TD cuda

### Exercice 1.

void vecADD(float \*A, float \*B, float \*C, int n) {

int bsize = 1024;

int gsize = ((n + bsize -1) /bsize);

int vecSize= n\*sizeof(float);

float\* dA;

float\* dB;

float\* dC;

cudaMalloc((void \*\*)&dA, vecSize);

cudaMemcpy(dA, A, vecSize, cudaMemcpyHostToDevice);

cudaMalloc((void \*\*)&dB, vecSize);

cudaMemcpy(dB, B, vecSize, cudaMemcpyHostToDevice);

cudaMalloc((void \*\*)&dC, vecSize);

vecADDKernel<<<gsize, bsize>>>(dA , dB, dC, n);

cudaMemcpy(C, dC, vecSize, cudaMemcpyDeviceToHost);

cudaFree(dA) ; cudaFree(dB) ; cudaFree(dC) ;

}

\_\_global\_\_

void vecADDKernel (float \*A, float \*B, float \*C, int n) {

int index = blockIdx.x \* blockDim.x + threadIdx.x;

float AValue;

float BValue;

if (index < n){

AValue = A[index];

BValue = B[index];

C[index] = AValue + BValue;

}

}

### Exercice 2.

# define BLURSIZE 3

# define BSIZE 32

void blur ( unsigned char \*in , unsigned char \* out , int width , int height ) {

int numbw = ( width + BSIZE - 1) / BSIZE ;

int numbh = ( height + BSIZE - 1) / BSIZE ;

dim3 gdim ( numbw , numbh , 1) ;

dim3 bdim ( BSIZE , BSIZE , 1) ;

unsigned char \* din , \* dout ;

int bytes = width \* height \* sizeof ( unsigned char ) ;

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 ) ;

}

\_\_global\_\_

void blurkernel (unsigned char \*din, unsigned char \*dout, int width, int height ) {

int row = blockIdx.y \* blockDim.y + threadIdx.y;

int col = blockIdx.x \* blockDim.x + threadIdx.x;

int blurmin = -BLURSIZE / 2

int blurmax = blurmin + BLURSIZE - 1

int add;

int nb = 0;

int pnt;

if ((row < height ) && (col < width)) {

pnt = row \* width + col ;

nb++

add += din[pnt]

for(int blurpntx = blurmin ; blurpntx < blurmax ; blurpntx ) {

blurrow = (row + blurpntx);

for(int blurpnty = blurmin ; blurpnty < blurmax ; blurpnty ) {

blurcol = (col + blurpnty);

if(blurcol >= 0 && blurcol < width && blurrow >= 0 && blurrow < height) {

blurpnt = blurrow \* width + blurcol ;

nb++

add += din[blurpnt]

}

}

}

din[dout] = add / nb

}

}

### Exercice 3.

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 , sizeof ( float ) , cudaMemcpyDeviceToHost ) ;

cudaFree ( d\_vec ) ;

}

#### Question 1:

\_\_global\_\_

void kreduce ( float \*d\_vec , int size ) {

int index = blockIdx.x \* blockDim.x + threadIdx.x;

for(int offset = 1 ; offset < size ; offset \*= 2) {

if(index % (offset \* 2) == 0 && index < size) {

d\_vec[index] += d\_vec[index + offset];

synchThreads();

}

}

}

| Iteration | # threads | # warps |
| --- | --- | --- |
| 1 | 512 | 32 |
| 2 | 256 | 32 |
| 4 | 128 | 32 |
| 8 | 64 | 32 |
| 16 | 32 | 32 |
| 32 | 16 | 16 |
| 64 | 8 | 8 |
| 128 | 4 | 4 |
| 256 | 2 | 2 |
| 512 | 1 | 1 |

#### Question 2:

\_\_global\_\_

void kreduce ( float \*d\_vec , int size ) {

int index = blockIdx.x \* blockDim.x + threadIdx.x;

for(int offset = size / 2 ; offset > 1 ; offset = offset / 2) {

if(index < offset) {

d\_vec[index] += d\_vec[index + offset];

\_\_synchThreads();

}

}

}

| Iteration | # threads | # warps |
| --- | --- | --- |
| 512 | 512 | 16 |
| 256 | 256 | 8 |
| 128 | 128 | 4 |
| 64 | 64 | 2 |
| 32 | 32 | 1 |
| 16 | 16 | 1 |
| 8 | 8 | 1 |
| 4 | 4 | 1 |
| 2 | 2 | 1 |
| 1 | 1 | 1 |

### 

### Exercice 4.

#define BSIZE 1024

#define RADIUS 3

void convolution(float \*in, float \*out, float \*weight, int size) {

int gsize = ((size + BSIZE -1) / BSIZE );

float \*d\_in, \*d\_out ;

int bytes = size \*sizeof ( float ) ;

cudaMalloc (( void \*\*) & d\_in, bytes ) ;

cudaMalloc (( void \*\*) & d\_out, bytes ) ;

cudaMemcpy ( d\_in, in, bytes , cudaMemcpyHostToDevice ) ;

\_\_constant\_\_ float d\_weight[2 \* RADIUS + 1]

cudaMemcpyToSymbol(d\_weight, weight, 2 \* RADIUS + 1)

kconvolution < < <gsize , BSIZE> > >( d\_in, d\_out, size ) ;

cudaMemcpy (out , d\_out, sizeof ( float ) , cudaMemcpyDeviceToHost ) ;

cudaFree ( d\_in) ;

cudaFree ( d\_out) ;

}

\_\_global\_\_

void kconvolution ( float \*d\_in, float \*d\_out, int size ) {

\_\_shared\_\_ float sh\_in[BSIZE + 2 \* RADIUS] ;

int index = blockIdx.x \* blockDim.x + threadIdx.x;

int tidx = threadIdx.x;

if(index < size) {

sh\_in[RADIUS + tidx] = in[index] ;

if(tidx < RADIUS) {

if(index >= RADIUS) {

sh\_in[tidx] = in[index - RADIUS] ;

} else {

sh\_in[tidx] = 0 ;

}

} else if(index + BSIZE < size) {

sh\_in[tidx + RADIUS \* 2 ] = in[index + RADIUS] ;

} else {

sh\_in[tidx + RADIUS \* 2 ] = 0 ;

}

}

\_\_synchThreads();

}

### 

### Exercice 5.

\_\_device\_\_ int search( int val, int \*vect, int size) ;

\_\_global\_\_

void k\_merge\_sort ( int \*res, int \*in1, int \*in2, int n) {

int tid = threadIdx.x;

\_\_shared\_\_ int sh1[1024], shi2[1024];

if(tid < n) {

sh1[tid] = in1[tid];

sh2[tid] = in2[tid];

}

\_\_syncthreads();

if(tid < n) {

res[tid + search(sh1[tid], sh2, n)] = sh1[tid];

res[tid + search(sh2[tid], sh1, n)] = sh2[tid];

}

}

### 

### Exercice 6.

\_\_device\_\_ float3 interaction ( float4 \* bi , float4 \* bj , float3 \* ai ) {

float3 r;

r.x = bj.x - bi.x ;

r.y = bj.y - bi.y ;

r.z = bj.z - bi.z ;

float dist\_sqr = r.x \* r.x + r.y \* r.y + r.z \* r.z + EPS2 ;

floar dist\_sixth = dist\_sqr + dist\_sqr \* dist\_sqr ;

float inv\_dist\_cube = 1.0 f / sqrtf ( dist\_sixth ) ;

float s = bj . w \* inv\_dist\_cube ;

ai.x += r. x \* s ;

ai.y += r. y \* s ;

ai.z += r. z \* s ;

return ai ;

}

\_\_constant\_\_ P = 32;

\_\_constant\_\_ N = ;

\_\_global\_\_

void k\_compute (void \*d\_bodies, void \*d\_accel) {

\_\_shared\_\_ float4 sh\_bodies[P];

float4 \*bodies = (float4\*) d\_bodies;

float3 \*accel = (float3\*) d\_accel;

int gidx = blockIdx.x \* blockDim.x + theardIdx.x;

float4 m\_bodies = bodies[gidx];

float3 acc = {0f, 0f, 0f};

for(int tile = 0 ; tile < N/P ; tile++) {

int idx = tile \* blockDim.x + theardIdx.x;

sh\_bodies[theardIdx.x] = bodies[idx]

\_\_synctheard()

for(int i = 0 ; i < P ; i++) {

interaction(&mybody, &sh\_bodies[i], &acc);

}

\_\_synctheard();

}

accel[gidx] = acc;

}