oneAPI使用示例：使用英特尔oneAPI工具加速矩阵乘法

导言

矩阵乘法是许多科学计算和数据处理任务中常见的操作，然而，对于大型矩阵而言，传统的串行方法可能效率低下。英特尔oneAPI工具提供了一种并行计算的解决方案，可以有效地加速矩阵乘法运算。本文将介绍如何使用oneAPI工具来实现矩阵乘法的加速，并给出相应的代码。

oneAPI概述

oneAPI是英特尔提供的一个综合性编程模型和工具集，旨在简化并行计算的开发和优化。它提供了一致的编程接口和工具，使开发者能够在不同的硬件架构上编写高性能并行代码。

实验：使用oneAPI进行加速

为了使用oneAPI工具加速矩阵乘法，我们将使用其中的DPC++编程模型。DPC++是oneAPI工具集中基于C++的的一种编程模型，提供了对并行计算的支持。

对照组：传统矩阵乘法

使用传统的矩阵乘法的时间复杂度是比较高的，作为对照实验，代码如下：

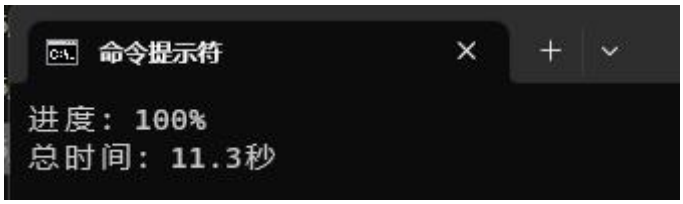
```
#include <chrono>
#include <iostream>
#include <random>
#include <vector>

// 定义矩阵的大小
const int matrix_size = 100;

// 生成随机矩阵
std::vector<std::vector<double>> generateRandomMatrix(int rows, int cols)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dis(0.0, 1.0);

    std::vector<std::vector<double>> matrix(rows, std::vector<double>(cols));
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            matrix[i][j] = dis(gen);
        }
    }

    return matrix;
}

// 计算矩阵乘法
std::vector<std::vector<double>> matrixMultiply(const
std::vector<std::vector<double>> &matrix1, const
std::vector<std::vector<double>> &matrix2)
```

```
{
    int rows1 = matrix1.size();
    int cols1 = matrix1[0].size();
    int cols2 = matrix2[0].size();

    std::vector<std::vector<double>> result(rows1, std::vector<double>(cols2));

    for (int i = 0; i < rows1; ++i) {
        for (int j = 0; j < cols2; ++j) {
            double sum = 0.0;
            for (int k = 0; k < cols1; ++k) {
                sum += matrix1[i][k] * matrix2[k][j];
            }
            result[i][j] = sum;
        }
    }

    return result;
}

int main()
{
    // 生成随机矩阵
    std::vector<std::vector<double>> matrix1 = generateRandomMatrix(matrix_size,
matrix_size);
    std::vector<std::vector<double>> matrix2 = generateRandomMatrix(matrix_size,
matrix_size);

    // 进行1000次随机矩阵乘法并计时
    auto start_time = std::chrono::high_resolution_clock::now();
    int total_iterations = 1000;
    int progress_interval = total_iterations / 100; // 计算进度的显示间隔
    for (int i = 0; i < total_iterations; ++i) {
        std::vector<std::vector<double>> result = matrixMultiply(matrix1,
matrix2);
        if ((i + 1) % progress_interval == 0) {
            // 清屏后显示
            std::cout << "\033[2J\033[1;1H";
            double progress = (static_cast<double>(i + 1) / total_iterations) *
100;
            std::cout << "进度: " << progress << "%" << std::endl;
        }
    }
    auto end_time = std::chrono::high_resolution_clock::now();

    // 计算总时间（以秒为单位）
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>
(end_time - start_time);
    double total_time = duration.count() / 1000.0;

    // 打印结果
    std::cout << "总时间: " << total_time << "秒" << std::endl;

    return 0;
}
```

这段代码实现了一个矩阵乘法的示例，并进行了多次计算和计时。具体流程如下：

1. 首先，通过调用 `generateRandomMatrix()` 函数生成了两个大小为 `matrix_size * matrix_size` 的随机矩阵 `matrix1` 和 `matrix2`。
2. 接下来，通过一个循环进行1000次矩阵乘法计算，并在每次计算完成后输出当前进度。循环中的每一次迭代都调用 `matrixMultiply()` 函数，将 `matrix1` 和 `matrix2` 作为参数传递进去。
3. 在每个计算的进度达到显示间隔时，通过输出特定的控制字符来清屏并显示当前进度百分比。
4. 最后，使用 `std::chrono` 库计算计时，并将总时间以秒为单位打印出来。

最后的结果如下：



实验组：使用oneAPI工具

以下是使用DPC++实现矩阵乘法的示例代码：

```
#include <CL/sycl.hpp>
#include <iostream>
#include <random>
#include <vector>

// 定义矩阵的大小
const int matrix_size = 100;

// 生成随机矩阵
std::vector<std::vector<double>> generateRandomMatrix(int rows, int cols)
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<double> dis(0.0, 1.0);

    std::vector<std::vector<double>> matrix(rows, std::vector<double>(cols));
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            matrix[i][j] = dis(gen);
        }
    }

    return matrix;
}

// 计算矩阵乘法
std::vector<std::vector<double>> matrixMultiply(const
std::vector<std::vector<double>> &matrix1, const
std::vector<std::vector<double>> &matrix2)
{
    int rows1 = matrix1.size();
    int cols1 = matrix1[0].size();
    int cols2 = matrix2[0].size();

    std::vector<std::vector<double>> result(rows1, std::vector<double>(cols2));
```

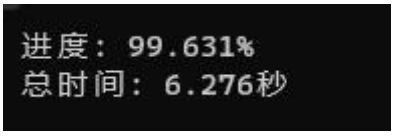
```
    // 使用 SYCL 编程模型
    {
        // 创建 SYCL 的队列和设备选择器
        sycl::queue queue(sycl::gpu_selector_v);

        // 创建输入和输出缓冲区

        cl::sycl::buffer<double, 2> buffer1(matrix1.data()->data(),
cl::sycl::range<2>(rows1, cols1));
        cl::sycl::buffer<double, 2> buffer2(matrix2.data()->data(),
cl::sycl::range<2>(cols1, cols2));
        cl::sycl::buffer<double, 2> buffer_result(result.data()->data(),
cl::sycl::range<2>(rows1, cols2));

        // 提交 SYCL 的计算任务
        queue.submit([&](cl::sycl::handler &cgh) {
            auto accessor1 = buffer1.get_access<cl::sycl::access::mode::read>
(cgh);
            auto accessor2 = buffer2.get_access<cl::sycl::access::mode::read>
(cgh);
            auto accessor_result =
buffer_result.get_access<cl::sycl::access::mode::write>(cgh);

            cgh.parallel_for<class MatrixMultiplyKernel>(cl::sycl::range<2>
(rows1, cols2), [=](cl::sycl::item<2> item) {
                int i = item[0];
                int j = item[1];
                double sum = 0.0;
                for (int k = 0; k < cols1; ++k) {
                    sum += accessor1[i][k] * accessor2[k][j];
                }
                accessor_result[i][j] = sum;
            });
        });

        // 同步等待计算任务完成
        queue.wait_and_throw();
    }

    return result;
}

int main()
{
    // 生成随机矩阵
    std::vector<std::vector<double>> matrix1 = generateRandomMatrix(matrix_size,
matrix_size);
    std::vector<std::vector<double>> matrix2 = generateRandomMatrix(matrix_size,
matrix_size);

    // 进行1000次随机矩阵乘法并计时
    auto start_time = std::chrono::high_resolution_clock::now();
    int total_iterations = 1000;
    int progress_interval = total_iterations / 100; // 计算进度的显示间隔
    for (int i = 0; i < total_iterations; ++i) {
        std::vector<std::vector<double>> result = matrixMultiply(matrix1,
matrix2);
        if ((i + 1) % progress_interval == 0) {
```

```
            // 清屏后显示
            std::cout << "\033[2J\033[1;1H";
            double progress = (static_cast<double>(i + 1) / total_iterations) *
100;
            std::cout << "进度: " << progress << "%" << std::endl;
        }
    }
    auto end_time = std::chrono::high_resolution_clock::now();

    // 计算总时间（以秒为单位）
    auto duration = std::chrono::duration_cast<std::chrono::milliseconds>
(end_time - start_time);
    double total_time = duration.count() / 1000.0;

    // 打印结果
    std::cout << "总时间: " << total_time << "秒" << std::endl;

    return 0;
}
```

在上述代码中，我们使用了`queue`来创建一个执行队列。

接下来，我们使用`sycl::handler`和`sycl::parallel_for`来创建并行执行的内核函数。在内核函数中，我们使用并行迭代遍历矩阵的每个元素，并使用累加求和的方式计算矩阵乘法的结果。起到了通过并行来加速的效果。

最后的结果如下：



总结

通过使用英特尔oneAPI工具中的DPC++编程模型，我们可以实现矩阵乘法的加速计算。oneAPI提供了一致的编程接口和工具，使开发者能够方便地进行并行计算的开发和优化。通过合理利用并行计算资源，我们可以显著提高矩阵乘法的计算性能，从而加速科学计算和数据处理任务的执行。