CS Space
Клуб технологий и науки

Вычисления на видеокартах

- План курса
- Пререквизиты
- Архитектура CPU
- История GPU и GPGPU
- Введение в OpenCL API
- Домашние задания

Vulkan
OpenCL™
NVIDIA
CUDA

polarnick239@gmail.com
Николай Полярный

План курса, задания на **github + экзамен**

- Введение в OpenCL/CUDA API
- Архитектура видеокарты
- Примеры оптимизаций с local memory
- Транспонирование, умножение матриц
- merge sort
- Разреженные матрицы
- Bitonic sort, radix sort
- LBVH construction + Ray Tracing
- Растеризация: Vulkan, Larrabee, cudaraster
- Ray marching (SDF, shadertoy)
- Nanite в UE5: Real-time rendering полигональной модели любого размера

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями:

```
int main() {
    std::vector<int> values;
    values[5] = values[11];
```

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями:

```
void* memcpy(void* destination, const void* source, size_t bytes_count);

int main() {
    std::vector<int> values;
    values[5] = values[11];

    memcpy(
}
```

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями:

```
void* memcpy(void* destination, const void* source, size_t bytes_count);

int main() {
    std::vector<int> values;
    values[5] = values[11];

    void *ptr = values.data();                    // адрес первого элемента в values




    memcpy(
}
```

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями:

```
void* memcpy(void* destination, const void* source, size_t bytes_count);

int main() {
    std::vector<int> values;
    values[5] = values[11];

    void *ptr = values.data();                  // адрес первого элемента в values

    void *source       = ???                    ; // откуда


    memcpy(
}
```

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями:

```
void* memcpy(void* destination, const void* source, size_t bytes_count);

int main() {
    std::vector<int> values;
    values[5] = values[11];

    void *ptr = values.data();                  // адрес первого элемента в values

    void *source       = ptr + 11 * sizeof(int); // откуда


    memcpy(
}
```

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями:

```
void* memcpy(void* destination, const void* source, size_t bytes_count);

int main() {
    std::vector<int> values;
    values[5] = values[11];

    void *ptr = values.data();                  // адрес первого элемента в values

    void *source      = ptr + 11 * sizeof(int); // откуда
    void *destination = ptr +  5 * sizeof(int); // куда

    memcpy(
}
```

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями:

```
void* memcpy(void* destination, const void* source, size_t bytes_count);

int main() {
    std::vector<int> values;
    values[5] = values[11];

    void *ptr = values.data();                  // адрес первого элемента в values

    void *source       = ptr + 11 * sizeof(int); // откуда
    void *destination  = ptr +  5 * sizeof(int); //   куда
    size_t bytes_count =            sizeof(int); // сколько байт
    memcpy(
}
```

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями:

```
void* memcpy(void* destination, const void* source, size_t bytes_count);

int main() {
    std::vector<int> values;
    values[5] = values[11];

    void *ptr = values.data();                  // адрес первого элемента в values

    void *source       = ptr + 11 * sizeof(int); // откуда
    void *destination  = ptr +  5 * sizeof(int); //   куда
    size_t bytes_count =            sizeof(int); // сколько байт
    memcpy(destination, source, bytes_count);    // копируем
}
```

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями

Самое сложное что встретим в алгоритмах:

- ***асимптотический анализ***
  - считать массив - **O(N)**
  - сравнить каждую пару чисел - **O(N*N)**

- ***bitonic/merge/radix sort***

Пререквизиты

Самое сложное что встретим в **C++** - арифметика над указателями

Самое сложное что встретим в алгоритмах:

- ***асимптотический анализ***
  - считать массив - **O(N)**
  - сравнить каждую пару чисел - **O(N*N)**

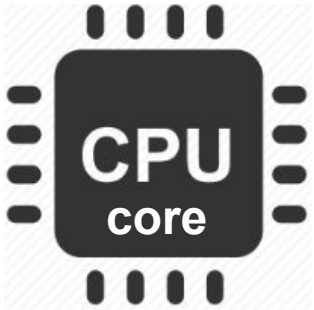
- ***bitonic/merge/radix sort***

Компьютер подойдет любой - код можно будет запустить на **CPU**

Глава 1: архитектура CPU

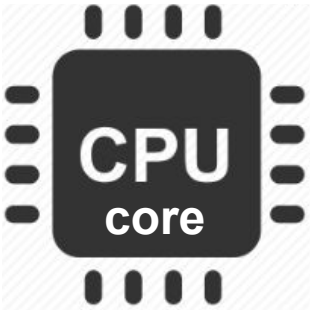
ALU, FPU, GFlops, FMA, Instruction Level Parallelism, Branch Prediction, Meltdown/Spectre, Cache Lines, Simultaneous Multithreading, Hyper-Threading, SIMD, SSE, AVX

Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

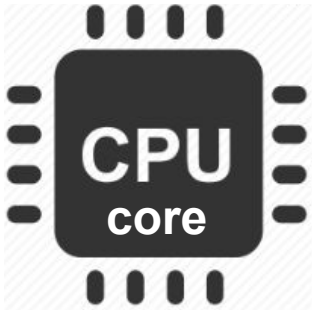
Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **это сколько?**

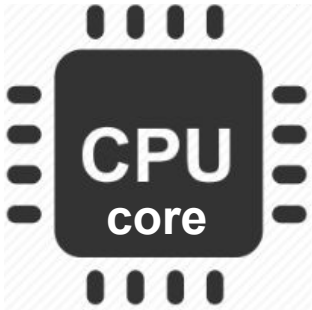
Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**

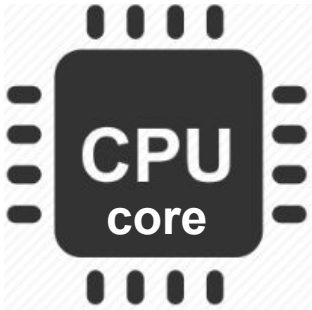
Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

Архитектура CPU

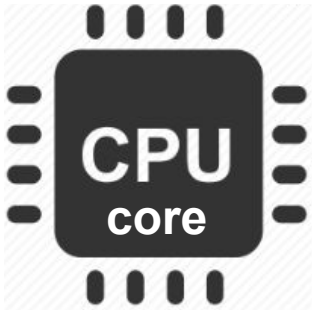


Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

А что такое **GFlops**?

Архитектура CPU

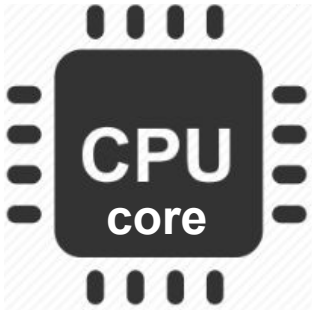


Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

А что такое **GFlops**?
Giga-Floating point operations per second
Условно у одного ядра **CPU** 10 GFlops

Архитектура CPU

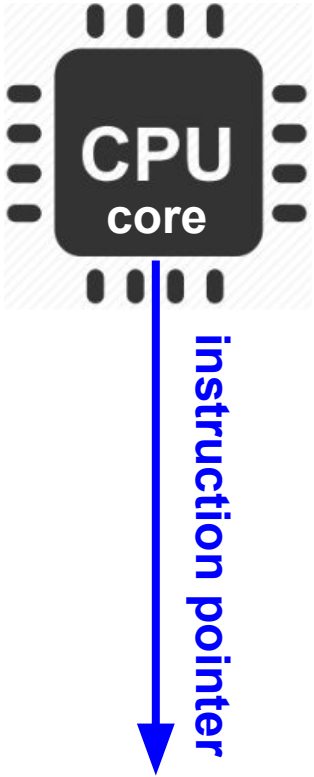


Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

А что такое **GFlops**?
Giga-Floating point operations per second
Условно у одного ядра **CPU** 10 GFlops
В два раза больше чем GHz, т.к. **FMA операция**: fused
multiply-add - две операции за такт (a * b + c)

Архитектура CPU

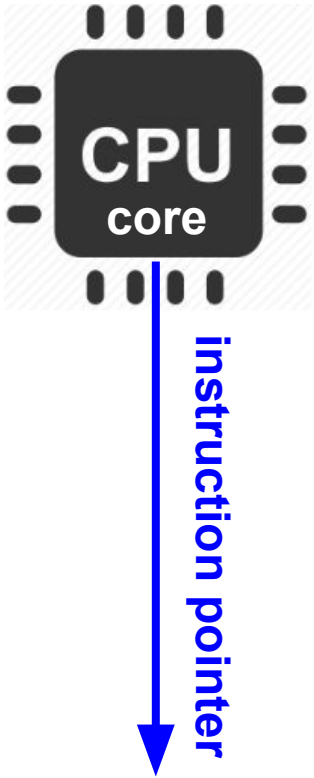


Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
c = a + b;
...
...
...
```

Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
c = a + b;
```

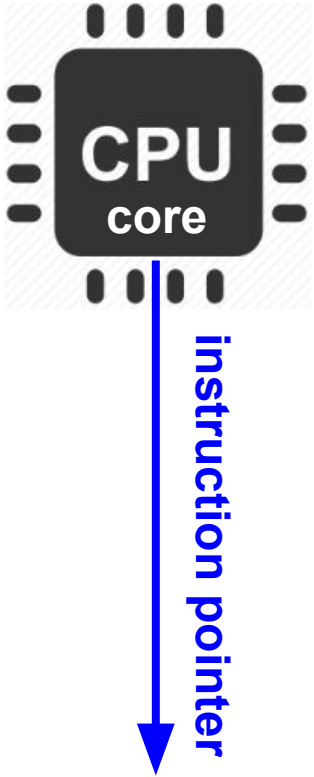
Из чего состоит такая команда?

...

...

...

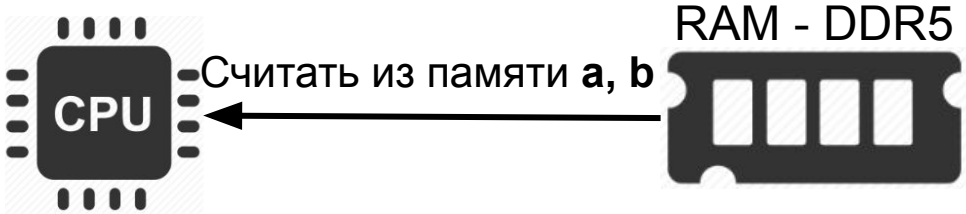
Архитектура CPU



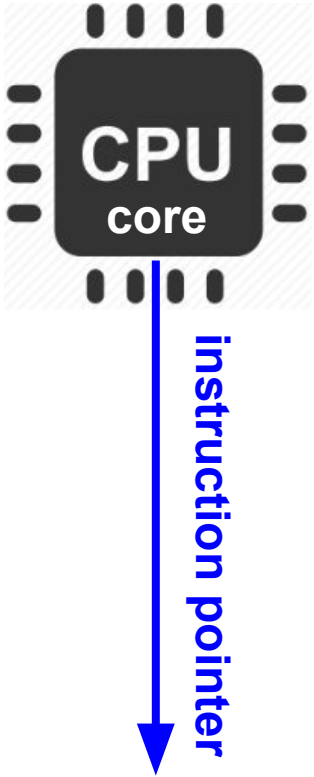
Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
c = a + b;
...
...
...
```



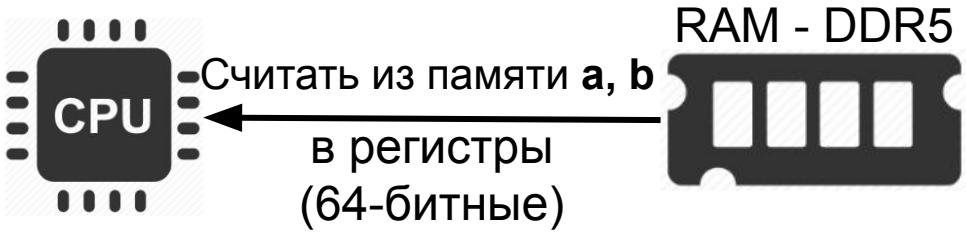
Архитектура CPU



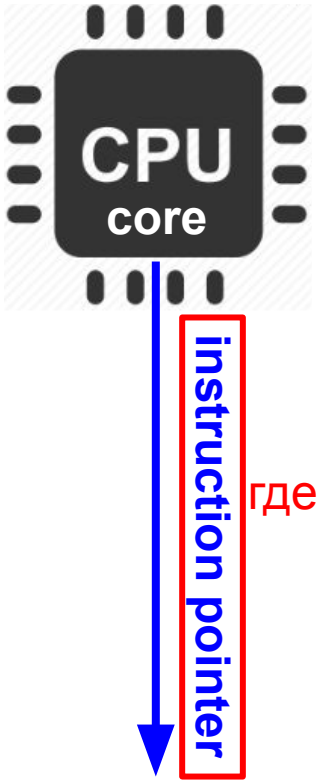
Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
c = a + b;
...
...
...
```



Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

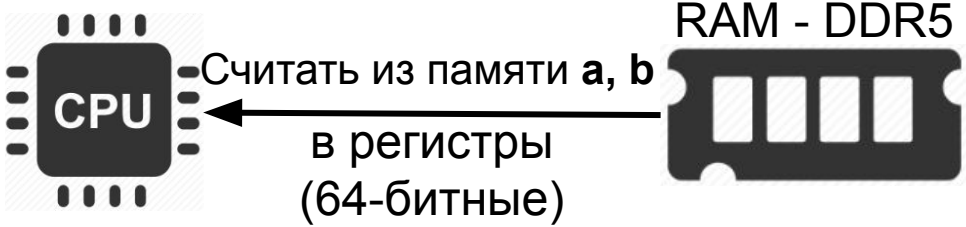
```
c = a + b;
```

где хранится?

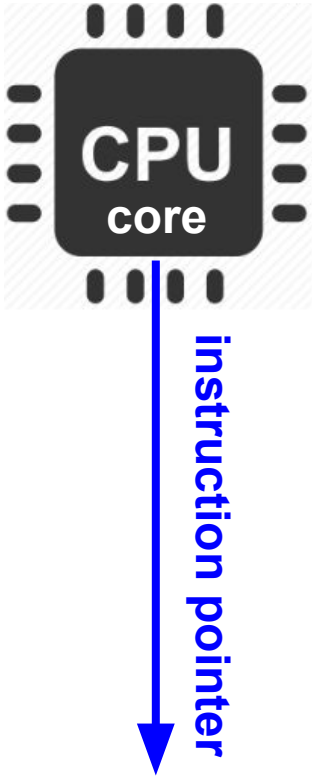
...

...

...



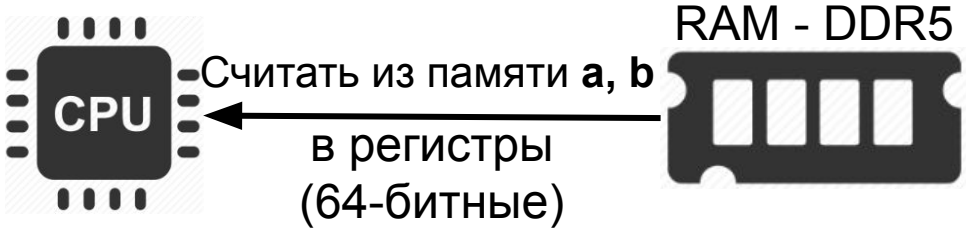
Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

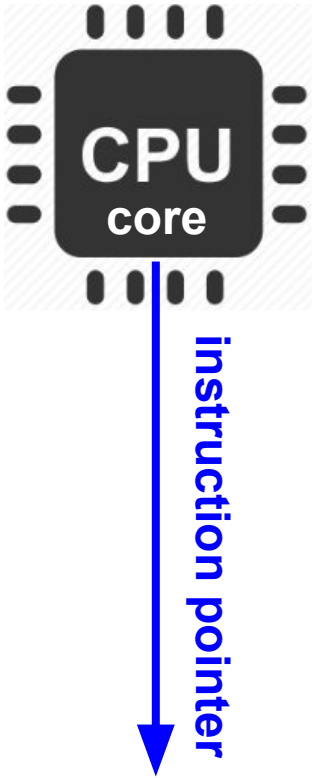
Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
c = a + b;
...
...
...
```



указатели в таких регистрах позволяют адресовать какой максимальный объем памяти?

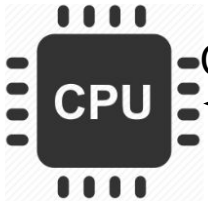
Архитектура CPU

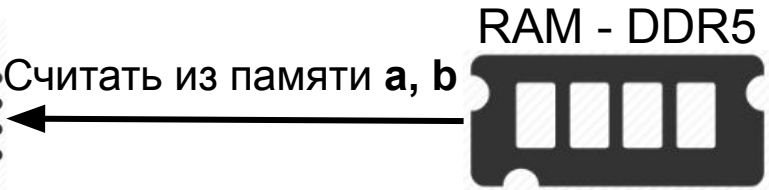


Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

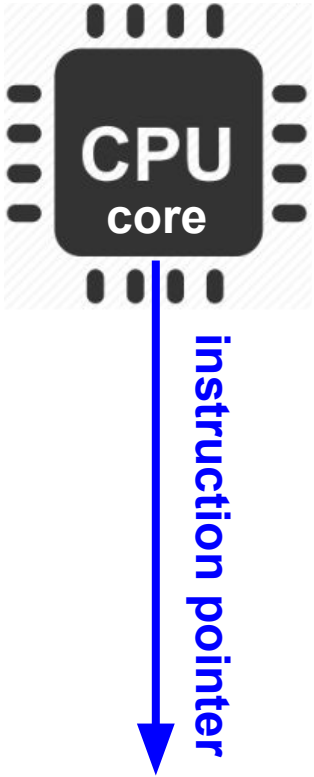
```
c = a + b;
...
...
...
```





За 1 наносекунду посчитать в регистрах c = a + b

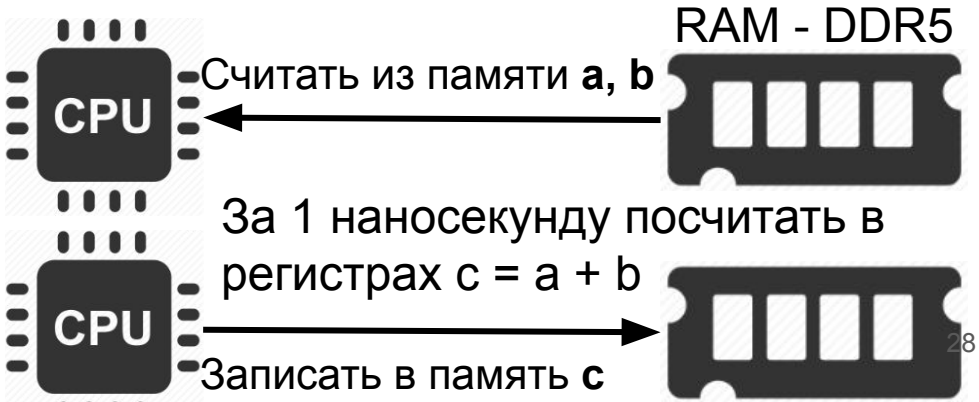
Архитектура CPU



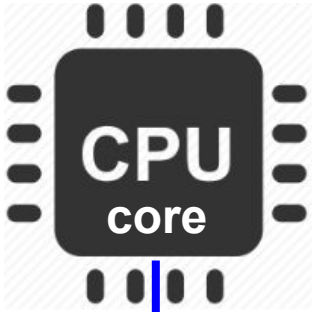
Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
c = a + b;
...
...
...
```



Архитектура CPU

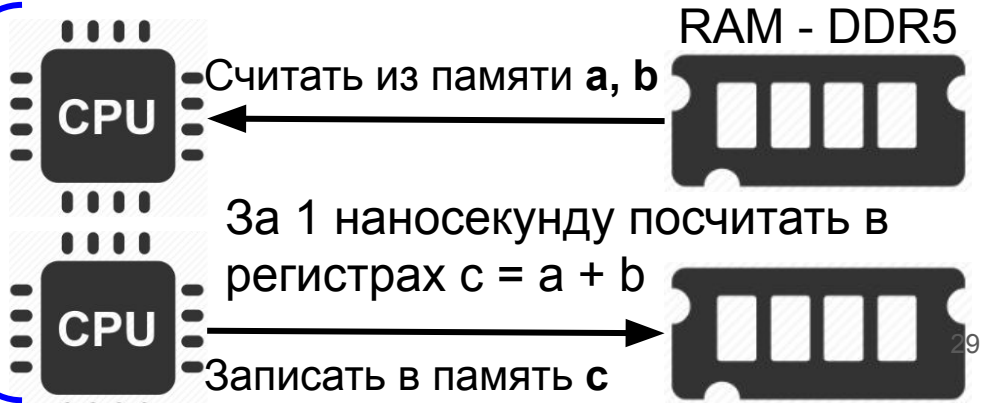


Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

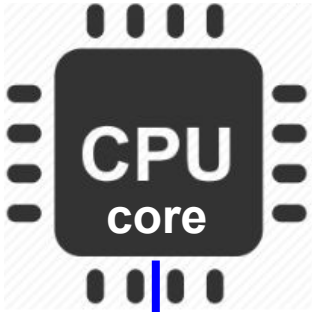
Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**


```
c = a + b;
```

Сколько времени?



Архитектура CPU

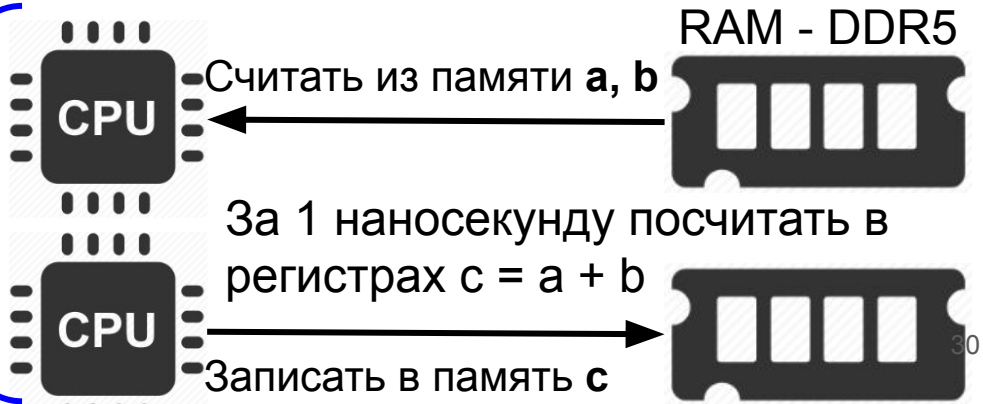


Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

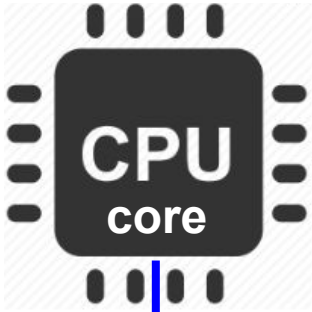
Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**


```
c = a + b;
```

Сколько времени?
DDR5 memory
bandwidth: 40 GB/s



Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

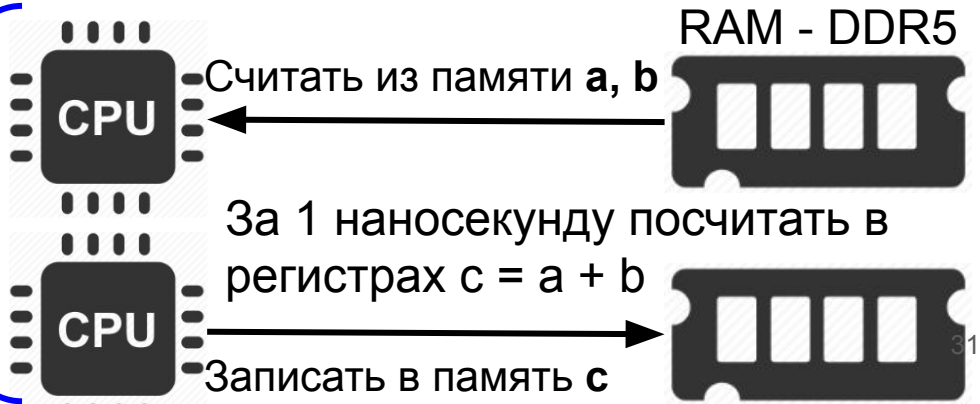
Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**


```
c = a + b;
```

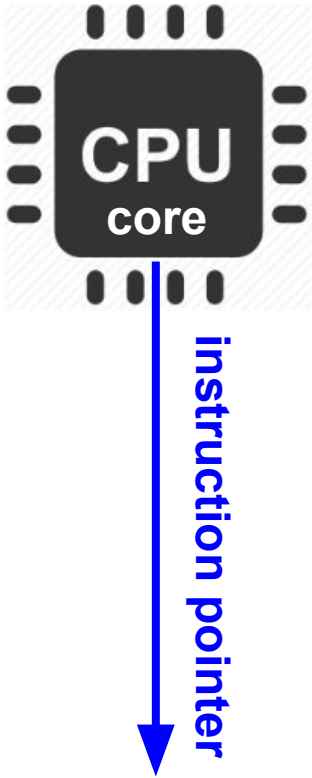
LATENCY!!!!11111

Сколько времени?

DDR5 memory
bandwidth: 40 GB/s



Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

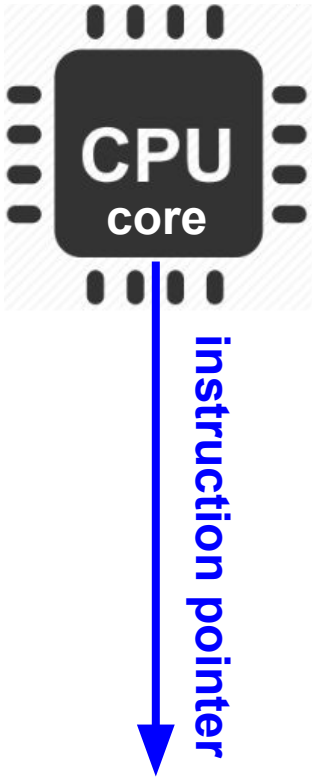
Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
c = a + b;
```

RAM - DDR5

- bandwidth
- latency

Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

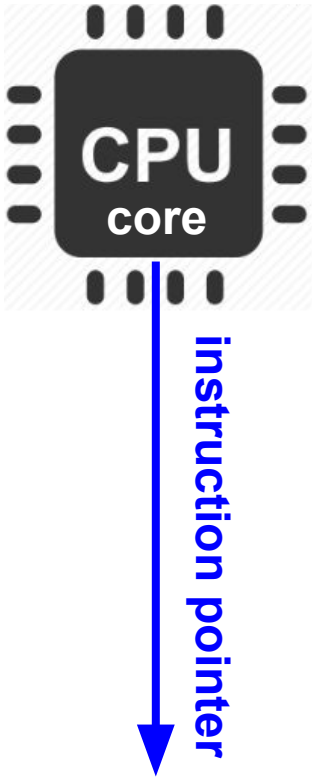
```
c = a + b;
f = d + e;
```

Как ускорить?

RAM - DDR5

- bandwidth
- latency

Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

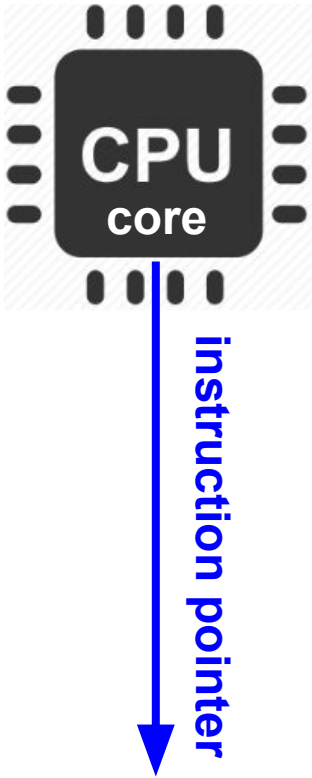
```
c = a + b;
f = d + e;
x = c + f;
```

Как ускорить?

RAM - DDR5

- bandwidth
- latency

Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
c = a + b;
f = d + e;
x = c + f;
```

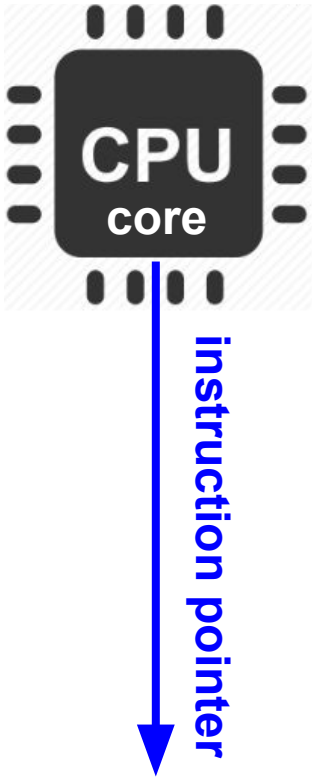
ILP - Instruction Level Parallelism

(out-of-order execution)

RAM - DDR5

- bandwidth
- latency

Архитектура CPU



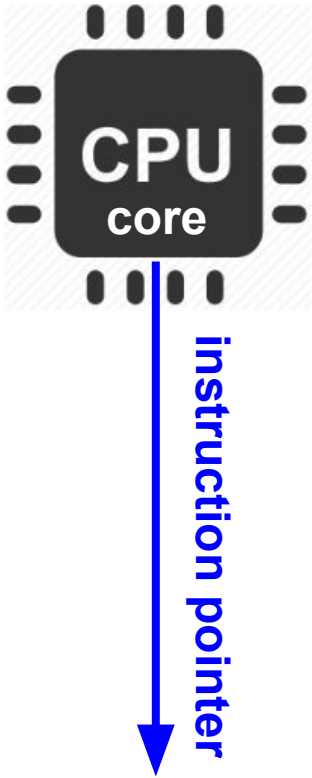
Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
if (a < b) {
    c = d + e;
} else {
    f = h + g;
}
```

ILP - Instruction Level Parallelism
Что делать?

Архитектура CPU



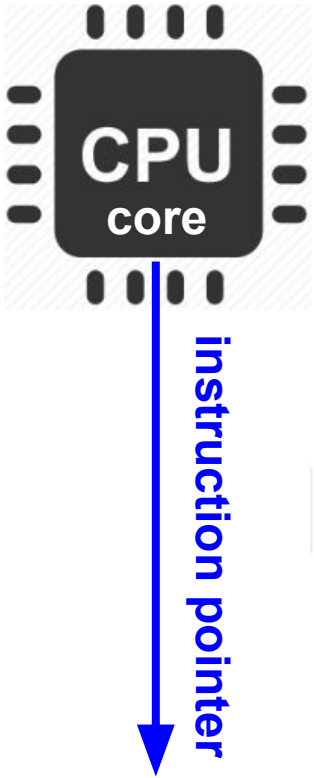
Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
if (a < b) {
    c = d + e;
} else {
    f = h + g;
}
```

ILP - Instruction Level Parallelism
+
Branch Prediction

Архитектура CPU



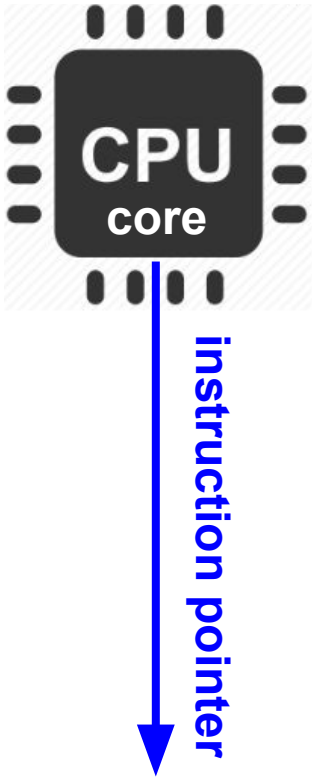
Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
for (int i = 0; i < n; ++i) {
    sum += a[i];
}
```

ILP - Instruction Level Parallelism
+
Branch Prediction

Архитектура CPU



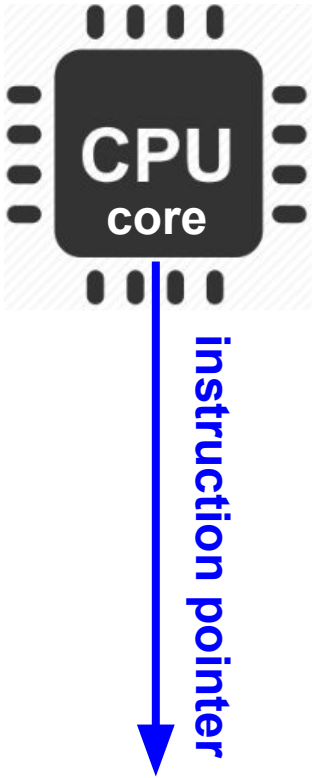
Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

```
start:
sum += a[i];
++i;
if (i < n)
    goto start;
```

ILP - Instruction Level Parallelism
+
Branch Prediction

Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

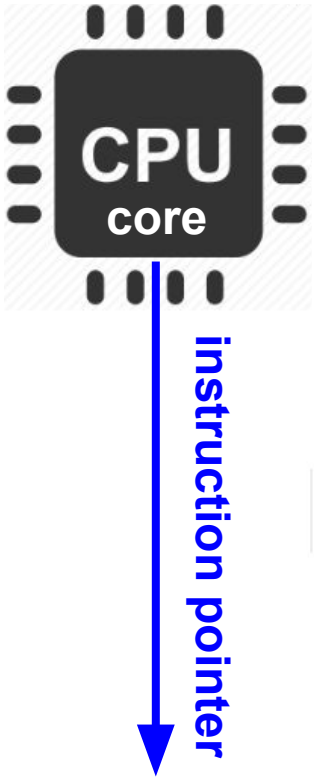
```
if (...) {
    a = b / value;
}
```

Деление на ноль!

А что будет если value=0?
А Branch Predictor не предупредили!!!

ILP - Instruction Level Parallelism
+
Branch Prediction

Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

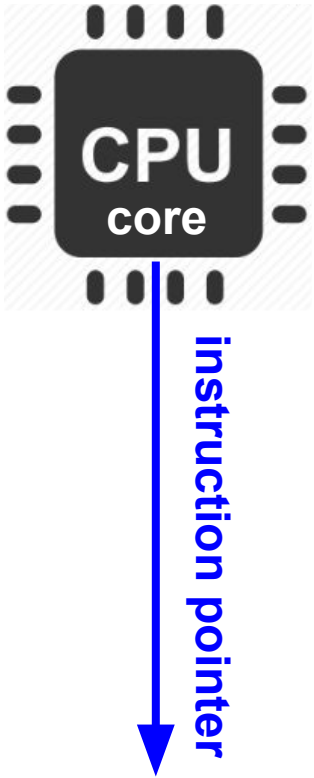
```
for (int i = 0; i < n; ++i) {
    sum += a[i];
}
```

Page Fault!

А что будет если i за пределами a[]?
А Branch Predictor не предупредили!!!

ILP - Instruction Level Parallelism
+
Branch Prediction

Архитектура CPU



Умеет считать:
- ALU - Arithmetic Logical Unit
- FPU - Floating-Point Unit

Делает это быстро! 5 GHz - **5 * 10^9 операций в секунду!**
На операцию тратит меньше **1 наносекунды!**

ILP - Instruction Level Parallelism
+
Branch Prediction

Значит ли это что ILP + Branch Prediction + Data Pre-Fetching не имеют побочных эффектов? Может тогда они и не нужны?!

Архитектура CPU
Meltdown/Spectre уязвимости

Архитектура CPU
Meltdown/Spectre уязвимости

```
myData          [N]   = [a, b, c, d]
myDataAndKernel[N+M] = [a, b, c, d, e, f, g, secret value]
```

Архитектура CPU
Meltdown/Spectre уязвимости


```
myData          [N]   = [a, b, c, d]
myDataAndKernel[N+M] = [a, b, c, d, e, f, g, secret value]

while (true) {
    if (myData[ myDataAndKernel[ptr] ]) { // много итераций ptr < N
        ...
    }
}
```

Архитектура CPU
Meltdown/Spectre уязвимости


```
myData         [N]   = [a, b, c, d]
myDataAndKernel[N+M] = [a, b, c, d, e, f, g, secret value]

while (true) {
    if (myData[ myDataAndKernel[ptr] ]) { // много итераций ptr < N
        ...                               // резко меняем   ptr = N+K
    }                                     //              + break
}
```

Архитектура CPU
Meltdown/Spectre уязвимости


```
myData         [N]   = [a, b, c, d]
myDataAndKernel[N+M] = [a, b, c, d, e, f, g, secret value]

while (true) {
    if (myData[ myDataAndKernel[ptr] ]) { // много итераций ptr < N
        ...                               // резко меняем   ptr = N+K
    }                                     //              + break
}

for (int i = 0; i < N; ++i) {

    ... = myData[i]; // считываем каждый элемент



}
```

Архитектура CPU
Meltdown/Spectre уязвимости


```
myData         [N]   = [a, b, c, d]
myDataAndKernel[N+M] = [a, b, c, d, e, f, g, secret value]

while (true) {
    if (myData[ myDataAndKernel[ptr] ]) { // много итераций ptr < N
        ...                               // резко меняем   ptr = N+K
    }                                     //              + break
}

for (int i = 0; i < N; ++i) {
    startTime = currentTime();
    ... = myData[i]; // считываем каждый элемент
    time = currentTime() - startTime;



}
```

Архитектура CPU
Meltdown/Spectre уязвимости


```
myData         [N]   = [a, b, c, d]
myDataAndKernel[N+M] = [a, b, c, d, e, f, g, secret value]

while (true) {
    if (myData[ myDataAndKernel[ptr] ]) { // много итераций ptr < N
        ...                               // резко меняем   ptr = N+K
    }                                     //              + break
}

for (int i = 0; i < N; ++i) {
    startTime = currentTime();
    ... = myData[i]; // считываем каждый элемент
    time = currentTime() - startTime;
    if (time < typicalTyme) {
        // i - индекс по которому сработал pre-fetching, значит ptr[N+K] = i
    }
}
```

Последствия:

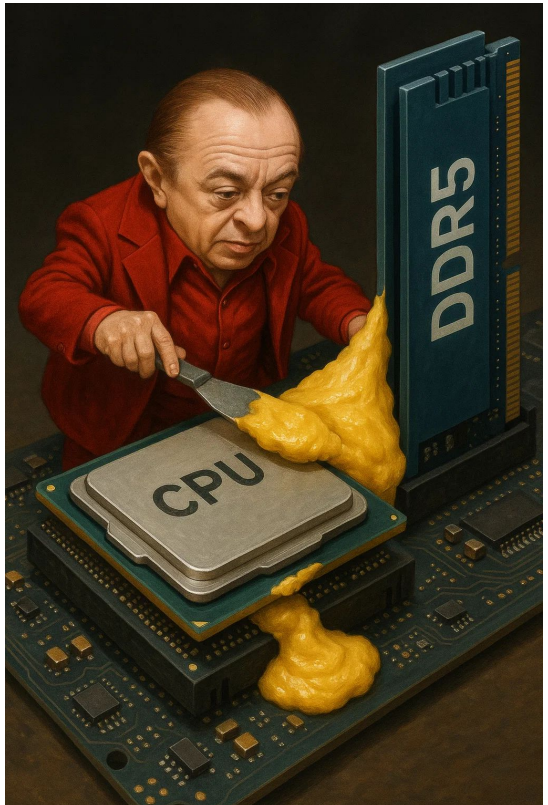
- в JavaScript точность performance.now(): стала 10^-4 секунд (ухудшили x20)
- микрокодом добавили ограничений в branch predictor

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс

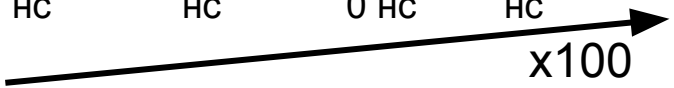
x100

Раз доступ к памяти настолько медленнее чем вычисления из регистров - промазываем **L1/L2/L3/L4 кэшами!**



Архитектура CPU

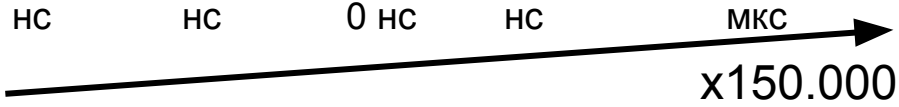
Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)
			???	
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	< 80 GB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс



* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс

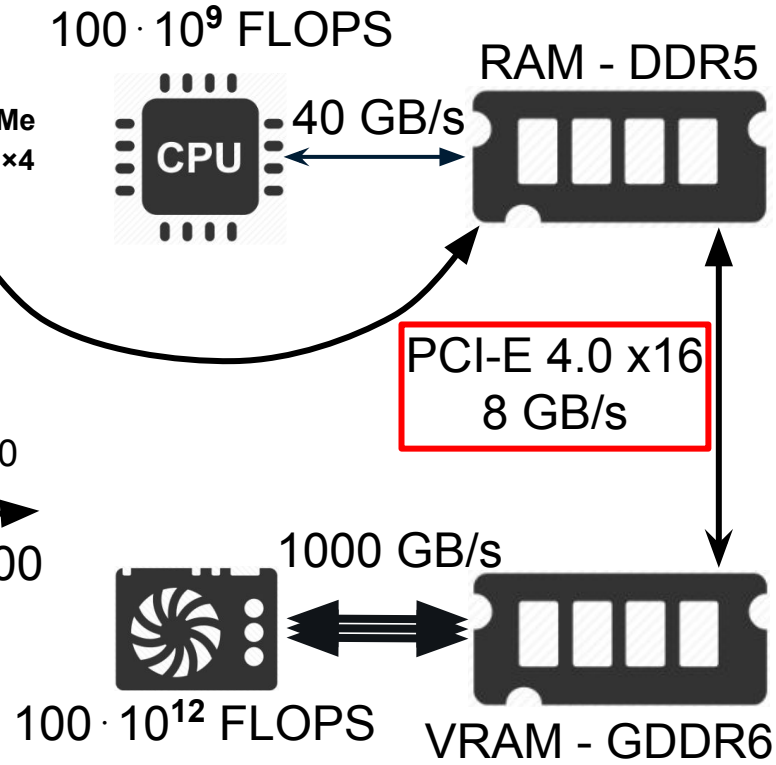


* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (1 eq)
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс

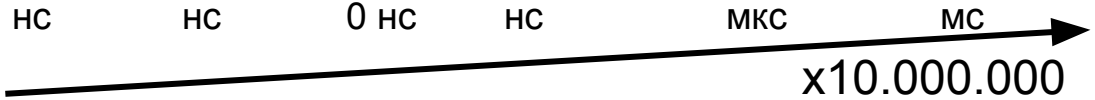
x150.000



* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

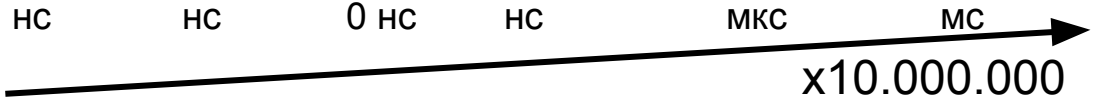
Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс



* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс

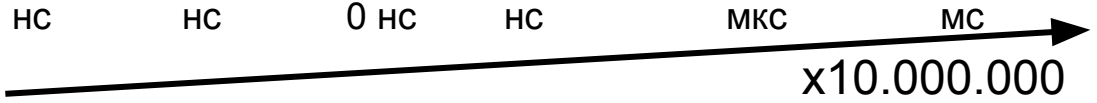


Это важный факт если вспомнить что в некоторых задачах объем данных **не влезает в RAM**!
См. алгоритмы во внешней памяти (**out-of-core**).

* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

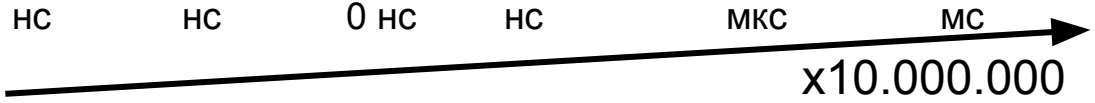
Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)	Ethernet 1 GbE
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s	120 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс	~0.2–2 мс



* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

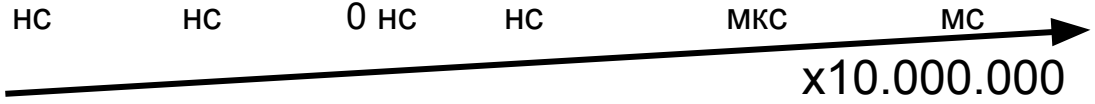
Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)	Ethernet 1 GbE	???
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s	120 MB/s	???
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс	~0.2–2 мс	???



* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)	Ethernet 1 GbE	Магнитная лента
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s	120 MB/s	300–400 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс	~0.2–2 мс	~10–120 сек до первого байта



* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Огромный объем!
Долгое хранение! (**20 лет**)
По промокоду **GPGPUCOURSE** - лилипут в подарок!

Тип	L1 cache	L2 cache	L3 cache	Магнитная лента
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	300–400 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~10–120 сек до первого байта

* числа условные, все совпадени

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)	Ethernet 1 GbE	Магнитная лента
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s	120 MB/s	300–400 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс	~0.2–2 мс	~10–120 сек до первого байта

Можем ли сделать HDD бесконечно долгим для хранения?

* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)	Ethernet 1 GbE	Магнитная лента
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s	120 MB/s	300–400 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс	~0.2–2 мс	~10–120 сек до первого байта

Можем ли сделать HDD бесконечно долгим для хранения?
Можем ли сделать HDD гораздо быстрее? (**bandwidth**)

* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)	Ethernet 1 GbE	Магнитная лента
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s	120 MB/s	300–400 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс	~0.2–2 мс	~10–120 сек до первого байта

Можем ли сделать HDD бесконечно долгим для хранения?
Можем ли сделать HDD гораздо быстрее? (**bandwidth**) **RAID!**

* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Тип	L1 cache	L2 cache	L3 cache	DDR5 (2-channels)	SSD NVMe PCIe 4.0×4 (seq)	HDD 7200 rpm (seq)	Ethernet 1 GbE	Магнитная лента
Пропускная способность (bandwidth)	60–120 GB/s per-core	30–80 GB/s per-core	10–40 GB/s per-core	80 GB/s	7 GB/s	250 MB/s	120 MB/s	300–400 MB/s
Задержка доступа (latency)	~0.5–1 нс	~2–5 нс	~10–20 нс	~60–120 нс	~70–150 мкс	~8–12 мс	~0.2–2 мс	~10–120 сек до первого байта

Можем ли сделать HDD бесконечно долгим для хранения?
Можем ли сделать HDD гораздо быстрее? (**bandwidth**) **RAID!**
Можем ли сделать HDD гораздо быстрее? (**latency**)

* числа условные, все совпадения с реальностью - случайны

Архитектура CPU

Что вы думаете про этот код?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```

Архитектура CPU

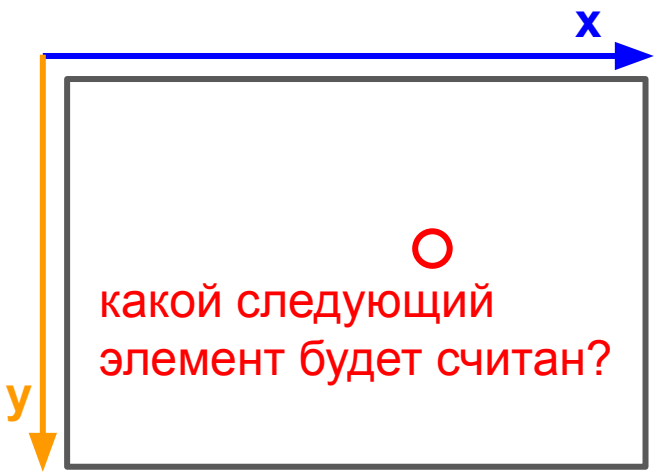
Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```

Архитектура CPU

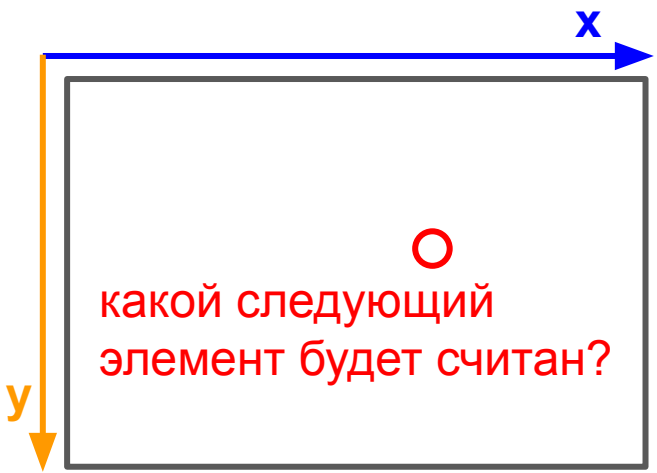
```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```




```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```

Архитектура CPU

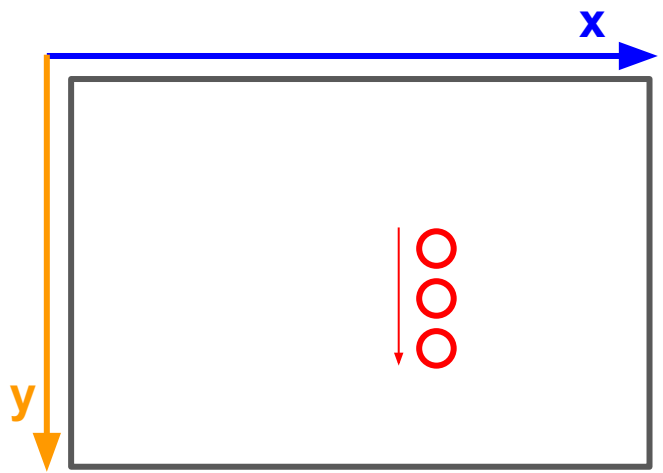
```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```




```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```

Архитектура CPU

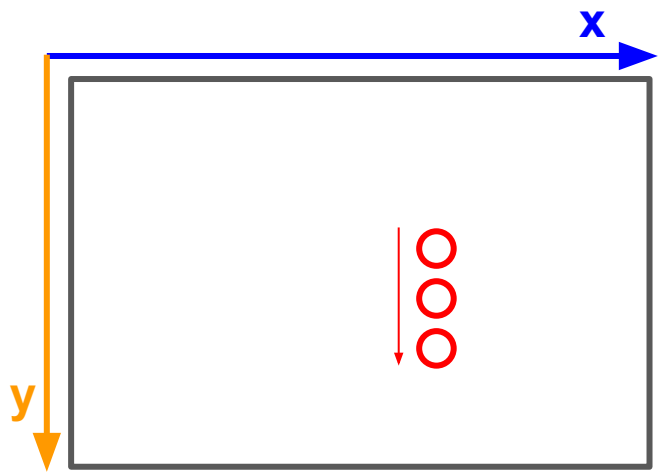
```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```



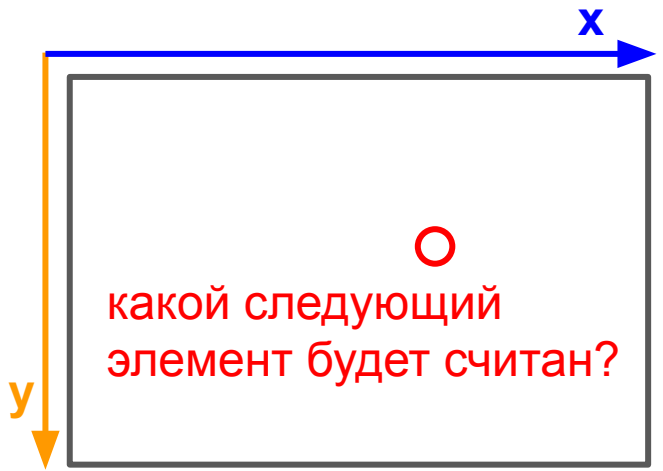

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```

Архитектура CPU

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```



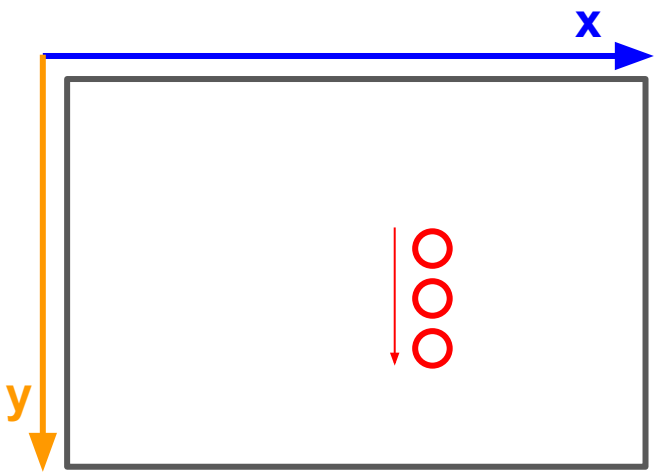

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



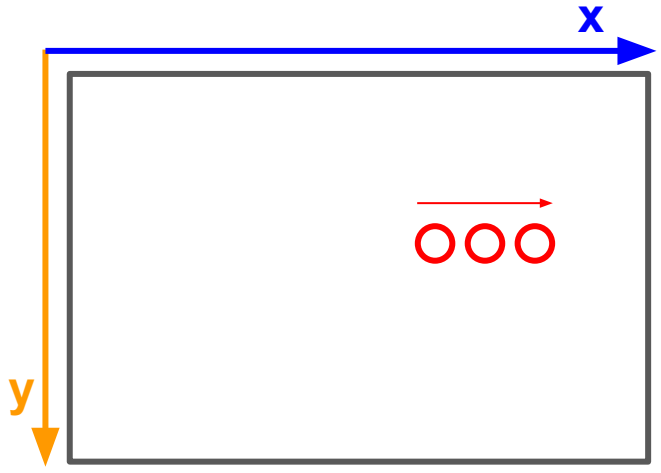
Архитектура CPU

Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```



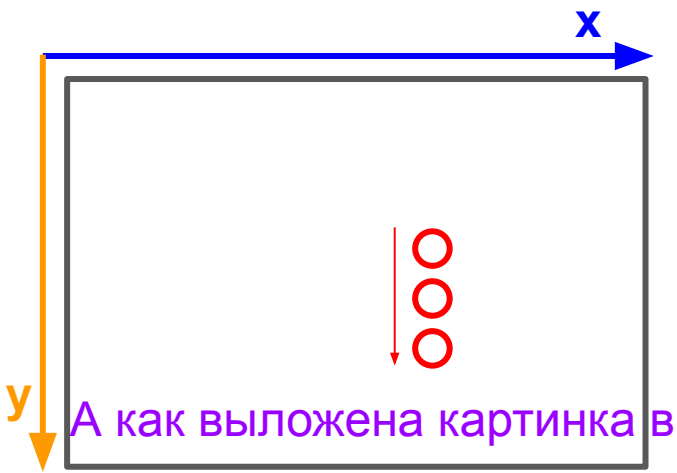

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



Архитектура CPU

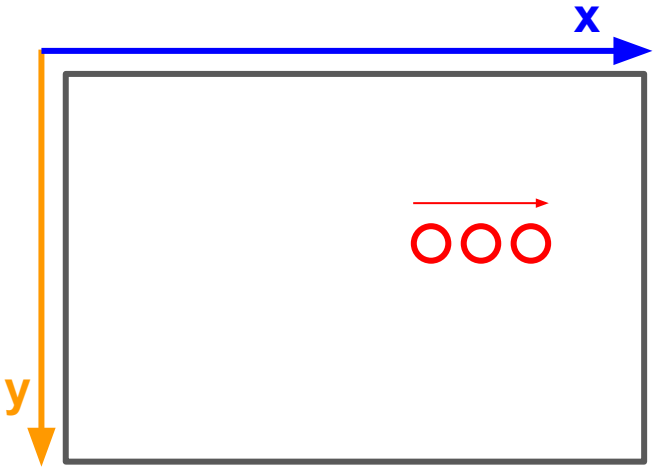
Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```



А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



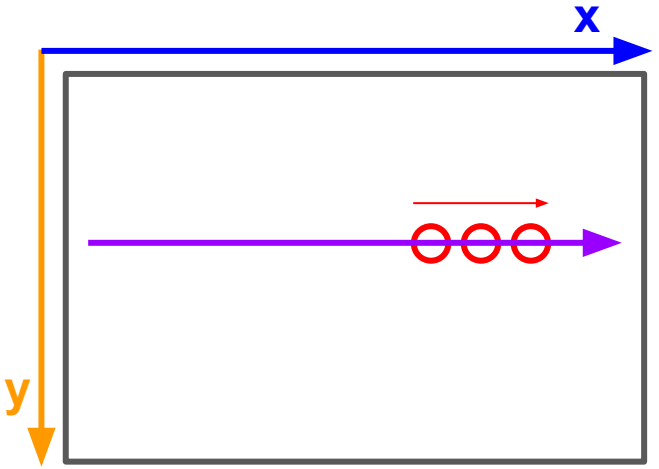
Архитектура CPU

Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```

А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



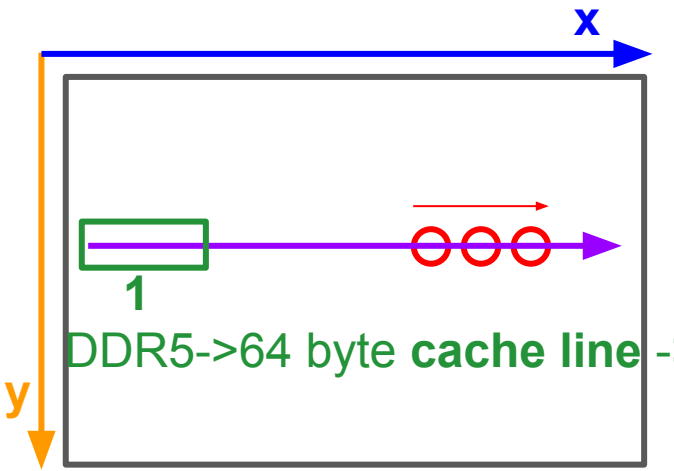
Архитектура CPU

Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```

А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



DDR5->64 byte **cache line** -> **CPU**

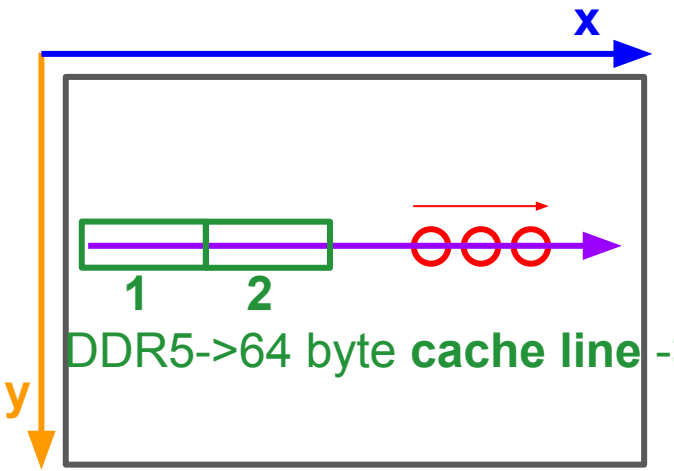
Архитектура CPU

Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```

А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



DDR5->64 byte **cache line** -> **CPU**

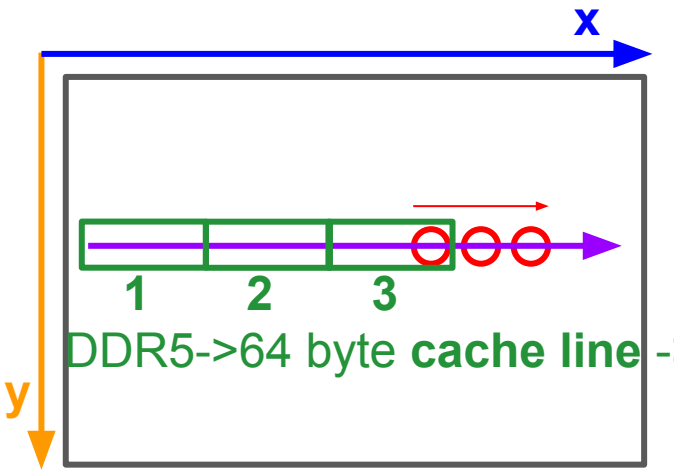
Архитектура CPU

Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```

А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



DDR5->64 byte **cache line** -> **CPU**

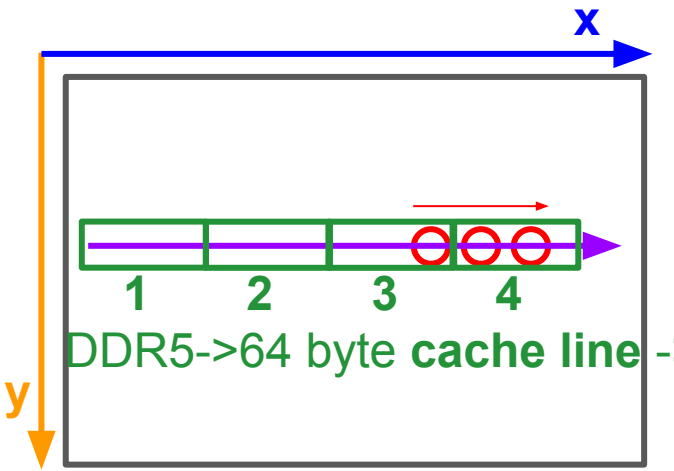
Архитектура CPU

Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```

А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



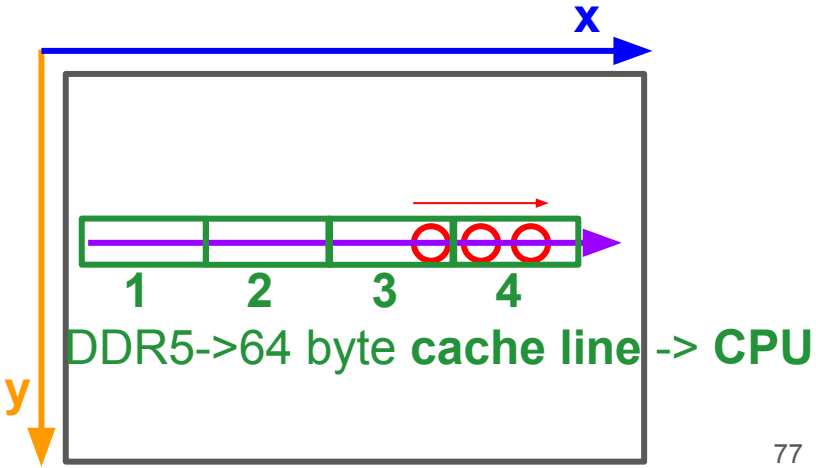
DDR5->64 byte **cache line** -> **CPU**

Архитектура CPU

Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```

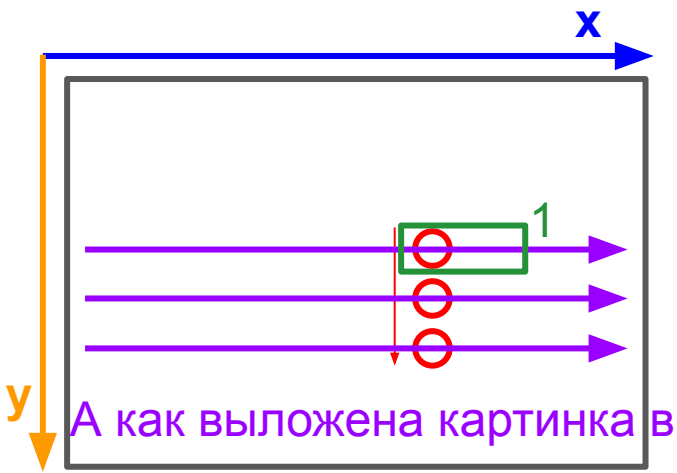

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



Архитектура CPU

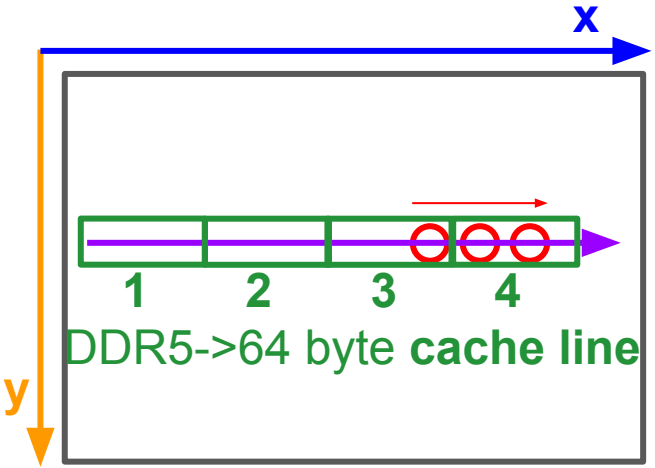
Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```



А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```

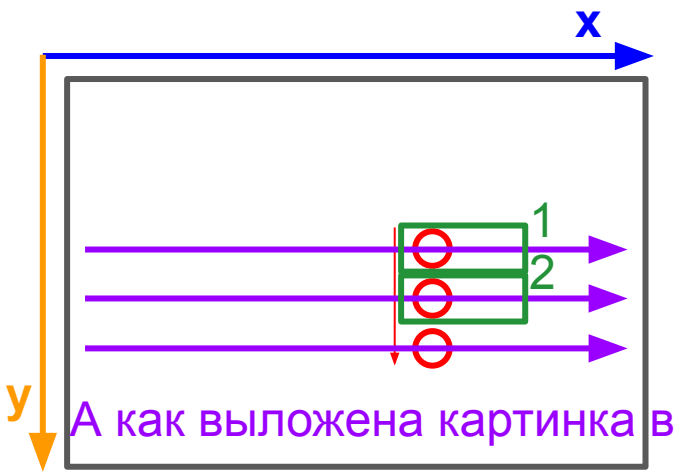


DDR5->64 byte **cache line** -> **CPU**

Архитектура CPU

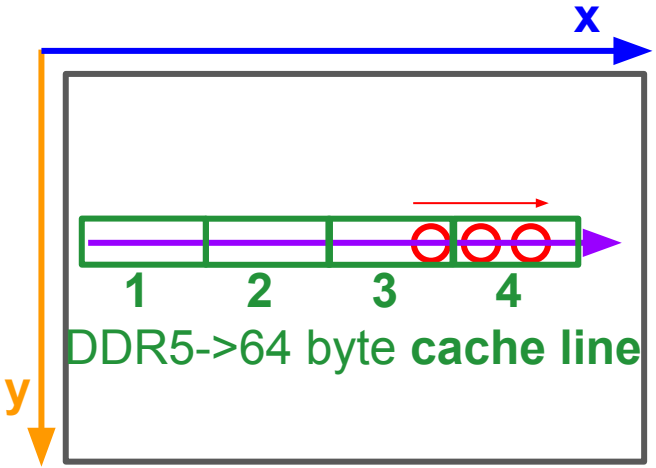
Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```



А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```

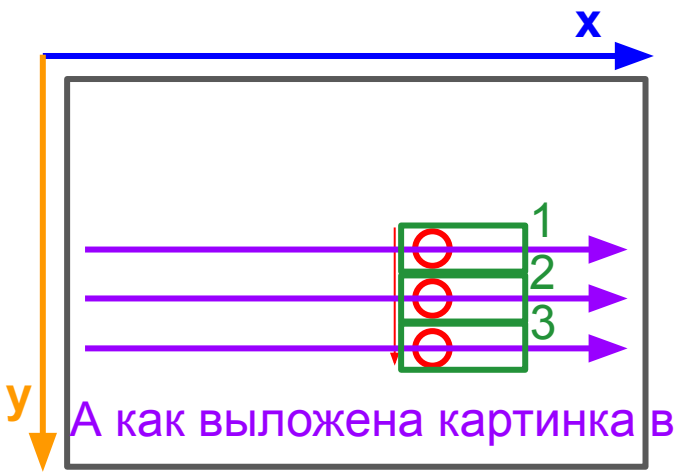


DDR5->64 byte **cache line** -> **CPU**

Архитектура CPU

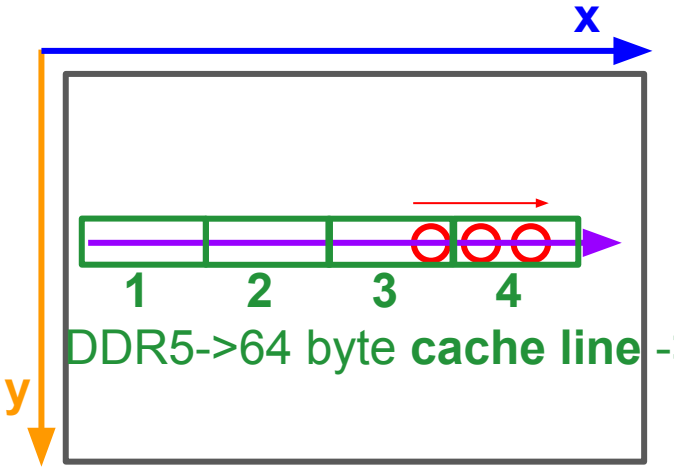
Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```



А как выложена картинка в памяти?

```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```



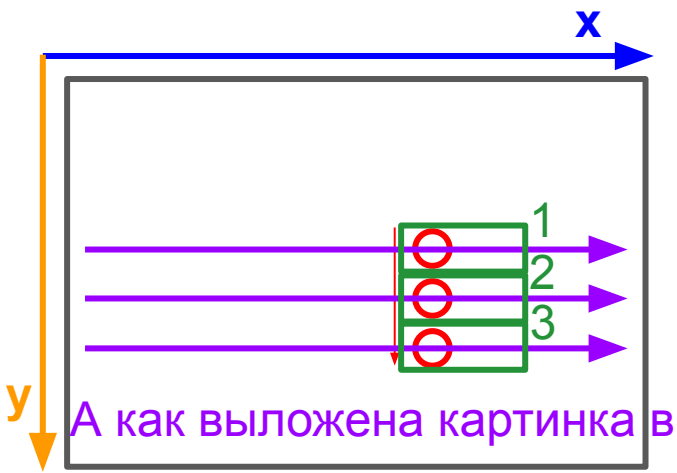
DDR5->64 byte **cache line** -> **CPU**

Архитектура CPU

Какая предельная разница в скорости?

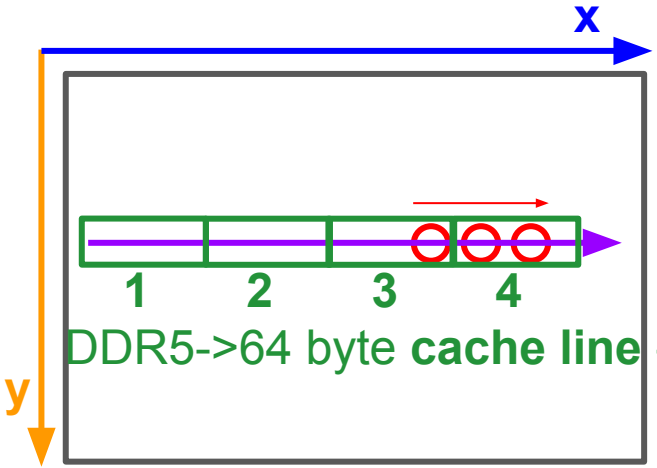
Какой код выбрали бы вы?

```
sum = 0;
for (int x = 0; x < width; ++x) {
    for (int y = 0; y < height; ++y) {
        sum += image[y * width + x];
    }
}
```



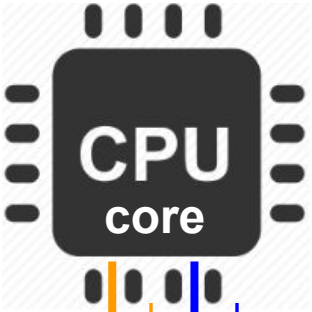
А как выложена картинка в памяти?

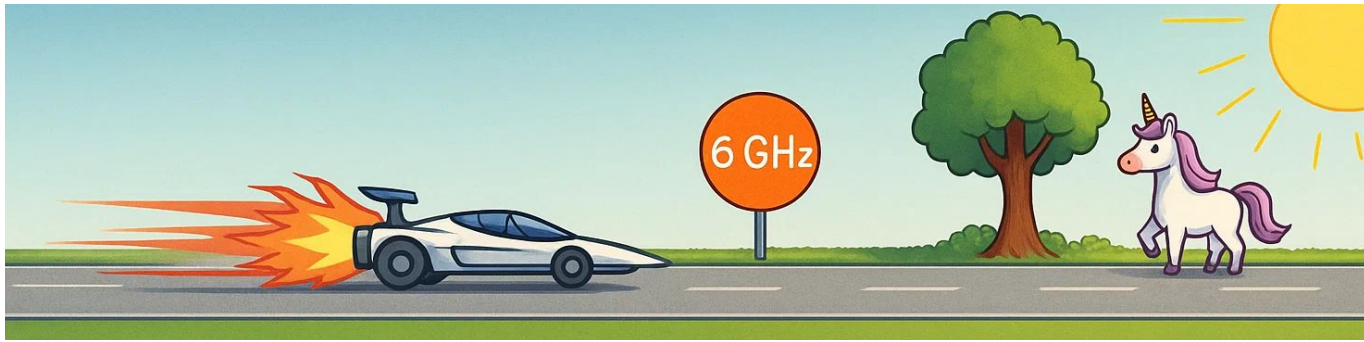
```
sum = 0;
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        sum += image[y * width + x];
    }
}
```

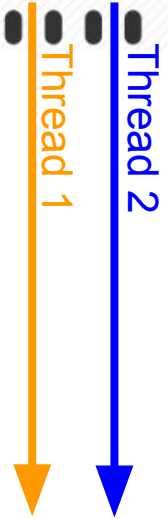


DDR5->64 byte **cache line** -> **CPU**

Архитектура CPU: многопоточность

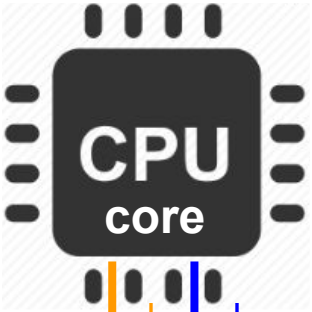


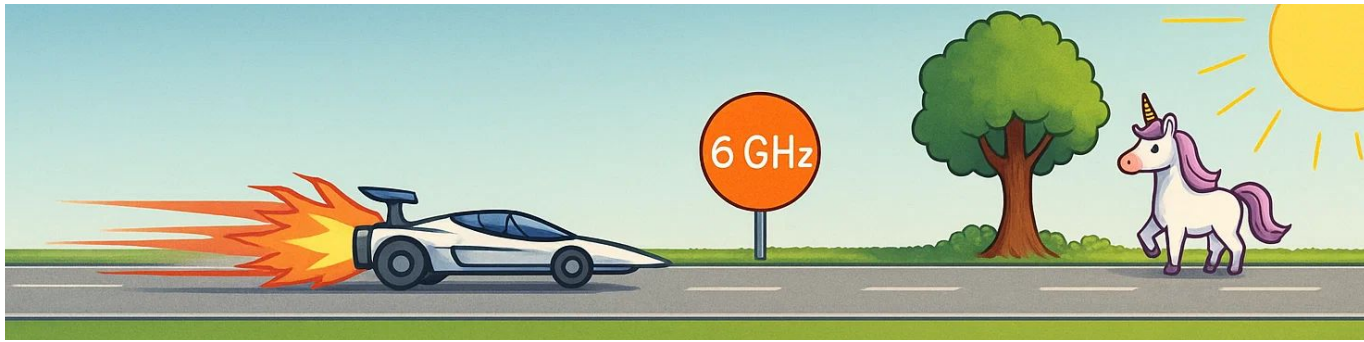


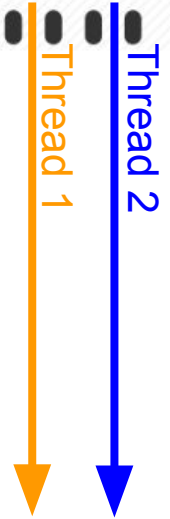


Многопоточность благодаря переключению контекста!

Архитектура CPU: многопоточность



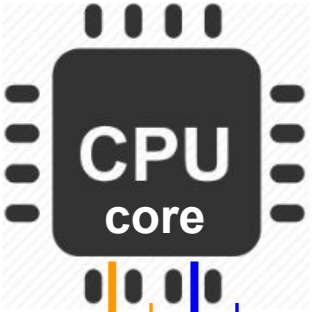


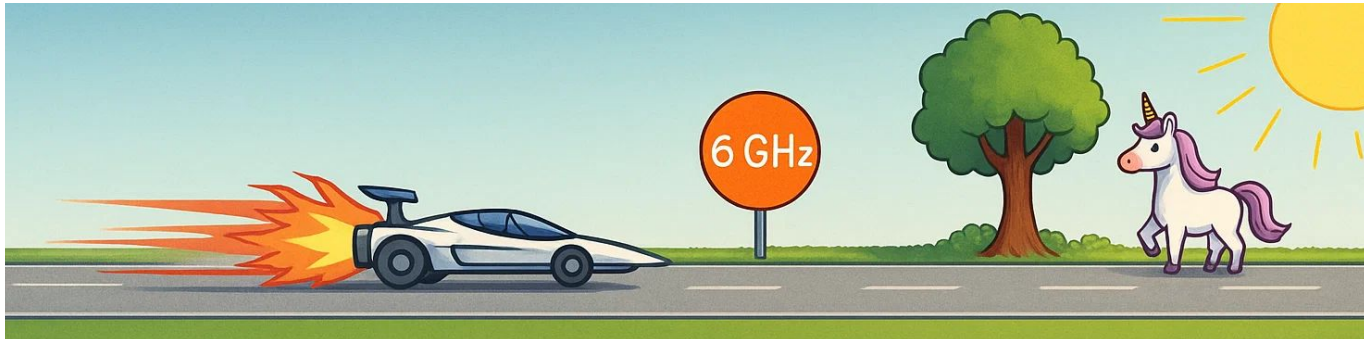


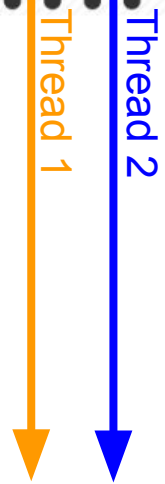
Многопоточность благодаря переключению контекста!

Что нужно перещелкнуть в состоянии ЦПУ ядра для другого потока?

Архитектура CPU: многопоточность



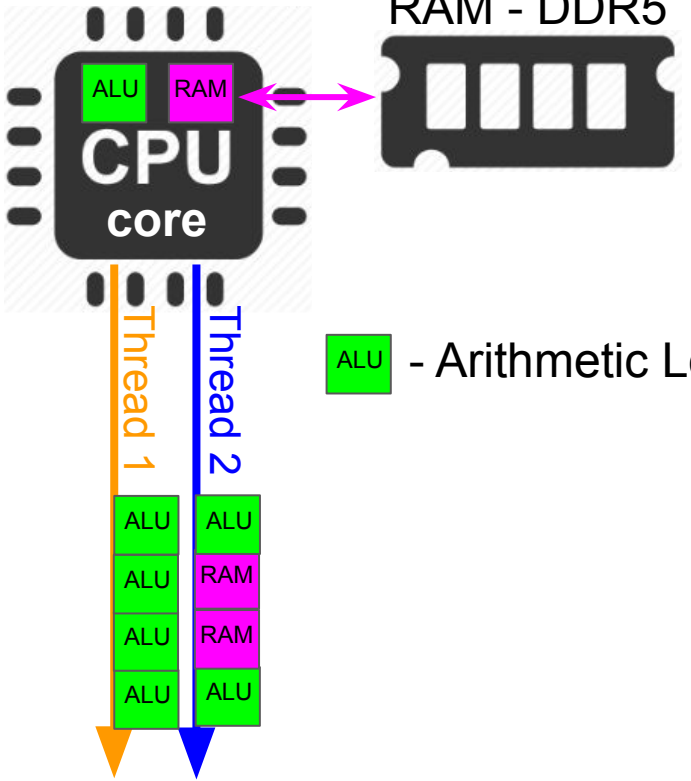




Многопоточность благодаря переключению контекста!

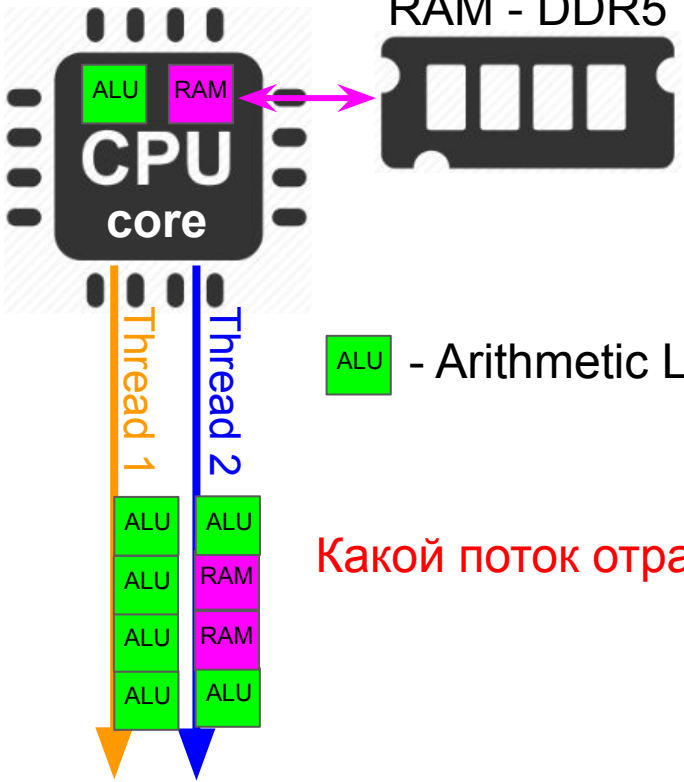
Context switch:
- Обновляет Instruction pointer (указатель на строку кода)
- Подгружает значения регистров процессора (откуда?)

Архитектура CPU: Hyper-Threading, SMT



ALU - Arithmetic Logical Unit

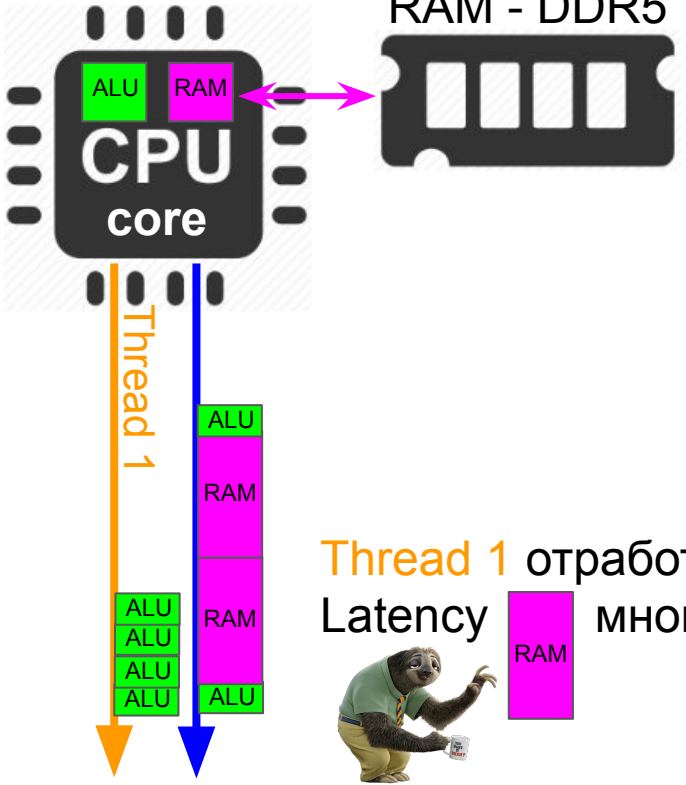
Архитектура CPU: Hyper-Threading, SMT

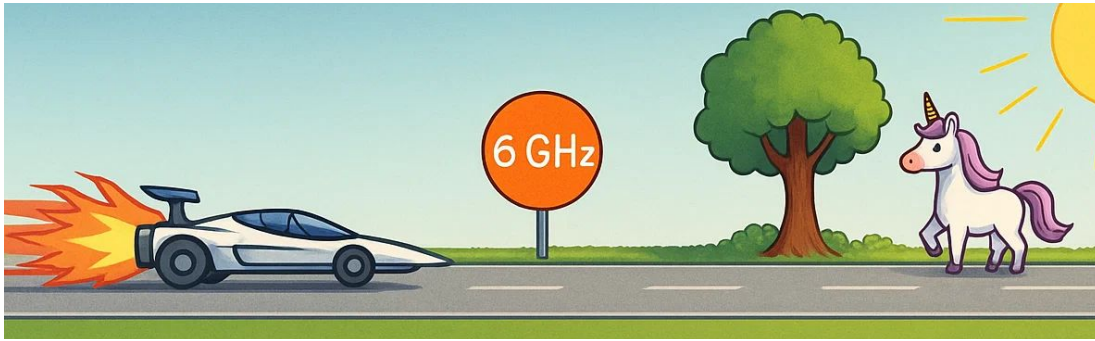


ALU - Arithmetic Logical Unit

Какой поток отработает быстрее?

Архитектура CPU: Hyper-Threading, SMT

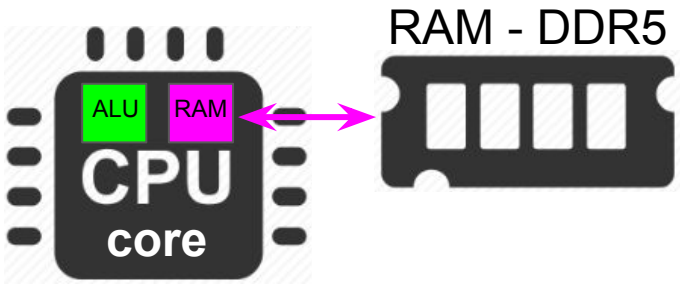


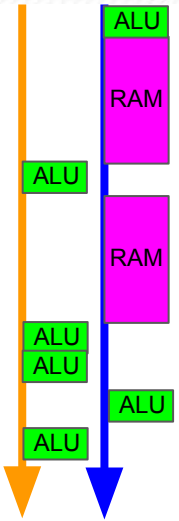


Thread 1 отработает быстрее!

Latency RAM много больше чем у ALU

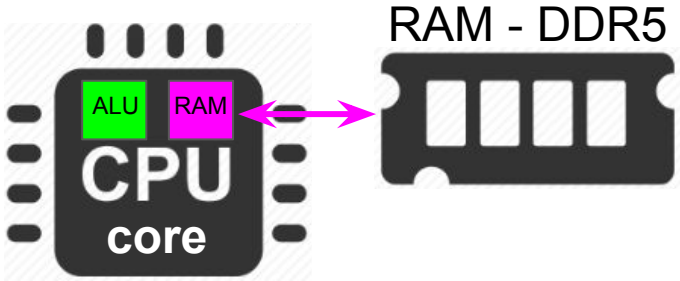
Архитектура CPU: Hyper-Threading, SMT

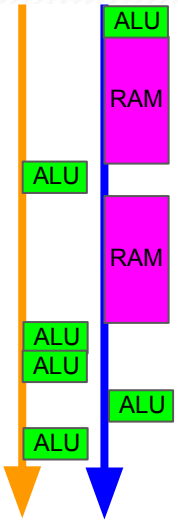




Какое суммарное время работы двух таких потоков?

Архитектура CPU: Hyper-Threading, SMT

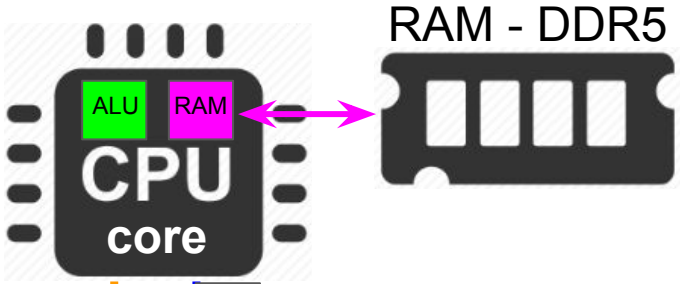


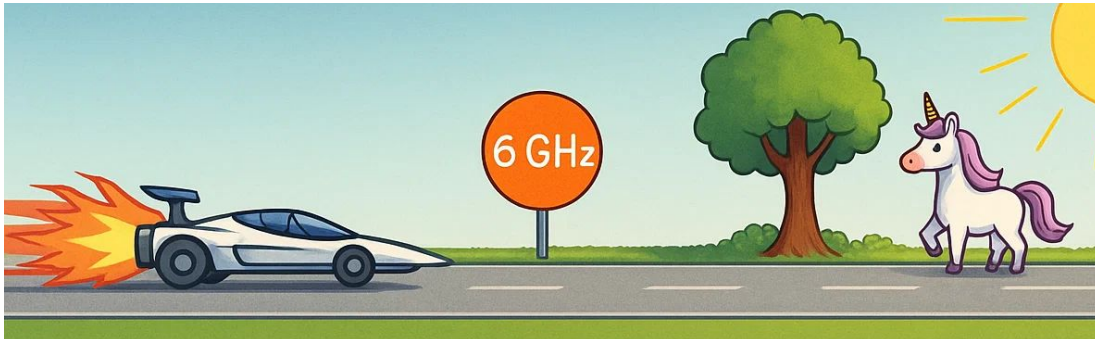


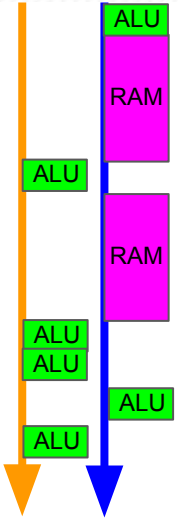
Какое суммарное время работы двух таких потоков?

6x ALU +2x RAM

Архитектура CPU: Hyper-Threading, SMT





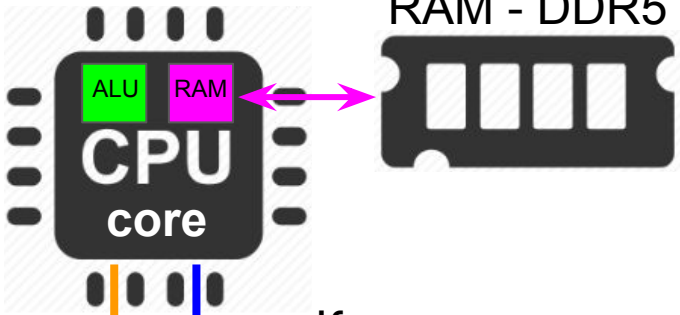


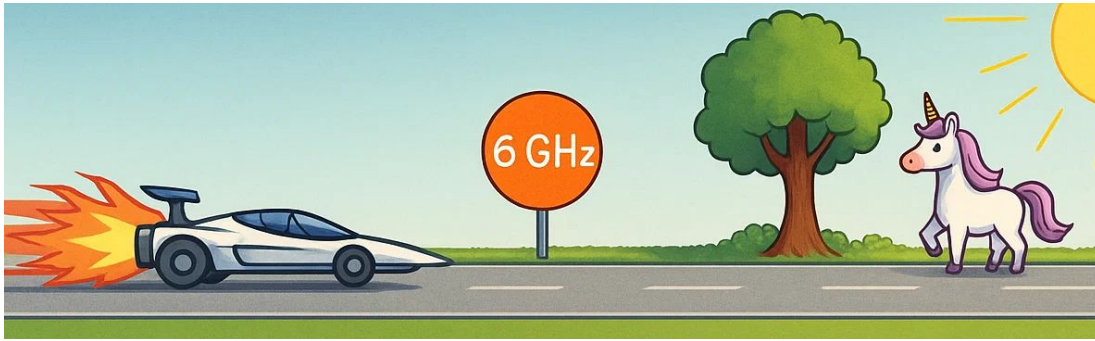
Какое суммарное время работы двух таких потоков?

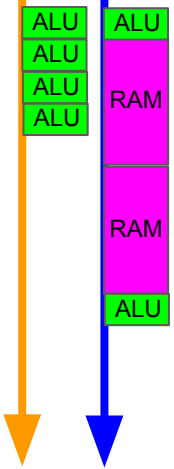
6x ALU +2x RAM

Можно ли быстрее?

Архитектура CPU: Hyper-Threading, SMT







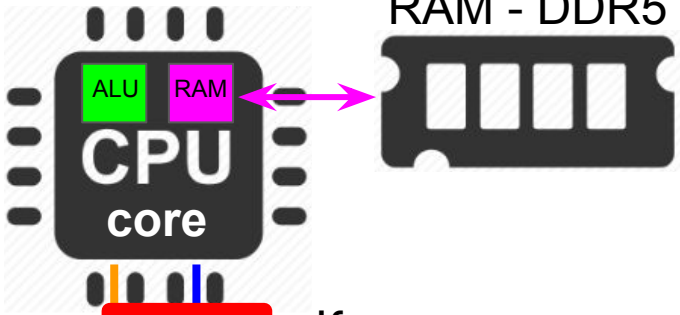
Какое суммарное время работы двух таких потоков?

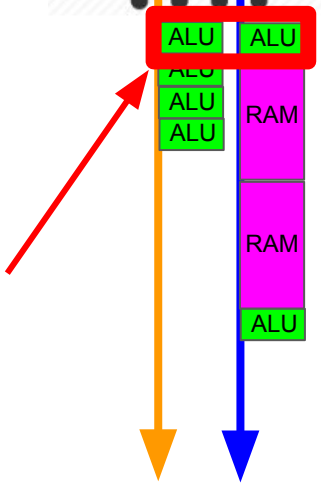
6x ALU +2x RAM

Можно ли делать ALU пока ждем RAM ?

2x ALU +2x RAM

Архитектура CPU: Hyper-Threading, SMT





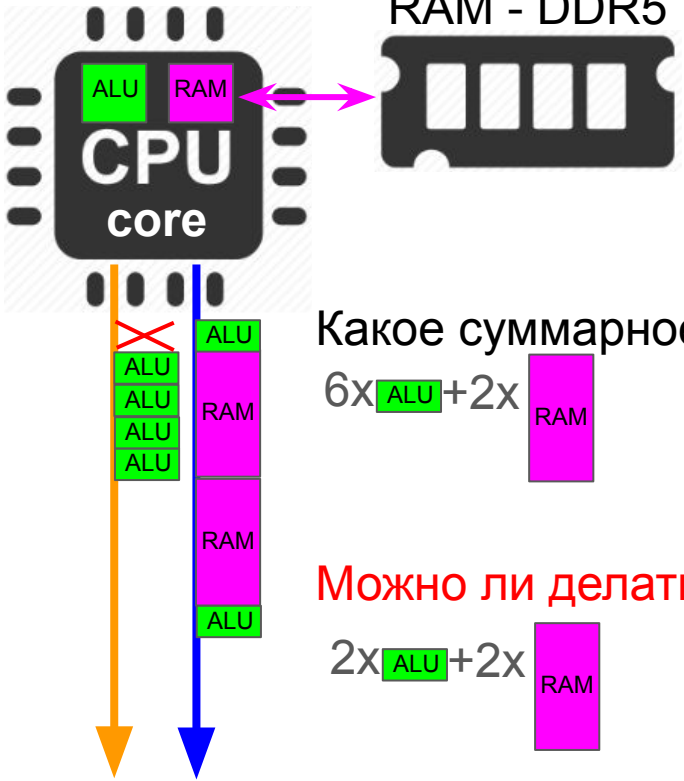
Какое суммарное время работы двух таких потоков?

6x ALU +2x RAM

Можно ли делать ALU пока ждем RAM ?

2x ALU +2x RAM

Архитектура CPU: Hyper-Threading, SMT



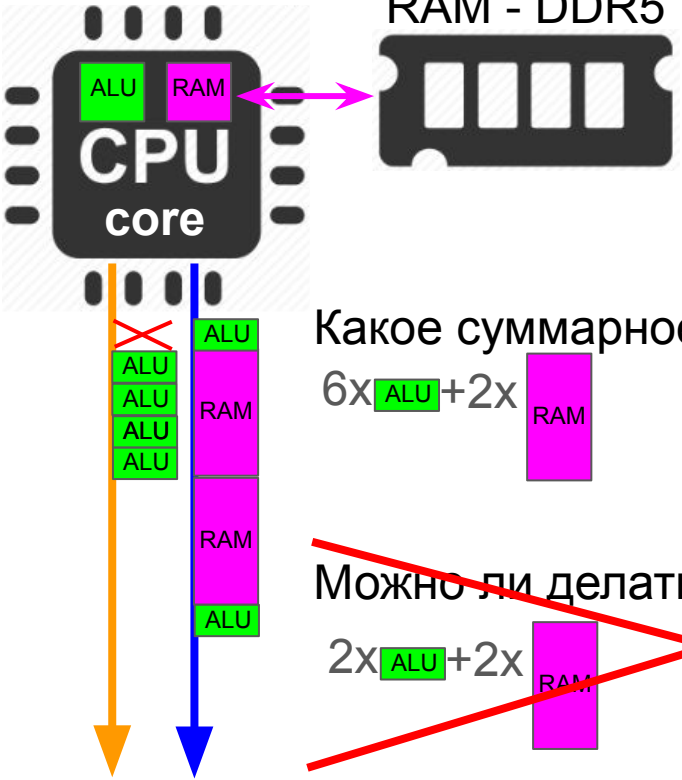
Какое суммарное время работы двух таких потоков?

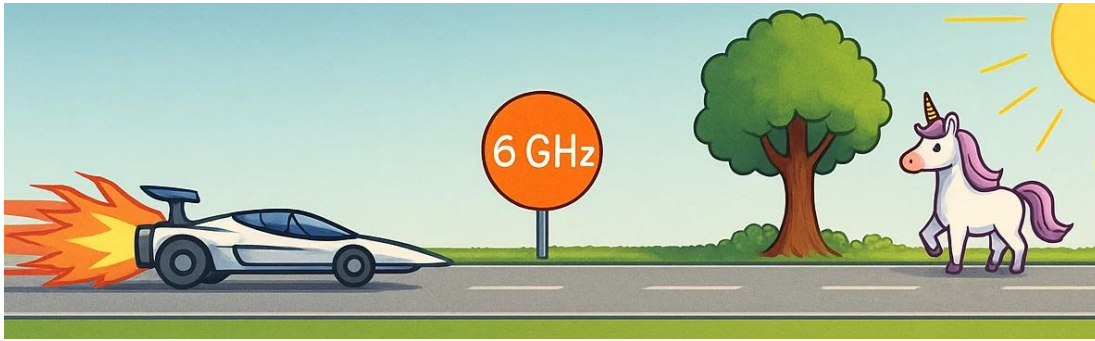
6x ALU +2x RAM

Можно ли делать ALU пока ждем RAM ?

2x ALU +2x RAM

Архитектура CPU: Hyper-Threading, SMT





Какое суммарное время работы двух таких потоков?

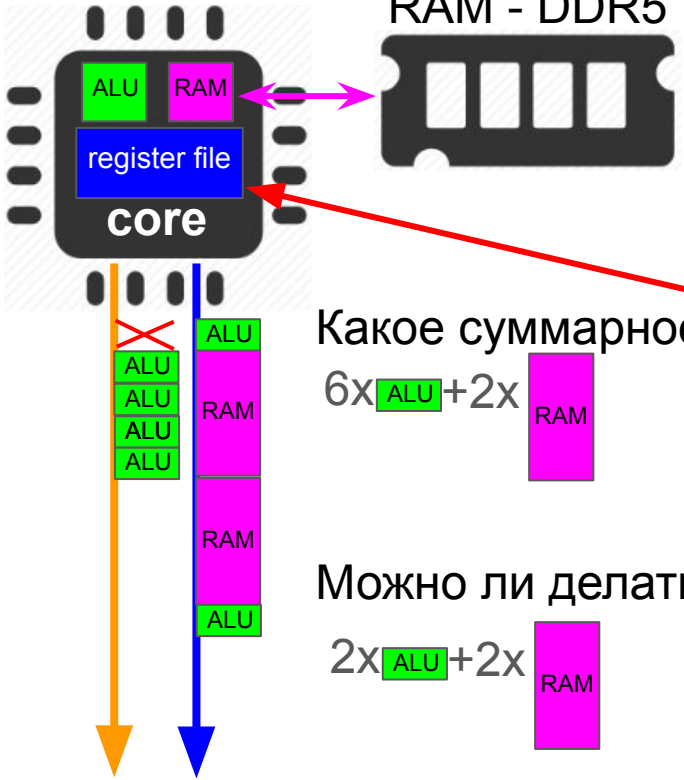
6x ALU +2x RAM

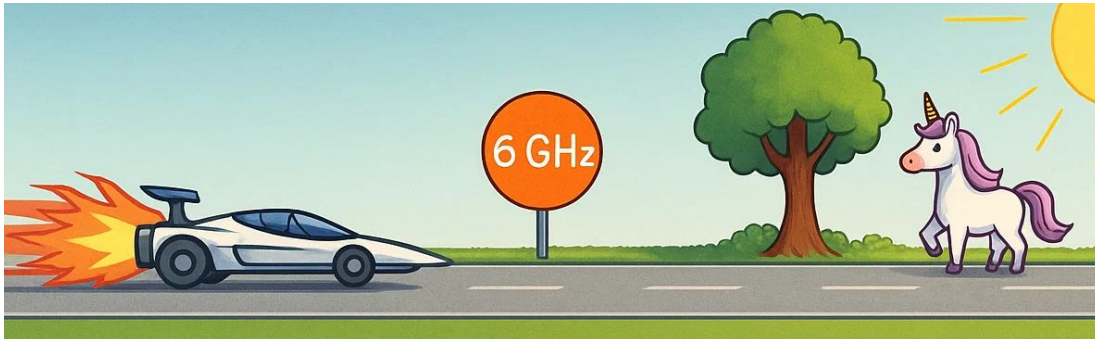
Можно ли делать ALU пока ждем RAM ?

2x ALU +2x RAM

Context switch - дорого!
Долго подгружает регистры!

Архитектура CPU: Hyper-Threading, SMT





Какое суммарное время работы двух таких потоков?

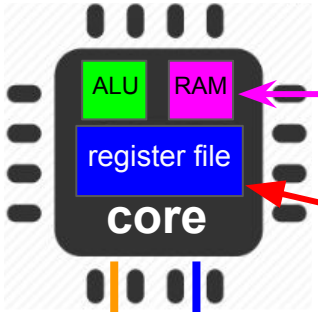
6x ALU +2x RAM

Давайте держать два множества регистров!

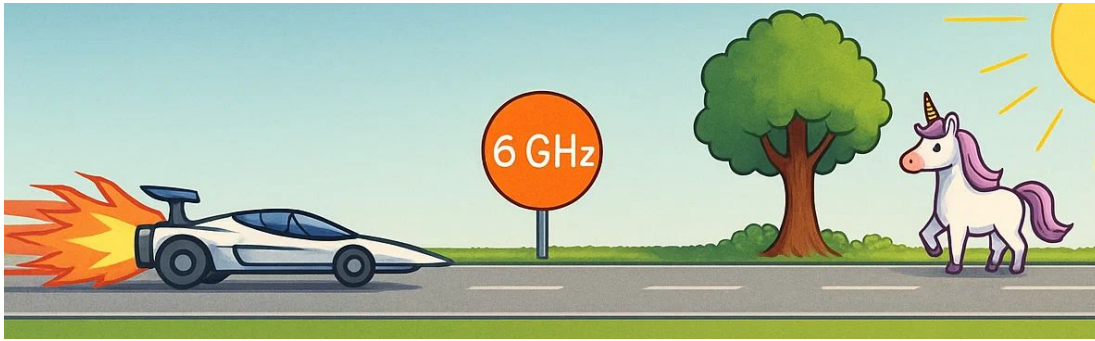
Можно ли делать ALU пока ждем RAM ?

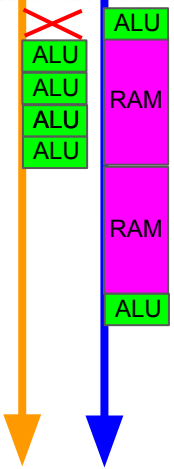
2x ALU +2x RAM

Архитектура CPU: Hyper-Threading, SMT



RAM - DDR5





Какое суммарное время работы двух таких потоков?

6x ALU +2x RAM

Давайте держать два множества регистров!

Можно ли делать ALU пока ждем RAM ?

Это и есть SMT и HT!

2x ALU +2x RAM

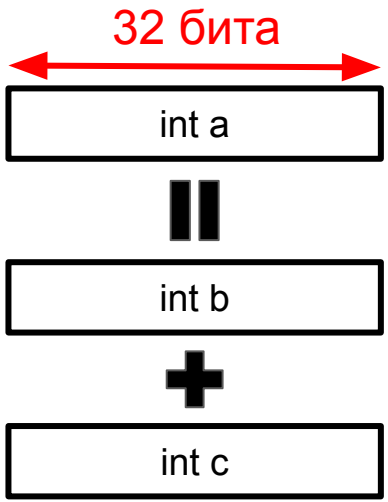
Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

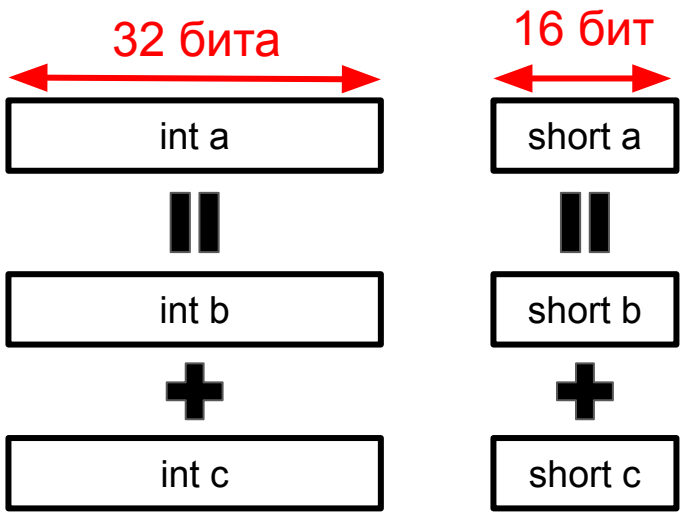
Было:



Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

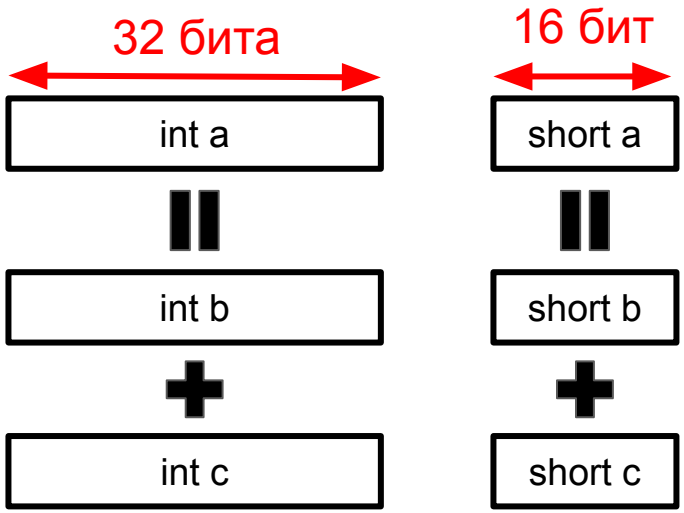
Было:



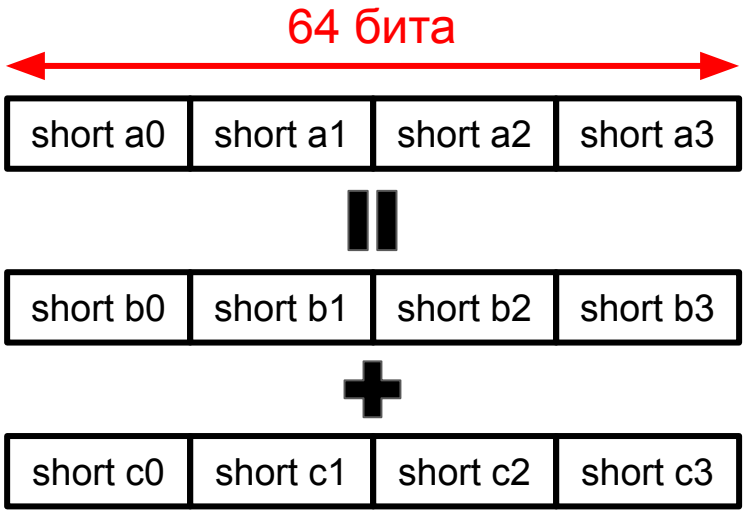
Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

Было:



Стало:

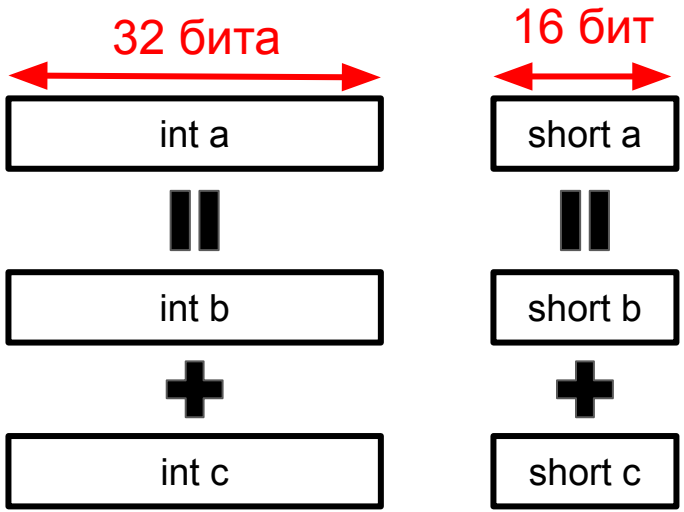


Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

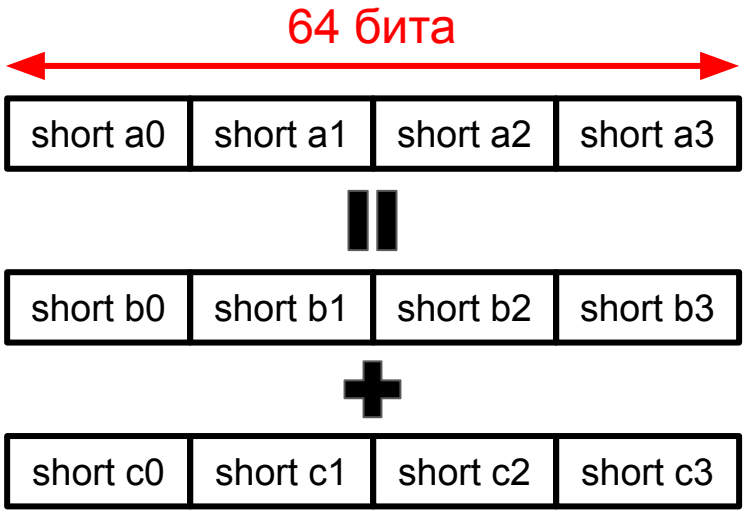
1997 - MMX - 64 бита: *8 x char,* *4 x short,* *2 x int*

Какое следовало бы ожидать ускорение?

Было:



Стало:



Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

1999 - SSE - 128 бит: *4 x float, 2 x double*

Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

1999 - SSE - 128 бит: *4 x float, 2 x double*

2000 - SSE2 - 128 бит, теперь и над целыми: *16 x char, 8 x short, 4 x int, 4 x float, 2 x double*

Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

1999 - SSE - 128 бит: *4 x float, 2 x double*

2000 - SSE2 - 128 бит, теперь и над целыми: *16 x char, 8 x short, 4 x int, 4 x float, 2 x double*

2004-2006 SSE3 / SSSE3 / SSE4 / SSE4.1 / SSE4.2 - расширения набора инструкций (горизонтальная работа в регистре, функции для видеокодирования и обработки строк)

Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

1999 - SSE - 128 бит: *4 x float, 2 x double*

2000 - SSE2 - 128 бит, теперь и над целыми: *16 x char, 8 x short, 4 x int, 4 x float, 2 x double*

2004-2006 SSE3 / SSSE3 / SSE4 / SSE4.1 / SSE4.2 - расширения набора инструкций (горизонтальная работа в регистре, функции для видеокодирования и обработки строк)

2011 - AVX - 256 бит: *8 x float, 4 x double*

Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

1999 - SSE - 128 бит: *4 x float, 2 x double*

2000 - SSE2 - 128 бит, теперь и над целыми: *16 x char, 8 x short, 4 x int, 4 x float, 2 x double*

2004-2006 SSE3 / SSSE3 / SSE4 / SSE4.1 / SSE4.2 - расширения набора инструкций (горизонтальная работа в регистре, функции для видеокодирования и обработки строк)

2011 - AVX - 256 бит: *8 x float, 4 x double*

2013 - AVX2 - 256 бит, теперь и над целыми: *32 x char, 16 x short, 8 x int, 4 x long*

Архитектура CPU: **SIMD intrinsics** (т.е. встроенные)

1997 - MMX - 64 бита: *8 x char, 4 x short, 2 x int*

1999 - SSE - 128 бит: *4 x float, 2 x double*

2000 - SSE2 - 128 бит, теперь и над целыми: *16 x char, 8 x short, 4 x int, 4 x float, 2 x double*

2004-2006 SSE3 / SSSE3 / SSE4 / SSE4.1 / SSE4.2 - расширения набора инструкций (горизонтальная работа в регистре, функции для видеокодирования и обработки строк)

2011 - AVX - 256 бит: *8 x float, 4 x double*

2013 - AVX2 - 256 бит, теперь и над целыми: *32 x char, 16 x short, 8 x int, 4 x long*

2016 - AVX512 - 512 бит, впервые появился в **Intel Xeon Phi**
(~60 pentium ядер, это pivot закрытого GPGPU-проекта Larrabee)

Архитектура CPU: **SIMD** (Фрактал Мандельброта)

```
float x = 0.0f;
float y = 0.0f;
```

1

Архитектура CPU: **SIMD** (Фрактал Мандельброта)

```
float x = 0.0f;
float y = 0.0f;
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
```

2

Архитектура CPU: **SIMD** (Фрактал Мандельброта)

```
float x = 0.0f;
float y = 0.0f;
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    float xn = x * x - y * y + x0;
    y = 2 * x * y + y0;
    x = xn;
```

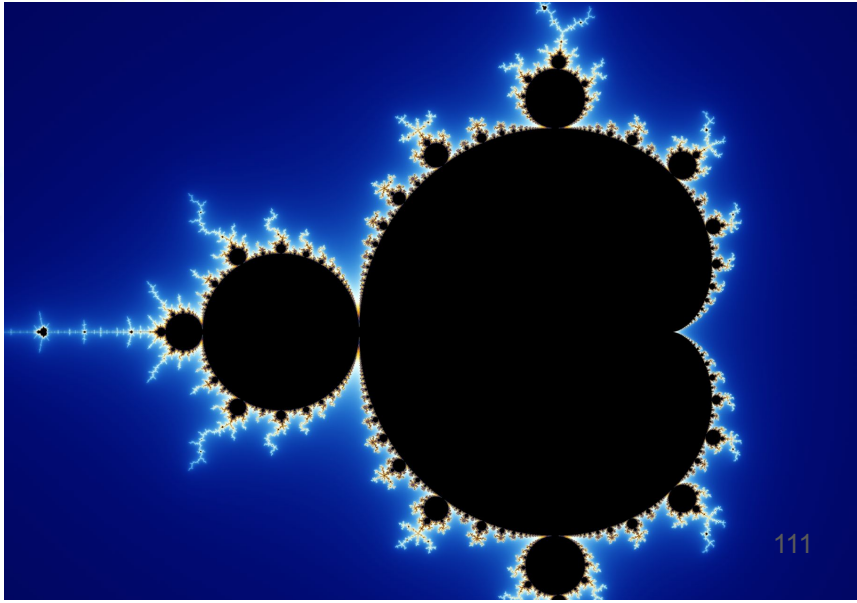
3

Архитектура CPU: **SIMD** (Фрактал Мандельброта)

```
float x = 0.0f;
float y = 0.0f;
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    float xn = x * x - y * y + x0;
    y = 2 * x * y + y0;
    x = xn;
    if (x * x + y * y > INFINITY) {
        break;
    }
}
```

4

быстрый выход
если ряд разошелся



Архитектура CPU: **SIMD** (Фрактал Мандельброта)

```
float x = 0.0f;
float y = 0.0f;
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    float xn = x * x - y * y + x0;
    y = 2 * x * y + y0;
    x = xn;
    if (x * x + y * y > INFINITY) {
        break;
    }
}
```

4

быстрый выход
если ряд разошелся

Во что мы упираемся?
Compute bound
или
memory-bandwidth bound?

Архитектура CPU: **SIMD** (Фрактал Мандельброта)

```
__m128i maskAll = _mm_setzero_si128();
__m128i iters = _mm_setzero_si128();
__m128 xs = _mm_set1_ps(0.0f);
__m128 ys = _mm_set1_ps(0.0f);
```

128 бит = 4 x float

1

Архитектура CPU: **SIMD** (Фрактал Мандельброта)

128 бит = 4 x float

```
__m128i maskAll = _mm_setzero_si128();
__m128i iters = _mm_setzero_si128();
__m128 xs = _mm_set1_ps(0.0f);
__m128 ys = _mm_set1_ps(0.0f);
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
```

2

Архитектура CPU: **SIMD** (Фрактал Мандельброта)

128 бит = 4 x float

```
__m128i maskAll = _mm_setzero_si128();
__m128i iters = _mm_setzero_si128();
__m128 xs = _mm_set1_ps(0.0f);
__m128 ys = _mm_set1_ps(0.0f);
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    __m128 xsn = _mm_add_ps(_mm_sub_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)), xs0);
    __m128 ysn = _mm_add_ps(_mm_mul_ps(_mm_mul_ps(_mm_set1_ps(2.0f), xs), ys), _mm_set1_ps(y0));
    xs = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, xsn), _mm_and_ps((__m128) maskAll, xs));
    ys = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, ysn), _mm_and_ps((__m128) maskAll, ys));
```

3

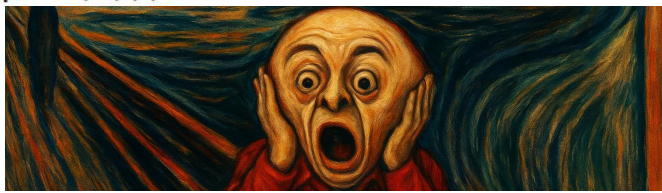
Архитектура CPU: **SIMD** (Фрактал Мандельброта)

128 бит = 4 x float

```
__m128i maskAll = _mm_setzero_si128();
__m128i iters = _mm_setzero_si128();
__m128 xs = _mm_set1_ps(0.0f);
__m128 ys = _mm_set1_ps(0.0f);
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    __m128 xsn = _mm_add_ps(_mm_sub_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)), xs0);
    __m128 ysn = _mm_add_ps(_mm_mul_ps(_mm_mul_ps(_mm_set1_ps(2.0f), xs), ys), _mm_set1_ps(y0));
    xs = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, xsn), _mm_and_ps((__m128) maskAll, xs));
    ys = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, ysn), _mm_and_ps((__m128) maskAll, ys));

    maskAll = _mm_or_si128(_mm_castps_si128(_mm_cmpge_ps(_mm_add_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)
    iters = _mm_add_epi16(iters, _mm_andnot_si128(maskAll, _mm_set1_epi16(1)));
    int mask = _mm_movemask_epi8(maskAll);
    if (mask == 0xffff) {
        break;
    }
}
iters = _mm_shuffle_epi8(iters, _mm_setr_epi8(12, 13, 8, 9, 4, 5, 0, 1, -1, -1, -1, -1, -1, -1, -1, -1));
```

4



Архитектура CPU: **SIMD** (Фрактал Мандельброта)

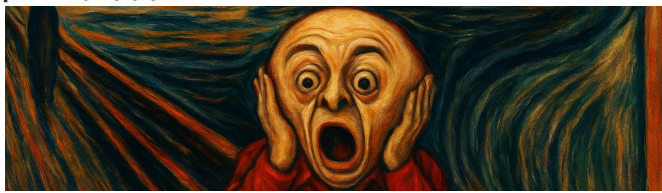
128 бит = 4 x float

```
__m128i maskAll = _mm_setzero_si128();
__m128i iters = _mm_setzero_si128();
__m128 xs = _mm_set1_ps(0.0f);
__m128 ys = _mm_set1_ps(0.0f);
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    __m128 xsn = _mm_add_ps(_mm_sub_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)), xs0);
    __m128 ysn = _mm_add_ps(_mm_mul_ps(_mm_mul_ps(_mm_set1_ps(2.0f), xs), ys), _mm_set1_ps(y0));
    xs = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, xsn), _mm_and_ps((__m128) maskAll, xs));
    ys = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, ysn), _mm_and_ps((__m128) maskAll, ys));

    maskAll = _mm_or_si128(_mm_castps_si128(_mm_cmpge_ps(_mm_add_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)
    iters = _mm_add_epi16(iters, _mm_andnot_si128(maskAll, _mm_set1_epi16(1)));
    int mask = _mm_movemask_epi8(maskAll);
    if (mask == 0xffff) {
        break;
    }
}
iters = _mm_shuffle_epi8(iters, _mm_setr_epi8(12, 13, 8, 9, 4, 5, 0, 1, -1, -1, -1, -1, -1, -1, -1, -1));
```

4

4.5

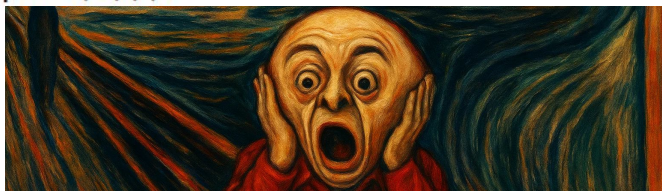


Архитектура CPU: **SIMD** (Фрактал Мандельброта)

128 бит = 4 x float

```
__m128i maskAll = _mm_setzero_si128();
__m128i iters = _mm_setzero_si128();
__m128 xs = _mm_set1_ps(0.0f);
__m128 ys = _mm_set1_ps(0.0f);
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    __m128 xsn = _mm_add_ps(_mm_sub_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)), xs0);
    __m128 ysn = _mm_add_ps(_mm_mul_ps(_mm_mul_ps(_mm_set1_ps(2.0f), xs), ys), _mm_set1_ps(y0));
    xs = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, xsn), _mm_and_ps((__m128) maskAll, xs));
    ys = _mm_add_ps(_mm_andnot_ps((__m128) maskAll, ysn), _mm_and_ps((__m128) maskAll, ys));

    maskAll = _mm_or_si128(_mm_castps_si128(_mm_cmpge_ps(_mm_add_ps(_mm_mul_ps(xs, xs), _mm_mul_ps(ys, ys)
    iters = _mm_add_epi16(iters, _mm_andnot_si128(maskAll, _mm_set1_epi16(1)));
    int mask = _mm_movemask_epi8(maskAll);
    if (mask == 0xffff) {
        break;
    }
}
iters = _mm_shuffle_epi8(iters, _mm_setr_epi8(12, 13, 8, 9, 4, 5, 0, 1, -1, -1, -1, -1, -1, -1, -1, -1));
```



Архитектура CPU: **SIMD** (Фрактал Мандельброта)

256 бит = 8 x float

```
__m256i maskAll = _mm256_setzero_si256();
__m256i iters = _mm256_setzero_si256();
__m256 xs = _mm256_set1_ps(0.0f);
__m256 ys = _mm256_set1_ps(0.0f);
for (iteration = 0; iteration < MAX_ITERATIONS; iteration++) {
    __m256 xsn = _mm256_add_ps(_mm256_sub_ps(_mm256_mul_ps(xs, xs), _mm256_mul_ps(ys, ys)), xs0);
    __m256 ysn = _mm256_add_ps(_mm256_mul_ps(_mm256_mul_ps(_mm256_set1_ps(2.0f), xs), ys), _mm256_set1_ps
    xs = _mm256_add_ps(_mm256_andnot_ps((__m256) maskAll, xsn), _mm256_and_ps((__m256) maskAll, xs));
    ys = _mm256_add_ps(_mm256_andnot_ps((__m256) maskAll, ysn), _mm256_and_ps((__m256) maskAll, ys));

    maskAll = (__m256i) _mm256_or_ps(_mm256_cmp_ps(_mm256_add_ps(_mm256_mul_ps(xs, xs), _mm256_mul_ps(ys,
    iters = _mm256_add_epi16(iters, _mm256_andnot_si256(maskAll, _mm256_set1_epi16(1)));
    int mask = _mm256_movemask_epi8(maskAll);
    if (mask == (int) 0xffffffff) {
        break;
    }
}
iters = _mm256_shuffle_epi8(iters, _mm256_setr_epi8(0, 1, -1, -1, 4, 5, -1, -1, 8, 9, -1, -1, 12, 13, -1,
```


Глава 2: история GPU
Fixed pipeline rendering, vertex + fragment shaders

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

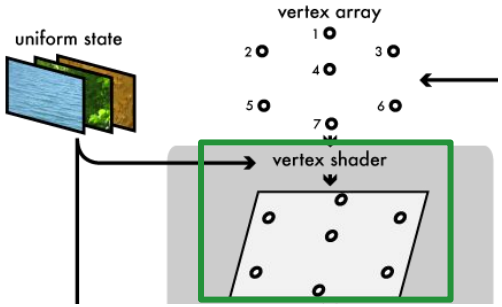
История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры***
(vertex + fragment shaders)

История GPU



До **1996** года 3D графика на CPU (DOOM, Quake,

1995-1997 - становятся популярны 3D-ускорители
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodo
Fixed pipeline, нет возможности исполнять произв

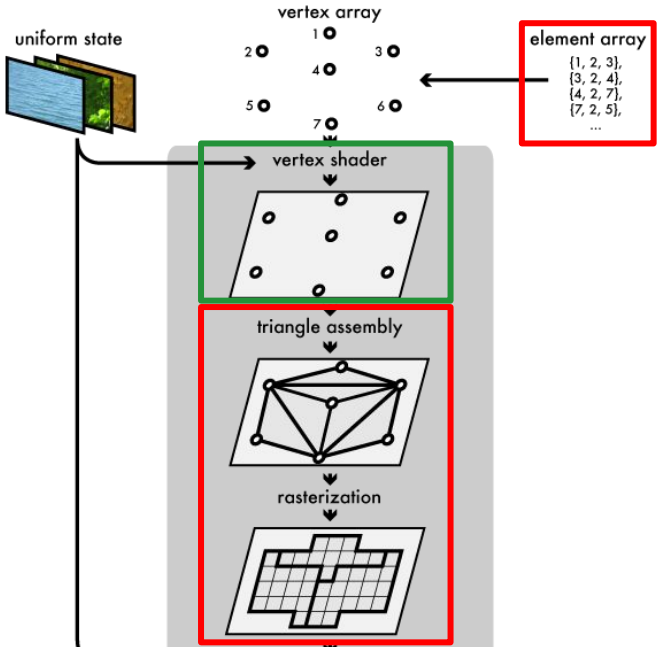
2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры***
(vertex + fragment shaders)

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake,

1995-1997 - становятся популярны 3D-ускорители
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodo
Fixed pipeline, нет возможности исполнять произв

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры***
(vertex + fragment shaders)

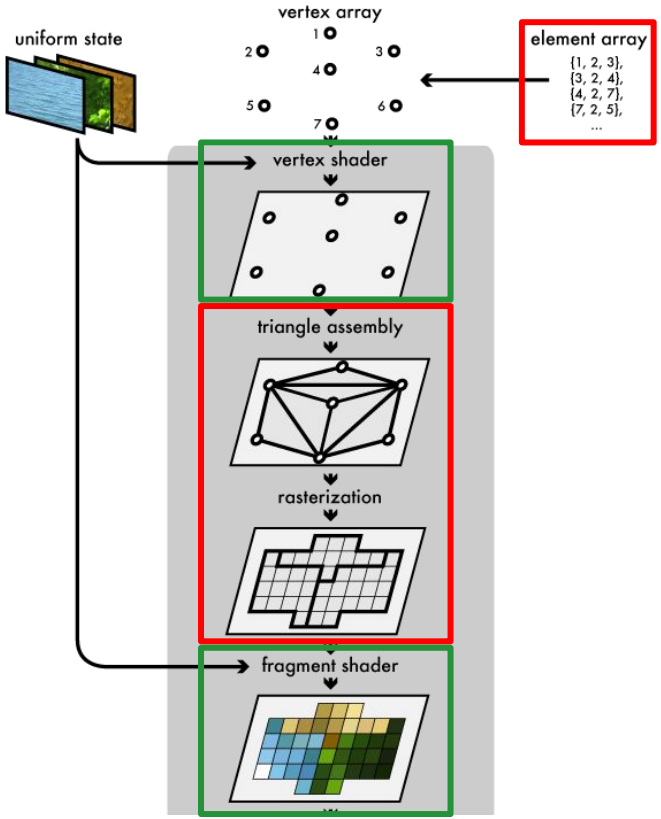


История GPU

До **1996** года 3D графика на CPU (DOOM, Quake,

1995-1997 - становятся популярны 3D-ускорители
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodo
Fixed pipeline, нет возможности исполнять произв

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры***
(vertex + fragment shaders)



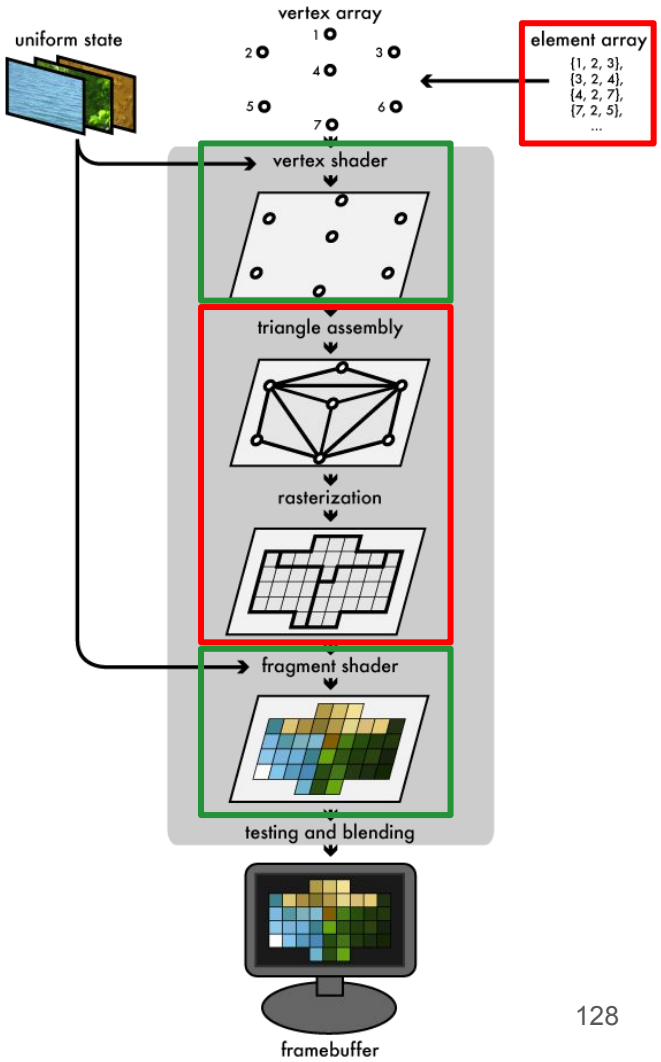
В каждом пикселе работает наш произвольный код!

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake,

1995-1997 - становятся популярны 3D-ускорители
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodo
Fixed pipeline, нет возможности исполнять произв

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры***
(vertex + fragment shaders)



История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

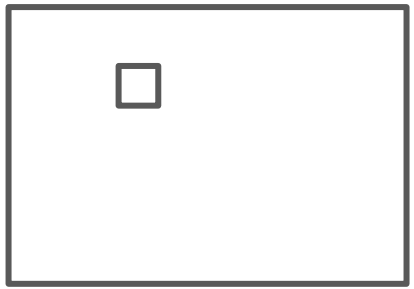
История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

Как посчитать какую-нибудь численную схему распространения тепла?



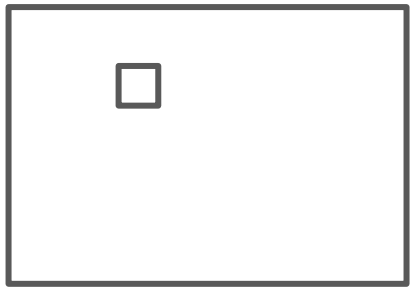
История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

Как посчитать какую-нибудь численную схему распространения тепла?



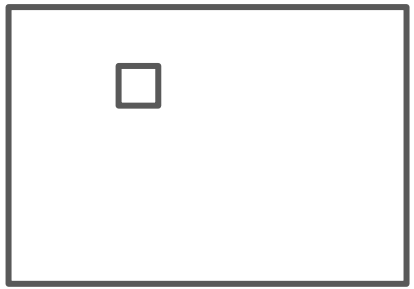
История GPU

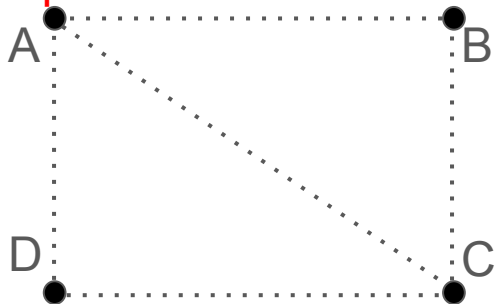
До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

Как посчитать какую-нибудь численную схему распространения тепла?





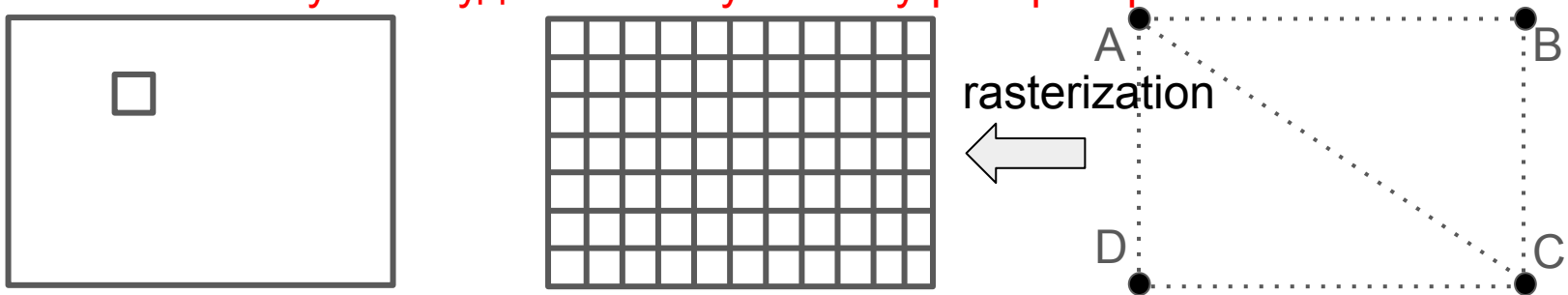
История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

Как посчитать какую-нибудь численную схему распространения тепла?



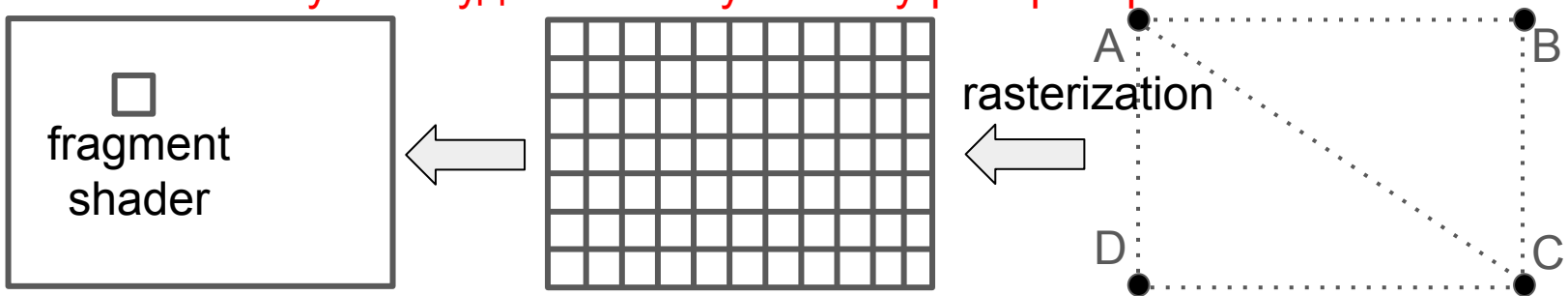
История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

Как посчитать какую-нибудь численную схему распространения тепла?



История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

2002-2004 - Cg (NVIDIA) и Brook (Stanford) - зарождение **GPGPU**

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

2002-2004 - Cg (NVIDIA) и Brook (Stanford) - зарождение **GPGPU**

2006-2008 - Close-to-Metal (ATI/AMD), **CUDA** (NVIDIA), **OpenCL** (Khronos)

История GPU

До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

2002-2004 - Cg (NVIDIA) и Brook (Stanford) - зарождение **GPGPU**

2006-2008 - Close-to-Metal (ATI/AMD), **CUDA** (NVIDIA), **OpenCL** (Khronos)
2012 - Compute Shaders в OpenGL

История GPU

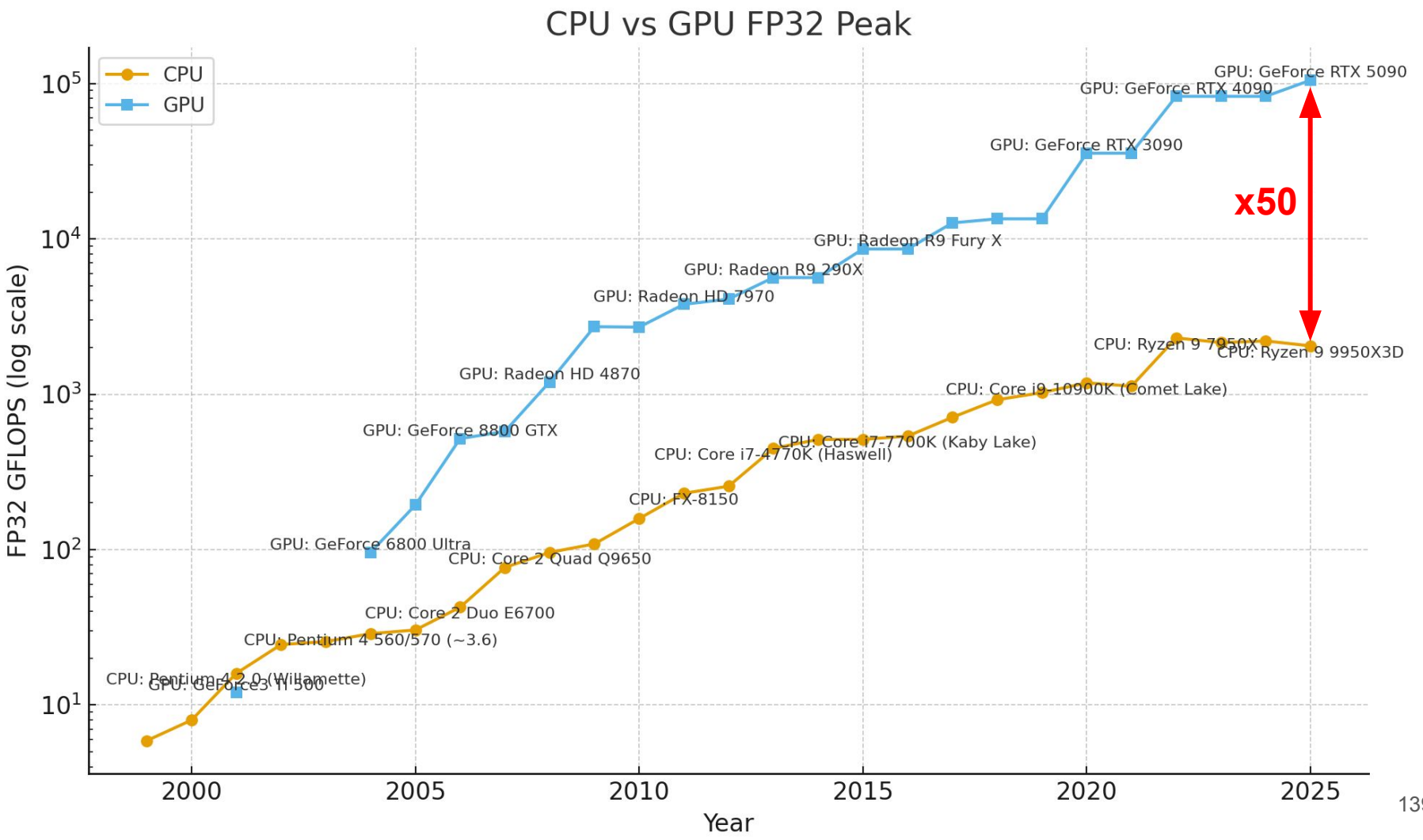
До **1996** года 3D графика на CPU (DOOM, Quake, Nukem 3D)

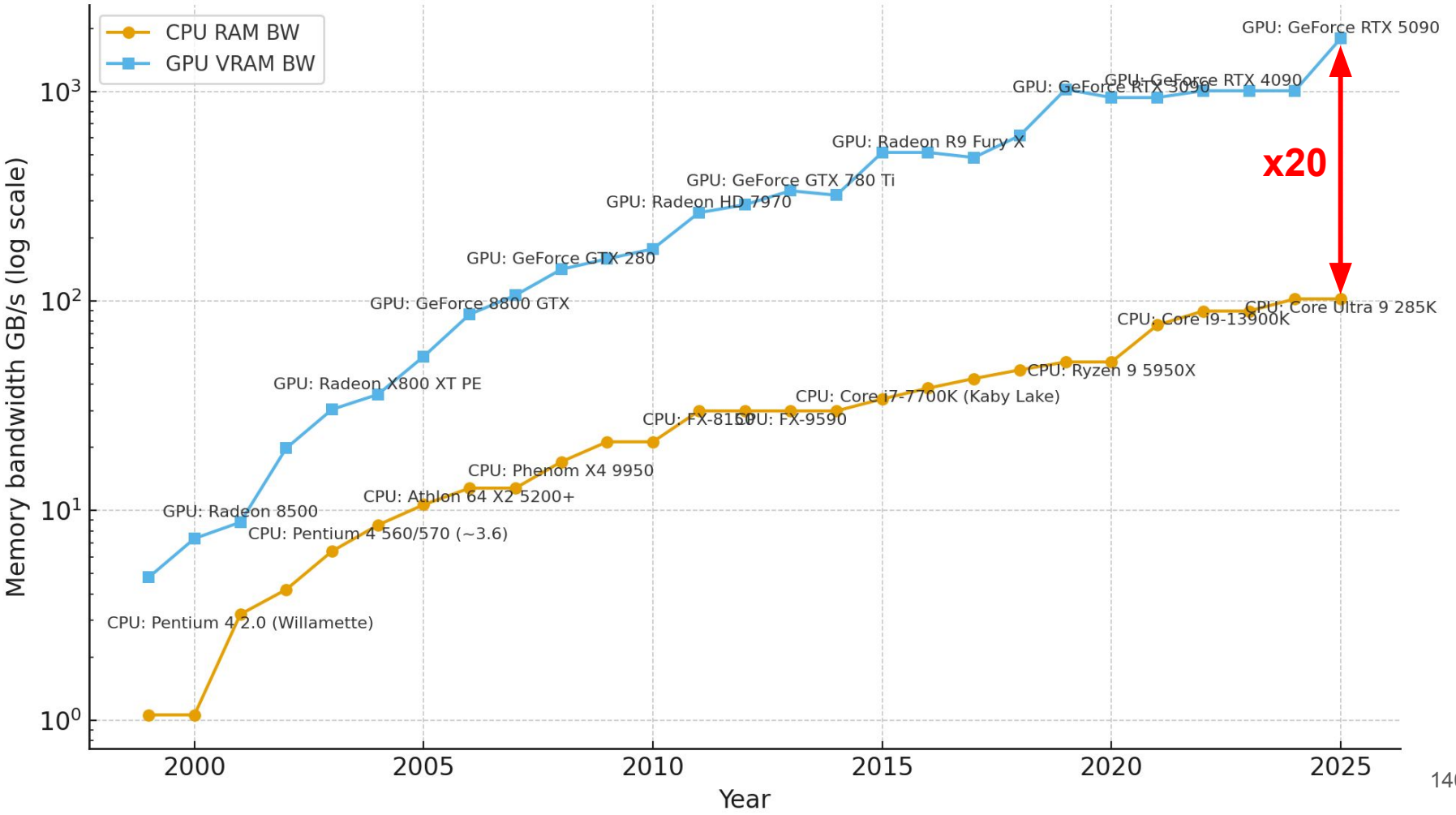
1995-1997 - становятся популярны 3D-ускорители:
S3 ViRGE, ATI 3D Rage (куплена **AMD**), 3dfx Voodoo (куплена **NVIDIA**)
Fixed pipeline, нет возможности исполнять произвольный код

2000-2002 - в DirectX 8.0 и OpenGL
появляются программируемые ***шейдеры*** ***(vertex + fragment shaders)***

2002-2004 - Cg (NVIDIA) и Brook (Stanford) - зарождение **GPGPU**

2006-2008 - Close-to-Metal (ATI/AMD), **CUDA** (NVIDIA), **OpenCL** (Khronos)
2012 - Compute Shaders в OpenGL
2016 - Vulkan


CPU vs GPU FP32 Peak
CPU
GPU
FP32 GFLOPS (log scale)
Year
10^1
10^2
10^3
10^4
10^5
2000
2005
2010
2015
2020
2025
GPU: GeForce RTX 5090
GPU: GeForce RTX 4090
GPU: GeForce RTX 3090
GPU: Radeon R9 Fury X
GPU: Radeon R9 290X
GPU: Radeon HD 7970
GPU: Radeon HD 4870
GPU: GeForce 8800 GTX
GPU: GeForce 6800 Ultra
GPU: GeForce3 Ti 500
x50
CPU: Ryzen 9 9950X
CPU: Ryzen 9 9950X3D
CPU: Core i9-10900K (Comet Lake)
CPU: Core i7-7700K (Kaby Lake)
CPU: Core i7-4770K (Haswell)
CPU: FX-8150
CPU: Core 2 Quad Q9650
CPU: Core 2 Duo E6700
CPU: Pentium 4 560/570 (~3.6)
CPU: Pentium 4 2.0 (Willamette)


CPU vs GPU Memory Bandwidth
CPU RAM BW
GPU VRAM BW
Memory bandwidth GB/s (log scale)
Year
10^0
10^1
10^2
10^3
2000
2005
2010
2015
2020
2025
GPU: GeForce RTX 5090
GPU: GeForce RTX 4090
GPU: GeForce RTX 3090
GPU: Radeon R9 Fury X
GPU: GeForce GTX 780 Ti
GPU: Radeon HD 7970
GPU: GeForce GTX 280
GPU: GeForce 8800 GTX
GPU: Radeon X800 XT PE
GPU: Radeon 8500
x20
CPU: Core Ultra 9 285K
CPU: Core i9-13900K
CPU: Ryzen 9 5950X
CPU: Core i7-7700K (Kaby Lake)
CPU: FX-9590
CPU: FX-8150
CPU: Phenom X4 9950
CPU: Athlon 64 X2 5200+
CPU: Pentium 4 560/570 (~3.6)
CPU: Pentium 4 2.0 (Willamette)

Глава 3: OpenCL API

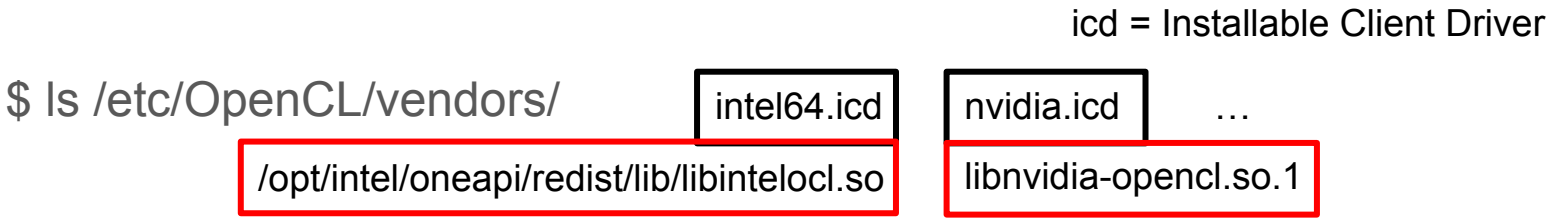
это красиво, это открыто, это стандарт

Введение в OpenCL API

```
$ ls /etc/OpenCL/vendors/
```

intel64.icd nvidia.icd

Введение в OpenCL API



Введение в OpenCL API

icd = Installable Client Driver

```
$ ls /etc/OpenCL/vendors/
```

intel64.icd

/opt/intel/oneapi/redist/lib/libintelocl.so

nvidia.icd

libnvidia-opencl.so.1

…

```
$ clinfo -l

Platform #0: NVIDIA CUDA
 `-- Device #0: NVIDIA GeForce RTX 5070 Ti
```

Введение в OpenCL API

icd = Installable Client Driver

```
$ ls /etc/OpenCL/vendors/
```

intel64.icd

/opt/intel/oneapi/redist/lib/libintelocl.so

nvidia.icd

libnvidia-opencl.so.1

…

```
$ clinfo -l

Platform #0: NVIDIA CUDA
 `-- Device #0: NVIDIA GeForce RTX 5070 Ti
Platform #1: Intel(R) OpenCL
 `-- Device #0: AMD Ryzen 9 9950X3D 16-Core Processor
```

Введение в OpenCL API

icd = Installable Client Driver

$ ls /etc/OpenCL/vendors/ intel64.icd nvidia.icd …

OpenCL **платформы**

Введение в OpenCL API

icd = Installable Client Driver

$ ls /etc/OpenCL/vendors/

intel64.icd nvidia.icd …

OpenCL **платформы**

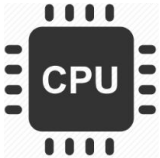
Введение в OpenCL API

icd = Installable Client Driver

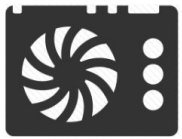
$ ls /etc/OpenCL/vendors/ intel64.icd nvidia.icd …

OpenCL **платформы**

OpenCL **устройства**



GPU



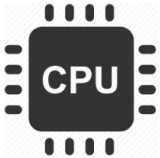
Введение в OpenCL API

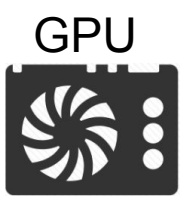
icd = Installable Client Driver

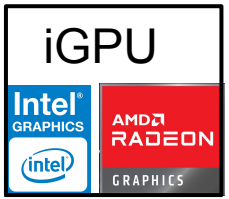
$ ls /etc/OpenCL/vendors/ intel64.icd nvidia.icd …

OpenCL **платформы**

OpenCL **устройства**







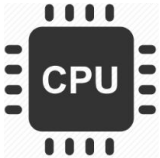
Введение в OpenCL API

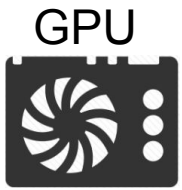
icd = Installable Client Driver

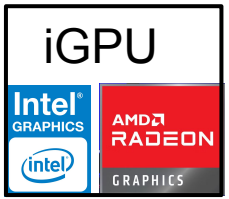
$ ls /etc/OpenCL/vendors/ intel64.icd nvidia.icd …

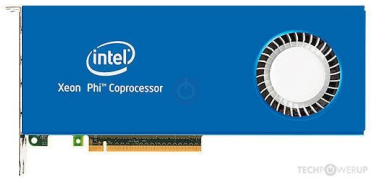
OpenCL **платформы**

OpenCL **устройства**









Введение в OpenCL API

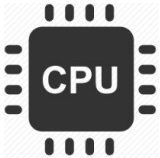
icd = Installable Client Driver

$ ls /etc/OpenCL/vendors/

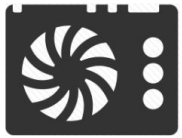
intel64.icd nvidia.icd …

OpenCL **платформы**

OpenCL **устройства**

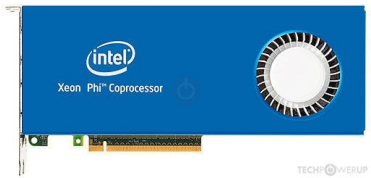


GPU



iGPU

FPGA



Введение в OpenCL API

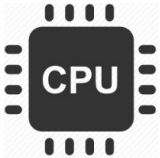
icd = Installable Client Driver

$ ls /etc/OpenCL/vendors/

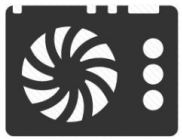
intel64.icd nvidia.icd …

OpenCL **платформы**

OpenCL **устройства**



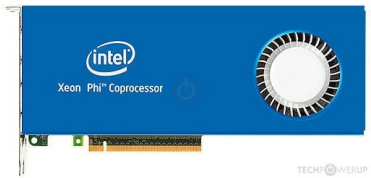
GPU



iGPU

FPGA

Может ли в одной платформе быть два устройства?



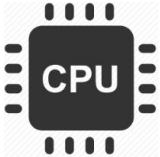
Введение в OpenCL API

icd = Installable Client Driver

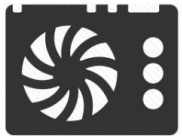
$ ls /etc/OpenCL/vendors/ intel64.icd nvidia.icd …

OpenCL **платформы**

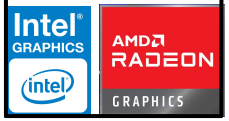
OpenCL **устройства**



GPU



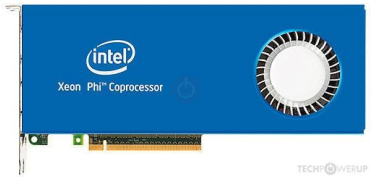
iGPU



FPGA

Может ли в одной платформе быть два устройства?

Может ли одно и то же hardware-устройство быть видно из двух платформ?



Домашние задания на github.com/GPGPUCourse

В этом репозитории предложены задания для курса по вычислениям на видеокартах 2025.

Остальные задания.

Задание 0. Вводное.

Установка OpenCL-драйвера для процессора

Установить OpenCL-драйвер для процессора полезно, даже если у вас есть видеокарта, т.к. на нем удобно тестировать приложение (драйвер видеокарты гораздо чаще может повиснуть вместе с ОС).

Windows

1. Откройте https://software.intel.com/content/www/us/en/develop/tools/opencl-cpu-runtime.html -> `For Intel CPUs` (для AMD тоже работает)
2. Скачайте, если у вас нет прокси то с зеркала - файл w_opencl_runtime_p_2025.2.0.768.exe
3. Установите

Linux (Рекомендуется Ubuntu 24.04)

Выполните скрипт install_intel_opencl.sh

Домашние задания на github.com/GPGPUCourse

Задание

0. Сделать fork проекта
1. Прочитать все комментарии подряд и выполнить все **TODO** в файле `src/main.cpp`. Для разработки под Linux рекомендуется использовать CLion. Под Windows рекомендуется использовать CLion+MSVC. Также под Windows можно использовать Visual Studio Community.
2. Отправить **Pull-request** с названием `Task00 <Имя> <Фамилия> <Аффиляция>`. **Аффиляция** - SPbU/HSE/ITMO.
3. В тексте **PR** укажите вывод программы при исполнении на сервере Github CI (Github Actions) и на вашем компьютере (в **pre**-тэгах, чтобы сохранить форматирование, см. пример). И ваш бранч должен называться так же, как и у меня - **task00**.
4. Убедиться что Github CI (Github Actions) смог скомпилировать ваш код и что все хорошо, при отправке первого задания CI может не запуститься пока я вручную не нажму `Approve` - если я этого не сделал в течение суток - напомните мне пожалуйста в чате курса
5. Ждать комментарии проверки

Дедлайн: 23:59 22 сентября. Но убедиться, что хотя бы одно OpenCL-устройство у вас обнаруживается, лучше как можно раньше, чтобы было больше времени на решение проблем если они возникнут (см. **Проверка окружения** выше).

Домашние задания на github.com/GPGPUCourse

```
29  #define OCL_SAFE_CALL(expr) reportError(expr, __FILE__, __LINE__)
30
31  int main()
32  {
33      // Пытаемся слинковаться с символами OpenCL API в runtime (через библиотеку libs/clew)
34      if(!ocl_init())
35          throw std::runtime_error( Message: "Can't init OpenCL driver!");
36
37      // Откройте
38      // https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/
39      // Нажмите слева: "OpenCL Runtime" -> "Query Platform Info" -> "clGetPlatformIDs"
40      // Прочитайте документацию clGetPlatformIDs и убедитесь, что этот способ узнать, сколько есть платформ, соответствует документации:
41      cl_uint platformsCount = 0;
42      OCL_SAFE_CALL(clGetPlatformIDs( num_entries: 0,  platforms: nullptr,  num_platforms: &platformsCount));
43      std::cout << "Number of OpenCL platforms: " << platformsCount << std::endl;
```

Домашние задания на github.com/GPGPUCourse

```
29  #define OCL_SAFE_CALL(expr) reportError(expr, __FILE__, __LINE__)
30
31  int main()
32  {
33      // Пытаемся слинковаться с символами OpenCL API в runtime (через библиотеку libs/clew)
34      if(!ocl_init())
35          throw std::runtime_error( Message: "Can't init OpenCL driver!");
36
37      // Откройте
38      // https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/
39      // Нажмите слева: "OpenCL Runtime" -> "Query Platform Info" -> "clGetPlatformIDs"
40      // Прочитайте документацию clGetPlatformIDs и убедитесь, что этот способ узнать, сколько есть платформ, соответствует документации:
41      cl_uint platformsCount = 0;
42      OCL_SAFE_CALL(clGetPlatformIDs( num_entries: 0,  platforms: nullptr,  num_platforms: &platformsCount));
43      std::cout << "Number of OpenCL platforms: " << platformsCount << std::endl;
44
45      // Тот же метод используется для того, чтобы получить идентификаторы всех платформ - сверьтесь с документацией, что это сделано верно:
46      std::vector<cl_platform_id> platforms( Count: platformsCount );
47      OCL_SAFE_CALL(clGetPlatformIDs( num_entries: platformsCount,  platforms: platforms.data(),  num_platforms: nullptr));
```

Домашние задания на github.com/GPGPUCourse

```
29  #define OCL_SAFE_CALL(expr) reportError(expr, __FILE__, __LINE__)
30
31  int main()
32  {
33      // Пытаемся слинковаться с символами OpenCL API в runtime (через библиотеку libs/clew)
34      if(!ocl_init())
35          throw std::runtime_error( Message: "Can't init OpenCL driver!");
36
37      // Откройте
38      // https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/
39      // Нажмите слева: "OpenCL Runtime" -> "Query Platform Info" -> "clGetPlatformIDs"
40      // Прочитайте документацию clGetPlatformIDs и убедитесь, что этот способ узнать, сколько есть платформ, соответствует документации:
41      cl_uint platformsCount = 0;
42      OCL_SAFE_CALL(clGetPlatformIDs( num_entries: 0,  platforms: nullptr,  num_platforms: &platformsCount));
43      std::cout << "Number of OpenCL platforms: " << platformsCount << std::endl;
44
45      // Тот же метод используется для того, чтобы получить идентификаторы всех платформ - сверьтесь с документацией, что это сделано верно:
46      std::vector<cl_platform_id> platforms( Count: platformsCount);
47      OCL_SAFE_CALL(clGetPlatformIDs( num_entries: platformsCount,  platforms: platforms.data(),  num_platforms: nullptr));
48
49      for(int platformIndex = 0; platformIndex < platformsCount; ++platformIndex)
50      {
51          std::cout << "Platform #" << (platformIndex + 1) << "/" << platformsCount << std::endl;
52          cl_platform_id platform = platforms[platformIndex];
```

CS Space
Клуб технологий и науки

Vulkan

NVIDIA CUDA

OpenCL™

Вопросы?

@UnicornGlade
@PolarNick239
polarnick239@gmail.com
Николай Полярный