基于 oneAPI 实现 Seam Carving 算法

一、 算法简介

Seam Carving 算法是经典的图片压缩算法，基本的思想为动态规划。这种算法考虑沿着图片的宽来进行压缩，实质为删去了若干条像素路径。在日常应用中，为了防止图片的失真，即保持图片在基本信息上的平滑，我们需要选择“能量”总和最小的一条路径进行删除，每个像素点“能量“的定义根据实际应用场景不同而有不同的定义，一般代表着去除后对原图片的影响程度。而每条路径的总能量即为该条路径所经过的像素点的能量总和。为了找出这条能量最小的路径，我们可以先计算出每个像素点的能量，而后将其转化为寻找最短路径问题，此时便可以使用动态规划算法计算出到达某个像素点时的最小能量值，生成一幅能量图，最后我们再根据这幅能量图去寻找最短路径，进行对应的删除。重复上述过程即可将图片压缩至任意大小。本次我们将借助 oneAPI 中的 SYCL 编程模型来实现 Seam Carving 算法。

二、 算法实现介绍

1. 像素点能量计算

calculateEnergy 函数负责计算每个像素点的对应能量，该函数接受一个像素值和相邻像素值的向量作为输入，计算出像素的能量值。

在实现中，我们使用差值的平方和作为能量值的计算方式。首先，通过迭代相邻像素的向量，计算像素与相邻像素之间的差值，并将差值的平方累加到能量值上。最后，返回计算得到的能量值。

2. 动态规划生成能量图

computeEnergyMap 函数负责使用动态规划方法生成能量图，该函数接受的参数为输入图片的信息数组、能量图数组的引用、图片的宽和高。该函数会遍历输入图像的每个像素，并调用 calculateEnergy 函数来计算能量值。相邻像素的数值通过访问输入图像数据来获取。而后再次遍历所有像素点，用动态规划算法找出相邻像素点到达当前像素点所用最小能量，将这个最小能量值填入能量图中。最后，将得到的能量图以引用方式返回。

3. 移除能量最小路径

removeMinEnergyPath 函数用于移除最小能量路径。该函数参数为图像数组引用、能量图引用、图像的宽度和高度。该函数将根据能量图的相关数据找出最小能量路径。

在计算好的能量图中，每个元素表示从最顶部到该位置的路径的最小能量累计值。因此我们可以先从第一行开始找到能量最小路径的起始位置，然后通过移动像素的方式来移除能量最小路径。具体操作是将能量最小路径上方的像素依次向左或向右移动一位，然后将能量最小路径上方的像素复制到能量最小路径上。这样就完成了能量最小路径的移除。最后，移除最上方的像素，即移除最后一行中的像素点。

4. SYCL 库的优化主函数的调用

在主函数中我们使用了 oneAPI 的 SYCL 编程模型来实现并行计算。对于图片的相关输入，我们将输入图像数据和输出图像数据存储在 SYCL 缓冲区中。而后根据输入的图片数据，在并行 for 循环中，对每个像素进行能量计算、最小能量路径的计算和最小能量路径的移除操作。计算完成后，只需要将结果从设备上读取到输出图像数据中，并打印输出图像的宽度和高度即可。

三、 代码展示

以下给出了相应的代码实现：

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

using namespace sycl;

constexpr int ENERGY_THRESHOLD = 100;

// 像素能量计算函数
int calculateEnergy(int pixel, const std::vector<int>& neighbors)
{
  // 根据像素的数值和相邻像素的数值计算能量值
  // 省略具体的能量计算逻辑
  int energy = 0;
  for (int neighbor : neighbors) {
    energy += abs(pixel - neighbor);
  }
  return energy;
}

// 计算像素的能量值
void computeEnergyMap(const std::vector<int>& inputImage,
std::vector<int>& energyMap,
                      int width, int height) {
  // 对每个像素进行能量计算
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      // 获取相邻像素的数值
      std::vector<int> neighbors;
      if (x > 0) {
        neighbors.push_back(inputImage[y * width + (x - 1)]);
      }
      if (x < width - 1) {
```

```
        neighbors.push_back(inputImage[y * width + (x + 1)]);
      }
      if (y > 0) {
        neighbors.push_back(inputImage[(y - 1) * width + x]);
      }
      if (y < height - 1) {
        neighbors.push_back(inputImage[(y + 1) * width + x]);
      }

      // 计算能量值
      int energy = calculateEnergy(inputImage[y * width + x],
neighbors);
      energyMap[y * width + x] = energy;
    }
  }
}


// 移除能量最小路径的函数
void removeMinEnergyPath(std::vector<int>& image,
std::vector<int>& energyMap, int width, int height) {
  // 计算累计能量矩阵
  std::vector<int> cumulativeEnergyMap(width * height, 0);
  for (int x = 0; x < width; x++) {
    cumulativeEnergyMap[x] = energyMap[x];
  }
  for (int y = 1; y < height; y++) {
    for (int x = 0; x < width; x++) {
      int minEnergy = cumulativeEnergyMap[(y - 1) * width + x];
      if (x > 0) {
        minEnergy = std::min(minEnergy, cumulativeEnergyMap[(y -
1) * width + (x - 1)]);
      }
      if (x < width - 1) {
        minEnergy = std::min(minEnergy, cumulativeEnergyMap[(y -
1) * width + (x + 1)]);
      }
      cumulativeEnergyMap[y * width + x] = energyMap[y * width +
x] + minEnergy;
    }
  }

  // 寻找能量最小路径的起始位置
  int minEnergyPathStartIndex = 0;
  for (int x = 1; x < width; x++) {
```

```
    if (cumulativeEnergyMap[(height - 1) * width + x] <
cumulativeEnergyMap[(height - 1) * width +
minEnergyPathStartIndex]) {
      minEnergyPathStartIndex = x;
    }
  }

  // 移除能量最小路径
  for (int y = height - 1; y > 0; y--) {
    for (int x = minEnergyPathStartIndex; x < width - 1; x++) {
      image[y * width + x] = image[y * width + x + 1];
    }
    if (minEnergyPathStartIndex > 0) {
      image[y * width + minEnergyPathStartIndex - 1] = image[y *
width + minEnergyPathStartIndex];
    }
    if (minEnergyPathStartIndex < width - 1) {
      image[y * width + minEnergyPathStartIndex] = image[y * width
+ minEnergyPathStartIndex + 1];
    }
    minEnergyPathStartIndex += cumulativeEnergyMap[(y - 1) * width
+ minEnergyPathStartIndex] < cumulativeEnergyMap[(y - 1) * width +
minEnergyPathStartIndex + 1] ? 0 : 1;
  }

  // 移除最上方的像素
  image.erase(image.begin(), image.begin() + width);
}


int main() {
  // 创建队列和设备选择器
  default_selector selector;
  queue q(selector);

  // 输入图像的宽度和高度
  int width = 800;
  int height = 600;

  // 输入和输出图像的数据
  std::vector<int> inputImage(width * height);
  std::vector<int> outputImage((width / 2) * height);

  // 将输入图像数据拷贝到输入缓冲区
```

```
  buffer<int> inputImageBuf(inputImage.data(),
range<1>(inputImage.size()));
  buffer<int> outputImageBuf(outputImage.data(),
range<1>(outputImage.size()));

  // 使用队列执行并行计算
  q.submit([&](handler& h) {
    // 获取输入和输出缓冲区的访问器
    auto inputAcc =
inputImageBuf.get_access<access::mode::read>(h);
    auto outputAcc =
outputImageBuf.get_access<access::mode::write>(h);

    // 使用并行 for 循环进行 Seam Carving 操作
    h.parallel_for(range<1>(height), [=](id<1> idx) {
      // 计算每个像素的能量值
      std::vector<int> energyMap(width);
      computeEnergyMap(energyMap,width,height);

      // 移除最小能量路径
      for (int x = 0; x < width - 1; x++) {
        int sourceIndex = idx * width + x;
        int targetIndex = idx * (width/2) + x;
        outputAcc[targetIndex] = inputAcc[sourceIndex];
        if (energyMap[x] > ENERGY_THRESHOLD) {
          removeMinEnergyPath(outputImage,
energyMap,width,height);
        }
      }
    });
  });

  // 将结果从设备上读取到输出图像数据
  q.submit([&](handler& h) {
    auto outputAcc =
outputImageBuf.get_access<access::mode::read>(h);
    h.copy(outputAcc, outputImage.data());
  });

  // 打印输出图像的宽度和高度
  std::cout << "输出图像的宽度：" << width - 1 << std::endl;
  std::cout << "输出图像的高度：" << height << std::endl;

  return 0;
```

```
}
```