General-Purpose Programming on the GPU

M1 Info – 2015/2016

Benoit.Crespin@unilim.fr
Nicolas.Pavie@unilim.fr

12h CM (8 séances)
18h TP (12 séances)

Evaluation par un mini-projet (derniers TP)
+ écrit final (documents autorisés)
Refs

- "OpenCL Programming Guide" & "Heterogeneous Computing with OpenCL"
  - http://www.heterogeneouscompute.org/
- Hands On OpenCL: http://handsonopencl.github.io/
- Programmation des systèmes parallèles hétérogènes
- CUDA by Example: An Introduction to General-Purpose GPU Programming
Schedule

1. Introduction to Parallel Programming, GPGPU and OpenCL
2. OpenCL Device Architectures
3. Basic OpenCL Examples
4. Understanding OpenCL’s Concurrency and Execution Model
5. Dissecting OpenCL Implementations (Histogram, Dijkstra, Sparse Matrix Multiplication)
OpenCL?

- **The Open Computing Language** (OpenCL) is a heterogeneous programming framework managed by the nonprofit consortium **Khronos Group**.

- It supports a wide range of levels of parallelism and efficiently maps to homogeneous or heterogeneous, single- or multiple-device systems consisting of CPUs or GPUs (**AMD, Intel, NVIDIA, IBM, ...**)

- OpenCL’s **cross-platform**, industrywide support makes it an excellent programming model for developers to learn and use, with the confidence that it will continue to be widely available for years to come with ever-increasing scope and applicability.
OpenCL 1.0, 1.1, 1.2, ...

- OpenCL was initially developed by Apple Inc
- OpenCL 1.0 released with Mac OS X Snow Leopard (2008) – *version en salle TP*
- OpenCL 1.1 (2010) – *most examples presented in this course*
- OpenCL 1.2 (2012)
- OpenCL 2.0 (2014) – supports an Android Extension
- OpenCL 2.1 (2015)
- OpenCL vs CUDA ?
Parallel Programming

A

unsorted integers

10, 3, 7, 12, 14, 1, 2, 15

distribute to parallel hardware

parallel compares

10, 3
7, 12
14, 1
2, 15

3, 10
7, 12
1, 14
2, 15

perform merge on sorted lists

3, 7, 10, 12
1, 12, 14, 15

perform merge on sorted lists

1, 2, 3, 7, 10, 12, 14, 15

B

vector A

parallel adds

add

parallel multiples

parallel adds
Parallel Programming

Document words
- Valence
- Base
- Acid

Search string
- Acid

String compare
- String compare
- String compare
- String compare

N parallel comparison tasks
- Oxygen

String compare
- String compare
- String compare
- String compare

Char compare
- Char compare
- Char compare
- Char compare
- Char compare
Parallel Programming

- Threads and Shared Memory
- Message-Passing Communication
- Data Sharing and Synchronization
- Different Grains of Parallelism...
Many-Core Future

- Many cores running at lower frequencies are fundamentally more power-efficient.
Heterogeneous Platforms

**CPU**
Several to many cores
Very fast processors
Very large memory

**GPU**
Thousands of cores
Fast processors
Limited memory

- GPUs were originally designed for parallel display of triangles (SIMD architecture)
- More and more cores on CPUs, more and more memory on GPUs...
- As long as the tasks are well matched to the processing device, the more specialized the silicon the better the power efficiency
- Especially important for mobile devices in which conservation of battery power is critical... but also for data centers
General Programming on the GPU

- Traditionally, modules are explicitly tied to the components in the heterogeneous platform. For example, graphics software runs on the GPU. Other software runs on the CPU.

- "Traditional" GPGPU broke this model: algorithms outside of graphics were modified to fit onto the GPU.

- The CPU sets up the computation and manages I/O, but all the “interesting” computation is offloaded to the GPU.
General Programming on Heterogeneous Platforms

- Actually OpenCL *discourages* this approach: an effective program should use all devices of heterogeneous platforms!
Conceptual Foundations of OpenCL

1. Discover the **components** that make up the heterogeneous system.

2. Probe the **characteristics of these components** so that the software can adapt to the specific features of different hardware elements.

3. Create the blocks of instructions (**kernels**) that will run on the platform.

4. Set up and manipulate **memory objects** involved in the computation.

5. Execute the kernels in the right order and on the right **components** of the system.

6. Collect the final results.
A device can be a CPU, a GPU, a DSP, or any other processor provided by the hardware and supported by the OpenCL vendor.

The OpenCL devices are divided into compute units which are further divided into one or more processing elements (PEs).
How a Kernel Executes on an OpenCL Device

1. A kernel is defined on the host.

2. The host program issues a command that submits the kernel for execution on an OpenCL device.

3. When this command is issued by the host :
   - the OpenCL runtime system creates an integer index space.
   - An instance of the kernel executes for each point in this index space, called a work-item.
   - Its coordinates in the index space are the global ID for the work-item.

4. Work-items are organized into work-groups which exactly span the global index space. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID.

5. OpenCL only assures that the work-items within a work-group execute concurrently on the processing elements of a single compute unit (and share processor resources on the device)
NDRange

• The index space spans an N-dimensioned range of values and thus is called an NDRange (N can be 1, 2 or 3)

• Inside an OpenCL program, an NDRange is defined by an integer array of length N specifying the size of the index space in each dimension.
Work-items and work-groups

Global ID \((gx, gy) = (6, 5)\)

Work-group ID \((wx, wy) = (1, 1)\)

Local ID \((lx, ly) = (2, 1)\)
Context

- **Devices**: the collection of OpenCL devices to be used by the host
- **Kernels**: the OpenCL functions that run on OpenCL devices
- **Program objects**: the program source code and executables that implement the kernels
- **Memory objects**: a set of objects in memory that are visible to OpenCL devices and contain values that can be operated on by instances of a kernel
Command-Queues

- The interaction between the host and the OpenCL devices occurs through commands posted by the host to the **command-queue**.

- These commands wait in the command-queue until they execute on the OpenCL device.

- A command-queue is created by the host and attached to a single OpenCL device after the context has been defined.

  - **Kernel execution commands** execute a kernel on the processing elements of an OpenCL device.
  
  - **Memory commands** transfer data between the host and different memory objects, move data between memory objects, or map and unmap memory objects from the host address space.

  - **Synchronization commands** put constraints on the order in which commands execute.
Memory Model

[Diagram showing a memory model with compute units, local memories, and a global/constant memory data cache.]
Memory Model

- **Host memory**: visible only to the host.
- **Global memory**: permits read/write access to all work-items in all work-groups. Reads and writes to global memory may be cached depending on the capabilities of the device.
- **Constant memory**: remains constant during the execution of a kernel (read-only access).
- **Local memory**: local to a work-group. It can be used to allocate variables that are shared by all work-items in that work-group. It may be implemented as dedicated regions of memory on the OpenCL device.
- **Private memory**: This region of memory is private to a work-item. Variables defined in one work-item’s private memory are not visible to other work-items.
Arrays of structures (AoS) vs Structures of arrays (SoA)

- For SIMD architectures, kernel execution is faster with contiguous memory access (or coalesced memory)
- In other words, reading/writing values in different parts of the memory should be avoided

```c
struct {
    uint8_t r, g, b;
} AoS[N];
```

```c
struct {
    uint8_t r[N];
    uint8_t g[N];
    uint8_t b[N];
} SoA;
```

- Which one is better if we want to:
  - Display all N pixel values?
  - Scale all the N values in the Blue channel by some factor?
Summary

[Diagram showing the relationship between CPU, GPU, Context, Programs, Kernels, Memory objects, and Command-queues, along with the steps Compile code, Create data and arguments, Send to execution.]

```c
kernel void dp_mul(global const float *a, 
global const float *b, 
global float *c) 
{
    int id = get_global_id(0); 
c[id] = a[id] * b[id];
}
```

[Code snippet for dp_mul kernel function]
Remember this?

1. Discover the **components** that make up the heterogeneous system.

2. Probe the **characteristics of these components** so that the software can adapt to the specific features of different hardware elements.

3. ...
#include <CL/cl.h>
#include <...>

#define CL_CHECK(_expr)  
   do { cl_int _err = _expr;  
      if (_err == CL_SUCCESS) break; 
      fprintf(stderr, "OpenCL Error: '%s' returned %d!
", #_expr, (int)_err); 
      abort(); 
   } while (0)

int main(int argc, char **argv) {
   cl_platform_id platforms[100];
   cl_uint platforms_n = 0;
   CL_CHECK(clGetPlatformIDs(100, platforms, &platforms_n));
   printf("=== %d OpenCL platform(s) found: ===\n", platforms_n);
   for (int i=0; i<platforms_n; i++) {
      char buffer[10240];
      printf(" -- %d --\n", i);
      CL_CHECK(clGetPlatformInfo(platforms[i], CL_PLATFORM_PROFILE, 10240, buffer, NULL));
      printf("  PROFILE = %s
", buffer);
      CL_CHECK(clGetPlatformInfo(platforms[i], CL_PLATFORM_VERSION, 10240, buffer, NULL));
      printf("  VERSION = %s\n", buffer);
      CL_CHECK(clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, 10240, buffer, NULL));
      printf("  NAME = %s\n", buffer);
      CL_CHECK(clGetPlatformInfo(platforms[i], CL_PLATFORM_VENDOR, 10240, buffer, NULL));
      printf("  VENDOR = %s\n", buffer);
      CL_CHECK(clGetPlatformInfo(platforms[i], CL_PLATFORM_EXTENSIONS, 10240, buffer, NULL));
      printf("  EXTENSIONS = %s\n", buffer);
   }
   if (platforms_n == 0) return 1;
   cl_device_id devices[100];
   cl_uint devices_n = 0;
   CL_CHECK(clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_DEFAULT, 100, devices, &devices_n));
   printf("  PROFILE = %s\n", buffer);
   CL_CHECK(clGetDeviceInfo(devices[0], CL_DEVICE_PROFILE, 10240, buffer, NULL));
   printf("  PROFILE = %s\n", buffer);
   CL_CHECK(clGetDeviceInfo(devices[0], CL_DEVICE_VERSION, 10240, buffer, NULL));
   printf("  VERSION = %s\n", buffer);
   CL_CHECK(clGetDeviceInfo(devices[0], CL_DEVICE_NAME, 10240, buffer, NULL));
   printf("  NAME = %s\n", buffer);
   CL_CHECK(clGetDeviceInfo(devices[0], CL_DEVICE_VENDOR, 10240, buffer, NULL));
   printf("  VENDOR = %s\n", buffer);
   CL_CHECK(clGetDeviceInfo(devices[0], CL_DEVICE_EXTENSIONS, 10240, buffer, NULL));
   printf("  EXTENSIONS = %s\n", buffer);
}
printf("=== %d OpenCL device(s) found on platform:\n", devices_n);

for (int i=0; i<devices_n; i++) {
  char buffer[10240];
  cl_uint buf_uint;
  cl_ulong buf_ulong;
  printf(" -- %d --\n", i);
  CL_CHECK(clGetDeviceInfo(devices[i], CL_DEVICE_NAME, sizeof(buffer), buffer, NULL));
  printf(" DEVICE_NAME = %s\n", buffer);
  CL_CHECK(clGetDeviceInfo(devices[i], CL_DEVICE_VENDOR, sizeof(buffer), buffer, NULL));
  printf(" DEVICE_VENDOR = %s\n", buffer);
  CL_CHECK(clGetDeviceInfo(devices[i], CL_DEVICE_VERSION, sizeof(buffer), buffer, NULL));
  printf(" DEVICE_VERSION = %s\n", buffer);
  CL_CHECK(clGetDeviceInfo(devices[i], CL_DRIVER_VERSION, sizeof(buffer), buffer, NULL));
  printf(" DRIVER_VERSION = %s\n", buffer);
  CL_CHECK(clGetDeviceInfo(devices[i], CL_DEVICE_MAX_COMPUTE_UNITS, sizeof(buf_uint), &buf_uint, NULL));
  printf(" DEVICE_MAX_COMPUTE_UNITS = %u\n", (unsigned int)buf_uint);
  CL_CHECK(clGetDeviceInfo(devices[i], CL_DEVICE_MAX_CLOCK_FREQUENCY, sizeof(buf_uint), &buf_uint, NULL));
  printf(" DEVICE_MAX_CLOCK_FREQUENCY = %u\n", (unsigned int)buf_uint);
  CL_CHECK(clGetDeviceInfo(devices[i], CL_DEVICE_GLOBAL_MEM_SIZE, sizeof(buf_ulong), &buf_ulong, NULL));
  printf(" DEVICE_GLOBAL_MEM_SIZE = %llu\n", (unsigned long long)buf_ulong);
}

if (devices_n == 0) return 1;
return 0;

} // end main
Compiling and running (finally !)

```
$> g++ platformAndDevices.c -IOpenCL
$> ./a.out

// Execution on an Nvidia GPU
=== 1 OpenCL platform(s) found: ===
  -- 0 –
  PROFILE = FULL_PROFILE
  VERSION = OpenCL 1.1 CUDA 4.2.1
  NAME = NVIDIA CUDA
  VENDOR = NVIDIA Corporation
  EXTENSIONS = cl_khr_icd cl_khr_gl_sharing
  cl_nv_compiler_options ...

=== 1 OpenCL device(s) found on platform:
  -- 0 –
  DEVICE_NAME = GeForce 8600 GT
  DEVICE_VENDOR = NVIDIA Corporation
  DEVICE_VERSION = OpenCL 1.0 CUDA
  DRIVER_VERSION = 319.17
  DEVICE_MAX_COMPUTE_UNITS = 4
  DEVICE_MAX_CLOCK_FREQUENCY = 1188
  DEVICE_GLOBAL_MEM_SIZE = 536150016

// Execution on an Intel CPU
=== 1 OpenCL platform(s) found: ===
  -- 0 –
  PROFILE = FULL_PROFILE
  VERSION = OpenCL 1.2 LINUX
  NAME = Intel(R) OpenCL
  VENDOR = Intel(R) Corporation
  EXTENSIONS = cl_khr_fp64 cl_khr_icd
  cl_khr_global_int32_base_atomics ...

=== 1 OpenCL device(s) found on platform:
  -- 0 –
  DEVICE_NAME = Intel(R) Core(TM) i3-3120M CPU @ 2.50GHz
  DEVICE_VENDOR = Intel(R) Corporation
  DEVICE_VERSION = OpenCL 1.2 (Build 67279)
  DRIVER_VERSION = 1.2
  DEVICE_MAX_COMPUTE_UNITS = 4
  DEVICE_MAX_CLOCK_FREQUENCY = 2500
  DEVICE_GLOBAL_MEM_SIZE = 8247922688

With AMD : g++ platformAndDevices.c -I/opt/AMDAPP/include/ -L/opt/AMDAPP/lib/x86_64 -IOpenCL
```
clGetPlatformIDs

cl_int clGetPlatformIDs (cl_uint num_entries,
                        cl_platform_id * platforms,
                        cl_uint * num_platforms) ;

• This command obtains the list of available platforms

• In the case that the argument platforms is NULL, then it returns the number of available platforms.

• Another example :

```c
errNum = clGetPlatformIDs(0, NULL, &numPlatforms);
platformIds = (cl_platform_id *)alloca(sizeof(cl_platform_id) * numPlatforms);
errNum = clGetPlatformIDs(numPlatforms, platformIds, NULL);
```
clGetPlatformInfo

cl_int clGetPlatformInfo (cl_platform_id platform,
    cl_platform_info param_name,
    size_t param_value_size,
    void * param_value,
    size_t * param_value_size_ret)

• This command returns specific information about the OpenCL platform: profile, version, name, ...

• Another example:

    err = clGetPlatformInfo(id, CL_PLATFORM_NAME, 0, NULL, &size);
    char * name = (char *)alloca(sizeof(char) * size);
    err = clGetPlatformInfo(id, CL_PLATFORM_NAME, size, name, NULL);
clGetDeviceIDs

cl_int clGetDeviceIDs (cl_platform_id platform,
                     cl_device_type device_type,
                     cl_uint num_entries,
                     cl_device_id *devices,
                     cl_uint *num_devices)

- This command obtains the list of available OpenCL devices associated with platform.

- **device_type** :
  - CL_DEVICE_TYPE_CPU : OpenCL device that is the host processor.
  - CL_DEVICE_TYPE_GPU : OpenCL device that is a GPU.
  - CL_DEVICE_TYPE_ACCELERATOR : OpenCL accelerator (e.g., IBM Cell Broadband).
  - CL_DEVICE_TYPE_DEFAULT : Default device.
  - CL_DEVICE_TYPE_ALL : All OpenCL devices associated with the corresponding platform.
clGetDeviceInfo

`cl_int clGetDeviceInfo (cl_device_id device, 
    cl_device_info param_name, 
    size_t param_value_size, 
    void * param_value, 
    size_t * param_value_size_ret)`

- This command returns specific information about the OpenCL device.

- param_name:
  - CL_DEVICE_TYPE, CL_DEVICE_VENDOR_ID, …
Programming steps to writing a complete OpenCL application

Diagram:
- Platform layer:
  - Query platform
  - Query devices
  - Command queue

- Runtime layer:
  - Create buffers
  - Compile program
  - Compile kernel
  - Set arguments
  - Execute kernel

- Compiler
Parallel vector addition in 13 steps

```c
#include <CL/cl.h>
#include <...>

const char* programSource[] = {
    "__kernel
    " void vecadd(__global int *A,
    "            __global int *B,
    "            __global int *C)                        
    
    // Get the work-item’s unique ID
    int idx = get_global_id(0);
    
    // Add the corresponding locations of 'A' and 'B', and store the result in 'C'.
};

int main() {
    int *A = NULL; // Input array
    int *B = NULL; // Input array
    int *C = NULL; // Output array

    const int elements = 2048;
    size_t datasize = sizeof(int)*elements;

    A = (int*)malloc(datasize);
    B = (int*)malloc(datasize);
    C = (int*)malloc(datasize);

    // Initialize the input data
    for(int i = 0; i < elements; i++) {
        A[i] = i; B[i] = i;
    }

    // Use this to check the output of each API call
    cl_int status;
```
STEP 1: Discover and initialize the platforms

STEP 2: Discover and initialize the devices

----------STEP 1----------

cl_uint numPlatforms = 0;
cl_platform_id *platforms = NULL;

// Use clGetPlatformIDs() to retrieve the number of platforms
status = clGetPlatformIDs(0, NULL, &numPlatforms);

// Allocate enough space for each platform
platforms = (cl_platform_id*)malloc(numPlatforms * sizeof(cl_platform_id));

// Fill in platforms with clGetPlatformIDs()
status = clGetPlatformIDs(numPlatforms, platforms, NULL);

----------STEP 2----------

cl_uint numDevices = 0;
cl_device_id *devices = NULL;

// Use clGetDeviceIDs() to retrieve the number of devices present
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0, NULL, &numDevices);

// Allocate enough space for each device
devices = (cl_device_id*)malloc(numDevices * sizeof(cl_device_id));

// Fill in devices with clGetDeviceIDs()
status = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, numDevices, devices, NULL);
STEP 3: Create a context
STEP 4: Create a command queue
STEP 5: Create device buffers

------------------------- STEP 3 -------------------------

cl_context context = NULL;

// Create a context using clCreateContext() and
// associate it with the devices
context = clCreateContext(NULL, numDevices,
                          devices, NULL, NULL,
                          &status);

------------------------- STEP 4 -------------------------

cl_command_queue cmdQueue;

// Create a command queue using
// clCreateCommandQueue(), and associate it with
// the device you want to execute on
cmdQueue = clCreateCommandQueue(context,
                                  devices[0],
                                  0, &status);

------------------------- STEP 5 -------------------------

cl_mem bufferA;  // Input array on the device
cl_mem bufferB;  // Input array on the device
cl_mem bufferC;  // Output array on the device

// Use clCreateBuffer() to create buffer objects
// that will contain the data from the host arrays
bufferA = clCreateBuffer(context,
                          CL_MEM_READ_ONLY,
                          datasize, NULL, &status);
bufferB = clCreateBuffer(context,
                          CL_MEM_READ_ONLY,
                          datasize, NULL, &status);
bufferC = clCreateBuffer(context,
                          CL_MEM_WRITE_ONLY,
                          datasize, NULL, &status);
STEP 6: Write host data to device buffers
STEP 7: Create and compile the program
STEP 8: Create the kernel

STEP 6: Write host data to device buffers

Use clEnqueueWriteBuffer() to write input array A to the device buffer bufferA.
```
status = clEnqueueWriteBuffer(cmdQueue, bufferA, CL_FALSE, 0, datasize, A, 0, NULL, NULL);
```

Use clEnqueueWriteBuffer() to write input array B to the device buffer bufferB.
```
status = clEnqueueWriteBuffer(cmdQueue, bufferB, CL_FALSE, 0, datasize, B, 0, NULL, NULL);
```

STEP 7: Create and compile the program

Create a program using clCreateProgramWithSource().
```
cl_program program = clCreateProgramWithSource(context, sizeof(programSource)/sizeof(*programSource), programSource, NULL, &status);
```

Build (compile) the program for the devices with clBuildProgram().
```
status = clBuildProgram(program, numDevices, devices, NULL, NULL, NULL);
```

STEP 8: Create the kernel

Use clCreateKernel() to create a kernel from the vector addition function (named "vecadd").
```
kkernel = clCreateKernel(program, "vecadd", &status);
```
STEP 9: Set the kernel arguments
STEP 10: Configure the work-item structure
STEP 11: Enqueue the kernel for execution

STEP 9

// Associate the input and output buffers with the kernel using clSetKernelArg()
status  = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                          &bufferA);
status |= clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          &bufferB);
status |= clSetKernelArg(kernel, 2, sizeof(cl_mem),
                          &bufferC);

STEP 10

// Define an index space (global work size) of work items for execution. A workgroup size (local work size) is not required, but can be used.
size_t globalWorkSize[1];

// There are 'elements' work-items
globalWorkSize[0] = elements;

STEP 11

// Execute the kernel by using clEnqueueNDRangeKernel().
// 'globalWorkSize' is the 1D dimension of the work-items
status = clEnqueueNDRangeKernel(cmdQueue,
                                kernel, 1, NULL, globalWorkSize, NULL, 0, NULL, NULL);
STEP 12: Read the output buffer back to the host

STEP 13: Release OpenCL resources

---

### STEP 12

// Use clEnqueueReadBuffer() to read the OpenCL output buffer (bufferC) to the host output array (C)
clEnqueueReadBuffer(cmdQueue, bufferC, CL_TRUE, 0, datasize, C, 0, NULL, NULL);

// Verify the output
bool result = true;
for(int i = 0; i < elements; i++) {
    if(C[i] != i+i) {
        result = false;
        break;
    }
}

if(result) printf("Output is correct\n");
else printf("Output is incorrect\n");

---

### STEP 13

// Free OpenCL resources
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmdQueue);
clReleaseMemObject(bufferA);
clReleaseMemObject(bufferB);
clReleaseMemObject(bufferC);
clReleaseContext(context);

// Free host resources
free(A);
free(B);
free(C);
free(platforms);
free(devices);
Memory mapping

get\_global\_id(0) = 7

\[
\begin{align*}
7 & 9 & 13 & 1 & 31 & 3 & 0 & 76 & 33 & 5 & 23 & 11 & 51 & 77 & 60 & 8 \\
34 & 2 & 0 & 13 & 18 & 22 & 6 & 22 & 47 & 17 & 56 & 41 & 29 & 11 & 9 & 82 \\
\end{align*}
\]

\[+\]

\[
\begin{align*}
41 & 11 & 13 & 14 & 49 & 25 & 6 & 98 & 80 & 22 & 79 & 52 & 80 & 88 & 69 & 90 \\
\end{align*}
\]

- Each executing work-item needs to know which individual elements from arrays \(a\) and \(b\) need to be summed.
- This must be a unique value for each work-item and should be derived from the N-D domain specified when queuing the kernel for execution.
- The \texttt{get\_global\_id(0)} returns the one-dimensional global ID for each work-item.
How to check for errors in your kernel: `clGetProgramBuildInfo`

- Should take place after `clBuildProgram` (step 7)

- Get the length of the log string:
  ```c
  size_t len;
  clGetProgramBuildInfo(program, devices[0],
                         CL_PROGRAM_BUILD_LOG, 0, NULL, &len);
  ```

- Get the log itself:
  ```c
  char *buffer = (char *)malloc(len);
  clGetProgramBuildInfo(program, devices[0],
                         CL_PROGRAM_BUILD_LOG, len, buffer, NULL);
  ```

- Typical results:
  - expected ';' after expression
  - use of undeclared identifier 'localId'
  - error: expected '}'
  - ...
  - ...
Exercices

• Testez l'environnement OpenCL en salle TP et sur votre propre machine
• Testez les performances du programme qui additionne deux vecteurs :
  – En comparant avec une implémentation séquentielle « classique »
  – En faisant varier le volume des données (« scalabilité »)
Simple Matrix Multiplication
(sequential code)

// Iterate over the rows of Matrix A
for(int i = 0; i < heightA; i++) {
    // Iterate over the columns of
    // Matrix B
    for(int j = 0; j < widthB; j++) {
        C[i][j] = 0;
        // Multiply and accumulate the
        // values in the current row
        // of A and column of B
        for(int k = 0; k < widthA; k++)
            C[i][j] += A[i][k] * B[k][j];
    }
}
Parallel Matrix Multiplication

- See previous example for context, device, …

- Buffers are now matrices (basically longer buffers):
  - `cl_mem bufferA = clCreateBuffer(ctx, CL_MEM_READ_ONLY, wA*hA*sizeof(float), NULL, &ciErrNum);
  - `ciErrNum = clEnqueueWriteBuffer(myqueue, bufferA, CL_TRUE, 0, wA*hA*sizeof(float), (void *)A, 0, NULL, NULL);`
How can we retrieve row and column indices from `get_global_id(0)`?
Parallel Matrix Multiplication Kernel

// widthA = heightB for valid matrix multiplication
__kernel void simpleMultiply(__global float* outputC,
    int widthA, int heightA,
    int widthB, int heightB,
    __global float* inputA,
    __global float* inputB) {

    //Get global position in Y direction
    int row = get_global_id(1);
    //Get global position in X direction
    int col = get_global_id(0);
    float sum = 0.0f;
    //Calculate result of one element of Matrix C
    for (int i = 0; i < widthA; i++)
        sum += inputA[row*widthA+i] * inputB[i*widthB+col];
    outputC[row*widthB+col] = sum;
}
Remember this?

Global ID \((gx, gy) = (6, 5)\)

Work-group ID \((wx, wy) = (1, 1)\)

Local ID \((lx, ly) = (2, 1)\)
Memory mapping

```c
size_t globalWorkSize[2];

globalWorkSize[0] = WB;
globalWorkSize[1] = HA;

errcode = clEnqueueNDRangeKernel
    (queue, kernel, 2, NULL, globalWorkSize, NULL, 0, NULL, NULL);
```
clEnqueueNDRangeKernel

- cl_int clEnqueueNDRangeKernel (command_queue, kernel, work_dim, global_work_offset, global_work_size, local_work_size, num_events_in_wait_list, event_wait_list, event)

- **work_dim**: the number of dimensions used to specify the global work-items and work-items in the work-group

- **global_work_offset**: must currently be a NULL value

- **global_work_size**: points to an array of work_dim unsigned values that describe the number of global work-items in work_dim dimensions that will execute the kernel function.
  - The total number of global work-items is computed as global_work_size[0] *...* global_work_size[work_dim – 1]

- **local_work_size**: points to an array of work_dim unsigned values that describe the number of work-items that make up a work-group (also referred to as the size of the work-group) that will execute the kernel specified by kernel.
  - The total number of work-items in a work-group is computed as local_work_size[0] *...* local_work_size[work_dim - 1].
  - If local_work_size is specified, the values specified in global_work_size[0],..., global_work_size[work_dim - 1] must be evenly divisible by the corresponding values specified in local_work_size[0],..., local_work_size[work_dim - 1].
  - **local_work_size** can also be a NULL value in which case the OpenCL implementation will determine how to break the global work-items into appropriate work-group instances.
Image Rotation Example

\[ x_2 = \cos(\theta) \times (x_1) + \sin(\theta) \times (y_1) \]
\[ y_2 = -\sin(\theta) \times (x_1) + \cos(\theta) \times (y_1) \]
__kernel void img_rotate(__global float* dest_data, __global float* src_data,
int W, int H,                               //Image Dimensions
float sinTheta, float cosTheta )             //Rotation Parameters
{
    //Work-item gets its index within index space
    const int ix = get_global_id(0);
    const int iy = get_global_id(1);
    //Calculate location of data to move into (ix,iy)
    float x0 = W/2.0f;
    float y0 = H/2.0f;
    float xOff = ix - x0;
    float yOff = iy - y0;
    int xpos = (int)(xOff*cosTheta + yOff*sinTheta + x0 );
    int ypos = (int)(yOff*cosTheta - xOff*sinTheta + y0 );
    //Bound Checking
    if(((int)xpos>=0) && ((int)xpos<W) && ((int)ypos>=0) && ((int)ypos<H)) {
        // Read (ix,iy) src_data and store at (xpos,ypos) in dest_data
        dest_data[iy*W+ix] = src_data[ypos*W+xpos];
    }
}
Image Rotation Example

See the full example to discover how to:

- Load a BMP image file into a buffer
  (http://fr.wikipedia.org/wiki/Windows_bitmap)
- Store OpenCL kernels in separated files and load them at runtime
- Code with OpenCL in C++!

```c++
// Discover platforms
cl::vector<cl::Platform> platforms;
cl::Platform::get(&platforms);
// Create a context with the first platform
cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,
   (cl_context_properties)(platforms[0])(), 0};
// Create a context using this platform for a GPU type device
cl::Context context(CL_DEVICE_TYPE_ALL, cps);
...
Exercices

- Codez avec une implémentation séquentielle classique puis en OpenCL :
  - la multiplication de deux matrices
  - la rotation d'image (en utilisant l'API « C »)
- Testez les performances en faisant varier le volume des données
Function Qualifiers

- __kernel or kernel

The following rules apply to kernel functions:

- The return type must be void. If the return type is not void, it will result in a compilation error.
- The function can be executed on a device by enqueuing a command to execute the kernel from the host.
- The function behaves as a regular function if it is called from a kernel function. The only restriction is that a kernel function with variables declared inside the function with the local qualifier cannot be called from another kernel function.
kernel void my_func_a(global float *src,  
global float *dst)  
{  
local float l_var[32];  
...  
}

kernel void my_func_b(global float *src,  
global float *dst)  
{  
// implementation-defined behavior  
my_func_a(src, dst);  
}

kernel void my_func_a(global float *src,  
global float *dst,  
local float *l_var)  
{  
...  
}

kernel void my_func_b(global float * src,  
global float *dst,  
local float *l_var)  
{  
my_func_a(src, dst, l_var);  
}
Address Space Qualifiers

- The type qualifier can be `global` (or `__global`), `local` (or `__local`), `constant` (or `__constant`), or `private` (or `__private`)

- If the type of an object is qualified by an address space name, the object is allocated in the specified address space (if not specified, then the object is allocated in the private address space)

- Pointers to the `global` address space are allowed as arguments to functions (including kernel functions) and variables declared inside functions. Variables declared inside a function cannot be allocated in the `global` address space.
Legal and illegal

- kernel void my_func(int *p)  // **illegal** because generic address space name for p is private.
- kernel void my_func(private int *p)  // **illegal** because memory pointed to by p is allocated in private.
- void my_func(int *p)  // generic address space name for p is private: **legal** as my_func is not a kernel function
- void my_func(private int *p)  // **legal** as my_func is not a kernel function
- void my_func(global float4 *vA, global float4 *vB) {
  global float4 *p;  // **legal**
  global float4 a;  // **illegal**
}
Constant Address Space

- Used to describe variables allocated in global memory that are accessed inside a kernel(s) as read-only variables

- // legal - program scope variables can be allocated only in the constant address space
  constant float wtsA[ ] = { 0, 1, 2, . . . }; // program scope

- // illegal - program scope variables can be allocated only in the constant address space
  global float wtsB[ ] = { 0, 1, 2, . . . };

- kernel void my_func(constant float4 *vA, constant float4 *vB) {
  constant float4 *p = vA; // legal
  constant float a; // illegal – not initialized
  constant float b = 2.0f; // legal – initialized with a compile-time constant
  p[0] = (float4)(1.0f); // illegal – p cannot be modified
  ...
Local Address Space

- Used to describe variables that need to be allocated in local memory and are shared by all work-items of a work-group but not across work-groups executing a kernel.

- A good analogy for local memory is a user-managed cache. It is preferable to read the required data from global memory (which is an order of magnitude slower) once into local memory and then have the work-items read multiple times from local memory.

```c
kernel void my_func(global float4 *vA, local float4 *l) {
    local float4 *p; // legal
    local float4 a; // legal
    a = 1;
    local float4 b = (float4)(0); // illegal – b cannot be initialized
    if (...) {
        local float c; // illegal – must be allocated at kernel function scope
        ...
    }
}
```
Image Convolution Example

Original

Three pixel radius

Ten pixel radius

Source image

Filtered image

Filter

\[
\begin{align*}
&(-1 \times 1) \\
&(0 \times 4) \\
&(1 \times 6) \\
&(-2 \times 5) \\
&(0 \times 3) \\
&(2 \times 8) \\
&(-1 \times 6) \\
&(0 \times 7) \\
&+ (1 \times 2) \\
&= 7
\end{align*}
\]
Sequential convolution

// Iterate over the rows of the source image
for(int i = halfFilterWidth; i < rows - halfFilterWidth; i++) {
    // Iterate over the columns of the source image
    for(int j = halfFilterWidth; j < cols - halfFilterWidth; j++) {
        sum = 0; // Reset sum for new source pixel
        // Apply the filter to the neighborhood
        for(int k = - halfFilterWidth; k <= halfFilterWidth; k++) {
            for(int l = - halfFilterWidth; l <= halfFilterWidth; l++) {
                sum += Image[i+k][j+l] * Filter[k+ halfFilterWidth][l+ halfFilterWidth];
            }
        }
        outputImage[i][j] = sum;
    }
}
A specific way to handle images in OpenCL 1.0

// The image format describes how the data will be stored in memory
cl_image_format format;
format.image_channel_order = CL_R;  // single channel (for greyscale)
format.image_channel_data_type = CL_FLOAT;

// Create space for the source image on the device
cl_mem d_inputImage = clCreateImage2D(context, 0, &format, imageWidth,
imageHeight, 0, NULL, &status);

// Copy the source image to the device
size_t origin[3] = {0, 0, 0}; // Offset within the image to copy from
size_t region[3] = {imageWidth, imageHeight, 1}; // Elements to per dimension
status = clEnqueueWriteImage(queue, d_inputImage, CL_FALSE,
origin, region, 0, 0, inputImage, 0, NULL, NULL);

// Create the image sampler
cl_sampler sampler = clCreateSampler(context, CL_FALSE,
CL_ADDRESS_CLAMP_TO_EDGE, CL_FILTER_NEAREST, &status);
Convolution kernel

```c
__kernel void convolution(
    __read_only image2d_t sourceImage,
    __write_only image2d_t outputImage,
    int rows, int cols,
    __constant float* filter,
    int filterWidth, sampler_t sampler) {

    // Store each work-item's unique row and column
    int column = get_global_id(0);
    int row    = get_global_id(1);

    // Half the width of the filter is needed for indexing memory later
    int halfWidth = (int)(filterWidth/2);

    // All accesses to images return data as four-element vector (i.e., float4), although only the 'x' component will contain meaningful data in this code
    float4 sum = {0.0f, 0.0f, 0.0f, 0.0f};

    // Iterator for the filter
    int filterIdx = 0;

    // Each work-item iterates around its local area based on the size of the filter
    // Coordinates for accessing the image
    int2 coords;

    // Iterate the filter rows
    for(int i = -halfWidth; i <= halfWidth; i++) {
        coords.y = row + i;
        // Iterate over the filter columns
        for(int j = -halfWidth; j <= halfWidth; j++) {
            coords.x = column + j;

            float4 pixel;

            // Read a pixel from the image. A single channel image stores the pixel in the 'x' coordinate of the returned vector
            pixel = read_imagef(sourceImage, sampler, coords);

            sum.x += pixel.x * filter[filterIdx++];
        }
    }

    // Copy the data to the output image if the work-item is in bounds
    if(row < rows && column < cols) {
        coords.x = column;
        coords.y = row;
        write_imagef(outputImage, coords, sum);
    }
}
```
Results with a 3x3 filter
Synchronization with OpenCL
Example of synchronization inside a kernel (local)

// Host code

\[
\text{cl\_mem input = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, 10*sizeof(float), 0, 0);} \\
\text{cl\_mem intermediate = clCreateBuffer(context, CL\_MEM\_READ\_ONLY, 10*sizeof(float), 0, 0);} \\
\text{cl\_mem output = clCreateBuffer(context, CL\_MEM\_WRITE\_ONLY, 10*sizeof(float), 0, 0);} \\
\]

\[
\text{clEnqueueWriteBuffer(queue, input, CL\_TRUE, 0, 10*sizeof(int), (void \*)hostInput, 0, NULL, NULL);} \\
\text{clSetKernelArg(kernel, 0, sizeof(cl\_mem), (void \*)\&input);} \\
\text{clSetKernelArg(kernel, 1, sizeof(cl\_mem), (void \*)\&intermediate);} \\
\text{size\_t localws[1] = \{2\};} \\
\text{size\_t globalws[1] = \{10\};} \\
\text{clEnqueueNDRangeKernel(queue, kernel, 1, NULL, globalws, localws, 0, NULL, NULL);} \\
\]

\[
\text{clSetKernelArg(kernel, 0, sizeof(cl\_mem), (void \*)\&intermediate);} \\
\text{clSetKernelArg(kernel, 1, sizeof(cl\_mem), (void \*)\&output);} \\
\text{clEnqueueNDRangeKernel(queue, kernel, 1, NULL, globalws, localws, 0, NULL, NULL);} \\
\text{clEnqueueReadBuffer(queue, output, CL\_TRUE, 0, 10*sizeof(float), (void \*)\&hostOutput, 0, NULL, NULL);} \\
\]

// Kernel

\[
\text{\_kernel void simpleKernel(__global float \*a, \_global float \*b, \_local float \*ldata)} \\
\{
\text{ldata[get\_local\_id(0)] = a[get\_global\_id(0)];} \\
\text{barrier(CLK\_LOCAL\_MEM\_FENCE);} \\
\text{// Now we are sure that all data is available in ldata}
\text{unsigned int otherAddress = (get\_local\_id(0) + 1) \% get\_local\_size(0);} \\
\text{b[get\_global\_id(0)] = ldata[get\_local\_id(0)] + ldata[otherAddress];} \\
\}
\]
Example of global synchronization

// Perform setup of platform, context and create buffers

...  
// Create queue leaving parameters as default so queue is in-order
queue = clCreateCommandQueue(context, devices[0], 0, 0);  
...
clEnqueueWriteBuffer(queue, bufferA, CL_TRUE, 0, 10 * sizeof(int), a, 0, NULL, NULL);
clEnqueueWriteBuffer(queue, bufferB, CL_TRUE, 0, 10 * sizeof(int), B, 0, NULL, NULL);

// Set kernel arguments

...
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, globalws, localws, 0, NULL, NULL);

// Perform blocking read-back to synchronize
clEnqueueReadBuffer(queue, bufferOut, CL_TRUE, 0, 10 * sizeof(int), out, 0, 0, 0);

// This last operation is equivalent to:
clEnqueueReadBuffer(queue, bufferOut, CL_FALSE, 0, 10 * sizeof(int), out, 0, 0, 0);
cFinish(queue);
Simple timing with OpenCL

- **Events** provide a gateway to a command’s history: they contain information detailing when the command was placed in the queue, when it was submitted to the device, and when it started and ended execution.

- `cl_int clGetEventProfilingInfo ( cl_event event, cl_profiling_info param_name, size_t param_value_size, void *param_value, size_t *param_value_size_ret)`

- Profiling is enabled when creating a command queue by setting the `CL_QUEUE_PROFILING_ENABLE` flag.
Using different local sizes

- If no local dimension is given to clEnqueueNDRangeKernel, then **OpenCL decides for the programmer**
- Otherwise it's safer to choose a **power of 2**
- Now we can test how performances are affected when we use this feature for vector addition or simple matrix multiplication
Optimizing matrix multiplication...

- With the previous implementation and order-1000 matrices, one work-item per matrix element results in a million work-items (approx. 511 MFLOPS)
- In the next version of the program, each work-item will compute a row of the matrix
- The NDRange is changed from a 2D range set to match the dimensions of the C matrix to a 1D range set to the number of rows in the C matrix.
- My device has four compute units. Hence for a order-1000 matrix we can set the work-group size to 250 and create four work-groups to cover the full size of the problem.
// Optimized matrix multiplication kernel
// Version 1

__kernel mmul(const int Mdim,
               const int Ndim,
               const int Pdim,
               __global float* A,
               __global float* B,
               __global float* C) {

  int k,j;
  int i = get_global_id(0);
  float tmp;
  if (i < Ndim) {
    for(j=0;j<Mdim;j++) {
      tmp = 0.0;
      for(k=0;k<Pdim;k++)
        tmp += A[i*Ndim+k] * B[k*Pdim+j];
      C[i*Ndim+j] = tmp;
    }
  }
}

In the host code, launching the kernel becomes:

```c
size_t localWorkSize[1],
    globalWorkSize[1];
localWorkSize[0] = HA/4;
globalWorkSize[0] = HA;
clEnqueueNDRangeKernel(
  clCommandQue, clkernel, 1,
  NULL, globalWorkSize,
  localWorkSize, ...);
```

The resulting performance is a considerable... **drop** from the performance of the initial code (258 MFLOPS !)
Optimizing again

- Our matrix multiplication kernels up to this point have **left all three matrices in global memory**. This means the computation streams rows and columns through the memory hierarchy (global to private) repeatedly for each dot product.

- We can reduce this memory traffic by recognizing that each work-item **reuses the same row of A** for each row of C that is computed.
// Optimized matrix multiplication kernel
// Version 2

__kernel mmul(
    const int Mdim,
    const int Ndim,
    const int Pdim,
    __global float* A,
    __global float* B,
    __global float* C) {

    int k,j;
    int i = get_global_id(0);
    float Awrk[1024];
    float tmp;
    if (i < Ndim) {
        for(k=0;k<Pdim;k++) Awrk[k] = A[i*Ndim+k];
        for(j=0;j<Mdim;j++) {
            tmp = 0.0;
            for(k=0;k<Pdim;k++)
                tmp += Awrk[k] * B[k*Pdim+j];
            C[i*Ndim+j] = tmp;
        }
    }
}
Optimizing once more

- The use of private memory has a dramatic impact on performance: 873 MFLOPS
- But a careful consideration shows that while each work-item reuses its own unique row of A, all the work-items in a group repeatedly stream the same columns of B
- We can reduce the overhead of moving data from global memory if the work-items in a work-group copy the columns of the matrix B into local memory before they start updating their rows of C.
// Optimized matrix multiplication kernel
// Version 3

__kernel mmul(
    const int Mdim, const int Ndim,
    const int Pdim,
    __global float* A,
    __global float* B,
    __global float* C,
    __local float* Bwrk
) {

    int k, j;
    int i = get_global_id(0);
    int iloc = get_local_id(0), nloc = get_local_size(0);
    float Awrk[1024];
    float tmp;
    if (i < Ndim) {
        for(k=0;k<Pdim;k++) Awrk[k] = A[i*Ndim+k];
        for(j=0;j<Mdim;j++) {
            for(k=iloc;k<Pdim;k=k+nloc) Bwrk[k] = B[k*Pdim+j];
            barrier(CLK_LOCAL_MEM_FENCE);
            tmp = 0.0;
            for(k=0;k<Pdim;k++)
                tmp += Awrk[k] * Bwrk[k];
            C[i*Ndim+j] = tmp;
        }
    }
}

Before launching the kernel:
clSetKernelArg(*kernel, 6,
    sizeof(float)*Pdim, NULL);

Relative to local space
Use local instead of global memory
Wait for other items in the group to complete
Performances

<table>
<thead>
<tr>
<th>Matrix Multiplication Optimization Case</th>
<th>MFLOPS</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU: Sequential code in C (no optimization)</td>
<td>167</td>
<td>1</td>
</tr>
<tr>
<td>CPU: $C(i,j)$ per work-item, all global memory</td>
<td>744</td>
<td>4.5</td>
</tr>
<tr>
<td>GPU: $C(i,j)$ per work-item, all global memory</td>
<td>511</td>
<td>3</td>
</tr>
<tr>
<td>GPU: $C$ row per work-item, all global memory</td>
<td>258</td>
<td>1.5</td>
</tr>
<tr>
<td>GPU: $C$ row per work-item, $A$ private, $B$ in global memory</td>
<td>873</td>
<td>5.2</td>
</tr>
<tr>
<td>GPU: $C$ row per work-item, $A$ private, $B$ in local memory</td>
<td>2472</td>
<td>15</td>
</tr>
</tbody>
</table>

The goal is to maximize the amount of work per kernel and optimize memory movement
Optimizing image convolution...

- Image support in OpenCL (\texttt{clCreateImage2D}, etc) provides automatic caching and data access transformations that improve memory system performance, especially on GPU.
- In many circumstances, however, it can be \textit{outperformed by efficient use of local memory}.
- An optimized convolution kernel can be naturally divided into three sections:
  1. The caching of input data from global to local memory
  2. Performing the convolution
  3. The writing of output data back to global memory
Two-dimensional locality (or "remember p.57 ?")

- A 7x7 filter will be considered, although the principles shown here should generalize to different configurations.
- We consider a mapping of work-items using a single work-item per output approach, leaving multi-output optimizations for later.
Selecting workgroup sizes and caching data

- In OpenCL, work-item creation and algorithm design must be considered simultaneously, especially when local memory is used.
- The first approach is to create the same number of work-items as there are data elements to be cached in local memory.
- Each element would simply copy one pixel from global to local memory... and then the work-items representing the border pixels would sit idle during the convolution.
- Consequently, large filter sizes will not allow many output elements to be computed per workgroup.
Selecting workgroup sizes and caching data

- The second approach is to create fewer work-items than pixels to be cached, so some work-items will have to copy multiple elements and none will sit idle during the convolution

- Selecting an efficient workgroup size requires consideration of the underlying memory architecture:
  - For the AMD 6970 GPU, 16 consecutive work-items issuing 128-bit reads on an aligned address can come closest to fully utilizing the memory bus bandwidth
  - The most favorable memory transactions on NVIDIA platforms come from 32 work-items issuing a combined request that is 128 bytes in size and 128-byte aligned (2009)
  - ...

- For this example, creating workgroups of either 32 or 16 items in width offers us a good chance for creating efficient memory requests regardless of platform.
Selecting workgroup sizes and caching data

- For an image with dimensions `imageWidth` and `imageHeight`, only \((imageWidth - paddingPixels) \times (imageHeight - paddingPixels)\) work-items are needed.

- Because the image will likely not be an exact multiple of the workgroup size, additional workgroups must be created: they will not be fully utilized, and this must be accounted for in the kernel.
Computing ND-Range

// This function takes a positive integer and rounds it up to
// the nearest multiple of another provided integer
unsigned int roundUp(unsigned int value, unsigned int multiple) {
    // Determine how far past the nearest multiple the value is
    unsigned int remainder = value % multiple;
    // Add the difference to make the value a multiple
    if(remainder != 0) value += (multiple-remainder);
    return value;
}

...  
int filterWidth = 7, paddingPixels = (int)(filterWidth/2) * 2;
int wgWidth = 16, wgHeight = 16;
// When computing the total number of work-items, the
// padding work-items do not need to be considered
int totalWorkItemsX = roundUp(imageWidth-paddingPixels, wgWidth);
int totalWorkItemsY = roundUp(imageHeight-paddingPixels, wgHeight);
// Size of a workgroup
size_t localSize[2] = {wgWidth, wgHeight};
// Size of the NDRange
size_t globalSize[2] = {totalWorkItemsX, totalWorkItemsY};
int localWidth = localSize[0] + paddingPixels;
int localHeight = localSize[1] + paddingPixels;
Caching Data to Local Memory

- The process of copying data from global memory to local memory is often the most error-prone operation when writing a kernel.
- The work-items first need to determine where in global memory to copy from and then ensure that they do not access a region that is outside of their working area or out of bounds for the image.
Caching Data to Local Memory

```c
__kernel void convolution(
    __global float* imageIn,
    __global float* imageOut,
    __constant float* filter,
    int rows, int cols, int filterWidth,
    __local float* localImage,
    int localHeight, int localWidth) {

    // Determine the amount of padding for this filter
    int filterRadius = (filterWidth/2);
    int padding = filterRadius * 2;
    // Determine the size of the work group output region
    int groupStartCol =
        get_group_id(0)*get_local_size(0);
    int groupStartRow =
        get_group_id(1)*get_local_size(1);
    // Determine the local ID of each work item
    int localCol = get_local_id(0);
    int localRow = get_local_id(1);
    // Determine the global ID of each work item.
    // Work items representing the output region will have a unique global ID
    int globalCol = groupStartCol + localCol;
    int globalRow = groupStartRow + localRow;

    // Cache the data to local memory
    // Step down rows
    for(int i = localRow; i < localHeight; i +=
        get_local_size(1)) {
        int curRow = groupStartRow+i;
        // Step across columns
        for(int j = localCol; j < localWidth; j +=
            get_local_size(0)) {
            int curCol = groupStartCol+j;
            // Perform the read if it is in bounds
            if(curRow < rows && curCol < cols)
                localImage[i*localWidth + j] =
                    imageIn[curRow*cols+curCol];
        } // For j
    } // For i

    barrier(CLK_LOCAL_MEM_FENCE);

    // Perform the convolution
    ...
```
Performance considerations

- Performance on both NVIDIA and AMD GPUs benefits from data alignment in global memory.
- In this example, the choice to have the border pixels not produce values determines that the offset for all workgroups will be a multiple of the workgroup dimensions (i.e., for a 16x16 workgroup, workgroup <N,M> will begin accessing data at column N*16).
Performance considerations

- The only requirement is to pad the input data with extra columns so that its width becomes a multiple of the X-dimension of the workgroup.

- But manually padding a data array on the host can be complicated, time-consuming, and sometimes infeasible.

- To avoid such tedious data fixup, OpenCL has a command called `clEnqueueWriteBufferRect` to copy a host array into the middle of a larger device buffer.

- Other improvements include using vector reads, for example reading `float4` data allows us to come closer to achieving peak memory bandwidth than reading `float` data.

- On the AMD Radeon 6970, a significant performance gain is achieved by using vector reads... but a slight performance degradation was seen on NVIDIA GPUs!
Performing the convolution

```c
// Perform the convolution

if(globalRow < rows-padding &&
   globalCol < cols-padding) {
   // Each work item will filter around its start
   // location (from the filter radius left and up)
   float sum = 0.0f;
   int filterIdx = 0;
   // Not unrolled
   for(int i = localRow; i < localRow+filterWidth; i++) {
      int offset = i*localWidth;
      for(int j = localCol; j < localCol+filterWidth; j++)
         sum += localImage[offset+j] * filter[filterIdx++];
   }
   // Write the data out
   imageOut[(globalRow+filterRadius)*cols + (globalCol+filterRadius)] = sum;
}
return;
```

// Inner loop unrolled
```c
for(int i = localRow; i < localRow+filterWidth; i++) {
   int offset = i*localWidth;
   for(int j = localCol; j < localCol+filterWidth; j++)
      sum += localImage[offset+j] * filter[filterIdx++];
}
```

On an AMD Radeon 6970, with a 7x7 filter and a 600x400 image, unrolling the innermost loop provides a 2.4 speedup. In general, this produces a substantial speedup on both AMD and NVIDIA devices.
Parallel data reduction

- A **reduction** is any algorithm that converts a large data set into a smaller data set using an operator on each element.

- A simple reduction example is to compute the **sum of the elements in an array**, but it could also be min, max, or keep only positive elements, etc.

  ```c
  float sum_array(float * a, int No_of_elements) {
    float sum = 0.0f;
    for (int i = 0; i < No_of_elements; i++) sum += a[i];
    return sum;
  }
  ```

- With OpenCL, the common way to parallelize a reduction is to divide the input data set between different work groups on a GPU, where each work group is responsible for computing a single element.
General methodology

- Within a work group, the reduction is performed over multiple stages
- At each stage, work-items sum an element and its neighbor that is one stride away.
Reduction kernel

// A simple reduction tree kernel where each work group reduces a set
// of elements to a single value in local memory and writes the
// resultant value to global memory.
__kernel void reduction_kernel( unsigned int N, // number of elements to reduce
                               __global float* input, __global float* output,
                               __local float* sdata) {

  // Get index into local data array and global array
  unsigned int localId = get_local_id(0), globalId = get_global_id(0);
  unsigned int groupId = get_group_id(0), wgSize = get_local_size(0);

  // Read in data if within bounds
  sdata[localId] = (globalId<N) ? input[globalId]: 0;

  // Synchronize since all data needs to be in local memory and visible to all work items
  barrier(CLK_LOCAL_MEM_FENCE);

  // Each work item adds two elements in parallel. As stride increases, work items remain idle.
  for(int offset = wgSize ; offset > 0; offset >>= 1) {
    if (localId < offset && localId + offset < wgSize)
      sdata[localId] += sdata[localId + offset];
  }

  // Only one work item needs to write out result of the work group’s reduction
  barrier(CLK_LOCAL_MEM_FENCE);
  if ( localId == 0 ) output[groupId] = sdata[0];
}
Improving reduction performances
(see "OpenCL Optimization Case Study: Simple Reductions")

• At each step of the reduction tree, the active work-items get sparser and sparser.

• This leads to poor SIMD efficiency: we have only about 30% of the work-items active, on average.
Improving reduction performances
(see "OpenCL Optimization Case Study: Simple Reductions")

- Reductions using atomics: operations such as `atom_add()` can reduce the partial results from each local reduction. But they are limited to the operators and data-types supported by the platform.

- Two-stage reduction: the input is divided up chunks large enough to keep all of processors busy. The final global reduction is performed sequentially, which improves efficiency compared to the fully-parallel multi-stage reduction.
CPU reduction (min) kernel

// Each work-group is a serial work-group with only one work-item
__kernel void reduce(__global float* buffer, __const int block,
    __const int length, __global float* result) {
    int global_index = get_global_id(0) * block;
    float accumulator = INFINITY;
    int upper_bound = (get_global_id(0) + 1) * block;
    if (upper_bound > length) upper_bound = length;
    while (global_index < upper_bound) {
        float element = buffer[global_index];
        accumulator = (accumulator < element) ? accumulator : element;
        global_index++;
    }
    result[get_group_id(0)] = accumulator ;
}
Another reduction example: computing an image histogram

- The **histogram of an image** provides a frequency distribution of pixel values in the image.
- We have either a single histogram if the luminosity is used as the pixel value or three histograms if the R, G, and B color channel values are used.
- The principle of the histogram algorithm is to perform an operation over each pixel of the image:
  ```cpp
  for (many input values)
  histogram[value]++;
  ```
Sequential Histogram

unsigned int * histogram_rgba (void *image_data, int w, int h) {
    int *ptr = (unsigned int*)malloc(256 * 3 * sizeof(unsigned int));
    int *ptr_orig = ptr ;
    memset(ptr, 0x0, 256 * 3 * sizeof(unsigned int));
    // compute histogram for R
    for (i=0; i<w*h*4; i+=4) ptr[img[i]]++;
    // compute histogram for G
    ptr += 256;
    for (i=1; i<w*h*4; i+=4) ptr[img[i]]++;
    // compute histogram for B
    ptr += 256;
    for (i=2; i<w*h*4; i+=4) ptr[img[i]]++;
    return ptr_orig ;
}
Parallelizing the Image Histogram

Input image

Computing local histograms in local memory

Input data
Local histogram bins
Local histogram bins
Local histogram bins
Local histogram bins

Local histograms in global memory

Final reduced histogram
Atomic operations

• Since many work-items execute in parallel in a workgroup, we cannot guarantee the ordering of read-after-write dependencies on our local histogram bins.

• We could reproduce the histogram bins multiple times, but this would require a copy of each bin for each work-item in the group.

• The alternative solution is to use hardware atomics: any time two threads operate on a shared variable concurrently, and one of those operations performs a write, both threads must use atomic operations.

• Using atomics requires an OpenCL device that implements the `cl_khr_local_int32_base_atomics` extension.

• [https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/atomicFunctions.html](https://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml/atomicFunctions.html)
parallel histogram kernel

```
kernel void histogram_image_rgba (image2d_t img, int num_pixels_per_workitem, global uint *histogram) {

    int local_size = get_local_size(0) * get_local_size(1);
    int image_width = get_image_width(img);
    int image_height = get_image_height(img);
    int group indx = 256 * 3 (get_group_id(1) * get_num_groups(0) + get_group_id(0));
    int x = get_global_id(0);
    int y = get_global_id(1);
    local uint tmp_histogram[256 * 3];
    int tid = get_local_id(1) * get_local_size(0) + get_local_id(0);
    int j = 256 * 3;
    int indx = 0;
    int i, idx;

    // clear the buffer for the partial histogram
    do {
        if (tid < j) tmp_histogram[indx+tid] = 0;
        j -= local_size; indx += local_size;
    } while (j > 0);

    barrier(CLK_LOCAL_MEM_FENCE);

    for (i=0, idx=x; i<num_pixels_per_workitem; i++, idx+=get_global_size(0)) {
        if ((idx < image_width) && (y < image_height)) {
            const sampler_t samplerA =
                CLK_NORMALIZED_COORDS_FALSE |
                CLK_ADDRESS_CLAMP_TO_EDGE |
                CLK_FILTER_NEAREST;
            float4 clr =
                read_imagef(img, samplerA, (int2)(idx, y));
            uchar indx_x, indx_y, indx_z;
            indx_x = convert_uchar_sat(clr.x * 255.0f);
            indx_y = convert_uchar_sat(clr.y * 255.0f);
            indx_z = convert_uchar_sat(clr.z * 255.0f);
            atom_inc(&tmp_histogram[indx_x]);
            atom_inc(&tmp_histogram[256+(uint)indx_y]);
            atom_inc(&tmp_histogram[512+(uint)indx_z]);
        }
    }

    barrier(CLK_LOCAL_MEM_FENCE);

    ...}
```
Parallel histogram kernel (continued)

... 

// copy the partial histogram to appropriate location in histogram given by group_indx
if (local_size >= (256 * 3)) {
  if (tid < (256 * 3))
    histogram[group_indx + tid] = tmp_histogram[tid];
} else {
  j = 256 * 3;
  indx = 0;
  do {
    if (tid < j)
      histogram[group_indx + indx + tid] =
        tmp_histogram[indx + tid];
    j -= local_size;
    indx += local_size;
  } while (j > 0);
}

kernel void histogram_sum_partial_results (  
global uint *partial_histogram, int num_groups,  
global uint *histogram) {

  int tid = (int)get_global_id(0);  
  int group_indx;
  int n = num_groups;  
  uint tmp_histogram;
  tmp_histogram = partial_histogram[tid];
  group_indx = 256*3;
  while (--n > 0) {
    tmp_histogram +=
      partial_histogram[group_indx + tid];
    group_indx += 256*3;
  }
  histogram[tid] = tmp_histogram;
}
GDB with OpenCL
Thrust, OpenAcc, etc