Умножение матриц

Вычисления на видеокартах. Лекция 4

Напоминание про barrier
Обсуждение задачи про максимальный префикс
Matrix transpose
Matrix multiplication

Полярный Николай

polarnick239@gmail.com

Напоминание

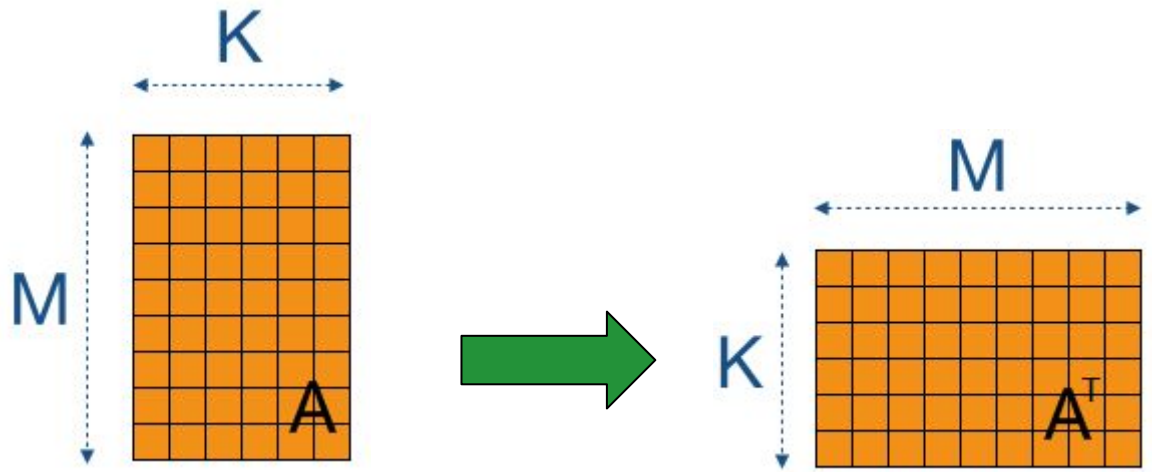
barrier - должен быть выполнен из каждого потока рабочей группы

```
__local local_xs[WORKSIZE];
if (get_global_id(0) >= size) {
    return;
}
local_xs[get_local_id(0)] = xs[get_global_id(0)];
barrier(CLK_LOCAL_MEM_FENCE); // Ошибка, если size % WorkGroupSize != 0
// ...
```

```
__local local_xs[WORKSIZE];
if (get_global_id(0) >= size) {
    local_xs[get_local_id(0)] = 0; // Иногда везет и можно использовать нейтральный элемент
}
local_xs[get_local_id(0)] = xs[get_global_id(0)];
barrier(CLK_LOCAL_MEM_FENCE);
// ...
```

Транспонирование матрицы. Наивная версия

Нужно транспонировать матрицу **A** размера **M**x**K**



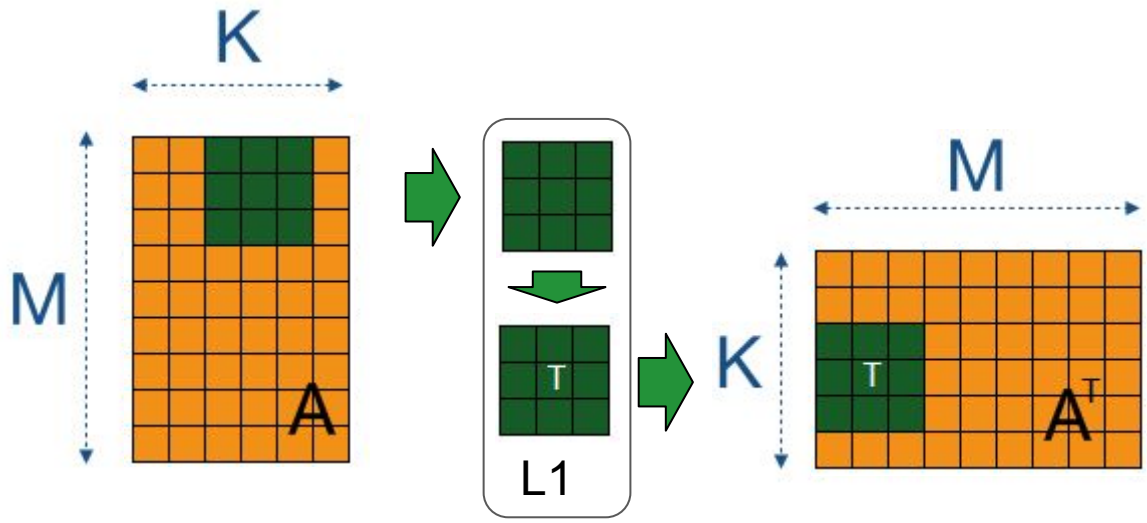

```
__kernel void transpose1(__global float *a, __global float *at, unsigned int m, unsigned int k)
{
    int i = get_global_id(0); // номер столбца A (от 0 до k)
    int j = get_global_id(1); // номер строчки A (от 0 до m)
    float x = a[j * k + i];
    at[i * m + j] = x;
}
```

- Сколько операций считывания? Они образуют memory coalesced access?
- Сколько операций записи? Они образуют memory coalesced access?

Транспонирование матрицы. Local memory

Как довести до идеального coalesced memory access?

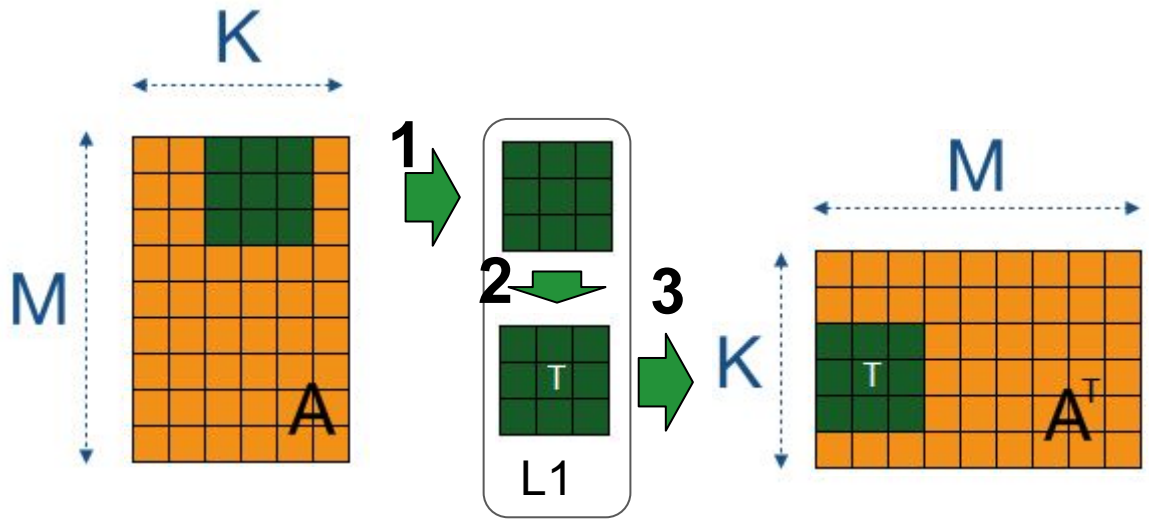
Использовать **local memory (L1)**!



Т.е. сместить проблему **non-coalesced** доступа на уровень локальной памяти где такой проблемы нет. (но есть потенциальные **bank-conflicts**)

Транспонирование матрицы. Local memory

1) Считать из глобальной памяти в локальную. **barrier(...)**
2) Транспонировать в локальной памяти. **barrier(...)**
3) Записать результат в глобальную память.



Транспонирование матрицы. Local memory

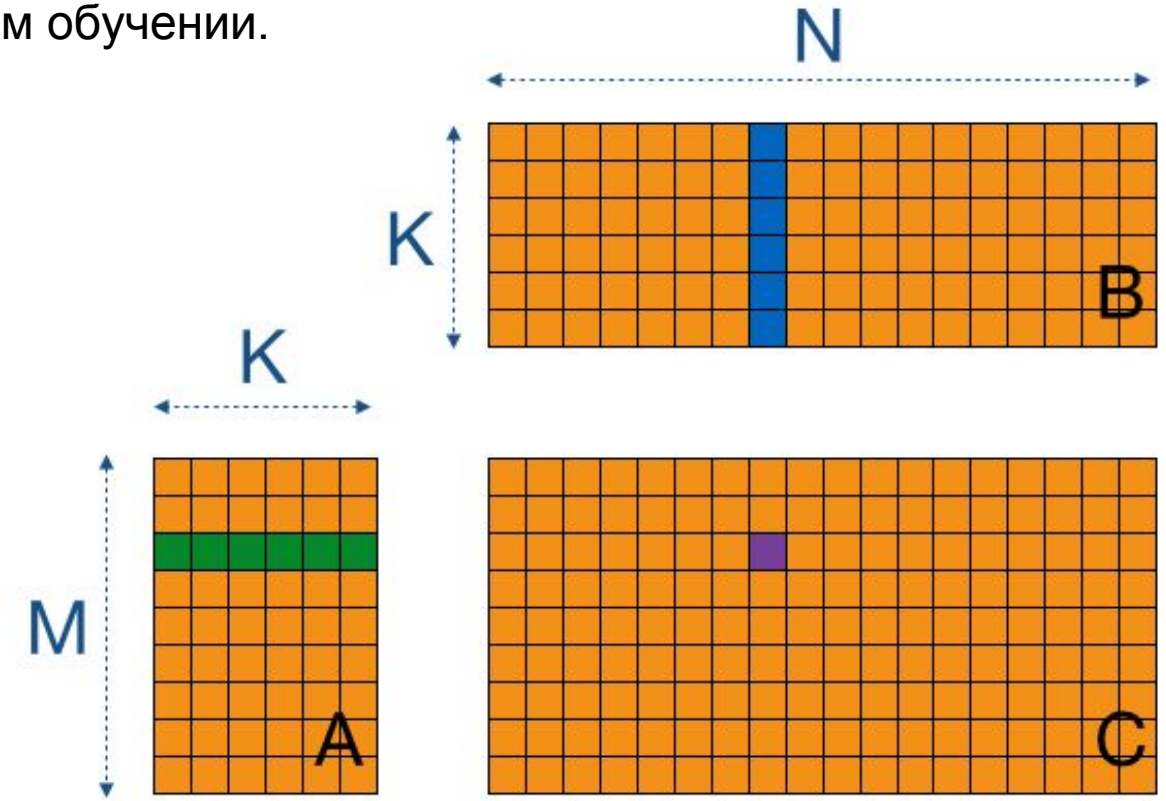
```
#define TILE_SIZE 32
__kernel void transpose2(__global float *a, __global float *at, unsigned int m, unsigned int k)
{
    int i = get_global_id(0); // номер столбца A (от 0 до k)
    int j = get_global_id(1); // номер строчки A (от 0 до m)
    // В compute unit обычно от 48 KB до 96 KB локальной памяти и до 10-20 warps/wavefronts
    __local float tile[TILE_SIZE][TILE_SIZE]; // 32*32*4 bytes = 4 KB
    int local_i = get_local_id(0); // номер столбца в кусочке (от 0 до TILE_SIZE)
    int local_j = get_local_id(1); // номер строчки в кусочке (от 0 до TILE_SIZE)

    // 1) Считать из глобальной памяти в локальную
    tile[j * TILE_SIZE][i] = a[j * k + i];              // TODO: проверить coalesced ли?

    // 2) Транспонировать в локальной памяти
    float tmp = tile[j * TILE_SIZE][i];                 // TODO: опционально подумать
    tile[j * TILE_SIZE][i] = tile[i * TILE_SIZE][j];    // есть ли здесь bank-conflicts
    tile[i * TILE_SIZE][j] = tmp;                       // и если есть - как уменьшить/убрать?

    // 3) Записать результат в глобальную память
    at[i * m + j] = tile[j * TILE_SIZE][i];             // TODO: проверить coalesced ли?

    // TODO: расставить барьеры
    // TODO: продумать как адаптировать код для рабочей группы 16*16=256 (при TILE_SIZE=32)
}
```

Умножение матриц

Размер **A** - **M*K**, размер **B** - **K*N**, размер **C** - **M*N**.

Асимптотика - **O(M*N*K)** - арифметика.

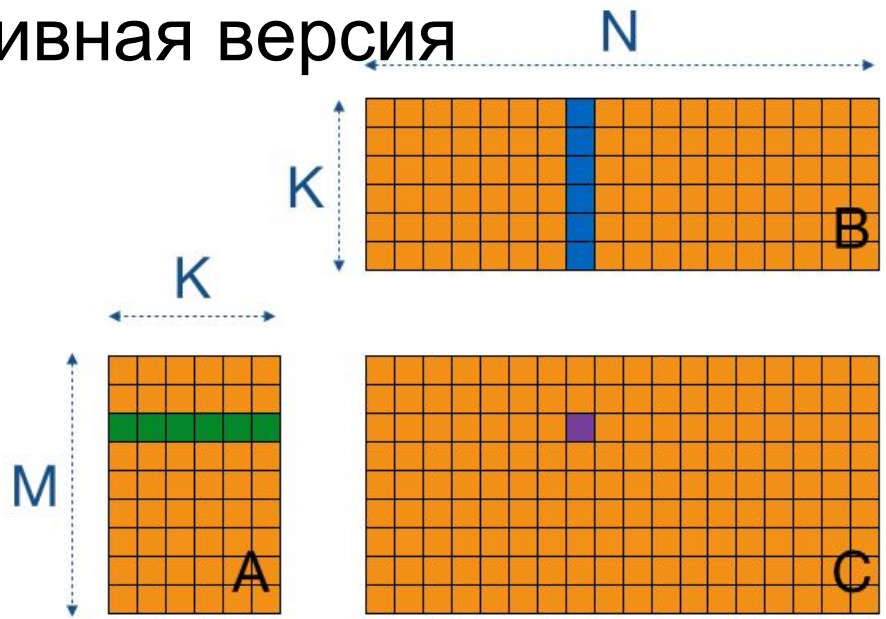
Используется в машинном обучении.



Умножение матриц 1: наивная версия

O(M*N*K) считываний данных!

Как не упереться в memory bandwidth?
Использовать local memory!

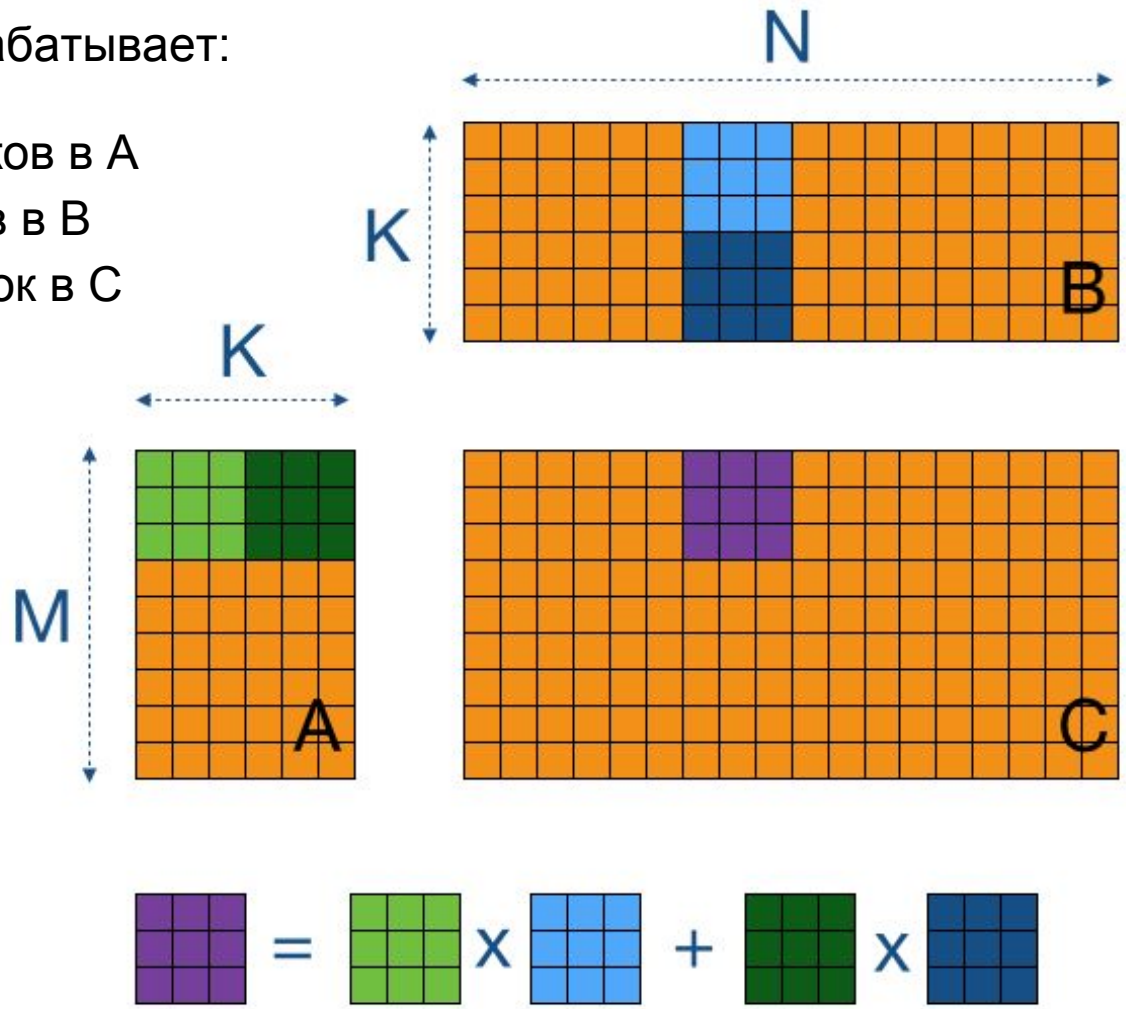



```
__kernel void matmul1(__global float *a, __global float *b, __global float *c,
                      unsigned int M, unsigned int K, unsigned int N) {
    int i = get_global_id(0); // номер столбца результирующей C (от 0 до N)
    int j = get_global_id(1); // номер строчки результирующей C (от 0 до M)

    float sum = 0.0f;
    for (int k = 0; k < K; ++k) {
        sum += a[j * K + k] * b[k * N + i];
    }
    c[j * N + i] = sum;
}
```

Умножение матриц 2: локальная память

Одна рабочая группа обрабатывает:

- Строчку зеленых блоков в A
- Столбец синих блоков в B
- Один фиолетовый блок в C



Т.е. давайте рабочей группой обрабатывать строку/столбец в A/B двигая и там и там синхронно два блока, подгружая их в локальную память, и добавляя произведение между ними в локальный аккумулятор.

Умножение матриц 2: локальная память

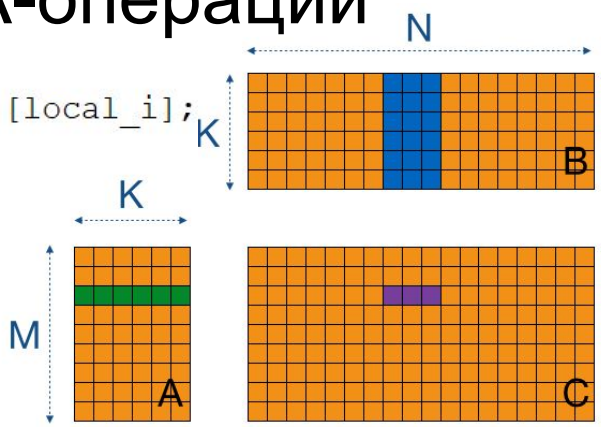
```
#define TILE_SIZE 32
__kernel void matmul2(__global float *a, __global float *b, __global float *c,
                      unsigned int M, unsigned int K, unsigned int N) {
    int i = get_global_id(0); // номер столбца результирующей C (от 0 до N)
    int j = get_global_id(1); // номер строчки результирующей C (от 0 до M)
    int local_i = get_local_id(0); // номер столбца в кусочке (от 0 до TILE_SIZE)
    int local_j = get_local_id(1); // номер строчки в кусочке (от 0 до TILE_SIZE)
    __local float tileA[TILE_SIZE][TILE_SIZE];
    __local float tileB[TILE_SIZE][TILE_SIZE];

    float sum = 0.0f;
    for (int tileK = 0; tileK * TILE_SIZE < K; ++tileK) {
        // Делаем очередной сдвиг куска tileA в A и tileB в B
        // 1. Считываем элементы из A в tileA
        // 2. Считываем элементы из B в tileB
        for (int k = 0; k < TILE_SIZE; ++k) {
            // 3. Добавляем в свой аккумулятор sum те произведения что нам нужны:
            sum += tileA[local_j * TILE_SIZE][k] * tileB[...][...];
        }
    }
    c[j * N + i] = sum;
}
```

Умножение матриц 2: локальная память

```
#define TILE_SIZE 32
__kernel void matmul2(__global float *a, __global float *b, __global float *c,
                      unsigned int M, unsigned int K, unsigned int N) {
    int i = get_global_id(0); // номер столбца результирующей C (от 0 до N)
    int j = get_global_id(1); // номер строчки результирующей C (от 0 до M)
    int local_i = get_local_id(0); // номер столбца в кусочке (от 0 до TILE_SIZE)
    int local_j = get_local_id(1); // номер строчки в кусочке (от 0 до TILE_SIZE)
    __local float tileA[TILE_SIZE][TILE_SIZE];
    __local float tileB[TILE_SIZE][TILE_SIZE];
    // TODO не забыть про барьеры
    float sum = 0.0f;
    for (int tileK = 0; tileK * TILE_SIZE < K; ++tileK) {
        // Считываем элементы из A в tileA и tileB в B:
        tileA[local_j * TILE_SIZE][local_i] = a[j * K + (tileK * TILE_SIZE + local_i)
        tileB[...][...] = b[...];
        for (int k = 0; k < TILE_SIZE; ++k) {
            // 3. Добавляем в свой аккумулятор sum те произведения что нам нужны:
            sum += tileA[local_j * TILE_SIZE][k] * tileB[...][...];
        }
    }
    c[j * N + i] = sum;
}
```

Умножение матриц 3: больше FMA-операций

```
for (int k = 0; k < TILE_SIZE; ++k) {
    sum += tileA[local_j * TILE_SIZE][k] * tileB[k * TILE_SIZE][local_i];
}
```



На каждой итерации внутреннего цикла:

1) подгрузить значение из Atile

2) подгрузить значение из Btile

3) умножить и добавить (FMA инструкция)

Можно заменить две подгрузки на одну если каждый поток аккумулирует сразу несколько значений из C - см. иллюстрацию.

```
float sum[THREAD_WORK];
for (int k = 0; k < TILE_SIZE; ++k) {
    for (int w = 0; w < THREAD_WORK; ++w) {
        sum[w] += tileA[local_j*TILE_SIZE][k] * tileB[k*TILE_SIZE][local_i*THREAD_WORK+w];
    }
}
```

```
float sum[THREAD_WORK];
for (int k = 0; k < TILE_SIZE; ++k) {
    float tmp = tileA[local_j*TILE_SIZE][k];
    for (int w = 0; w < THREAD_WORK; ++w) {
        sum[w] += tmp * tileB[k*TILE_SIZE][local_i*THREAD_WORK+w];
    }
}
```

Умножение матриц 4: local memory -> registers

Теперь на каждой итерации внутреннего цикла:
1) подгружается очередное значение из **tileB**
2) умножить и добавить (FMA инструкция)

Оптимзация:

Хочется подгружать значение из **tileB** не из **local memory**, а из регистров.

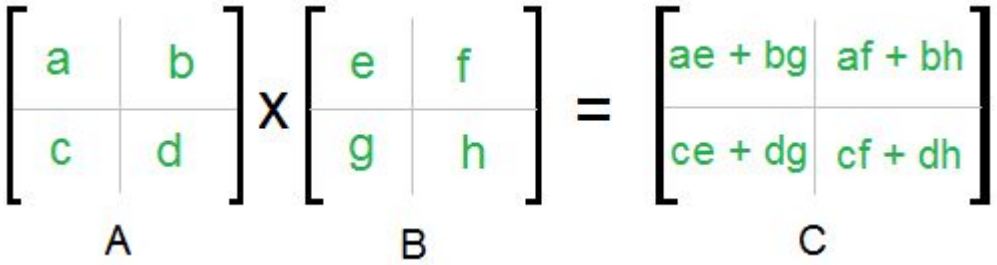
Т.е. сначала прогрузить из **local memory** в **регистры**, а потом быстро забирать оттуда.

Но это бессмысленно пока поток обрабатывает всего лишь ряд внутри **C**.
Поэтому нужно обрабатывать микро-блок внутри **C**.

Подробнее см. https://cnugteren.github.io/tutorial/pages/page8.html

Умножение матриц методом Штрассена

Пусть все три матрицы **N**x**N**. Подразобьем их следующим образом:



Получили рекуррентную асимптотику T(N)=8T(N/2)+4O(N^2) => T(N)=**O(N^3)**

Умножение матриц методом Штрассена

p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}_A \times \begin{bmatrix} e & f \\ g & h \end{bmatrix}_B = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}_C$$

Получили рекуррентную асимптотику T(N)=**7**T(N/2)+8O(N^2) => T(N)=O(N^log7)=**O(N^2.8)**

Метод Виноградова - тоже **O(N^2.8)**

Источник: https://www.geeksforgeeks.org/strassens-matrix-multiplication/

Умножение матриц методом Штрассена на GPU

Плюсы:

- Метод Штрассена быстрее на 32%
- Метод Виноградова быстрее на 33%

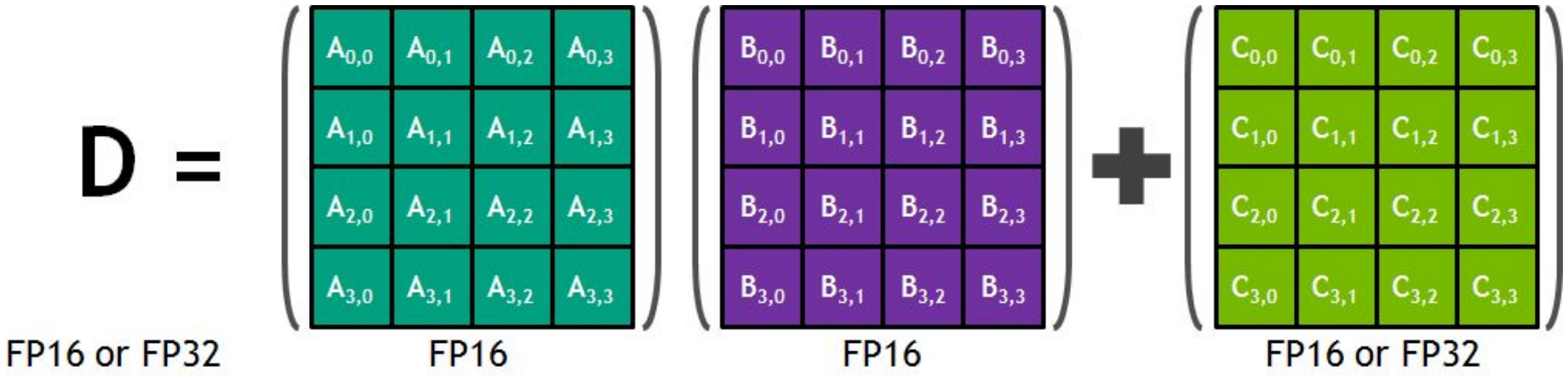
Минусы:

- Большая константа
- Точность падает на один-два порядка

Источник:

Strassen’s Matrix Multiplication on GPUs, Junjie Li, Sanjay Ranka, Sartaj Sahni

Tensor Cores, NVIDIA



Tensor processing unit, Google