Parallélisme Cuda - TD1

Elana Courtines

2022-09-12

1 Travaux Dirigés de Programmation CUDA

Exercice 1.
Fonction CPU :

```
#define BLOCK_SIZE 1024

void vecADD(float *A, float *B, float *C, int n){
    // determining the amount of data to declare
    int bytes = n * sizeof(float);
    int num_block = (n - 1 + BLOCK_SIZE) / BLOCK_SIZE;

    // create blocks and grid dimension
    dim3 grid_size = (num_block, 1, 1);
    dim3 bsize = (BLOCK_SIZE, 1, 1);

    // create the new variables
    int *dA; int *dB; int *dC;

    // allocate the required memory space
    cudaMalloc((void **)&dA, bytes);
    cudaMalloc((void **)&dB, bytes);
    cudaMalloc((void **)&dC, bytes);

    // copy the given data to the variables used by the GPU
    cudaMemcpy(dA, A, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(dB, B, bytes, cudaMemcpyHostToDevice);

    // launch the GPU function
    vecAddKernel<<<grid_size,bsize>>>(dA,dB,dC,n);

    // copy the result to the given pointer
    cudaMemcpy(C, dC, bytes, cudaMemcpyDeviceToHost);
    cudaFree(dA); cudaFree(dB); cudaFree(dC);
}
```

Fonction GPU :

```
__global__
void vecAddKernel(float *dA, float *dB, float *dC, int n){
    int indice = blockIdx.x * blockDim.x + threadIdx.x;
    if (indice < n)
        dC[indice] = dA[indice] + dB[indice];
}
```

Exercice 2 :

Fonction CPU (donnée) :

```
#define BLUR_SIZE 3
#define BLOCK_SIZE 32

void blur(unsigned char *in, unsigned char *out, int width, int height){
    int numbw = (width + BLOCK_SIZE - 1) / BLOCK_SIZE;
    int numbh = (height + BLOCK_SIZE - 1) / BLOCK_SIZE;

    int bytes = width * height * sizeof(unsigned char);
    dim3 grid_size = (numbw, numbh, 1);
    dim3 bsize = (BLOCK_SIZE, BLOCK_SIZE, 1);

    unsigned char *din;
    unsigned char *dout;

    cudaMalloc((void **)&din, bytes);
    cudaMalloc((void **)&dout, bytes);

    cudaMemcpy(din, in, bytes, cudaMemcpyHostToDevice);

    blurKernel<<<gdim,bdim>>>(din, dout, width, height);

    cudaMemcpy(out, dout, bytes, cudaMemcpyDeviceToHost);
    cudaFree(din); cudaFree(dout);
}
```

Fonction GPU :

```
__global__
void blurKernel(unsigned char *din, unsigned char *dout, int width, int height){
    int lig = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if ((lig<height) && (col<width)){   // the pixel must be wihtin the frame
        int res = 0;                    // temporary res for the sum
        int nb = 0;                     // temporary variable to count the amount of pixels summed
        // iterate through the width and the height of the Blur Frame
        for (int currLig = lig-BLUR_SIZE; currLig < lig+BLUR_SIZE+1; currLig++){
            for (int currCol = col-BLUR_SIZE; currCol < col+BLUR_SIZE+1; currCol++){
                if ((currLig >= 0) && (currLig < height) && (currCol >= 0) && (currCol < width)){
                    res += din[currLig*width+currCol];
                    nb++;
                }
            }
        }
        dout[lig*width + col] = (unsigned char) (res / nb);
    }
}
```

Parallélisme Cuda - TD2

Elana Courtines

2022-09-19

Exercice 3 :
Fonction CPU (donnée) :

```
void reduce(float *vec, float *sum, int size){
    float *d_vec;
    int bytes = size * sizeof(float);

    cudaMalloc((void **)&d_vec, bytes);

    cudaMemcpy(d_vec, vec, bytes, cudaMemcpyHostToDevice);

    kreduce<<<1,size>>>(d_vec, size);

    cudaMemcpy(sum, d_vec, bytes, cudaMemcpyDeviceToHost);
    cudaFree(d_vec);
}
```

Fonction GPU v1:

```
__global__
void kreduce(float *d_vec, int size){
    unsigned int tid = threadIdx.x;
    if (tid < size) {
        for(int offset = 1; offset<size; offset=offset*2) {
            if ( tid % 2*offset == 0 ) {
                d_vec[tid]+= d_vec[tid+offset];
            }
            __syncThreads();
        }
    }
}
```

Analyse de divergence pour la fonction GPU v1:

itération	#threads	#warps
1	512	32
2	256	32
3	128	32
4	64	32
5	32	32
6	16	16
7	8	8
8	4	4
9	2	2
10	1	1

Fonction GPU v2:

```
__global__
void kreducev2(float *d_vec, int size){
    unsigned int tid = threadIdx.x;
    if (tid < size) {
        for(int offset = size/2; offset>0; offset=offset/2) {
            if ( tid < offset ) {
                d_vec[tid]+= d_vec[tid+offset];
            }
            __syncThreads();
        }
    }
}
```

Analyse de divergence pour la fonction GPU v2 :

itération (offset)	#threads	#warps
1 (512)	512	16
2 (256)	256	8
3 (128)	128	4
4 (64)	64	2
5 (32)	32	1
6 (16)	16	1
7 (8)	8	1
8 (4)	4	1
9 (2)	2	1
10 (1)	1	1

Exercice 4 :

Fonction CPU :

```
#define BLOCK_SIZE 1024
#define RADIUS 3

void convolution(float *in, float *out, float *weight, int size){
    int num_block = (n - 1 + BLOCK_SIZE) / BLOCK_SIZE;
    int bytes = size * sizeof(float);
    float *din;
    float *dout;
    cudaMalloc((void **)&din, bytes);
    cudaMalloc((void **)&dout, bytes);
    cudaMemcpy(din, in, bytes, cudaMemcpyHostToDevice);

    __constant__ float dweight[2*RADIUS+1];
    cudaMemcpyToSymbol(dweight, weight, (2*RADIUS+1)*sizeof(float));

    convKernel<<<num_block,BLOCK_SIZE>>>(din, dout, size);

    cudaMemcpy(out, dout, bytes, cudaMemcpyDeviceToHost);
    cudaFree(din); cudaFree(dout);
}
```

Fonction GPU :

```
__global__
void convKernel(float *in, float *out, int size){
    int gid = blockIdx.x * blockDim.x + threadIdx.x
    int tid = threadIdx.x;
    __shared__ float *sh_in[BLOCK_SIZE + 2*RADIUS];
    if ( gid < size ){ // start by copying the aligned ones
        // copy in the cell shifted by 1 #Radius
        sh_in[tid+RADIUS] = in[gid];
        // one of the #Radius first threads of the block
        if (tid < RADIUS) {
            if ( gid >= RADIUS ){
                // copy in the cell shifted by 0 #Radius
                sh_in[tid] = in[gid - RADIUS]
            } else {
                sh_in[tid] = 0;
            }
        }
        // one of the #Radius last threads of the block
        if (tid > BLOCK_SIZE - RADIUS) {
            if ( gid + BLOCK_SIZE < size ){
                // copy in the cell shifted by 2 #Radius
                sh_in[tid + RADIUS*2] = in[gid+RADIUS];
            } else {
                sh_in[tid + RADIUS*2] = 0;
            }
        }
    }
    // The section below has not been corrected in TD
    __syncthreads();
    int res=0;
    for (int i=0; i<2*RADIUS+1; i++)[
        res = res + sh_in[tid-RADIUS+i] * dweight[i];
    ]
    out[tid]=res;
}
```

Parallélisme Cuda - TD3

Elana Courtines

2022-09-26

Exercice 5 :

Fonction CPU :

```
void fusion(int *vect1, int *vect2, int size){
    int bytes = size * sizeof(int);
    int *d_vin1;
    int *d_vin2;
    int *d_vout;

    cudaMalloc((void **)&d_vin1, bytes);
    cudaMalloc((void **)&d_vin2, bytes);
    cudaMalloc((void **)&d_vout, 2*bytes);

    cudaMemcpy(d_vin1, in, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_vin2, in, bytes, cudaMemcpyHostToDevice);
    cudaMemcpy(d_vout, in, bytes, cudaMemcpyHostToDevice);

    fusionKernel<<<1,size>>>(d_vin1, d_vin2, d_vout, size);

    cudaMemcpy(out, d_vout, bytes, cudaMemcpyDeviceToHost);
    cudaFree(d_vin1); cudaFree(d_vin2); cudaFree(d_vout);
}
```

Fonction GPU :

```
__global__
void fusionKernel(int *vect1, int *vect2, int *res, int size){
    int tid = threadIdx.x;
    __shared__ int *sh_vin1[BLOCK_SIZE]; // it is forbidden to use variables to
    __shared__ int *sh_vin2[BLOCK_SIZE]; // declare in shared memory
    int val1;
    int val2;
    int pos1;
    int pos2;
    if ( tid < size ){
        sh_vin1[tid] = vect1[tid];
        sh_vin2[tid] = vect2[tid];
    }
    __syncthreads();
    if ( tid < size ){
        val1 = sh_vin1[tid];
        val2 = sh_vin2[tid];
        pos1 = tid + search(val1, sh_vin2, size)
        pos2 = tid + search(val2, sh_vin1, size)
        res[pos1] = val1;
        res[pos2] = val2;
    }
}
```

Exercice 6 :

Fonction GPU :

```
__global__
void computeKernel(void *d_bodies, void *d_accel){
    float4 *bodies = (float4*) d_bodies;
    float3 *accel = (float3*) d_aceel;

    int tid = threadIdx.x;
    int gidx = blockIdx.x * blockDim.x + tid;

    __shared__ float4 *sh_bodies[TILE_SIZE];

    float4 mybody = bodies[gidx];
    float3 acc = {0f, 0f, 0f};


    for (int tile=0; tile<NB_BODIES/TILE_SIZE; tile++){
        int idx = tile * blockDim.x + tid;
        sh_bodies[tid] = bodies[idx];
        __syncthreads();
        for (int k=0; k<TILE_SIZE; k++){
            interaction(&mybody, &(sh_bodies[k]), &acc); //does the sum for us
        }
        __syncthreads();
    }
    accel[gidx] = acc;
}
```