La Programmation GPU

Programmation séquentielle

Traditionnellement, les logiciels sont basés sur le calcul **séquentiel**:

- Un problème est découpé en instructions.
- Ces instructions sont exécutées séquentiellement les unes après les autres.
- Elles sont exécutées par un seul processeur.
- À un instant donné, une seule instruction est exécutée.
- La performance est determinée principalement par la frèquence (Hz) du processeur.

programmation parallèle

La programmation parallèle permet l'utilisation de plusieurs ressources de calcul pour résoudre un problème donné:

- Un problème est découpé en parties qui peuvent être lancées simultanément.
- Chaque partie est découpée en instructions.
- Les instructions de chaque partie sont exécutées en parallèle en utilisant plusieurs processeurs.
- La performance est déterminée par: La fréquence et le nombre de processeurs, Le degré de parallélisation du problème

Architecture

CPU:"Central Processing Unit" :
Plusieurs unités d'exécutions (cœurs)
Plusieurs niveaux de mémoire (registres, L1, L2, L3, RAM)
Plusieurs ports d'exécution dans chaque cœur (ALUs, unités vectorielles)
Unités vectorielles , Exécution de quelques threads (1-4) simultanément Capable d'exploiter le parallelisme au niveau des instructions (micro-op buffer, renumérotation d'instructions, renommer les registres, ...)

Architecture

GPU: "Graphical Processing Unit", unité de calcul spécifique véctorielle consistant à:
Plusieurs unités d'exécutions(symmetric multiprocesseurs (SM)). Plusieurs niveaux de mémoire . Plusieurs unités vectorielles (2-4) larges (16-32 flottants) dans chaque SM. Exécution de milliers de threads simultanément. Grand tableau de registres (65K). Échange de threads très rapide. La plupart du circuit est consacrée aux unités vectorielles. Parallélisation prend l'effort

Programmation GPU

La prog GPU est adaptée au modèle d'exécution (single instruction multiple thread) (SIMT)
Un GPU consiste à plusieurs processeurs appelés "streaming multiprocessors" (SM).
__global__ précise la définition d'un kernel GPU (sinon fonction CPU par défaut)
blockIdx.x est prédefini et donne l'identifiant d'un bloc dans un kernel GPU.
gridDim.x est prédefini et donne le nombre de blocs utilisés dans le kernel GPU en cours d'exécution.
Compilation: nvcc programme.cu -o programme
Exécution: ./programme
Chaque appel de kernel est non-bloquant , mais on peut le rendre bloqant si on veut.
Threads idéntiques (exécutent le même code)
Threads organisés en blocs (de taille 32-1024)
Chaque bloc idéntique s'exécute sur un SM.
Blocs organisés en grilles et répartis sur tous les SMs

Note

Applications du calcul parallèle: Traitement des reseaux neurones. Graphiques (rendering, jeux vidéo, etc.). Simulations en physique.
Le transfert entre CPU et GPU de donnés se fait sur le bus PCI express (32Go/s débit chaque direction pour PCIe4).
Supercalculateur / Cluster : Un ensemble de machines (CPU+GPU) connecté. Connexion par un réseau avec une topologie particulière (anneau, grille, torus, clique, etc.)
omp.h: est la bibliothèque OpenMP qui fournit les fonctions nécessaires (e.g., pour obtenir thid, numth)

code

Listing 1: Hello World en OpenMP

```
1  # include <cstdio>
2  # include "omp.h"
3  int main(int argc, char **argv)
4  {
5  # pragma omp parallel num_threads(3)
6    {
7    int thid = omp_get_thread_num ();
8    int numth = omp_get_num_threads();
9    printf ("Hello from thread %d /% d.\n",
         thid ,OpenMP numth);
10   }
11 return 0 ;
12 }
```

Listing 2: Hello World en CUDA

```
1  #include <cstdio>
2  #include "cuda.h"
3
4  __global__ void helloWorld()
5  {
6    printf("Hello from block %d/%d\n",
7        blockIdx.x, gridDim.x);
8  }
9
10 int main(int argc, char **argv)
11 {
12   helloWorld<<<3, 1>>>();
13   cudaDeviceSynchronize();
14   return 0;
15 }
```

code

Listing 3: Multiplier un tableau en CUDA

```
#include <cstdio>
#include "cuda.h"

#define N 1024
float A[N];
float c = 2.0;

__device__ float dA[N];

__global__ void multiplyArray(int n,
    float c)
{
  int elemParBlock = n / gridDim.x;
  int begin = blockIdx.x * elemParBlock;
  int end;
  if (blockIdx.x < gridDim.x - 1) {
    end = (blockIdx.x + 1) * elemParBlock
        ;
  } else {
    end = n;
  }
  for (int i = begin; i < end; i++) { dA[
      i] *= c; }
}

int main(int argc, char **argv)
{
  // Initialisation
  for (int i = 0; i < N; i++) { A[i] = i;
      }
  // Copier le tableau vers le GPU
  cudaMemcpyToSymbol(dA, A, N * sizeof(
      float), 0,
      cudaMemcpyHostToDevice);
  multiplyArray<<<4, 1>>>(N, c);
  // Recopier le tableau multiplie vers
      le CPU
  cudaMemcpyFromSymbol(A, dA, N * sizeof(
      float), 0,
      cudaMemcpyDeviceToHost);
  printf("%lf\n", A[2]);
  return 0;
}
```

code

Listing 4: Multiplier un tableau en CUDA

```
#include <cstdio>
#include <iostream>
#include "cuda.h"
#define N 513
#define BSXY 32
float A[N][N], B[N][N], C[N][N];
__device__ float dA[N][N], dB[N][N], dC[N
    ][N];
__global__ void multiplyMatrixGPUByBlocks
    (int n)
{int i = blockIdx.x; int j = blockIdx.y;
  float c = 0.0;
  for (int k = 0; k < n; k++) { c += dA[i
      ][k] * dB[k][j]; }
  dC[i][j] = c;}
//calculer avec blockDim.x threads par
    bloc.
__global__ void
    multiplyMatrixGPUByBlocksThreads1D(
    int n)
{int i = blockIdx.x;
  int j = threadIdx.x + blockIdx.y *
      blockDim.x;
  float c = 0.0;
  for (int k = 0; k < n; k++) { c += dA[i
      ][k] * dB[k][j]; }
  dC[i][j] = c;}
// Creer un bloc pour le calcul de
    blockDim.x elements de C, calculer
    avec blockDim.x threads par bloc.
__global__ void multiplyMatrixGPUBy
    BlocksThreads1DNonMultiple(int n)
{int i = blockIdx.x;element C[ i ] [ j ]
    ,  c a l c u l e r  avec 1
  int j = threadIdx.x + blockIdx.y *
      blockDim.x;
  if (j < n) {
    float c = 0.0;
    for (int k = 0; k < n; k++) { c += dA
        [i][k] * dB[k][j]; }
    dC[i][j] = c;}}
// Creer un bloc pour le calcul de
    blockDim.x * blockDim.y elements de C
    , calculer avec blockDim.x * blockDim
    .y threads par bloc.
__global__ void
    multiplyMatrixGPUByBlocksThreads2D(
    int n)
{int i = threadIdx.y + blockIdx.x *
    blockDielementsm.y;
  int j = threadIdx.x + blockIdx.y *
      blockDim.x;
  float c = 0.0;
  for (int k = 0; k < n; k++) { c += dA[i
      ][k] * dB[k][j]; }
  dC[i][j] = c;}
```

code

Listing 5: Multiplier un tableau en CUDA

```
__global__ void multiplyMatrixGPU
    ByBlocksThreads2D(int n)
{int i = threadIdx.y + blockIdx.x *
    blockDim.y;
  int j = threadIdx.x + blockIdx.y *
      blockDim.x;
  float c = 0.0;
  for (int k = 0; k < n; k++) { c += dA[i
      ][k] * dB[k][j]; }
  dC[i][j] = c;}
__global__ void multiplyMatrixGPUBy
    BlocksThreads2DNonMultiple(int n)
{ int i = threadIdx.y + blockIdx.x *
    blockDim.y;
  int j = threadIdx.x + blockIdx.y *
      blockDim.x;
  if (i < n && j < n) {float c = 0.0;
    for (int k = 0; k < n; k++) { c += dA
        [i][k] * dB[k][j]; }
    dC[i][j] = c; }}
__global__ void
    multiplyMatrixGPUByBlocksThreads 2
    DNonMultipleSharedMemory(int n)
{ int row = threadIdx.y + blockIdx.x *
    blockDim.y;
  int col = threadIdx.x + blockIdx.y *
      blockDim.x;
  __shared__ float shA[BSXY][BSXY];
  __shared__ float shB[BSXY][BSXY];
  float c = 0.0;
  const int nsteps = (n - 1) / BSXY + 1;
  for (int step = 0; step < nsteps; step
      ++) {
    int offset = step * BSXY;
    int stepRowB = offset + threadIdx.y;
    int stepColA = offset + threadIdx.x;
    if (row < n && stepColA < n) {
      shA[threadIdx.y][threadIdx.x] = dA[
          row][stepColA];
    } else {
      shA[threadIdx.y][threadIdx.x] =
          0.0;}
    if (stepRowB < n && col < n) {
      shB[threadIdx.y][threadIdx.x] = dB[
          stepRowB][col];
    } else {
      shB[threadIdx.y][threadIdx.x] =
          0.0;}
    __syncthreads();
    for (int k = 0; k < BSXY; k++) {
      c += shA[threadIdx.y][k] * shB[k][
          threadIdx.x];}
    __syncthreads();}
  if (row < n && col < n) { dC[row][col]
      = c; }}
```

code

Listing 6: Multiplier un tableau en CUDA

```
void multiplyMatrixCPU()
{for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      C[i][j] = 0.0f;
      for (int k = 0; k < N; k++) {
        C[i][j] += A[i][k] * B[k][j];}}}}
void verifyResults()
{for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      float c = 0.0f;
      for (int k = 0; k < N; k++) {
        c += A[i][k] * B[k][j];}
      if (std::abs(C[i][j] - c) > 1e-6) {
        std::cout << "Multiplication is
            incorrect for the element C["
             << i << "][" << j << "]" <<
            std::endl;
        return;}}}
  std::cout << "Multiplication is correct
      !" << std::endl;
}
int main(int argc, char **argv)
{for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      A[i][j] = i + j;
      B[i][j] = i - j;}}
  cudaMemcpyToSymbol(dA, A, N * N *
      sizeof(float), 0,
      cudaMemcpyHostToDevice);
  cudaMemcpyToSymbol(dB, B, N * N *
      sizeof(float), 0,
      cudaMemcpyHostToDevice);
  dim3 dimGrid;
  dimGrid.x = (N - 1) / 32 + 1;
  dimGrid.y = (N - 1) / 32 + 1;
  dimGrid.z = 1;dim3 dimBlock;
  dimBlock.x = 32;dimBlock.y = 32;
  dimBlock.z = 1;
  // multiplyMatrixGPUByBlocks<<<dimGrid,
       dimBlock>>>(N);
  // multiplyMatrixGPUByBlocksThreads1D
      <<<dimGrid, dimBlock>>>(N);
  // multiplyMatrixGPUByBlocksThreads2D
      <<<dimGrid, dimBlock>>>(N);
  // Recopier le tableau dC vers le CPU
  cudaMemcpyFromSymbol(C, dC, N * N *
      sizeof(float), 0,
      cudaMemcpyDeviceToHost);
  // multiplyMatrixCPU();
  verifyResults();
  return 0;
}
```