**OpenCL 簡介**

OpenCL是由 [Khronos Group](http://www.khronos.org/) 針對異質性計算裝置(heterogeneous device)進行平行化運算所設計的標準API以及程式語言。「異質性計算裝置」是指在同一個電腦系統中，有兩種以上架構差異很大的計算裝置，例如一般的CPU以及顯示晶片，或是像CELL的PPE以及SPE。目前最為常見的是GPGPU應用，利用顯示晶片(即GPU)進行3D繪圖以外的計算工作。

最早的GPGPU，多半是直接透過3D繪圖的API進行，例如OpenGL或D3D的HLSL(High Level Shading Language)。這樣做有很多缺點，主要是即使想要進行的運算和3D繪圖無關，仍需要處理很多3D繪圖方面的動作(如建立texture，安排render-to-texture動作等等)。讓GPGPU變得複雜。後來出現把3D繪圖部份隱藏的想法，包括ATI針對其產品設計的 Close to Metal(以及後來的 AMD Stream)，以及 NVIDIA的CUDA。Microsoft也在DirectX 11中加入了特別為GPGPU 設計的 DirectCompute。

由於各家廠商的GPGPU方案互不相容，對GPGPU的發展不利，因為程式開發者必須為不同廠商的顯示晶片分別撰寫程式，或是選擇只支援某個顯示晶片廠商。由於顯示晶片的發展愈來愈彈性化，GPGPU 的應用範圍也增加，因此Apple提出統一的 GPGPU方案。這個方案得到包括AMD、IBM、Intel、NVIDIA等廠商的支持，並交由Khronos Group進行標準化。

OpenCL設計目的，是提供一個容易使用、且適用於各種不同裝置的平行化計算平台。它提供兩種平行化模式，包task parallel 及data parallel。目前GPGPU的應用，主要是以data parallel為主。data parallel是指大量的資料，都進行同樣的處理。如影像處理的程式，常要對影像的每個pixel進行同樣的動作(例如 Gaussian blur)。這類工作很適合data parallel的模式。

**OpenCL 的架構**

OpenCL包括一組API和一個程式語言。程式透過OpenCL API取得OpenCL裝置(例如顯示晶片)的相關資料，並將要在裝置上執行的程式(使用OpenCL程式語言撰寫)編繹成適當的格式，在裝置上執行。OpenCL API也提供許多裝置控制方面的動作，例如在 OpenCL裝置上取得一塊記憶體、把資料從主記憶體複製到OpenCL裝置上(或從OpenCL裝置上複製到主記憶體中)、取得裝置動作的資訊(例如上一個程式執行所花費的時間)等等。先考慮一個簡單的工作：把一群數字相加。在一般的 C 程式中，可能是如下：

*float a[DATA\_SIZE];*

*float b[DATA\_SIZE];*

*float result[DATA\_SIZE];*

*// ...*

*for(int i = 0; i < DATA\_SIZE; i++) {*

*result[i] = a[i] + b[i];*

*}*

在 OpenCL 中，則大致的流程是：

1. 把OpenCL裝置初始化。
2. 在OpenCL裝置上配置三塊記憶體，以存放a、b、c三個陣列的資料。
3. 把a陣列和b陣列的內容，複製到OpenCL裝置上。
4. 編譯要執行的OpenCL程式(稱為 kernel)。
5. 執行編譯好的kernel。
6. 把計算結果從 OpenCL 裝置上，複製到result陣列中。

透過data parallel的模式，這裡的OpenCL程式非常簡單，如下所示：

*\_\_kernel void adder(\_\_global const float\* a, \_\_global const float\* b, \_\_global float\* result) {*

*int idx = get\_global\_id(0);*

*result[idx] = a[idx] + b[idx];*

*}*  
在一般的版本中，是透過一個迴圈，執行DATA\_SIZE次數的加法動作。而在OpenCL中，則是建立DATA\_SIZE個work item，每個work item都執行上面所示的 kernel。OpenCL 程式語言和一般的C 語言非常類似。*\_\_kernel*表示這個函式在 OpenCL裝置上執行的。*\_\_global* 則表示這個指標是在 global memory 中(即OpenCL 裝置上的主要記憶體)。而 *get\_global\_id(0)*會傳回work item的編號，例如有1024個work item，則編號會分別是0 ~ 1023(實際上編號可以是二維或三維，但在這裡先只考慮一維的情形)。

**開始撰寫 OpenCL 程式**

在使用OpenCL API前，都要include相關的header檔案。如下所示：

*#ifdef \_\_APPLE\_\_*

*#include <OpenCL/opencl.h>*

*#else*

*#include <CL/cl.h>*

*#endif*

接著要取得系統上所有的OpenCL platform。在MacOS X 10.6下，目前只有一個由Apple提供的OpenCL platform，但是在其它系統上，可能會有不同廠商提供的多個不同的OpenCL platform，因此需要先取得platform的數目(See <http://stackoverflow.com/questions/3444664/does-any-opencl-host-have-more-than-one-platform> for detailed info)：

*cl\_int err;*

*cl\_uint num;*

*err = clGetPlatformIDs(0, 0, &num);*

*if(err != CL\_SUCCESS) {*

*std::cerr << "Unable to get platforms\n";*

*return 0;*

*}*

大部份的OpenCL API會傳回錯誤值。如果傳回值是*CL\_SUCCESS*表示執行成功，否則會傳回某個錯誤值，表示失敗的原因。接著再取得platform的ID，在建立OpenCL context時會用到：

*std::vector<cl\_platform\_id> platforms(num);*

*err = clGetPlatformIDs(num, &platforms[0], &num);*

*if(err != CL\_SUCCESS) {*

*std::cerr << "Unable to get platform ID\n";*

*return 0;*

*}*

在OpenCL中，類似模式常出現：先呼叫第一次以取得數目，以便配置足夠的記憶體量。接著再呼叫第二次，取得實際的資料。接下來，要建立一個OpenCL context。如下：

*cl\_context\_properties prop[] = { CL\_CONTEXT\_PLATFORM, reinterpret\_cast<cl\_context\_properties>(platforms[0]), 0 };*

*cl\_context context = clCreateContextFromType(prop, CL\_DEVICE\_TYPE\_DEFAULT, NULL, NULL, NULL);*

*if(context == 0) {*

*std::cerr << "Can't create OpenCL context\n";*

*return 0;*

*}  
clReleaseContext(context);*return 0;

*clCreateContextFromType* 是一個OpenCL的API，它可以從指定的裝置類別中，建立一個 OpenCL context。第一個參數是指定context的property。在OpenCL中，透過一個property的陣列，以「property 種類」及「property 內容」成對出現，並以0做為結束。例如上面的例子，指定的property種類是*CL\_CONTEXT\_PLATFORM*，即要使用的platform ID，而property內容則由之前取得的 platform ID中的第一個(即 platforms[0])。由於property的內容可能是不同的資料型態，因此要使用*reinterpret\_cast*強制轉型。第二個參數可以指定要使用的裝置類別。目前可以使用的類別包括：

* *CL\_DEVICE\_TYPE\_CPU*：使用 CPU 裝置
* *CL\_DEVICE\_TYPE\_GPU*：使用顯示晶片裝置
* *CL\_DEVICE\_TYPE\_ACCELERATOR*：特定的 OpenCL 加速裝置，例如 CELL
* *CL\_DEVICE\_TYPE\_DEFAULT*：系統預設的 OpenCL 裝置
* *CL\_DEVICE\_TYPE\_ALL*：所有系統中的 OpenCL 裝置

這裡使用的是*CL\_DEVICE\_TYPE\_DEFAULT*，也就是指定使用預設的裝置。另外在這裡直接使用了之前取得的 OpenCL platform ID 中的第一個 ID(實際的程式中，可能會需要讓使用者可以指定要使用哪一個 platform)。

如果建立OpenCL context失敗會傳回 0。因此要進行檢查，並顯示錯誤訊息。如果建立成功，使用完後要記得將context釋放。透過呼叫 *clReleaseContext* 來達成。

一個OpenCL context中包括一個或多個裝置，所以接下來是要取得裝置的列表。要取得任何和OpenCL context相關的資料，可以使用 *clGetContextInfo* 函式。以下是取得裝置列表的方式：

*size\_t cb;*

*clGetContextInfo(context, CL\_CONTEXT\_DEVICES, 0, NULL, &cb);*

*std::vector<cl\_device\_id> devices(cb / sizeof(cl\_device\_id));*

*clGetContextInfo(context, CL\_CONTEXT\_DEVICES, cb, &devices[0], 0);*

CL\_CONTEXT\_DEVICES表示要取得裝置的列表。和前面取得platform ID的情形相同，*clGetContextInfo* 被呼叫了兩次：第一次是取得需要存放裝置列表所需的記憶體空間大小(也就是傳入&cb)，第二次呼叫才真正取得所有裝置的列表。

接下來會想要確定找到的OpenCL 裝置是什麼。透過OpenCL API取得裝置的名稱，並將它印出來。取得和裝置相關的資料，是使用*clGetDeviceInfo* 函式，和前面的*clGetContextInfo*函式類似。以下是取得裝置名稱的方式：

*clGetDeviceInfo(devices[0], CL\_DEVICE\_NAME, 0, NULL, &cb);*

*std::string devname;*

*devname.resize(cb);*

*clGetDeviceInfo(devices[0], CL\_DEVICE\_NAME, cb, &devname[0], 0);*

*std::cout << "Device: " << devname.c\_str() << "\n";*

到目前為止，完整的程式應該如下所示：

*// OpenCL tutorial 1*

*#include <iostream>*

*#include <string>*

*#include <vector>*

*#ifdef \_\_APPLE\_\_*

*#include <OpenCL/opencl.h>*

*#else*

*#include <CL/cl.h>*

*#endif*

*int main()*

*{*

*cl\_int err;*

*cl\_uint num;*

*err = clGetPlatformIDs(0, 0, &num);*

*if(err != CL\_SUCCESS) {*

*std::cerr << "Unable to get platforms\n";*

*return 0;*

*}*

*std::vector<cl\_platform\_id> platforms(num);*

*err = clGetPlatformIDs(num, &platforms[0], &num);*

*if(err != CL\_SUCCESS) {*

*std::cerr << "Unable to get platform ID\n";*

*return 0;*

*}*

*cl\_context\_properties prop[] = { CL\_CONTEXT\_PLATFORM, reinterpret\_cast<cl\_context\_properties>(platforms[0]), 0 };*

*cl\_context context = clCreateContextFromType(prop, CL\_DEVICE\_TYPE\_DEFAULT, NULL, NULL, NULL);*

*if(context == 0) {*

*std::cerr << "Can't create OpenCL context\n";*

*return 0;*

*}*

*size\_t cb;*

*clGetContextInfo(context, CL\_CONTEXT\_DEVICES, 0, NULL, &cb);*

*std::vector<cl\_device\_id> devices(cb / sizeof(cl\_device\_id));*

*clGetContextInfo(context, CL\_CONTEXT\_DEVICES, cb, &devices[0], 0);*

*clGetDeviceInfo(devices[0], CL\_DEVICE\_NAME, 0, NULL, &cb);*

*std::string devname;*

*devname.resize(cb);*

*clGetDeviceInfo(devices[0], CL\_DEVICE\_NAME, cb, &devname[0], 0);*

*std::cout << "Device: " << devname.c\_str() << "\n";*

*clReleaseContext(context);*

*return 0;*

*}*

執行這個程式，如果建立OpenCL context成功，會顯示出找到的OpenCL裝置的名稱，如Device: GeForce GTX 285。

**建立 Command Queue**

大部份OpenCL都要透過command queue操作。Command queue可接收對一個OpenCL裝置的各種操作，並按順序執行(OpenCL容許把command queue指定成不照順序執行，即out-of-order execution)。建立一個command queue：

*cl\_command\_queue queue = clCreateCommandQueue(context, devices[0], 0, 0);*

*if(queue == 0) {*

*std::cerr << "Can't create command queue\n";  
clReleaseContext(context);*

*return 0;*

*}*

和context一樣，程式結束前要把command queue釋放，即：*clReleaseCommandQueue(queue)*;

上面的程式中，是把裝置列表中的第一個裝置(即 devices[0])建立 command queue。如果要同時使用多個OpenCL裝置，則每個裝置都要有自己的command queue。

**產生資料**

由於這個程式的目的是要把一大堆數字進行相加，所以需要產生一些「測試資料」：

*const int DATA\_SIZE = 1048576;*

*std::vector<float> a(DATA\_SIZE), b(DATA\_SIZE), res(DATA\_SIZE);*

*for(int i = 0; i < DATA\_SIZE; i++) {*

*a[i] = std::rand();*

*b[i] = std::rand();*

*}*

**配置記憶體並複製資料**

OpenCL裝置運算時，通常要在OpenCL裝置上配置記憶體，並把資料從主記憶體中複製到裝置上。有些OpenCL裝置可直接從主記憶體存取資料，但速度通常較慢，因OpenCL裝置(如顯示卡)常有專用的高速記憶體。以下程式配置三塊記憶體：

*cl\_mem cl\_a = clCreateBuffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, sizeof(cl\_float) \* DATA\_SIZE, &a[0], NULL);*

*cl\_mem cl\_b = clCreateBuffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, sizeof(cl\_float) \* DATA\_SIZE, &b[0], NULL);*

*cl\_mem cl\_res = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, sizeof(cl\_float) \* DATA\_SIZE, NULL, NULL);*

*if(cl\_a == 0 || cl\_b == 0 || cl\_res == 0) {*

*std::cerr << "Can't create OpenCL buffer\n";*

*clReleaseMemObject(cl\_a);*

*clReleaseMemObject(cl\_b);*

*clReleaseMemObject(cl\_res);*

*clReleaseCommandQueue(queue);*

*clReleaseContext(context);*

*return 0;*

*}*

*clCreateBuffer* 函式可以用來配置記憶體。它的第二個參數可以指定記憶體的使用方式，包括：

* *CL\_MEM\_READ\_ONLY*：表示OpenCL kernel只會對這塊記憶體進行讀取的動作。
* *CL\_MEM\_WRITE\_ONLY*：表示OpenCL kernel只會對這塊記憶體進行寫入的動作。
* *CL\_MEM\_READ\_WRITE*：表示OpenCL kernel會對這塊記憶體進行讀取和寫入的動作。
* *CL\_MEM\_USE\_HOST\_PTR*：表示希望OpenCL裝置直接使用指定的主記憶體位址。要注意的是，如果OpenCL裝置無法直接存取主記憶體，它可能會將指定的主記憶體位址的資料複製到 OpenCL 裝置上。
* *CL\_MEM\_ALLOC\_HOST\_PTR*：表示希望配置的記憶體在主記憶體中，而不是在OpenCL裝置上。不能和 *CL\_MEM\_USE\_HOST\_PTR* 同時使用。
* *CL\_MEM\_COPY\_HOST\_PTR*：將指定的主記憶體位址的資料，複製到配置好的記憶體中。不能和 *CL\_MEM\_USE\_HOST\_PTR* 同時使用。

第三個參數是指定要配置的記憶體大小，以bytes為單位。在上面的程式中，指定的大小是 *sizeof(cl\_float) \* DATA\_SIZE*。第四個參數是指定主記憶體的位置。因為對*cl\_a*和*cl\_b*來說，在第二個參數中，指定了*CL\_MEM\_COPY\_HOST\_PTR*，因此要指定想要複製的資料的位址。*cl\_res* 則不需要指定。第五個參數是指定錯誤碼的傳回位址。在這裡並沒有使用到。

如果*clCreateBuffer* 因某些原因無法配置記憶體(例如OpenCL裝置上的記憶體不夠)，則會傳回0。要釋放配置的記憶體，可以使用*clReleaseMemObject* 函式。

**編譯 OpenCL kernel 程式**

執行OpenCL kernel的準備工作大致完成了。接著要把OpenCL kernel程式編釋並執行。首先把OpenCL kernel程式，存放在一個文字檔中，命名為 shader.cl：

*\_\_kernel void adder(\_\_global const float\* a, \_\_global const float\* b, \_\_global float\* result) {*

*int idx = get\_global\_id(0);*

*result[idx] = a[idx] + b[idx];*

*}*

要編譯kernel程式，要把檔案內容讀進來，再用*clCreateProgramWithSource*函式，然後再用 *clBuildProgram* 編譯。如下所示：

*cl\_program load\_program(cl\_context context, const char\* filename) {*

*std::ifstream in(filename, std::ios\_base::binary);*

*if(!in.good()) return 0;  
// get file length  
 in.seekg(0, std::ios\_base::end);  
 size\_t length = in.tellg();  
 in.seekg(0, std::ios\_base::beg);  
// read program source  
 std::vector<char> data(length + 1);  
 in.read(&data[0], length);  
 data[length] = 0;  
// create and build program   
 const char\* source = &data[0];  
 cl\_program program = clCreateProgramWithSource(context, 1, &source, 0, 0);  
 if(program == 0) return 0;  
 if(clBuildProgram(program, 0, 0, 0, 0, 0) != CL\_SUCCESS) return 0;  
 return program;*

*}*

上面的程式將檔案讀到記憶體中，再呼叫*clCreateProgramWithSource*建立一個program object。建立成功後，再呼叫 *clBuildProgram* 函式編譯程式。*clBuildProgram* 函式可以指定很多參數，不過在這裡暫時沒有使用到。

有了這個函式，在*main*函式中，直接呼叫：

*cl\_program program = load\_program(context, "shader.cl");*

*if(program == 0) {*

*std::cerr << "Can't load or build program\n";*

*clReleaseMemObject(cl\_a);*

*clReleaseMemObject(cl\_b);*

*clReleaseMemObject(cl\_res);*

*clReleaseCommandQueue(queue);*

*clReleaseContext(context);*

*return 0;*

*}*

同樣在程式結束前，要記得將 program object 釋放：*clReleaseProgram(program)*;

一個OpenCL kernel程式裡可以有很多個函式。因此要取得程式中函式的進入點：

*cl\_kernel adder = clCreateKernel(program, "adder", 0);*

*if(adder == 0) {*

*std::cerr << "Can't load kernel\n";*

*clReleaseProgram(program);*

*clReleaseMemObject(cl\_a);*

*clReleaseMemObject(cl\_b);*

*clReleaseMemObject(cl\_res);*

*clReleaseCommandQueue(queue);*

*clReleaseContext(context);*

*return 0;*

*}*

和program object一樣，取得的kernel object也需要在程式結束前釋放：*clReleaseKernel(adder)*;

**執行 OpenCL kernel**

要執行kernel程式，要先設定好函式的參數。adder 函式有三個參數要設定：

*clSetKernelArg(adder, 0, sizeof(cl\_mem), &cl\_a);*

*clSetKernelArg(adder, 1, sizeof(cl\_mem), &cl\_b);*

*clSetKernelArg(adder, 2, sizeof(cl\_mem), &cl\_res);*

設定參數是使用*clSetKernelArg*函式：第一個參數是要設定的kernel object，第二個是參數的編號(從0開始)，第三個參數是要設定的參數的大小，第四個參數則是實際上要設定的參數內部。以adder 函式來說，三個參數都是指向memory object 的指標。設定好參數後，就可以開始執行了。如下：

*size\_t work\_size = DATA\_SIZE;*

*err = clEnqueueNDRangeKernel(queue, adder, 1, 0, &work\_size, 0, 0, 0, 0);*

*clEnqueueNDRangeKernel* 會把執行一個kernel的動作加到command queue裡。第三個參數(1)是指定work item數目的維度，在這裡是一維。第五個參數是指定work item的總數目，也就是*DATA\_SIZE*。後面的參數現在暫時先不用管。如果成功加入的話，會傳回 *CL\_SUCCESS*。否則會傳回錯誤值。

在kernel被加到command queue之後，就可能開始執行(如果command queue現在沒有別的工作)。但 clEnqueueNDRangeKernel是非同步的，它不會等待OpenCL裝置執行完畢才傳回。這樣可以讓CPU在OpenCL裝置在進行運算的同時，進行其它的動作。

執行的結果在OpenCL裝置的記憶體中，要取得結果，需把內容複製到CPU能存取的主記憶體中。透過下面的程式完成：

*if(err == CL\_SUCCESS) {*

*err = clEnqueueReadBuffer(queue, cl\_res, CL\_TRUE, 0, sizeof(float) \* DATA\_SIZE, &res[0], 0, 0, 0);*

*}*

clEnqueueReadBuffer 函式會把「將記憶體資料從 OpenCL 裝置複製到主記憶體」的動作加到command queue中。第三個參數表示是否要等待複製的動作完成才傳回，CL\_TRUE 表示要等待。第五個參數是要複製的資料大小，第六個參數則是目標的位址。編譯後執行，如果順利的話，應該會印出Data is correct的訊息。以下是整個程式的全貌：

*// OpenCL tutorial 1*

*#include <iostream>*

*#include <fstream>*

*#include <string>*

*#include <vector>*

*#include <cstdlib>*

*#ifdef \_\_APPLE\_\_*

*#include <OpenCL/opencl.h>*

*#else*

*#include <CL/cl.h>*

*#endif*

*cl\_program load\_program(cl\_context context, const char\* filename)*

*{*

*std::ifstream in(filename, std::ios\_base::binary);*

*if(!in.good()) return 0;*

*// get file length*

*in.seekg(0, std::ios\_base::end);*

*size\_t length = in.tellg();*

*in.seekg(0, std::ios\_base::beg);*

*// read program source*

*std::vector<char> data(length + 1);*

*in.read(&data[0], length);*

*data[length] = 0;*

*// create and build program*

*const char\* source = &data[0];*

*cl\_program program = clCreateProgramWithSource(context, 1, &source, 0, 0);*

*if(program == 0) return 0;*

*if(clBuildProgram(program, 0, 0, 0, 0, 0) != CL\_SUCCESS) return 0;*

*return program;*

*}*

*int main()*

*{*

*cl\_int err;*

*cl\_uint num;*

*err = clGetPlatformIDs(0, 0, &num);*

*if(err != CL\_SUCCESS) {*

*std::cerr << "Unable to get platforms\n";*

*return 0;*

*}*

*std::vector<cl\_platform\_id> platforms(num);*

*err = clGetPlatformIDs(num, &platforms[0], &num);*

*if(err != CL\_SUCCESS) {*

*std::cerr << "Unable to get platform ID\n";*

*return 0;*

*}*

*cl\_context\_properties prop[] = { CL\_CONTEXT\_PLATFORM, reinterpret\_cast<cl\_context\_properties>(platforms[0]), 0 };*

*cl\_context context = clCreateContextFromType(prop, CL\_DEVICE\_TYPE\_DEFAULT, NULL, NULL, NULL);*

*if(context == 0) {*

*std::cerr << "Can't create OpenCL context\n";*

*return 0;*

*}*

*size\_t cb;*

*clGetContextInfo(context, CL\_CONTEXT\_DEVICES, 0, NULL, &cb);*

*std::vector<cl\_device\_id> devices(cb / sizeof(cl\_device\_id));*

*clGetContextInfo(context, CL\_CONTEXT\_DEVICES, cb, &devices[0], 0);*

*clGetDeviceInfo(devices[0], CL\_DEVICE\_NAME, 0, NULL, &cb);*

*std::string devname;*

*devname.resize(cb);*

*clGetDeviceInfo(devices[0], CL\_DEVICE\_NAME, cb, &devname[0], 0);*

*std::cout << "Device: " << devname.c\_str() << "\n";*

*cl\_command\_queue queue = clCreateCommandQueue(context, devices[0], 0, 0);*

*if(queue == 0) {*

*std::cerr << "Can't create command queue\n";*

*clReleaseContext(context);*

*return 0;*

*}*

*const int DATA\_SIZE = 1048576;*

*std::vector<float> a(DATA\_SIZE), b(DATA\_SIZE), res(DATA\_SIZE);*

*for(int i = 0; i < DATA\_SIZE; i++) {*

*a[i] = std::rand();*

*b[i] = std::rand();*

*}*

*cl\_mem cl\_a = clCreateBuffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, sizeof(cl\_float) \* DATA\_SIZE, &a[0], NULL);*

*cl\_mem cl\_b = clCreateBuffer(context, CL\_MEM\_READ\_ONLY | CL\_MEM\_COPY\_HOST\_PTR, sizeof(cl\_float) \* DATA\_SIZE, &b[0], NULL);*

*cl\_mem cl\_res = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, sizeof(cl\_float) \* DATA\_SIZE, NULL, NULL);*

*if(cl\_a == 0 || cl\_b == 0 || cl\_res == 0) {*

*std::cerr << "Can't create OpenCL buffer\n";*

*clReleaseMemObject(cl\_a);*

*clReleaseMemObject(cl\_b);*

*clReleaseMemObject(cl\_res);*

*clReleaseCommandQueue(queue);*

*clReleaseContext(context);*

*return 0;*

*}*

*cl\_program program = load\_program(context, "shader.cl");*

*if(program == 0) {*

*std::cerr << "Can't load or build program\n";*

*clReleaseMemObject(cl\_a);*

*clReleaseMemObject(cl\_b);*

*clReleaseMemObject(cl\_res);*

*clReleaseCommandQueue(queue);*

*clReleaseContext(context);*

*return 0;*

*}*

*cl\_kernel adder = clCreateKernel(program, "adder", 0);*

*if(adder == 0) {*

*std::cerr << "Can't load kernel\n";*

*clReleaseProgram(program);*

*clReleaseMemObject(cl\_a);*

*clReleaseMemObject(cl\_b);*

*clReleaseMemObject(cl\_res);*

*clReleaseCommandQueue(queue);*

*clReleaseContext(context);*

*return 0;*

*}*

*clSetKernelArg(adder, 0, sizeof(cl\_mem), &cl\_a);*

*clSetKernelArg(adder, 1, sizeof(cl\_mem), &cl\_b);*

*clSetKernelArg(adder, 2, sizeof(cl\_mem), &cl\_res);*

*size\_t work\_size = DATA\_SIZE;*

*err = clEnqueueNDRangeKernel(queue, adder, 1, 0, &work\_size, 0, 0, 0, 0);*

*if(err == CL\_SUCCESS) {*

*err = clEnqueueReadBuffer(queue, cl\_res, CL\_TRUE, 0, sizeof(float) \* DATA\_SIZE, &res[0], 0, 0, 0);*

*}*

*if(err == CL\_SUCCESS) {*

*bool correct = true;*

*for(int i = 0; i < DATA\_SIZE; i++) {*

*if(a[i] + b[i] != res[i]) {*

*correct = false;*

*break;*

*}*

*}*

*if(correct) {*

*std::cout << "Data is correct\n";*

*}*

*else {*

*std::cout << "Data is incorrect\n";*

*}*

*}*

*else {*

*std::cerr << "Can't run kernel or read back data\n";*

*}*

*clReleaseKernel(adder);*

*clReleaseProgram(program);*

*clReleaseMemObject(cl\_a);*

*clReleaseMemObject(cl\_b);*

*clReleaseMemObject(cl\_res);*

*clReleaseCommandQueue(queue);*

*clReleaseContext(context);*

*return 0;*