天津大学

《异构计算》

实验 3：基于 OpenCL 实现矩阵的幂

学　院　智能与计算学部

专　业　计算机科学与技术

年　级　2019

姓　名　张明君（留学生）

学　号　6319000359

2021 年 11 月 3 日

一， 实验内容

本课程实验目的为提升 我们对异构计算的理解认识，培养编写 GPU 与 CPU 异构程序的能力，加深对 OpenCL 异构并行编程的理解认识。

二， 实验原理

本次实验要求使用 OpenCL，分别使用暴力法和结合律的方法，以及多核 CPU 与 GPU 分别作为 device，计算矩阵的幂

3. 基于 OpenCL 实现矩阵的幂

用 OpenCL 编程模型实现矩阵A的n次幂。

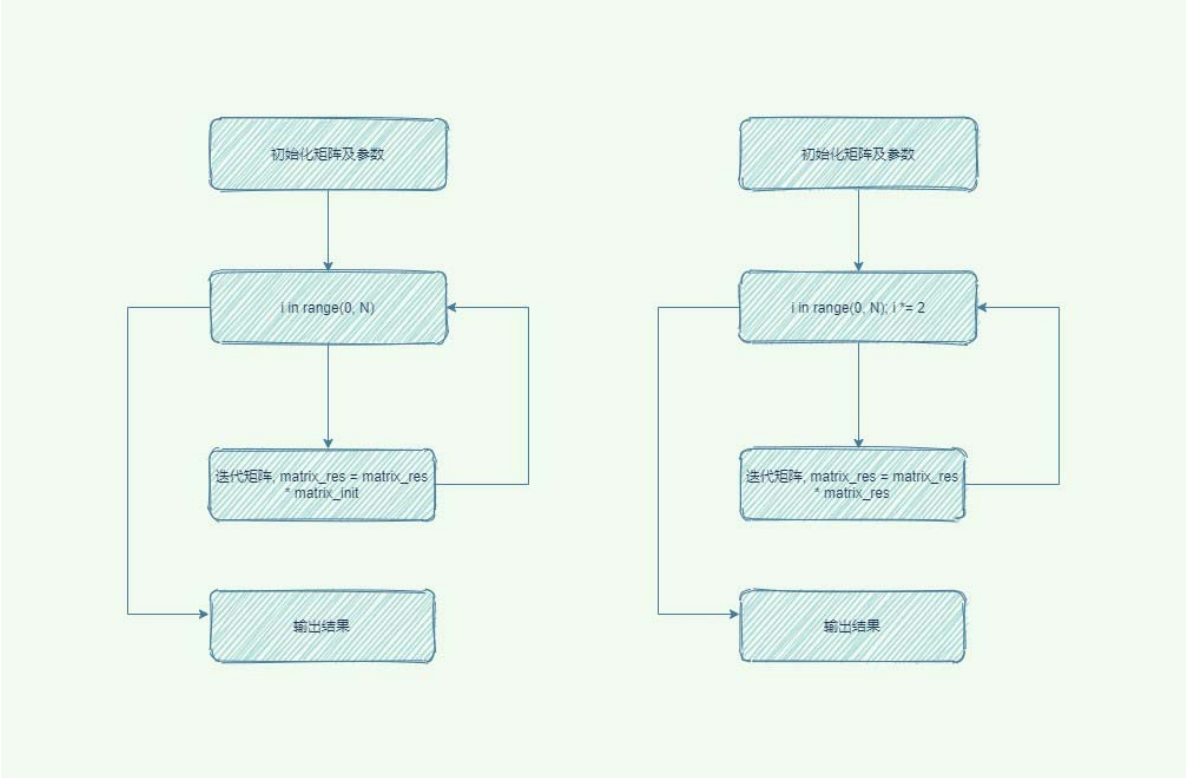
要求实现暴力算法和高效算法，同时对比分析一下相同 OpenCL 程序分别运行在纯多核 CPU 环境下以及异构 GPU 环境下的性能。

三， 程序流程图

OpenCL程序设计流程

1 查询平台	clGetPlatformIDs()
2 查询设备	clGetDeviceIDs()
3 创建上下文：将平台设备与上下文关联起来	clCreateContext() 或 clCreateContextFromType()
4 创建命令队列	clCreateCommandQueue()
5 读取、编译内核	clCreateProgramWithSource()或 clCreateProgramWithBinary() clBuildProgram()
6 打包生成内核	clCreateKernel()
7 创建缓存对象或图像对象，为内核参数分配内存	clCreateBuffer()、clCreateSubBuffer() clCreateImage2D()、clCreateImage3D()
8 设置内核参数，将上面分配的内存发送到设备上	clSetKernelArg()
9 执行内核	clEnqueueTask()或 clEnqueueNDRangeKernel()
10 读取设备上的处理结果	clEnqueueReadBuffer()
11 释放创建的资源：创建的内存、命令队列、内核、打包的程序、上下文	clReleaseMemObject()、 clReleaseCommandQueue()、 clReleaseKernel()、 clReleaseProgram()、 clReleaseContext()

数学计算模型



- 多核 CPU 设计代码

```
#include<iostream>
#include<CL/cl.h>
#include<ctime>
#include<cstdio>
#include<cstdlib>
#include<fstream>
#include<cstring>
#include<sstream>

using namespace std;

const int M = 64;
const int N = 2048;

#define CPU_BRUTE 0;
#define CPU_EFFICIENT 1;

string kernels[2] = { "KERNEL_CPUS_BRUTE", "KERNEL_CPUS_EFFICIENT" };
void OCLMatrixPower(FILE*, int);

int main(int argc, char** argv){
    srand(time(NULL));
  FILE* fp;
  fp = fopen("./data", "a+");
    for (int i = 0; i < 5; i++) {
        OCLMatrixPower(fp, 0);
        OCLMatrixPower(fp, 1);
    }
  fprintf(fp, "\n");
  fclose(fp);
  return 0;
}
```

```
cl_command_queue CreateCommandQueue(int type, cl_device_id *device, cl_uint *numDevices, cl_context *context){
    cl_int errNum;
    cl_uint numPlatforms;
    cl_platform_id platformId[3];
    cl_command_queue commandQueue;

    errNum = clGetPlatformIDs(3, platformId, &numPlatforms);
    if(errNum != CL_SUCCESS || numPlatforms <= 0){
        cerr << "Error getting platform IDs.\n";
        return NULL;
    }

    char param_value[512];
    clGetDeviceIDs(platformId[2], CL_DEVICE_TYPE_CPU, 1, device, numDevices);

    *context = clCreateContext(NULL, 1, device, NULL, NULL, &errNum);
    if(errNum != CL_SUCCESS){
        cerr << "Error creating context.\n";
        return NULL;
    }

    commandQueue = clCreateCommandQueue(*context, *device, 0, NULL);
    if(commandQueue == NULL){
        cerr << "Error creating command queue.\n";
        return NULL;
    }

    return commandQueue;
}
```

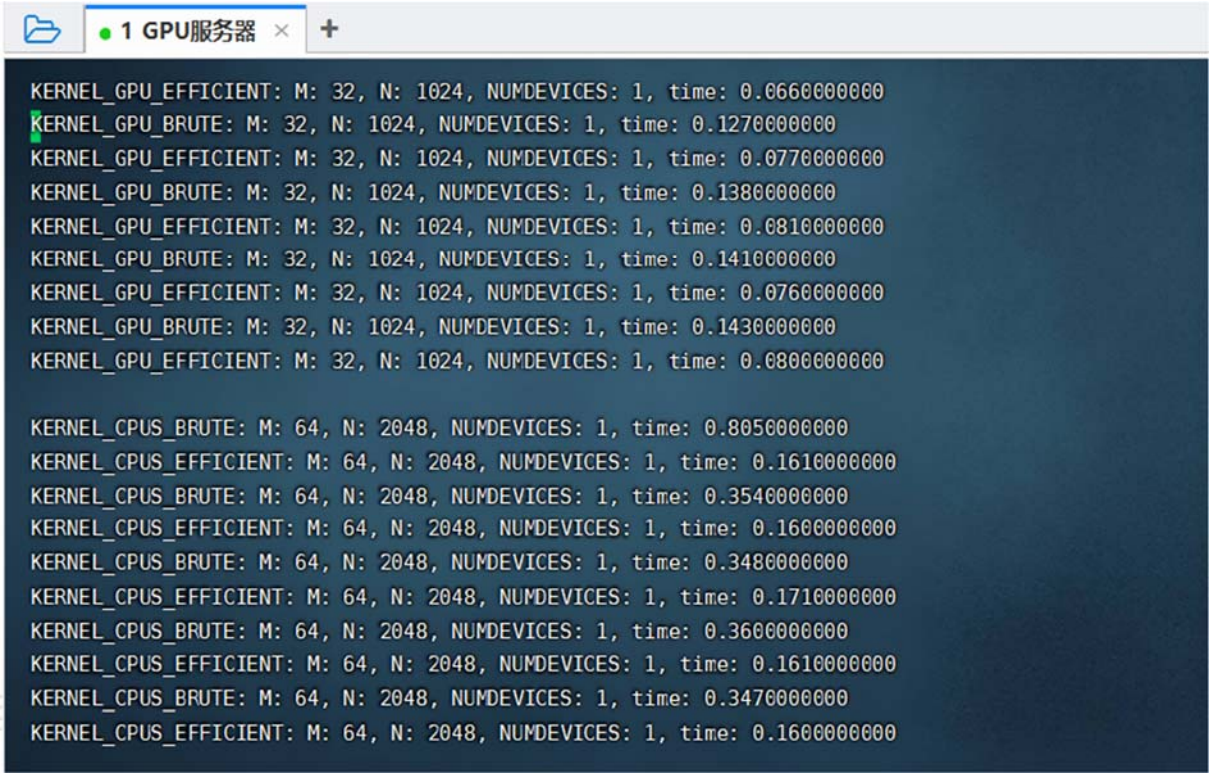
```
cl_program CreateProgram(cl_context *context, cl_device_id *device, const char *fileName){
    cl_int errNum = CL_SUCCESS;
    cl_program program;

    ifstream kernelFile(fileName, ios::in);
    ostringstream oss;
    oss << kernelFile.rdbuf();

    string srcStdStr = oss.str();
    const char * srcStr = srcStdStr.c_str();

    program = clCreateProgramWithSource(*context, 1, (const char **)&srcStr, NULL, NULL);
    if(program == NULL){
        cerr << "Error creating program.\n";
        return NULL;
    }

    errNum = clBuildProgram(program, 1, device, NULL, NULL, NULL);
    if(errNum != CL_SUCCESS){
        cerr << "Error building program. errNum: " << errNum << endl;
        char param_value[512];
        clGetProgramBuildInfo(program, *device, CL_PROGRAM_BUILD_LOG, 512, param_value, NULL);
        printf("%s\n", param_value);
        return NULL;
    }

    return program;
}
```

因为代码有点多所以我只放了一些代码在这里，源代码我也跟报告一起交的。

- GPU 设计代码

```
#include<iostream>
#include<CL/cl.h>
#include<ctime>
#include<cstdio>
#include<cstdlib>
#include<fstream>
#include<cstring>
#include<sstream>

using namespace std;

const int M = 32;
const int N = 1024;

#define GPU_BRUTE 0;
#define GPU_EFFICIENT 1;

string kernels[2] = { "KERNEL_GPU_BRUTE", "KERNEL_GPU_EFFICIENT" };
void OCLMatrixPower(FILE*, int);

int main(int argc, char** argv){
    srand(time(NULL));
    FILE* fp;
  fp = fopen("./data", "a+");
    for(int i = 0; i < 5; i++){
        OCLMatrixPower(fp, 0);
        OCLMatrixPower(fp, 1);
    }
  fprintf(fp, "\n");
  fclose(fp);
  return 0;
}
cl_command_queue CreateCommandQueue(int type, cl_device_id *device, cl_uint *numDevices, cl_context *context){
    cl_int errNum;
    cl_uint numPlatforms;
    cl_platform_id platformId;
    cl_command_queue commandQueue;

    errNum = clGetPlatformIDs(1, &platformId, &numPlatforms);
    if(errNum != CL_SUCCESS || numPlatforms <= 0){
        cerr << "Error getting platform IDs.\n";
        return NULL;
    }

    clGetDeviceIDs(platformId, CL_DEVICE_TYPE_GPU, 1, device, NULL);

    *context = clCreateContext(NULL, 1, device, NULL, NULL, &errNum);
    if(errNum != CL_SUCCESS){
        cerr << "Error creating context.\n";
        return NULL;
    }

    commandQueue = clCreateCommandQueue(*context, *device, 0, NULL);
    if(commandQueue == NULL){
        cerr << "Error creating command queue.\n";
        return NULL;
    }

    return commandQueue;
}
```

四， 实验结果及分析



现在我们来统计一下数据如下表所示：

暴力算法

	多核 CPU	GPU
平均时间(s)	0.35	0.138
M, N	64, 2048	64, 2048

高效算法

	多核 CPU	GPU
平均时间(s)	0.16	0.075
M, N	64, 2048	64, 2048

加速比

	暴力法	结合律法
加速比	2.546	2.133

实验结果分析：

1. 首先是方法上的比较，根据多次测试所得平均数据的结果来看，暴力法相对于结合律法非常低效， 在 OpenCL + 多核 CPU 时间大致是结合律两倍+；在 OpenCL + GPU 上，差别稍微小 一些，但也 是将近两倍的时间，可见设计一个良好的算法对于程序的改进是很大的。

2. 然后是多核 CPU 和 GPU 的比较，无论是暴力法还是结合律法，使用 GPU 的 OpenCL 程序 都减少 了大量时间，效率相对于多核 CPU 有很大提高，尤其是在暴力法上，加速比达到了 2.5+， 可见在 处理矩阵方面，GPU 相对于 CPU 或多核 CPU，都有着天然的优势。

五， 实验总结

本次实验中我了解到了如何使用 OpenCL 来编写通用性的并行程序，同时也从头配置了 OpenCL，并自己写 OpenCL 程序并编译运行，在书写的时候也由于对于 OpenCL 提供的 API 不够了解而导致出错， 经过查阅资料与尝试后成功解决了。