Politechnika Poznańska

Wydział Informatyki

Przetwarzanie Równoległe

# Programowanie CUDA na NVIDIA GPU

*Autorzy:*
Adam Szczepański
Mateusz Czajka

*Prowadzący:*
dr Rafał Walkowiak


25 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

Celem projektu było zapoznanie się z możliwościami przetwarzania na kartach graficznych na przykładzie technologi CUDA.

Przygotowaliśmy 5 wersji programu, którego zadaniem było mnożenie macierzy kwadratowych o jednakowych wymiarach na GPU. Każda kolejna wersja jest modyfikacją poprzedniej.

Efektywność programów zbadaliśmy przy pomocy profilera Nsight dla Visual Studio. Dla każdej instancji mierzyliśmy:

- czas wykonania
- ilość operacji zmienno przecinkowych na sekundę (GFLOPS)
- ilość instrukcji wykonanych na sekundę (GIPS)
- przepustowość pamięci globalnej
- stosunek operacji zmienno przecinkowych do ilości operacji odczytu/zapisu z pamięci globalnej (CGMA)
- zajętość warpami multiprocesorów

# 3 Ocena efektywności przetwarzania

## 3.1 Wersja 1

W pierwszej wersji programu wykorzystany został tylko jeden blok wątków. Jeśli rozmiar bloku jest równy rozmiarowi macierzy to każdy z wątków oblicza 1 element wyniku. W przeciwnym przypatku każdy z wątków oblicza

$$\left(\frac{\text{rozmiar macierzy}}{\text{rozmiar bloku}}\right)^2$$

elementów wyniku. Pamięć współdzielona nie jest wykorzystywana.

```
__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.

Ponieważ wykorzystany jest tylko jeden blok, obliczenia przeprowadzane są na jednym SM, a co za tym idzie w danej chwili aktywny może być tylko jeden warp.
Duży wpływ na prędkość przetwarzania ma wielkość bloku, dla małych bloków efektywność jest bardzo niska – duża ilość dostępów do pamięci ogranicza prędkość przetwarzania.

| Rozmiar macierzy | Rozmiar bloku | | |
|---|---|---|---|
| | 8x8 | 16x16 | 22x22 |
| 176x176 | 49.79 | 13.17 | 7.45 |
| 352x352 | 401.35 | 103.85 | 59.70 |
| 528x528 | 1354.40 | 349.77 | 200.36 |

Tablica 2: Czas obliczeń [ms] – wersja 1.

## 3.2 Wersja 2

W drugiej wersji wykorzystywany jest grid wieloblokowy o rozmiarze

$$\frac{\text{rozmiar macierzy}}{\text{rozmiar bloku}}$$

Każdy wątek oblicza jeden element macierzy wynikowej. Pamięć współdzielona nie jest wykorzystywana.

```
__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.

| Rozmiar macierzy | Rozmiar bloku | | |
|---|---|---|---|
| | 8x8 | 16x16 | 22x22 |
| 176x176 | 1.47 | 1.38 | 2.56 |
| 352x352 | 12.69 | 11.68 | 21.46 |
| 528x528 | 38.57 | 36.82 | 67.02 |

Tablica 3: Czas obliczeń [ms] – wersja 2.

## 3.3 Wersja 3

W trzecim podejściu wykorzystana została pamięć współdzielona. W kolejnych iteracjach pętli po blokach najpierw wczytywany jest blok do pamięci współdzielonej (każdy wątek wczytuje jedną komórkę), a następnie wykonywane są obliczenia na dostępnych danych. W tym podejściu niezbędne jest synchronizowanie się wątków dwukrotnie w każdej iteracji.

```
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.

| Rozmiar macierzy | Rozmiar bloku | | |
|---|---|---|---|
| | 8x8 | 16x16 | 22x22 |
| 176x176 | 0.31 | 0.19 | 0.32 |
| 352x352 | 2.33 | 1.27 | 2.23 |
| 528x528 | 7.66 | 4.07 | 7.39 |

Tablica 4: Czas obliczeń [ms] – wersja 3.

| Rozmiar macierzy | Rozmiar bloku |
|---|---|
| | 16x16 |
| 64x64 | 0.02 |
| 128x128 | 0.09 |
| 256x256 | 0.52 |
| 384x384 | 1.62 |
| 512x512 | 3.76 |

Tablica 5: Czas obliczeń [ms] – wersja 3.

### 3.4 Wersja 4

Jest to rozszerzona wersja 3 o równoległe z obliczeniami pobranie kolejnych danych (na poziomie bloku). Ma to spowodować złagodzenie kosztów synchronizacji.

```
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.

| Rozmiar macierzy | Rozmiar bloku |
|---|---|
| | 16x16 |
| 64x64 | 0.02 |
| 128x128 | 0.10 |
| 256x256 | 0.60 |
| 384x384 | 1.89 |
| 512x512 | 4.44 |

Tablica 6: Czas obliczeń [ms] – wersja 4.

## 3.5 Wersja 5

Ostatnia wersja rozszerza wersję 4 – każdy wątek wykonuje większą pracę. Zostało to zrealizowane przez zwiększenie pobieranych i obliczanych danych z jednej do czterech.

```
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.

| Rozmiar macierzy | Rozmiar bloku |
|---|---|
| | 16x16 |
| 64x64 | 0.06 |
| 128x128 | 0.22 |
| 256x256 | 1.23 |
| 384x384 | 3.66 |
| 512x512 | 8.80 |

Tablica 7: Czas obliczeń [ms] – wersja 5.

# 4 Podsumowanie

# 5 Załączniki

1. Kody źródłowe

# Spis rysunków

# Spis tablic

# Listingi