项目信息：

基于昇思 MindQuantum，对已有的梯度计算模块实现加速

项目编号：221cb0173

导师：Tree xushu91@mail.ustc.edu.cn

学生：仰宗焱 i@YangZongYan.com

方案描述：

量子机器学习的常用方法是通过计算目标哈密顿量期望对含参数量子线路中各参数的导数信息从而更新量子线路参数，使得模型获得更好的精确度/准确度。量子机器学习中，一般训练集中包含许多样本数据。如何利用多核处理器并行的计算不同样本哈密顿量期望对参数的导数是个重要问题。本任务要求基于 MindQuantum 现有的梯度计算功能，提升梯度计算的性能 1 倍以上。

时间规划：

7.1——7.20　熟悉相关理论、查询相关资料

7.21——8.10 分析程序结构

8.11——8.20 尝试分离梯度算法模块

8.21——9.20 对梯度算法模块进行优化

9.20——9.30 撰写报告

项目进度：

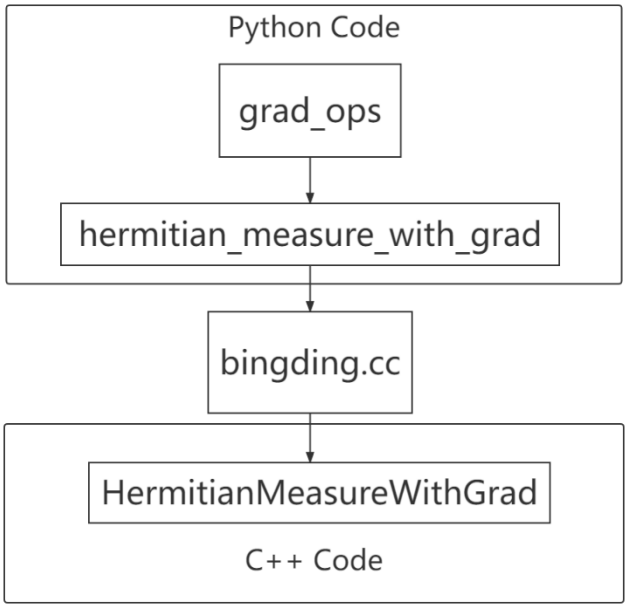
已完成的工作：

昇思 MindQuantum 中目前使用的梯度计算模块，是通过多线程的方式实现对梯度计算的加速。然而，计算模块在每次进行函数调用的时候创建线程，等函数调用结束之后线程就被销毁，大量的线程资源申请和删除造成了比较高的性能浪费。

考虑到线程创建和销毁的频繁性，针对的改进思路为，在对应模块中实现一个线程池。线程重复查询任务队列，求梯度任务进入线程池后进入等待队列，轮到任务后开始计算，计算结束后对应线程继续重复查询。

目前的梯度调用模块为 grad_ops。

函数的调用如下图：



可以看出，整体函数优化的重点在于 HermitianMeasureWithGrad 函数。该函数多次被调用，频繁地进行线程的创建和删除，消耗了大量的时间。对实验优化效果的评价我们也以该函数的运行时间的变化为评价标准。

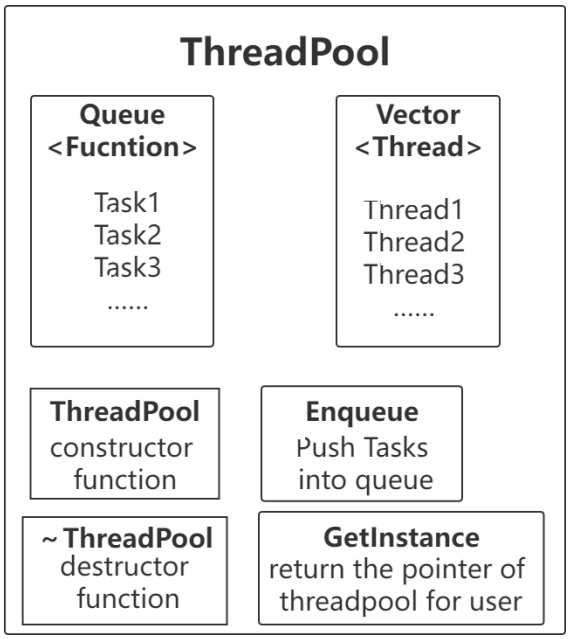
局部代码分析:

其中，HermitianMeasureWithGrad 核心的代码为多线程对梯度求导(mindquantum/third_party/projectq/projectq.h)，改进后，多线程梯度求导任务将会进入线程池执行，优化前核心代码如下。

```
std::vector<std::thread> tasks;
tasks.reserve(batch_threads);
size_t end = 0;
size_t offset = n_prs / batch_threads;
size_t left = n_prs % batch_threads;
for (size_t i = 0; i < batch_threads; ++i) {
    size_t start = end;
    end = start + offset;
    if (i < left) {
        end += 1;
    }
    auto task = [&, start, end]() {
        for (size_t n = start; n < end; n++) {
            ParameterResolver<T> pr = ParameterRe
            pr.SetItems(enc_name, enc_data[n]);
            pr.SetItems(ans_name, ans_data);
            auto f_g = HermitianMeasureWithGrad(h
            output[n] = f_g;
        }
    };
    tasks.emplace_back(task);
}
for (auto &t : tasks) {
    t.join();
}
```

线程池设计

线程池结构设计如图所示:



从图中可以看出，线程池结构体中，函数的队列用于存储待执行的函数。线程的 Vector 容器中的线程循环查询是否有待执行的函数队列，若有则从队列中取出执行。线程池的构造函数用于线程池的初始化，如线程容器的创建、函数队列的创建、初始变量的设定等。线程池的析构函数用于等待线程池线程的结束并关闭线程池。Enqueue 函数用于接受任务并加入等待的任务队列。GetInstance 函数用于向其他函数提供访问线程池的渠道，其他函数通过 GetInstance 函数得到指向线程池的指针。考虑到线程池的创建和访问涉及到多线程操作，因此线程池将采用饿汉模式创建，在模拟器创建的时候即创建，从而杜绝创建同步冲突的问题。

使用线程池的如下图所示。

```
size_t end = 0;
size_t offset = n_hams / mea_threads;
size_t left = n_hams % mea_threads;
ThreadPool* pp=ThreadPool::getInstance();
for (size_t i = 0; i < mea_threads; ++i) {
    size_t start = end;
    end = start + offset;
    if (i < left) {
        end += 1;
    }
    auto task = [&, start, end]() {
        for (size_t n = start; n < end; n++) {
            auto f_g = sim.RightSizeGrad(sim.vec_, sim.vec_, hams[n
            for (size_t g = 1; g < n_params + 1; g++) {
                f_g[g] += std::conj(f_g[g]);
            }
            output[n] = f_g;
        }
    };
    pp->enqueue(task);
```

求梯度运算模块的任务创建好后将送入线程池的等待队列中。

测试机制：

测试操作系统为：Ubuntu 20.04

测试配置为：8 核心处理器 8GB RAM 20GB SSD

测试方法为，引入 chrono 头文件，在 C++函数 HermitianMeasureWithGrad 中添加锚点计算函数的执行时间，比较改进前与改进后的时间，判断性能是否有提升，代码示例如下。

```
auto start = std::chrono::system_clock::now();
//Do Funciton
auto endt = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed = endt-start;
std::cout<<"Elapsed time: "<<elapsed.count()<<" s\n";
```

为了防止实验误差对结果准确性造成的影响，每次测试执行多组实验，多次测试。对比每次、每组测试之间的耗时差异。如下图，每组测试执行 5 次函数。

```
grad_ops = Sim.get_expectation_with_grad(hams, encoder, parallel_worker=6)
#Simulator('projectq', 1).resize_threads(8)
#print(grad_ops1)
#Sim.resize_threads(3)
alpha = np.array([[np.arctan(np.sqrt(2))], [np.pi/2], [np.pi/2], [np.pi/2]])

f, g = grad_ops(alpha)
f, g = grad_ops(alpha)
f, g = grad_ops(alpha)
f, g = grad_ops(alpha)
f, g = grad_ops(alpha)
```

结果表格如下图

实验组	第1组耗时	第2组耗时	第3组耗时	第4组耗时	第5组耗时
1	0.000800342	0.000861477	0.000709215	0.000801551	0.000684
2	0.000396419	0.0003615	0.000414915	0.000441398	0.000522
3	0.000348696	0.00042831	0.000402196	0.000377121	0.000369
4	0.00033498	0.000354087	0.000367803	0.000372173	0.000357
5	0.000333739	0.000388958	0.000354178	0.000362308	0.000441
对照组	第1组耗时	第2组耗时	第3组耗时	第4组耗时	第5组耗时
1	0.00110115	0.000849036	0.00107045	0.000979203	0.000961
2	0.000657845	0.000890805	0.000755049	0.000828071	0.00084
3	0.000618383	0.000745176	0.00062522	0.000749651	0.000766
4	0.000609464	0.000717964	0.00071344	0.000758437	0.000582
5	0.000703002	0.000542021	0.00053533	0.000715901	0.000879

可以看出，实验组每组第一次的耗时约为 0.8 毫秒，对照组（即未优化版本）每组第一次耗时约为 1 毫秒，其提升约为 20%。而每组其后开始，实验组的提升久非常可观了，实验组每组 2—5 次的耗时约为 0.3—0.4 毫秒，对照组每组 2—5 次耗时约为 0.6—0.8 毫秒，其速度提升达到了约一倍，达到了实验的要求。

实验出现这样的结果并不难理解，实验组在第一次调用函数的时候，需要初始化线程池，和对照组同样需要创建线程（但是不需要销毁线程），因此仅有约 20%的性能优化。而再次调用函数，此时实验组可以直接将任务送入线程池而不需要重新创建新的线程和销毁线程，从而实验了一倍的性能提升。

遇到的问题：

问题 1：多线程访问线程池问题

问题描述：线程池涉及到了多线程的访问，如果对进程的访问不加以限制很容易出现冲突。

解决方法：引入 mutex 和条件变量，在进行多线程任务的时候进行上锁处理，防止多线程访问出现错误。

```
std::function<void()> task;

{
    std::unique_lock<std::mutex> lock(this->queue_mutex);
    this->condition.wait(lock,
        [this]{ return this->stop || !this->tasks.empty(); });
    if(this->stop && this->tasks.empty())
```

问题 2：空线程的判断问题

问题描述：线程池的原理是线程无限循环查询任务队列中是否有等待执行的任务，如果有则取出执行，执行结束后回到无限查询状态。因此，对于程序而已，只能判断当前存在 N 个线程，具体其中的线程是不是在执行任务并不知道，这让接下来的代码出现了很多问题。

解决方法：增加一个原子变量 idle，记录目前空进程的数量，当队列中的任务被线程接受执行和任务结束后，修改 idle 的数量，从而实现当前是否有空进程的判断。

```
std::atomic<int> idle{0};
std::queue< std::function<void()>> tasks;
static ThreadPool* getInstance()
```

问题 3：结束的判断问题

问题描述：未优化的代码中，多线程任务全部丢入 Vector 容器中，然后挨个对容器内的线程进行 join 操作，线程执行结束后，资源自动释放。但是在我们的优化的代码中并不能这样操作，因为线程需要在线程池中长期保持，如果使用 join 操作，则执行一个任务后线程释放，接下来对任务就不能执行了。

解决方法：使用到了问题 2 中创建的 idle 原子变量，在所有的任务丢入线程池的任务队列后，循环查询 idle 是否为 0 和线程池的任务队列是否为空。若同时满足，才会接触循环执行下一步的代码。

后续工作安排：

考虑对线程池进行进一步的优化，进一步优化性能降低代码耦合度。