OpenCL&Cuda

姓名：张煜玮　　班级：计科1905　　学号:19291255

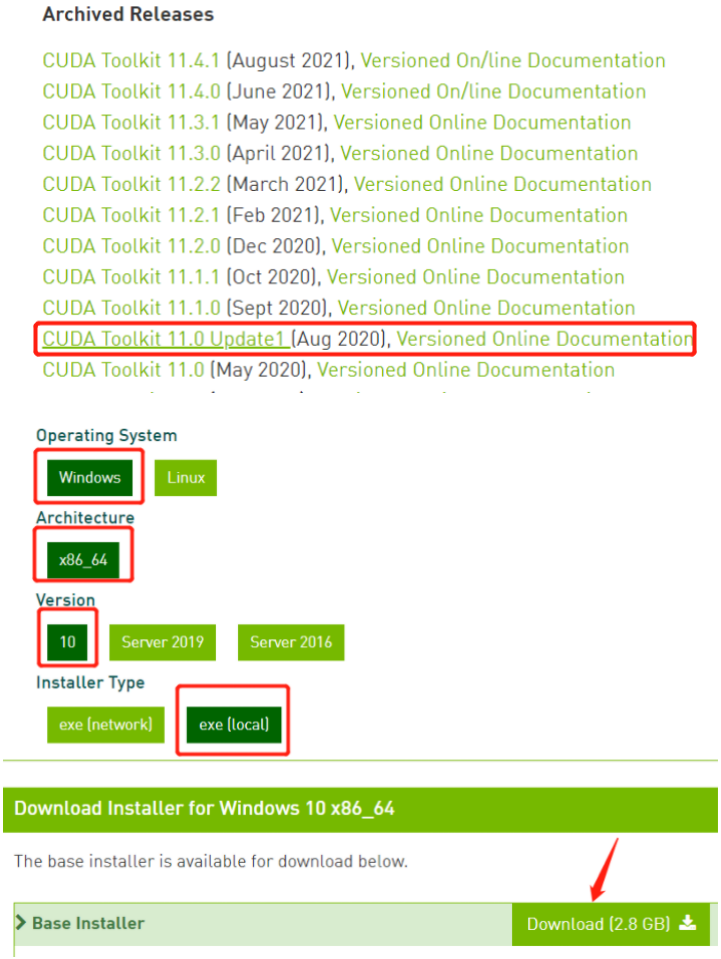
OpenCL&Cuda

1. 环境配置

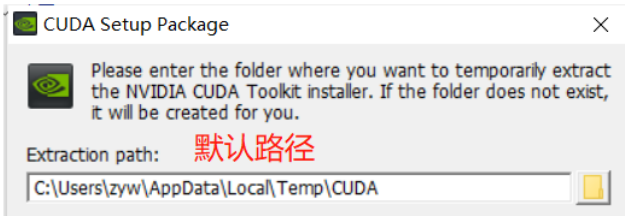
1.1 cuda的安装和配置

1.1.1 安装CUDA

1. 从CUDA Toolkit 官网下载地址上选择合适的版本进行下载

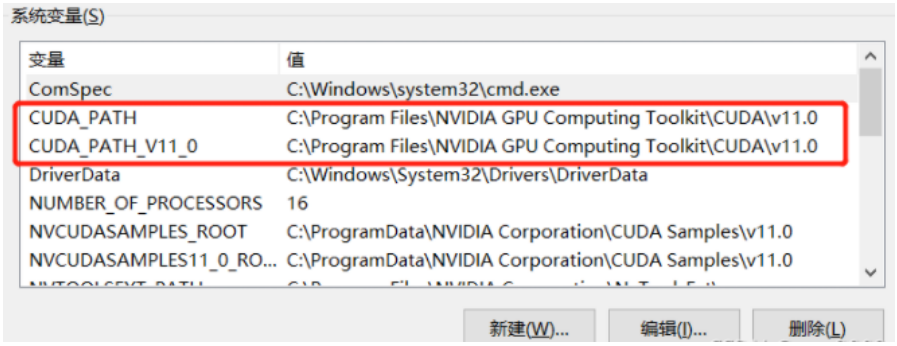


2. 选择默认的路径进行安装



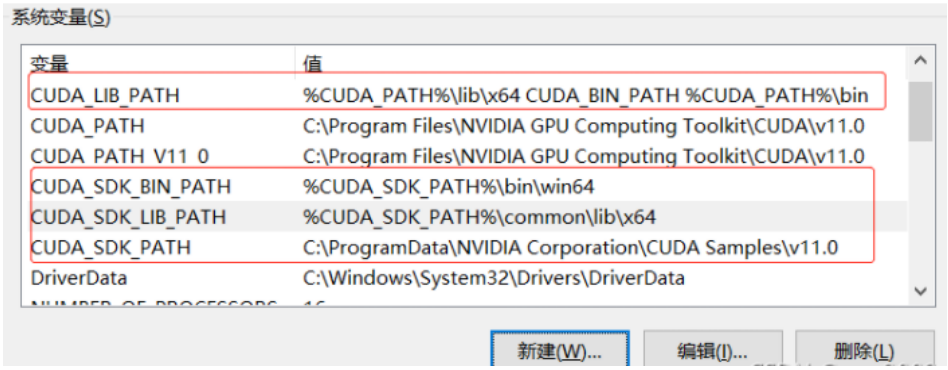
1.1.2 配置环境变量

1. 检查是否安装成功
   安装成功后在系统的环境变量中会自动添加两个变量，如图：“CUDA_PATH", 和”CUDA_PATH_V11_0"



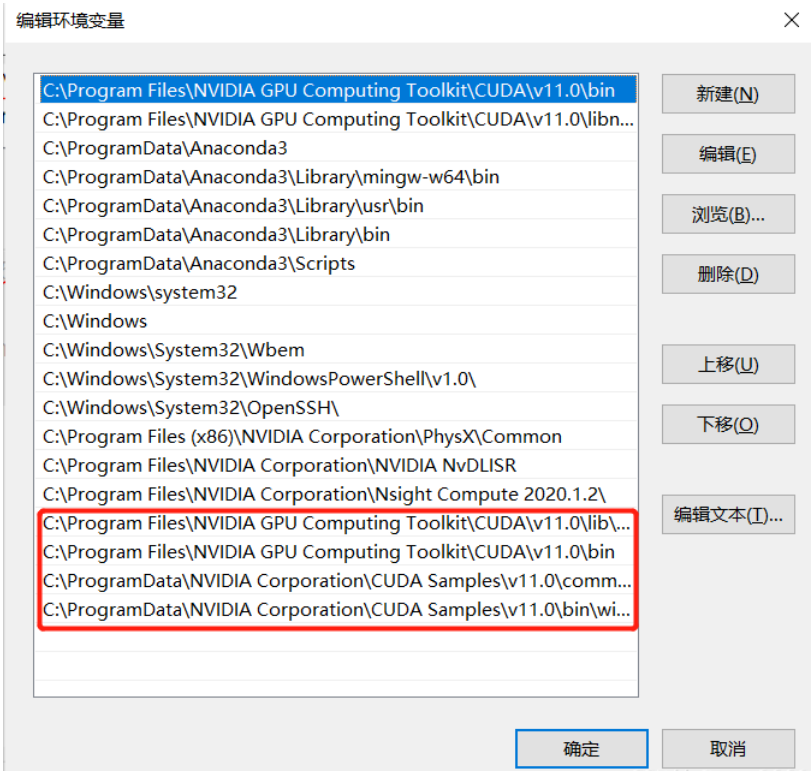
2. 添加系统变量
   点击“新建”， 然后依次添加下述5个系统变量，然后点击“确认”

```
变量                    值
CUDA_SDK_PATH           C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.0
CUDA_LIB_PATH           %CUDA_PATH%\lib\x64 CUDA_BIN_PATH %CUDA_PATH%\bin
CUDA_SDK_BIN_PATH       %CUDA_SDK_PATH%\bin\win64
CUDA_SDK_LIB_PATH       %CUDA_SDK_PATH%\common\lib\x64
```



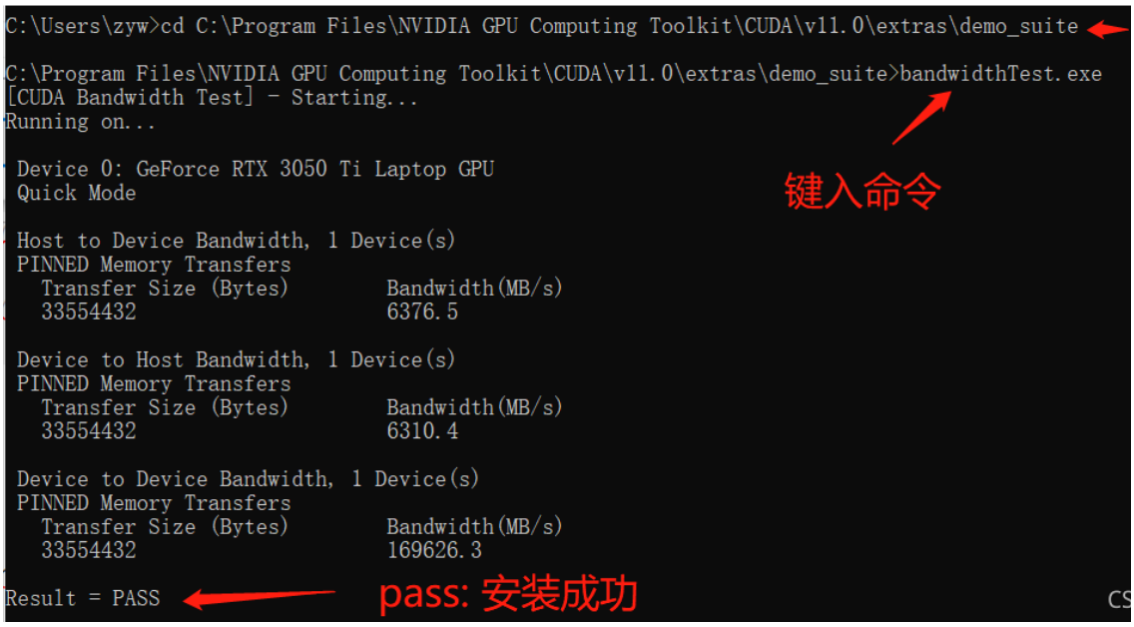
然后在系统变量Path里添加变量，双击Path添加如下变量

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.0\lib\x64
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.0\bin
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.0\common\lib\x64
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.0\bin\win64
```

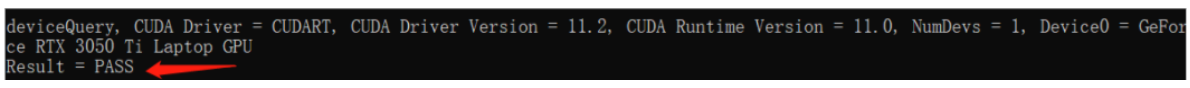


1.1.3 检查cuda是否安装成功

1. 打开cmd命令窗口，切换路径
   cd C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.0\extras\demo_suite
2. 分别运行如下命令:
   `bandwidthTest.exe`
   如下图，Result=Pass

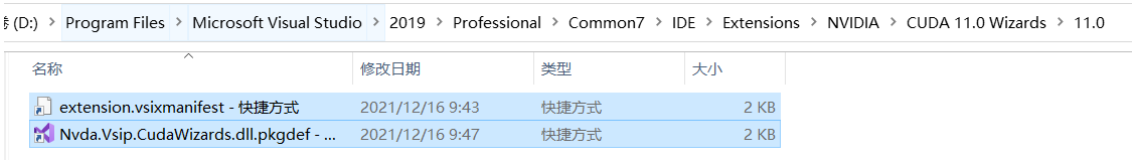


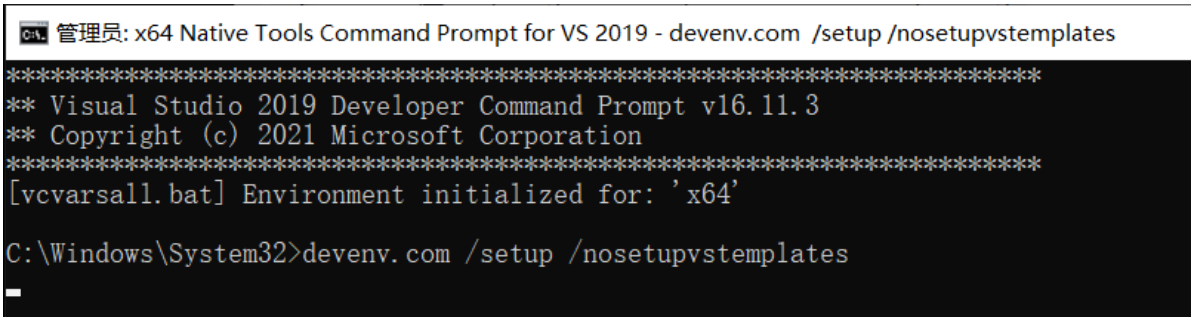
   `deviceQuery.exe`
   如下图，Result=Pass , 安装成功。



1.2 VS中配置cuda

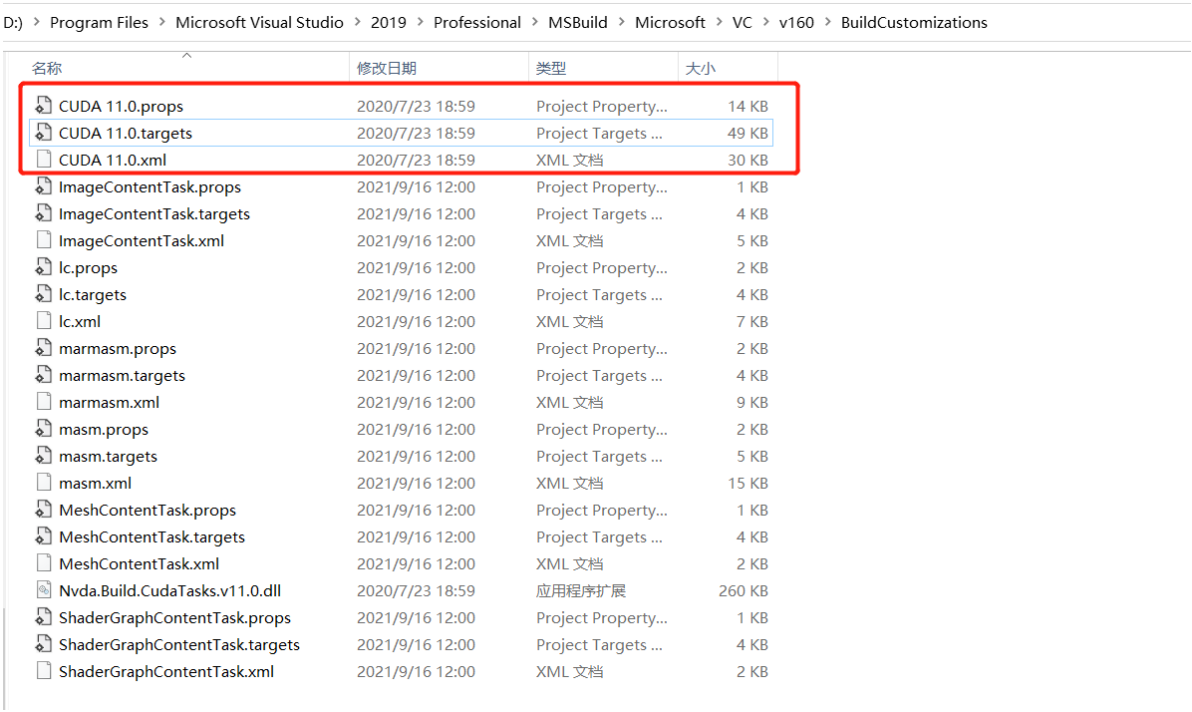
1. 在`C:\ProgramData\NVIDIA GPU Computing Toolkit\v10.1\extras\visual_studio_integration\CudaProjectVsWizards`中找到`Nvda.Vsip.CudaWizards.dll.pkgdef`并创建快捷方式
2. 在`C:\ProgramData\NVIDIA GPU Computing Toolkit\v11.0\extras\visual_studio_integration\CudaProjectVsWizards\2019`中找到`extension.vsixmanifest`并创建快捷方式
3. 在`D:\Program Files\Microsoft Visual Studio\2019\Professional\Common7\IDE\Extensions`下创建新的文件夹`NVIDIA`->`CUDA 11.0 Wizards`->11.0。





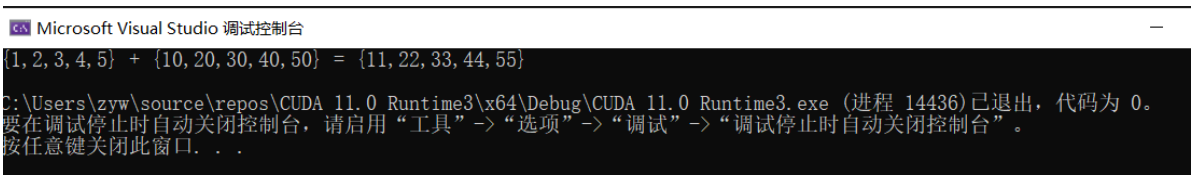
4. 将`C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.0\extras\visual_studio_integration\MSBuildExtensions`下的文件复制，并粘贴到

   `D:\Program Files\Microsoft Visual Studio\2019\Professional\MSBuild\Microsoft\VC\v160\BuildCustomizations`文件下。



5. 新建立一个cuda runtime 的文件

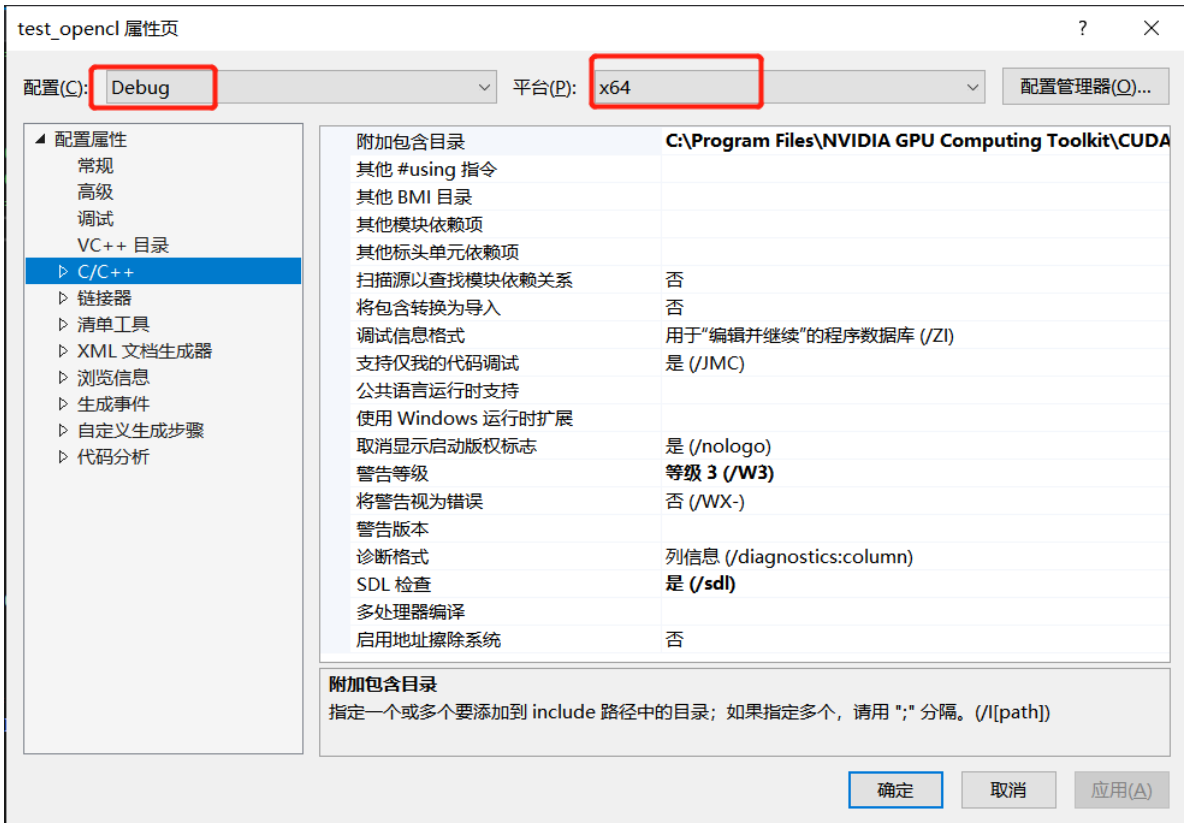
6. 测试运行



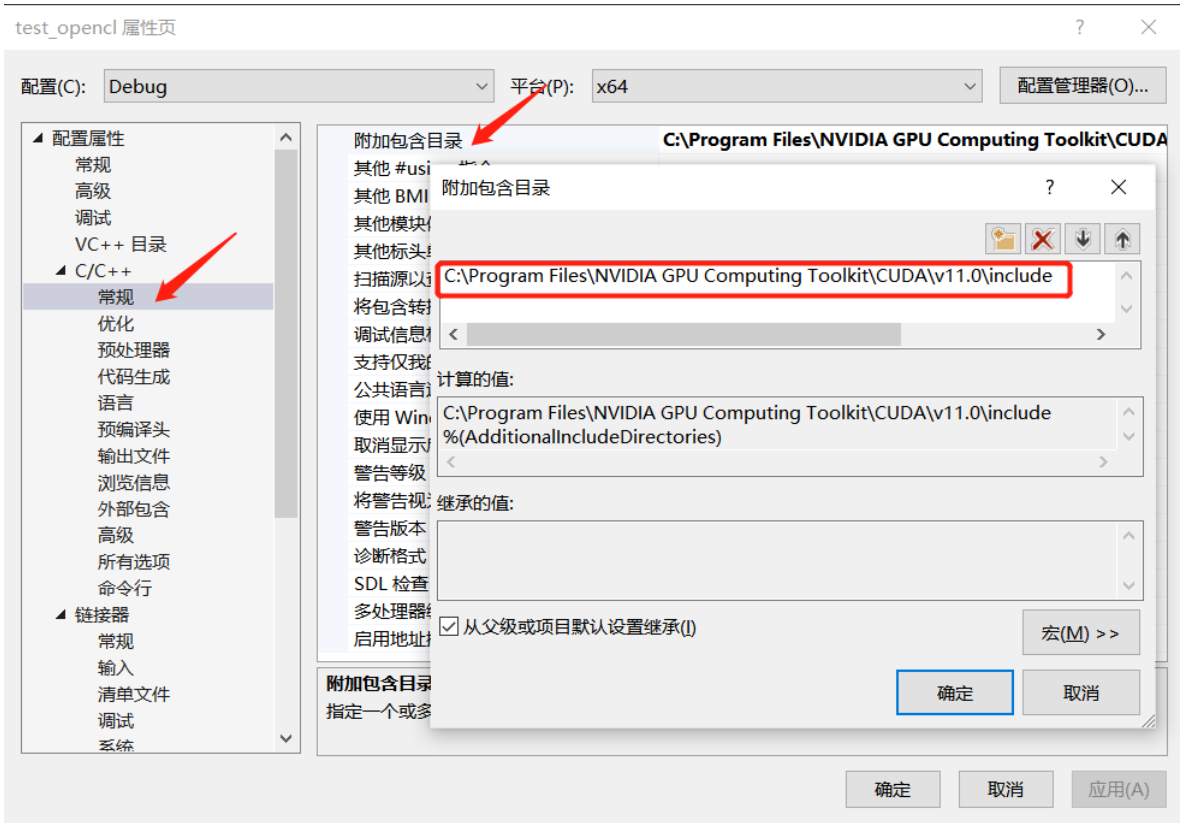
1.3 VS中配置opencl

前提: 下载好了cuda

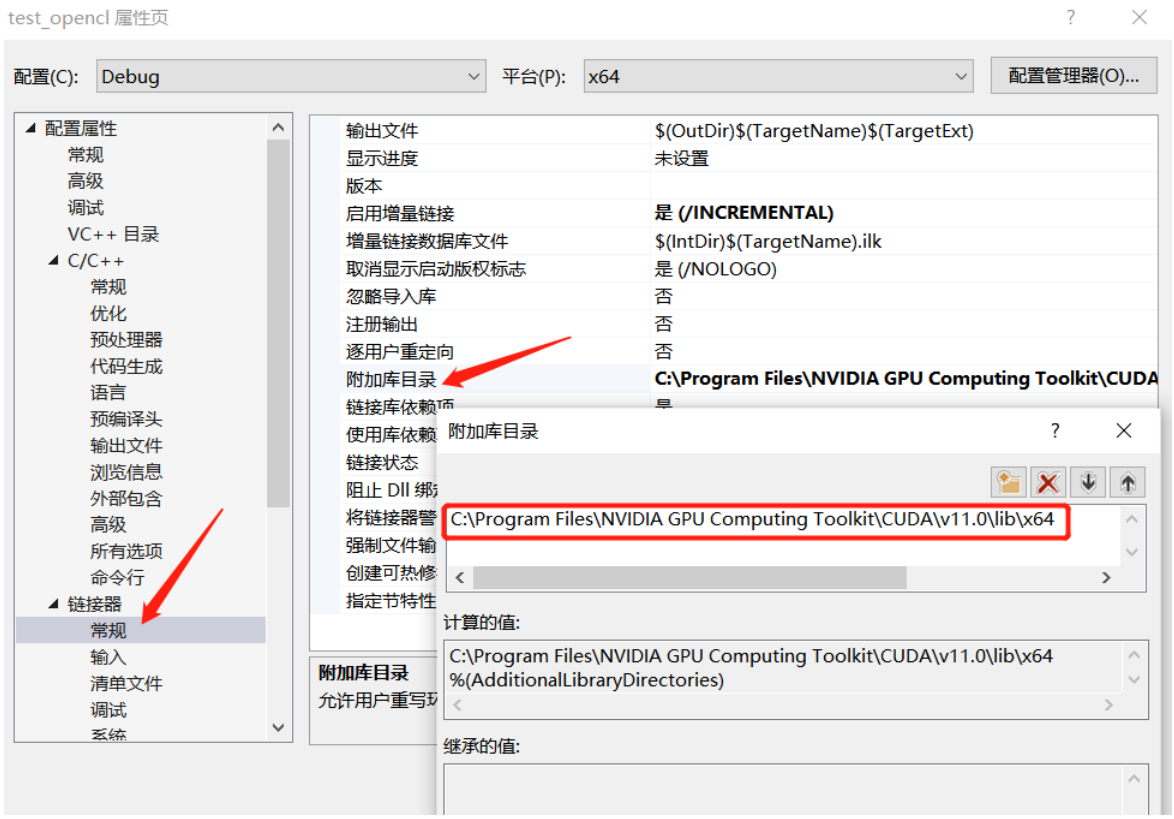
1. 在vs中新建一个项目，然后点击项目属性，选择配置debug,平台x64



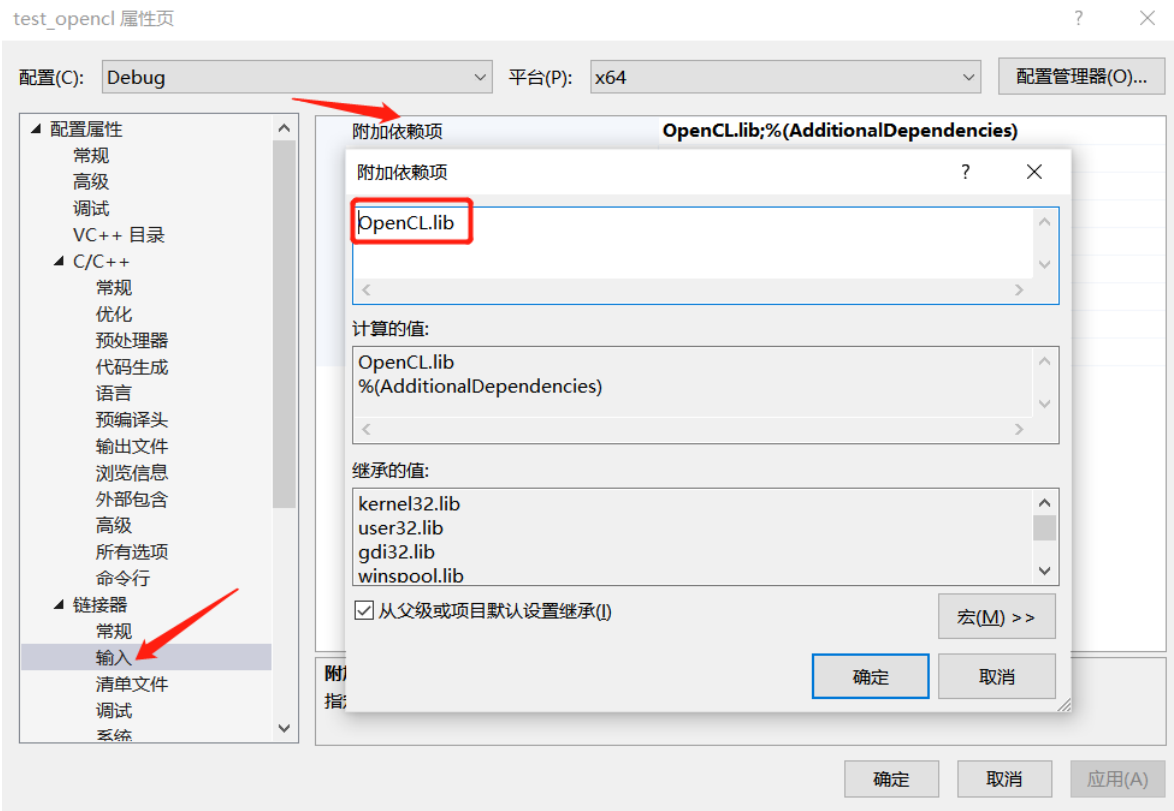
2. 添加附加包含目录



3. 添加附加库目录



4. 添加依赖项



2. OpenCL&Cuda编程

2.1 编程基础

opencl编程的基本流程

- **选择OpenCL平台并创建一个上下文**

平台（Platform）是指主机和OpenCL管理框架下的若干个设备构成的可以运行OpenCL程序的完整硬件系统，这个是跑OpenCL程序的基础，所以第一步要选择一个可用的OpenCL平台。一台机器上可以有不止一个这样的平台，一个平台也可以有不止一个GPU。

`clGetPlatformIDs()`: 用于获取可用的平台

`clCreateContextFromType()`: 创建一个openCL运行时的上下文环境

- **选择设备并创建命令队列**

选择平台并创建好OpenCL上下文环境之后，要做的事选择运行时用到的设备，还要创建一个命令队列，命令队列里定义了设备要完成的操作，以及各个操作的运行次序。

`clCreateCommandQueue()`: 用于创建一个指定设备上的上下文环境，第二个参数定义了选择的设备。

- **创建和构建程序对象**

程序对象用来存储与上下文相关联的设备的已编译可执行代码，同时也完成内核源代码的加载编译工作。

`clCreateProgramWithSource()`：这个函数会创建一个程序对象，在创建的同时，把已经转化成字符串形式的内核源代码加载到该程序对象中。

`clBuildProgram()`: 用于编译指定程序对象中的内核源代码，编译成功之后，再把编译代码存储在程序对象中。

- **创建内核和内存对象**

要执行程序对象中的已编译成功的内核运算，需要在内存中创建内核并分配内核函数的参数，在GPU上定义内存对象并分配存储空间。

`clCreateKernel`: 创建内核

`clCreateBuffer()`: 分配内存对象的存储空间，这些对象可以由内核函数直接访问。

- **设置内核数据并执行内核**

创建内核和内存对象之后，接下来要设置核函数的数据，并将要执行的内核排队。

`clEnqueueNDRangeKernel()` 用于设置内核函数的所有参与运算的数据。 利用命令队列对要在设备上执行的内核排队。需要注意的是，执行内核排队之后并不意味着这个内核一定会立即执行，只是排队到了执行队列中。

- 读取执行结果并释放OpenCL资源

内核执行完成之后，需要把数据从GPU拷贝到CPU中，供主机进一步处理，所有者写工作完成之后需要释放所有的OpenCL资源。

`clEnqueueReadBuffer()`: 读取设备内存数据到主机内存

`clReleaseXXX()`: 释放OpenCL的资源

2.2 编程案例

2.2.1 查看配置信息

运行CUDA的安装默认路径下的deviceQuery工程 `C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.0\1_Utilities\deviceQuery`，执行结果如下图：

Microsoft Visual Studio 调试控制台

```
C:\ProgramData\NVIDIA Corporation\CUDA Samples\v11.0\bin\win64\Debug\deviceQuery.exe Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA GeForce RTX 3050 Ti Laptop GPU"
  CUDA Driver Version / Runtime Version          11.4 / 11.0
  CUDA Capability Major/Minor version number:    8.6
  Total amount of global memory:                 4096 MBytes (4294967296 bytes)
MapSMtoCores for SM 8.6 is undefined.  Default to use 64 Cores/SM
MapSMtoCores for SM 8.6 is undefined.  Default to use 64 Cores/SM
  (20) Multiprocessors, ( 64) CUDA Cores/MP:     1280 CUDA Cores
  GPU Max Clock rate:                            1485 MHz (1.49 GHz)
  Memory Clock rate:                             6001 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 2097152 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1536
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 5 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  CUDA Device Driver Mode (TCC or WDDM):         WDDM (Windows Display Driver Model)
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA Runtime Version = 11.0, NumDevs = 1
Result = PASS
```

序号	名称	值	解释
1	Detected 1 CUDA Capable device(s)	1	检测到1个可用的NVIDIA显卡设备
2	Device 0: "NVIDIA GeForce RTX 3050 Ti Laptop GPU"	NVIDIA GeForce RTX 3050 Ti Laptop GPU	当前显卡型号为"NVIDIA GeForce RTX 3050 Ti Laptop GPU"
3	CUDA Driver Version / Runtime Version	11.4 / 11.0	CUDA驱动版本
4	**CUDA Capability Major/Minor version number**	8.6	CUDA设备支持的计算架构版本，即计算能力，该值越大越好
5	**Total amount of global memory**	4096Mbytes	Global memory全局存储器的大小。使用CUDA RUNTIME API调用函数cudaMalloc后，会消耗GPU设备上的存储空间，合理分配和释放空间避免程序出现crash
6	**(3) Multiprocessors, (128) CUDA Cores/MP**	1280 CUDA Cores	10个流多处理器（即SM），每个多处理器中包含128个流处理器，共1280 个CUDA核
7	GPU Max Clock rate	1485 MHz	GPU最大频率
8	Memory Clock rate	6001 MHz	显存的频率
9	Memory Bus Width	128-bit	内存总线带宽
10	L2 Cache Size	2097152 bytes	第二级缓冲区大小
11	Maximum Texture Dimension Size (x, y, z)	1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)	
12	Maximum Layered 1D Texture Size, (num) layers	1D=(32768), 2048 layers	
13	Maximum Layered 2D Texture Size, (num) layers	2D=(32768, 32768), 2048 layers	
14	**Total amount of constant memory**	65536 bytes	常量存储器的大小
15	**Total amount of shared memory per block**	49152 bytes	共享存储器的大小，共享存储器速度比全局存储器快；多处理器上的所有线程块可以同时共享这些存储器
16	Total number of registers available per block	65536	
17	**Warp Size**	32	Warp，线程束，是SM运行的最基本单位，一个线程束含有32个线程
18	**Maximum number of threads per multiprocessor**	1536	一个SM中最多有2048个线程，即一个SM中可以有2048/32=64个线程束Warp
19	**Maximum number of threads per block**	1024	一个线程块最多可用的线程数目
20	**Max dimension size of a thread block (x, y, z)**	(1024,1024,64)	ThreadIdx.x<=1024,ThreadIdx.y<=1024,ThreadIdx.z<=64Block内三维中各维度的最大值
21	**Max dimension size of a grid size** (x, y, z)	(2147483647,65535,65535)	Grid内三维中各维度的最大值
22	Maximum memory Pitch	2147483647 bytes	显存访问时对齐时的pitch的最大值
23	Texture alignment	512 bytes	纹理单元访问时对其参数的最大值
24	Concurrent copy and kernel execution	Yes with 1 copy engine(s)	并发复制和内核执行
25	Run time limit on kernels	Yes	内核上的运行时间限制

序号	名称	值	解释
26	Integrated GPU sharing Host Memory	No	集成GPU共享主机内存
27	Support host page-locked memory mapping	Yes	支持主机页面锁定内存映射
28	Alignment requirement for Surfaces	Yes	表面对中要求
29	Device has ECC support	Disabled	设备是否有ECC支持
30	CUDA Device Driver Mode (TCC or WDDM)	WDDM (Windows Display Driver Model)	CUDA设备驱动模式(TCC或WDDM)
31	Device supports Unified Addressing (UVA)	Yes	设备支持统一寻址(UVA)
32	Device supports Managed Memory:	Yes	设备支持托管内存
33	Device supports Compute Preemption:	Yes	设备支持计算抢占
34	Supports Cooperative Kernel Launch:	Yes	支持协作内核启动
35	Supports MultiDevice Co-op Kernel Launch:	No	不支持多设备合作内核启动
36	Device PCI Domain ID / Bus ID / location ID:	0 / 1 / 0	Device PCI Domain ID / Bus ID / location ID

2.2.2. 向量运算

实验目的

利用OpenCL平台模型进行规模大的向量加法运算

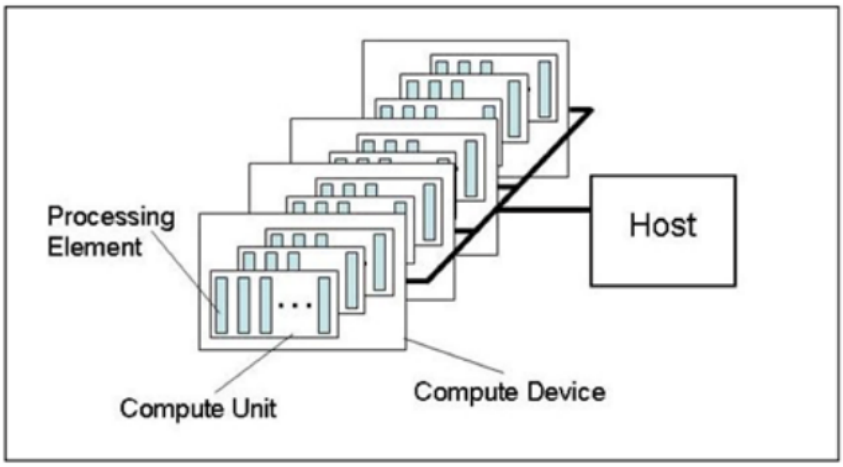
实验步骤

opencl 代码分析

基于openCL的实验代码分析如下：

- 首先进行定义变量，并初始化。

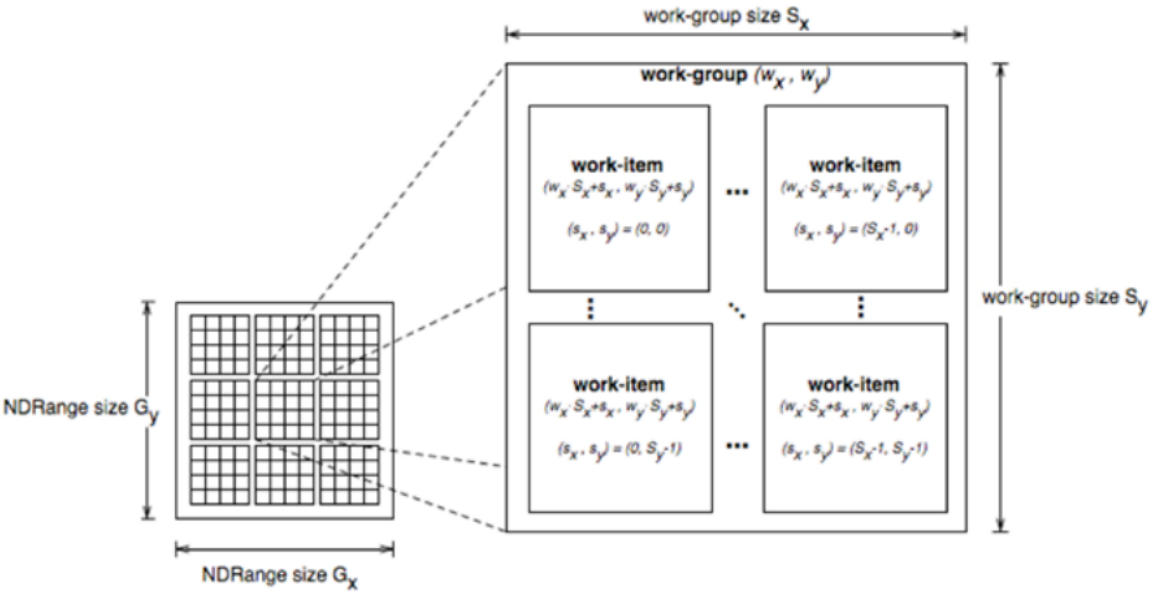
OpenCL 平台通常包括一个主机 (Host) 和多个 OpenCL 设备（device），每个 OpenCL 设备包括一个或多个 CU(compute units)，每个 CU 包括又一个或多个 PE（process element）。每个 PE 都有自己的程序计数器 (PC)。




```
 int i=0;
 size_t globalSize, localSize;
 cl_int err;
 double sum = 0;
// 向量的维度大小n
int n = 100000;

//在主机中指向待相加的向量a,b的指针 h_a,h_b
double *h_a;
double *h_b;
//在主机中指向存放求和结果c的指针h_c
double *h_c;

//设备中存放待相加数据的缓冲区
cl_mem d_a;
cl_mem d_b;
//设备中存放求和结果数据的缓冲区
cl_mem d_c;

cl_platform_id platform;          // OpenCL 平台
cl_device_id device_id;           // 设备的ID
cl_context context;               // 上下文
cl_command_queue queue;           // 命令队列
cl_program program;               // 程序
cl_kernel kernel;                 // 核函数

// 按照字节的方式统计一个向量占用缓冲区大小
size_t bytes = n*sizeof(double);

// 在主机当中为abc向量开辟内存空间，并用指针h_a,h_b,h_c指向该内存空间
h_a = (double*)malloc(bytes);
h_b = (double*)malloc(bytes);
h_c = (double*)malloc(bytes);

//初始化主机中的向量数据，使得分别为sin^2和cos^2，目的是求和时结果为1
for(  i = 0; i < n; i++ ){
    h_a[i] = sinf(i)*sinf(i);
    h_b[i] = cosf(i)*cosf(i);
}
```

- 定义globalSize 和 localSize 的daxiao

OpenCL 中的线程结构是可缩放的，Kernel 的每个运行实例称作 WorkItem(也就是线程），WorkItem 组织在一起称 作 WorkGroup，OpenCL 中，每个 workgroup 之间都是相互独立的。通过一个 global id(在索引空间，它是唯一的）或者 一个 work group id 和一个 work group 内的 local id，就能标定一个 workitem。




```
//规定每个work-group 中work-item的数量为64
localSize = 64;

// 计算所占用的work group块的所有空间大小
globalSize = ceil(n/(float)localSize)*localSize;// work-group 的数量用ceil函数向上取整求得n/localsize, 然后再乘上localsize就时globalsize
```

- 选择一个 OpenCL 平台

```
    err = clGetPlatformIDs(1, &platform, NULL); //第一次调用，得到系统中可使用的平台数目，为（Platform）平台对象分配空间
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL); //第二次调用，是查询所有的平台，选择自己需要的 OpenCL 平台
```

通常调用`clGetPlatformIDs`来获得平台，这个函数要调用 2 次，第一次得到系统中可使用的平台数目，然后为（Platform）平台对象分配空间，第二次 调用就是查询所有的平台，选择自己需要的 OpenCL 平台。将函数的返回值赋给err, 如果函数执行正确，err的值是 0。

- 设备与context关联

Context 是指管理 OpenCL 对象和资源的上下文环境。为了管 理 OpenCL 程序，下面的一些对象都要和 Context 关联起来： • 设备（Devices）: 执行 Kernel 程序对象。 • 程序对象（Program objects）: kernel 程序源代码 • Kernels: 运行在 OpenCL 设备上的函数 • 内存对象（Memory objects）: 设备上存放数据 • 命令队列（Command queues）: 设备的交互机制 • 内存命令（Memory commands）(用于在主机内存和设备内存之间拷贝数据) • Kernel 执行（Kernel execution） • 同步（Synchronization）

```
context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
```

`clCreateContext` 函数指定了和 context 关联的一个或多个设备对象，传入的参数device_id表示绑定的设备编号，如参数值为NULL,则厂商选 择的缺省值被使用.

- 命令队列

在 OpenCL 中，命令队列就是主机的请求，在设备上执行的一种机制。命令队列在 device 和 context 之间建立了一个连接。 对于不同的设备，它们都有自己的独立的命令队列；命令队列中的命令 (kernel 函数）可能是 同步的，也可能是异步的，它们的执行顺序可以是有序的，也可以是乱序的。

```
queue = clCreateCommandQueue(context, device_id, 0, &err);
```

- Opencl程序对象

程序对象就是通过读入 Kernel 函数源代码或二进制文件，然后在指定的设备上进行编译而产生的 OpenCL 对象。

```
program = clCreateProgramWithSource(context, 1,
                            (const char **) & kernelSource, NULL, &err);
```

`clCreateProgramWithSource` 个函数通过源代码 (kernelSource,)，创建一个程序对象，其中 counts=1 指定源代码串的数量，lengths 指定源代码串的长度 （为 NULL 结束的串时，可以省略）

此外还需要用 `clBuildProgram` 编写一个从文件中读取源代码串的函数。

```
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

对 context 中的每个设备，这个函数编译、连接源代码对象，产生 device 可以执行的文件

- Kernel 对象

Kernel 就是在程序代码中的一个函数，这个函数能在 OpenCL 设备上执行。一个 Kernel 对象就是 kernel 函数以及 其相关的输入参数。Kernel 对象通过程序对象以及指定的函数名字创建。

```
    kernel = clCreateKernel(program, "vecAdd", &err);
```

然后为Kernel 对象设置参数。我们可以在 Kernel 运行后，重新设置参数再 次运行。

```
    err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
    err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
    err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
```

调用函数 `clSetKernelArg` , 该函数的第二个参数指定该参数为 Kernel 函数中的第几个参数 (比如第一个参数为 0，第二个为 1,...)。内存对象和单个的值都 可以作为 Kernel 参数。

- Opencl 内存对象

  OpenCL 内存对象就是一些 OpenCL 数据，这些数据一般在设备内存中，能够被拷入也能够被拷出。将d_a, d_b指定为只读的，d_c指定为只写的。函数创建 OpenCL buffer 对象后，会把对应 host buffer 的内容拷贝到 OpenCL buffer 中。

```
    d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
    d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
    d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);
```

在 Kernel 执行之前，host 中原始输入数据必须显式的传到 device 中，Kernel 执行完后，结果也要从 device 内存中传 回到 host 内存中。我们主要通过函数 clEnqueue{Read/Write}Buffer/Image} 来实现这两种操作。从 host 到 device， 我们用 clEnqueueWrite，从 device 到 host，我们用 clEnqueueRead。clEnqueueWrite 命令包括初始化内存对象以及把 host 数据传到 device 内存这两种操作。

```
    err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0,
                                    bytes, h_a, 0, NULL, NULL);
    err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0,
                                    bytes, h_b, 0, NULL, NULL);
```

通过`clEnqueueWriteBuffer`函数将主机中的h_a,和h_b分别传到设备内存中的d_a,d_b。

```
    clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0,
                                  bytes, h_c, 0, NULL, NULL );
```

通过`clEnqueueWriteBuffer`函数将设备中计算结果d_c送回到内存中的h_c。

- kernel执行

通过函数`clEnqueueNDRangeKerne`把 Kernel 放在一个队列里，但不保证它马上执行，OpenCL driver 会管理 队列，调度 Kernel 的执行。注意：每个线程执行的代码都是相同的，但是它们执行数据却是不同的。

```
   err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize,
&localSize, 0, NULL, NULL);
```

- 主机中计算求和

在本地计算对h_c指向的缓冲区的值进行累加求得sum并输出

```
    clFinish(queue);
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);
```

- 释放资源

OpenCL 资源都是指针，不使用的时候需要用`clReleaseMemObject`,`clReleaseProgram`,`clReleaseKernel`,`clReleaseCommandQueue`,`clReleaseContext`释放掉在设备中的内存空间。而在主机中的内存空间是用malloc开辟的，因此对应的要用`free`函数释放。

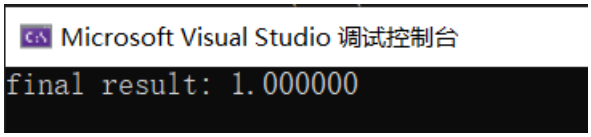
```
    //释放OPencl的空间
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);

    //释放主机的内存空间
    free(h_a);
    free(h_b);
    free(h_c);
```

cuda 代码分析

```
#include <stdio.h>
#include <cuda.h>

// 运行在cuda设备上的核函数
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;//id编号为块数*每个块的大小
  if (idx<N) a[idx] = a[idx] * a[idx];//计算每个Id对应的平方和
}

int main()
{
  float *a_h, *a_d;  // a_h指向主机数组内存首地址，a_d指向设备数组内存首地址
  const int N = 10;  // 数组中一共10个元素
  size_t size = N * sizeof(float);  //数组缓冲区的大小
  a_h = (float *)malloc(size);        // 分配主机上的数组内内存存
  cudaMalloc((void **) &a_d, size);   // 分配设备上的数组
  // 初始化主机上的数组
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  //把主机上的数组拷贝到设备的内存中
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  int block_size = 32;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1); //这一步主要是为了向上取整
  //对于大小为block_size 的n_blocks个块计算平方和
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // 将设备上计算的结果拷贝到主机中
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // 打印结果
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // 释放主机和设备的资源
  free(a_h); cudaFree(a_d);
  return 0;
}
```

实验结果

opencl 代码运行结果：



cuda 代码运行结果：

结果分析

opencl的代码输出结果是$sin^2 + cos^2$,其对应的结果应该是1。

cuda的代码计算的是对等差数组（d=1）中的每个数求平方, 故其结果分别是0，1，4，9...

2.2.3 Pi计算

实验目的

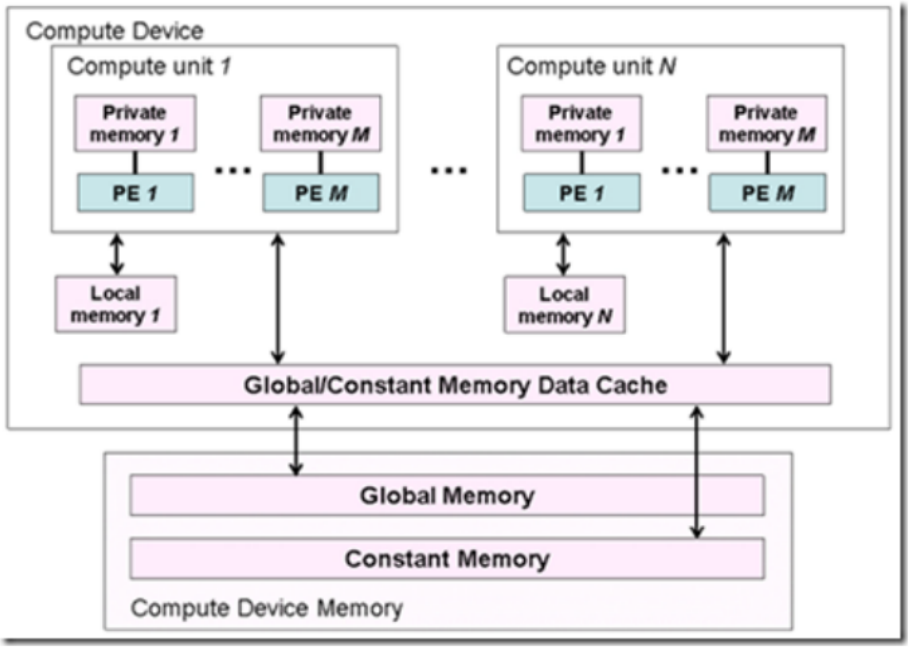
利用OpenCL平台模型进行pi值的计算

实验步骤

opencl代码分析

- 核函数

每个线程（workitem）都有一个 kenerl 函数的实例。每个 Kernel 函数都必须以 __kernel 开始，而且必须返回 void。每个输入参数都必须声明使用的内存类型。通过 一些 API，比如 get_global_id 之类的得到线程 id。WorkGroup 被映射到硬件的 CU 上执行。



响应的内存对象的地址空间标识符有一下几种：

```
__global  //主机中的全局内存空间
__constant //主机中一块只读内存，不允许修改
__local  // work group 共享的内存空间
__private //每个work item 私有的内存空间
```

- 核函数的定义

```
const char* kernelSource = KERNEL(
    __kernel void Pi(__global float* workGroupBuffer, // 0..NumWorkGroups-1
        __local float* insideWorkGroup,  // 0..workGroupSize-1
        const uint n,        // Total iterations
        const uint chunk)        // Chunk size
{
    const uint lid = get_local_id(0); // work group内每个work_item的编号
    const uint gid = get_global_id(0); //work group编号

    const float step = (1.0 / (float)n); //步长
    float partial_sum = 0.0; // 每个小部分的和

    // 每一个work item 迭代chunks 次数（即计算chunks个小长方形）
    for (uint i = gid * chunk; i < (gid * chunk) + chunk; i++) {
        float x = step * ((float)i - 0.5);
        partial_sum += 4.0 / (1.0 + x * x);
    }

    // insideWorkGroup[lid]存放着work item 编号为lid的分组计算的值
    insideWorkGroup[lid] = partial_sum;

    //设置一个barrier，使得所有的work item 都完成运算
    barrier(CLK_LOCAL_MEM_FENCE);

    float local_pi = 0;

    // 然后对work group 中的所有work item 计算的值累加
    if (lid == 0) {
```

```
        const uint length = lid + get_local_size(0);//work item的个数
        for (uint i = lid; i < length; i++) {
            local_pi += insideWorkGroup[i];
        }
        //最终对所有块的值进行规约
        workGroupBuffer[get_group_id(0)] = local_pi;
    }
  }

```

- 主函数变量初始化

```
    int i = 0;
    float pi; //最终的pi值
    float* pi_partial;// 每个部分的pi
    size_t maxWorkGroupSize;//work group的最大值
    cl_int err;//
    cl_mem memObjects; //
    int niter, chunks, workGroups;//
    size_t globalWorkSize;//work group的总大小
    size_t localWorkSize;// 一个work group 的大小

    cl_platform_id platform;        // OpenCL 平台
    cl_device_id device_id;           // 设备 ID
    cl_context context;               // 上下文
    cl_command_queue queue;           // 命令队列
    cl_program program;               // 程序
    cl_kernel kernel;                 // 核

    niter = 262144;  //总计算次数（分割成的小长方形数）
    chunks = 64;
```

- 主函数程序运行逻辑

```
    //获取平台的ID
    err = clGetPlatformIDs(1, &platform, NULL);
    //获取设备的ID
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id, NULL);
    //获取设备信息
    clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
sizeof(size_t),
        &maxWorkGroupSize, NULL);
    //work group的数量=（总迭代次数/chunk数）/每个work group的最大值
    workGroups = ceil((float)(niter / maxWorkGroupSize / chunks));
    //为每个部分计算pi的中间结果分配内存空间
    pi_partial = (float*)malloc(sizeof(float) * workGroups);
    // 创建上下文
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    // 创建命令队列
    queue = clCreateCommandQueue(context, device_id, 0, &err);
    //创建程序
    program = clCreateProgramWithSource(context, 1,
        &kernelSource, NULL, &err);
    //建立可执行程序
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
```

```
    localWorkSize = maxWorkGroupSize;//每个work group 中work item的数量
    globalWorkSize = niter / chunks; //所有的work group 的大小

    // 创建核函数
    kernel = clCreateKernel(program, "Pi", &err);

    // 创建 OpenCL 设备缓冲 (buffer)
    memObjects = clCreateBuffer(context, CL_MEM_READ_WRITE,
        sizeof(float) * workGroups, NULL, &err);
    //创建 Kernel 后，运行 Kernel 之前，为 Kernel 对象设置参数
    err |= clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects);
    err = clSetKernelArg(kernel, 1, sizeof(float) * maxWorkGroupSize, NULL);
    err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &niter);
    err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &chunks);
    //把 Kernel 放在一个队列里
    err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalWorkSize,
&localWorkSize,
        0, NULL, NULL);
    //结束命令队列
    clFinish(queue);
    //从设备中读取每个pi_partial 的值
    err = clEnqueueReadBuffer(queue, memObjects, CL_TRUE, 0,
        sizeof(float) * workGroups, pi_partial, 0, NULL, NULL);
    pi = 0;
    //cpu将每个pi_paritical 的值累加在一起
    for (i = 0; i < workGroups; i++) {
        pi += pi_partial[i];
    }
    //计算最终的pi值
    pi *= (1.0 / (float)niter);
    printf("final result: %f\n", pi);
    //释放OpenCL资源
    clReleaseMemObject(memObjects);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
    //释放主机资源
    free(pi_partial);
    return 0;
}
```

cuda 代码分析

- 核函数的定义

```
__global__ void cal_pi(double *sum, long long nbin, float step, long long nthreads, long long nblocks) {
    long long i;
    float x;
    long long idx = blockIdx.x*blockDim.x+threadIdx.x; //每一个work  item的编号=块编号*块的维度+块内编号
    for (i=idx; i< nbin; i+=nthreads*nblocks) {//每个线程每隔
        x = (i+0.5)*step;
        sum[idx] = sum[idx]+4.0/(1.+x*x);
    }
}
```

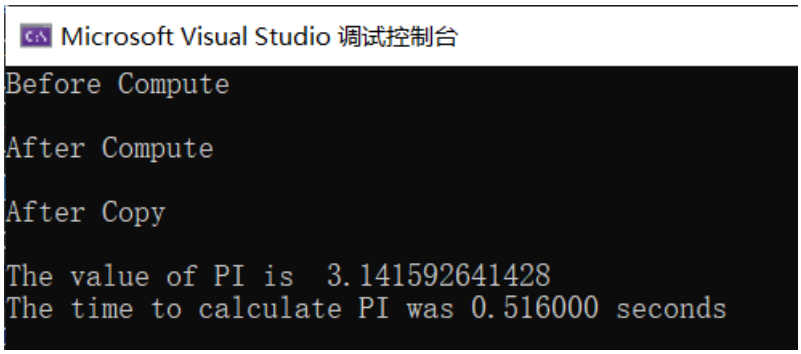
- 主函数的编写

```
int main()
{
    long long tid;
    double pi = 0;
    long long num_steps = 100000000;//总共迭代的次数
    float step = 1./(float)num_steps; //步长
    long long size = NUM_THREAD*NUM_BLOCK*sizeof(double);//总线程数
    clock_t before, after; //时钟标志
    double *sumHost, *sumDev;
    sumHost = (double *)malloc(size);//为主机分配内存空间
    cudaMalloc((void **)&sumDev, size);//为设备分配内存空间
    // 初始化为全0
    cudaMemset(sumDev, 0, size);
    before = clock();
    // Do calculation on device
    printf("Before Compute \n\n");
    dim3 numBlocks(NUM_BLOCK,1,1);
    dim3 threadsPerBlock(NUM_THREAD,1,1);
    cal_pi <<<numBlocks, threadsPerBlock>>> (sumDev, (int)num_steps, step, NUM_THREAD, NUM_BLOCK); //调用kernel 函数实现pi值的计算

    printf("After Compute \n\n");
    //将设备的计算结果复制到主机中
    cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
    printf("After Copy \n\n");
    for(tid=0; tid<NUM_THREAD*NUM_BLOCK; tid++){ //对每一个线程的计算结果累加
        pi = pi+sumHost[tid];
    }
    pi = pi*step;//最后乘以步长，得到最终的pi值
    after = clock();
    printf("The value of PI is %15.12f\n",pi);
    printf("The time to calculate PI was %f seconds\n",((float)(after - before)/1000.0));
    free(sumHost); //释放主机内存空间
    cudaFree(sumDev); //释放设备的内存空间
    return 0;
}
```

实验结果

opencl代码结果

cuda代码结果



结果分析

opencl和cuda版本都可以正确的计算出pi值的大小，但是cuda版本设置的迭代次数要更多，因此更加准确。

3. 总结

openCL 和cuda编程的核心思想是一样的，多核 cpu 就适合基于任务的并行编程，而 GPU 更适应于数据并行编程，即 GPU 上执行一个任务，都是把任务中的数据分配到各个独立 的 core 中执行。

无论是计算矩阵求和，或者是计算pi，是把本身在CPU上串行进行的代码改成在GPU上并行计算的。GPU 上，都是轻量级的线程，创建、调度线程的开销比较小，所以我们 可以做到把循环完全展开，一个线程处理一个数据。最后把每个线程的运算结果从设备复制到主机中，再在CPU上进行后续的处理,如累加（计算pi）,或者输出数组中的每个值（计算数组和）

4. 附录

cuda 完整代码

1. **计算pi**

```
#include <stdio.h>
#include <cuda.h>
#include <math.h>
#define NUM_THREAD 1024
#define NUM_BLOCK 1

__global__ void cal_pi(double *sum, long long nbin, float step, long long nthreads, long long nblocks) {

    long long i;
    float x;
    long long idx = blockIdx.x*blockDim.x+threadIdx.x;

```

```
    for (i=idx; i< nbin; i+=nthreads*nblocks) {
        x = (i+0.5)*step;
        sum[idx] = sum[idx]+4.0/(1.+x*x);
    }

}

int main()
{
    long long tid;
    double pi = 0;
    long long num_steps = 100000000;

    float step = 1./(float)num_steps;
    long long size = NUM_THREAD*NUM_BLOCK*sizeof(double);
    clock_t before, after;
    double *sumHost, *sumDev;
    sumHost = (double *)malloc(size);
    cudaMalloc((void **)&sumDev, size);
    // Initialize array in device to 0
    cudaMemset(sumDev, 0, size);
    before = clock();
    // Do calculation on device
    printf("Before Compute \n\n");
    dim3 numBlocks(NUM_BLOCK,1,1);
    dim3 threadsPerBlock(NUM_THREAD,1,1);
    cal_pi <<<numBlocks, threadsPerBlock>>> (sumDev, (int)num_steps, step,
NUM_THREAD, NUM_BLOCK); // call CUDA kernel

    printf("After Compute \n\n");
    // Retrieve result from device and store it in host array
    cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
    printf("After Copy \n\n");
    for(tid=0; tid<NUM_THREAD*NUM_BLOCK; tid++){
        pi = pi+sumHost[tid];
    }
    pi = pi*step;
    after = clock();
    printf("The value of PI is %15.12f\n",pi);
    printf("The time to calculate PI was %f seconds\n",((float)(after -
before)/1000.0));
    free(sumHost);
    cudaFree(sumDev);
    return 0;
}
```

2. 向量运算

```
#include <stdio.h>
#include <cuda.h>

// Kernel that executes on the CUDA device
__global__ void square_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
```

```
  if (idx<N) a[idx] = a[idx] * a[idx];
}

int main()
{
  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 10;  // Number of elements in arrays
  size_t size = N * sizeof(float);
  a_h = (float *)malloc(size);        // Allocate array on host
  cudaMalloc((void **) &a_d, size);   // Allocate array on device
  // Initialize host array and copy it to CUDA device
  for (int i=0; i<N; i++) a_h[i] = (float)i;
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  // Do calculation on device:
  int block_size = 32;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);
  // Retrieve result from device and store it in host array
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
  // Print results
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  // Cleanup
  free(a_h); cudaFree(a_d);
  return 0;
}
```

openCL 完整代码

1. **计算Pi**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <CL/opencl.h>


// OpenCL kernel. many workGroups compute n iterations
#define KERNEL(...) #__VA_ARGS__
const char* kernelSource = KERNEL(
    __kernel void Pi(__global float* workGroupBuffer, // 0..NumWorkGroups-1
        __local float* insideWorkGroup,  // 0..workGroupSize-1
        const uint n,        // Total iterations
        const uint chunk)        // Chunk size
{
    const uint lid = get_local_id(0);
    const uint gid = get_global_id(0);

    const float step = (1.0 / (float)n);
    float partial_sum = 0.0;

    // Each work-item computes chunk iterations
    for (uint i = gid * chunk; i < (gid * chunk) + chunk; i++) {
        float x = step * ((float)i - 0.5);
        partial_sum += 4.0 / (1.0 + x * x);
```

```
    }

    // Each work-item stores its partial sum in the workgroup array
    insideWorkGroup[lid] = partial_sum;

    // Synchronize all threads within the workgroup
    barrier(CLK_LOCAL_MEM_FENCE);

    float local_pi = 0;

    // Only work-item 0 of each workgroup perform the reduction
    // of that workgroup
    if (lid == 0) {
        const uint length = lid + get_local_size(0);
        for (uint i = lid; i < length; i++) {
            local_pi += insideWorkGroup[i];
        }
        // It store the workgroup sum
        // Final reduction, between block, is done out by CPU
        workGroupBuffer[get_group_id(0)] = local_pi;
    }
  }
);

int main(int argc, char* argv[])
{
    int i = 0;
    float pi;
    float* pi_partial;
    size_t maxWorkGroupSize;
    cl_int err;
    cl_mem memObjects;
    int niter, chunks, workGroups;
    size_t globalWorkSize;
    size_t localWorkSize;


    cl_platform_id platform;        // OpenCL platform
    cl_device_id device_id;           // device ID
    cl_context context;               // context
    cl_command_queue queue;           // command queue
    cl_program program;               // program
    cl_kernel kernel;                 // kernel

    niter = 262144;
    chunks = 64;


    err = clGetPlatformIDs(1, &platform, NULL);

    // Get ID for the device
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
NULL);
    clGetDeviceInfo(device_id, CL_DEVICE_MAX_WORK_GROUP_SIZE,
sizeof(size_t),
        &maxWorkGroupSize, NULL);

    workGroups = ceil((float)(niter / maxWorkGroupSize / chunks));
```

```
82
83      pi_partial = (float*)malloc(sizeof(float) * workGroups);
84
85      // Create a context
86      context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
87
88      // Create a command queue
89      queue = clCreateCommandQueue(context, device_id, 0, &err);
90
91      // Create the compute program from the source buffer
92
93      program = clCreateProgramWithSource(context, 1,
94          &kernelSource, NULL, &err);
95      // Build the program executable
96      err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
97      localWorkSize = maxWorkGroupSize;
98      globalWorkSize = niter / chunks;
99
100     // Create the compute kernel in the program we wish to run
101     kernel = clCreateKernel(program, "Pi", &err);
102
103     // Create the input and output arrays in device memory for our
    calculation
104     memObjects = clCreateBuffer(context, CL_MEM_READ_WRITE,
105         sizeof(float) * workGroups, NULL, &err);
106
107     err |= clSetKernelArg(kernel, 0, sizeof(cl_mem), &memObjects);
108     err = clSetKernelArg(kernel, 1, sizeof(float) * maxWorkGroupSize,
    NULL);
109     err |= clSetKernelArg(kernel, 2, sizeof(unsigned int), &niter);
110     err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &chunks);
111
112     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalWorkSize,
    &localWorkSize,
113         0, NULL, NULL);
114     clFinish(queue);
115     err = clEnqueueReadBuffer(queue, memObjects, CL_TRUE, 0,
116         sizeof(float) * workGroups, pi_partial, 0, NULL, NULL);
117     pi = 0;
118
119     for (i = 0; i < workGroups; i++) {
120         pi += pi_partial[i];
121     }
122     pi *= (1.0 / (float)niter);
123     printf("final result: %f\n", pi);
124
125     // release OpenCL resources
126     clReleaseMemObject(memObjects);
127     clReleaseProgram(program);
128     clReleaseKernel(kernel);
129     clReleaseCommandQueue(queue);
130     clReleaseContext(context);
131
132     //release host memory
133     free(pi_partial);
134
135     return 0;
136 }
```

2. 向量运算

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <CL/opencl.h>

// OpenCL kernel. Each work item takes care of one element of c
const char *kernelSource =                                      "\n" \
"#pragma OPENCL EXTENSION cl_khr_fp64 : enable                   \n" \
"__kernel void vecAdd(  __global double *a,                      \n" \
"                       __global double *b,                      \n" \
"                       __global double *c,                      \n" \
"                       const unsigned int n)                   \n" \
"{                                                              \n" \
"    //Get our global thread ID                                  \n" \
"    int id = get_global_id(0);                                 \n" \
"                                                               \n" \
"    //Make sure we do not go out of bounds                     \n" \
"    if (id < n)                                                \n" \
"        c[id] = a[id] + b[id];                                 \n" \
"}                                                              \n" \
                                                               "\n" ;

int main( int argc, char* argv[] )
{
     int i=0;
     size_t globalSize, localSize;
     cl_int err;
     double sum = 0;

    // Length of vectors
   // unsigned int n = 100000;
 int n = 100000;
    // Host input vectors
    double *h_a;
    double *h_b;
    // Host output vector
    double *h_c;

    // Device input buffers
    cl_mem d_a;
    cl_mem d_b;
    // Device output buffer
    cl_mem d_c;

    cl_platform_id platform;        // OpenCL platform
    cl_device_id device_id;           // device ID
    cl_context context;               // context
    cl_command_queue queue;           // command queue
    cl_program program;               // program
    cl_kernel kernel;                 // kernel

    // Size, in bytes, of each vector
```

```
    size_t bytes = n*sizeof(double);

    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes);
    h_b = (double*)malloc(bytes);
    h_c = (double*)malloc(bytes);

    // Initialize vectors on host

    for(  i = 0; i < n; i++ ){
        h_a[i] = sinf(i)*sinf(i);
        h_b[i] = cosf(i)*cosf(i);
    }

   // size_t globalSize, localSize;

    //cl_int err;

    // Number of work items in each local work group
    localSize = 64;

    // Number of total work items - localSize must be devisor
    globalSize = ceil(n/(float)localSize)*localSize;

    // Bind to platform
    err = clGetPlatformIDs(1, &platform, NULL);

    // Get ID for the device
    err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device_id,
NULL);

    // Create a context
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);

    // Create a command queue
    queue = clCreateCommandQueue(context, device_id, 0, &err);

    // Create the compute program from the source buffer
    program = clCreateProgramWithSource(context, 1,
                            (const char **) & kernelSource, NULL, &err);

    // Build the program executable
    clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

    // Create the compute kernel in the program we wish to run
    kernel = clCreateKernel(program, "vecAdd", &err);

    // Create the input and output arrays in device memory for our
calculation
    d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
    d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, bytes, NULL, NULL);
    d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, bytes, NULL, NULL);

    // Write our data set into the input array in device memory
    err = clEnqueueWriteBuffer(queue, d_a, CL_TRUE, 0,
                                    bytes, h_a, 0, NULL, NULL);
    err |= clEnqueueWriteBuffer(queue, d_b, CL_TRUE, 0,
                                    bytes, h_b, 0, NULL, NULL);
```

```
109
110     // Set the arguments to our compute kernel
111     err  = clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_a);
112     err |= clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_b);
113     err |= clSetKernelArg(kernel, 2, sizeof(cl_mem), &d_c);
114     err |= clSetKernelArg(kernel, 3, sizeof(unsigned int), &n);
115
116     // Execute the kernel over the entire range of the data set
117     err = clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &globalSize,
    &localSize,
118                                                             0, NULL,
    NULL);
119
120     // Wait for the command queue to get serviced before reading back
    results
121     clFinish(queue);
122
123     // Read the results from the device
124     clEnqueueReadBuffer(queue, d_c, CL_TRUE, 0,
125                                 bytes, h_c, 0, NULL, NULL );
126
127     //Sum up vector c and print result divided by n, this should equal 1
    within error
128     //double sum = 0;
129     for(i=0; i<n; i++)
130         sum += h_c[i];
131     printf("final result: %f\n", sum/n);
132
133     // release OpenCL resources
134     clReleaseMemObject(d_a);
135     clReleaseMemObject(d_b);
136     clReleaseMemObject(d_c);
137     clReleaseProgram(program);
138     clReleaseKernel(kernel);
139     clReleaseCommandQueue(queue);
140     clReleaseContext(context);
141
142     //release host memory
143     free(h_a);
144     free(h_b);
145     free(h_c);
146
147     return 0;
148 }
```