고급소프트웨어실습I

11주차 보고서

20171646 박태윤

(실습1)

- 그래픽카드

텍스트이(가) 표시된 사진

자동 생성된 설명

NVIDIA GeForce GTX 1050

- compute capability



- compute capability에 따른 성능 및 스펙

Maximum number of resident grids per device : 32  
Maximum dimensionality of grid of thread blocks : 3

Maximum x-dimension of a grid of thread blocks : 2^31 – 1

Maximum y-, or z-dimension of a grid of thread blocks : 65535

Maximum dimensionality of thread block : 1024

Maximum number of resident blocks per multiprocessor : 32

(실습2)

랜덤한 두 행렬(A, B)를 생성하여 행렬C에 C = 1 / (sinA \* cosB + cosA \* sinB)를 계산하여 CPU, GPU에서 계산하는 시간을 측정하는 프로그램이다. Release모드로 블락의 크기를 나타내는 BLOCK\_SIZE를 32로 설정하여 컴파일 했을 때 다음과 같은 결과를 얻을 수 있다. 데이터 개수는 2^27 = 134217728로 설정하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

블록 크기를 4, 8, 16, 64으로 설정하여 프로그램을 수행시켰을 때는 다음과 같은 결과를 얻을 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

4, 8, 16, 32순으로 블락 사이즈가 커질 때 GPU계산의 시간이 점점 줄어드는 것을 확인할 수 있다. 하지만 블락 사이즈를 64로 설정한 후 프로그램을 실행시켰을 때는 제대로 계산을 수행하지 못하는 것을 확인할 수 있다. 33으로 설정을 하고 프로그램을 실행시켜도 제대로 계산을 못하는 것을 알 수 있는데, 이유는 CUDA에서 maximum으로 잡고 있는 dimension of a block이 1024이기 때문에,

dim3 dimBlock(BLOCK\_SIZE, BLOCK\_SIZE);

로 블락의 차원을 BLOCK\_SIZE \* BLOCK\_SIZE로 잡는 해당 실습 프로그램에서 BLOCK\_SIZE가 33인 경우 33\*33 = 1089이라 BLOCK\_SIZE가 32보다 커지면 제대로 계산을 수행할 수 없는 것이다. BLOCK\_SIZE <= 32인 경우 블락의 크기가 늘어날 수 록 분할되는 작업이 많아지기 때문에 블락 크기가 커질수록 실행속도가 빨라지는 것을 알 수 있다.

(실습3)

두 행렬의 곱을 계산하는 프로그램이다. 역시 CPU와 GPU에서 계산하는 시간을 구분해 측정하며 Release로 컴파일 하였다. block의 사이즈를 32, element size를 2^22(4194304)로 설정하여 프로그램을 구동시켰을 때 다음과 같은 결과를 얻을 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

이 역시 GPU를 이용하여 계산하는 것이 더 빠른 것을 확인할 수 있으며 블록 크기를 4, 8, 16으로 설정하여 실행했을 때 다음과 같은 결과를 얻을 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

이 역시 블록의 사이즈를 늘릴수록 GPU계산에서의 실행 시간이 짧아지는 것을 확인할 수 있다.

(과제1)

이차방정식의 근을 구하는 프로그램이다. 이 역시 CPU와 GPU를 이용하여 계산을 진행하였다. 주요 코드는 다음과 같이 작성하였다.

void find\_roots\_CPU(float\* A, float\* B, float\* C, float\* X0, float\* X1, float\* FX0, float\* FX1, int n) {

int i;

float a, b, c, d, x0, x1, tmp;

for (i = 0; i < n; i++) {

a = A[i]; b = B[i]; c = C[i];

d = sqrtf(b \* b - 4.0f \* a \* c);

tmp = 1.0f / (2.0f \* a);

X0[i] = x0 = (-b - d) \* tmp;

X1[i] = x1 = (-b + d) \* tmp;

FX0[i] = (a \* x0 + b) \* x0 + c;

FX1[i] = (a \* x1 + b) \* x1 + c;

}

}

CPU에서 계산을 진행할 때 사용하는 함수이다. float형 변수d는 판별식을 나타내며 두 근은 각각 X0, X1배열로 표현하였고 , 을 나타내게 하여 X1이 항상 X0보다 큰 근을 나타낼 수 있도록 구현하였다. FX0와 FX1은 해당 이차방정식에 구한 X0, X1근을 각각 대입한 값을 나타낸다.

\_\_global\_\_ void find\_roots\_Kernel(float\* d\_A, float\* d\_B, float\* d\_C, float\* d\_X0, float\* d\_X1, float\* d\_FX0, float\* d\_FX1)

{

float a, b, c, d, x0, x1, tmp;

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

a = d\_A[idx]; b = d\_B[idx]; c = d\_C[idx];

d = sqrtf(b \* b - 4.0f \* a \* c);

tmp = 1.0f / (2.0f \* a);

d\_X0[idx] = x0 = (-b - d) \* tmp;

d\_X1[idx] = x1 = (-b + d) \* tmp;

d\_FX0[idx] = (a \* x0 + b) \* x0 + c;

d\_FX1[idx] = (a \* x1 + b) \* x1 + c;

}

GPU계산 시에 수행하는 계산 함수이다. d\_X0, d\_X1이 각각 근을 나타내며 역시 d\_X1가 d\_X0보다 항상 큰 근을 나타내도록 구현하였고 d\_FX0과 d\_FX1은 앞서 구한 d\_X0, d\_X1을 다시 식에 대입한 값을 나타낸다.

void init\_bin\_file(int size) {

srand((unsigned)time(NULL));

printf("\*\*\*Binary File init Start!!\n");

FILE\* fp\_a = fopen("A.bin", "wb");

FILE\* fp\_b = fopen("B.bin", "wb");

FILE\* fp\_c = fopen("C.bin", "wb");

fwrite(&size, sizeof(int), 1, fp\_a);

float random\_num;

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

random\_num = (float)(rand() % 10) - 5.0f;

if (random\_num == 0.0f)

random\_num = 1.0f;

fwrite(&random\_num, sizeof(float), 1, fp\_a);

}

fclose(fp\_a);

fwrite(&size, sizeof(int), 1, fp\_b);

random\_num = 0.0f;

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

random\_num = (float)(rand() % 10) + 30.0f;

if (random\_num == 0.0f)

random\_num = 4.0f;

fwrite(&random\_num, sizeof(float), 1, fp\_b);

}

fclose(fp\_b);

fwrite(&size, sizeof(int), 1, fp\_c);

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

random\_num = (float)(rand() % 10) - 5.0f;

if (random\_num == 0.0f)

random\_num = 2.0f;

fwrite(&random\_num, sizeof(float), 1, fp\_c);

}

fclose(fp\_c);

printf("\*\*\*Binary File init End!!\n\n");

}

이차방정식의 랜덤한 계수(a, b, c)를 생성하는 함수이다. 이차방정식이 두 근을 가지려면 가 양수를 나타내면 되는데, 이를 위해 b값을 a, c에 비해 훨씬 크게 설정하여 정상적인 실행이 될 수 있도록 구현하였다.

cudaError\_t find\_roots\_GPU(float\* A, float\* B, float\* C, float\* gpu\_X0, float\* gpu\_X1, float\* gpu\_FX0, float\* gpu\_FX1)

{

CHECK\_TIME\_INIT\_GPU();

cudaError\_t cudaStatus;

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");

goto Error;

}

float\* d\_A, \* d\_B, \* d\_C;

float\* d\_X0, \* d\_X1;

float\* d\_FX0, \* d\_FX1;

dim3 dimBlock(BLOCK\_SIZE, 1);

dim3 dimGrid(N / dimBlock.x, 1);

size\_t size;

size = N \* sizeof(float);

cudaMalloc(&d\_A, size);

cudaMalloc(&d\_B, size);

cudaMalloc(&d\_C, size);

cudaMalloc(&d\_X0, size);

cudaMalloc(&d\_X1, size);

cudaMalloc(&d\_FX0, size);

cudaMalloc(&d\_FX1, size);

cudaMemcpy(d\_A, A, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_B, B, size, cudaMemcpyHostToDevice);

cudaMemcpy(d\_C, C, size, cudaMemcpyHostToDevice);

size = N;

CHECK\_TIME\_START\_GPU();

find\_roots\_Kernel << < dimGrid, dimBlock >> > (d\_A, d\_B, d\_C, d\_X0, d\_X1, d\_FX0, d\_FX1);

CHECK\_TIME\_END\_GPU(device\_time);

size = N \* sizeof(float);

cudaGetLastError();

cudaDeviceSynchronize();

cudaMemcpy(gpu\_X0, d\_X0, size, cudaMemcpyDeviceToHost);

cudaMemcpy(gpu\_X1, d\_X1, size, cudaMemcpyDeviceToHost);

cudaMemcpy(gpu\_FX0, d\_FX0, size, cudaMemcpyDeviceToHost);

cudaMemcpy(gpu\_FX1, d\_FX1, size, cudaMemcpyDeviceToHost);

CHECK\_TIME\_DEST\_GPU();

Error:

cudaFree(d\_A);

cudaFree(d\_B);

cudaFree(d\_C);

cudaFree(d\_X0);

cudaFree(d\_X1);

cudaFree(d\_FX0);

cudaFree(d\_FX1);

return cudaStatus;

}

CUDA에서 GPU를 이용한 계산을 위해 호출이 되는 함수이다. 계수, 근, 근을 대입한 결과 등을 배열에 저장하기 위해 cudaMalloc함수를 이용하여 동적할당을 하였으며 앞서 CPU를 이용한 계산에서 활용한 계수 배열 A, B, C를 각각 d\_A, d\_B, d\_C에 cudaMemcpy를 해주었고 GPU계산을 위해 find\_roots\_Kernel함수를 호출하였다.

데이터의 크기를 2^20(1048576), 블록의 사이즈를 32로 한 뒤 실행을 시키면 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

CPU, GPU로 계산했을 때 걸리는 시간과 2번째 식에 대한 근과 이를 대입한 결과를 각각 출력했다. 두 경우 모두 같은 근을 구한 것을 확인할 수 있으며 꽤 적절한 근을 구한 것을 확인할 수 있다. 또한 GPU를 이용하여 계산한 경우가 역시 CPU를 이용하여 계산했을 때 보다 훨씬 빠르게 프로그램을 수행하는 것을 확인할 수 있다.

이 역시 블락의 크기를 4, 8, 16, 64, 96으로 설정하여 시행했을 때 다음과 같은 결과를 얻을 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

이번에도 역시 블락의 크기를 늘릴수록 GPU를 이용한 계산 속도가 빨라지는 것을 확인할 수 있었다. 32, 64, 96으로 늘렸을 때는 GPU계산속도가 크게 차이가 나지 않는 것을 확인할 수 있다.

(과제2)

피보나치 수열의 값을 계산하는 프로그램이다.

텍스트, 시계이(가) 표시된 사진

자동 생성된 설명

이와 같은 피보나치 수열에 대한 값은

텍스트, 시계, 손목시계이(가) 표시된 사진

자동 생성된 설명

다음과 같은 식으로 계산할 수 있다.

이에 대한 주요 코드는 다음과 같이 작성하였다.

void Fibonacci\_CPU(int \*h\_Fibo\_n, int \*h\_Fibo\_v) {

int i, j;

float sqrt\_5, x\_0, x\_1, tmp\_0, tmp\_1;

sqrt\_5 = sqrtf(5.0f);

x\_0 = (1.0f + sqrt\_5) / 2.0f; x\_1 = (1.0f - sqrt\_5) / 2.0f;

for (i = 0; i < N; i++) {

tmp\_0 = tmp\_1 = 1.0f;

for (j = 0; j < h\_Fibo\_n[i]; j++) {

tmp\_0 \*= x\_0; tmp\_1 \*= x\_1;

}

h\_Fibo\_v[i] = (int)((tmp\_0 - tmp\_1) / sqrt\_5 + 0.5f);

}

}

CPU를 이용한 피보나치 수열의 값을 계산하는 함수이다. x\_0 = , x\_0 = 을 나타내며 이를 h\_Fibo\_n[i]만큼 제곱해 각각 tmp\_0, tmp\_1에 assign하여 피보나치 수열값을 계산해 h\_Fibo\_v배열에 저장하였다.

\_\_global\_\_ void Fibonacci\_Kernel(int \*x, int\* y)

{

/\*Thread Idx\*/

int row;

int col;

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

float sqrt\_5, x\_0, x\_1, tmp\_0, tmp\_1;

int i;

/\*Todo\*/

sqrt\_5 = sqrtf(5.0f);

x\_0 = (1.0f + sqrt\_5) / 2.0f; x\_1 = (1.0f - sqrt\_5) / 2.0f;

tmp\_0 = tmp\_1 = 1.0f;

for (i = 0; i < x[tid]; i++) {

tmp\_0 \*= x\_0; tmp\_1 \*= x\_1;

}

y[tid] = (int)((tmp\_0 - tmp\_1) / sqrt\_5 + 0.5f);

}

GPU를 이용한 피보나치 수열에서 사용되는 함수이다. x\_0와 x\_1은 앞선 CPU계산 함수와 동일하고 계산된 결과를 인자로 받은 y배열에 저장하였다.

cudaError\_t Fibonacci\_GPU(int \*h\_Fibo\_n, int \*h\_Fibo\_v, int \*h\_Fibo\_v\_GPU)

{

int(\*d\_Fibo\_n), (\*d\_Fibo\_v);

size\_t size = N \* sizeof(int);;

dim3 dimBlock(BLOCK\_SIZE, 1);

dim3 dimGrid(N / dimBlock.x, 1);

CHECK\_TIME\_INIT\_GPU();

/\*Todo\*/

cudaError\_t cudaStatus;

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");

goto Error;

}

cudaMalloc(&d\_Fibo\_n, size);

cudaMalloc(&d\_Fibo\_v, size);

cudaMemcpy(d\_Fibo\_n, h\_Fibo\_n, size, cudaMemcpyHostToDevice);

CHECK\_TIME\_START\_GPU();

Fibonacci\_Kernel << < dimGrid, dimBlock >> > (d\_Fibo\_n, d\_Fibo\_v);

CHECK\_TIME\_END\_GPU(device\_time)

cudaGetLastError();

cudaDeviceSynchronize();

cudaMemcpy(h\_Fibo\_v\_GPU, d\_Fibo\_v, size, cudaMemcpyDeviceToHost);

CHECK\_TIME\_DEST\_GPU();

Error:

cudaFree(d\_Fibo\_n);

cudaFree(d\_Fibo\_v);

return cudaStatus;

}

데이터를 나타내는 d\_Fibo\_n과 계산된 결과를 나타내는 d\_Fibo\_v배열을 cudaMalloc으로 동적할당 한 뒤 d\_Fibo\_n배열에 앞선 CPU계산에서 쓰인 데이터 h\_Fibo\_n을 그대로 cudaMemcpy하여 Fibonacci\_Kernel함수를 호출하였다.

Release모드로 블록 사이즈를 32로 설정해 프로그램을 구동시키면 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

CPU와 GPU로 같은 데이터에 대해 피보나치 값을 계산한 뒤 각각 h\_fibonacci\_value[4], h\_fibonacci\_value\_GPU[4]를 출력해보았다. 두 값이 같은 값을 나타내는 것을 확인할 수 있고, GPU로 계산하는 것이 CPU로 계산하는 것 보다 훨씬 빠른 것을 확인할 수 있다.

블락 사이즈를 4, 8, 16, 64, 128으로 설정하여 해당 프로그램을 실행한 결과이다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

역시 블락의 사이즈를 4, 8, 16, 32로 늘릴수록 GPU를 이용한 계산의 속도가 빨라지는 것을 확인할 수 있다. 32, 64, 128로 늘렸을 때는 계산 속도 차이가 그닥 나지 않는 것을 확인할 수 있다.

(과제3)

-100 ~ 100사이의 값을 가지는 X수열에 대해 다음과 같은 수열을 구하는 프로그램이다.



이에 대한 주요 코드들은 다음과 같이 작성하였다.

void Sum\_n\_elements\_CPU(IN int \*p\_ArrayElements, OUT int \*p\_SumOfElements\_CPU, int Nf) {

/\*Todo\*/

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

p\_SumOfElements\_CPU[i] = 0;

for (int k = i - Nf; k <= i + Nf; k++) {

if (k < 0 || k >= N) continue;

p\_SumOfElements\_CPU[i] += p\_ArrayElements[k];

}

}

}

설정된 Nf값에 대해 p\_SumOfElements\_CPU에 데이터를 담고 있는 p\_ArrayElements를 더한 결과를 저장하는데, 이 때 데이터의 인덱스를 나타내는 k가 범위를 벗어나는 경우는 당연히 더하면 안되므로 continue로 예외 처리를 해주었다.

\_\_global\_\_ void Sum\_n\_elements\_Kernel(IN int \*d\_ArrayElements, OUT int \*d\_SumOfArrayElements, int N, int Nf) {

/\*Todo\*/

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

d\_SumOfArrayElements[idx] = 0;

for (int k = idx - Nf; k <= idx + Nf; k++) {

if (k < 0 || k >= N)

continue;

d\_SumOfArrayElements[idx] += d\_ArrayElements[k];

}

}

GPU를 이용한 계산에서 사용되는 함수이다. 이 역시 k에 대해 예외처리를 해주었으며 계산 결과는 d\_SumOfArrayElements라는 인자로 받은 배열에 저장을 해주었다.

cudaError\_t Sum\_n\_elements\_GPU(IN int \*p\_ArrayElements, OUT int \*p\_SumOfElements\_GPU, int Nf) {

cudaError\_t cudaStatus;

/\*Todo\*/

dim3 dimBlock(BLOCK\_SIZE, 1);

dim3 dimGrid(N / dimBlock.x, 1);

int(\*d\_Array), (\*d\_Sum);

size\_t size = N \* sizeof(int);

CHECK\_TIME\_INIT\_GPU();

cudaStatus = cudaSetDevice(0);

if (cudaStatus != cudaSuccess) {

fprintf(stderr, "cudaSetDevice failed! Do you have a CUDA-capable GPU installed?");

goto Error;

}

cudaMalloc(&d\_Array, size);

cudaMalloc(&d\_Sum, size);

cudaMemcpy(d\_Array, p\_ArrayElements, size, cudaMemcpyHostToDevice);

CHECK\_TIME\_START\_GPU()

Sum\_n\_elements\_Kernel << < dimGrid, dimBlock >> > (d\_Array, d\_Sum, N, Nf);

CHECK\_TIME\_END\_GPU(device\_time)

cudaGetLastError();

cudaDeviceSynchronize();

cudaMemcpy(p\_SumOfElements\_GPU, d\_Sum, size, cudaMemcpyDeviceToHost);

CHECK\_TIME\_DEST\_GPU();

Error:

cudaFree(d\_Array);

cudaFree(d\_Sum);

return cudaStatus;

}

데이터를 나타내는 d\_Array와 계산 결과를 나타내는 d\_Sum배열을 cudaMalloc한 뒤 GPU를 이용한 계산 함수인 Sum\_n\_elements\_Kernel함수를 호출하였다.

Release모드로 데이터 사이즈는 2^26, Nf는 2^6, 블락의 사이즈도 2^6으로 설정하여 실행시키면 다음과 같은 결과를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

역시 GPU를 활용한 계산이 CPU보다 빠른 것을 확인할 수 있다.

같은 조건 하에 Nf의 크기를 기존의 64에서 1, 4, 16, 256, 1024으로 각각 변경하여 계산하면 다음과 같은 결과를 얻을 수 있었다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

Nf의 크기가 커질수록 CPU, GPU를 이용한 계산 모두 시간이 늘어나는 것을 확인할 수 있는데 이는 Nf가 커짐에 따라 수열 합을 구하는 계산 과정도 늘어나기 때문이다.

다음은 Nf를 다시 64로 설정하고 블록 사이즈를 8, 16, 32, 64, 128로 설정하여 계산했을 때의 결과이다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명

블락의 사이즈를 8, 16, 32, 64로 늘리면 계산의 분배 처리가 늘어나기 때문에 GPU를 활용한 계산의 수행 속도가 빨라지는 것을 확인할 수 있다. 블락 사이즈가 64에서 128로 늘어나면 계산 속도가 크게 차이 나지 않는 것을 확인할 수 있다.