Raport: Mnożenie macierzy

Heorhii Lopatin (hl456366)

2 maja 2023

1 Wstęp

Celem tego zadania jest stworzenie, przetestowanie oraz porównanie wydajności różnych wersji kernela CUDA, które służą do obliczania iloczynu macierzy.

2 Opis implementacji

Zaimplementowano 5 wersij kernela. Pierwsza - przeniesienie kodu CPU na GPU. Pozostałe wersje dzielą macierz na bloki o rozmiarach K na K i liczą każdy blok za pomocą jednego wątku.

Kernel 1

```
__global__ void mult_gpu1(real* A, real* B, real* C, int N) {
  int column = blockIdx.x * blockDim.x + threadIdx.x;
  int row = blockIdx.y * blockDim.y + threadIdx.y;
  int locA, locB;
  int k;
  real c = 0.0;
  locA = N * row;
  locB = N * column;
  if (column < N && row < N) {
    for (k = 0; k < N; k++) {
      c += A[row * N + k] * B[k*N + column];
    }
    C[row * N + column] = c;
  }
}

```

Kernel 2

```
template <uint32_t dim>
__global__ void mult_gpu2(real* A, real* B, real* C, int N) {
  int start_x = blockIdx.x * dim;
  int start_y = blockIdx.y * dim;

  int iter = blockIdx.x < gridDim.x ? DIM : DIM - (DIM * gridDim.x - N);

  __shared__ real OUT[dim][dim];
  __shared__ real rowB[dim];

  int i = threadIdx.x;

  for (int n = 0; n < dim; n++)
    OUT[n][i] = 0;
  __syncthreads();

  int row = start_y + i;
  for (int k = 0; k < N; ++k) {
    if (i < iter)
      rowB[i] = B[k * N + start_x + i];
    __syncthreads();

    real colA = row < N ? A[row * N + k] : 0;
    __syncthreads();

    for (int n = 0; n < dim; n++) {
      OUT[i][n] += colA * rowB[n];
    }
    __syncthreads();
  }
  __syncthreads();
  if (row < N) {
    for (int n = 0; n < dim; n++) {
      C[row * N + start_x + n] = OUT[i][n];
    }
  }
  __syncthreads();
}

```

Kernel 3

```
template <uint32_t dim>
__global__ void mult_gpu3(real* A, real* B, real* C, int N) {
  int start_x = blockIdx.x * dim;
  int start_y = blockIdx.y * dim;

  int iter = blockIdx.x < gridDim.x ? DIM : DIM - (DIM * gridDim.x - N);

  __shared__ real OUT[dim][dim + 1];
  __shared__ real rowB[dim];

  int i = threadIdx.x;

  for (int n = 0; n < dim; n++)
    OUT[n][i] = 0;
  __syncthreads();

  int row = start_y + i;
  for (int k = 0; k < N; ++k) {
    if (i < iter)
      rowB[i] = B[k * N + start_x + i];
    __syncthreads();

    real colA = row < N ? A[row * N + k] : 0;
    __syncthreads();

    for (int n = 0; n < dim; n++) {
      OUT[i][n] += colA * rowB[n];
    }
    __syncthreads();
  }
  __syncthreads();
  if (row < N) {
    for (int n = 0; n < dim; n++) {
      C[row * N + start_x + n] = OUT[i][n];
    }
  }
  __syncthreads();
}

```

Kernel 4

```
template <uint32_t dim>
__global__ void mult_gpu4(real* A, real* B, real* C, int N) {
  int start_x = blockIdx.x * dim;
  int start_y = blockIdx.y * dim;

  int iter = blockIdx.x < gridDim.x ? DIM : DIM - (DIM * gridDim.x - N);

  __shared__ real OUT[dim][dim + 1];
  real rowB;

  int i = threadIdx.x;

  for (int n = 0; n < dim; n++)
    OUT[n][i] = 0;
  __syncthreads();

  int row = start_y + i;
  for (int k = 0; k < N; ++k) {
    if (i < iter)
      rowB = B[k * N + start_x + i];
    __syncthreads();

    real colA = row < N ? A[row * N + k] : 0;
    __syncthreads();

    for (int n = 0; n < dim; n++) {
      OUT[i][n] += colA * __shfl_sync(0xffffffff, rowB, n);
    }
    __syncthreads();
  }
  __syncthreads();
  if (row < N) {
    for (int n = 0; n < dim; n++) {
      C[row * N + start_x + n] = OUT[i][n];
    }
  }
  __syncthreads();
}

```

Kernel 5

```
template <uint32_t dim>
__global__ void mult_gpu5(real* A, real* B, real* C, int N) {
  int start_x = blockIdx.x * dim;
  int start_y = blockIdx.y * dim;

  int iter = blockIdx.x < gridDim.x ? DIM : DIM - (DIM * gridDim.x - N);

  real OUT[dim];
  real rowB;

  int i = threadIdx.x;

  for (int n = 0; n < dim; n++)
    OUT[n] = 0;
  __syncthreads();

  int row = start_y + i;
  for (int k = 0; k < N; ++k) {
    if (i < iter)
      rowB = B[k * N + start_x + i];
    __syncthreads();

    real colA = row < N ? A[row * N + k] : 0;
    __syncthreads();

    for (int n = 0; n < dim; n++) {
      OUT[n] += colA * __shfl_sync(0xffffffff, rowB, n);
    }
    __syncthreads();
  }
  __syncthreads();
  if (row < N) {
    for (int n = 0; n < dim; n++) {
      C[row * N + start_x + n] = OUT[n];
    }
  }
  __syncthreads();
}

```

3 Wyniki

Kod przetestowano na macierzach 1000x1000. Wyniki przeprowadzonego eksperymentu przedstawiono w tabeli:

3.1 Tabela 1. Podsumowanie wyników

Wersja kodu	Konfiguracja kernela	Średni czas wykonania (ms)	Przyspieszenie względem CPU
CPU	-	4356.31±0.61	1.00
Kernel #1	8 x 1	48.27±0.01	90.25±0.02
Kernel #1	16 x 1	33.17±0.02	131.31±0.08
Kernel #1	32 x 1	26.80±0.01	162.55±0.06
Kernel #2	32 x 1	120.51±6.95	36.15±2.22
Kernel #2	64 x 1	257.80±0.45	16.90±0.03
Kernel #2	96 x 1	321.30±0.07	13.56±0.01
Kernel #2	128 x 1	321.38±0.09	13.55±0.01
Kernel #3	32 x 1	19.48±0.01	223.68±0.09
Kernel #3	64 x 1	23.30±0.05	186.96±0.46
Kernel #4	32 x 1	17.24±0.01	252.71±0.07
Kernel #5	32 x 1	10.54±0.01	413.15±0.10

4 Podsumowanie

Patrząc na wyniki, możemy wyciągnąć kilka wniosków:

- Należy dbać o możliwych konfliktach banków pamięci.
- Jeśli można uniknąć dostępu do/utworzenia obiektów w pamięci , należy to zrobić

Najszybszy był kernel w 5 wersji. Wydaje się, że przez małą liczbę zmiennych użytych w funkcji, mogą być przechowywane w rejestrach, a to oznacza, że dostęp do zawartości oraz wymiana zmiennych są szybkie. Dodatkowo, wyniki przechowywane w pamięci lokalnej, co też ułatwia dostęp.

Kernele w wersji 3,4 też były w miarę szybkie, jednak wolniejsze od 5 wersji przez dostęp do pamięci współdzielonej w każdym obrocie pętli.