Systèmes concurrents et parallélisme

Chapitre 3/4/5/6 - Parallélisme/Étude de cas

Gabriel Girard

2012

Chapitre 3/4/5/6 - Cuda/OpenCl

1. GPGPU
   - GPU
   - CUDA
   - OpenCL

Chapitre 3/4/5/6 - Cuda/OpenCl

1. GPGPU
   - GPU
   - CUDA
   - OpenCL

GPU

- GPGPU : General Purpose Computation on GPU
- Le + puissant par $
- CPU : optimisé pour code séquentiel (cache, prédiction, ...)
- GPU : optimisé pour les calculs arithmétiques parallèles
- GPU : fournit moins de précision et moins d'opérations
- GPU : modèle de calcul original

GPU

- Début 2006 avec CUDA (NVidia) et Stream (ATI-AMD)
- OpenCl est un API unifié pour tout type de périphériques
- Inspiré de BrookGPU (Stanford) et Sh (Waterloo)

GPGPU : Deux concepts importants

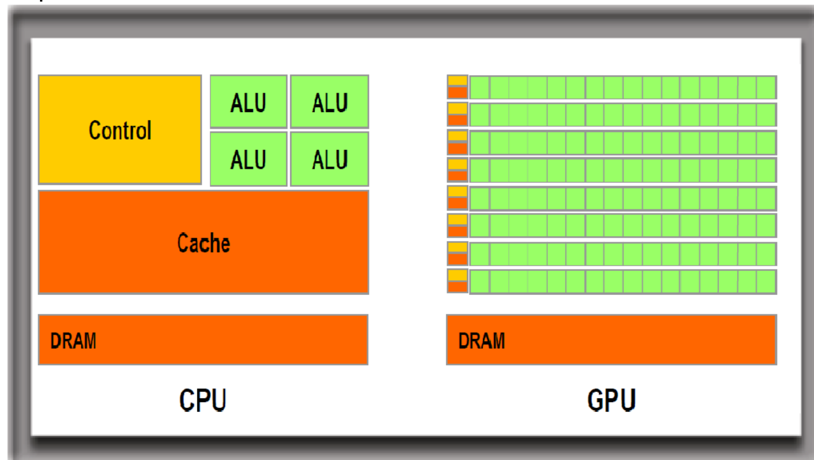
- Stream processing (SIMD ou data parallelism)
- Les kernels appliqués sur chaque élément de la Stream

GPGPU

- Les GPU traitent les éléments de façon indépendante
- Pas de données partagées ou statiques
- On lit les données en entrée et écrit les résultats
- Pas de mémoire en lecture/écriture

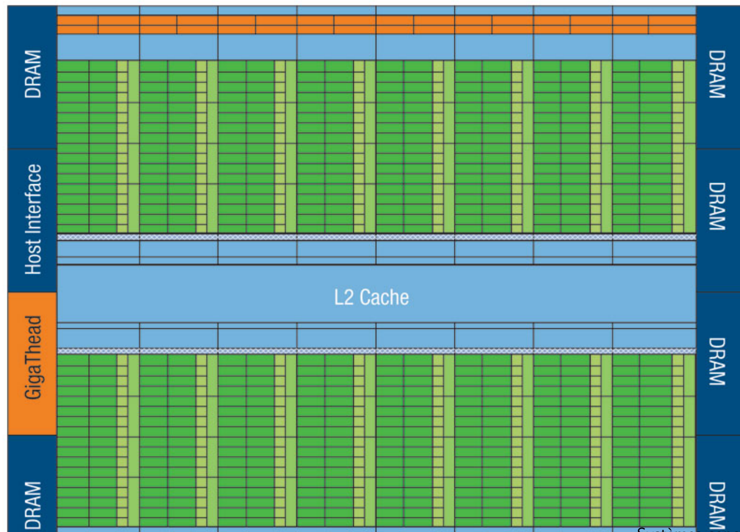
GPGPU

Les GPU ont moins de cache et une unité de contrôle moins importante.



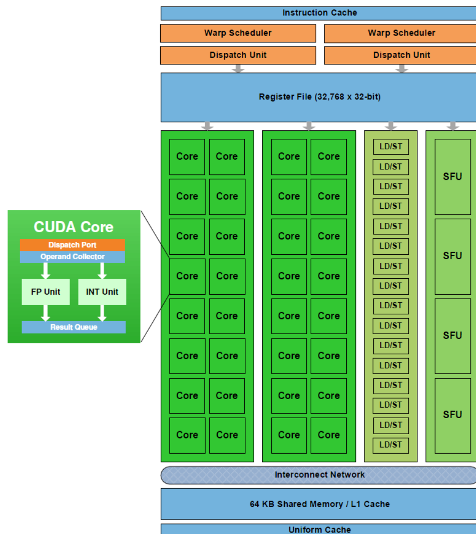
GPGPU - Fermi (Nvidia)

Jusqu'à 16 Streaming Multiprocessors (SM) de 32 coeurs



GPGPU - Fermi (Nvudia)

Un SM = 32 coeurs avec de mémoire partagée et registres



GPGPU - Hiérarchie de mémoire

- Sur la puce
  - registres (32 768 par SM)
  - mémoire partagée (64k par SM)
- Hors de la puce
  - mémoire globale (lente)
  - mémoire de texture, de constantes et locale

CUDA : Terminologie

- Hôte $\longrightarrow$ CPU
- Périphérique $\longrightarrow$ GPU
- Kernel $\longrightarrow$ code à exécuter sur le périphérique
- Grille $\longrightarrow$ tâches à exécuter, divisée en blocs
- Blocs $\longrightarrow$ contiennent plusieurs fils (threads)

CUDA : Concept

- Le programme contient le code pour l'hôte et le périphérique
- Allocation de mémoire sur le GPU et transfert du CPU
- Lancement du «kernel» sur le périphérique par l'hôte en précisant le contexte d'exécution

CUDA - Kernel

- Fonction exécutée par le GPU
- Ne s'appelle pas de la même manière qu'une fonction
- 3 types :
  - __global__ : appelé par CPU et exécuté par GPU
  - __device__ : appelé et exécuté par GPU
  - __host__ : appelé et exécuté par CPU (défaut)

CUDA - Kernel

- Appel au kernel :
  - kernel << NBlocs, ThreadPerBlock>> (arguments)
- NBlocs : Nombre de blocs
- ThreadPerBlock : nombre de fils à exécuter simultanément

CUDA - Kernel

- Variables implicites :
  - blocidx : index du bloc dans la grille (x,y,z)
  - threadidx : index du thread dans le bloc
  - blockdim : nombre de fils dans le bloc

CUDA - Exemple

```
_global__ void vecAdd(float * A, float * B, float * C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    vecAdd <<1, N>> (A, B, C);
      //     |-> vecteurs additionnés une seule fois
      //        |-> nombre de composante des vecteurs
}
```

CUDA - Exemple

```
_global__ void vecAdd(float * A, float * B, float * C)
{
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    const int threadsPerBlock = 256;
    // int nBlocks = N/threadsPerBlock + (N%threadsPerBlock == 0 ? 0:1)
    int nBlocks = (N + threadsPerBlock - 1) / threadsPerBlock;
    vecAdd <<nBlocks, nThreadsPerBlocks>> (A, B, C);
}
```

CUDA - Kernel

- On doit allouer la mémoire :
  - cudaMalloc
  - cudaFree
  - cudaMemCpy

CUDA - Exemple

```
int main(void)
{
  float *VH, *VP;
  const int N = 10;
  size_t size = N * sizeof(float);

  VH = (float *)malloc(size);
  cudaMalloc((void **) &VP, size);

  // Initialisation et copie
  for (int i=0; i<N; i++) A[i] = (float)i;
     cudaMemcpy(VP, VH, size, cudaMemcpyHostToDevice);

  // On calcule
  int block_size = 4;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  square_array <<< n_blocks, block_size >>> (a_d, N);

  // Récupère les résultats
  cudaMemcpy(a_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

  // Impression et nettoyage
  for (int i=0; i<N; i++) printf("%d %f\n", i, a_h[i]);
  free(a_h); cudaFree(a_d);
}
```

CUDA - Exemple

```
// Kernel
__global__ void square_array(float *a, int N)
{
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   if (idx<N) a[idx] = a[idx] * a[idx];
}
```

CUDA - Kernel

- Compilation : NVCC

OpenCl

- Standard pour programmation parallèle
- Similaire à CUDA
  - Kernel
  - Application Hôte
- Le code à exécuter sur l'hôte est séparé du code exécuté sur le périphérique
- Le code sur l'hôte doit charger et compiler le code destiné au périphérique

OpenCl

```
__kernel void vecadd(__global const float * A,
                     __global const float * B,
                     __global const float * C)
{
   unsigned int const i = get_global_id(0);

    C[i] = A[i] + B[i];
}
```

OpenCl

- Périphérique
- kernel
- Program
- Command queue
- Context
- Platerforme

OpenCl

- Work item (fil)
  Chacun possède un ID local et un ID global
- Work group
  Chacun possède un ID et une taille

OpenCL

- Fonctions prédéfinies
  - get_global_id
  - get_local_id
  - get_work_dim
  - get_global_size

OpenCL - Mémoire

- Mémoire globale
- Mémoire constante
- Mémoire locale
- Mémoire privée

OpenCL - Structure du programme hôte

1. Obtenir information sur les périphériques
   `clGetPlatformInfo`, `clGetDeviceIDs`, `clGetDeviceInfo`
2. Créer un contexte
   `clCreateContext`, `clCreateContextFromType`, `clGetContextInfo`
3. Créer une file de commandes
   `clCreateCommandQueue`
4. Charger, compiler le programme
   `clCreateProgramFromSource`, `clBuildProgram`, `clCreateKernel`
5. Passer des paramètres
   `clCreateBuffer`, `clSetKernelArg`, `clEnqueueWriteBuffer`
6. Exécuter le programme
   `clEnqueueNDRangeKernel`
7. Récupérer les résultats
   `clEnqueueReadBuffer`