，量子线路中每个单元，包括初末态矢量、量子门等对应的张量，以及张量之间的连接关系等。

3.1.2 tensorq.QuantumCircuit

class tensorq.QuantumCircuit

封装好的量子线路类。将量子线路转换成张量网络。

```
class QuantumCircuit(self, n = 53, bitstring = None, fname = None, fix_pairs = [],
                     twoqubit_simplify = True, package = torch, complex128 = False,
                     circuit_package = 'Cirq')
    """
    初始化量子线路
        Args:
            n (int): 量子比特数目
            bitstring (str): 需要计算振幅的比特串
            fname (str): 量子线路文件名（去掉后缀）
            fix_pairs (list): 固定的节点对
            twoqubit_simplify (bool): 是否对两比特门化简
```

```
        package (module): 选择代数计算的软件包, 可选torch与numpy
        complex128 (bool): 选择complex128数据类型(True)或者complex64数据类型(False)
        circuit_package (str, optional): 用哪个包生成量子线路,
            目前支持: 'Cirq', 'MindQuantum'
                Default:``Cirq``

    Attributes:
        n (int): 量子比特数
        gates (list): 量子门
        neighbors (list): 每个张量的邻近张量
        datatype (tyep): 张量的数据类型
        tensors (list): 由量子线路生成的张量
    """
```

fucntion tesnsorq.QuantunCircuit.get_circuit

获取量子线路

```
def get_circuit(self, fname, circuit_package = 'Cirq'):
    """
    获取量子线路。
    Args:
        fname (str): 量子线路文件名 (去掉后缀)
        circuit_package (str, optional): 用哪个包生成量子线路,
            目前支持: 'Cirq', 'MindQuantum'
                Default:``Cirq``

    Returns:
        gates (list): 量子门
    """
```

fucntion tesnsorq.QuantunCircuit.gates_to_tensors

将量子门转化成张量

```
def gates_to_tensors(self, gates, simplify=True, fix_pairs=[]):
    """
    将量子门转化成张量.

    Args:
        gates (list): 量子门
        simplify (bool, optional): 是否对张量网络进行化简
            默认: ``True``.
        fix_pairs (list, optional): 固定的节点对.
            默认: ``[]``.

    Returns:
        - **tensors_simp** (list), 化简后的张量网络.
        - **labels_simp** (list), 每个张量的邻近张量.
```

```
"""
```

3.2 搜索缩并方案

3.2.1 使用教程

TensorQ 可以根据张量网络中每个张量的阶数与张量之间的连通关系自动生成低复杂度的缩并方案，用于后续的张量缩并。TensorQ 采用的是随机搜索算法来寻找低复杂度的缩并方案，TensorQ 的一个特点是，采用无关联多振幅复用算法降低了缩并的时间复杂度，采用了切片算法有效限制了缩并的空间复杂度，保证缩并过程中不会超过设备内存。在寻找缩并方案这给模块中，TensorQ 使用了 arTensor 工具包，其实现了上述功能。

我们以图3.2中的线路为例说明如何使用 TensorQ 产生缩并方案。首先，我们将需要计算振幅的比特串写在一个 txt 文件中。由于图3.2中的线路只有两个比特串的振幅非 0，如图3.3所示，因此我们将'00000', '11111' 写入 txt 文件中，并命名为`two_bitstrings_n5.txt`。

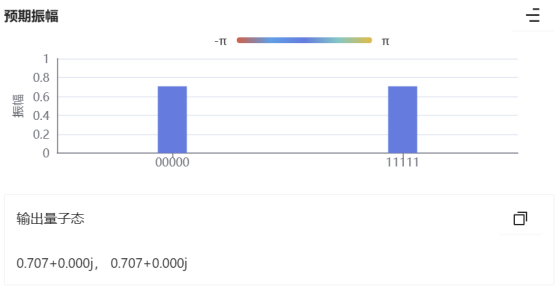


图 3.3: 预期振幅。采用 HiQ 量子计算云平台 [8] 可视化。

接下来就可以通过下列代码，使用 TensorQ 中的 search_order 函数搜索缩并方案

```
from tensorq import search_order
n = 5
contract_scheme = search_order(n = n,device = 'cuda',qc = qc_n5, sc_target = 24,
    bitstrings_txt = 'two_bitstrings_n5.txt', max_bitstrings=2)
```

其中，n = n 表示量子线路 n 个比特；device = 'cuda' 表示用 GPU 进行计算，用 CPU 计算使用 device = 'cpu'；qc = qc_n5 输入之前定义的量子线路。

sc_target = 24 表示限制最大张量小于 24GiB 内存，这是 tensorq 中很核心的一个参数，可以根据自己的设备内存进行设置。同样的缩并任务，内存越大时间复杂度越低，内存越小将会对张量做更多的分解导致更多的浮点运算，这是用时间换空间的办法解决量子模拟中的指数墙困难。

bitstrings_txt = 'two_bitstrings_n5.txt' 表示将我们需要计算振幅的比特串写在 'two_bitstrings_n5.txt'文件里面。

max_bitstrings=2，表示最多计算 2 两个比特串的振幅。

search_order 会打印该缩并方案的复杂度，时间复杂度为浮点数运算次数，以 10 为底的对数表示；空间复杂度为最大张量的元素个数（与 sc_target 单位并不相同），以 2 为底的对数表示；空间复杂度为最大张量元素个数乘以单个数据的存储大小（与数据类型有关）。

```
time complexity (tc,log10), space complexity (sc,log2), memory complexity (mc) =
    (1.9731278535996988, 4.0, 2.238046103128795)
```

contract_scheme 中包含了缩并张量 tensors，缩并步骤 scheme，切片指标 slicing_indices（与张量网络切片分解有关），计算振幅的比特串 bitstrings。

3.2.2 tensorq.search_order

function tensorq.search_order

封装好的函数，搜索张量网络的缩并方案。

```
def search_order(n = 30, m = 14, device = 'cuda', sc_target = 24, seed = 0,
    bitstrings_txt = None,
    max_bitstrings = 1, save_scheme = False, qc = None):
    """
    搜索缩并方案.

    Args:
        n (int, optional): 量子比特的数量, Default: ``30``,
        m (int, optional): 在线路的深度, Default: ``14``,
        device (strings, optional): 计算设备,
            'cuda' for GPU, 'cpu' for CPU,
            Default: ``cuda`` ,
        sc_target (int, optional): 目标空间复杂度, 建议输入设备的内存 (GB),
            Default: ``24``,
        seed (int, optional): 随机种子数, Default: ``24``,
        bitstrings_txt (strings, optional): 一个保存需要计算振幅的比特串的文件,
            Default: ``None``,
        max_bitstrings (int, optional): 需要计算振幅的最大比特串数量,
            Default: ``1``
        save_scheme (bool, optional): 是否将方案保存为文件,
            Default: ``False``
        qc (QuantumCircuit, optional): 量子线路,
            Default: ``None``

    Returns:
        - **result** (tuple), 一个包含缩并需要的数据的元组:
            (tensors_save, scheme_sparsestate, slicing_indices, bitstrings_sorted)
            tensors_save (list): 张量网络中张量的值,
            scheme_sparsestate (list): 缩并步骤,
            slicing_indices (dict): {张量序号: 切片指标},
            bitstrings_sorted (list): 振幅相应的比特串顺序
    """
```

3.3 张量缩并

3.3.1 使用教程

TensorQ 提供与 arTensor 产生的缩并方案适配的缩并运算，可以根据张量网络与缩并方案完成缩并，并且返回最后的计算结果，即目标比特串的振幅。TensorQ 的特点是支持设置子网络计算数量，支持多卡并行计算。在2.3.3中，我们介绍了 TensorQ 采用的张量网络切片分解算法，这个算法会产生很多的子网络，TensorQ 能够设

置计算多少个子网络，同时支持将这些子网络放在多个 GPU 上进行并行计算。

我们以图3.2中的线路为例说明如何使用 TensorQ 完成缩并获得振幅。通过下列代码，就使用 TensorQ 中的 contraction_single_task 函数完成缩并得到振幅。

```
from tensorq import contraction_single_task
tensors, scheme, slicing_indices, bitstrings = contract_scheme
result = contraction_single_task(
    tensors = tensors,
    scheme = scheme,
    slicing_indices = slicing_indices
)
print('fianl amplitue:', result)
```

TensorQ 生成的缩并方案中包括，张量 tensors、缩并步骤 scheme、切片指标 slicing_indices 等变量，将这些变量输入到 contraction_single_task，就会返回张量网络的缩并结果，即需要的振幅。

```
fianl amplitue: tensor([0.7071+0.j, 0.7071+0.j])
```

这与图3.3中的结果一致。

3.3.2 tensorq.contraction_single_task

function tensor_contraction_sparse

根据缩并方案与张量网络完成单的子网络的缩并。

```
def tensor_contraction_sparse(tensors, contraction_scheme, use_cutensor=False):
    '''
    根据缩并方案对张量网络进行缩并

    Args:
        tensors (list): 张量网络的数值
        contraction_scheme (list):
            缩并步骤列表，每个步骤的多个条目:
            step[0]: 要缩并的张量的位置
            step[1]: 张量缩并的爱因斯坦指标求和等式
            step[2]: 缩并过程中的多振幅对应的维度
            step[3]: 可选，如果第二个张量有多振幅维度，这是reshape序号
            step[4]: 可选，如果第二个张量有多振幅维度，这是用于验证reshape序列的正确维度
        use_cutensor (bool, optional):
            是否在einsum中使用cutensor软件包，
            Default: ``False``


    Returns:
        - **tensors[i]** (torch.Tensor)，最终得到的振幅
    '''
```

function contraction_single_task

根据切片算法对张量进行化简，并且可以将多个子网络的缩并作为一个任务。

```
def contraction_single_task(
        tensors:list, scheme:list, slicing_indices:dict,
        task_id = 0, device = 'cuda:0', n_sub_task = 1, use_cutensor = False, file_save =
            False
    ):
    """
    将张量网络切片，完成一个缩并任务。

    Args:
        tensors (list): 张量网络的数值，
        scheme (list):
            list of contraction step, defintion of entries in each step:
            step[0]: 要缩并的张量的位置
            step[1]: 张量缩并的爱因斯坦指标求和等式
            step[2]: 缩并过程中的多振幅对应的维度
            step[3]: 可选，如果第二个张量有多振幅维度，这是reshape序号
            step[4]: 可选，如果第二个张量有多振幅维度，这是用于验证reshape序列的正确维度
        slicing_indices (dict): {张量序号: 切片指标},,
        task_id (int, optional): 任务编号, Default: ``0``
        device (string, optional): 用于计算的设备，
            'cuda:0' for GPU, 'cpu' for CPU,
            Default: ``cuda:0``
        n_sub_task (int, optional): 每个任务中子任务的数量
            Default: ``1``,
        use_cutensor (bool, optional):
            是否在einsum中使用cutensor软件包，
            Default: ``False``
        file_save (bool, optional): 是否将结果保存为文件，
            Default: ``False``

    Returns:
        - **collect_tensor.cpu()** (torch.Tensor), 最终得到的振幅
    """
```

第 4 章 案例测试与分析

这一章将通过一些案例来展示 TensorQ 的功能特点。首先 TensorQ 能模拟大规模的量子线路，我们通过一个 35 比特的 GHZ 态制备来体现这个功能。

4.1 35 比特 GHZ 态制备

这个案例的完整代码已经上传到了 GitHub 仓库Xenon-zhao/TensorQ/examples/quick_start_advance。

我们展示一个制备 35 比特 GHZ 态的线路，这个线路类似图3.2，用 cirq 模拟将需要 256GB 内存，导致 cirq 报错无法模拟。而使用 tensorq 可以完成模拟。

首先，我们用 cirq 模拟 10 比特的类似线路，验证代码的正确性。

```
import cirq

# Set n qubit.
n = 10
qubits = [cirq.GridQubit(0, i) for i in range(n)]

# Create a circuit
# circuit = cirq.Circuit(cirq.Moment([cirq.H(qubits[0])]))
circuit = cirq.Circuit(
    moments = [
        cirq.Moment([cirq.H(qubits[0])]),
        (cirq.Moment([cirq.CNOT(qubits[i], qubits[i+1])]) for i in range(n-1))
    ]
)

# Simulate the circuit several times.
simulator = cirq.Simulator()
result = simulator.simulate(circuit)
print("Results:")
print(result)
```

输出：

```
Results:
measurements: (no measurements)
output vector: 0.707|0000000000> + 0.707|1111111111>
```

成功制备了 10 比特 GHZ 态.

接下来我们把 n 设置为 35，表示模拟 35 比特的线路

```
import cirq

# Set n qubit.
n = 35
```

```
qubits = [cirq.GridQubit(0, i) for i in range(n)]

# Create a circuit
# circuit = cirq.Circuit(cirq.Moment([cirq.H(qubits[0])]))
circuit = cirq.Circuit(
    moments = [
        cirq.Moment([cirq.H(qubits[0])]),
        (cirq.Moment([cirq.CNOT(qubits[i], qubits[i+1])]) for i in range(n-1))
    ]
)
print("Circuit:")
print(circuit)

# Simulate the circuit several times.
simulator = cirq.Simulator()
result = simulator.simulate(circuit)
print("Results:")
print(result)
```

输出：

```
MemoryError: Unable to allocate 256. GiB for an array with shape (34359738368,) and data
    type complex64
```

这里发生报错！需要 256GB 内存，无法模拟！

接下来，我们用 tensorq 模拟同样的线路。首先在'circuit.py' 文件中构建量子线路，然后用 tensorq 中的 QuantumCircuit 将量子子线路转换成张量网络（注意修改'circuit.py' 文件后，要 Restart，tensorq 才能读取到新线路）

```
from tensorq import QuantumCircuit
n = 35
qc_n35 = QuantumCircuit(n = n,twoqubit_simplify = False,fname = 'circuit')
```

其中，n = 35 表示 35 个比特；twoqubit_simplify = False 表示不进行两比特门优化，因为该线路中两比特门不必优化；fname = 'circuit' 表示线路在文件'circuit.py' 中。

用 tensorq 中的 search_order 搜索缩并方案

```
from tensorq import search_order
contract_scheme = search_order(n = n,device = 'cuda',qc = qc_n35, sc_target = 24,
    bitstrings_txt = 'two_bitstrings_n35.txt', max_bitstrings=2)
```

输出：

```
time complexity (tc,log10), space complexity (sc,log2), memory complexity (mc) =
    (3.1535099893008374, 4.0, 3.338456493604605)
```

其中，n = n 表示 n 个比特；device = 'cuda' 表示用 GPU 进行计算，用 CPU 计算使用 device = 'cpu'；qc = qc_n35 输入之前定义的量子线路。

sc_target = 24 表示限制最大张量小于 24GB 内存，这是 tensorq 中很核心的一个参数，可以根据自己的设备内存进行设置。同样的缩并任务，内存越大时间复杂度越低，内存越小将会对张量做更多的分解导致更多的浮点运算，这是用时间换空间的办法解决量子模拟中的指数墙困难。

bitstrings_txt = 'two_bitstrings_n35.txt' 表示将我们需要计算振幅的比特串写在 'two_bitstrings_n35.txt'文件里面，因为只有'00000000000000000000000000000000000000', '11111111111111111111111111111111111111' 两个比特串的振幅非 0，因此这里只计算两个比特串的振幅。

max_bitstrings=2，表示最多计算 2 两个比特串的振幅。

search_order 会打印该缩并方案的复杂度，时间复杂度为浮点数运算次数，以 10 为底的对数表示；空间复杂度为最大张量的元素个数（与 sc_target 单位并不相同），以 2 为底的对数表示；空间复杂度为最大张量元素个数乘以单个数据的存储大小（与数据类型有关）。空间复杂度为 4 说明，最大张量为 2^4，由于 TensorQ 默认采用的是 complex64 数据类型，每个数据占 8 个字节，张量网络缩并通常对内存的需求为最大张量的两倍，因此这里 TensorQ 需要的内存为 256 bytes，还不到 1MB，远小于 256GB。

最后，用 tensorq 的 contraction_single_task 对张量进行缩并。

```
from tensorq import contraction_single_task
tensors, scheme, slicing_indices, bitstrings = contract_scheme
result = contraction_single_task(
    tensors,
    scheme,
    slicing_indices
)
print('fianl amplitue:', result)
```

输出：

```
fianl amplitue: tensor([0.7071+0.j, 0.7071+0.j])
```

contract_scheme 中包含了缩并张量 tensors，缩并步骤 scheme，切片指标 slicing_indices（与张量分解有关），计算振幅的比特串 bitstrings，将这些变量输入到 contraction_single_task，就会返回张量网络的缩并结果，就是全部末态比特串的振幅。

在这个例子中，由于目标比特串比较少，仅通过调整张量网络的缩并顺序就可以避免出现较大的张量，因此 search_order 中只优化了缩并顺序，并没有采用切片策略来减小张量维度。

4.2 模拟 Sycamore 量子计算机

这个例子模拟 Google 的 Sycamore 量子计算机，30 量子比特 14 层量子线路，TensorQ 可以给出保真度接近 1 的振幅。这个案例的完整代码已经上传到了 GitHub 仓库Xenon-zhao/TensorQ/examples/simulate_sycamore。

```
from tensorq import QuantumCircuit
n = 30
qc_n = QuantumCircuit(n = n, fname = 'circuit_n30_m14_s0_e0_pEFGH')
```

n = 30 表示 30 量子比特。circuit_n30_m14_s0_e0_pEFGH 是线路文件，可以在Google's data repository for the Sycamore circuits.下载

```
from tensorq import search_order
contract_scheme = search_order(n = n, device = 'cuda', qc = qc_n, sc_target = 24,
    bitstrings_txt = 'amplitudes_n30_m14_s0_e0_pEFGH_10000.txt', max_bitstrings=1000)
```

输出：

```
Finish find contraction tree, next construct the scheme.
```

```
Finish construct the scheme.
time complexity (tc,log10), space complexity (sc,log2), memory complexity (mc) =
    (12.014205108939997, 30.0, 10.304655927314002)
```

其中，n = n 表示 n 个比特；device = 'cuda' 表示用 GPU 进行计算，用 CPU 计算使用 device = 'cpu'；qc = qc_n 输入之前定义的量子线路。

sc_target = 24 表示限制最大张量小于 24GB 内存，这是 tensorq 中很核心的一个参数，可以根据自己的设备内存进行设置。同样的缩并任务，内存越大时间复杂度越低，内存越小将会对张量做更多的分解导致更多的浮点运算，这是用时间换空间的办法解决量子模拟中的指数墙困难。

bitstrings_txt = 'amplitudes_n30_m14_s0_e0_pEFGH_10000.txt' 表示将我们需要计算振幅的比特串写在 'amplitudes_n30_m14_s0_e0_pEFGH_10000.txt'文件里面。

max_bitstrings=10000，表示计算 10000 个比特串的振幅。

search_order 会打印该缩并方案的复杂度，时间复杂度为浮点数运算次数，以 10 为底的对数表示；空间复杂度为最大张量的元素个数（与 sc_target 单位并不相同），以 2 为底的对数表示；空间复杂度为最大张量元素个数乘以单个数据的存储大小（与数据类型有关）。

用 TensorQ 计算振幅

```
from tensorq import contraction_single_task
tensors, scheme, slicing_indices, bitstrings = contract_scheme
result = contraction_single_task(
    tensors,
    scheme,
    slicing_indices,
)
print('fianl amplitue:', result)
```

输出：

```
fianl amplitue: tensor([ 4.3482e-08-1.7836e-05j, -3.2258e-05+8.9927e-05j,
        1.0095e-05+1.8853e-05j, -2.1370e-05-4.6581e-06j,
       -2.2554e-06-6.5780e-06j, 2.2370e-05+1.8043e-05j,
...
        4.7438e-05-2.5849e-05j, 1.6054e-05-1.9300e-05j,
        1.0810e-05+5.2313e-06j, 8.9989e-06-3.1390e-06j,
       -1.9119e-05-7.8497e-07j, -1.0527e-05-1.4118e-05j])
```

contract_scheme 中包含了缩并张量 tensors，缩并步骤 scheme，切片指标 slicing_indices（与张量分解有关），计算振幅的比特串 bitstrings，将这些变量输入到 contraction_single_task，就会返回张量网络的缩并结果，就是全部末态比特串的振幅。

从'amplitudes_n30_m14_s0_e0_pEFGH_10000.txt' 文件中读取相应的精确振幅以及比特串。

```
from tensorq import read_samples
import numpy as np
data = read_samples('amplitudes_n30_m14_s0_e0_pEFGH_10000.txt')
max_bitstrings = 10000
bitstrings_exact = [data[i][0] for i in range(max_bitstrings)]
amplitude_exact = np.array([data[i][1] for i in range(max_bitstrings)])
```

将精确振幅的顺序按照我们计算的比特串顺序排列。

```
import torch
amplitude_exact_1000 = []
for i in range(1000):
    amplitude_exact_1000.append(amplitude_exact[bitstrings_exact.index(bitstrings[i])])
amplitude_exact_1000 = torch.tensor(amplitude_exact_1000, dtype=torch.complex64)
```

计算 TensorQ 结果的保真度。

```
fidelity = (
        (amplitude_exact_1000.conj() @ result.reshape(-1)).abs() /
        (amplitude_exact_1000.abs().square().sum().sqrt() * result.abs().square().sum().sqrt
            ())
    ).square().item()
fidelity
```

输出：

```
0.9999997615814209
```

TensorQ 结果的保真度接近 1。

计算一组均匀分布的白噪音的保真度。

```
noisy = torch.tensor([1/np.sqrt(len(result)) for i in range(len(result))], dtype=torch.
    complex64)
fidelity2 = (
        (amplitude_exact_1000.conj() @ noisy.reshape(-1)).abs() /
        (amplitude_exact_1000.abs().square().sum().sqrt() * noisy.abs().square().sum().sqrt()
            )
    ).square().item()
fidelity2
```

输出：

```
0.0013713767984881997
```

白噪音的保真度远小于 1。

第 5 章 总结

TensorQ 基于张量网络算法，通过优化缩并顺序，无关联多振幅复用，张量网络切片等算法，实现了对大规模量子线路的模拟，在一定程度上解决了量子模拟中的指数墙问题。利用 TensorQ 工具包，可以实现在个人电脑上模拟 35 比特甚至更大的量子线路，这是 Cirq，MindQuantum 等工具包都无法完成的事情。

在本教程中，我们面向初学者介绍了，张量网络的基本概念以及张量网络缩并中用到的基础算法，希望能让 TensorQ 软件包的使用者能够更加直观地了解 TensorQ，并且能够从底层原理上明白 TensorQ 的功能。如今发展出来了大量基于张量网络的算法，TensorQ 提供一些最基础的框架与功能，并且展示了如何利用 TensorQ 去使用这些算法。

本教程还介绍了 TensorQ 中包含的一些基础模块，有相应的教程说明如何使用这些模块，帮助使用者轻松上手；同时对于想进一步使用 TensorQ 的使用者，本教程提供了各个模块的技术文档，包括输出与输出的格式与含义。如果想深如研究 TensorQ，也可以去 Github 仓库上查看源代码，TensorQ 所有的原代码都已经开源。

本教程通过一些完整的例子——制备 35 比特 GHZ 态，将 TensorQ 与 Cirq 进行比较，在这个例子中采用 Cirq 软件包需要的内存过大，无法运行的问题；而 TensorQ 需要的内存很小，并且给出了正确的结果，这个例子展示了 TensorQ 的强大性。本教程通过另外一个例子模拟了 30 比特的 sycamore 量子计算机，在 30 比特的情况下可以求解精确振幅；推广到比特数量更多的情况，TensorQ 也可以进行模拟，不过模拟的精度会降低，模拟时间会有所增加。这个例子展示了 TensorQ 可以应用于量子计算前沿研究，具有重要的价值。

最后，非常欢迎各位对 TensorQ 做出一份贡献！

参考文献

[1] Frank Arute et al. “Quantum supremacy using a programmable superconducting processor”. In: *Nature* 574 (2019), pp. 505–510. DOI: 10.1038/s41586-019-1666-5. URL: https://doi.org/10.1038/s41586-019-1666-5.

[2] Yaojian Chen et al. “Lifetime-Based Optimization for Simulating Quantum Circuits on a New Sunway Supercomputer”. In: *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. PPoPP ’23. Montreal, QC, Canada: Association for Computing Machinery, 2023, pp. 148–159. DOI: 10.1145/3572848.3577529. URL: https://doi.org/10.1145/3572848.3577529.

[3] J. Ignacio Cirac et al. “Matrix product states and projected entangled pair states: Concepts, symmetries, theorems”. In: *Rev. Mod. Phys.* 93 (4 Dec. 2021), p. 045003. DOI: 10.1103/RevModPhys.93.045003. URL: https://link.aps.org/doi/10.1103/RevModPhys.93.045003.

[4] S J Denny et al. “Algebraically contractible topological tensor network states”. In: *Journal of Physics A: Mathematical and Theoretical* 45.1 (Dec. 2011), p. 015309. DOI: 10.1088/1751-8113/45/1/015309. URL: https://dx.doi.org/10.1088/1751-8113/45/1/015309.

[5] Google. “Cirq home”. In: (2019). URL: https://quantumai.google/cirq.

[6] Johnnie Gray and Stefanos Kourtis. “Hyper-optimized tensor network contraction”. In: *Quantum* 5 (Mar. 2021), p. 410. ISSN: 2521-327X. DOI: 10.22331/q-2021-03-15-410. URL: https://doi.org/10.22331/q-2021-03-15-410.

[7] Johannes Hauschild and Frank Pollmann. “Efficient numerical simulations with Tensor Networks: Tensor Network Python (TeNPy)”. In: *SciPost Phys. Lect. Notes* (2018), p. 5. DOI: 10.21468/SciPostPhysLectNotes.5. URL: https://scipost.org/10.21468/SciPostPhysLectNotes.5.

[8] HUAWEI. “HiQ composer”. In: (2021). URL: https://hiq.huaweicloud.com/portal/programming/hiq-composer.

[9] HUAWEI. “HiQ home”. In: (2021). URL: https://hiq.huaweicloud.com/home.

[10] HUAWEI. “MindSpore gitee”. In: (2019). URL: ttps://gitee.com/mindspore/mindquantum.

[11] Yong (Alexander) Liu et al. “Closing the ”Quantum Supremacy” Gap: Achieving Real-Time Simulation of a Random Quantum Circuit Using a New Sunway Supercomputer”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3487399. URL: https://doi.org/10.1145/3458817.3487399.

[12] Changhun Oh et al. “Tensor network algorithm for simulating experimental Gaussian boson sampling”. In: (2023). arXiv: 2306.03709 [quant-ph].

[13] Feng Pan, Keyang Chen, and Pan Zhang. “Solving the Sampling Problem of the Sycamore Quantum Circuits”. In: *Phys. Rev. Lett.* 129 (9 Aug. 2022), p. 090502. DOI: 10.1103/PhysRevLett.129.090502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.129.090502.

[14] Feng Pan and Pan Zhang. *Simulating the Sycamore quantum supremacy circuits*. 2021. arXiv: 2103.03074 [quant-ph].

[15] Shi-Ju Ran. “张量网络基础课程”. In: (2020). URL: https://www.bilibili.com/video/BV17z411i7yM/?vd_source=3cb43e4790ca78e9a67387f3843feb4d.

[16] L.G. Valiant. “The complexity of computing the permanent”. In: *Theoretical Computer Science* 8.2 (1979), pp. 189–201. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(79)90044-6. URL: https://www.sciencedirect.com/science/article/pii/0304397579900446.

[17] Han-Sen Zhong et al. “Quantum computational advantage using photons”. In: *Science* 370.6523 (2020), pp. 1460–1463. DOI: 10.1126/science.abe8770. eprint: https://www.science.org/doi/pdf/10.1126/science.abe8770. URL: https://www.science.org/doi/abs/10.1126/science.abe8770.