

# P1 封面

各位老师好，我是来自 张老师 课题组的周予恺。今天非常感谢各位老师参加我的中期答辩。

本次汇报将围绕我目前正在进行的研究工作展开，主题是基于流方程的量子动力学模拟方法的优化与GPU实现。该工作聚焦于突破当前多体量子系统动力学模拟中的规模瓶颈与算法效率问题，尝试引入扰乱变换与GPU并行计算策略提升传统 CUT 方法的可扩展性。

## P2 研究背景

先简单介绍一下这个工作的背景。

量子动力学研究的是量子多体系统在时间上的演化行为，比如一个系统从初态演化到某个远期态，它的能量是怎么转移的、信息是怎么扩散的等等。

但模拟这样的系统非常困难，原因是希尔伯特空间的维度随着粒子数呈指数增长——这意味着，系统越大，需要模拟的数据量也急剧变大。

量子系统的演化由**时依薛定谔方程**控制：

$$i\hbar \frac{d}{dt} |\psi(t)\rangle = H |\psi(t)\rangle \quad (1)$$

- $|\psi(t)\rangle$  是系统的**量子态（波函数）**
- $H$  是系统的**哈密顿量**（Hermitian 运算符，表示能量）
- $\hbar$  是普朗克常数，设为 1 可简化讨论（自然单位）

# 补充

## 薛定谔方程的形式解

当  $H$  是时间无关的（常见情形），解可以形式写为：

$$|\psi(t)\rangle = e^{-iHt/\hbar}|\psi(0)\rangle \quad (2)$$

即状态的时间演化是由一个**指数矩阵算符**  $e^{-iHt}$  作用在初始态上。

## 模拟任务的本质

从数值角度看，模拟量子系统的核心任务是：

将哈密顿量  $H$  多次作用在波函数  $|\psi\rangle$  上，以获得时间演化状态。

这就需要：

- 表示  $|\psi\rangle$ （一个  $d^L$  维的复向量）
- 存储和操作  $H$ （一个  $d^L \times d^L$  的稀疏矩阵）
- 实现  $e^{-iHt}|\psi(0)\rangle$  的近似计算

## 为什么操作哈密顿量非常昂贵？

### 1. 状态空间维度指数增长

假设每个粒子是自旋- $\frac{1}{2}$  ( $d = 2$ )，粒子数为  $L$ ，那么：

- $|\psi(t)\rangle \in \mathbb{C}^{2^L}$
- $H \in \mathbb{C}^{2^L \times 2^L}$

即使  $H$  是**稀疏矩阵**，作用一次  $H|\psi\rangle$  仍然是  $O(2^L)$  的计算量和内存消耗。

### 2. 演化算符的计算代价高

你不能真的“写出” $e^{-iHt}$ ，只能近似它。

常见近似方法包括：

- **幂级数展开**（收敛性差）
- **Trotter 分解**（将  $H = H_1 + H_2 + \dots$  拆开，小步推进）：

$$e^{-iHt} \approx \left( e^{-iH_1\Delta t} e^{-iH_2\Delta t} \dots \right)^n \quad (3)$$

- 每步都要将  $H_i$  作用到  $|\psi\rangle$  上，代价仍为  $O(2^L)$

### 3. 急剧增加的内存压力

很多量子态**无法压缩表示**，因为它们包含高度纠缠信息，不能用稀疏/低秩形式近似。

## P3

在刚才介绍了哈密顿量之后，我们接下来引出本研究中采用的核心方法：**连续酉变换 (Continuous Unitary Transformations)**，也就是所谓的 CUT 方法。

这个方法最初由 Wegner 和 Glazek 等人提出，其基本思想是：

我们不直接对  $H$  做对角化，而是引入一个“流动参数”  $l$ ，令哈密顿量随着  $l$  演化：

$$\frac{dH(l)}{dl} = [\eta(l), H(l)] \quad (4)$$

这里的  $\eta(l)$  被称为“生成元”，它控制了每一步变换的方向。

当  $l \rightarrow \infty$  时， $H(l)$  会逐渐变成对角形式，我们就可以更方便地处理系统的演化问题。

这个变换是酉的，也就是说它不会改变系统的物理性质，只是换了一个更好处理的表示方式。

在这项工作中，我们结合了这个 CUT 框架，去研究复杂量子系统的动力学行为，同时探索如何让整个过程在 GPU 上高效运行。

## 补充

### 为什么选择 CUT 方法：与其他方法的对比

在模拟多体量子系统的非平衡演化时，常见方法及其局限如下：

方法	原理	局限性
精确对角化 (ED)	全量表示哈密顿量并求解本征态	指数计算复杂度，粒子数受限 ( $\lesssim 20$ )
张量网络方法 (TEBD、TDVP)	低纠缠压缩表示量子态	纠缠增长快，难以处理长时间动力学和高维系统
量子蒙特卡罗 (QMC)	概率采样计算	存在严重的符号问题，尤其在非平衡态中不稳定
机器学习方法 (如 RBM)	神经网络表示波函数	表达能力和可解释性有限，优化过程复杂

相比之下，CUT 方法具备如下优势：

- 利用酉变换，不改变系统物理性质；
- 可以逐步逼近对角形式，避免直接操作大矩阵；
- 适用于高维系统，具备良好的 GPU 并行实现基础；
- 在截断控制下，能够获得系统在中长时间尺度下的稳定演化解。

### 什么是酉变换？为什么它不会改变物理结果？

酉变换是满足下列条件的线性变换：

$$U^\dagger U = U U^\dagger = I \quad (5)$$

也即， $U$  是一个酉矩阵，其逆等于厄米共轭。

性质：

- 保持内积不变： $\langle \psi | \phi \rangle = \langle U\psi | U\phi \rangle$
- 保持谱结构不变：本征值不变，只改变表象（basis）
- 保持观测物理量期望值不变：

$$\langle \psi | O | \psi \rangle = \langle U\psi | U O U^\dagger | U\psi \rangle \quad (6)$$

因此，虽然 CUT 改变了哈密顿量的表示形式（变成对角形式），但系统的物理演化完全不变。

## CUT 中的“流时间”、“生成元”与“流动演化”机制

### 1. 流动参数 $l$ ：

- 并非物理时间，而是一个“对角化进程”的参数；
- 初始时  $H(0)$  是原始哈密顿量， $H(l)$  随  $l$  渐趋对角。

### 2. 生成元 $\eta(l)$ ：

- 控制每一步变换方向；
- 常用形式是 Wegner 生成元：

$$\eta(l) = [H_0(l), V(l)] \quad (7)$$

其中  $H_0$  为对角部分， $V$  为非对角部分。

### 3. 演化过程：

- 每一步小变换： $H(l + \delta l) = H(l) + \delta l[\eta(l), H(l)]$
- 通过数值 ODE（如 RK4）迭代实现；
- 直至  $V(l)$  足够小（如  $< 10^{-3}$ ），认为系统已“对角化”。

RK4

CUT 的核心是流方程：

$$\frac{dH(l)}{dl} = [\eta(l), H(l)] \tag{8}$$

这是一个关于哈密顿量  $H(l)$  的常微分方程（ODE），我们需要数值积分来求解其随流时间  $l$  的演化。

## Runge-Kutta 四阶方法（RK4）

RK4 是一种常用的高精度积分器，其公式如下：

$$\begin{aligned} k_1 &= f(l, H(l)) = [\eta(l), H(l)] \\ k_2 &= f\left(l + \frac{\delta l}{2}, H(l) + \frac{\delta l}{2} k_1\right) \\ k_3 &= f\left(l + \frac{\delta l}{2}, H(l) + \frac{\delta l}{2} k_2\right) \\ k_4 &= f(l + \delta l, H(l) + \delta l \cdot k_3) \\ H(l + \delta l) &= H(l) + \frac{\delta l}{6} (k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \tag{9}$$

每一步都要：

- 构造当前的生成元  $\eta(l) = [H_0(l), V(l)]$ ；
- 计算对易子  $[\eta(l), H(l)]$ ；
- 利用 RK4 方法推进  $H(l) \rightarrow H(l + \delta l)$ 。

## 为什么用 RK4？

方法	精度	适用性
Euler	一阶	误差较大，不适合复杂量子系统
RK4	四阶，误差 $\mathcal{O}(\delta l^5)$	 准确稳定，最常用
自适应步长	自动调整 $\delta l$	用于高精度、长时间演化

现代实现中，常用 `JAX`，`SciPy` 等库的 `odeint` 或 `solve_ivp` 函数封装这些方法，并支持 GPU 并行计算。

```

from jax.experimental.ode import odeint # 或使用
scipy.integrate.solve_ivp

def flow_eq(H_l, l):
    eta = commutator(H_diag(H_l), H_offdiag(H_l))
    return commutator(eta, H_l)

H_trajectory = odeint(flow_eq, H0, l_span) # 自动执行 RK 或更高阶
方法

```

## 什么是“将哈密顿量分解为低阶张量”？

理论上，哈密顿量  $H$  是一个  $d^L \times d^L$  的矩阵（ $L$  为粒子数， $d$  为局域维度）。

在 CUT 中，为了实现逐步演化与截断控制，我们采用如下表示：

- 将  $H$  写成**多体算符的展开**，如：

$$H = \sum_{i,j} H_{ij}^{(2)} : c_i^\dagger c_j : + \sum_{i,j,k,l} H_{ijkl}^{(4)} : c_i^\dagger c_j c_k^\dagger c_l : + \dots \quad (10)$$

- $H^{(2)}$  是 2 阶张量（矩阵）；
- $H^{(4)}$  是 4 阶张量；
- 更高阶可以按需加入。

## 为什么这样做？

- 我们不直接存整个  $H$  的大矩阵，而是存储其构成项的系数；
- 每一步的变换只作用于这些系数，通过**张量收缩**计算生成元与流动；
- 显著减少内存和计算复杂度，尤其适合 GPU 并行化

## CUT 中的张量展开：什么是多体算符？为什么是高阶张量？

在 CUT 方法中，我们不直接保存哈密顿量的大矩阵，而是将它展开为一系列具有物理含义的“多体算符”项。形式如下：

$$H = \sum_{i,j} H_{ij}^{(2)} : c_i^\dagger c_j : + \sum_{i,j,k,l} H_{ijkl}^{(4)} : c_i^\dagger c_j c_k^\dagger c_l : + \dots \quad (11)$$

其中：

- $:\cdot:$  表示正规序 (normal ordering)，确保反对易规则满足；
- $c_i^\dagger$ 、 $c_j$  是费米子的产生与湮灭算符；
- $H^{(2)}$  是一体项，对应一个  $L \times L$  的矩阵（2 阶张量）；
- $H^{(4)}$  是二体相互作用，对应一个  $L \times L \times L \times L$  的 4 阶张量；
- 更高阶项可表示三体、四体相互作用等。

## 多体算符的含义

- **一体算符**：例如  $c_i^\dagger c_j$ ，表示一个粒子从  $j$  跃迁到  $i$ ；
- **二体算符**：例如  $c_i^\dagger c_j c_k^\dagger c_l$ ，表示两个粒子之间的相互作用；
- 一般而言， $2n$  个算符构成的算符表示  $n$  体相互作用，称为  $n$ -体算符。

## 为什么使用高阶张量？

虽然整个哈密顿量  $H$  是一个  $2^L \times 2^L$  的大矩阵，但将其展开为多体算符项后，可以只保存每种结构对应的系数张量，例如：

- $H^{(2)}$ ：存  $L^2$  个参数；
- $H^{(4)}$ ：存  $L^4$  个参数；
- 更高阶的项在弱相互作用下可以忽略。

这种表示方式可以大大降低内存开销，并适用于张量收缩与 GPU 加速。



这一页介绍的是 CUT 方法如何**处理高维系统**的问题。

通常我们处理二维格点系统时，格点只和自己临近的格点发生作用，属于“局域哈密顿量”。

为了让 CUT 方法可以处理这些高维系统，我们采用一种技巧：**将系统一维展开**。具体方法如下：

- 将  $d$  维晶格用某种顺序编号成 1 到  $L$ ，将所有格点排成一条线；
- 原本的近邻耦合项在展开后就变成了一维系统中的“长程跃迁”。

也就是说，原来只连接  $(i, j)$  近邻格点的项，现在会变成连接线性编号上较远的格点。

但这对 CUT 方法不是问题：

- 因为 CUT 的基本操作是计算 operator 的对易子，这完全取决于代数结构（例如泡利代数），
- 它**不依赖系统的几何维度**。

因此，这种一维展开策略使 CUT 可以处理任意维度的系统，包括高维无序体系、拓扑材料等。

P5

图片展示

P6&P7

以上是原始的 CUT 方法，但这个 CUT 的核心问题之一是可能遇到 **不稳定固定点**。

CUT 采用的基本方程是：

$$\frac{dH(l)}{dl} = [\eta(l), H(l)] \quad (12)$$

常用的 Wegner 生成元为：

$$\eta(l) = [H_0(l), V(l)] \quad (13)$$

其中  $H_0$  是对角部分， $V$  是非对角部分。

如果系统存在许多近简并能级，即  $H_{ii}^{(2)} \approx H_{jj}^{(2)}$ ，那么：

$$\eta_{ij} \propto (H_{ii} - H_{jj}) \cdot V_{ij} \rightarrow 0 \quad (14)$$

就会导致对角化速度变慢，甚至停滞，出现所谓“固定点”。

这限制了 CUT 在某些系统中的收敛性和精度。

## P8

为了解决 CUT 方法中“固定点”带来的对角化停滞问题，有一篇论文提出了引入了 **扰乱变换 (Scrambling Transform)** 的方法。

其思想是先通过一个酉变换  $S(l)$  对系统进行预处理，打破系统中的近简并结构，使后续 CUT 更容易收敛。

扰乱变换本身不改变系统物理性质，其数学形式为：

$$dS(l) = \exp(-\lambda(l) dl) \quad (15)$$

其中  $\lambda(l)$  是专门设计的扰动生成元，用于激活能级差很小的部分。

通过这种变换，可以显著提升 CUT 的稳定性与数值效率。

## 补充

### 扰乱变换的数学形式

扰乱变换自身也遵循一个微分方程：

$$dS(l) = \exp \left( - \lambda(l) dl \right) \tag{16}$$

其中：

- $\lambda(l)$  是扰动生成元；
- 用于激活哈密顿量中小能级差部分的非对角耦合。

一个典型的构造方式：

$$\lambda_{ij}(l) = \frac{[H(l)]_{ij}}{\epsilon + |\epsilon_i - \epsilon_j|} \tag{17}$$

- $\epsilon_i, \epsilon_j$  为哈密顿量中对应本征态的能级；
- $\epsilon$  为防止除以零的小常数；
- 保证能级越接近，扰动越强。

更新公式：

$$S(l + \delta l) = e^{-\lambda(l) \delta l} \cdot S(l) \tag{18}$$

更新后的哈密顿量：

$$H(l) \leftarrow S(l) H(l) S^\dagger(l) \tag{19}$$

再继续标准 CUT 流动方程。

# P9 展示图片

## P10

除了算法上的优化，也可以在 gpu 上对 CUT 的流程进行加速。

首先，哈密顿量的演化过程可以用gpu加速

### 数值方法与流方程核心逻辑

- 实现了 CUT 的完整演化流程，通过流方程  $\frac{dH}{dt} = [\eta, H]$  实现系统哈密顿量的对角化；
- 将哈密顿量明确分为 **对角项与非对角项**，分别处理生成元构造与演化更新；
- 支持 **相互作用与非相互作用项** 的分离处理，便于扩展多体耦合场景。

### GPU 并行与张量优化

- 全部张量操作基于 PyTorch 并 **原生支持 GPU 并行计算**；
- 利用 `torch.float32` 精度平衡内存消耗与计算性能；
- 所有张量在 GPU 上原位创建，避免不必要的数据搬移；
- 核心张量收缩逻辑（位于 `contract_torch.py`）针对 4 阶结构进行了专门优化，自动选择 `einsum` / `tensordot` 等最佳路径；
- 使用 `@torch.jit.script` 加速高频收缩与循环计算，减少内存访问开销。

### 自动微分与求解器架构

- 依托 PyTorch 自动微分功能实现流方程右端的动态构造；

- 支持多种数值积分方法（如固定步长、Runge-Kutta、误差控制等），并集成 **自适应步长控制机制**；
- 动态监控误差变化以判断演化是否收敛。

## 性能与扩展性设计

- 所有计算模块均支持 **复数计算** 与 **高精度（如 float64）** 切换；
- 利用批处理与流水并行策略显著提升吞吐；
- 整体架构模块化、易于扩展，例如替换生成元策略或引入新型交互项。

---

## 总体优势

- **GPU 高效加速**：适合大规模量子系统；
- **张量计算优化**：面向结构稀疏、秩较低问题的收缩策略；
- **灵活数值方案**：可根据问题特点调整微分与误差策略；
- **良好可扩展性**：便于嵌入其他量子算法模块或用于不同哈密顿量结构。

P11&P12

## 量子动力学中的 GPU 优化策略说明

为了提升量子动力学模拟，尤其是在 CUT 流方程演化过程中的数值效率，代码实现中引入了多种 **GPU 加速与内存优化技术**：

### 动态演化优化

在 `flow_dyn_int` 中，系统观测量（如粒子数分布）随流动参数  $l$  的演化被动态更新。核心张量计算通过 `contract(..., method='jit')` 实现，该函数底层使用 TorchScript JIT 编译器对张量收缩函数进行优化，加速循环迭代中的重复操作，显著减少 CPU-GPU 通信瓶颈。

## 本征值计算优化

在 `flow_levels` 中，系统哈密顿量  $H$  被送入 GPU 上的 `torch.linalg.eigvalsh()` 进行本征值求解，利用了高效的 CUDA 实现的大型 Hermitian 矩阵对角化流程，加速能谱提取步骤。将张量预先移动到 device 上避免了不必要的数据搬移开销。

## 张量网络收缩优化

在 `con_vec42_comp` 函数中，采用 `@torch.jit.script` 对张量收缩逻辑进行 JIT 编译，优化了形如  $C_{ijklm} = A_{ijkl} \cdot B_{lm}$  的操作。通过 einsum 实现的多索引 contraction 支持 GPU 并行计算，并避免了显式的中间变量存储，提高内存带宽利用率。

## 内存管理优化

在 `optimize_memory()` 函数中，合理管理 CUDA 内存使用：主动清理缓存、启用 `pin_memory=True` 固定页锁内存、使用 CUDA stream 进行异步拷贝操作，从而提升了数据传输效率并减少 kernel 启动等待时间，尤其适用于大规模模拟任务。

# P13

以上是现有的 CUT 方法的gpu加速的一些方式，但这个实现是比较粗糙的。一个问题是，主流的 CUT 方法虽然在逻辑上采用了稀疏矩阵的方式，但在实际存储的时候，依然是全量存储。这导致了内存的大量浪费，从而使得模拟无法在更多粒子数的系统上进行。

# 缺乏结构压缩的详细分析（答辩准备）

## 1. 理论结构上的稀疏性

- CUT 中的哈密顿量  $H(l)$  通常为局域相互作用的和，具有天然的稀疏性：
  - 一维系统中，多体项受限于局域范围；
  - 二维系统中虽然连通性上升，但仍受限于物理模型（例如最近邻、弱耦合）；
- 实际上：
  - 在  $L \sim 50$  的一维链上，活跃 Pauli 项的数量仅占全展开空间 ( $4^L$ ) 的一小部分；
  - 初始扰动和流动过程中，系数演化受限于初始耦合结构，不会激活远离对角线的项。

## 2. 实现上的资源浪费

- 尽管哈密顿量在数学上是稀疏的，但在 GPU 实现中，作者使用了**密集张量**结构：
  - 每一项都对应一个完整的 Pauli 字符串与浮点数系数；
  - 没有采用如 COO、CSR、TT 等结构压缩形式；
  - 导致显存消耗和数据访问开销显著增加。

## 3. 潜在优化方向

- 若能结合：
  - 稀疏表示（如 COO 或稀疏哈希映射）；
  - 稀疏乘法优化；
  - 或低秩张量分解（如 Tensor Train）；
- 将显著降低存储压力，提升 GPU 并行效率。

## CUT 中压缩策略的深入解析

### 1. 为什么需要压缩？

- 四阶张量项  $H_{ijkq}^{(4)}$  在  $L$  点系统中规模为  $\mathcal{O}(L^4)$ ；
  - CUT 流动过程需持续更新这些项；
  - 不压缩的张量表示在 GPU 或 CPU 上都难以支撑较大系统；
  - 因此需要结构压缩或变换策略。
- 

### 2. Tensor Train (TT) 分解

- 将高阶张量重构为多个三阶张量的链式乘积：

$$H(l) \approx \sum_{i_1, \dots, i_L} G_{i_1}^{(1)} G_{i_2}^{(2)} \dots G_{i_L}^{(L)} \quad (20)$$

- 存储成本降低为  $\mathcal{O}(LD^2)$ ，支持 GPU contraction；
  - 存在截断误差，需要调节 bond dimension 以平衡效率与精度；
  - 实验中误差通常控制在 5% 以内。
- 



### 3. Low-Rank 分解

- 将高阶张量 reshape 为矩阵，做秩分解（如 SVD）；
- 初始压缩率高，但 CUT 更新流动中秩常常增长；
- 难以稳定控制秩，误差迅速积累；
- 可用于短时间或近似模拟。



---

## 5. 矩阵分块 (Block Structure Exploitation)

- 若系统具有守恒量或对称性，可将哈密顿量划分为块对角或稀疏带状结构；
- 例如：
  - 粒子数守恒  $\rightarrow$  不同粒子数 sector 可分块；
  - 自旋守恒  $\rightarrow S_z$  保持下的子空间分块；
- 每一块独立计算，可显著减少冗余自由度；
- 在 ED 与张量网络中广泛应用，但当前 CUT 实现未显式利用此策略；
-  精确，无误差；
-  实现需结合对称性识别与张量再构造。

## P15

另外针对这个问题，还可以利用量子领域的惯用方式，使用 pauli 基来展开字符串，达到减少内存占用的目的。

连续酉变换 (CUT) 方法中，哈密顿量随流动参数  $l$  演化：

$$\frac{dH(l)}{dl} = [\eta(l), H(l)] \quad (21)$$

我们将  $H(l)$  表示为 Pauli 字符串的线性组合：

$$H(l) = \sum_{\alpha} c_{\alpha}(l) P_{\alpha} \quad (22)$$

- 每个  $P_{\alpha}$  是一个  $L$  位的 Pauli 字符串 (如  $\sigma_1^z \sigma_3^x$ ) ；
- $c_{\alpha}(l)$  是该字符串的系数，随  $l$  演化；
- 表达结构稀疏，适合 GPU 并行操作。

# 为什么使用 PAULI 张量展开？

- 任意量子多体哈密顿量都可以展开为 Pauli 字符串基底上的线性组合；
- Pauli 字符串构成  $2^L$  维希尔伯特空间的完备正交基；
- 多数字符串中包含大量单位算符  $I$ ，使得整体表达**稀疏可压缩**；
- 非常适合符号计算、张量表示以及 GPU 并行执行。

示例：

```
# 一个 Pauli 字符串示例：Z⊗I⊗X⊗I
pauli_string = ['Z', 'I', 'X', 'I']
c_alpha = 0.135
```

## 补充

### 在 GPU 上的表示与并行

- 每个 Pauli 项  $P_\alpha$  可以用长度为  $L$  的数组编码（如字符数组或 2-bit 编码）；
- 所有  $c_\alpha$  组成一个浮点数组，作为 CUT 中的动态变量；
- CUT 的流动方程写作：

$$\frac{dH}{dl} = [\eta, H] = \sum_{\mu, \nu} c_\mu c_\nu [P_\mu, P_\nu] \quad (23)$$

- 每个线程可独立处理一对  $(P_\mu, P_\nu)$  的对易子计算：

$$[P_\mu, P_\nu] = 2if_{\mu\nu} \cdot P_\lambda \quad (24)$$

其中  $f_{\mu\nu} = \pm 1, \pm i$  是 Pauli 对易规则所决定的对易系数。

### ◆ 3. 数值实现细节与优化策略

- **查表优化**：预构建对易规则  $\text{lookup}[(P_{\mu}, P_{\nu})] \rightarrow (P_{\lambda}, f_{\{\mu\nu\}})$ ；
- **字符串哈希**：将 Pauli 字符串哈希为整数，以便快速比较与合并；
- **GPU reduction**：并行计算后使用归约操作合并重复项；
- **RK4 整步法**：数值推进  $c_{\alpha}(l)$  的演化，提高精度与稳定性。

## ✓ CUT 每步迭代伪代码：

```
for (μ, Pμ) in pauli_list:
    for (ν, Pν) in pauli_list:
        Pλ, coeff = pauli_commutator(Pμ, Pν)
        dc[Pλ] += coeff * c[μ] * c[ν]
```

## Pauli 字符串如何作用于 CUT 方法

在连续酉变换（CUT, Continuous Unitary Transformations）方法中，我们追踪哈密顿量  $H(l)$  的演化：

$$\frac{dH(l)}{dl} = [\eta(l), H(l)] \quad (25)$$

为了高效计算这个对易子，我们将  $H(l)$  和  $\eta(l)$  都展开为 Pauli 字符串的线性组合：

$$H(l) = \sum_{\alpha} c_{\alpha}(l) P_{\alpha} \quad \eta(l) = \sum_{\beta} g_{\beta}(l) P_{\beta} \quad (26)$$

其中：

- 每个  $P_{\alpha}, P_{\beta}$  是长度为  $L$  的 Pauli 字符串；
- $c_{\alpha}(l)$  与  $g_{\beta}(l)$  是随流动参数  $l$  变化的实系数；
- Pauli 字符串构成  $2^L \times 2^L$  哈密顿量空间的完备正交基。

## ◆ 对易子的展开形式

我们要计算：

$$\frac{dH(l)}{dl} = [\eta(l), H(l)] = \left[ \sum_{\beta} g_{\beta} P_{\beta}, \sum_{\alpha} c_{\alpha} P_{\alpha} \right] \quad (27)$$

展开为：

$$\sum_{\alpha, \beta} g_{\beta} c_{\alpha} \cdot [P_{\beta}, P_{\alpha}] \quad (28)$$

## ◆ Pauli 字符串的对易性质

对于单量子比特，Pauli 矩阵满足：

$$[\sigma^a, \sigma^b] = 2i\epsilon^{abc}\sigma^c \quad (29)$$

对于多比特 Pauli 字符串  $P_{\alpha} = \sigma^{a_1} \otimes \cdots \otimes \sigma^{a_L}$ ，可以逐位计算对易性：

- 若在某一位  $\sigma^{a_j}, \sigma^{b_j}$  反对易（如  $X$  与  $Z$ ），则整体贡献非零；
- 对易结果为新的 Pauli 字符串  $P_{\lambda}$ ，并带有因子  $2if_{\mu\nu}$ ；
- 如果全部位置对易，则对易子为 0。

因此：

$$[P_{\mu}, P_{\nu}] = 2if_{\mu\nu} \cdot P_{\lambda} \quad \text{或} \quad 0 \quad (30)$$

## 在 CUT 中的数值演化流程

1. 遍历所有非零项  $P_{\mu}, P_{\nu}$ ；
2. 计算对易子  $[P_{\mu}, P_{\nu}]$ ，得新项  $P_{\lambda}$ ；
3. 累积导数项：

$$\frac{dc_{\lambda}}{dl} + = 2if_{\mu\nu} \cdot g_{\mu} c_{\nu} \quad (31)$$

4. 对  $c_{\alpha}(l)$  使用数值 ODE 方法推进（如 RK4）。

# P17&P18

除了上面的稳定点问题外，传统 CUT 方法中的一个瓶颈是 —— 物理量的计算需要变换回原始基底：

$$\langle \psi | U^\dagger(l) O(l) U(l) | \psi \rangle \quad (32)$$

由于  $U(l)$  是  $2^L \times 2^L$  的大矩阵，对大系统来说计算代价极高。

为此提出一种改进方案：

- 直接在 CUT 的对角基中进行采样；
- 本征态是张量积态，形式简单；
- 可用随机采样近似期望值：

$$\langle O \rangle \approx \frac{1}{\mathcal{N}_s} \sum_{n=1}^{\mathcal{N}_s} \langle E_n | O | E_n \rangle \quad (33)$$

- 避免还原，显著降低成本，尤其适合高温系统。

## CUT 方法的数值瓶颈

### 背景

- CUT 将哈密顿量与算符变换到对角形式：
  - $H(l) \rightarrow H_{\text{diag}}$ ,  $O(l)$  为演化算符
- 物理量仍需在原始基底上计算：
  - $O(t) = U^\dagger(l) O(l) U(l)$

### 数值困难分析

- $U(l)$  是作用在 Hilbert 空间的幺正变换，维度为  $2^L \times 2^L$ ；
- 为了计算物理量，需执行以下操作：

$$\langle \psi | U^\dagger(l) O(l) U(l) | \psi \rangle \quad (34)$$

- 即使  $O(l)$  是对角表示，也需要将其变回原始表示，代价为  $\mathcal{O}(2^{3L})$ ；
  - 大系统下内存和乘法开销巨大。
- 

## 在对角基中直接计算观测量

- 在 CUT 过程中， $O(l)$  与  $H(l)$  一同变换到对角基；
- $H(l) \rightarrow H_{\text{diag}}$ ，其本征态  $|E_n\rangle$  是张量积态，结构简单；
- 可在对角基中估算物理量，避免显式使用  $U(l)$ ：

$$\langle O \rangle = \frac{1}{\mathcal{N}_s} \sum_{n=1}^{\mathcal{N}_s} \langle E_n | O | E_n \rangle \quad (35)$$

- 当  $\mathcal{N}_s \ll 2^L$  时可有效近似  $\text{Tr}(O)$  或热平均；
  - 本征态为占据数表述，例如  $|E_n\rangle = |0\rangle \otimes |1\rangle \otimes \dots$ ，作用规则明确。
- 

## 什么是算符？

### 概念解释

在量子力学中，**算符**是一种作用在量子态上的线性映射，用于表示可观测量或演化过程。

例如：

- 动量算符： $\hat{p} = -i\hbar \nabla$

- 粒子数算符： $n_i = c_i^\dagger c_i$
- 哈密顿量： $H = \sum t_{ij} c_i^\dagger c_j + \dots$

## 多体系统中的基本算符

名称	表达式	物理意义
产生算符	$c_i^\dagger$	在格点 $i$ 上创建一个粒子
湮灭算符	$c_i$	在格点 $i$ 上湮灭一个粒子
数目算符	$n_i = c_i^\dagger c_i$	计数格点 $i$ 上是否有粒子
跳跃项	$c_i^\dagger c_j$	粒子从 $j$ 跃迁到 $i$
相互作用项	$c_i^\dagger c_j c_k^\dagger c_l$	粒子间相互作用或交换过程

## 在 CUT 中的三类算符

角色	符号	说明
哈密顿量	$H(l)$	系统总能量，目标是对角化它
生成元	$\eta(l)$	控制系统如何流动（变化）
观测算符	$O(l)$	被观测的物理量，需同步变换

## 算符如何作用在量子态上？

量子态常以 Fock 基底表示，例如：

- $|\psi\rangle = |1, 0, 1, 0\rangle = c_1^\dagger c_3^\dagger |0\rangle$

费米子算符满足反对易关系：

- $c_i^\dagger |0\rangle = |i\rangle$
- $c_i |i\rangle = |0\rangle$
- $\{c_i, c_j\} = \{c_i^\dagger, c_j^\dagger\} = 0, \{c_i, c_j^\dagger\} = \delta_{ij}$

# 总结

- CUT 中所有对象 ( $H, \eta, O$ ) 都是**算符表达式**, 由基本的  $c^\dagger, c$  组成 ;
- 每个算符可以视为一种“对粒子的操作规则” ;
- 算符结构可通过表达式管理 (表达式级 CUT) , 也可构造为矩阵 (矩阵级 CUT) ;
- 不论表达形式如何, 它本质是 Hilbert 空间上的线性映射。

P18

P19

## 图片展示

展示了不同类型无序系统下, 无限温度关联函数  $C(t) = \langle O(t)O(0) \rangle_{\beta=0}$  随时间的变化。  
该图间接地验证了 CUT 与 ED 结果在一定时间尺度内的一致性 :

灰色点划线 表示 CUT 方法可靠性的时间上限 (称为 cutoff time) ;

在 cutoff 前, CUT 与 ED 对  $C(t)$  的计算结果非常吻合 ;

即便超过该时间段, CUT 的结果仍在物理合理范围内, 没有数值不稳定或误差爆炸。

这一观察说明 :



CUT 方法可以在中短时间尺度内稳定运行，并精确复现 ED 所预测的量子关联行为。

## P20

### 1. 数据传输瓶颈分析

#### 1.1 系数剪枝操作

当前在 `flow_static_int_torch` 中，系统需对张量系数  $c_\alpha$  进行动态筛选，剔除小于阈值的冗余项。由于剪枝逻辑定义在 CPU 侧，GPU 中的张量需在每轮演化结束后全量传回 CPU 进行判断，这造成了大量的 PCIe 通信负担。

同时，剪枝函数 `event()` 每一轮都触发，进一步加剧数据传输压力。由于 GPU 无法原地变更张量结构，导致传输开销不可避免。

#### 1.2 观测量计算传输

在 `flow_dyn_int` 函数中，为了记录演化过程中的物理量（如粒子数密度或相关函数），每个时间步都需从 GPU 中读取观测值并输出至 CPU。这种实时输出打断了 GPU 的计算图执行，降低了并行效率，属于典型的“同步读回”瓶颈。

建议改进方式包括使用 pinned memory + 异步复制（`cudaMemcpyAsync`），或采用延迟计算策略集中输出。

### 2. 计算结构与内存瓶颈

#### 2.1 张量结构更新

在非局域 CUT 流方程 (`int_ode_nloc`) 中，Pauli 字符串张量项在每轮迭代中不断生成。由于张量合并、去重操作依赖哈希结构，且 Python/GPU 缺乏灵活集合类型，目前只能在 CPU 侧完成结构维护。

这意味着：每轮都需将新生成的 Pauli 项传回 CPU、完成合并、再拷回 GPU，进一步拖慢流程。

## 2.2 内存管理问题

频繁的张量生成导致 GPU allocator 压力极大。大量中间张量（如  $[\eta, H]$  的每一项）未能复用，造成内存碎片与溢出风险。当前 PyTorch 缺乏对 CUT 模型的内存优化策略，造成显著浪费。

建议引入 tensor pooling 机制，缓解 allocator 频繁分配与释放的问题，并对典型张量形状启用手动缓存。

## 3. 控制流中断与同步问题

### 3.1 调试过程打断 GPU 流水线

调试 CUT 动力学通常需对中间结果（如  $H(l)$ 、 $c_\alpha$  分布）进行跟踪。目前主要采用 `print()` 或状态回传，频繁干扰 GPU 内核执行顺序。

缺乏有效的 GPU 内调试机制，尤其是无法在不打断主流图执行的前提下获取中间值，导致调试成本上升，整体吞吐率下降。

### 3.2 强同步点阻塞

如在 `diag.py` 中，多个位置调用 `torch.cuda.synchronize()` 或隐式同步逻辑（如 `torch.linalg.eigvalsh(H)`），迫使 GPU 所有任务完成后才进入下一阶段。

这些“强 barrier”破坏了异步流水线和数据并行性，是 GPU 利用率下降的另一关键因素。

## 4. 并行粒度与调度瓶颈

### 4.1 并行粒度不均衡

不同计算阶段的操作密集度差异大。比如张量 contraction 具备高度并行性，而张量筛选、哈希合并等操作粒度低且不可并发，造成调度瓶颈。

建议采用 warp-level 或 block-level dynamic kernel 分发，对不同 workload 动态调整调度策略。

## 4.2 动态结构演化

CUT 中的哈密顿量结构是随流时间演化不断变化的，新的 Pauli 项会在不确定时间产生。系统难以预估下一步张量大小和形状，导致资源预分配失败或浪费。

这一动态特性导致 CUT GPU 化较为困难，需结合 sparse tensor 架构和 lazy construction 来改善适配性。

# 可能的解决方案

## 1. 减少 GPU ↔ CPU 数据传输

问题：剪枝判断与观测量输出发生在 CPU 上，导致频繁的数据传输。

解决方案：

✅ 在 GPU 上完成剪枝判断：

将  $|c_\alpha| < \varepsilon$  的判断逻辑写成 CUDA kernel（或 PyTorch 自定义 CUDA Extension），结合 mask 筛选；

使用 `torch.masked_select()` 或稀疏张量进行就地更新；

避免数据在每轮流动后从 GPU 拷回 CPU。

✅ 观测量计算异步输出：

改用 `torch.cuda.stream()` + pinned memory + `torch.asynchronous copy` 实现观测量的延迟传输；

将多个时间点观测量合并打包后统一传输；

进一步延迟观测量输出至演化结束后统一统计。

✅ GPU 上的哈希索引合并：

使用 CUB 或 Thrust 实现 GPU 上的哈希表或稀疏映射；

用定长编码（如将 Pauli 字符串编码为  $2\text{-bit} \times L$  的整数）作为键进行快速匹配；

配合 segment reduction 合并同类项。

✅ 采用稀疏张量结构表示 CUT 哈密顿量：

PyTorch 支持的 `torch.sparse_csr_tensor` 或 `torch.sparse_coo_tensor` 可用于稀疏 Pauli 项存储；

避免大规模全连接张量的冗余存储；

更适合动态扩展。