Politechnika Poznańska

Wydział Informatyki

Przetwarzanie Równoległe

Programowanie CUDA na NVIDIA GPU

Autorzy:
Adam Szczepański
Mateusz Czajka

Prowadzący:
dr Rafał Walkowiak

24 lutego 2014

Spis treści

1 Informacje o projekcie

1.1 Dane autorów

Mateusz Czajka	106596
Adam Szczepański	106593

1.2 Historia projektu

1. Jest to pierwsza wersja projektu. Dokumentacja elektroniczna została przesłana w dniu 25 lutego 2014.

1.3 Parametry karty graficznej

Nazwa	GeForce GT 240
Typ RAM	DDR3
Frame Buffer Bandwidth (GB/s)	25.6
Graphics Clock (MHz)	575
Processor Clock (MHz)	1400
Memory Clock (MHz)	800
SM Count	12
CUDA Cores	96
MAX_THREADS_PER_BLOCK	512
MAX_BLOCK_DIM_X	512
MAX_BLOCK_DIM_Y	512
MAX_BLOCK_DIM_Z	64
MAX_GRID_DIM_X	65535
MAX_GRID_DIM_Y	65535
MAX_GRID_DIM_Z	1
MAX_SHARED_MEMORY_PER_BLOCK	16384
TOTAL_CONSTANT_MEMORY	65536
WARP_SIZE	32
MAX_REGISTERS_PER_BLOCK	16384
MULTIPROCESSOR_COUNT	12
Compute Capability	1.2
MAX_THREADS_PER_MULTIPROCESSOR	1024

Tablica 1: Parametry wykorzystanej karty graficznej GeForce GT 240.

2 Wprowadzenie

3 Ocena efektywności przetwarzania

3.1 Wersja 1

```
__global__ void MatrixMulKernel_1(const float* Ad, const float* Bd,
    float* Cd, const int WIDTH) {
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  float C_local;

  for (int i=0; i<WIDTH/blockDim.y; i++) {
    for (int j=0; j<WIDTH/blockDim.x; j++) {
      C_local = 0.0f;
      for (int k = 0; k < WIDTH; ++k) {
        float A_d_element = Ad[i*WIDTH*blockDim.y + ty*WIDTH + k];
        float B_d_element = Bd[j*blockDim.y + k*WIDTH + tx];
        C_local += A_d_element * B_d_element;
      }

      Cd[i*WIDTH*blockDim.y + j*blockDim.y + ty*WIDTH + tx] = C_local;
    }
  }
}
```

Listing 1: Mnożenie macierzy kwadratowych na GPU – wersja 1.

3.2 Wersja 2

```
__global__ void MatrixMulKernel_2(float* Ad, float* Bd, float* Cd, int
    WIDTH) {
  int Row = blockIdx.y * blockDim.y + threadIdx.y;
  int Col = blockIdx.x * blockDim.x + threadIdx.x;
  float C_local = 0.0f;

  for (int k = 0; k < WIDTH; ++k)
    C_local += Ad[Row*WIDTH + k] * Bd[k*WIDTH + Col];

  Cd[Row*WIDTH + Col] = C_local;
}
```

Listing 2: Mnożenie macierzy kwadratowych na GPU – wersja 2.

3.3 Wersja 3

```
template <int BLOCK_SIZE> __global__ void
MatrixMulKernel_3(float *C, const float *A, const float *B, const int
    arraySize) {
    int bx = blockIdx.x;
    int by = blockIdx.y;
```

```
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int aBegin = arraySize * BLOCK_SIZE * by;
    int aEnd = aBegin + arraySize - 1;
    int aStep = BLOCK_SIZE;

    int bBegin = BLOCK_SIZE * bx;
    int bStep = BLOCK_SIZE * arraySize;

    float Csub = 0;

    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep)
    {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        As[ty][tx] = A[a + arraySize * ty + tx];
        Bs[ty][tx] = B[b + arraySize * ty + tx];

        __syncthreads();

#pragma unroll
        for (int k = 0; k < BLOCK_SIZE; ++k) {
            Csub += As[ty][k] * Bs[k][tx];
        }

        __syncthreads();
    }

    int c = arraySize * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + arraySize * ty + tx] = Csub;
}
```

Listing 3: Mnożenie macierzy kwadratowych na GPU – wersja 3.

3.4 Wersja 4

```
template <int BLOCK_SIZE> __global__ void
MatrixMulKernel_4(float *C, const float *A, const float *B, const int
    arraySize) {
    int bx = blockIdx.x;
    int by = blockIdx.y;

    int tx = threadIdx.x;
    int ty = threadIdx.y;
```

```
    int aBegin = arraySize * BLOCK_SIZE * by;
    int aEnd = aBegin + arraySize - 1;
    int aStep = BLOCK_SIZE;

    int bBegin = BLOCK_SIZE * bx;
    int bStep = BLOCK_SIZE * arraySize;

    float Csub = 0.0f;

  float fetchA = A[aBegin + arraySize * ty + tx];
    float fetchB = B[bBegin + arraySize * ty + tx];

    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep)
    {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

    As[ty][tx] = fetchA;
    Bs[ty][tx] = fetchB;

        __syncthreads();

    if (a < aEnd) {
      fetchA = A[a + aStep + arraySize * ty + tx];
      fetchB = B[b + bStep + arraySize * ty + tx];
    }

#pragma unroll
       for (int k = 0; k < BLOCK_SIZE; ++k) {
           Csub += As[ty][k] * Bs[k][tx];
       }

        __syncthreads();
    }

    int c = arraySize * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + arraySize * ty + tx] = Csub;
}
```

Listing 4: Mnożenie macierzy kwadratowych na GPU – wersja 4.

3.5 Wersja 5

```
template <int BLOCK_SIZE> __global__ void
MatrixMulKernel_5(float *C, const float *A, const float *B, const int
    arraySize) {
    int bx = blockIdx.x;
    int by = blockIdx.y;
```

```
  int tx = threadIdx.x;
  int ty = threadIdx.y;

  int aBegin = 2 * arraySize * BLOCK_SIZE * by;
  int aEnd = aBegin + arraySize - 1;
  int aStep = 2 * BLOCK_SIZE;

  int bBegin = 2 * BLOCK_SIZE * bx;
  int bStep = 2 * BLOCK_SIZE * arraySize;

float Csub00=0.0f, Csub01=0.0f, Csub10=0.0f, Csub11=0.0f;
float fetchA00, fetchA01, fetchA10, fetchA11;
float fetchB00, fetchB01, fetchB10, fetchB11;

fetchA00 = A[aBegin + arraySize * (2*ty+0) + 2*tx+0];
fetchA01 = A[aBegin + arraySize * (2*ty+0) + 2*tx+1];
fetchA10 = A[aBegin + arraySize * (2*ty+1) + 2*tx+0];
fetchA11 = A[aBegin + arraySize * (2*ty+1) + 2*tx+1];
fetchB00 = B[bBegin + arraySize * (2*ty+0) + 2*tx+0];
fetchB01 = B[bBegin + arraySize * (2*ty+0) + 2*tx+1];
fetchB10 = B[bBegin + arraySize * (2*ty+1) + 2*tx+0];
fetchB11 = B[bBegin + arraySize * (2*ty+1) + 2*tx+1];

  for (int a = aBegin, b = bBegin;
       a <= aEnd;
       a += aStep, b += bStep)
  {
      __shared__ float As[2 * BLOCK_SIZE][2 * BLOCK_SIZE];
      __shared__ float Bs[2 * BLOCK_SIZE][2 * BLOCK_SIZE];

  As[2*ty+0][2*tx+0] = fetchA00;
  As[2*ty+0][2*tx+1] = fetchA01;
  As[2*ty+1][2*tx+0] = fetchA10;
  As[2*ty+1][2*tx+1] = fetchA11;
  Bs[2*ty+0][2*tx+0] = fetchB00;
  Bs[2*ty+0][2*tx+1] = fetchB01;
  Bs[2*ty+1][2*tx+0] = fetchB10;
  Bs[2*ty+1][2*tx+1] = fetchB11;

      __syncthreads();

  fetchA00 = A[a + aStep + arraySize * (2*ty+0) + 2*tx+0];
  fetchA01 = A[a + aStep + arraySize * (2*ty+0) + 2*tx+1];
  fetchA10 = A[a + aStep + arraySize * (2*ty+1) + 2*tx+0];
  fetchA11 = A[a + aStep + arraySize * (2*ty+1) + 2*tx+1];
  fetchB00 = B[b + bStep + arraySize * (2*ty+0) + 2*tx+0];
  fetchB01 = B[b + bStep + arraySize * (2*ty+0) + 2*tx+1];
  fetchB10 = B[b + bStep + arraySize * (2*ty+1) + 2*tx+0];
  fetchB11 = B[b + bStep + arraySize * (2*ty+1) + 2*tx+1];
```

```
    for (int k = 0; k < (2*BLOCK_SIZE); ++k) {
      Csub00 += As[2*ty+0][k] * Bs[k][2*tx+0];
    }
    for (int k = 0; k < (2*BLOCK_SIZE); ++k) {
      Csub01 += As[2*ty+0][k] * Bs[k][2*tx+1];
    }
    for (int k = 0; k < (2*BLOCK_SIZE); ++k) {
      Csub10 += As[2*ty+1][k] * Bs[k][2*tx+0];
    }
    for (int k = 0; k < (2*BLOCK_SIZE); ++k) {
      Csub11 += As[2*ty+1][k] * Bs[k][2*tx+1];
    }

        __syncthreads();
    }

  int c = 2 * arraySize * BLOCK_SIZE * by + 2 * BLOCK_SIZE * bx;
  C[c + arraySize * (2*ty+0) + 2*tx+0] = Csub00;
  C[c + arraySize * (2*ty+0) + 2*tx+1] = Csub01;
  C[c + arraySize * (2*ty+1) + 2*tx+0] = Csub10;
  C[c + arraySize * (2*ty+1) + 2*tx+1] = Csub11;

}
```

Listing 5: Mnożenie macierzy kwadratowych na GPU – wersja 5.

4 Podsumowanie

5 Załączniki

1. Jakiś załącznik

Spis rysunków

Spis tablic

Listingi