# 用oneAPI加速矩阵乘法

## 概述

加速矩阵乘法是高性能计算中的一个常见需求。为了实现矩阵乘法的加速，我们可以利用英特尔的oneAPI工具集，其中包括了SYCL编程模型和DPC++语言，使我们能够在不同类型的加速器上实现并行计算。

## 使用oneAPI实现矩阵乘法

#### 矩阵乘法的加速核心函数

定义一个名为 MatrixMultiplyKernel 的类，用于表示矩阵相乘的内核。该类具有一个构造函数，接受三个 sycl::accessor 对象作为参数，用于指定输入矩阵A、B和输出矩阵C的访问方式。类中重载了圆括号运算符 operator()，用于定义内核函数的实现。

class MatrixMultiplyKernel {

public:

MatrixMultiplyKernel(sycl::accessor<float, 2, sycl::access::mode::read> accessorA,

sycl::accessor<float, 2, sycl::access::mode::read> accessorB,

sycl::accessor<float, 2, sycl::access::mode::write> accessorC)

: accessorA(accessorA), accessorB(accessorB), accessorC(accessorC) {}

void operator()(sycl::id<2> idx) {

float sum = 0.0f;

for (int k = 0; k < N; ++k) {

sum += accessorA[idx[0]][k] \* accessorB[k][idx[1]];

}

accessorC[idx] = sum;

}

private:

sycl::accessor<float, 2, sycl::access::mode::read> accessorA;

sycl::accessor<float, 2, sycl::access::mode::read> accessorB;

sycl::accessor<float, 2, sycl::access::mode::write> accessorC;

};

#### 初始化矩阵

在main 函数中，首先定义了三个向量 matrixA、matrixB 和 matrixC，分别用于存储输入矩阵A、B和输出矩阵C的数据，矩阵大小为N\*N（N=1024）。

int main() {

std::vector<float> matrixA(N \* N, 2.0f);

std::vector<float> matrixB(N \* N, 3.0f);

std::vector<float> matrixC(N \* N, 0.0f);

//. . .

}

#### 加速矩阵乘法

加速矩阵乘法的关键在于使用了SYCL编程模型，并利用了设备的并行计算能力，具体步骤如下：

1. 创建了一个SYCL队列，并选择默认的设备。SYCL队列是一个任务队列，可以在其中提交计算任务。
2. 创建了SYCL缓冲区 bufferA、bufferB 和 bufferC，用于在主机和设备之间传输矩阵数据。这些缓冲区可以被访问器访问，访问器定义了对数据的访问权限和模式。
3. 在使用 myQueue.submit 提交的 lambda 函数中，使用 buffer.get\_access 方法获取矩阵 A、B 和 C 的访问器。这些访问器指定了访问模式，A、B矩阵只读，C矩阵只写。
4. 使用 cgh.parallel\_for 并传递矩阵大小和一个 lambda 函数作为内核函数，用于并行执行矩阵相乘的计算。parallel\_for 函数指定了要执行的计算范围，并在设备上启动多个工作项来执行计算。在每个工作项中，创建了一个 MatrixMultiplyKernel 类对象，并调用该对象的 operator() 函数进行计算。
5. 调用 myQueue.wait 等待队列中的任务完成，确保计算已经完成。

通过使用SYCL编程模型，可以将计算任务提交给设备进行并行计算，从而加速矩阵的乘法计算。设备可以同时处理多个元素，利用其并行计算能力，可以在较短的时间内完成矩阵乘法的计算。

try {

sycl::queue myQueue(sycl::default\_selector\_v);

sycl::range<2> size(N, N);

sycl::buffer<float, 2> bufferA(matrixA.data(), size);

sycl::buffer<float, 2> bufferB(matrixB.data(), size);

sycl::buffer<float, 2> bufferC(matrixC.data(), size);

myQueue.submit([&](sycl::handler& cgh) {

auto accessorA = bufferA.get\_access<sycl::access::mode::read>(cgh);

auto accessorB = bufferB.get\_access<sycl::access::mode::read>(cgh);

auto accessorC = bufferC.get\_access<sycl::access::mode::write>(cgh);

cgh.parallel\_for<class MatrixMultiply>(size, [=](sycl::id<2> idx) {

MatrixMultiplyKernel kernel(accessorA, accessorB, accessorC);

kernel(idx);

});

});

myQueue.wait();

} catch (sycl::exception const& e) {

std::cerr << "An exception occurred: " << e.what() << std::endl;

return 1;

}

## 运行环境

本项目是在Intel DevCloud 平台上编译运行，免安装本地环境即可进行代码的开发和测试，Kernel为Python3(Intel oneAPI 2023.1)。

## 总结

使用英特尔oneAPI工具集加速矩阵乘法的关键步骤包括准备数据和缓冲区、定义和执行加速核心函数以及将结果从设备内存复制回主机内存。SYCL和DPC++提供了强大的编程模型和语言特性，使我们能够在不同类型的加速器上实现高性能的并行计算。

## 代码

#include <CL/sycl.hpp>

#include <iostream>

constexpr size\_t N = 1024;

class MatrixMultiplyKernel {

public:

MatrixMultiplyKernel(sycl::accessor<float, 2, sycl::access::mode::read> accessorA,

sycl::accessor<float, 2, sycl::access::mode::read> accessorB,

sycl::accessor<float, 2, sycl::access::mode::write> accessorC)

: accessorA(accessorA), accessorB(accessorB), accessorC(accessorC) {}

void operator()(sycl::id<2> idx) {

float sum = 0.0f;

for (int k = 0; k < N; ++k) {

sum += accessorA[idx[0]][k] \* accessorB[k][idx[1]];

}

accessorC[idx] = sum;

}

private:

sycl::accessor<float, 2, sycl::access::mode::read> accessorA;

sycl::accessor<float, 2, sycl::access::mode::read> accessorB;

sycl::accessor<float, 2, sycl::access::mode::write> accessorC;

};

int main() {

std::vector<float> matrixA(N \* N, 2.0f);

std::vector<float> matrixB(N \* N, 3.0f);

std::vector<float> matrixC(N \* N, 0.0f);

try {

sycl::queue myQueue(sycl::default\_selector\_v);

sycl::range<2> size(N, N);

sycl::buffer<float, 2> bufferA(matrixA.data(), size);

sycl::buffer<float, 2> bufferB(matrixB.data(), size);

sycl::buffer<float, 2> bufferC(matrixC.data(), size);

myQueue.submit([&](sycl::handler& cgh) {

auto accessorA = bufferA.get\_access<sycl::access::mode::read>(cgh);

auto accessorB = bufferB.get\_access<sycl::access::mode::read>(cgh);

auto accessorC = bufferC.get\_access<sycl::access::mode::write>(cgh);

cgh.parallel\_for<class MatrixMultiply>(size, [=](sycl::id<2> idx) {

MatrixMultiplyKernel kernel(accessorA, accessorB, accessorC);

kernel(idx);

});

});

myQueue.wait();

} catch (sycl::exception const& e) {

std::cerr << "An exception occurred: " << e.what() << std::endl;

return 1;

}

// 打印结果

for (size\_t i = 0; i < N; ++i) {

for (size\_t j = 0; j < N; ++j) {

std::cout << matrixC[i \* N + j] << " ";

}

std::cout << std::endl;

}

return 0;

}