WYDZIAŁ FIZYKI I ASTRONOMII
UNIWERSYTET WROCŁAWSKI

Projekt CUDA

Paweł Grabiński

4 lutego 2015

1 OBLICZENIA RÓWNOLEGŁE PRZY UŻYCIU TECHNOLOGII NVIDIA CUDA

1.1 TREŚĆ PROBLEMU

Wybrane zadanie: nr 9

Napisz program, który na GPU rozwiąże następujący problem: dany jest ciąg trójek liczb (x_i, y_i, z_i), $i = 0, ..., N-1$, definiujących współrzędne środków gwiazd w przestrzeni oraz N-elementowa tablica v przekazana poprzez wskaźnik do pierwszego elementu. W środku układu współrzędnych znajduje się Bardo Masywny Obiekt i interesują nas energie potencjalne gwiazd w polu grawitacyjnym Obiektu.

- Napisz program, który w elemencie i tablicy v, $i = 0, ..., N-1$, zapisze odwrotność odległości punktu (x_i, y_i, z_i) od środka układu współrzędnych.
- Przetestuj efektywność swojego programu dla kilku wartości N.
- Porównaj wydajność swojego rozwiązania z wydajnością analogicznego programu rozwiązującego ten problem na CPU.

Uwaga: Trójki liczb zmiennopozycyjnych wygodnie zapisuje się w typie float3 lub double3. Możesz też użyć typów float4 i double4 i założyć, że potencjały zapisuje się w czwartej składowej tych struktur zamiast w tablicy v.

1.2 METODOLOGIA ROZWIĄZANIA

Podczas pracy programu zachodzą następujące działania:

1. Na wejściu obliczeń podana musi zostać liczba obiektów N oraz rozmiar obszaru w jakim mogą się znajdować $Size$.
2. Dalej generowane są współrzędne danych obiektów. Liczby generowane są przy pomocy algorytmu Mersenne Twister zaimplementowanego w blibliotece GSL.
3. Zapis danych odbywa się w zmiennych typu float4, co ma zoptymalizować proces przesyłu danych do i z karty graficznej. W trzech pierwszych składowych tej zmiennej zapisujemy współrzędne obiektu, a w czwartej wartość odwrotności odległości.
4. Następnie dane przesyłane są na kartę.

5. Inicjalizowany jest kernel na GPU, który dla każdego elementu wywołuje wątek obliczający odwrotność odległości tego punktu od środka układu i zapisuje go w czwartej składowej przesłanej tablicy.

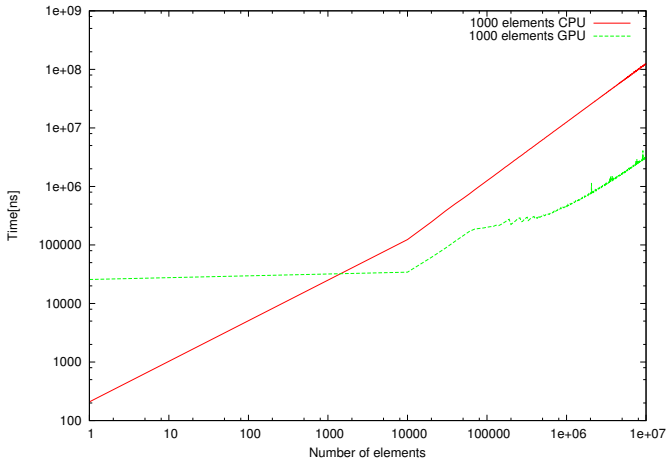
6. Dane są przesyłane z powrotem z karty.

7. Po tym na CPU uruchamiana jest analogiczna funkcja, która szeregowo oblicza odwrotności odległości punktów.

Czas pracy funkcji obliczeniowych mierzony jest przy pomocy funkcji $clock\_gettime(CLOCK\_MONOTONIC, \&timer)$ z dokładnością do $1\,ns$.

1.3 Wyniki

By zbadać jak wydajnym rozwiązaniem jest obliczanie na GPU w porównianiu do CPU zbadaliśmy zależność czasu wykonywania funkcji od ilości elementów układu.



Wykres ma logarytmiczne osie. Widzimy, że czas wykonania zależy od ilości elementów. Początkowo czas CPU jest znacznie mniejszy od czasu wykonania na GPU, jednak wraz ze wzrostem liczby elementów rośnie on zancznie szybciej niż na GPU.
Jak widzimy początkowo koszt inicjalizacji kernela jest znacznie większy od czasu wykonania samych obliczeń.
Istotną uwagą jest fakt, że porównywanie szeregowych obliczeń do równoległych jest nierozsądne. Adekwatnym byłoby porównywanie równoległych obliczeń na CPU do równoległych na GPU.
Isotnym czynnikiem jest także odpowiednia manipulacja wczytywaniem danych. W funkcji $PotentialOnDevice()$ wprowadzenie jednokrotnego wczytywania elementu tablicy typu $float4$ przy 4×10^7 elementów zmieniło czas wykonania z $22\,ms$ do $16\,ms$, co jest istotnym wzrostem efektywności.

1.4 KOD ŹRÓDŁOWY ROZWIĄZANIA

```
#include<stdio.h>
#include<cuda.h>
#include<cuda_runtime.h>
#include<assert.h>
#include<gsl/gsl_rng.h>
#include<time.h>

#define CHECK_CUDA(x)                \
{                                    \
  cudaError_t err = x;               \
  if (err != cudaSuccess)            \
  {                                  \
     printf("!!! CUDA ERROR: \"%s\" at file %s, line %d !!!\n", cudaGetErrorString(err),
     __FILE__, __LINE__);\
     exit(1);                        \
  }                                  \
}

void CreateUniverse(gsl_rng * rng_r, int n_N, int n_Size, float4 *Galaxy_h){
    int i;
    for(i=0;i<n_N;i++){
        Galaxy_h[i].x=gsl_rng_uniform(rng_r)*n_Size-n_Size*0.5;
        Galaxy_h[i].y=gsl_rng_uniform(rng_r)*n_Size-n_Size*0.5;
        Galaxy_h[i].z=gsl_rng_uniform(rng_r)*n_Size-n_Size*0.5;
    }
}

//Kernel
__global__ void PotentialOnDevice(float4 *Galaxy_d, int n_N){
    int idx = blockIdx.y  * gridDim.x * blockDim.x + blockIdx.x*blockDim.x + threadIdx.x
    ;
    if(idx<n_N){
        float4 Star=Galaxy_d[idx];
        Star.w=1.0/sqrt(Star.x*Star.x+Star.y*Star.y+Star.z*Star.z);
        Galaxy_d[idx].w=Star.w;
    }
}

//CPU
void PotentialOnHost(float4 *Galaxy_h, int n_N){
    int i;
    for(i=0;i<n_N;i++)
        Galaxy_h[i].w=1/sqrt(Galaxy_h[i].x*Galaxy_h[i].x+Galaxy_h[i].y*Galaxy_h[i].y+
    Galaxy_h[i].z*Galaxy_h[i].z);
}


int main(int argc, char* argv[]){
    if (argc != 3)
    {
      printf ("error: expected 2 arguments usage\n");
      return -1;
    }

    //Number of objects
    const int n_N=atoi(argv[1]);

    //Size of Galaxy
    const int n_Size=atoi(argv[2]);

    gsl_rng_env_setup();
    gsl_rng *rng_r=gsl_rng_alloc(gsl_rng_mt19937);
```

```
61      float4 *Galaxy_h;      //pointers to host memory
        float4 *Galaxy_d;      //pointers to device memory
63
        //allocate arrays on host
65      Galaxy_h = new float4[n_N];

67      //allocate arrays on device
        CHECK_CUDA(cudaMalloc((void **) &Galaxy_d, sizeof(float4)*n_N));
69
        //Generate objects coord.
71      CreateUniverse(rng_r,n_N,n_Size,Galaxy_h);

73      CHECK_CUDA(cudaMemcpy(Galaxy_d, Galaxy_h, sizeof(float4)*n_N, cudaMemcpyDefault));

75      //KernelSize
        dim3 blockSize = 512;
77      dim3 gridSize (1,1,1);
        const int max_block_size = 65535;
79      int nBlocks = n_N/blockSize.x + (n_N%blockSize.x == 0 ? 0 : 1);
        gridSize.y = 1 + nBlocks/max_block_size;
81      gridSize.x = (nBlocks > max_block_size) ? max_block_size : nBlocks;


83
        //GPU Potential
85      struct timespec startGPU,stopGPU;
        clock_gettime(CLOCK_MONOTONIC, &startGPU);
87      PotentialOnDevice <<< gridSize, blockSize >>> (Galaxy_d, n_N);
        cudaDeviceSynchronize();
89      clock_gettime(CLOCK_MONOTONIC, &stopGPU);
        CHECK_CUDA(cudaMemcpy(Galaxy_h, Galaxy_d, sizeof(float4)*n_N, cudaMemcpyDefault));
91      long int d_TotTimeGPU=(stopGPU.tv_sec-startGPU.tv_sec)*1000000000+stopGPU.tv_nsec-
        startGPU.tv_nsec;

93      //CPU Potential
        struct timespec startCPU,stopCPU;
95      clock_gettime(CLOCK_MONOTONIC, &startCPU);
        PotentialOnHost(Galaxy_h, n_N);
97      clock_gettime(CLOCK_MONOTONIC, &stopCPU);
        long int d_TotTimeCPU=(stopCPU.tv_sec-startCPU.tv_sec)*1000000000+(stopCPU.tv_nsec-
        startCPU.tv_nsec);
99      printf("%d %d %ld %ld\n",n_Size,n_N,d_TotTimeCPU,d_TotTimeGPU);

101     cudaFree(Galaxy_d);
        return 0;
103 }
```

Galaxy.c