Autres Outils pour le GPGPU

Xavier JUVIGNY

ONERA

December 3, 2019

Plan du cours

OpenCL en quelques mots

Pourquoi OpenCL

- CUDA : bibliothèque conviviale, puissante et rapide mais uniquement portable sur des cartes NVIDIA !;
- Besoin d'avoir une bibliothèque plus universelle permettant de gérer des accélérateurs de calcul, d'autres cartes graphiques, utilisable sur smartphone et tablettes, etc..
- Permettre une accélération de calcul pour les pages web : WebCL.

OpenCL en quelques mots

- Standard mis au point par le Khronos Group (qui font aussi la standardisation d'OpenGL);
- Permet la programmation des GPGPUs, mais aussi des CPUs (Intel mais aussi les CELLs d'IBM);
- Compilateur intégré à la bibliothèque (comme pour les shaders avec OpenGL);

OpenCL : Pour et Contre

Pros

- Portable sur un grand nombre de plateformes;
- Programmation des noyaux proche de CUDA;
- Standard ouvert non propriétaire;
- Support de plusieurs versions d'OpenCL prévu !

Cons

- L'API pour la compilation et l'exécution des noyaux est complexe et lourde;
- Moins performante que CUDA sur les NVIDIAs;
- Intel pour ces processeurs many-cœurs a plutôt choisi les options multithreading (TBB en particuliers pour les Knights Landing);

Programmation du noyau

Noyau OpenCL

```
global float filter[N];
kernel void
convolve(float* image) {
    local float region[M];
    ...
    int ind =
          get_global_id(0);
    region[ind] = image[i];
    barrier(CLK_LOCAL_MEM_FENCE);
    ...
    image[j] = result;
}
```

Noyau CUDA

```
__device__ float filter[N];
__global__ void
    convolve(float* image) {
    __shared__ float region[M];
    ...
    int ind = threadId.x+
          blockId.x*blockDim.x;
    region[ind] = image[i];
    __syncthreads();
    ...
    image[j] = result;
}
```

API d'OpenCL : Plateforme

Plateforme

- Plateforme OpenCL $\equiv$ mise en œuvre du standard OpenCL;
- Plusieurs plateformes possibles sur une machine donnée;
- clGetPlatformIDs(cl_uint nb_entries,cl_platform_id *platforms,cl_uint *nb_platforms) :

```
cl_uint nbEntries;
clGetPlatformIDs(0, nullptr, &nbEntries);
std::vector<cl_platform_id> platforms(nbEntries);
clGetPlatformIDs(platforms.size(), platforms.data(), nullptr);
```

- On peut ensuite interroger chaque plateforme pour connaître les device supportés et leur type (CPU ou GPGPU) `clGetDeviceIDs(cl_platform_id platform, cl_device_type device_type, cl_uint nb_entries, cl_device_id *dev, cl_uint* nb_dev )` :

```
cl_uint nbDev;
clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 0,
               nullptr, &nbDev);
std::vector<cl_device> devs(nbDev);
clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, nbDev,
               devs.data(), nullptr);
```

API d'OpenCL : contexte

- Pour chaque device utilisé, il faut créer un contexte;
- Un contexte en OpenCL permet de gérer les queues de commande, la mémoire
- le programme et les noyaux OpenCL;
- `cl_context clCreateContext( cl_context_properties *properties, cl_uint num_devices, const cl_device_id *devices, void *pfn_notify ( const char *errinfo, const void *private_info, size_t cb, void *user_data), void *user_data, cl_int *errcode_ret )` : Créé un contexte !

```
cl_int ret;
context = clCreateContext(nullptr, 1, &devs[0],
                          nullptr, nullptr, &ret);
```

API d'OpenCL : Queue de commande

- Permet de configurer une queue de commande qui : exécute les noyaux dans l'ordre d'appel ou dans un ordre dicté uniquement par la dépendance des données;
- `cl_command_queue clCreateCommandQueue( cl_context context, cl_device_id device, cl_command_queue_properties properties, cl_int *errcode_ret)` :

```
cl_command_queue command_queue;
command_queue = clCreateCommandQueue(context, device_id, 0, &ret);
```

API d'OPENCL : Allocation mémoire

- Se fait au travers des objets de type `cl_mem`
- Permet de réserver et de copier ou de réserver seulement.
- `cl_mem clCreateBuffer ( cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret)`

```
cl_mem u_dev, v_dev, w_dev;
u_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,
                        dim * sizeof(float), u, &ret);
v_dev = clCreateBuffer(context, CL_MEM_READ_ONLY,
                        dim * sizeof(float), v, &ret);
w_dev = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
                        dim * sizeof(float), nullptr, &ret);
```

Création d'un noyau de calcul

Code source : `vecadd.cl`

```
kernel void vect_add_cl(global const float* u,
                        global const float* v,
                        global       float* w,
                        const int    dim )
{
    const ind = get_global_id(0);
    if ( ind < dim )
      w[ind] = u[ind] + v[ind];
    }
```

Création d'un noyau de calcul (suite)

Création d'un programme composé de noyaux :

```
FILE *fp;
char fileName[] = "./vecadd.cl";
char *source_str;
size_t source_size;

/* Load the source code containing the kernel*/
fp = fopen(fileName, "r");
source_str = (char*)malloc(MAX_SOURCE_SIZE);
source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
fclose(fp);

cl_program program =
   clCreateProgramWithSource(context, 1,
                             (const char **)&source_str,
                             (const size_t *)&source_size, &ret);

ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);

kernel = clCreateKernel(program, "vecadd", &ret);
```

Exécution du noyau et lecture du résultat

Passage des arguments

```
ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&u_dev);
ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&v_dev);
ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&w_dev);
ret = clSetKernelArg(kernel, 3, sizeof(int), (void *)&dim);
```

Exécution du noyau

```
ret = clEnqueueTask(command_queue, kernel, 0, NULL,NULL);
```

Recopie du résultat en mémoire vive

```
ret = clEnqueueReadBuffer(command_queue, w_dev, CL_TRUE, 0,
                          dim * sizeof(float),w, 0, NULL, NULL);
```

Finalisation et libération des ressources

Finalisation

```
clFlush(command_queue);
clFinish(command_queue);
```

Libération

```
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseMemObject(u_dev);
clReleaseMemObject(v_dev);
clReleaseMemObject(w_dev);
clReleaseCommandQueue(command_queue);
clReleaseContext(context);
```

Pourquoi OpenACC ?

Naissance d'OpenACC

- En 2012, le comité de standardisation d'OpenMP veut étendre le langage OpenMP pour gérer les GPGPUs;
- Difficultés de trouver un consensus parmi tous les intervenants du comité;
- Cray, CAPS, NVidia et PGI décident en attendant que le consensus soit trouvé de créer un autre standard de programmation OpenACC pour gérer les GPGPUs "à la OpenMP".

Pour

- Non intrusif : permet de rapidement porter du code sur GPGPU;
- Permet d'utiliser des plateformes Nvidia mais aussi ATI;
- Simplicité d'utilisation d'OpenACC : permet d'obtenir une bonne accélération à moindre coût;

Contre

- Ne permet pas des performances optimales comme Cuda;
- Peu de compilateur le supportent : les compilateurs PGI (gratuits pour usage non commercial) et gnu c/c++ à partir de la version 6.1 (encore au stage d'ébauche !)

Exemple de code

```
#include <stdlib.h>
#include <stdio.h>
void saxpy(long n, float a, float *x, float * y) {
#pragma acc parallel loop
  for (long i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}
int main(int argc, char **argv) {
  float sum;
  long N = 1000000000; // 1 billion floats
  if (argc > 1) N = atoi(argv[1]);
  float *x = (float*)malloc(N * sizeof(float));
  float *y = (float*)malloc(N * sizeof(float));
  for (long i = 0; i < N; ++i) {
    x[i] = 2.0f; y[i] = 1.0f;
  }
  saxpy(N, 3.0f, x, y);
  sum = 0.0f;
  for ( long i = 0; i < N; ++i ) sum += y[i];
  free(x); free(y);
  printf("sum = %f\n",sum);
  return 0;
}
```

GPGPU avec OpenMP

Historique

- Support des GPGPUs par OpenMP depuis la version 4.0 de la norme;
- Pour l'instant, encore très limité : les compilateurs Intel ne supportent que les Xeon Phi, Cray ne propose que OpenACC.
- OpenMP 4.0 pour GPU encore au stade rudimentaire pour GCC
- Valable pour Clang et compilateurs PGIs

Pour

- Approche unifiée avec le reste d'OpenMP;
- Même simplicité que OpenACC;
- Évite de mélanger plusieurs directives de compilation !

Contre

- Ne permet pas d'avoir des performances optimales;
- Peu de compilateur supportent OpenMP 4.0 avec GPU aujourd'hui !

Exemple de code OpenMP pour GPGPU

```
#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: %s \n", argv[0]); return 0;
    }
    int n = atoi(argv[1]);
    double* x = (double*)malloc(sizeof(double) * n);
    double* y = (double*)malloc(sizeof(double) * n);
    double idrandmax = 1.0 / RAND_MAX, a = idrandmax * rand();
    for (int i = 0; i < n; i++) {
        x[i] = idrandmax * rand(); y[i] = idrandmax * rand();
    }
#   pragma omp target data map(tofrom: x[0:n],y[0:n])
    {
        #pragma omp target
        #pragma omp for
        for (int i = 0; i < n; i++)
            y[i] += a * x[i];
    }
    double avg = 0.0, min = y[0], max = y[0];
    for (int i = 0; i < n; i++) {
        avg += y[i];
        if (y[i] > max) max = y[i]; if (y[i] < min) min = y[i];
    }
    printf("min = %f , max = %f , avg = %f\n", min, max, avg / n);
    free(x); free(y);
    return 0;
}
```