

课程作业

sycl 作业—矩阵乘法。

矩阵乘法是并行编程里面一个比较经典的问题，也是科学计算的基本构建块。进行矩阵乘法我们知道就是第*i*行的所有元素各自乘以第*j*列的所有元素相加后得到坐标为 (*i,j*) 的矩阵元素。如果将该代码并行化，最简单的思路那就是针对单个result(*i,j*)遍历第*i*行和第*j*列相加得到结果，然后使用GPU进行并行加速。

由于之前学过一部分cuda，所以是先用cuda编写的代码，然后使用visual studio 2019里面的oneapi扩展

- Intel® oneAPI DPC++/C++ Compiler
- Intel® oneAPI DPC++ Compatibility Tool component)

进行migrate得到的DPC++的代码。**推荐** 使用这种方法进行cuda代码迁移。之前整了半天oneapi专门的迁移工具，环境感觉很麻烦，迁移后的代码要经过一些检查和debug，不过大部分都是可以正确迁移的。下面是使用SYCL代码进行编写的矩阵乘法。

主要用到Intel的oneAPI工具集中的SYCL库来实现并行计算。SYCL是一种基于标准C++的编程模型，旨在提供对异构计算设备（如GPU、FPGA）的统一编程接口。代码中使用了SYCL的队列（`sycl::queue`）和并行计算（`sycl::parallel_for`）来将计算任务提交到设备上执行。为了在设备上分配内存，代码使用了SYCL的 `malloc_device` 函数，并在计算完成后使用 `free` 函数释放设备内存。

具体分析一下代码就是main函数中。

- 主要就是选择device，
- 然后在host端和设备端都为三个矩阵分配了内存。并且把host端的内容要复制到设备端。
- 然后调用DPC++上的parallel_for函数来进行并行计算。

其中 `dpct::get_in_order_queue()` 返回一个 SYCL 队列对象，可以用于并行计算的任务调度。

`parallel_for` 函数接受一个范围和一个 lambda 函数作为参数，用于指定并行计算的范围和计算操作。`sycl::nd_range<3>(numblock * threadperblock, threadperblock)`：这行代码定义了并行计算的范围，使用 `nd_range<3>` 表示一个三维的范围。`numblock * threadperblock` 表示每个维度的大小，这里是第一个维度和第二个维度的大小。`threadperblock` 表示第三个维度的大小，即工作组的大小。然后每个工作组都会执行 `mulKernel`，最后实现了并行化。

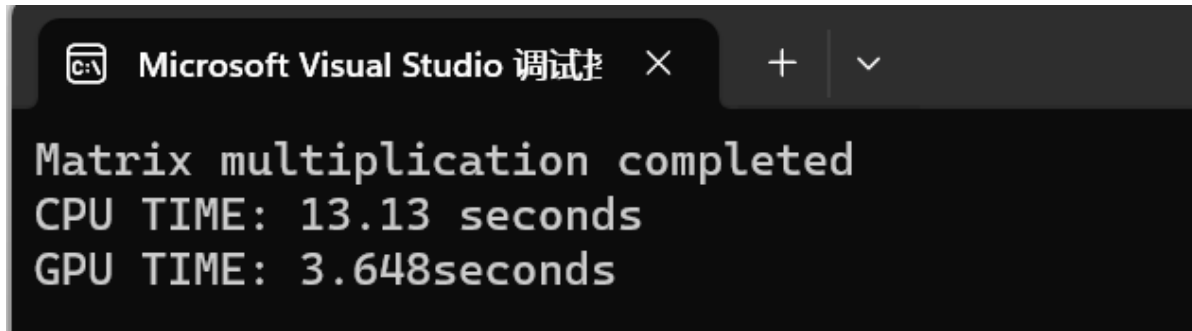
```
1 dpct::get_in_order_queue().parallel_for(  
2     sycl::nd_range<3>(numblock * threadperblock, threadperblock),  
3     [=](sycl::nd_item<3> item_ct1) {  
4         mulKernel(m1_gpu, m2_gpu, result_gpu, m, n, k, item_ct1);  
5     });
```

关于优化的话个人感觉主要有两个方面。

一个是减少访存。由于在这里的代码我是针对每个元素都去访问了m1和m2,可以考虑使用共享内存，这个版本的DPC++版本还没Debug完，所以就没贴出来。

二是可以考虑每一次计算时，计算多个结果，而不是只计算一个元素，这样可以让启动的线程变少，在线程不够时效果比较好。但是由于在我自己的电脑上跑出来的时间更长了所以没有往下研究。

最后对于两个1600*1600的矩阵相乘，在自己电脑上的运行得到结果：



```
Microsoft Visual Studio 调试 × + ∨  
Matrix multiplication completed  
CPU TIME: 13.13 seconds  
GPU TIME: 3.648seconds
```

```
1  #define blockw 4  
2  void mulkernel(int *m1, int *m2, int *result, int m, int n, int k,  
3      const sycl::nd_item<3> &item_ct1)  
4  {  
5      /* DPCT_ORIG      int col = blockDim.x * blockIdx.x + threadIdx.x;  
6      计算当前工作项的列索引  
7      */  
8      int col = item_ct1.get_local_range(2) * item_ct1.get_group(2) +  
9      item_ct1.get_local_id(2);  
10     /* DPCT_ORIG      int row = blockDim.y * blockIdx.y + threadIdx.y;  
11     计算当前工作项的行索引  
12     */  
13     int row = item_ct1.get_local_range(1) * item_ct1.get_group(1) +  
14     item_ct1.get_local_id(1);  
15     int re = 0;  
16     /*  
17     遍历m1的row行和m2的col列，对应相乘然后累加。存到结果矩阵中。  
18     */  
19     for (int i = 0; i < n; i++) {  
20         re += m1[row * n + i] * m2[i * k + col];  
21     }  
22     int index = row * k + col;  
23     result[index] = re;  
24  
25 }
```

```
1  int main()  
2  {  
3      int* m1; //m*n  
4      int* m2; //n*k  
5      int* result;  
6      // Add vectors in parallel.  
7      int m, n, k;  
8      m = 1600; n = 800; k = 1600;  
9      clock_t start, stop;  
10     m1 = sycl::malloc_host<int>(m * n, dpct::get_in_order_queue());  
11     m2 = sycl::malloc_host<int>(k * n, dpct::get_in_order_queue());  
12     result = sycl::malloc_host<int>(m * k, dpct::get_in_order_queue());  
13     for (int i = 0; i < m; i++) {  
14         for (int j = 0; j < n; j++) {  
15             m1[i * n + j] = i;  
16         }  
17     }  
18     for (int i = 0; i < n; i++) {  
19         for (int j = 0; j < k; j++) {
```

```

20         m2[i * k + j] = i;
21     }
22 }
23 int* m1_gpu, * m2_gpu, * result_gpu;
24
25 /*下面这段计算一下CPU进行矩阵乘法用时*/
26 clock_t start2 = clock();
27 for (int i = 0; i < m; i++) {
28     for (int j = 0; j < k; j++) {
29         result[i * k + j] = 0;
30         for (int x = 0; x < n; x++) {
31             result[i * k + j] += m1[i * n + x] * m2[x * k + j];
32         }
33     }
34 }
35
36 clock_t stop2 = clock();
37 double elapsedTime = static_cast<double>(stop2 - start2) /
CLOCKS_PER_SEC;
38
39 // Print the result and execution time
40 std::cout << "Matrix multiplication completed" << std::endl;
41 std::cout << "Execution time: " << elapsedTime << " seconds" <<
std::endl;
42
43 // Choose which GPU to run on, change this on a multi-GPU system.
44 dpct::select_device(0);
45 // Allocate GPU buffers for three vectors (two input, one output)
46 m1_gpu = sycl::malloc_device<int>(m * n, dpct::get_in_order_queue());
47 m2_gpu = sycl::malloc_device<int>(k * n, dpct::get_in_order_queue());
48 result_gpu = sycl::malloc_device<int>(m * k,
dpct::get_in_order_queue());
49
50 // Copy input vectors from host memory to GPU buffers.
51 /* DPCT_ORIG      cudaMemcpy(m1_gpu, m1, m * n * sizeof(int),
52 * cudaMemcpyHostToDevice);*/
53 dpct::get_in_order_queue().memcpy(m1_gpu, m1, m * n *
sizeof(int)).wait();
54 /* DPCT_ORIG      cudaMemcpy(m2_gpu, m2, n * k * sizeof(int),
55 * cudaMemcpyHostToDevice);*/
56 dpct::get_in_order_queue().memcpy(m2_gpu, m2, n * k *
sizeof(int)).wait();
57 // Launch a kernel on the GPU with one thread for each element.
58
59 sycl::range<3> threadperblock(1, blockw, blockw);
60 sycl::range<3> numblock(1, (m + blockw - 1) / blockw,
(k + blockw - 1) / blockw);
61
62 start = clock();
63 dpct::get_in_order_queue().parallel_for(
64     sycl::nd_range<3>(numblock * threadperblock, threadperblock),
65     [=](sycl::nd_item<3> item_ct1) {
66         mulkernel(m1_gpu, m2_gpu, result_gpu, m, n, k, item_ct1);
67     });
68 dpct::get_current_device().queues_wait_and_throw();
69
70 stop = clock();

```

```

71     std::cout << "Block size: " << blockw << "x" << blockh << ", GPU TIME:
    " << (double)(stop - start) / CLOCKS_PER_SEC * 1000000.0 << " microseconds"
    << std::endl;
72     // Copy output vector from GPU buffer to host memory.
73     /* DPCT_ORIG      cudaMemcpy(result, result_gpu, m * k * sizeof(int),
74      * cudaMemcpyDeviceToHost);*/
75     dpct::get_in_order_queue()
76         .memcpy(result, result_gpu, m * k * sizeof(int))
77         .wait();
78
79     sycl::free(m1_gpu, dpct::get_in_order_queue());
80     sycl::free(m2_gpu, dpct::get_in_order_queue());
81     sycl::free(result_gpu, dpct::get_in_order_queue());
82
83     std::ofstream file("result.txt");
84     if (file) {
85         for (int i = 0; i < m*k; ++i) {
86             file << result[i] << " ";
87         }
88         file.close();
89     }
90     else {
91         std::cerr << "Failed to open file for writing." << std::endl;
92     }
93     sycl::free(m1, dpct::get_in_order_queue());
94     sycl::free(m2, dpct::get_in_order_queue());
95     sycl::free(result, dpct::get_in_order_queue());
96     return 0;
97 }
98 int sdiv(int x, int y) {
99     if (y == 0) {
100         return 0;
101     }
102
103     int result = (x + y - 1) / y;
104     return result;
105 }

```

SYCL 作业二归并排序

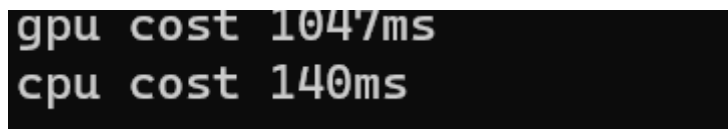
实现归并排序的GPU加速，我们知道归并排序主要的思想就是**分治**，也就是把一个大问题分解成多个小问题，小问题的求解之间是没有依赖关系的，因此可以用来并行化。并行化的主要思路就是将数组分成多段，然后把相邻两段合并，因为是多段，所以这个阶段可以并行处理。合并之后会继续迭代地合并，直到我们整个数组都排好序，也就是说仅剩一段数组。然后代码如下。

首先是Merge函数，主要在并行环境中进行归并排序的合并操作。首先根据当前的 SYCL 执行项计算出唯一的 `id`，用于确定当前执行项在整个并行计算中的唯一位置。然后，根据 `id` 计算出合并操作的起始索引和结束索引，以及存储结果的目标索引。其中，`index1` 和 `endIndex1` 表示第一个子数组的起始和结束索引，`index2` 和 `endIndex2` 表示第二个子数组的起始和结束索引，`targetIndex` 表示存储结果的目标索引。最后使用一个循环将两个子数组合并到临时数组中。

接下来是mergesort函数，函数中获取当前的设备并创建一个 SYCL 队列 `q_ct1`，以便在设备上执行并行计算。然后，使用 `sycl::malloc_device` 分配设备上的内存来存储排序算法所需的数组 `dev_a` 和 `dev_temp`。在循环中，迭代地调用 `q_ct1.parallel_for` 来进行归并排序的每一轮合并操作。在并行执行的每个工作组内，调用 `merge` 函数来执行合并操作。合并操作的参数包括设备上的数组 `dev_a` 和 `dev_temp`、已排序的子数组大小 `sortedsize` 和待排序数组的总大小 `N`。

最后得到结果对102400长的数组进行归并排序，与调用sort库函数得到的结果是一样的，保证了正确性。

与串行版本cpu的归并排序结果对比如下： 可以看到结果和预期不符，推断是机器问题，在gpu上内存传输可能耗费了大部分时间。



```
gpu cost 1047ms
cpu cost 140ms
```

```
1  #include <sycl/sycl.hpp>
2  #include <dpct/dpct.hpp>
3  using namespace std;
4  #include <cstdlib>
5  #include <stdio.h>
6  #include <iostream>
7  #include <windows.h>
8  #include <fstream>
9  #include <ctime>
10 #include <vector>
11 #include <algorithm>
12 void merge(int* a, int* temp, int sortedsize, int N,
13           const sycl::nd_item<3> &item_ct1)
14 {
15     // int id = blockIdx.x * blockDim.x + threadIdx.x;
16
17     int blockid = item_ct1.get_group(0) * item_ct1.get_group_range(2) *
18                 item_ct1.get_group_range(1) +
19                 item_ct1.get_group(1) * item_ct1.get_group_range(2) +
20                 item_ct1.get_group(2);
21     int id = blockid * item_ct1.get_local_range(2) +
22             item_ct1.get_local_id(2);
23
24     unsigned long index1, index2, endIndex1, endIndex2, targetIndex;
25     index1 = id * 2 * sortedsize;
26     endIndex1 = index1 + sortedsize;
27     index2 = endIndex1;
28     endIndex2 = index2 + sortedsize;
29     targetIndex = id * 2 * sortedsize;
30
31     if (index1 >= N) return;
32
33     if (endIndex1 > N)
34     {
35         endIndex1 = N;
36         index2 = endIndex2 = N;
37     }
38     if (index2 > N)
39     {
40         index2 = endIndex2 = N;
41     }
42     if (endIndex2 > N)
43     {
44         endIndex2 = N;
45     }
46     int done = 0;
47     while (!done)
48     {
```

```

46         if ((index1 == endIndex1) && (index2 < endIndex2))
47             temp[targetIndex++] = a[index2++];
48         else if ((index2 == endIndex2) && (index1 < endIndex1))
49             temp[targetIndex++] = a[index1++];
50         else if (a[index1] < a[index2])
51             temp[targetIndex++] = a[index1++];
52         else
53             temp[targetIndex++] = a[index2++];
54
55         if ((index1 == endIndex1) && (index2 == endIndex2))
56             done = 1;
57     }
58 }
59
60 int mergesort(int* data, int N, float& cost_time)
61 {
62     dpct::device_ext &dev_ct1 = dpct::get_current_device();
63     sycl::queue &q_ct1 = dev_ct1.in_order_queue();
64     int* dev_a, * dev_temp;
65
66     dev_a = sycl::malloc_device<int>(N, q_ct1);
67     dev_temp = sycl::malloc_device<int>(N, q_ct1);
68     q_ct1.memcpy(dev_a, data, sizeof(int) * N).wait();
69     int blocks = 512;
70     sycl::range<3> grids(1, 1, 128);
71     float t0 = GetTickCount();
72     int sortedsize = 1;
73     while (sortedsize < N)
74     {
75         /*
76         DPCT1049:0: The work-group size passed to the SYCL kernel may
77         exceed the
78         limit. To get the device limit, query
79         info::device::max_work_group_size.
80         Adjust the work-group size if needed.
81         */
82         q_ct1.parallel_for(
83             sycl::nd_range<3>(grids * sycl::range<3>(1, 1, blocks),
84                             sycl::range<3>(1, 1, blocks)),
85             [=](sycl::nd_item<3> item_ct1) {
86                 merge(dev_a, dev_temp, sortedsize, N, item_ct1);
87             });
88         q_ct1.memcpy(dev_a, dev_temp, N * sizeof(int));
89         sortedsize *= 2;
90     }
91
92     q_ct1.memcpy(data, dev_a, N * sizeof(int)).wait();
93     cost_time = GetTickCount() - t0;
94     sycl::free(dev_a, q_ct1);
95     sycl::free(dev_temp, q_ct1);
96
97     dev_ct1.queue_wait_and_throw();
98
99     return 0;
100 }
101
102 int main(int argc, char* argv[])
103 {

```

```

102     int N = 160000;
103     int* data = new int[N];
104     std::vector<int> data_vec;
105     for (int k = 0; k < N; k++)
106     {
107         data[k] = rand() % 4096;
108         data_vec.push_back(data[k]);
109         //std::cout << data[k] << ", ";
110     }
111     std::cout << std::endl;
112
113     //float t0 = GetTickCount();
114     float cost_gpu;
115     mergesort(data, N, cost_gpu);
116     //float t1 = GetTickCount();
117
118     float tt0 = GetTickCount();
119     mergeSort(data_vec, 0, data_vec.size() - 1);
120     float tt1 = GetTickCount();
121
122     int flag = 0;
123     for (int k = 0; k < N; k++)
124     {
125         if (data[k] == data_vec[k])
126         {
127             flag++;
128         }
129     }
130     std::cout << std::endl;
131     std::cout << "check result (" << flag << ", " << N << ") = " << (flag ==
132     N) << std::endl;
133
134     std::cout << "gpu cost " << cost_gpu << "ms" << std::endl;
135     std::cout << "cpu cost " << tt1 - tt0 << "ms" << std::endl;
136     return 0;
137 }

```

作业三：图像卷积并行加速

图像卷积是一种常见的图像处理操作，用于应用各种滤波器和特征检测器。其原理可以简单地描述为在图像的每个像素上应用一个小的矩阵（通常称为卷积核或滤波器），并将卷积核中的元素与图像中对应位置的像素值相乘，然后将所有乘积的和作为结果。 `convolution` 函数接受输入图像、卷积核以及相关的尺寸参数，并在设备上执行矩阵卷积运算。最后，计算结果被复制回主机内存并存储在 `output` 向量中。

下文也是用了Intel的oneAPI工具集中的SYCL库来实现并行计算。首先是定义了一个卷积函数。每个工作项负责计算输出图像的一个卷积后的像素。然后在`convolution`函数中多次调用卷积函数。

- 使用SYCL的 `parallel_for` 函数来启动并行计算。指定工作项的范围和工作组大小。
- 在内核函数中，可以使用 `nd_item` 对象获取工作项的索引和工作组信息，以便确定要计算的像素位置。

最后一部分就是使用 `memcpy` 函数将计算得到的输出结果从设备内存复制回主机内存中存储的输出向量中并释放相关内存。

最后得到的结果经过验证也是正确的。

```

1  #include <sycl/sycl.hpp>
2  #include <dpct/dpct.hpp>
3  #include <iostream>
4  #include <vector>
5
6  void convolutionKernel(const int* input, const int* kernel, int* output,
7                        int inputwidth, int inputHeight, int kernelwidth, int kernelHeight, int
8                        outputwidth, int outputHeight,
9                        const sycl::nd_item<3> &item_ct1)
10 {
11     int x = item_ct1.get_group(2) * item_ct1.get_local_range(2) +
12             item_ct1.get_local_id(2);
13     int y = item_ct1.get_group(1) * item_ct1.get_local_range(1) +
14             item_ct1.get_local_id(1);
15
16     if (x < outputwidth && y < outputHeight)
17     {
18         int sum = 0;
19         for (int ky = 0; ky < kernelHeight; ky++)
20         {
21             for (int kx = 0; kx < kernelwidth; kx++)
22             {
23                 int inputX = x + kx;
24                 int inputY = y + ky;
25                 sum += input[inputY * inputwidth + inputX] * kernel[ky *
26                             kernelwidth + kx];
27             }
28         }
29         output[y * outputwidth + x] = sum;
30     }
31 }
32
33 // 定义卷积函数
34 void convolution(const std::vector<int>& input, const std::vector<int>&
35                 kernel, std::vector<int>& output, int inputwidth, int inputHeight, int
36                 kernelwidth, int kernelHeight)
37 {
38     sycl::device dev_ct1;
39     sycl::queue q_ct1(dev_ct1,
40
41
42     sycl::property_list{sycl::property::queue::in_order()});
43     int outputwidth = inputwidth - kernelwidth + 1;
44     int outputHeight = inputHeight - kernelHeight + 1;
45     int inputSize = inputwidth * inputHeight * sizeof(int);
46     int kernelSize = kernelwidth * kernelHeight * sizeof(int);
47     int outputSize = outputwidth * outputHeight * sizeof(int);
48
49     // 在设备上分配内存
50     int* d_input;
51     int* d_kernel;
52     int* d_output;
53     d_input = (int *)sycl::malloc_device(inputSize, q_ct1);
54     d_kernel = (int *)sycl::malloc_device(kernelSize, q_ct1);
55     d_output = (int *)sycl::malloc_device(outputSize, q_ct1);
56
57     // 将输入数据复制到设备内存
58     q_ct1.memcpy(d_input, input.data(), inputSize).wait();
59     q_ct1.memcpy(d_kernel, kernel.data(), kernelSize).wait();

```



```

53
54     sycl::range<3> blockSize(1, 16, 16);
55     sycl::range<3> gridSize(1, (outputHeight + blockSize[1] - 1) /
blockSize[1],
56                                     (outputWidth + blockSize[2] - 1) /
blockSize[2]);
57
58     /*
59     DPCT1049:0: The work-group size passed to the SYCL kernel may exceed
the
60     limit. To get the device limit, query
info::device::max_work_group_size.
61     Adjust the work-group size if needed.
62     */
63     q_ct1.parallel_for(sycl::nd_range<3>(gridSize * blockSize, blockSize),
64                         [=](sycl::nd_item<3> item_ct1) {
65         convolutionKernel(
66             d_input, d_kernel, d_output, inputWidth,
67             inputHeight, kernelWidth, kernelHeight,
68             outputWidth, outputHeight, item_ct1);
69     });
70
71     // 将计算结果复制回主机内存
72     q_ct1.memcpy(output.data(), d_output, outputsize).wait();
73
74     // 释放设备内存
75     sycl::free(d_input, q_ct1);
76     sycl::free(d_kernel, q_ct1);
77     sycl::free(d_output, q_ct1);
78 }
79
80 void test()
81 {
82     // 定义输入图像和卷积核
83     std::vector<int> input = { 1, 2, 3, 4, 5,
84                               6, 7, 8, 9, 10,
85                               11, 12, 13, 14, 15,
86                               16, 17, 18, 19, 20,
87                               21, 22, 23, 24, 25 };
88     std::vector<int> kernel = { 1, 5, -1,
89                                2, 0, -2,
90                                1, 0, -1 };
91
92     int inputWidth = 5;
93     int inputHeight = 5;
94     int kernelWidth = 3;
95     int kernelHeight = 3;
96     int outputWidth = inputWidth - kernelWidth + 1;
97     int outputHeight = inputHeight - kernelHeight + 1;
98
99     // 创建输出图像
100     std::vector<int> output(outputWidth * outputHeight);
101
102     // 进行卷积运算
103     convolution(input, kernel, output, inputWidth, inputHeight,
kernelWidth, kernelHeight);
104
105     // 打印输出图像

```

```
106     for (int y = 0; y < outputHeight; y++)
107     {
108         for (int x = 0; x < outputWidth; x++)
109         {
110             std::cout << output[y * outputWidth + x] << " ";
111         }
112         std::cout << std::endl;
113     }
114 }
115
116 int main()
117 {
118     test();
119     return 0;
120 }
```