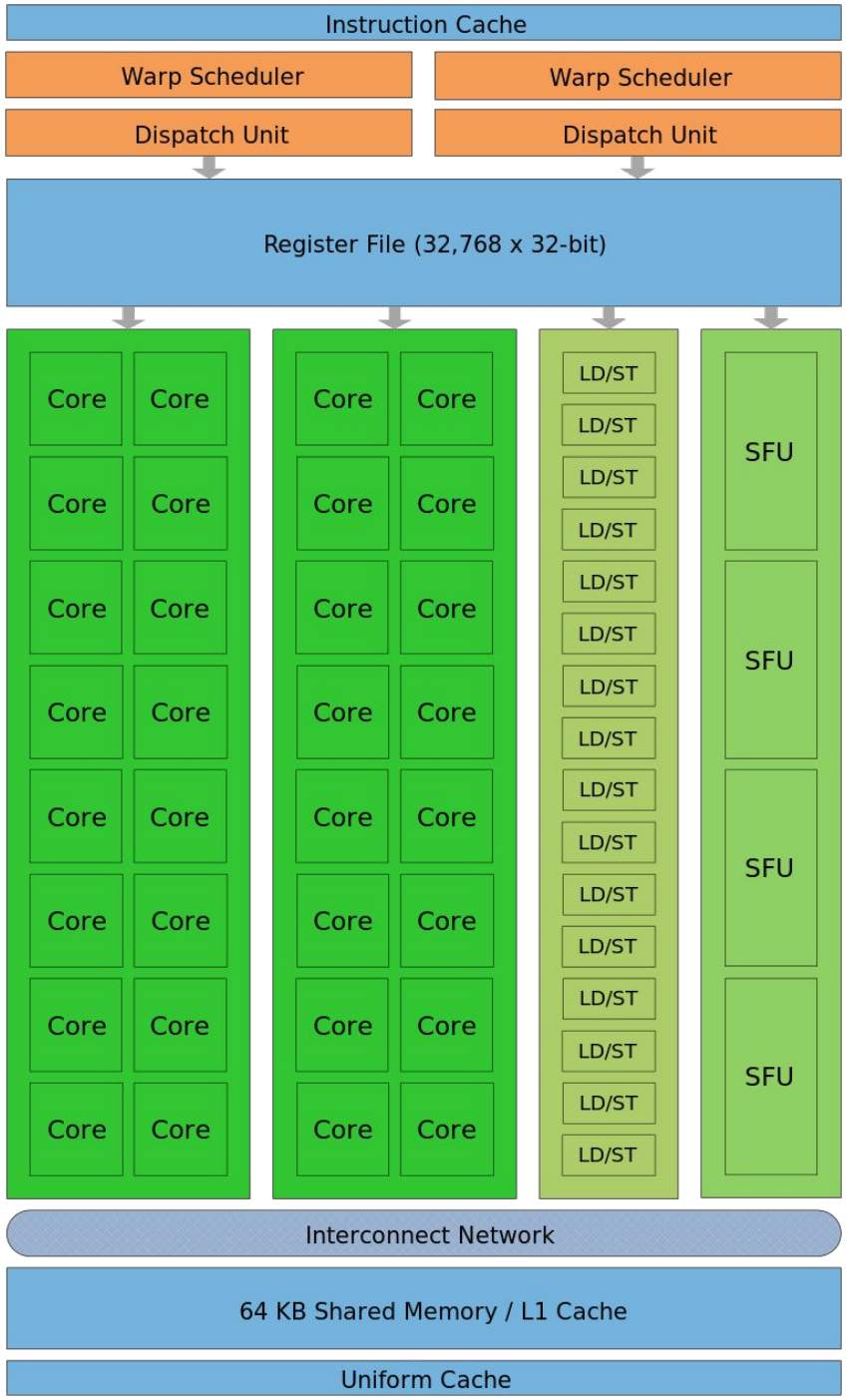
Примеры оптимизаций с local memory.

Вычисления на видеокартах. Лекция 3

Немного уточнений по архитектуре
N-body simulation
Brute force matching
Key points detection
Matrix transpose
Matrix multiplication

Полярный Николай

polarnick239@gmail.com



NVIDIA: **32** threads in warp (**32**x**Core** слева)

AMD: **64** threads in wavefront

Каждая WorkGroup состоит из одного или более warp/wavefront-ов.

Например если рабочая группа размера 128, а размер wavefront=64, то **рабочая группа выполняется** двумя wavefront-ами **в рамках одного compute unit**. Но эти два wavefront не обязаны выполняться действительно параллельно, да и не всегда могут - ведь в compute unit может быть всего один warp/wavefront.

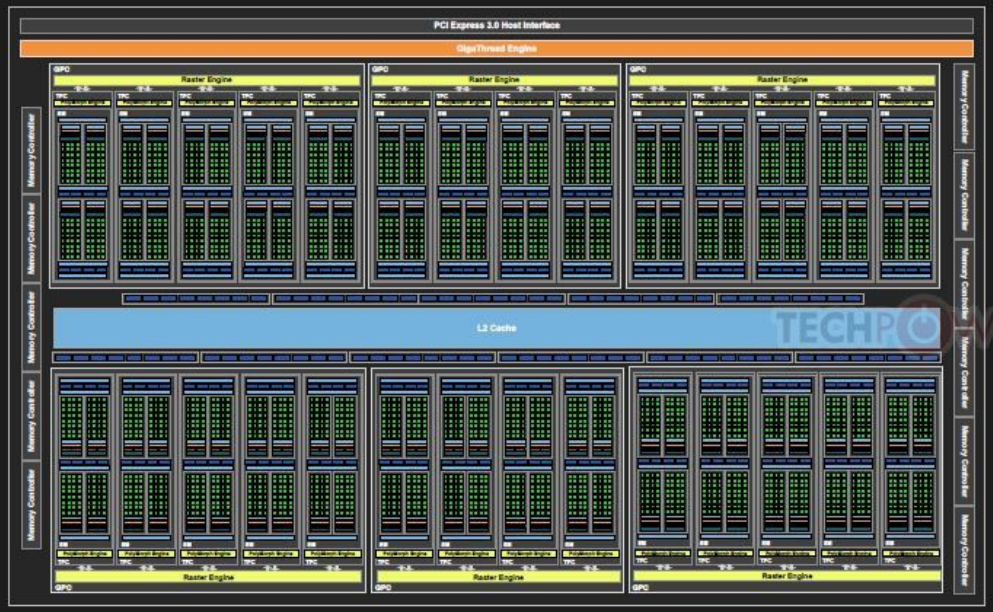
Но они могут выполняться чередуясь на одном и том же множестве **Cores (т.н. ALUs = Arithmetic Logic Units).**

Т.о. из всех потоков одной WorkGroup доступна одна и та же local memory.

NVIDIA GTX 1080ti

28 compute units (i.e. Streaming Multiprocessor) * **128 cores** (i.e. 4 warps with 32 ALUs in each)
= **3584 ALUs** (CUDA cores)

GTX 1080 TI OVERVIEW



12B Transistors
1.6 GHz Boost, 2 GHz OC
28 SMs, 128 cores each
3584 CUDA cores
28 Geometry units
224 Texture units
6 GPCs
88 ROP units
352 bit GDDR5x

AMD Vega 64

64 compute units * **64 cores** (i.e. 1 wavefront - 64 ALUs)
= **4096 ALUs** (stream processors)

"Vega 10" by the numbers

1 Graphics Engine

4 Asynchronous Compute Engines

4 Next-Gen Geometry Engines

64 Next-Gen Compute Units

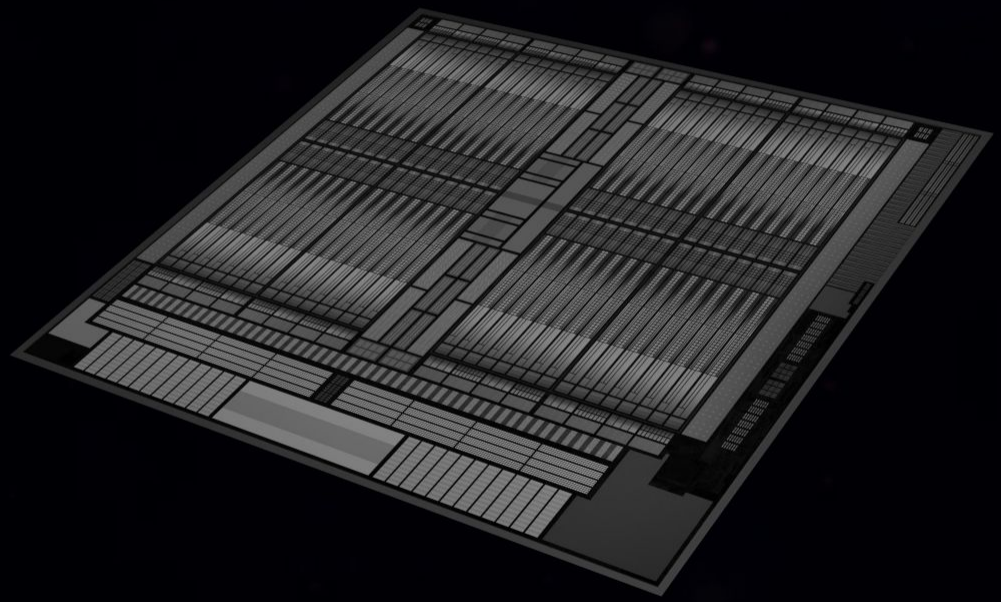
4096 Stream Processors

256 Texture Units

64 Render Back-Ends

4 MB L2 Cache

2048-bit HBM2



N-body simulation

Есть N частиц с массой, нужно промоделировать взаимное притяжение каждой пары частиц.

Асимптотика - **$O(N^2)$**.

Как не упереться в **memory bandwidth**? (как при суммировании чисел)

Использовать **local memory**!

Brute Force feature matching

Есть N ключевых точек на одной картинке, и M ключевых точек на второй картинке. Нужно найти для каждой парную (самую похожую).

Асимптотика - **O(N*M)**.

Как не упереться в **memory bandwidth**?

Использовать **local memory**!

Key points detection

Есть картинка, хочется в каждом пикселе проверить “не угол ли это на границе” и если да - добавить этот пиксель в результат.

Как осуществить добавление пикселя в конец результата?
atomic_add + локальный аккумулятор “аллокаций”

Как аллоцировать буфер в видеопамяти под результат чей размер заранее мы не знаем?

Транспонирование матрицы

Как довести наивную версию до идеального coalesced memory access?

Использовать **local memory**!

Умножение матриц

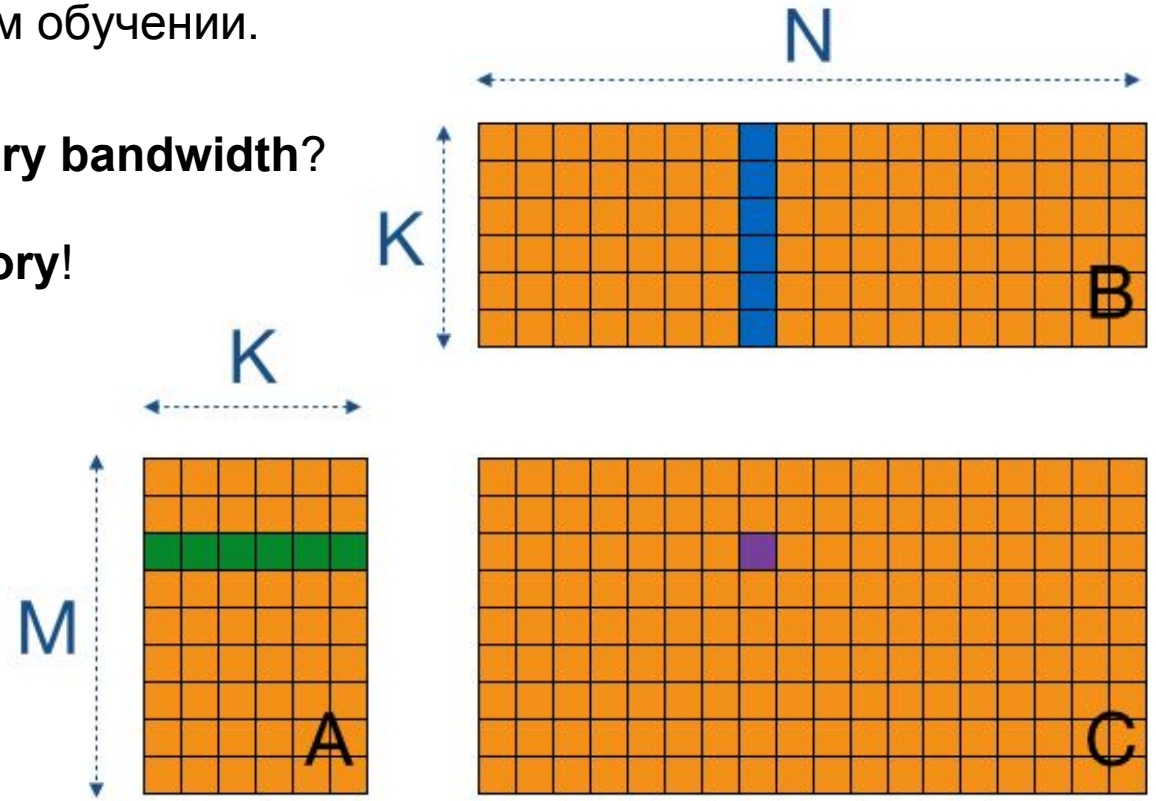
Размер **A** - **M*K**, размер **B** - **K*N**, размер **C** - **M*N**.

Асимптотика - **O(M*N*K)**.

Используется в машинном обучении.

Как не упереться в **memory bandwidth**?

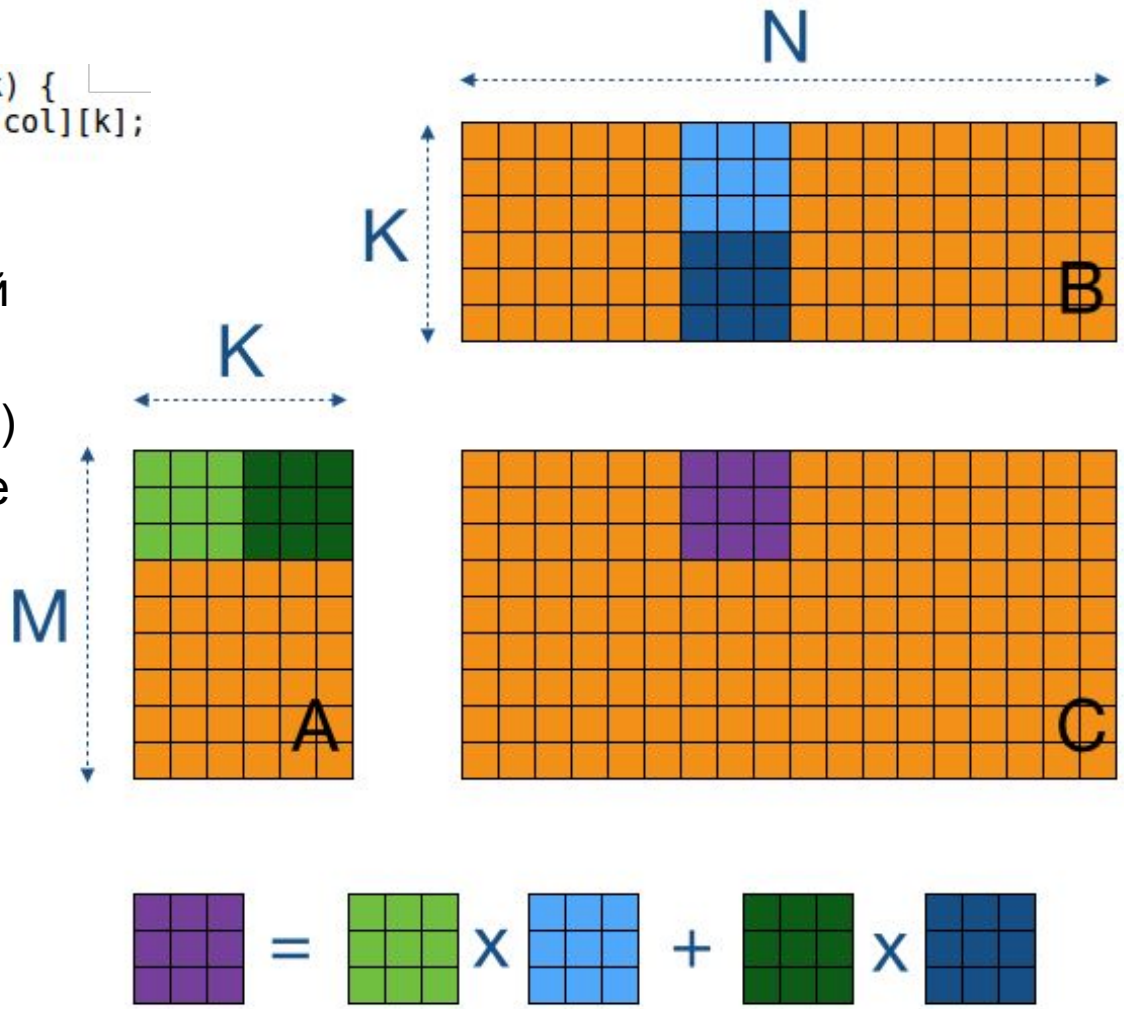
Использовать **local memory**!



Умножение матриц 1: локальная память

```
for (int k = 0; k < TileSize; ++k) {
    acc += Atile[k][row] * Btile[col][k];
}
```

Т.е. подгружать очередной
блок из A и из B
в local memory (Atile и Btile)
и добавлять произведение
в аккумулятор потока.

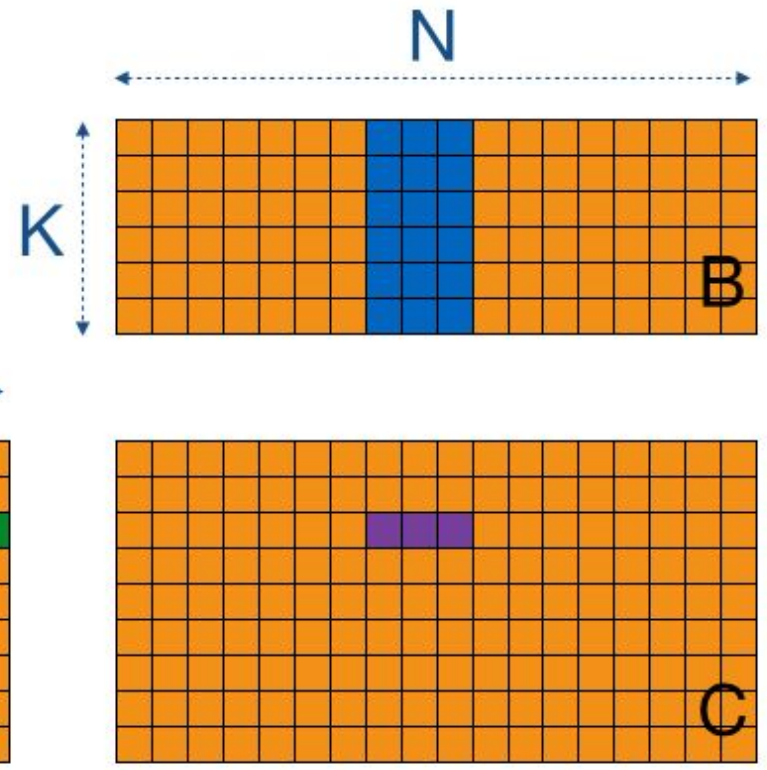


Умножение матриц 2: больше FMA-операций

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.



Было:

```
for (int k = 0; k < TileSize; ++k) {
    acc += Atile[k][row] * Btile[col][k];
}
```

Стало:

```
for (int k = 0; k < TileSize; ++k) {
    for (int w = 0; w < TileSize; ++w) {
        acc[w] += Atile[k][row] * Btile[col + w * TileSize][k];
    }
}
```

Умножение матриц 3: local memory -> registers

На каждой итерации внутреннего цикла:
1) подгрузить значение из Btile
2) умножить и добавить (FMA инструкция)

Оптимзация:

Хочется подгружать значение из Btile 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**.

Тут уже надо аккуратно балансировать между высоким числом используемых регистров и удельным количеством обращений к локальной памяти. Т.к. при большом числе используемых регистров будет низкая occupancy и начнет стучать global memory access (т.к. низкий процент occupancy - плохое сокрытие latency).

Умножение матриц

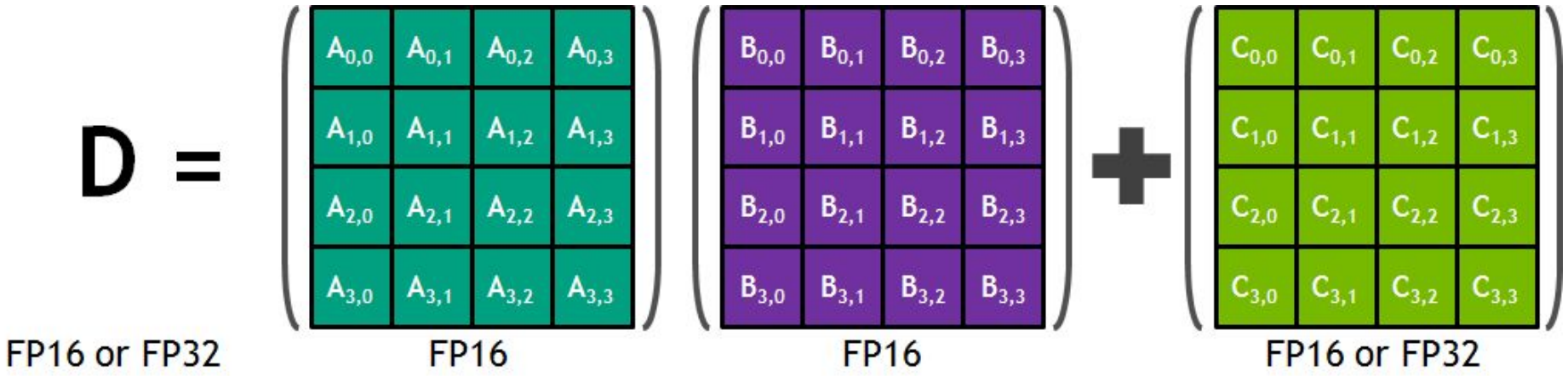
Но можно ли быстрее?

- loop-unrolling
- минимизация числа барьеров/CUDA shuffle instruction
- считать в half-precision
- на ассемблере ради минимального числа регистров и ради интринсик

Но можно ли быстрее?

Да, но специализированное железо и/или считать в half-precision.

Tensor Cores, NVIDIA



Tensor processing unit, Google