OpenCL

Module 6

INF8601 Systèmes informatiques parallèles

Michel Dagenais

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

Sommaire

OpenCL

Historique

- Open Computing Language.
- Proposé en 2008 par Apple après avoir été retravaillé avec l'aide de AMD (ATI), Intel (MIC), IBM (Cell processor) et NVIDIA (CUDA).
- Standard développé et maintenu par le consortium Khronos (OpenGL, Collada).
- Version 1.0 en 2008, 1.1 en 2010, 1.2 en 2011
- Version 2.0 en 2013 (mémoire virtuelle partagée, opérations atomiques C11, tubes), 2.1 en 2015 (un peu de C++, intégration avec Vulkan, priorités, minuteries), 2.2 (C++, optimisations)
- Support actif par AMD, IBM, Intel, ARM et aussi NVIDIA sous CUDA.

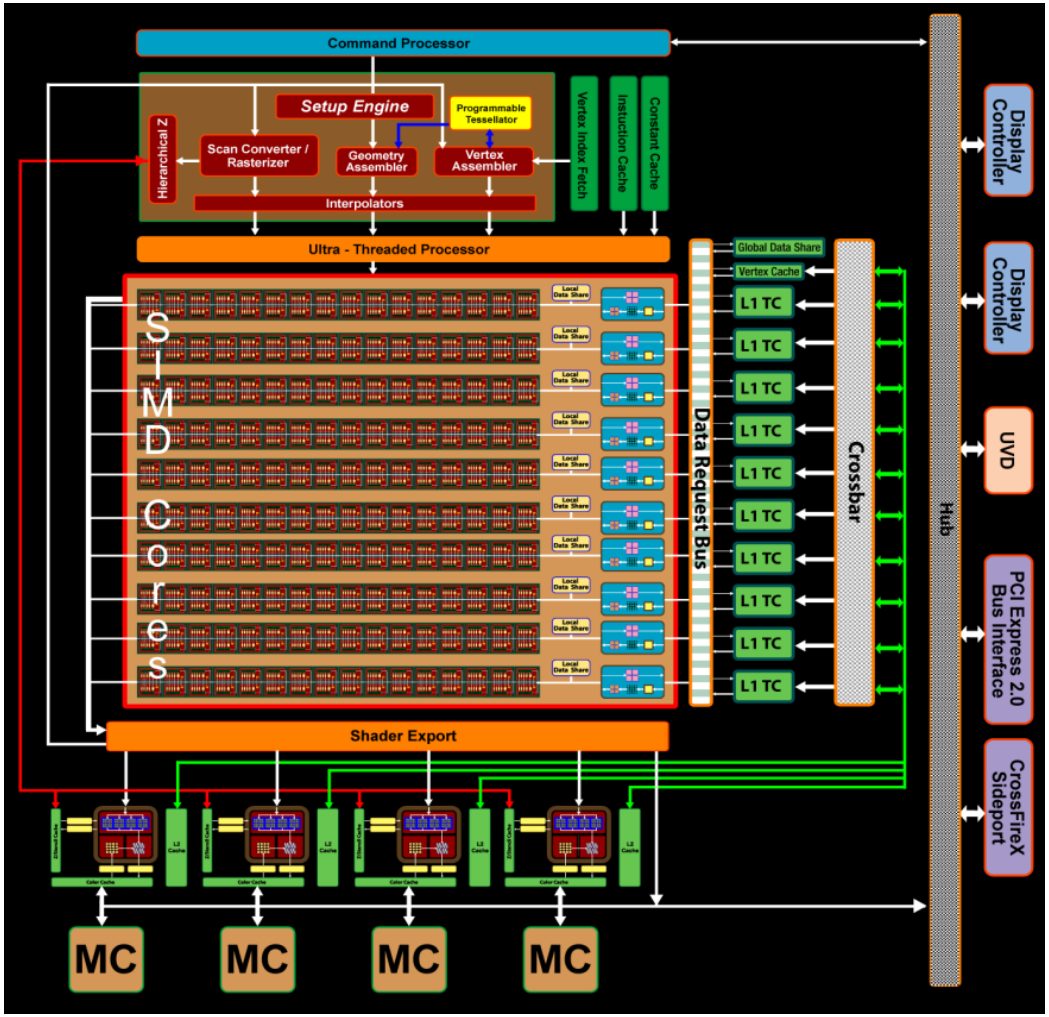
Objectifs

- Permettre d'utiliser les processeurs parallèles auxiliaires, unités vectorielles (MMX), processeurs graphiques GPGPU (AMD, NVIDIA), autres processeurs de calcul (IBM Cell Synergistic Processor Elements, Intel Many Integrated Core), et même les processeurs dans les FPGA.
- Offre un environnement et langage normalisé pour utiliser de manière portable ces systèmes parallèles hétérogènes.
- Standard ouvert avec implémentation de référence libre.

Public cible

- Potentiel d'accélération important, e.g. 10 ou 100 fois plus rapide.
- Décrit comme des outils qui permettent aux programmeurs experts de faire des gains de vitesse appréciables pour les applications exigeantes.
- Coût de développement important en temps et en complexité.
- Outils de mise au point et d'analyse de performance spécifiques, et moins disponible et développés.
- Exemples d'applications: jeux, logiciels de CAO, calculs scientifiques (grappes de calcul), craquer des codes de chiffrement...

Exemple: ATI RADEON HD4870



Source: AMD.com, OpenCL - Parallel computing for CPUs and

Configurations typiques

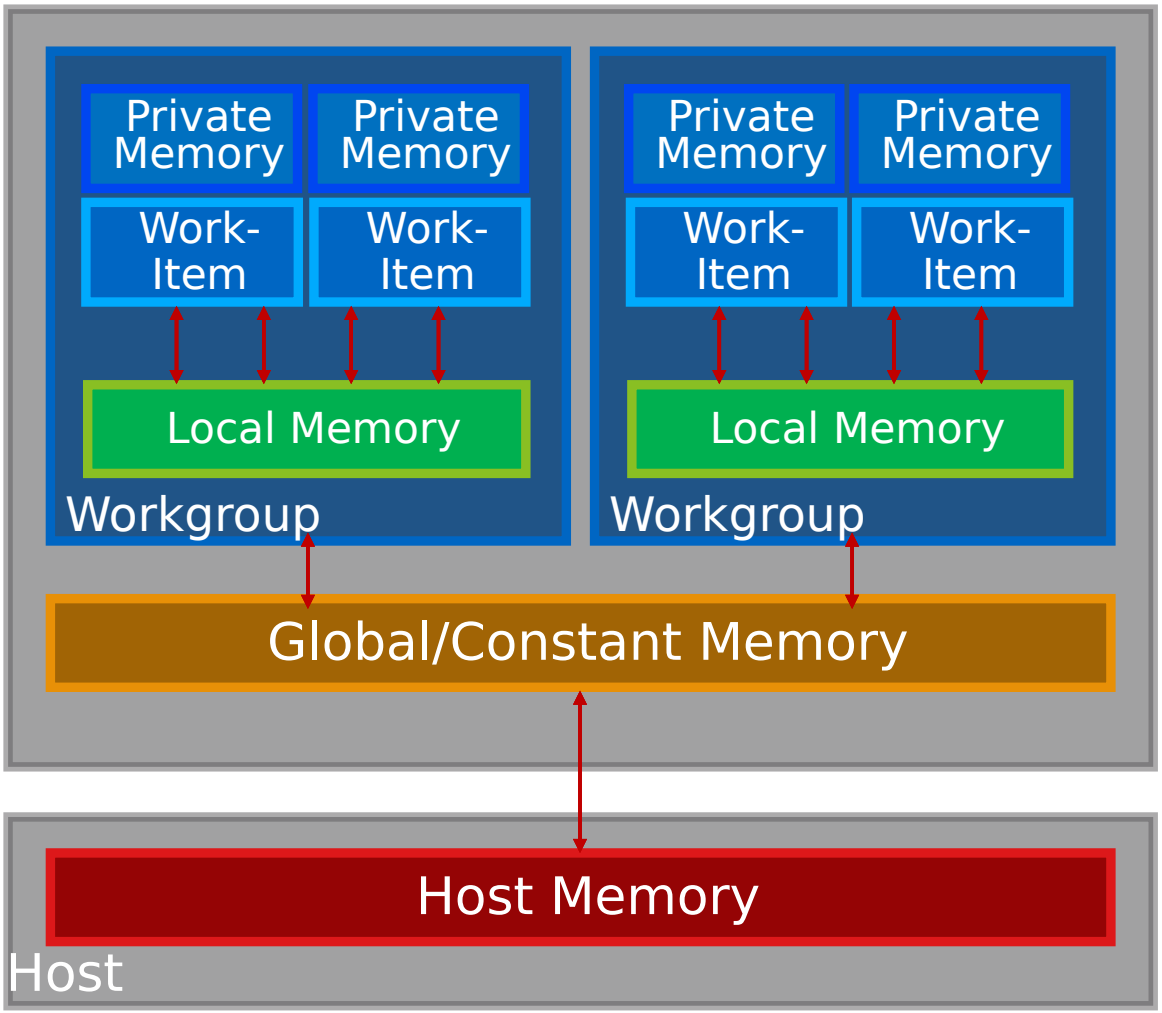
- AMD Radeon RX 7900 XTX, 96 processeurs SIMD, 6144 ALU, bus 384 bits, 57 milliards transistors, 122/61/1.9 TFLOPS.
- NVIDIA RTX 4090, 128 processeurs SIMD, 16384 ALU, bus 384 bits, 76 milliards transistors, 82/82/1.2 TFLOPS.
- AMD Radeon MI300, 220 processeurs SIMD, 14080 ALU, bus 8192 bits, 146 milliards transistors, 383/47/47 TFLOPS.
- NVIDIA GH100, 132 processeurs SIMD, 16896 ALU, bus 5120 bits, 80 milliards transistors, 267/66/33 TFLOPS.

Concepts importants

- Ordinateur hôte (host), exécute le programme principal.
- Dispositif de calcul (device), par exemple une carte graphique.
- Un dispositif est constitué de processeurs SIMD (compute unit) qui peuvent prendre en charge un ou des groupes de travail (work group).
- Un processeur SIMD est constitué de plusieurs ALU (processing element), chacun pouvant prendre un ou plusieurs item de travail (work item).
- Le programme principal définit des objets en mémoire (buffer, image), des fonctions parallèles (kernel) et des événements (events). Il met en queue les fonctions parallèles à exécuter.

OpenCL

Le modèle de mémoire



Source: AMD.com, OpenCL - Parallel computing for CPUs and

Modèle de mémoire

Mémoire	Globale	Constante	Locale	Privée
Host	Allocation dynamique Lecture et écriture	Allocation dynamique Lecture et écriture	Allocation dynamique Pas d'accès	Acun accès ni allocation
Kernel	Ne peut allouer Lecture et écriture	Allocation statique Lecture seulement	Allocation statique Lecture et écriture	Allocation statique Lecture et écriture

Modèle de mémoire

- La mémoire globale est très lente (DRAM) comparée à la mémoire locale (SRAM). Une mémoire cache est insérée entre les deux.
- Le programme principal peut allouer des tampons (buffer) ou images (2D et 3D).
- Le programme principal peut les copier ou les calquer entre sa mémoire et la mémoire globale du dispositif.
- Cohérence faible, des barrières mémoire sont requises entre les opérations sur des mêmes éléments de données.

OpenCL

Accéder au dispositif

```
/* notre ordinateur est la plate-forme */

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

/* liste des dispositifs disponibles, usuellement 1 GPU
   on peut spécifier le type cherché avec device type */

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

/* Le device info donne l'information sur le nombre de
   workgroup ainsi que le nombre de work item sur le
   dispositif ainsi que le support 64 bits, les tailles
   d'images... */

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)
```

Contexte et queues

```
/* Le contexte définit tout ce qui est associé à notre
   programmation sur ce dispositif pour un travail. */

cl_context clCreateContext (const cl_context_properties
    *properties, cl_uint num_devices, const cl_device_id
    *devices, void (*pfn_notify)(const char *errinfo...),
    void *user_data, cl_int *errcode_ret)

/* Une queue de commande permet d'envoyer le travail au GPU.
   La queue peut être demandée IN_ORDER ou non. On peut avoir
   plus d'une queue pour des tâches indépendantes. */

cl_command_queue clCreateCommandQueue (cl_context context,
    cl_device_id device, cl_command_queue_properties
    properties, cl_int *errcode_ret)
```

Divers

- Pour plusieurs types d'objets différents (e.g. context, command queue...) on retrouve les fonctions suivantes.
- Retain/Release pour faire un décompte de référence et libérer l'objet lorsque le décompte tombe à 0.
- GetInfo pour savoir les différentes propriétés d'un objet, incluant le décompte de référence.
- Plusieurs fonctions prennent des événements pré-requis en entrée et founissent un événement en sortie à des fins de synchronisation.

Exemple complet: device et context

```
/* Initialiser le matériel, contexte et queue de commandes */

cl_int error = 0;
cl_platform_id platform;
cl_context context;
cl_command_queue queue;
cl_device_id device;

error = clGetPlatformIDs(1, &platform, NULL);
if (error != CL_SUCCESS) { ErrorExit(error); }

error = clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1,
    &device, NULL);
if (err != CL_SUCCESS) { ErrorExit(error); }

context = clCreateContext(0, 1, &device, NULL, NULL, &error);
if (error != CL_SUCCESS) { ErrorExit(error); }

queue = clCreateCommandQueue(context, device, 0, &error);
if (error != CL_SUCCESS) { ErrorExit(error); }
```

Gestion de la mémoire

- Un tampon peut être créé pour lecture, écriture ou les deux (par les fonctions kernel).
- Le tampon peut prendre la mémoire hôte spécifiée, allouer de la mémoire hôte, ou allouer de la mémoire globale. La mémoire allouée peut être copiée à partir d'une adresse hôte.
- On peut définir un sous-tampon dans un tampon.
- Des commandes permettent de lire ou écrire, bloquant ou non, un tampon, sous-tampon, ou section de tampon vers la mémoire hôte, ou vers un autre tampon.
- Une commande permet de calquer un tampon sur la mémoire hôte.

Tampons en mémoire

```
/* créer un tampon */
cl_mem clCreateBuffer (cl_context context, cl_mem_flags flags,
    size_t size, void *host_ptr, cl_int *errcode_ret)

/* définir un sous-tampon */
cl_mem clCreateSubBuffer (cl_mem buffer, cl_mem_flags flags,
    cl_buffer_create_type buffer_create_type, const void
    *buffer_create_info, cl_int *errcode_ret)

/* commande pour lire du tampon vers la mémoire hôte */
cl_int clEnqueueReadBuffer (cl_command_queue command_queue,
    cl_mem buffer, cl_bool blocking_read, size_t offset,
    size_t cb, void *ptr, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)

/* commande pour écrire de la mémoire hôte vers le tampon */
cl_int clEnqueueWriteBuffer (cl_command_queue command_queue,
    cl_mem buffer, cl_bool blocking_write, size_t offset,
    size_t cb,const void *ptr,cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)
```

Images

- Une image est un tampon dans un format pré-défini.
- On peut créer une image 2D ou 3D dans un des formats supportés (clGetSupportedImageFormats) avec une certaine taille, bits par couleur/pixel...
- Des fonctions permettent d'accéder au contenu de l'image.
- Il est possible d'envoyer des commandes pour lire, écrire ou copier des images, avec la mémoire hôte, une autre image ou un tampon.
- Il est aussi possible de calquer une image en mémoire hôte.

Images en mémoire

```
/* créer une image */
cl_mem clCreateImage2D (cl_context context,cl_mem_flags flags,
    const cl_image_format *image_format, size_t image_width,
    size_t image_height, size_t image_row_pitch,
    void *host_ptr, cl_int *errcode_ret)

/* lister les formats suppportés */
cl_int clGetSupportedImageFormats (cl_context context,
    cl_mem_flags flags, cl_mem_object_type image_type,
    cl_uint num_entries, cl_image_format *image_formats,
    cl_uint *num_image_formats)

/* lire une image */
cl_int clEnqueueReadImage (cl_command_queue command_queue,
    cl_mem image, cl_bool blocking_read, const size_t
    origin[3], const size_t region[3], size_t row_pitch,
    size_t slice_pitch,void *ptr, cl_uint
    num_events_in_wait_list, const cl_event *event_wait_list,
    cl_event *event)
```

Exemple complet: allocation

```
/* Nous voulons 3 vecteurs de float */

const int size = 1000000;
float* a = new float[size];
float* b = new float[size];
float* C = new float[size];

for (int i = 0; i < size; i++) {
  a = i; b = size - i;
}

const int buffer_size = sizeof(float) * size;

cl_buffer_a = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, buffer_size, a, &error);
cl_buffer_b = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, buffer_size, b, &error);
cl_buffer_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    buffer_size, NULL, &error);
```

OpenCL

Programme hôte versus dispositif

- La plus grande partie du programme est en C ou C++ et est compilée pour l'hôte.
- La partie qui s'exécute sur les ALU, fonctions avec l'attribut kernel, doit être compilée pour le bon dispositif à l'exécution. Elle peut être pré-compilée en langage intermédiaire.
- Le programme principal doit charger les fichiers OpenCL, les compiler et ensuite mettre en queue de commande les fonctions kernel désirées.

Le programme OpenCL

```
/* Lire les fichiers OpenCL pour en faire un programme */
cl_program clCreateProgramWithSource (cl_context context,
    cl_uint count, const char **strings,const size_t *lengths,
    cl_int *errcode_ret)

/* Lire les fichiers OpenCL pré-compilés en IR
   pour en faire un programme */
cl_program clCreateProgramWithBinary (cl_context context,
    cl_uint num_devices, const cl_device_id *device_list,
    const size_t *lengths, const unsigned char **binaries,
    cl_int *binary_status, cl_int *errcode_ret)

/* Compiler le programme */
cl_int clBuildProgram (cl_program program,cl_uint num_devices,
    const cl_device_id *device_list, const char *options,
    void (CL_CALLBACK *pfn_notify)(cl_program program,
    void *user_data), void *user_data)
```

Le programme OpenCL

```
/* Obtenir le point d'entrée d'une fonction spécifiée */
cl_kernel clCreateKernel (cl_program program, const char
    *kernel_name, cl_int *errcode_ret)

/* Obtenir toutes les fonctions */
cl_int clCreateKernelsInProgram (cl_program program,
    cl_uint num_kernels, cl_kernel *kernels,
    cl_uint *num_kernels_ret)

/* Spécifier les arguments à associer à la fonction en vue
   de mettre en queue l'exécution de cette fonction */
cl_int clSetKernelArg (cl_kernel kernel, cl_uint arg_index,
    size_t arg_size, const void *arg_value)
```

Exemple complet: le kernel

```
size_t size;
const char* src = oclLoadProgSource("/tmp/abc.cl", "", &size);
cl_program pgm = clCreateProgramWithSource(context, 1, &src,
    &size, &error);
if(error != CL_SUCCESS) { ErrorExit(error); }

error = clBuildProgram(pgm, 1, &device, NULL, NULL, NULL);
if(error != CL_SUCCESS) { ErrorExit(error); }

char* bld_info;
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
    0, NULL, &size);
bld_info = new char[size+1]; bld_info[size] = '\0';
clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
    size, bld_info, NULL);

cl_kernel abc_kernel = clCreateKernel(pgm,"abc",&error);
if(error != CL_SUCCESS) { ErrorExit(error); }
```

Exécution de fonctions OpenCL

- Exécuter une tâche sur un processeur (SIMD). Cette tâche peut opérer sur des vecteurs pour tirer parti des nombreux ALU, différentes tâches peuvent aller sur les différents processeurs, et plusieurs tâches peuvent être assignées au même processeur en hyperthreading.
- Exécuter une même fonction sur un grand nombre d'ALU dans plusieurs processeurs. Le travail peut être décrit sur 1, 2 ou 3 dimensions, avec pour chaque dimension une taille globale et une taille locale (e.g. 1024x1024 versus 128x128).

Exécution de fonctions OpenCL

```
/* Mettre en queue sur global size par groupe de local size */
cl_int clEnqueueNDRangeKernel (cl_command_queue command_queue,
    cl_kernel kernel, cl_uint work_dim, const size_t
    *global_work_offset, const size_t *global_work_size,
    const size_t *local_work_size, cl_uint
    num_events_in_wait_list, const cl_event *event_wait_list,
    cl_event *event)

/* Mettre en queue pour exécution sur un processeur */
cl_int clEnqueueTask (cl_command_queue command_queue,
    cl_kernel kernel, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list, cl_event *event)

/* Mettre en queue sur un processeur de type hôte */
cl_int clEnqueueNativeKernel (cl_command_queue command_queue,
    void (*user_func)(void *) void *args, size_t cb_args,
    cl_uint num_mem_objects, const cl_mem *mem_list,
    const void **args_mem_loc, cl_uint num_events_in_wait_list,
    const cl_event *event_wait_list,cl_event *event)
```

Synchronisation par événements

- Le programme hôte peut créer des événements et changer leur statut pour contrôler quand certaines commandes pourront commencer.
- Il peut attendre après certains événements.
- Il peut demander une fonction de rappel lors du changement d'état d'un événement. La fonction de rappel est très limitée dans ce qu'elle peut appeler.

Synchronisation par événements

```
/* Créer un événement sur lequel faire dépendre */
cl_event clCreateUserEvent (cl_context context, cl_int
    *errcode_ret)

/* Changer le statut à prêt ou à erreur */
cl_int clSetUserEventStatus (cl_event event, cl_int
    execution_status)

/* Attendre après des événements */
cl_int clWaitForEvents (cl_uint num_events, const cl_event
    *event_list)

/* Associer une fonction de rappel à un évémement */
cl_int clSetEventCallback (cl_event event, cl_int
    command_exec_callback_type, void (CL_CALLBACK
    *pfn_event_notify)(cl_event event, cl_int
    event_command_exec_status, void *user_data),
    void *user_data)
```

Barrières

- Les commandes dans une queues IN_ORDER sont sérialisées et les résultats de la commande précédente sont disponibles pour la suivante.
- Les différentes commandes d'une queue OUT_OF_ORDER, ou les commandes de queues distinctes ne sont pas synchronisées.
- Marqueur pour savoir lorsque les commandes précédentes d'une queue sont terminées.
- Barrière pour assurer que toutes les commandes précédentes sont faites avant de commencer les suivantes.
- Commande d'attente d'événements peut être mise en queue.

Barrières

```
/* Indique que toutes les commandes antérieures sont
   terminées */
cl_int clEnqueueMarker (cl_command_queue command_queue,
    cl_event *event)

/* Assure que toutes les commandes antérieures sont terminées
   avant que les commandes postérieures ne commencent */
cl_int clEnqueueBarrier (cl_command_queue command_queue)

/* Attend après les événements spécifiés avant de poursuivre */
cl_int clEnqueueWaitForEvents (cl_command_queue command_queue,
    cl_uint num_events, const cl_event *event_list)
```

Divers

- L'événement de fin d'une commande peut contenir de l'information sur l'exécution: temps de mise en queue, soumission, début et fin. Ceci permet de comprendre la performance de l'application.
- Lorsqu'une fonction de rappel met une commande en queue, celle-ci n'est pas soumise à ce moment. Il faut faire un Flush sur la queue.
- On peut attendre pour la fin de l'exécution de toutes les commandes dans une queue (Finish).

Divers

```
/* Extraire l'information sur le temps d'exécution d'un
   événement */
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)

/* Activer la soumission des commandes de la queue */
cl_int clFlush (cl_command_queue command_queue)

/* Attendre que toutes les commandes en queue soient
   terminées */
cl_int clFinish (cl_command_queue command_queue)
```

Exemple complet: exécution

```
error = clSetKernelArg(abc_kernel, 0, sizeof(cl_mem),
    &buffer_a);
error |= clSetKernelArg(abc_kernel, 1, sizeof(cl_mem),
    &buffer_b);
error |= clSetKernelArg(abc_kernel, 2, sizeof(cl_mem),
    &buffer_c);
error |= clSetKernelArg(abc_kernel, 3, sizeof(size_t), &size);
if(error != CL_SUCCESS) { ErrorExit(error); }

const size_t wg_size = 512;
const size_t total_size = ((1000000 / 512) + 1) * 512;
error = clEnqueueNDRangeKernel(queue, abc_kernel, 1, NULL,
    &total_size, &wg_size, 0, NULL, NULL);
if(error != CL_SUCCESS) { ErrorExit(error); }

clEnqueueReadBuffer(queue, buffer_c, CL_TRUE, 0, buffer_size,
    c, 0, NULL, NULL);
```

Exemple complet: finalisation

```
delete[] a;
delete[] b;
delete[] c;
clReleaseKernel(abc_kernel);
clReleaseCommandQueue(queue);
clReleaseContext(context);
clReleaseMemObject(buffer_a);
clReleaseMemObject(buffer_b);
clReleaseMemObject(buffer_c);
```

OpenCL

Les fonction kernel OpenCL

- Sous-ensemble du C99 avec extensions spécifiques dans un fichier .cl.
- Types scalaires usuels, bool, char, short, int, long, signés ou non, ainsi que float, half, (double), size_t, ptr_diff_t, intptr_t.
- Mêmes types vectoriels (de 2, 3, 4, 8 ou 16 éléments) spécifiés par exemple sous la forme intn ou floatn (int8, float4).
- Par défaut, les composantes d'un vecteur de 4 sont x, y, z et w.

Types OpenCL

```
float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
pos.xw = (float2)(5.0f, 6.0f);
pos.xyz = (float3)(3.0f, 5.0f, 9.0f);

a.xyzw = f.s0123;

float4 vf;
float2 low = vf.lo;
float2 high = vf.hi;
float2 even = vf.even;
float2 odd = vf.odd;

uchar4 u;
int4 c = convert_int4(u);

float f;
int i = convert_int(f);

union{ float f; uint u; double d;} u;
```

Opérations mathématiques

- Tous les opérateurs arithmétiques usuels, relations logiques et comparaisons sur les scalaires, scalaires-vecteurs et vecteurs-vecteurs.
- La plupart des fonctions usuelles, incluant les fonctions trigonométriques sont offertes et fonctionnent avec des vecteurs.
- Fonctions pour lire ou écrire un vecteur à partir d'un pointeur.

Opérations mathématiques

```
float4 u, v;
float f;

v = u + f;

u = atan2(v);

float distance(floatn p0, floatn p1);

float length(floatn p);

float8 vload8(size_t offset, const __local float *p);

void vstore8(float8 data, size_t offset, __local float *p);
```

Attributs

- Variables dans 4 espaces distincts: __global, __local (partagé dans le workGroup), __constant (lecture seulement), __private (par défaut, pour le workItem).
- Les types image2d_t et image3d_t sont des objets dans __global.
- Arguments read_only ou write_only pour les fonctions kernel.
- Peut spécifier l'alignement, __attribute__ ((aligned(8)))

Zones mémoire

```
__global float4 *color;

typedef struct {
  float a[3];
  intb[2];
} foo_t;

__global foo_t *my_info;

__kernel void my_func(...) {
  __local float a;
  __local float b[10];
  float c;
  ...
}
```

Synchronisation

- void work_group_barrier (cl_mem_fence_flags flags): synchroniser tous les workItem du workGroup. (Anciennement barrier).
- void mem_fence (cl_mem_fence_flags flags) , void read_mem_fence (cl_mem_fence_flags flags), void write_mem_fence (cl_mem_fence_flags flags): barrières mémoires pour les lectures ou écritures en mémoire du workItem.

Autres fonctions

- Copie asynchrone entre la mémoire locale et globale. Un événement est associé à chaque opération et peut être utilisé par wait_group_events.
- Opérations atomiques (add, sub, xchg, inc, dec, cmpxchg, min, max, and, or, xor) sur des entiers ou float en mémoire locale ou globale.
- Lecture ou écriture d'un pixel d'une image 2d ou 3d. Différents modes d'accès (sampler_t) pour les images.

Exemple complet: code OpenCL

```
/* Simple addition vectorielle c = a + b */

__kernel void abc_kernel (__global const float* buffer_a,
    __global const float* buffer_b, __global float* buffer_c,
    const int nb) {

  /* Nous sommes en une seule dimension, l'index global nous
     indique l'élément sur lequel travailler */

  const int id = get_global_id(0);

  /* Nous avons quelques work_item de plus que la var taille
     il faut donc s'assurer de ne pas dépasser. */

  if (id < nb) buffer_c[id] = buffer_a[id] + buffer_b[id];
}
```

OpenCL

Optimisation

- Minimiser les interactions entre l'hôte et le GPU.
- Regrouper les accès mémoire consécutifs en lignes complètes alignées.
- Avoir un grand nombre de fils possibles sur chaque processeur SIMD afin d'avoir toujours quelque chose à faire lors des attentes pour la mémoire et le décodage des instructions.
- Utiliser le matériel au maximum en tenant compte de la capacité (mémoire locale, registres) des processeurs SIMD.
- Utiliser les instructions vectorielles (au moins jusqu'à taille 4).

Accès en mémoire

```
/* c[m][n], OpenCL organisé en m*n */

int x = get_global_id(0);
int y = get_global_id(1);

c[x][y] = a[x] * b[y];

/* OpenCL organisé en n*m */

int x = get_global_id(1);
int y = get_global_id(0);

c[x][y] = a[x] * b[y];

/* get_global_id(0) varie le plus vite, il faut donc le mettre
   pour y, de manière à faire des accès à des cases mémoire
   consécutives en même temps qui pourront être regroupés.
   Peut être plusieurs fois plus rapide! */
```

Maximisation du nombre d'items

- Déterminer le nombre de registre et la quantité de mémoire locale requise par un Work Item.
- Selon la quantité de registre et de mémoire disponible sur un SIMD, cela détermine la taille du Work Group.
- De cette manière, on maximise le nombre de fils possible sur chaque processeur SIMD, permettant de maintenir le matériel toujours occupé.
- On suppose que la taille totale divisée par la taille de Work Group est largement supérieure au nombre de SIMD.

Vectorisation

- L'augmentation du nombre de Work Item est plus fiable que l'utilisation d'opérations vectorielles pour augmenter la performance.
- Une instruction vectorielle fait des accès mémoire regroupés et demande une seule instruction.
- Sur processeur Intel ou sur GPU AMD, les vecteurs float4 sont traités efficacement car certains éléments travaillent sur 4 mots à la fois.

OpenCL

Conclusion

- Quel langage prendre, CUDA qui est plus répandu mais propriétaire et spécifique à une plate-forme, ou OpenCL qui est une norme reconnue?
- HIP est une alternative proposée par AMD, langage ouvert très proche de CUDA qui permet de cibler les processeurs NVIDIA sous CUDA ou les processeurs AMD.
- La programmation en OpenCL demande des connaissances plus poussées en programmation.
- Les gains possibles en performance sont très importants.
- Optimiser un programme sur un GPU n'est pas évident!