



# Вычисления на видеокартах

## Лекция 2

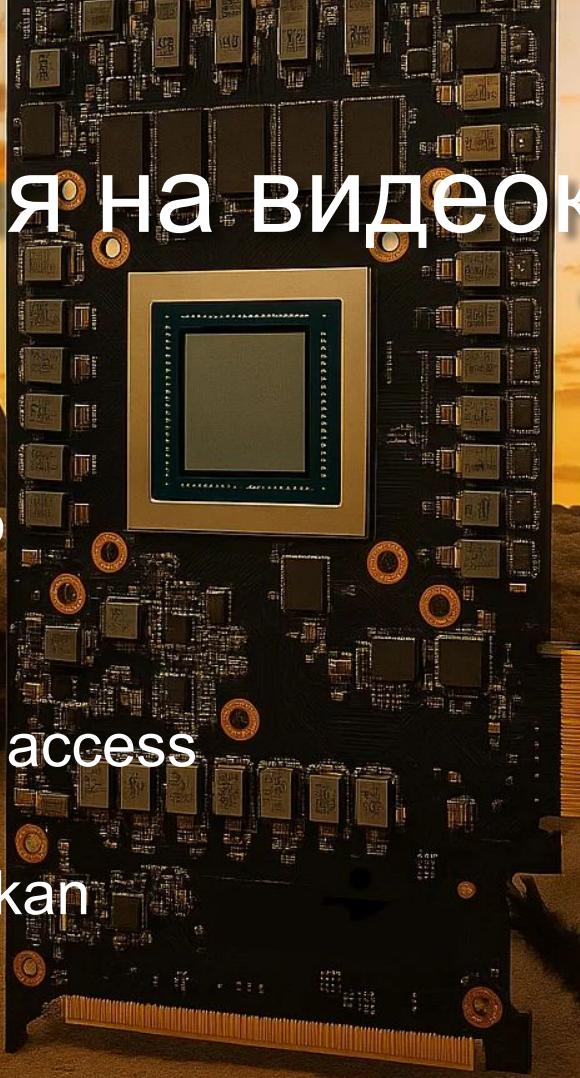
- Архитектура GPU
- Модель массового параллелизма
- code divergence
- coalesced memory access
- Код кернела:  
OpenCL CUDA Vulkan



polarnick239@gmail.com



Николай Полярный



Vulkan



nVIDIA

CUDA

OpenCL™

Activate Ubuntu

Go to Settings to activate Ubuntu.

# План лекции

Большая часть из лекции “CS Space: Видеокарты: что они могут?”

- 1) **Вычисления:**  
*shared instruction pointer, code divergence*
- 2) **Работа с памятью:**  
*hyper-threading, latency hiding, occupancy, registers pressure/spilling, coalesced memory access pattern, cache lines, local/shared memory*
- 3) **Общая картина ЭВМ-архитектуры:** CPU - RAM - PCI-E - VRAM - GPU
- 4) **Модель вычислений массового параллелизма**
- 5) **Профилирование и оптимизация:** NVIDIA Nsight, SoL анализ

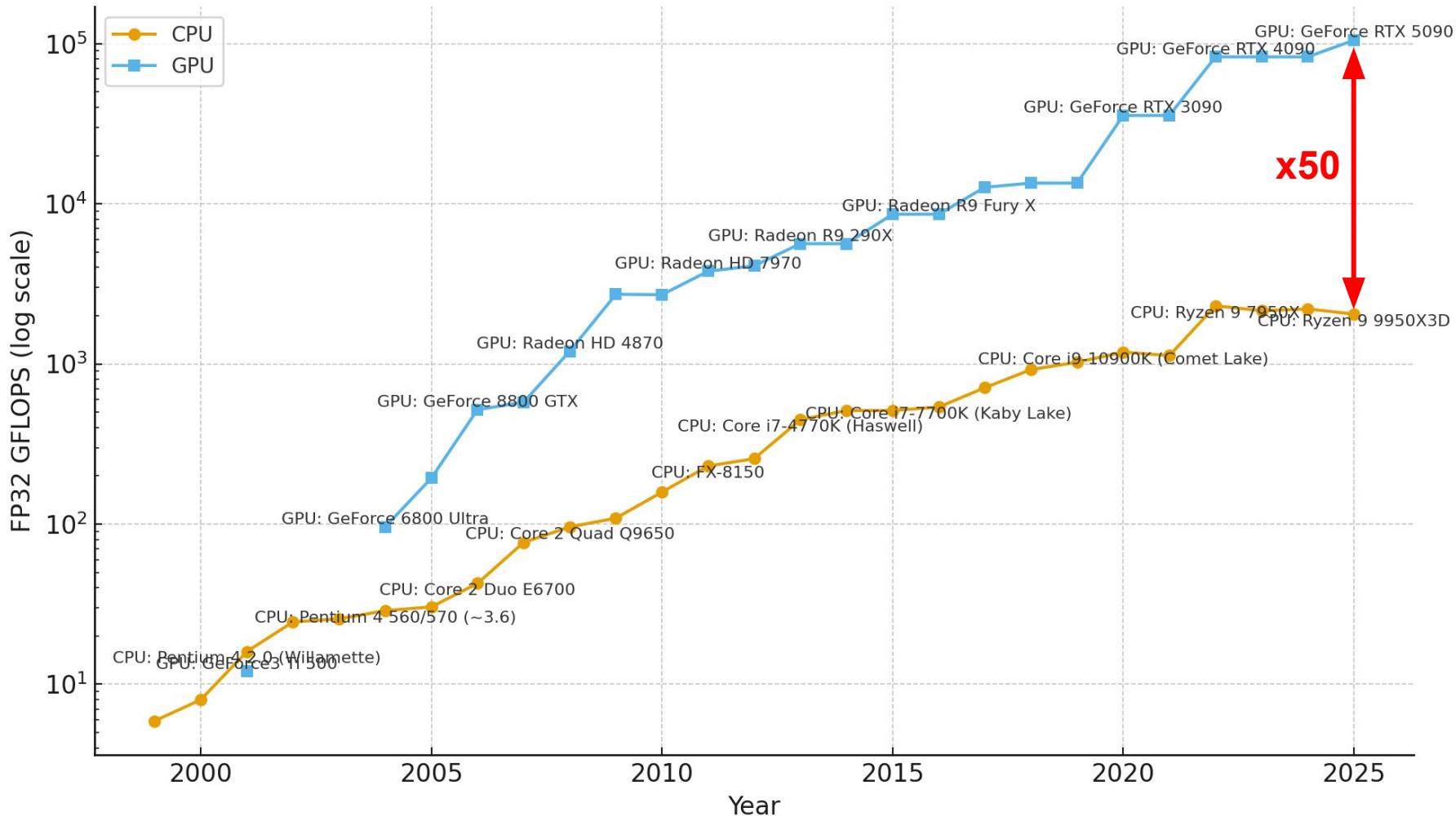
И обсудим как выглядит код на видеокарте:

- 6) **Пример кода:** A+B на C++ (OpenMP), OpenCL, CUDA, Vulkan (GLSL)

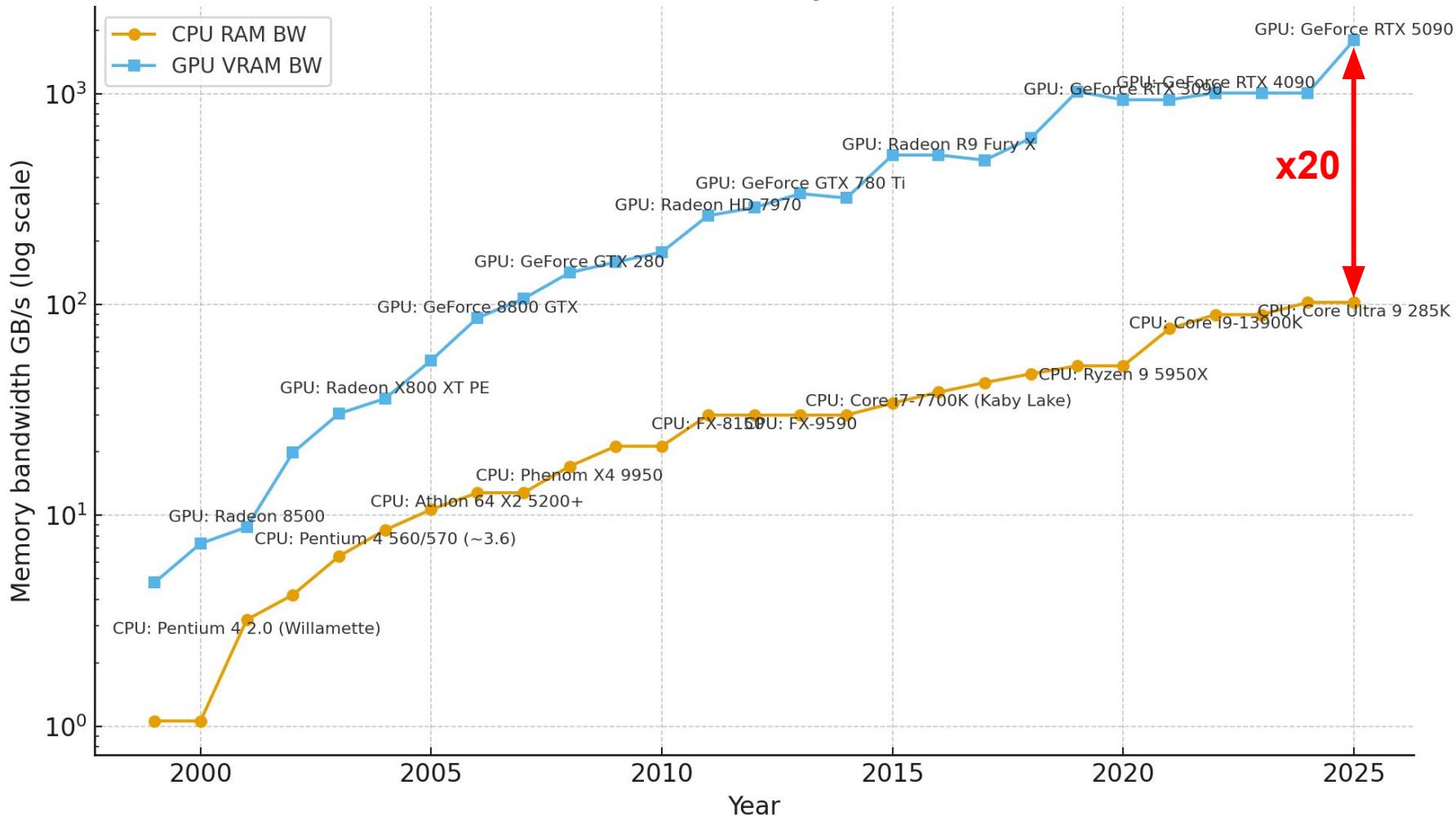
# Глава 1: Вычисления

shared instruction pointer, code divergence

# CPU vs GPU FP32 Peak

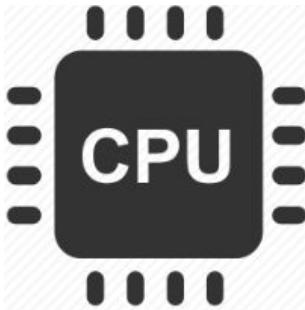


# CPU vs GPU Memory Bandwidth



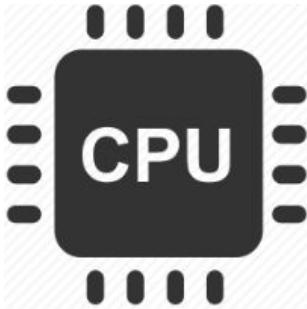
# Архитектура

$100 \cdot 10^9$  FLOPS (Floating-point operations per second)



# Архитектура

$100 \cdot 10^9$  FLOPS (Floating-point operations per second)



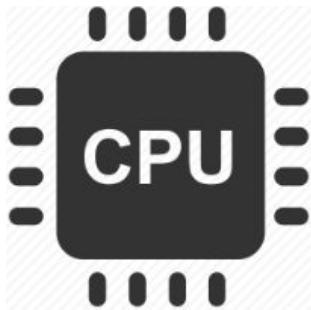
$100 \cdot 10^{12}$  FLOPS (**x1000 раз больше**)



GPU

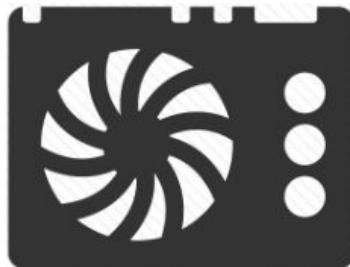
# Архитектура

$100 \cdot 10^9$  FLOPS



40 GB/s memory bandwidth

$100 \cdot 10^{12}$  FLOPS (**x1000 раз больше**)

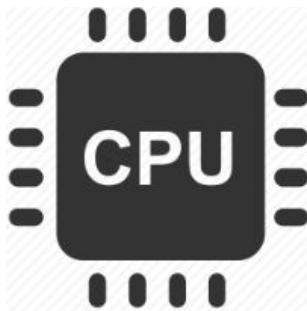


1000 GB/s (**x25 раз больше**)

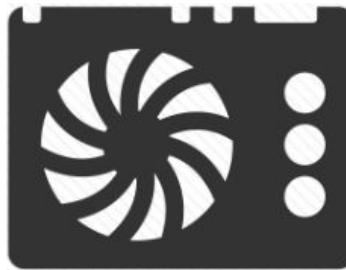
GPU

# Архитектура

$100 \cdot 10^9$  FLOPS



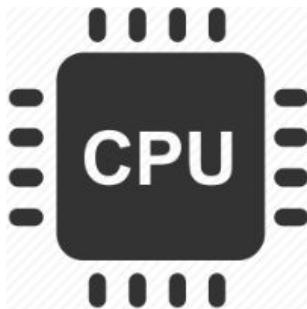
$100 \cdot 10^{12}$  FLOPS



GPU

# Архитектура

$100 \cdot 10^9$  FLOPS



$100 \cdot 10^{12}$  FLOPS



GPU

Мало ядер, но они **МОЩНЫЕ**

6 GHz

ТЫСЯЧИ слабых ядер

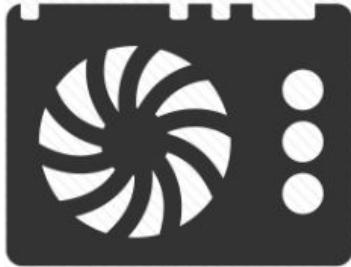
1 GHz



# Архитектура

Как уместить ТЫСЯЧИ ядер?  
На CPU ведь это почему-то невозможно!

$100 \cdot 10^{12}$  FLOPS



GPU

ТЫСЯЧИ слабых ядер

1GHz

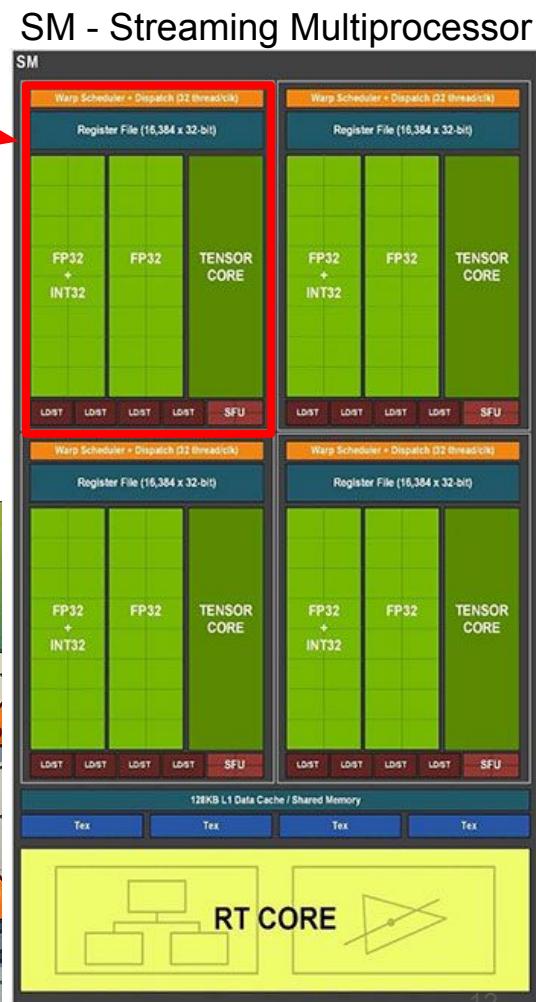


# Архитектура



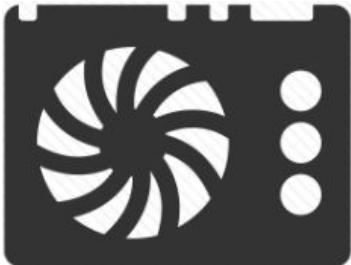
warp

32 слабых медленных CUDA ядер



Как уместить ТЫСЯЧИ ядер?  
На CPU ведь это почему-то невозможно!

$100 \cdot 10^{12}$  FLOPS



GPU

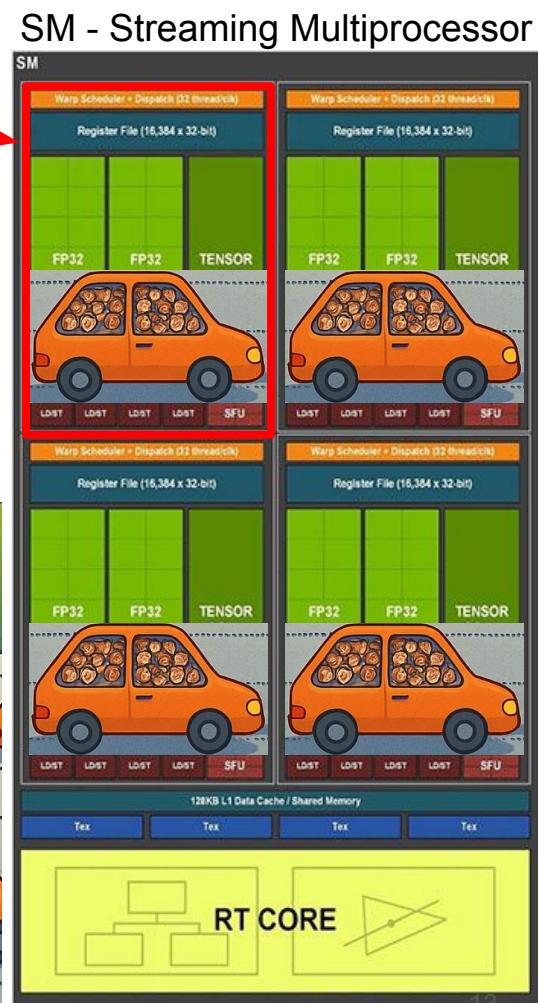


# Архитектура



warp

32 слабых медленных CUDA ядер

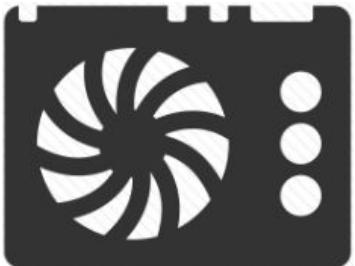


Как уместить ТЫСЯЧИ ядер?  
На CPU ведь это почему-то невозможно!

ТЫСЯЧИ слабых ядер

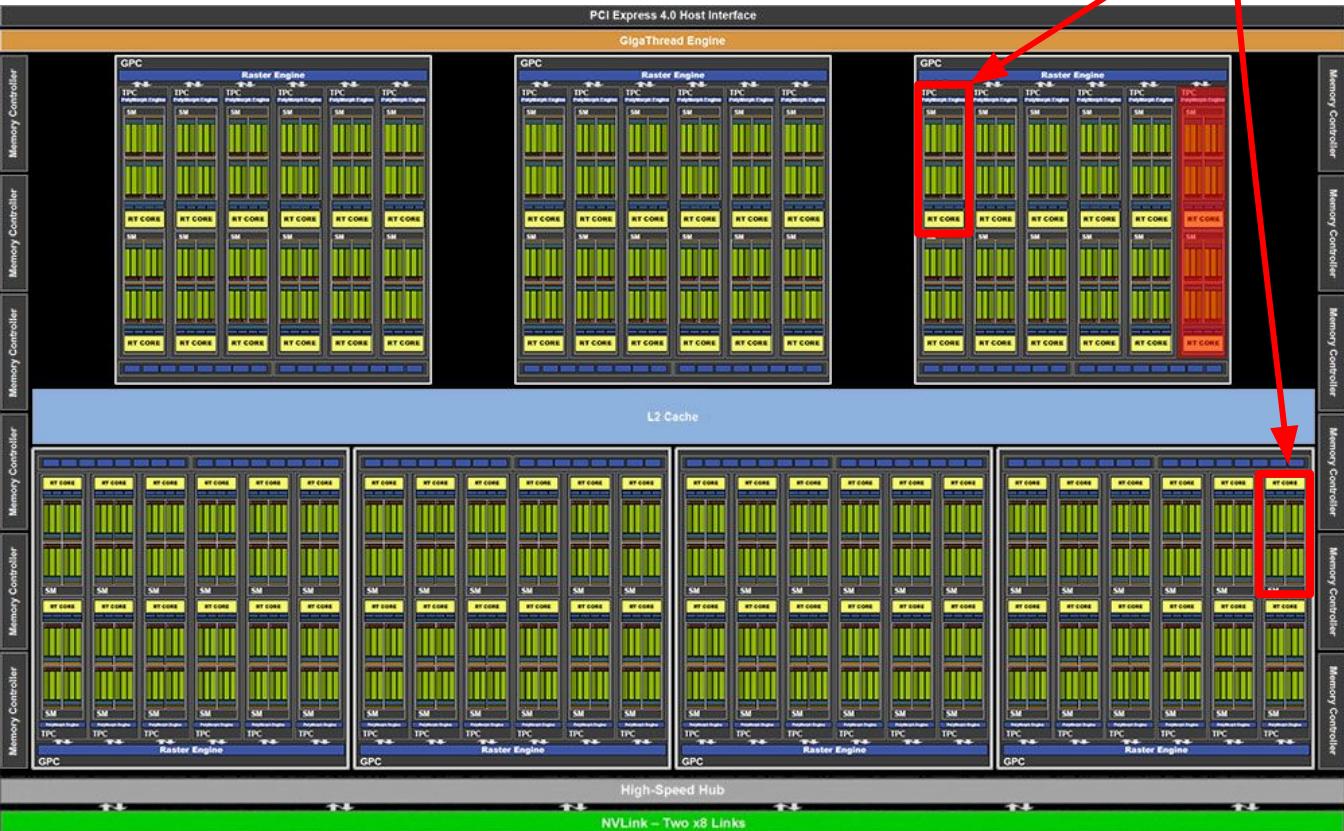


$100 \cdot 10^{12}$  FLOPS



GPU

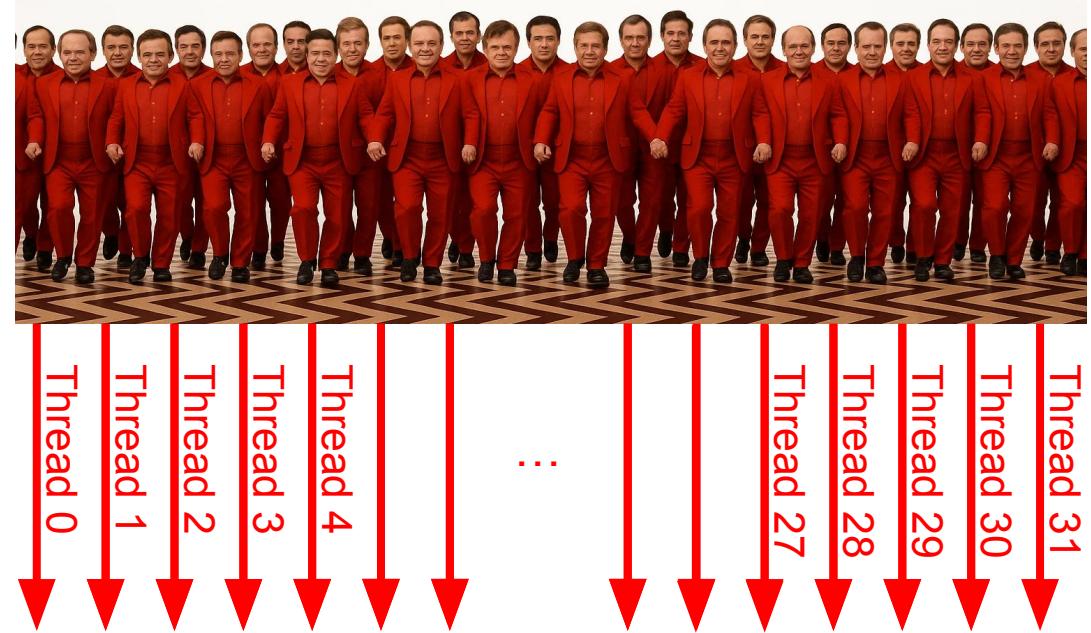
RTX 3090: 10496 CUDA cores = 82 SM · 4 warps · 32 ALUs



## CPU ядро



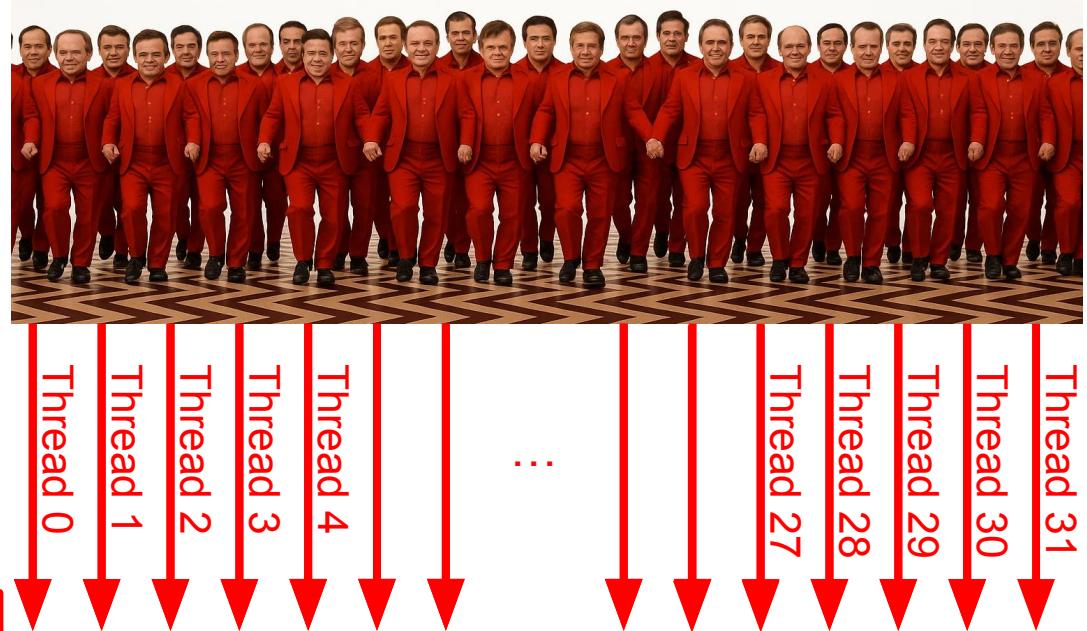
## GPU warp



Как уместить ТЫСЯЧИ ядер?  
На CPU ведь это почему-то невозможно!

# Архитектура

## GPU warp

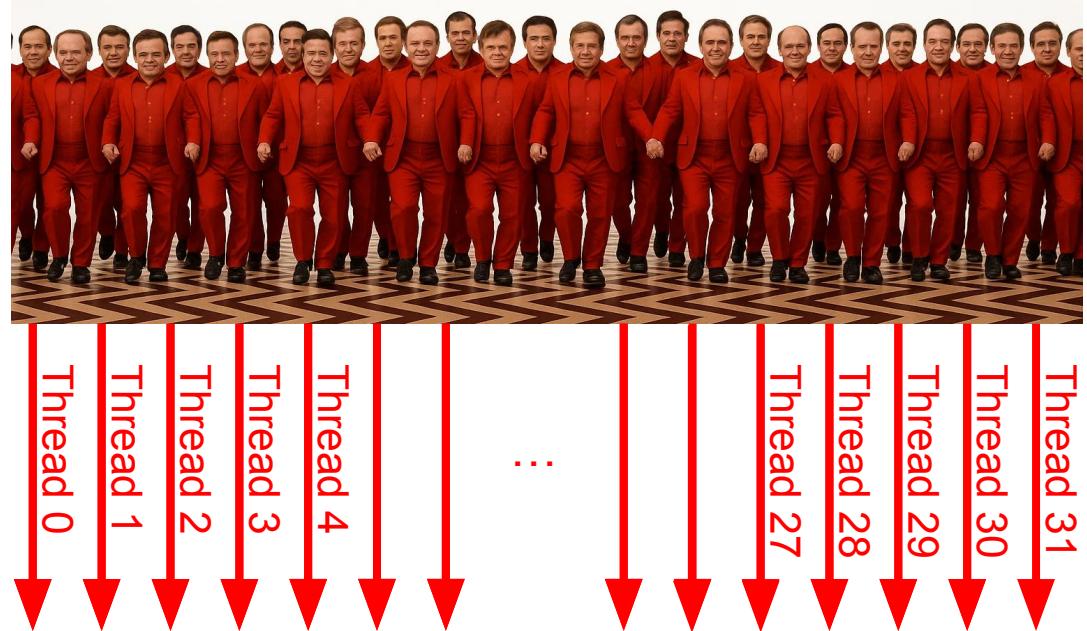


```
239 → int warpThreadID = threadIdx.x % 32;  
240  
241     int result = 0;  
242     if (warpThreadID < 16) {  
243         result = dataA[warpThreadID];  
244     } else {  
245         result = dataB[warpThreadID];  
246     }
```

Один Instruction pointer  
на все потоки warp-a!

# Архитектура

## GPU warp

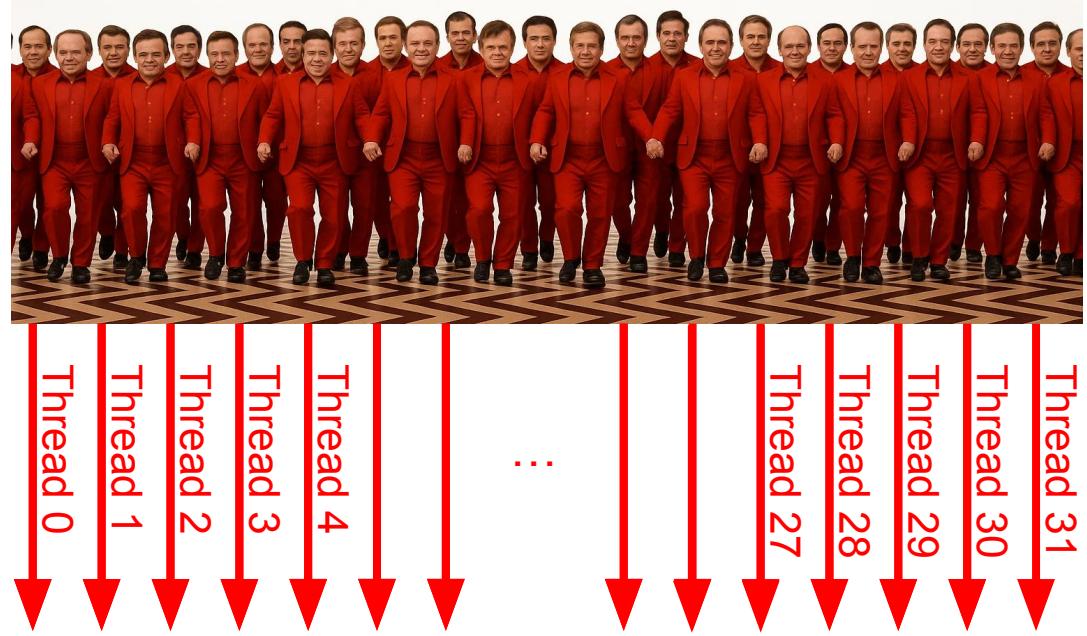


Один Instruction pointer  
на все потоки warp-a!

```
239     int warpThreadID = threadIdx.x % 32;  
240     → int result = 0;  
241     if (warpThreadID < 16) {  
242         result = dataA[warpThreadID];  
243     } else {  
244         result = dataB[warpThreadID];  
245     }
```

# Архитектура

## GPU warp

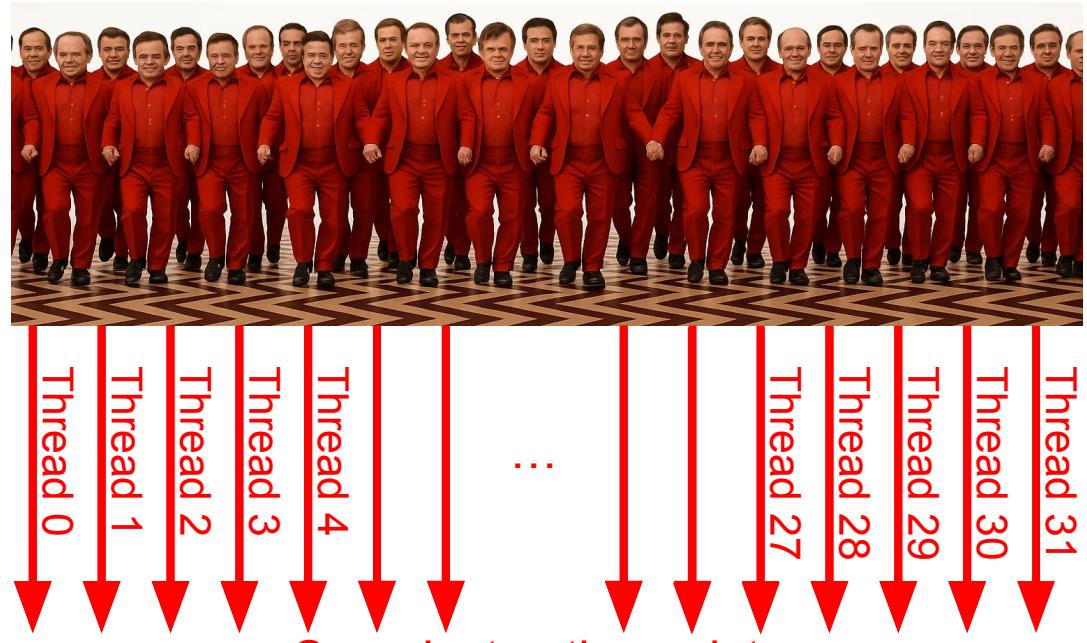


Один Instruction pointer  
на все потоки warp-a!

```
239 int warpThreadID = threadIdx.x % 32;  
240 int result = 0;  
241 → if (warpThreadID < 16) {  
242     result = dataA[warpThreadID];  
243 } else {  
244     result = dataB[warpThreadID];  
245 }
```

# Архитектура

## GPU warp

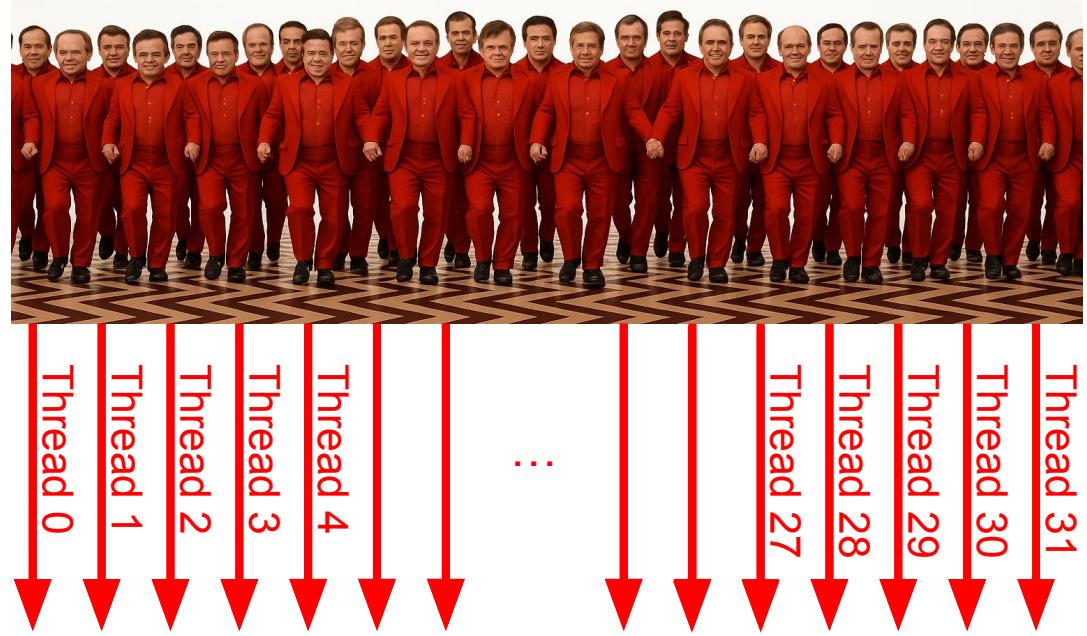


```
239 int warpThreadId = threadIdx.x % 32;
240 int result = 0;
241 if (warpThreadId < 16) {
242     result = dataA[warpThreadId];
243 } else {
244     result = dataB[warpThreadId];
245 }
```

Один Instruction pointer  
на все потоки warp-a!

# Архитектура

## GPU warp

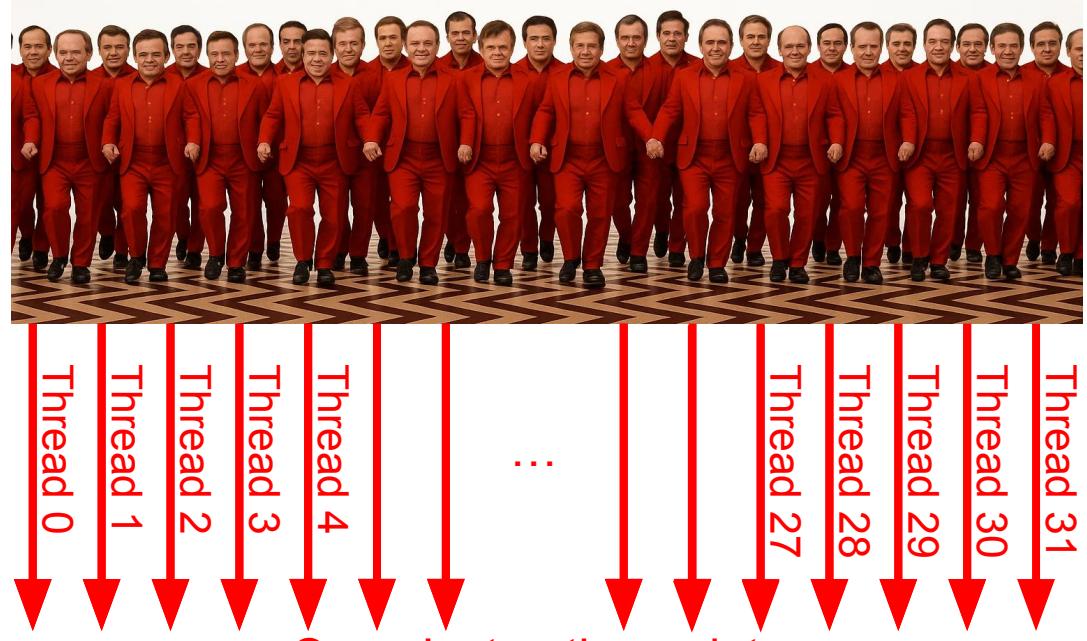


Один Instruction pointer  
на все потоки warp-a!

```
239 int warpThreadID = threadIdx.x % 32;  
240 int result = 0;  
241 if (warpThreadID < 16) {  
242     result = dataA[warpThreadID];  
243 } else {  
244     result = dataB[warpThreadID];  
245 }
```

# Архитектура

## GPU warp



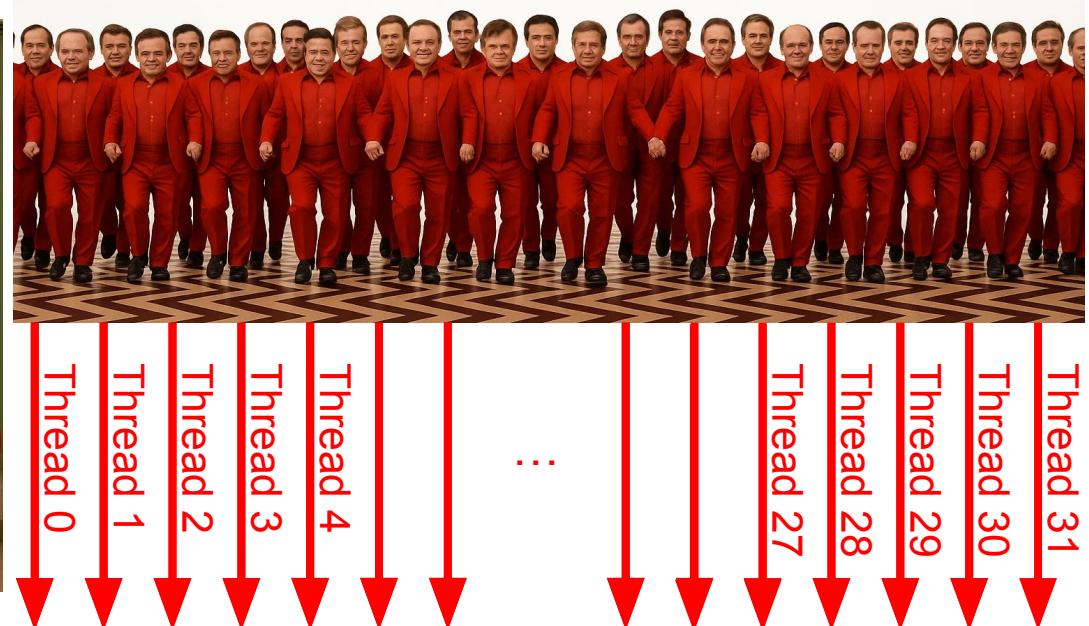
```
239 int warpThreadID = threadIdx.x % 32;
240 int result = 0;
241 if (warpThreadID < 16) {
242     result = dataA[warpThreadID];
243 } else {
244     result = dataB[warpThreadID];
245 }
```

# Архитектура

## GPU warp



```
239     int warpThreadID = threadIdx.x % 32;  
240     int result = 0;  
241     if (warpThreadID < 16) {  
242         result = dataA[warpThreadID];  
243     } else {  
244         result = dataB[warpThreadID];  
245     }
```

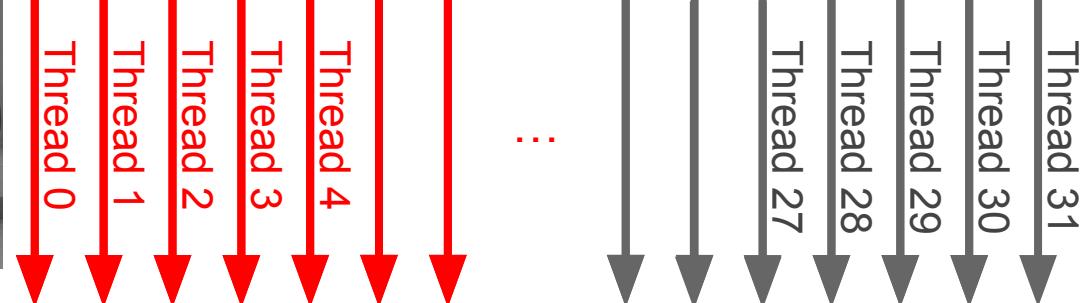


Один Instruction pointer  
на все потоки warp-a!

Выходит зайдем в обе ветки if-а?

# Архитектура

## GPU warp



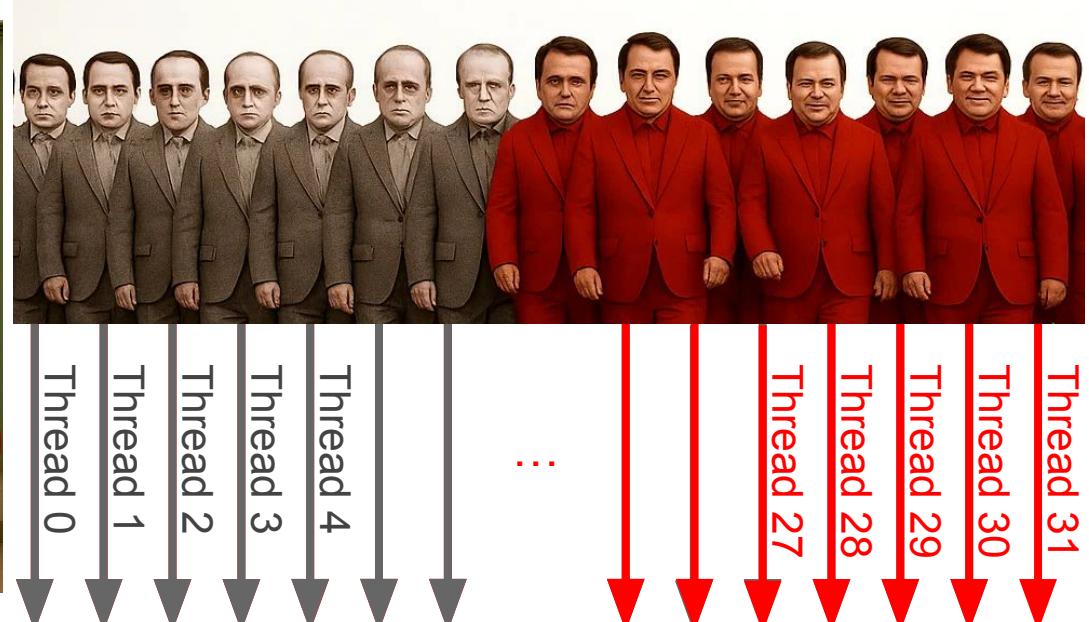
```
239 int warpThreadID = threadIdx.x % 32;
240 int result = 0;
241 if (warpThreadID < 16) {
242     result = dataA[warpThreadID]; // exec mask: [++++ +++++ +++++ ++++ - - - - - - - -]
243 } else {
244     result = dataB[warpThreadID]; // exec mask: [---- - - - - - - - - + + + + + + + + + + + +]
245 }
```

# Архитектура



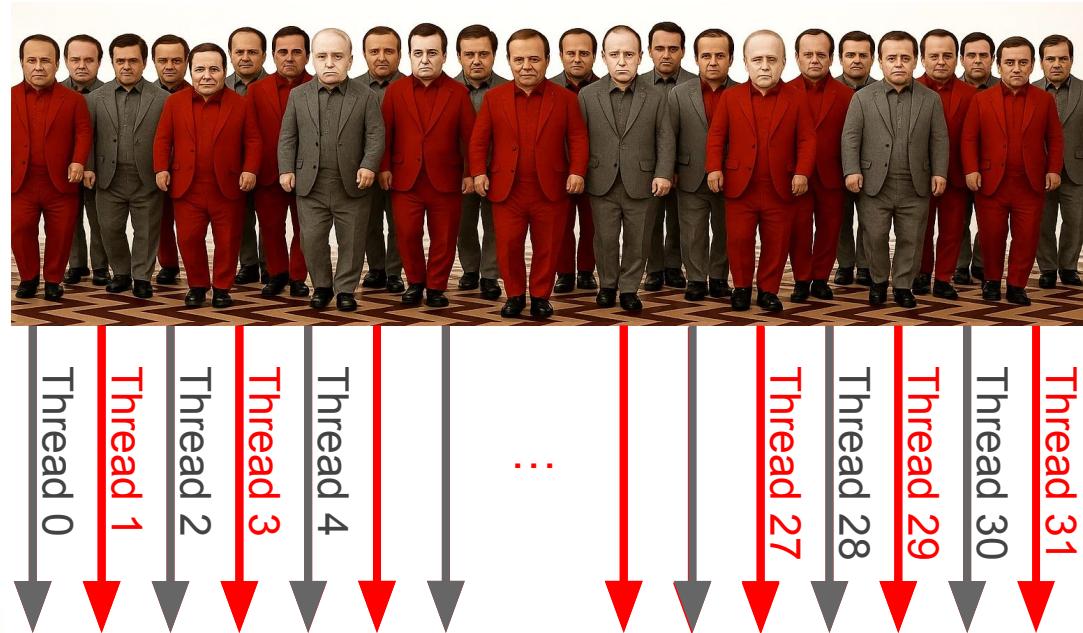
```
239 int warpThreadID = threadIdx.x % 32;  
240 int result = 0;  
241 if (warpThreadID < 16) {  
    result = dataA[warpThreadID]; // exec mask: [+++++ +++++ +++++ ++++]  
} else {  
    result = dataB[warpThreadID]; // exec mask: [----- ----- ----- +++++ +++++ +++++ +++++]  
}
```

## GPU warp



# Архитектура

## GPU warp



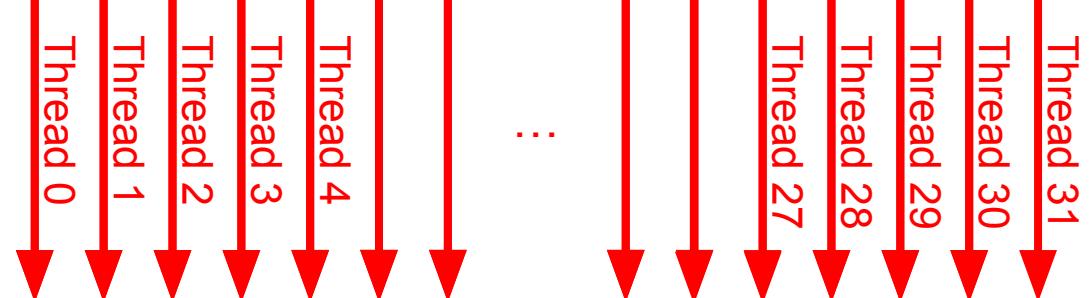
```
239 int warpThreadID = threadIdx.x % 32;
240 int result = 0;
241 if (warpThreadID % 2 == 0) {
242     result = dataA[warpThreadID]; // exec mask: [+--- +--- +--- +--- +--- +--- +--- +--- +--- +---]
243 } else {
244     result = dataB[warpThreadID]; // exec mask: [-++ -++ -++ -++ -++ -++ -++ -++ -++ -++]
245 }
```

# Архитектура

```
241 if (predicate1) {  
242     if (predicate2) {  
243         if (predicate3) {  
244             // A1  
245         } else {  
246             // A2  
247         }  
248     } elif (predicate4) {  
249         // A3  
250     }  
251 } else {  
252     // A4  
253 }
```

Сколько “времени” будет работать warp?

## GPU warp



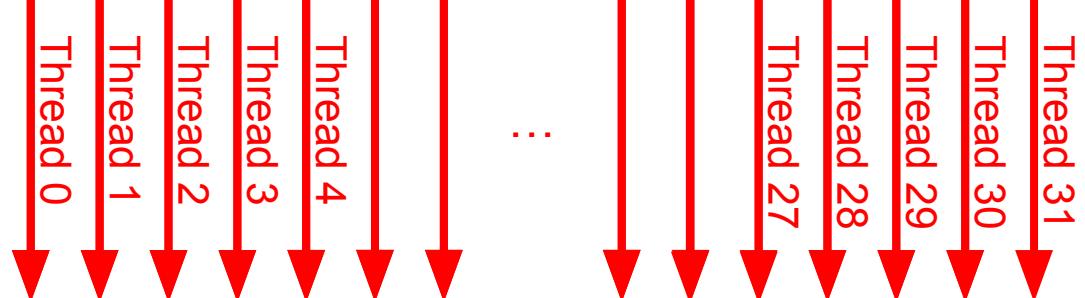
Один Instruction pointer  
на все потоки warp-a!

# Архитектура

```
241 if (predicate1) {  
242     if (predicate2) {  
243         if (predicate3) {  
244             // A1  
245         } else {  
246             // A2  
247         }  
248     } elif (predicate4) {  
249         // A3  
250     }  
251 } else {  
252     // A4  
253 }
```

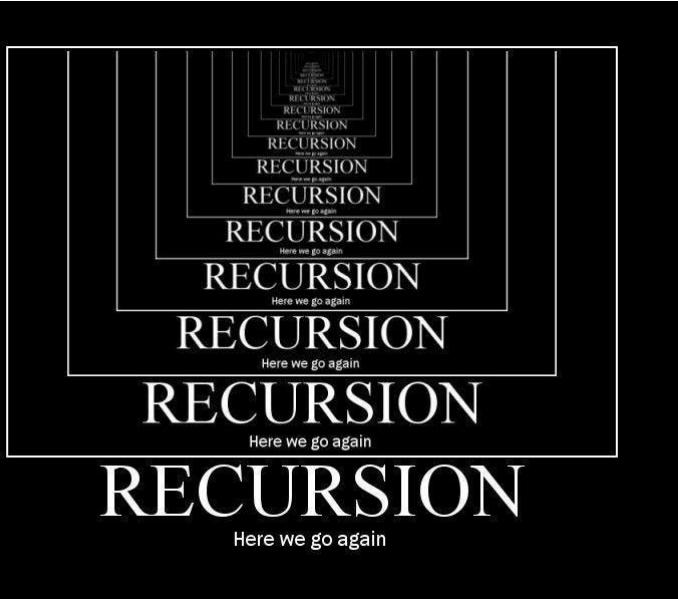
Сколько “времени” будет работать warp?  
 $A1 + A2 + A3 + A4$  при **code divergence**

## GPU warp

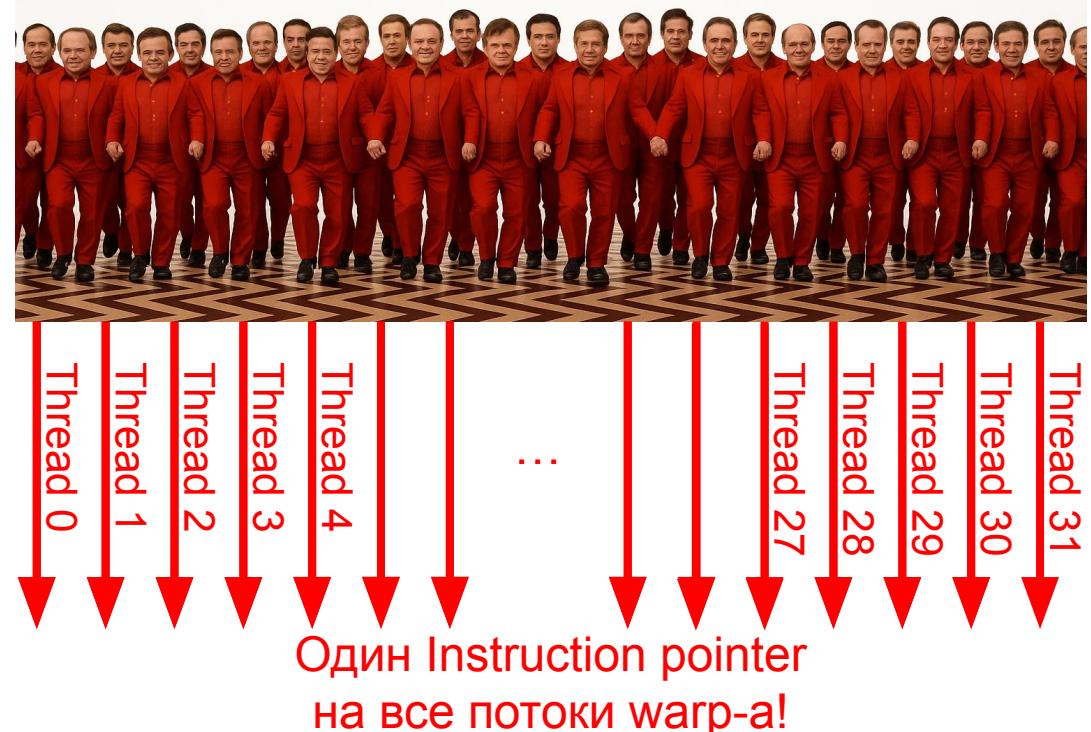


Один Instruction pointer  
на все потоки warp-a!

# Архитектура

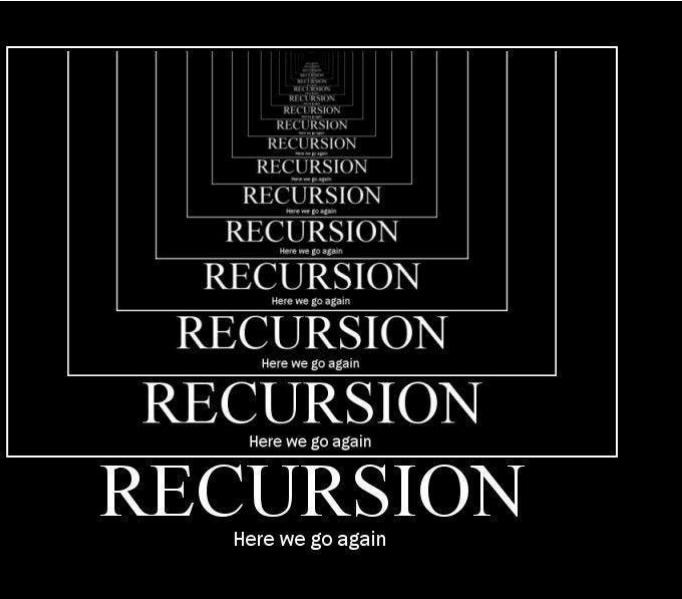


## GPU warp

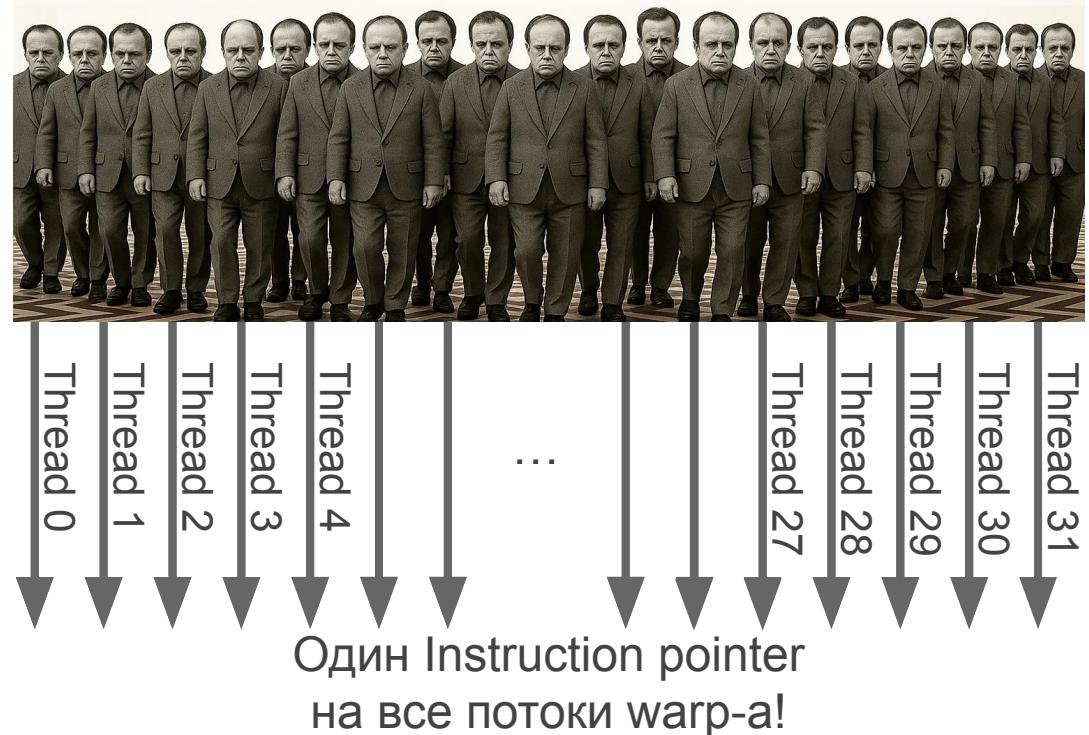


Сколько “времени” будет работать достаточно глубокая рекурсия?

# Архитектура



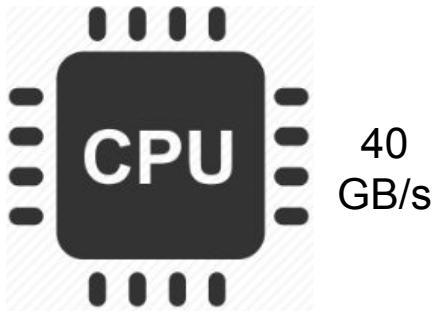
## GPU warp



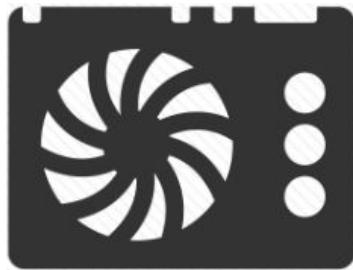
# Глава 2: Работа с памятью

hyper-threading, latency hiding, occupancy, registers pressure/spilling,  
coalesced memory access pattern, cache lines, local/shared memory, bank conflicts

# Архитектура



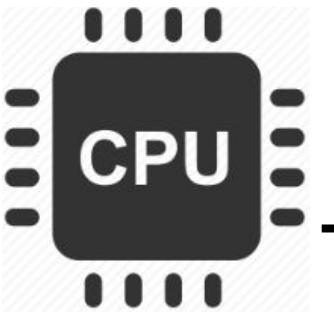
40  
GB/s



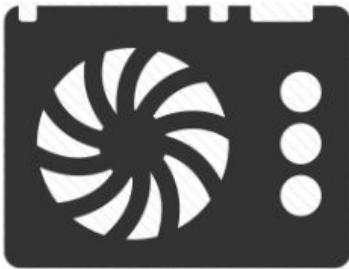
1000  
GB/s

GPU

# Архитектура



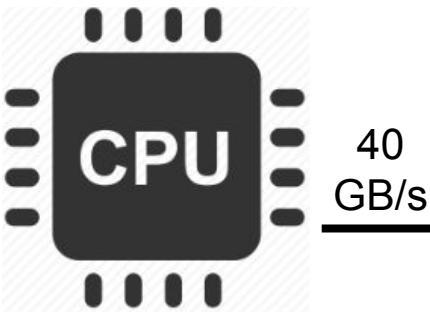
40  
GB/s



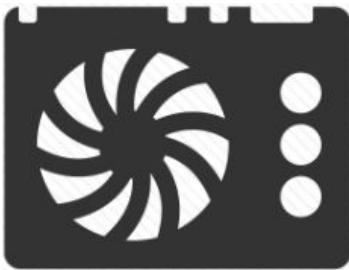
1000  
GB/s

GPU

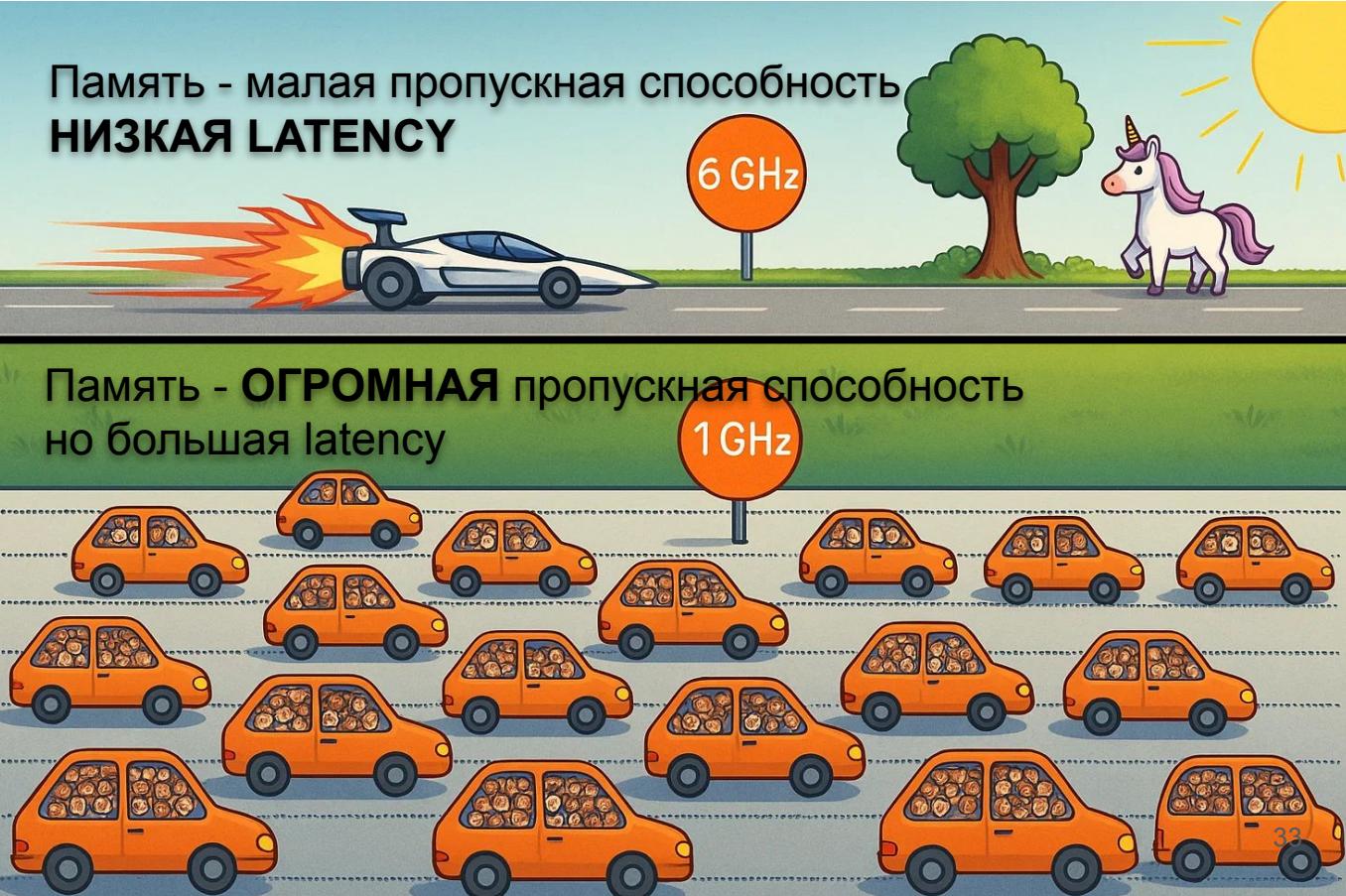
# Архитектура



40  
GB/s

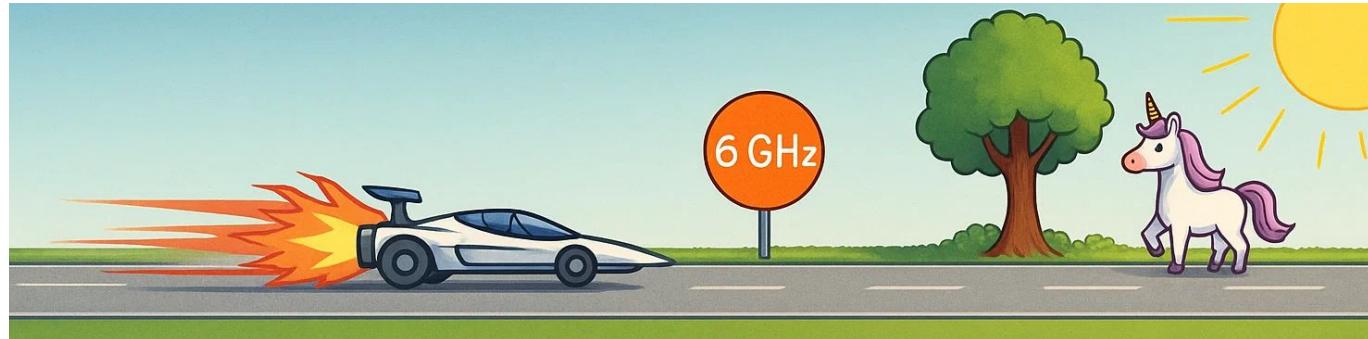
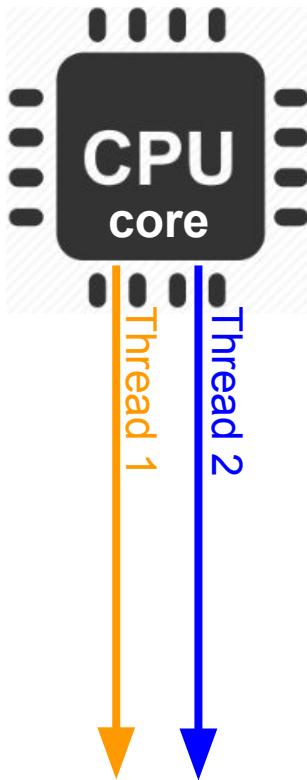


1000  
GB/s



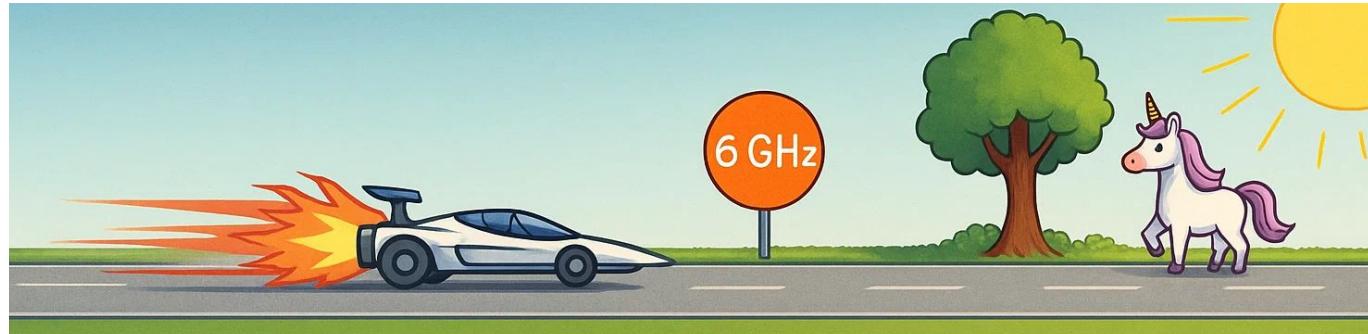
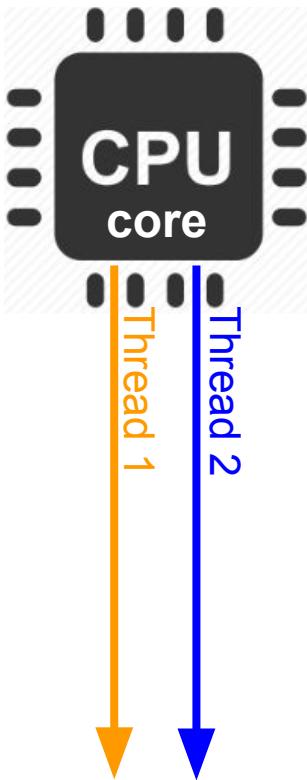
GPU

# Архитектура CPU: многопоточность



Многопоточность благодаря переключению контекста!  
Что нужно перещелкнуть в состоянии ЦПУ ядра для другого потока?

# Архитектура CPU: многопоточность

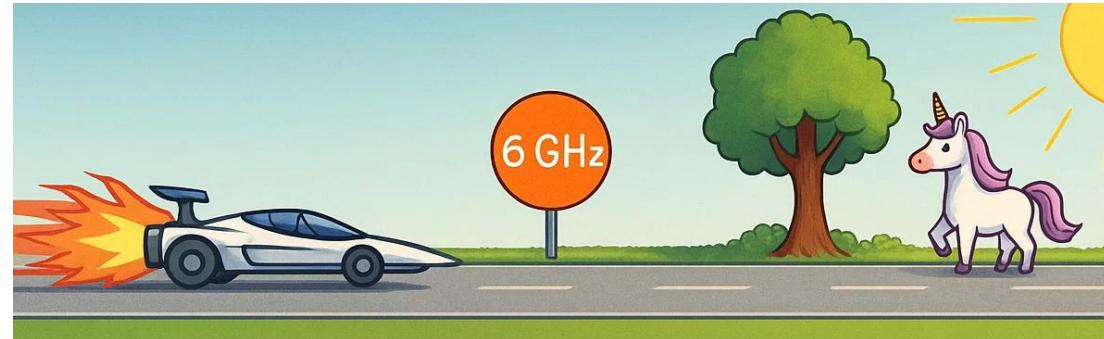
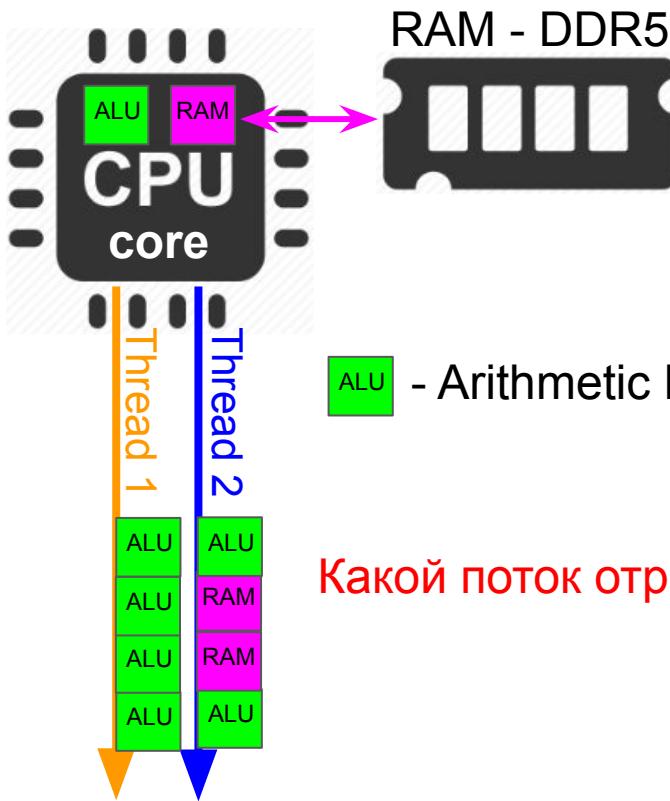


Многопоточность благодаря переключению контекста!

## Context switch:

- Обновляет Instruction pointer (указатель на строку кода)
- Подгружает значения регистров процессора

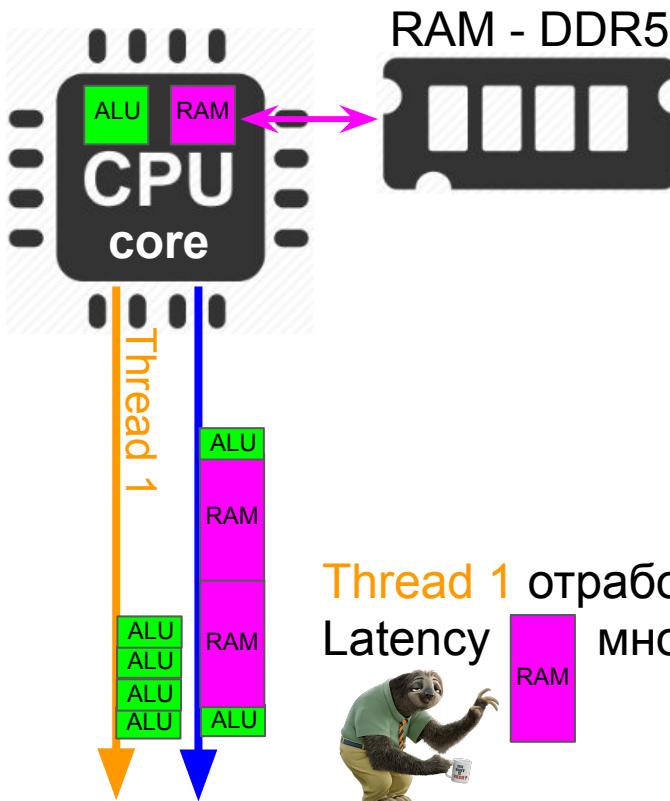
# Архитектура CPU: Hyper-Threading, SMT



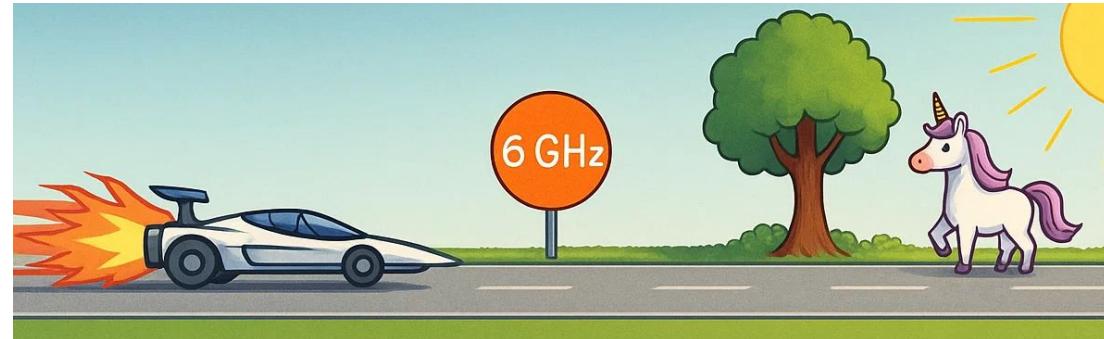
ALU - Arithmetic Logical Unit

Какой поток отработает быстрее?

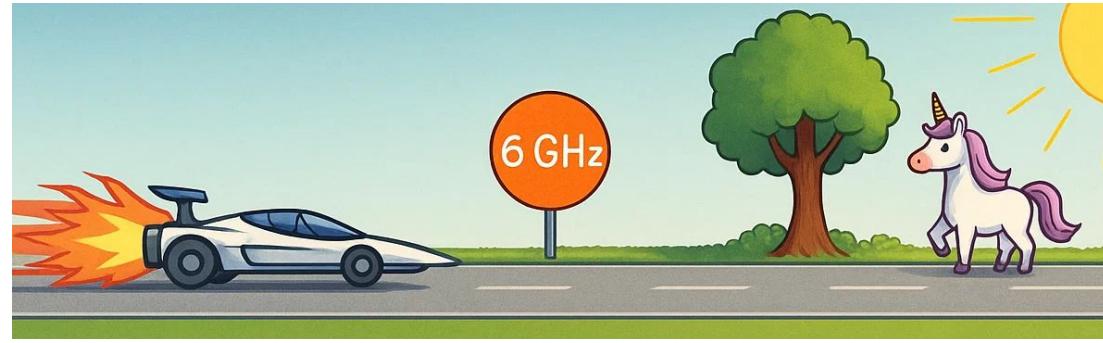
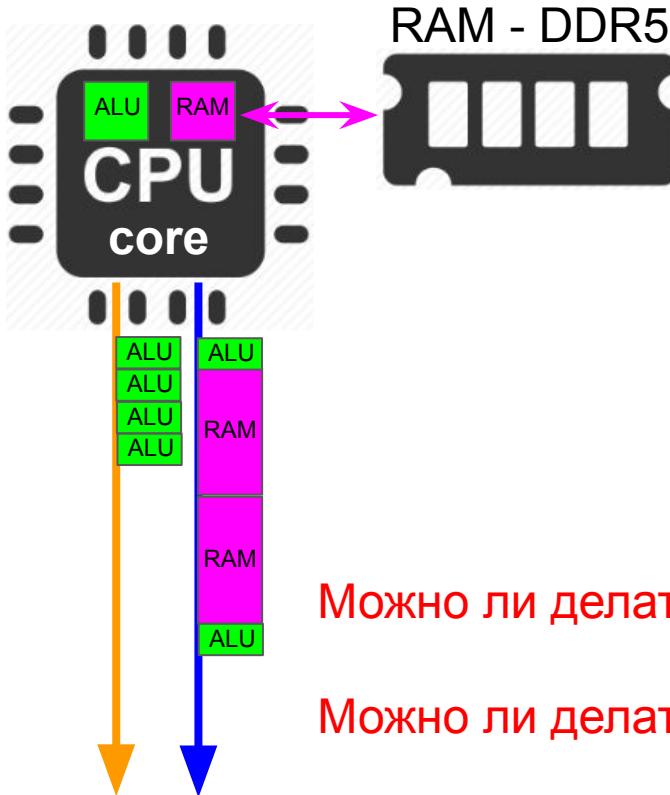
# Архитектура CPU: Hyper-Threading, SMT



Thread 1 отработает быстрее!  
много больше чем у



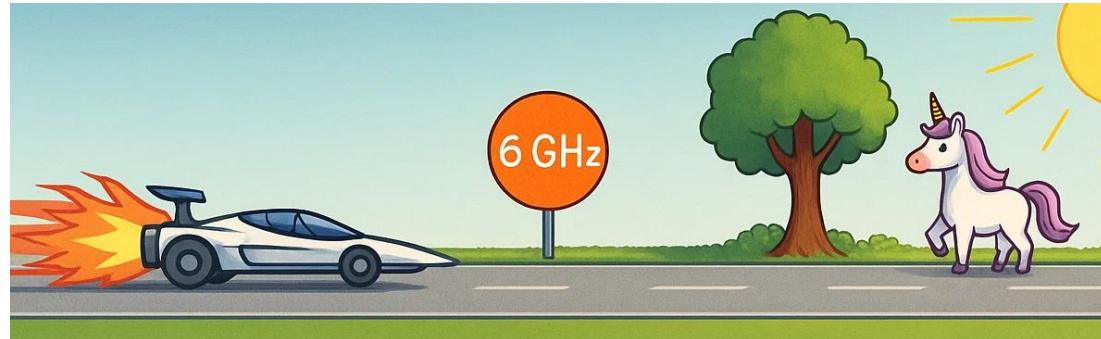
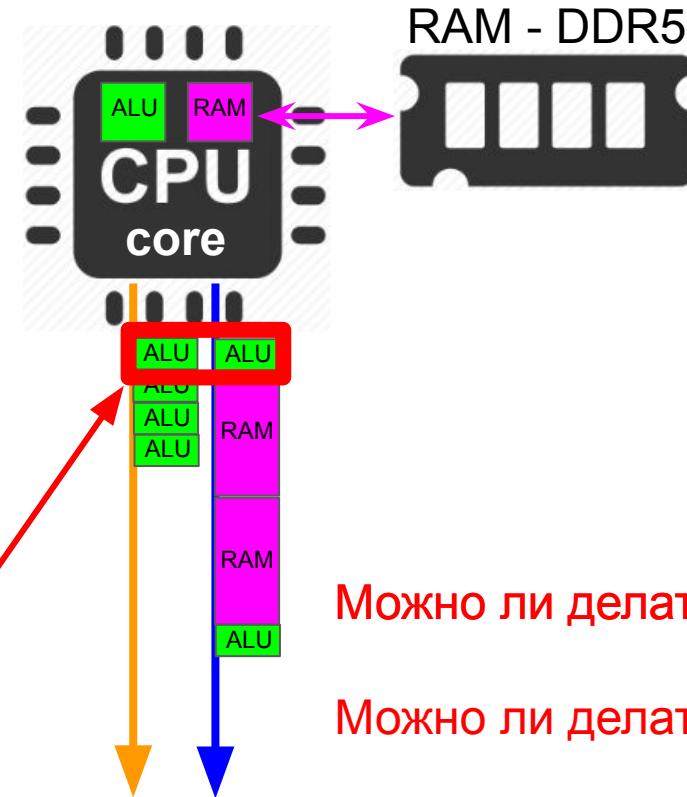
# Архитектура CPU: Hyper-Threading, SMT



Можно ли делать **ALU** пока ждем **RAM** ?

Можно ли делать **ALU** пока ждем **ALU** ?

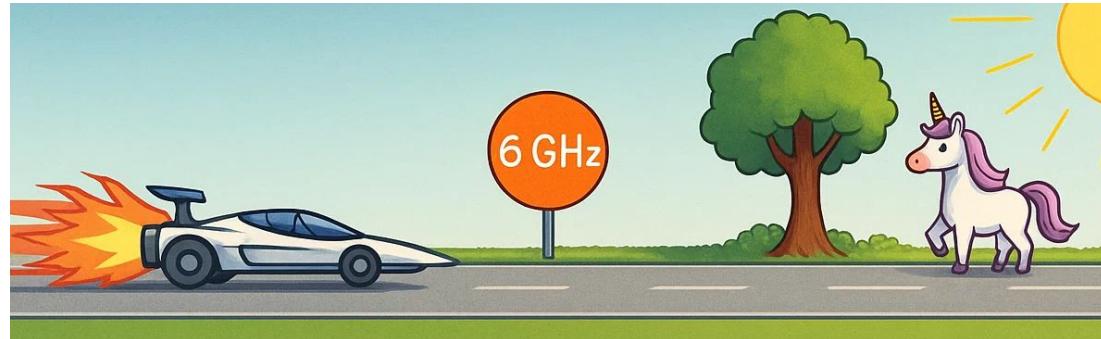
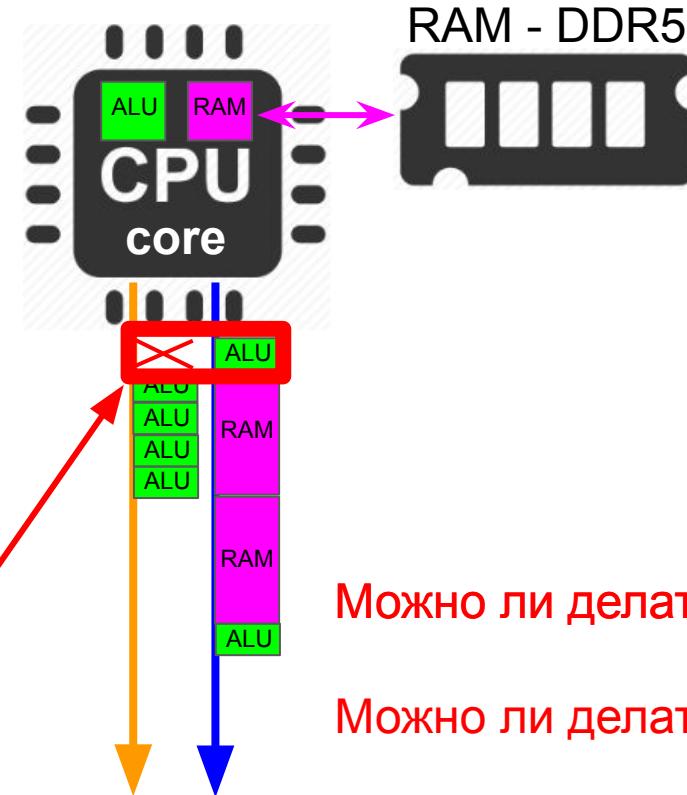
# Архитектура CPU: Hyper-Threading, SMT



Можно ли делать **ALU** пока ждем **RAM** ?

Можно ли делать **ALU** пока ждем **ALU** ?

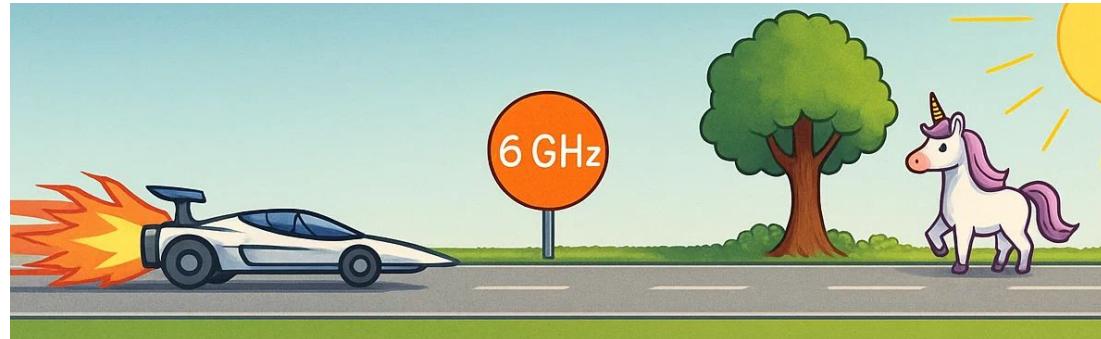
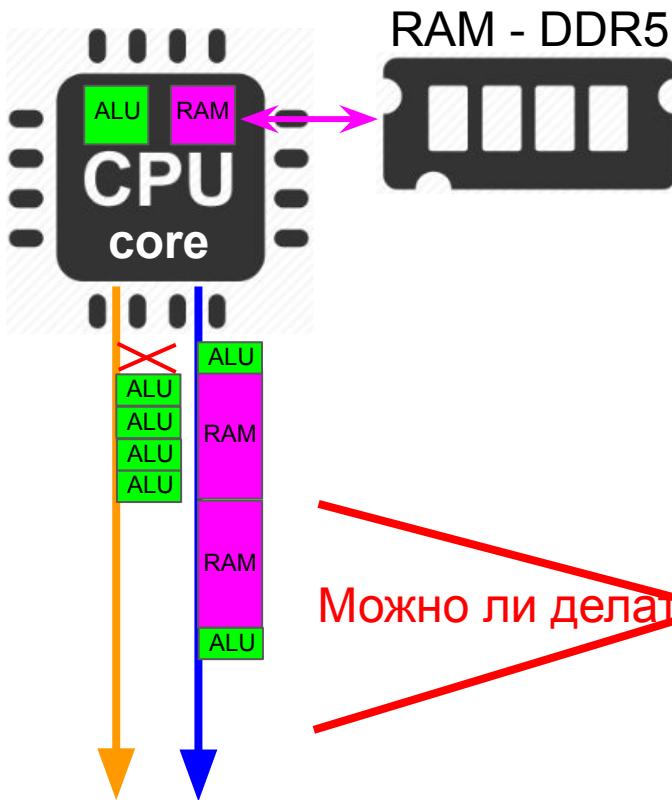
# Архитектура CPU: Hyper-Threading, SMT



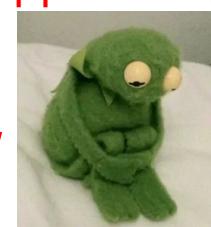
Можно ли делать **ALU** пока ждем **RAM** ?

Можно ли делать **ALU** пока ждем **ALU** ?

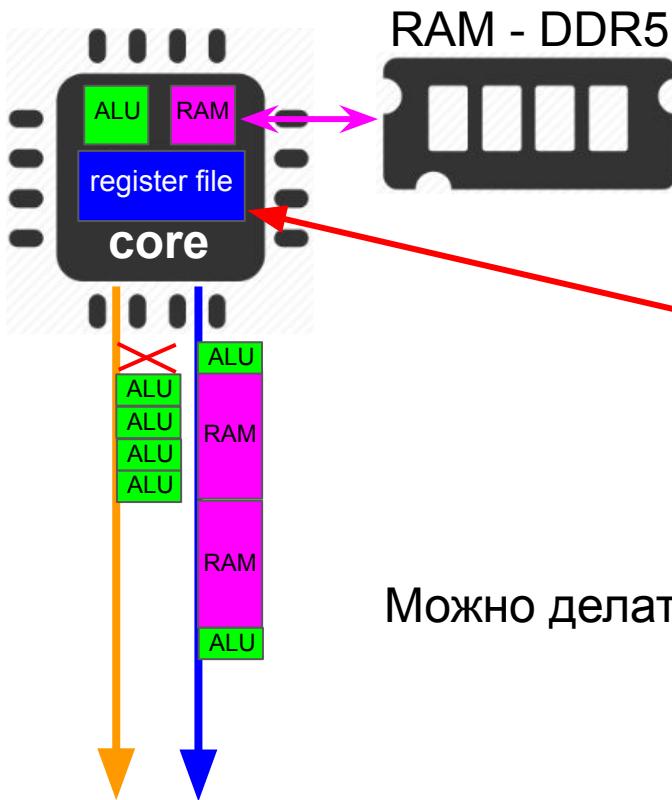
# Архитектура CPU: Hyper-Threading, SMT



Context switch - дорого!  
Долго подгружает регистры!



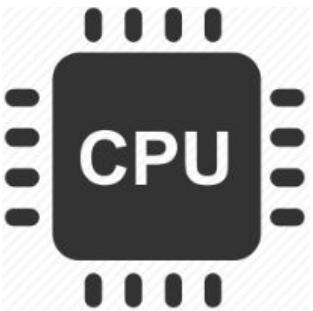
# Архитектура CPU: Hyper-Threading, SMT



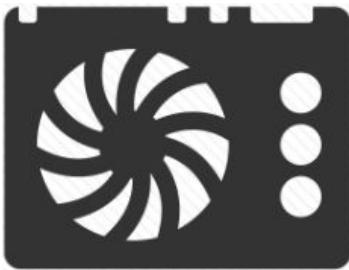
Давайте держать два множества регистров!

Можно делать **ALU** пока ждем **RAM**! Это и есть SMT и HT!

# Архитектура GPU



40  
GB/s



1000  
GB/s

Память - малая пропускная способность  
**НИЗКАЯ LATENCY**

6 GHz

Память - **ОГРОМНАЯ** пропускная способность  
но большая latency

1 GHz



# Архитектура GPU

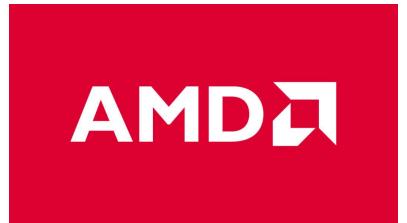


32 слабых медленных CUDA ядер

warp



# Архитектура GPU



*wavefront*  
*64 threads*



*warp*  
*32 threads*

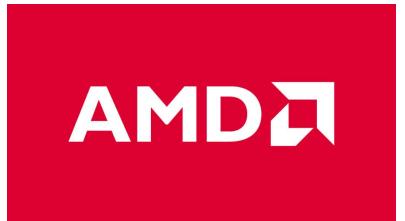


32 слабых медленных  
CUDA ядер

*warp*



# Архитектура GPU



*wavefront*

*64 threads*

**Compute Units  
(CU)**



*warp*

*32 threads*

**Streaming Multiprocessor  
(SM)**



32 слабых медленных  
CUDA ядер

*warp*



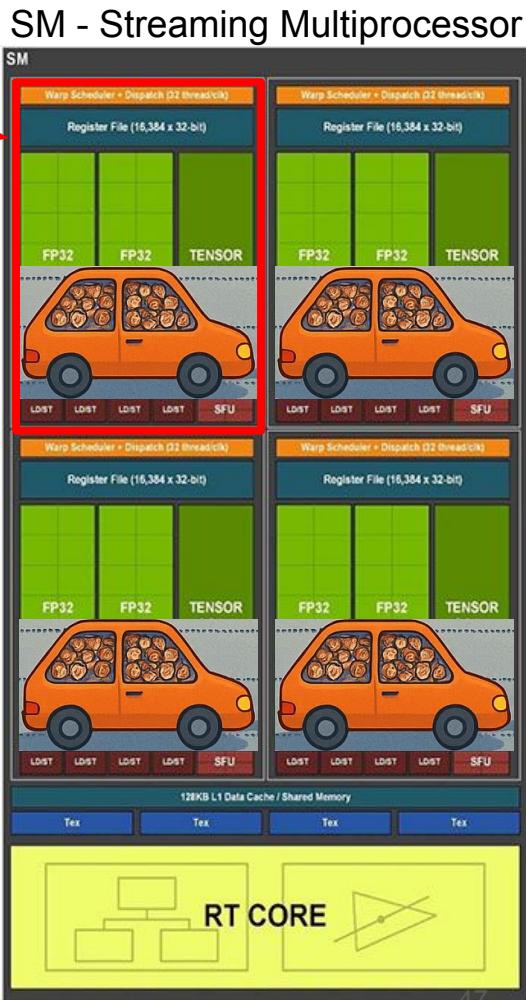
# Архитектура GPU



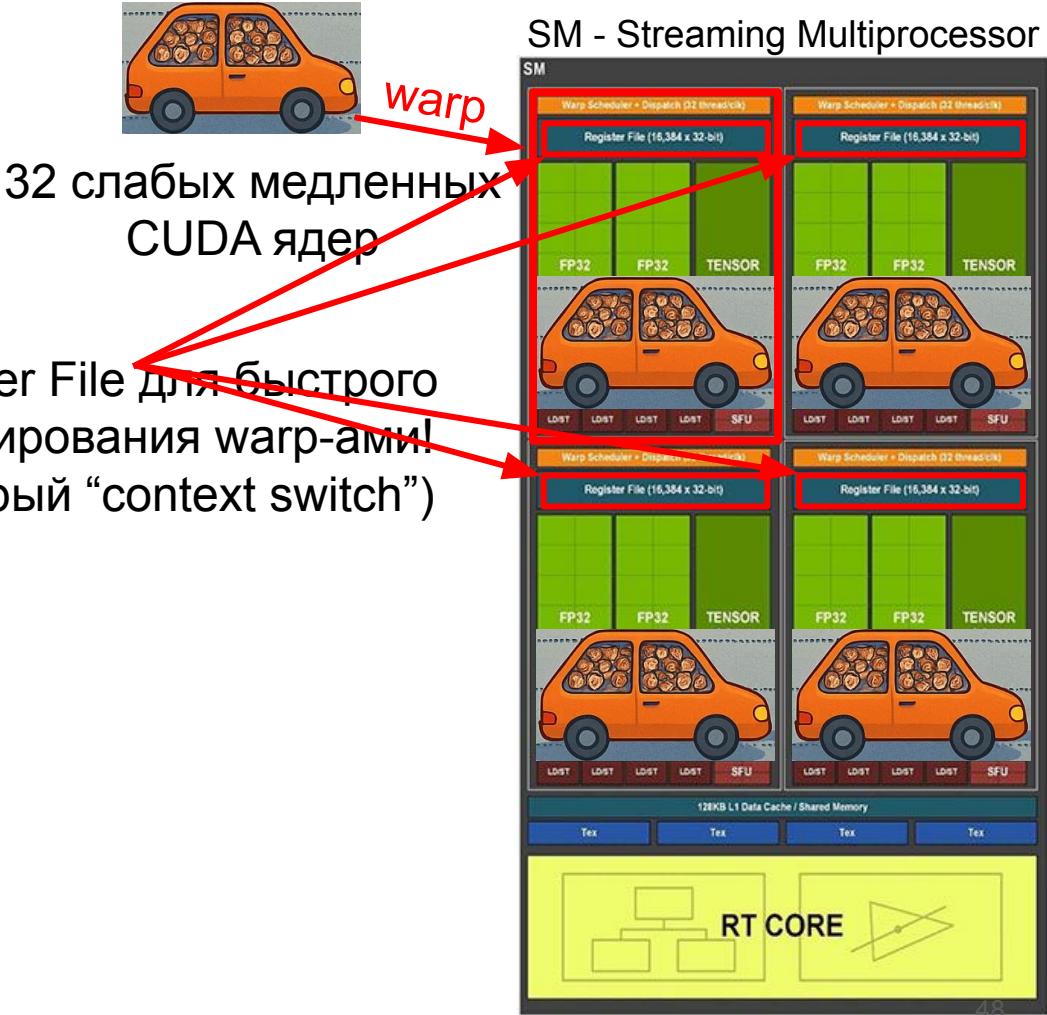
32 слабых медленных CUDA ядер



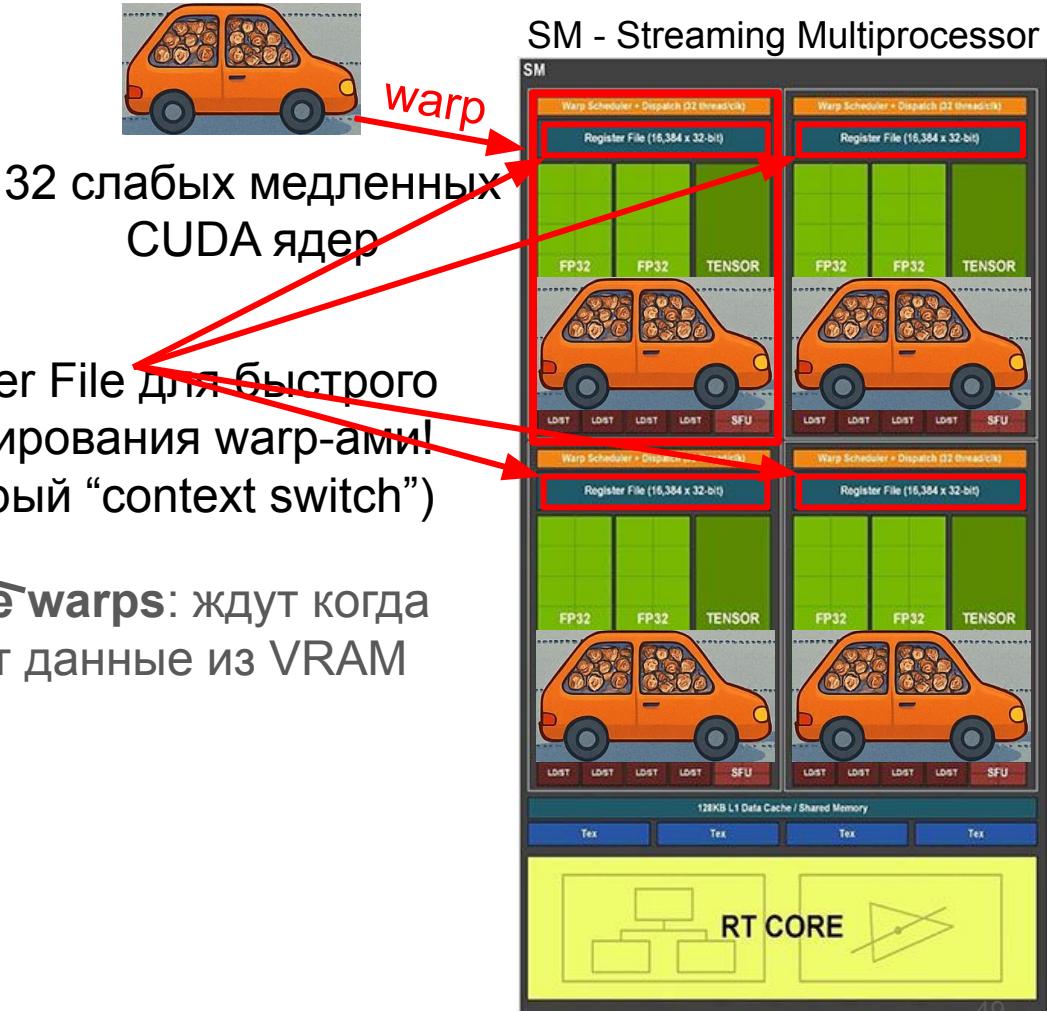
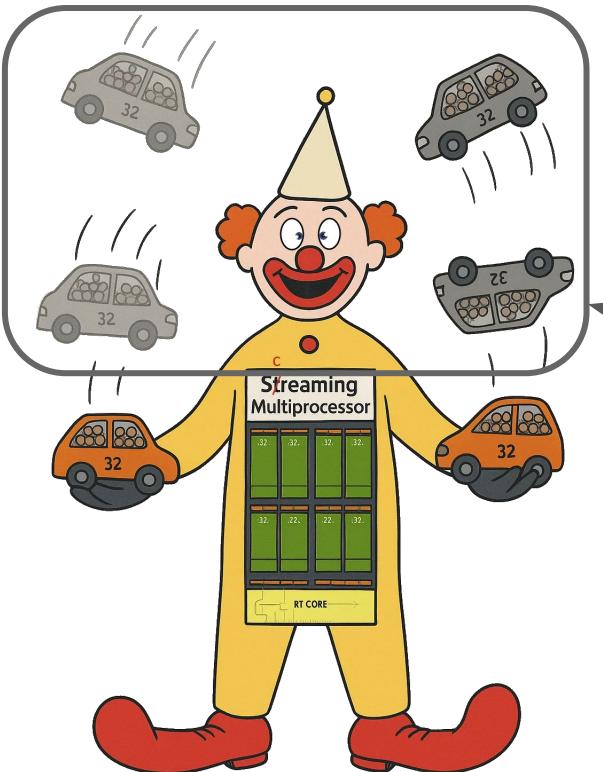
warp



# Архитектура GPU

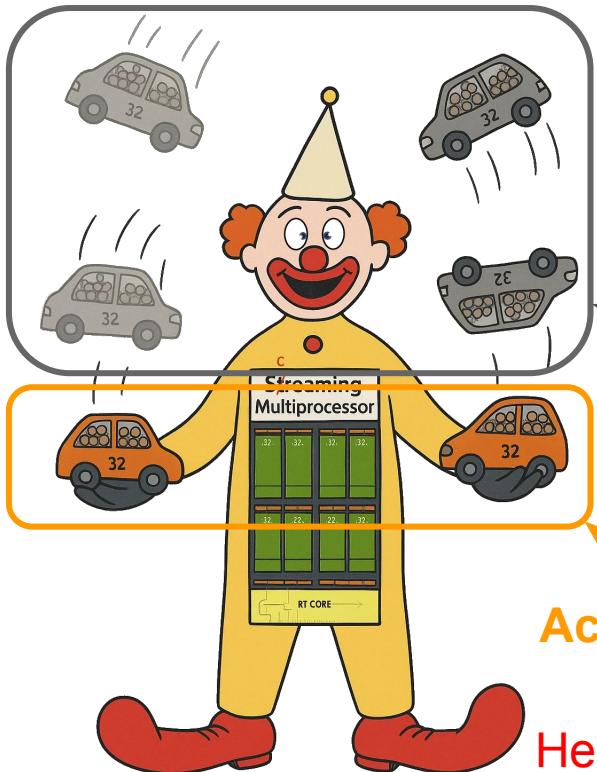


# Архитектура GPU



SM - Streaming Multiprocessor

# Архитектура GPU



Register File для быстрого  
жонглирования warp-ами!  
(быстрый “context switch”)

Inactive warps: ждут когда  
придут данные из VRAM

Active warps: данные в регистрах  
Работаем-работаем! 😎

Не напоминает ли вам это что-то?

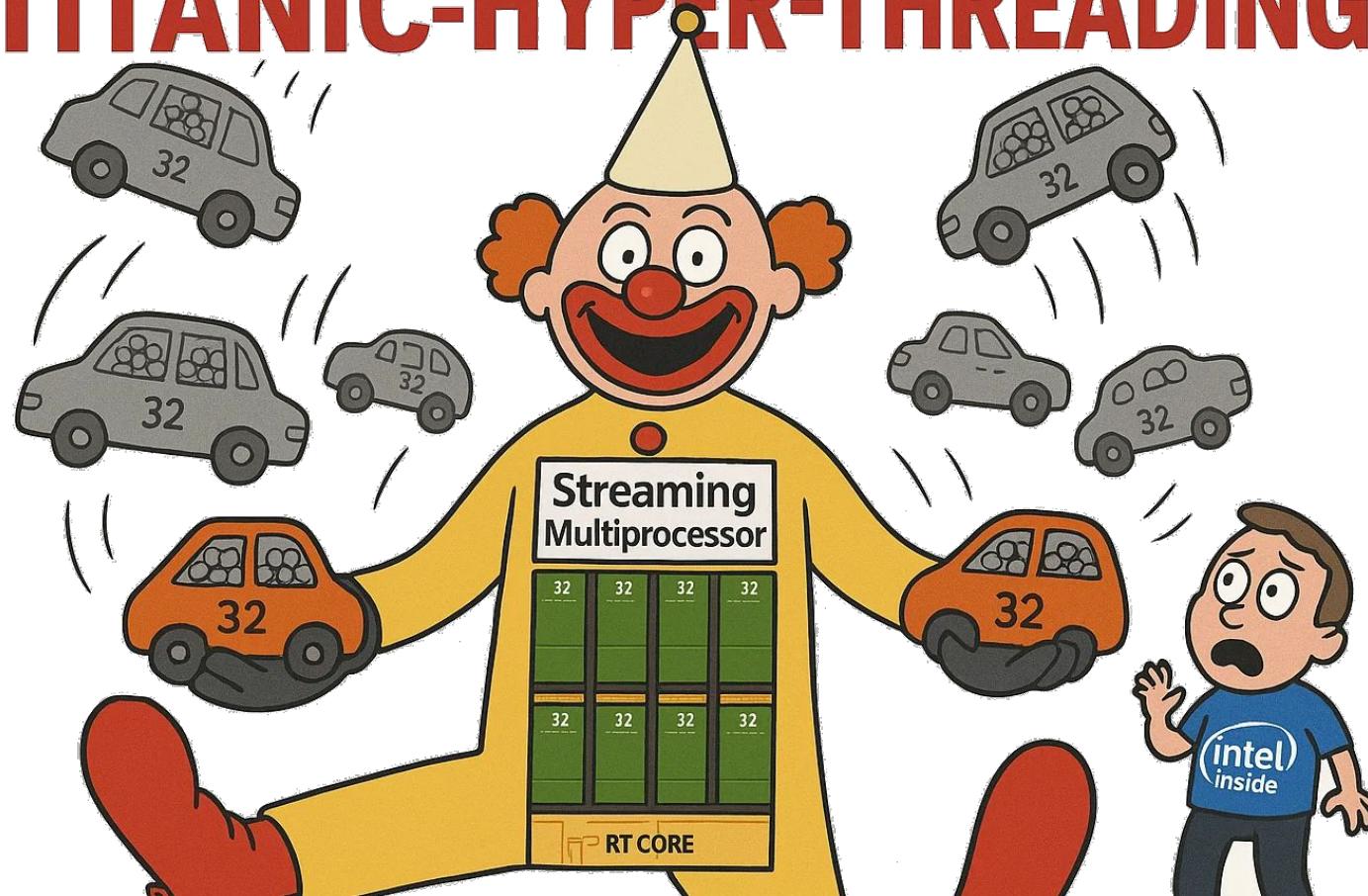


warp

32 слабых медленных  
CUDA ядер



# ULTRA SUPER MEGA GIGA TITANIC-HYPER-THREADING



# Архитектура GPU



32 слабых медленных CUDA ядер

Register File для быстрого жонглирования warp-ами!  
(быстрый “context switch”)

**Occupancy** = насколько много доступно warp-ов для жонглирования.



warp



# Архитектура GPU



32 слабых медленных CUDA ядер

Register File для быстрого жонглирования warp-ами!  
(быстрый “context switch”)

**Occupancy** = насколько много доступно warp-ов для жонглирования.

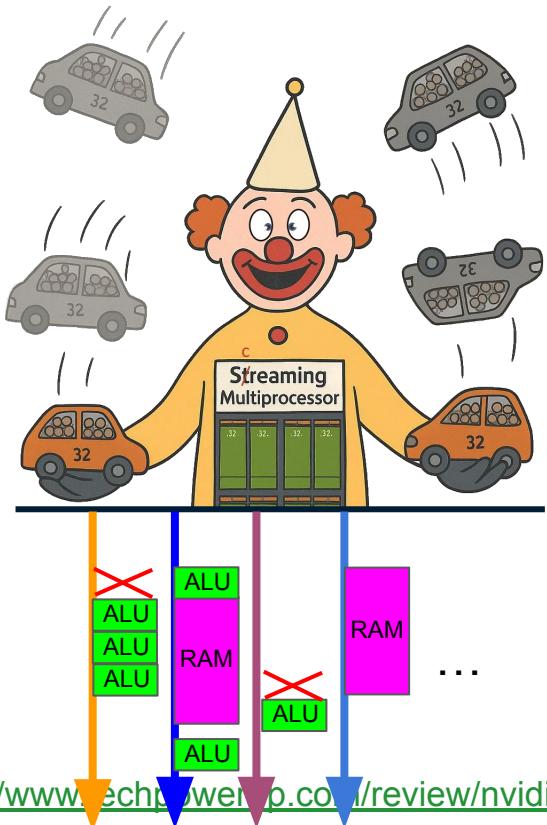
**Если occupancy высокая, то что?**



warp



# Архитектура GPU

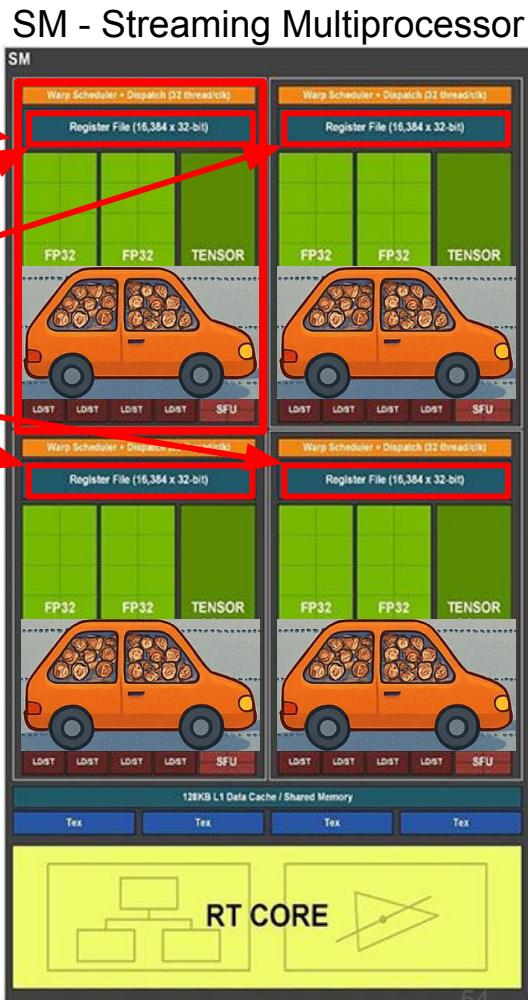


32 слабых медленных CUDA ядер

Register File для быстрого жонглирования warp-ами!  
(быстрый “context switch”)

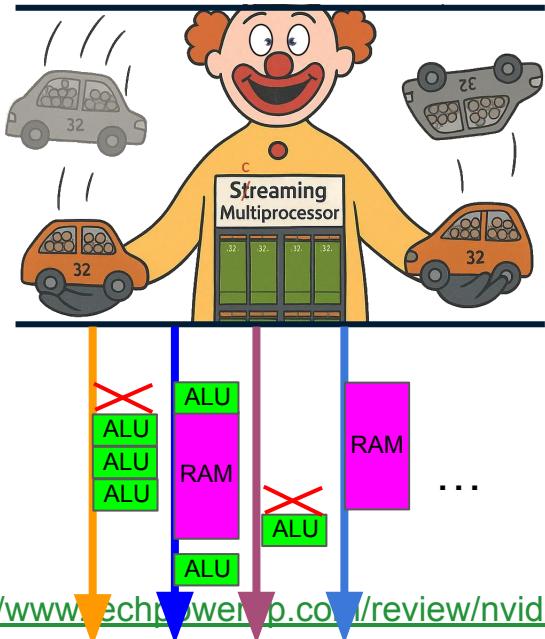
**Occupancy** = насколько много доступно warp-ов для жонглирования.

Если occupancy высокая, то **хорошо скрывается latency доступа к памяти** т.к. всегда найдется готовый warp!



# Архитектура GPU

А почему у SM может быть разное число warp-ов?



32 слабых медленных CUDA ядер

warp



Register File для быстрого жонглирования warp-ами!  
(быстрый “context switch”)

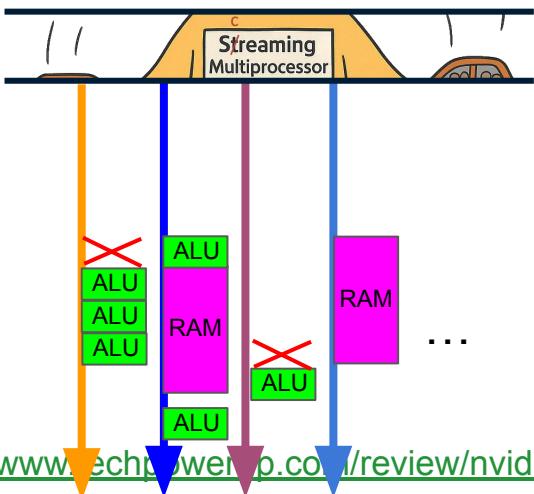
**Occupancy** = насколько много доступно warp-ов для жонглирования.

Если occupancy высокая, то **хорошо скрывается latency доступа к памяти** т.к. всегда найдется готовый warp!

# Архитектура GPU

А почему у SM может быть разное число warp-ов?

Registers Pressure!



32 слабых медленных CUDA ядер

warp



Occupancy = насколько много доступно warp-ов для жонглирования.

Если occupancy высокая, то **хорошо скрывается latency доступа к памяти** т.к. всегда найдется готовый warp!

# Архитектура GPU

А почему у SM может быть разное число warp-ов?

**Registers Pressure!**

Register File для быстрого жонглирования warp-ами!  
(быстрый “context switch”)

32 слабых медленных CUDA ядер

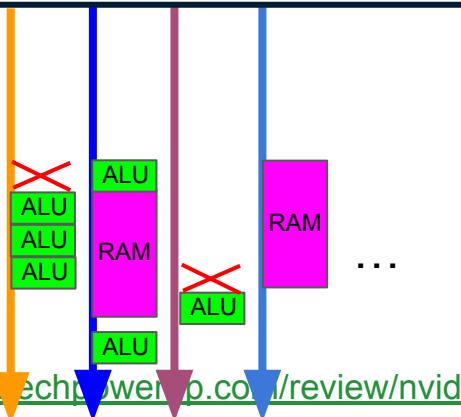


warp

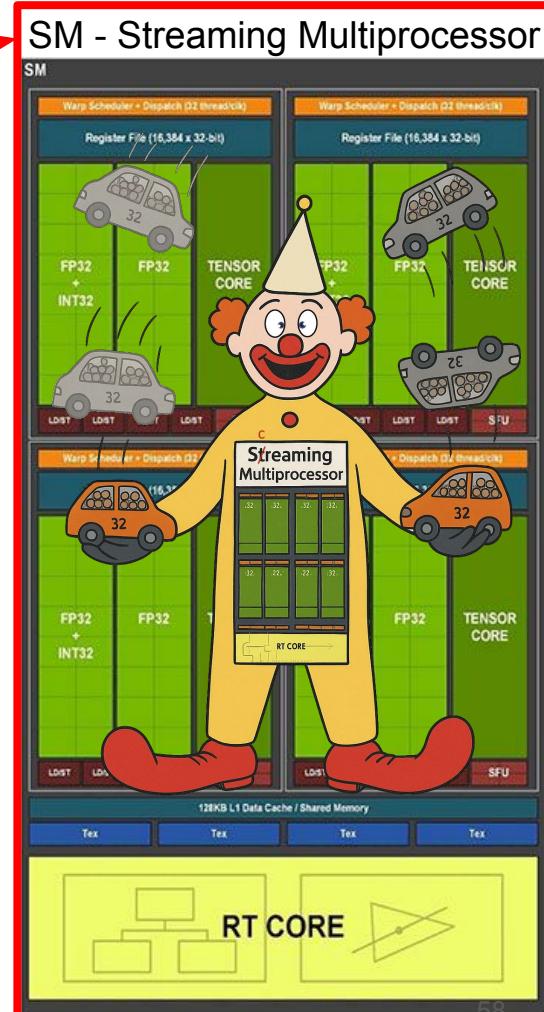
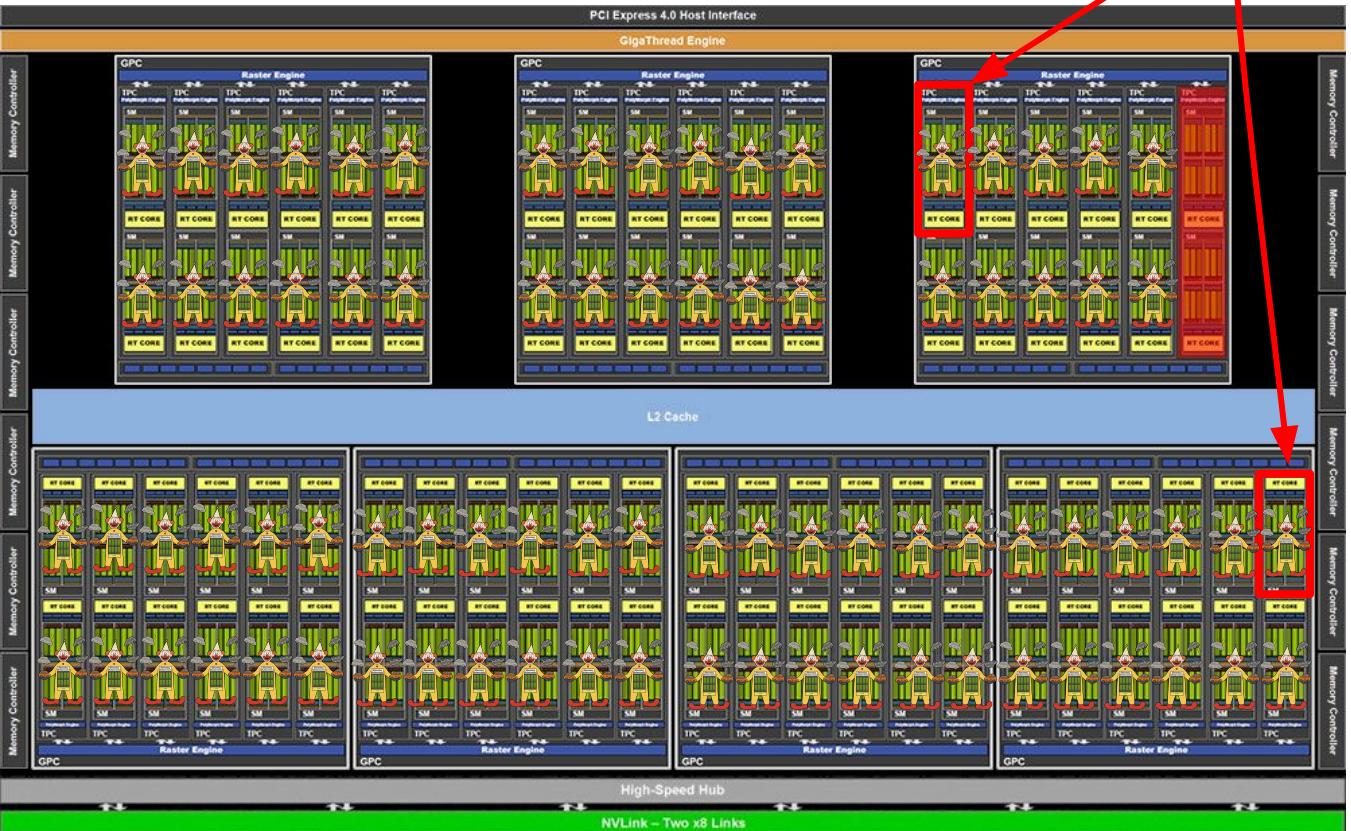


Вплоть до Registers Spilling! Occupancy = насколько много доступно warp-ов для жонглирования.

Если occupancy высокая, то **хорошо скрывается latency доступа к памяти** т.к. всегда найдется готовый warp!

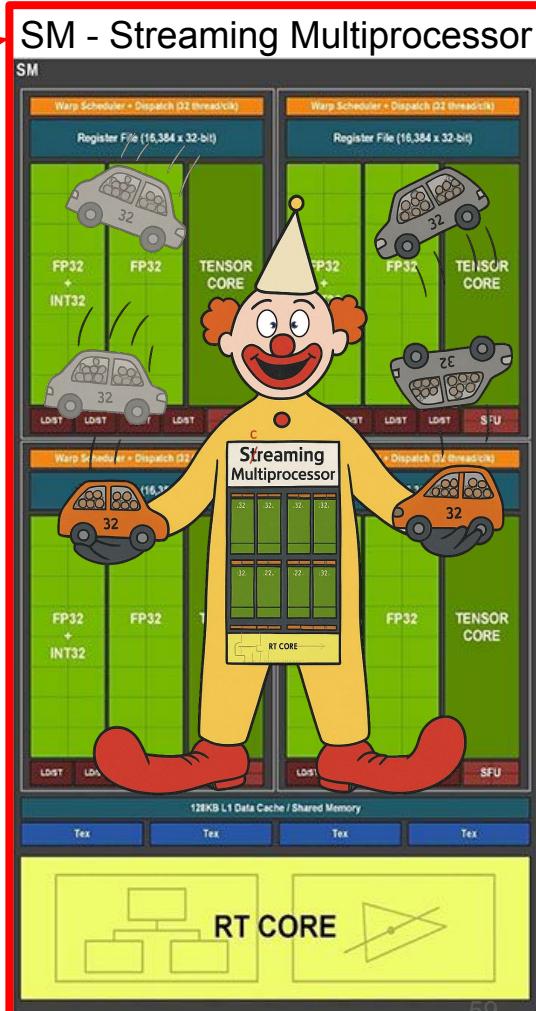


RTX 3090: 10496 CUDA cores = 82 SM · 4 warps · 32 ALUs



RTX 3090: 10496 CUDA cores = 82 SM · 4 warps · 32 ALUs

Куда сбежали два клоуна?



# Архитектура VRAM (**coalesced** memory access pattern)



```
float value = data[index];
```

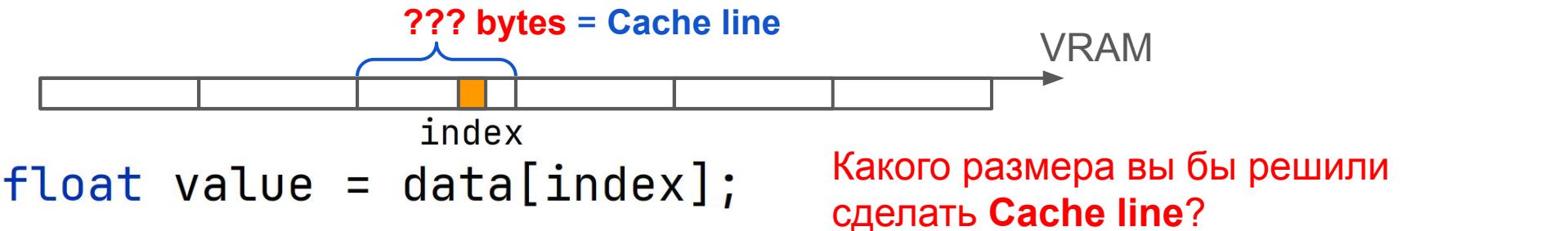
# Архитектура VRAM (coalesced memory access pattern)



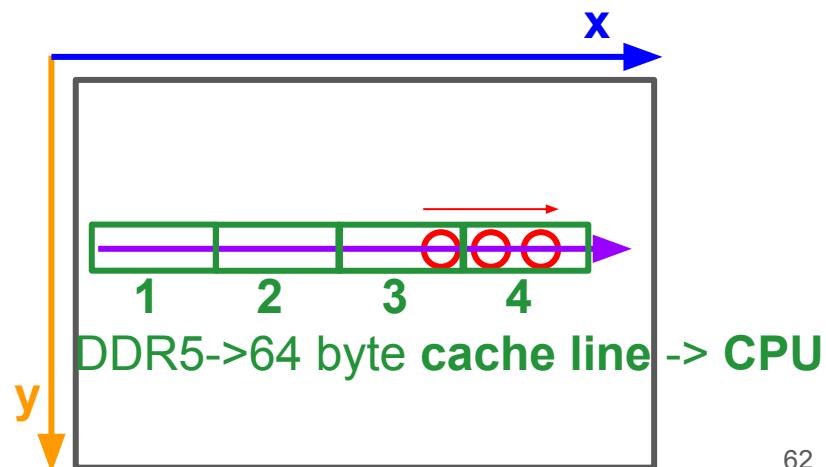
```
float value = data[index];
```

Какого размера вы бы решили  
сделать Cache line?

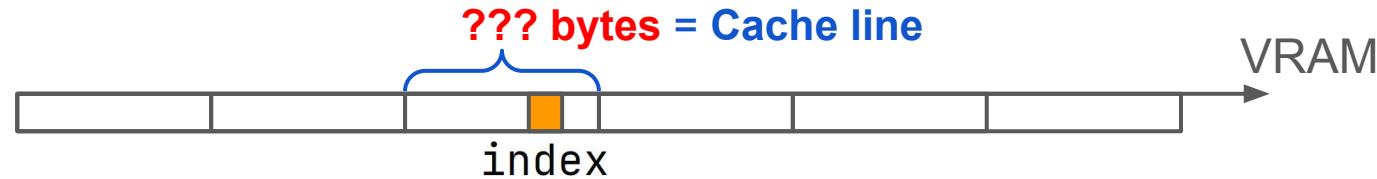
# Архитектура VRAM (coalesced memory access pattern)



Напоминание про CPU:



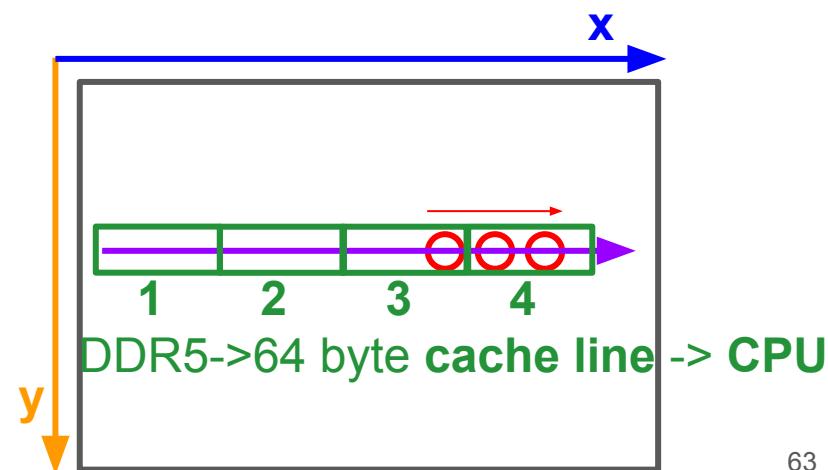
# Архитектура VRAM (coalesced memory access pattern)



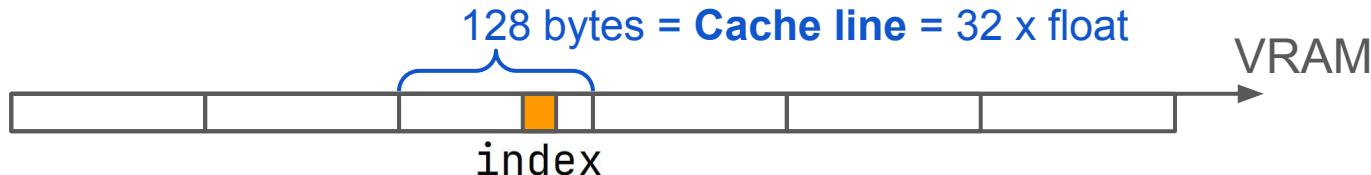
```
float value = data[index];
```

Какого размера вы бы решили  
сделать Cache line?

Напоминание про CPU:

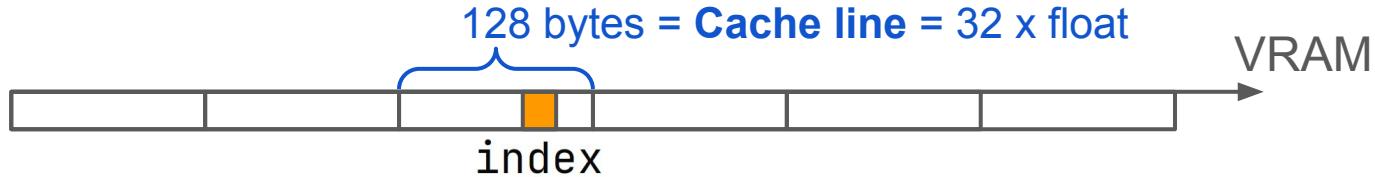


# Архитектура VRAM (coalesced memory access pattern)



```
float value = data[index];
```

# Архитектура VRAM (coalesced memory access pattern)



```
float value = data[index];
```



# Архитектура VRAM (**coalesced** memory access pattern)

128 bytes = Cache line = 32 x float



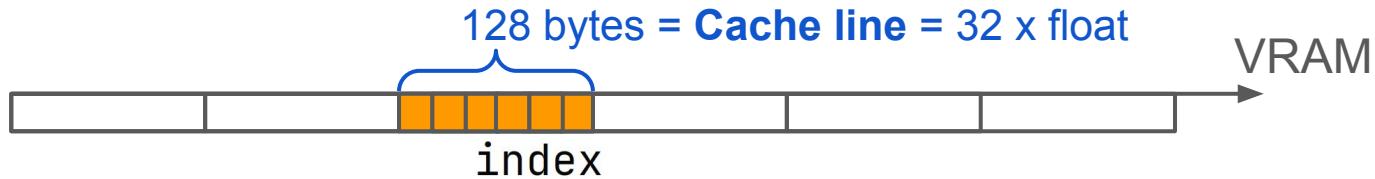
`float value = data[index];`



index	cache lines count
[1024+0; 1024+32)	???

Сколько cache line-ов чтобы покрыть заказ?  
Сколько потребуется транзакций из VRAM?

# Архитектура VRAM (coalesced memory access pattern)



`float value = data[index];`



index	cache lines count
[1024+0; 1024+32)	1 транзакция (coalesced)

# Архитектура VRAM (**coalesced** memory access pattern)

128 bytes = Cache line = 32 x float



**float value = data[index];**      index      | cache lines count



[1024+0; 1024+32)

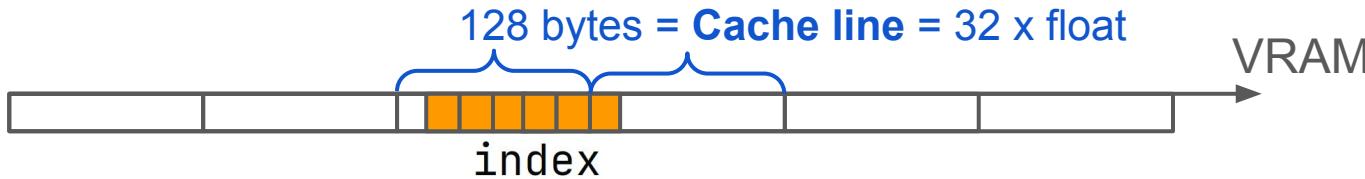
1 транзакция  
(**coalesced**)

[1024+1; 1024+33)

???

Сколько cache line-ов чтобы покрыть заказ?  
Сколько потребуется **транзакций** из VRAM?

# Архитектура VRAM (coalesced memory access pattern)



`float value = data[index];`



index	cache lines count
[1024+0; 1024+32)	1 транзакция (coalesced)
[1024+1; 1024+33)	2 транзакции (coalesced)

Насколько просела достигнутая полезная пропускная способность VRAM?

# Архитектура VRAM (**coalesced** memory access pattern)

128 bytes = Cache line = 32 x float



float value = data[index];      index	cache lines count
[1024+0; 1024+32)	1 транзакция <b>(coalesced)</b>
[1024+1; 1024+33)	2 транзакции <b>(coalesced)</b>
{1024 + i*32}	???



Сколько cache line-ов чтобы покрыть заказ?  
Сколько потребуется транзакций из VRAM?

# Архитектура VRAM (coalesced memory access pattern)

128 bytes = Cache line = 32 x float

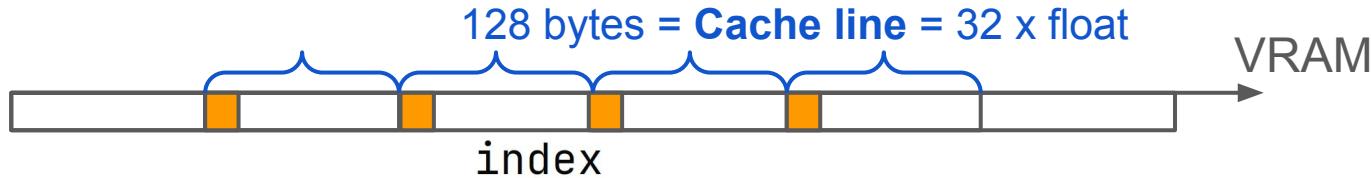


float value = data[index]; index	cache lines count
[1024+0; 1024+32)	1 транзакция <b>(coalesced)</b>
[1024+1; 1024+33)	2 транзакции <b>(coalesced)</b>
{1024 + i*32}	???



Сколько cache line-ов чтобы покрыть заказ?  
Сколько потребуется транзакций из VRAM?

# Архитектура VRAM (coalesced memory access pattern)

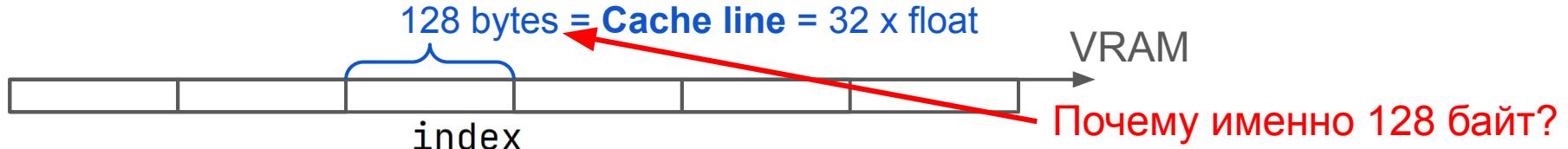


float value = data[index]; index	cache lines count
[1024+0; 1024+32)	1 транзакция <b>(coalesced)</b>
[1024+1; 1024+33)	2 транзакции <b>(coalesced)</b>
{1024 + i*32}	32 транзакции <b>(uncoalesced)</b>



Насколько просела достигнутая  
полезная пропускная способность VRAM?<sup>72</sup>

# Архитектура VRAM (coalesced memory access pattern)

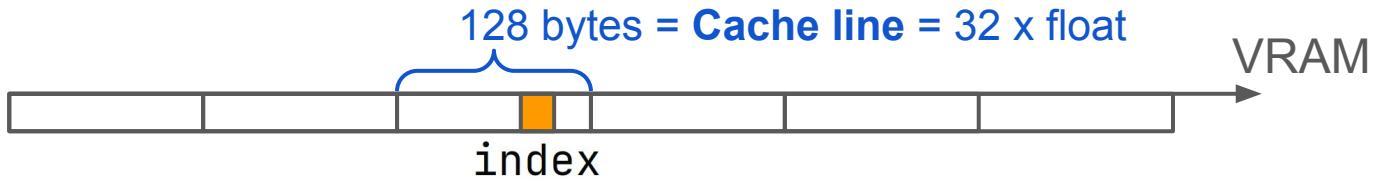


`float value = data[index];`      index      cache lines count

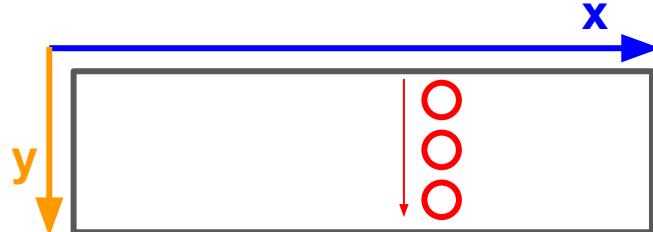


[1024+0; 1024+32)	1 транзакция (coalesced)
[1024+1; 1024+33)	2 транзакции (coalesced)
{1024 + i*32}	32 транзакции (uncoalesced)

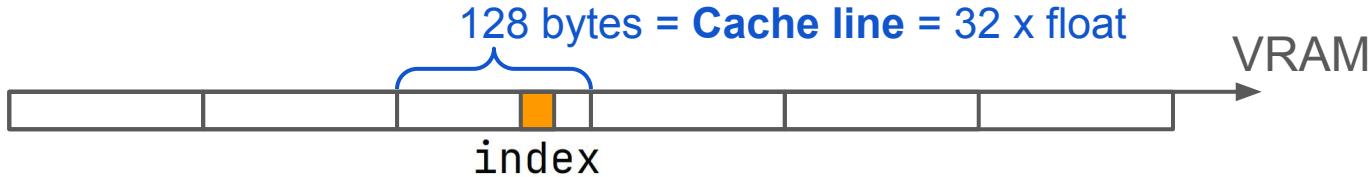
# Архитектура VRAM (coalesced memory access pattern)



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

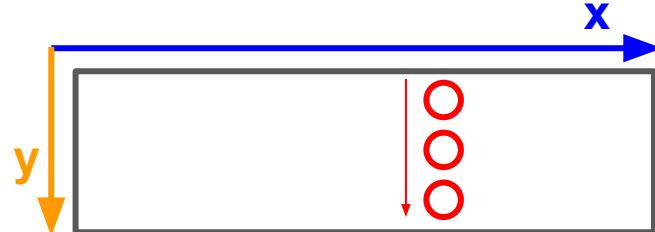


# Архитектура VRAM (coalesced memory access pattern)

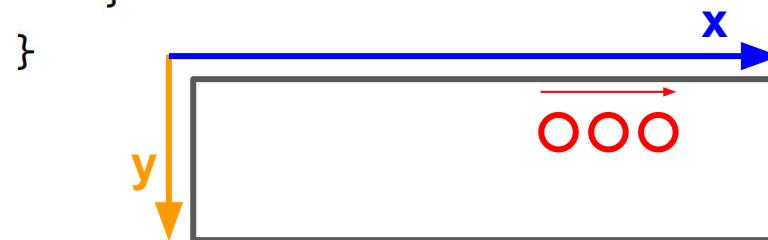


Но это однопоточный код, как в него добавить массовый параллелизм?

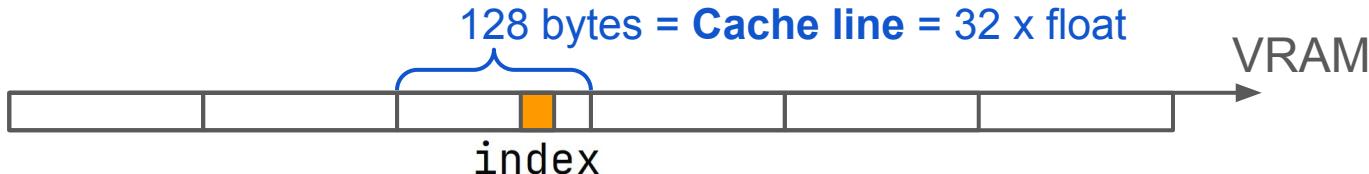
```
sum = 0;  
for (int x = 0; x < width; ++x) {  
    for (int y = 0; y < height; ++y) {  
        sum += image[y * width + x];  
    }  
}
```



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



# Архитектура VRAM (coalesced memory access pattern)

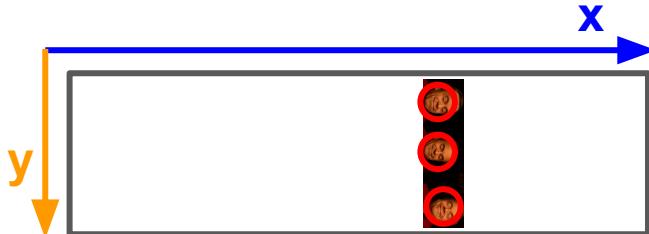


Но это однопоточный код, как в него добавить массовый параллелизм?

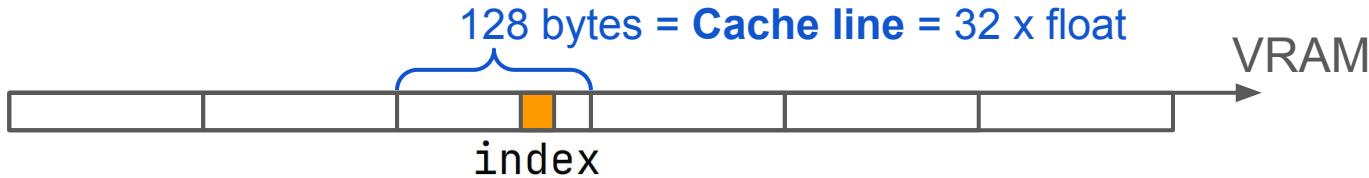
```
sum = 0;  
for (int x = 0; x < width; ++x) {  
    y =   
    sum += image[y * width + x];  
}  
}
```

Номер потока в *warp*

```
}
```

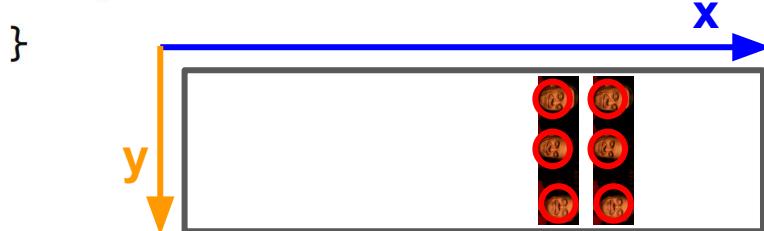


# Архитектура VRAM (coalesced memory access pattern)

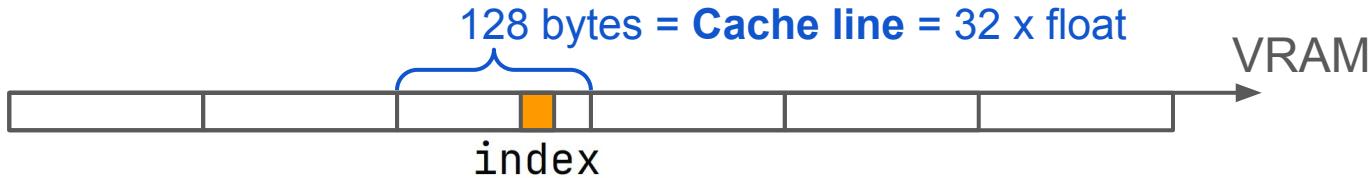


Но это однопоточный код, как в него добавить массовый параллелизм?

```
sum = 0;  
for (int x = 0; x < width; ++x) {  
    y = ; // Номер потока в warp  
    sum += image[y * width + x];  
}  
}
```



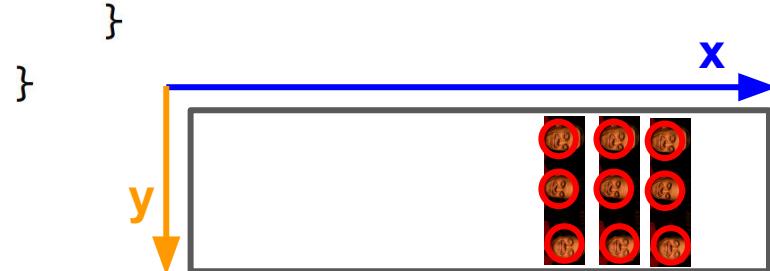
# Архитектура VRAM (coalesced memory access pattern)



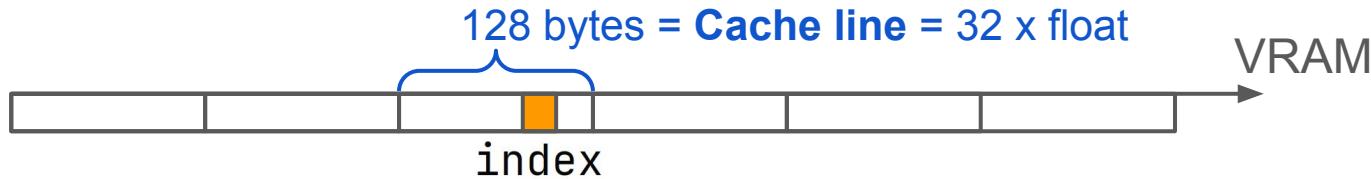
Но это однопоточный код, как в него добавить массовый параллелизм?

```
sum = 0;  
for (int x = 0; x < width; ++x) {  
    y = ; // Three faces of a man  
    sum += image[y * width + x];  
}  
}
```

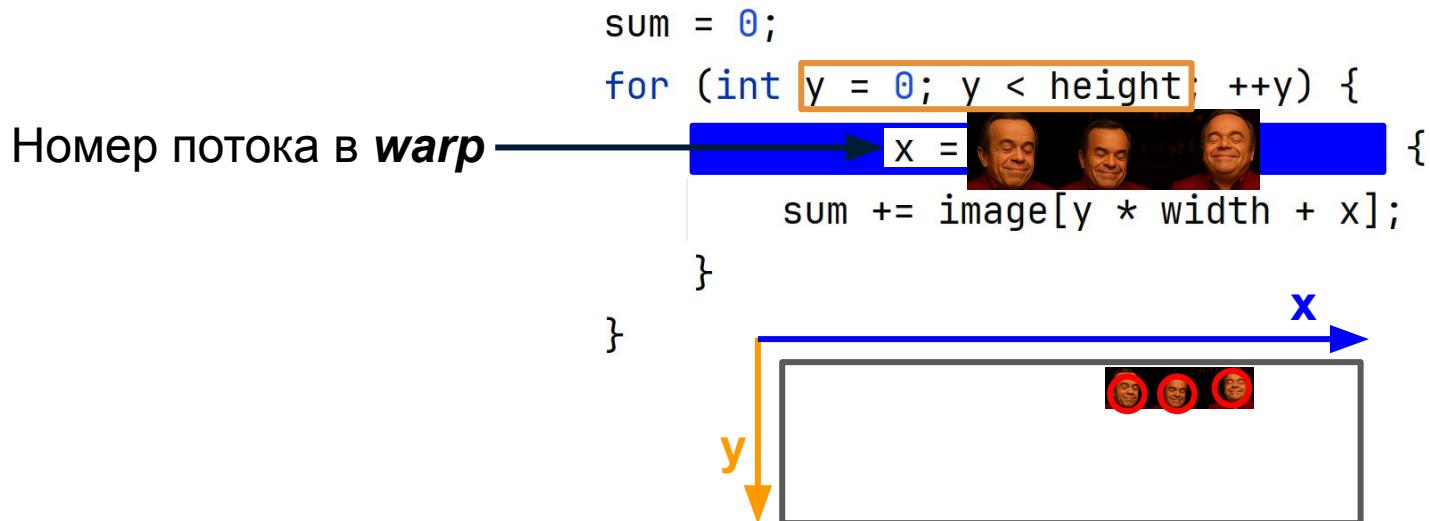
Номер потока в *warp*



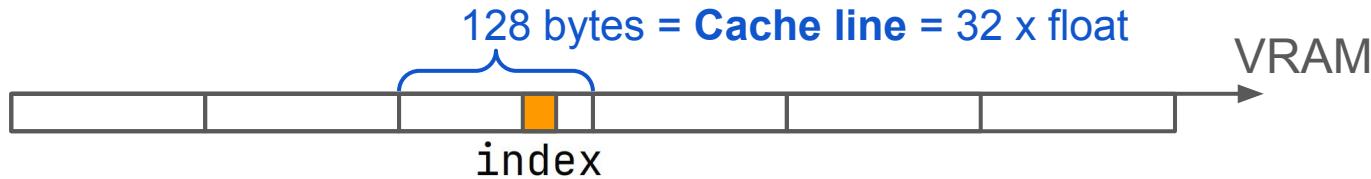
# Архитектура VRAM (coalesced memory access pattern)



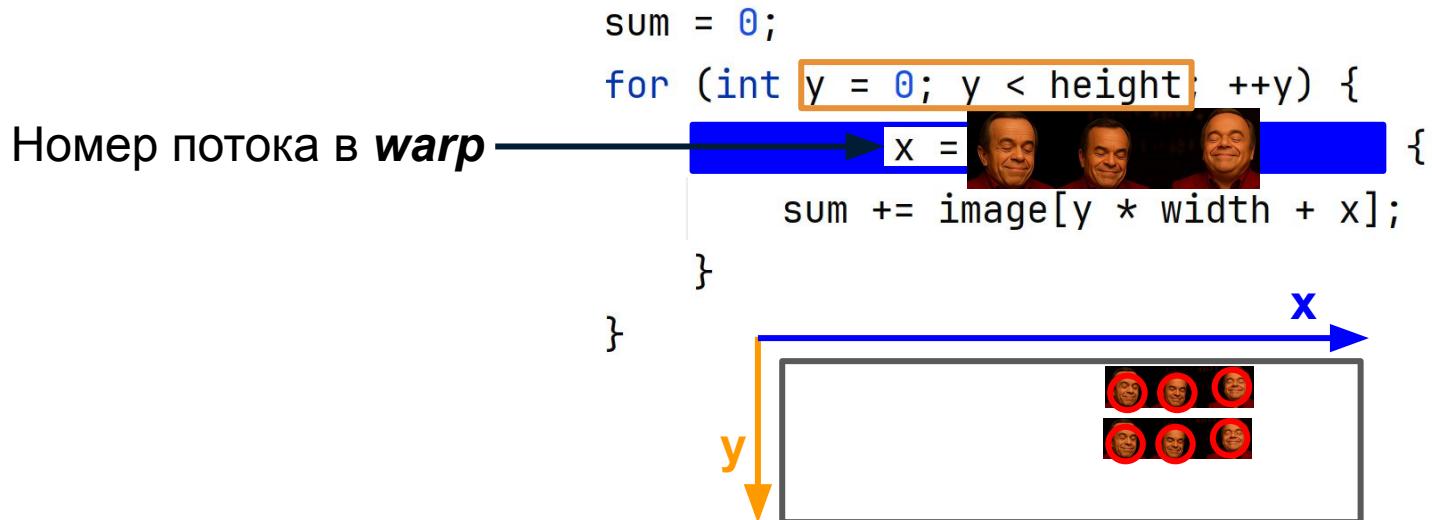
Но это однопоточный код, как в него добавить массовый параллелизм?



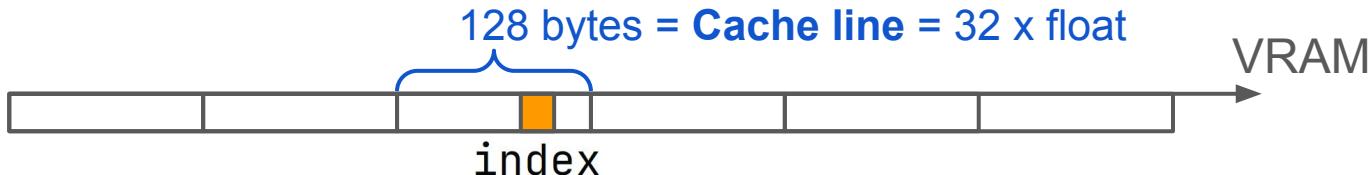
# Архитектура VRAM (coalesced memory access pattern)



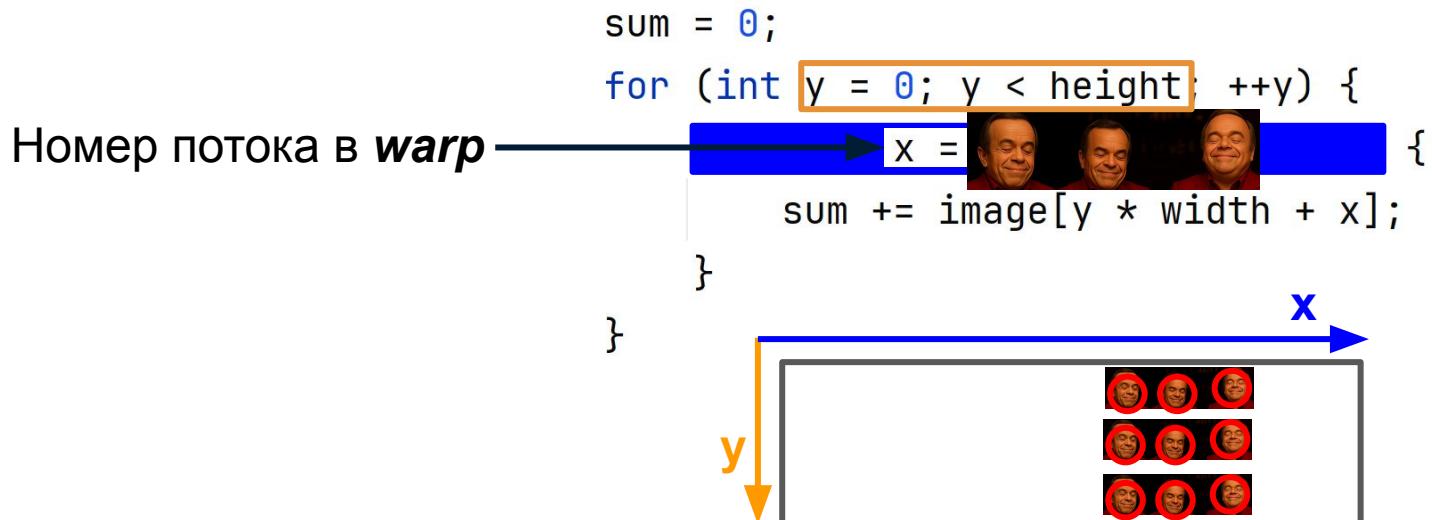
Но это однопоточный код, как в него добавить массовый параллелизм?



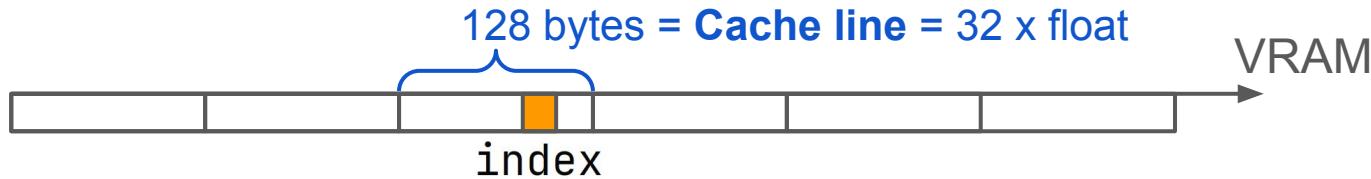
# Архитектура VRAM (coalesced memory access pattern)



Но это однопоточный код, как в него добавить массовый параллелизм?

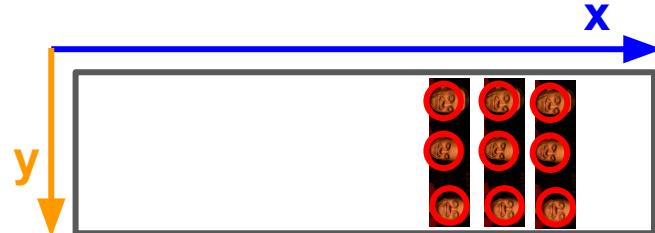


# Архитектура VRAM (coalesced memory access pattern)

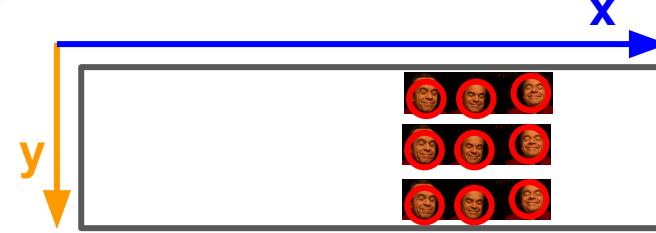


Какой код выбрали бы вы?

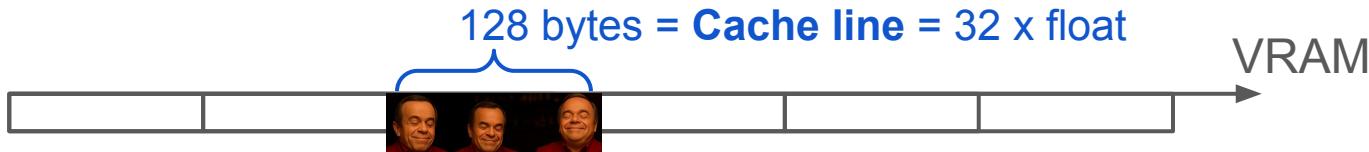
```
sum = 0;  
for (int x = 0; x < width; ++x) {  
    y =   
    sum += image[y * width + x];  
}  
}
```



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



# Архитектура VRAM (coalesced memory access pattern)



128 bytes = Cache line = 32 x float

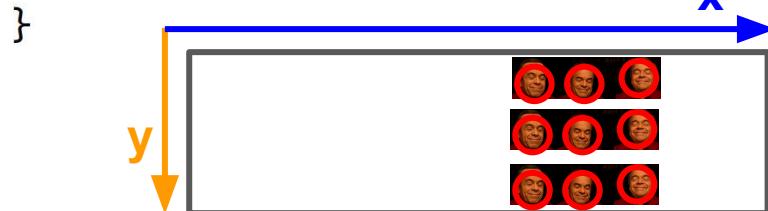
VRAM



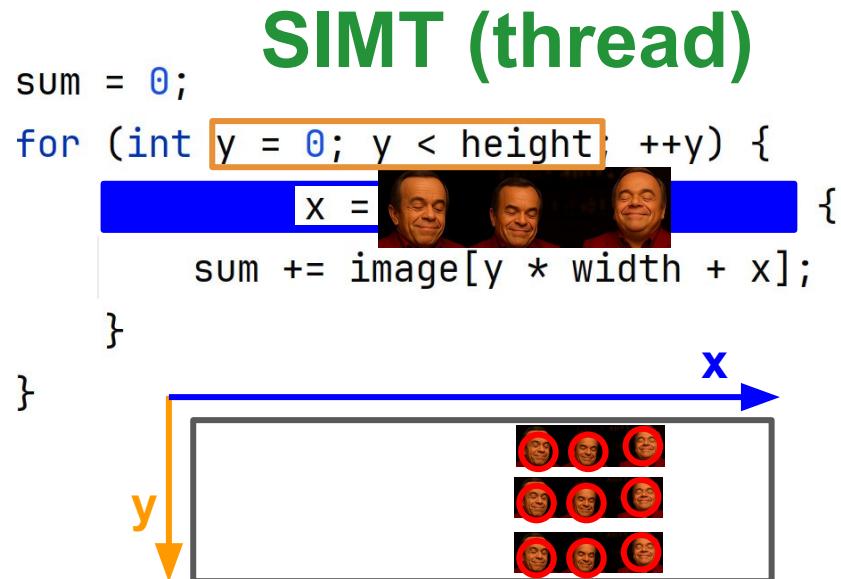
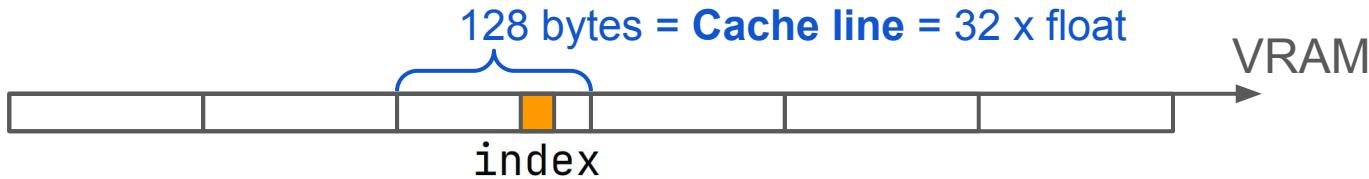
Какой код выбрали бы вы?

Они загружают cache line  
и распределят в рамках *warp*-а  
байты, произойдет *broadcast*

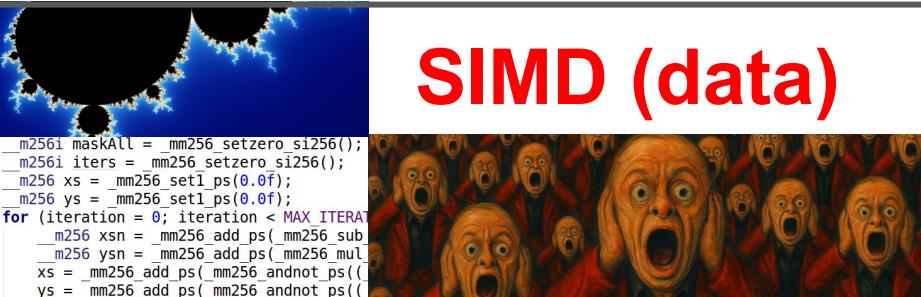
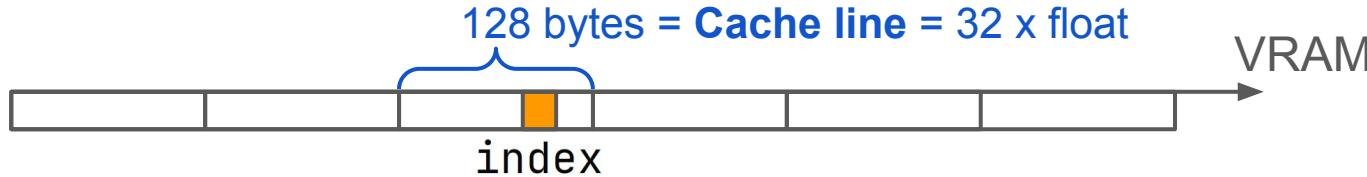
```
sum = 0;  
for (int y = 0; y < height; ++y) {  
    x = ;  
    sum += image[y * width + x];  
}
```



# Архитектура VRAM (coalesced memory access pattern)



# Архитектура VRAM (coalesced memory access pattern)



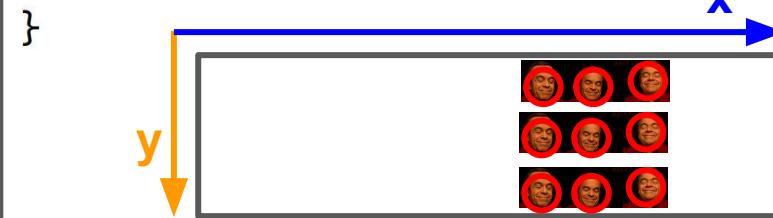
short b0	short b1	short b2	short b3
----------	----------	----------	----------



short c0	short c1	short c2	short c3
----------	----------	----------	----------

## SIMT (thread)

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

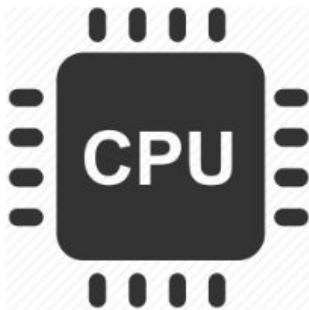


# Глава 3: Общая картина ЭВМ-архитектуры

CPU - RAM - PCI-E - VRAM - GPU

# Архитектура

$100 \cdot 10^9$  FLOPS



40 GB/s  
low latency

RAM - DDR5

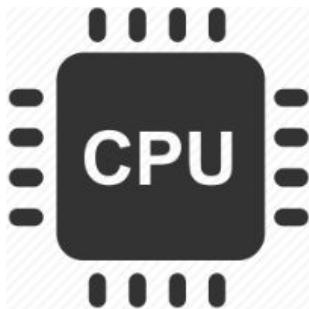


$100 \cdot 10^{12}$  FLOPS



# Архитектура

$100 \cdot 10^9$  FLOPS



40 GB/s  
low latency

RAM - DDR5



$100 \cdot 10^{12}$  FLOPS



1000 GB/s

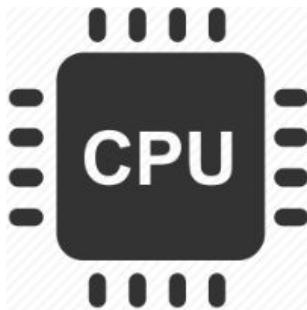
(x25 раз больше)



VRAM - GDDR6 или HBM (high bandwidth)

# Архитектура

$100 \cdot 10^9$  FLOPS

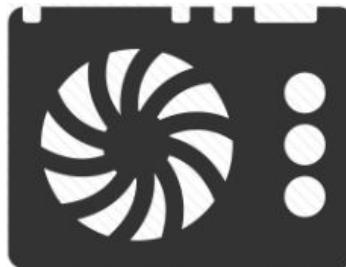


Где лучше исполнять OS?  
(например реагировать на клики пользователя)  
Какие есть метрики качества?

40 GB/s  
**low latency**



$100 \cdot 10^{12}$  FLOPS



1000 GB/s  
**(x25 раз больше)**

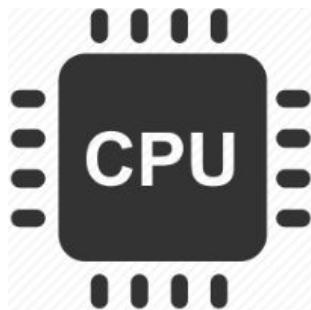


VRAM - GDDR6 или HBM (**high bandwidth**)

# Архитектура

Где быстрее сложить два массива чисел?

$100 \cdot 10^9$  FLOPS



40 GB/s

RAM - DDR5



$100 \cdot 10^{12}$  FLOPS



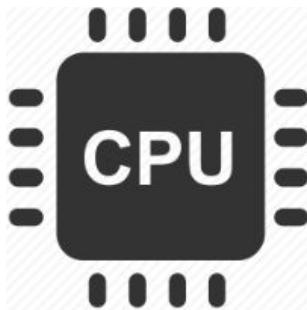
1000 GB/s



VRAM - GDDR6 или HBM (high bandwidth)

# Архитектура

$100 \cdot 10^9$  FLOPS

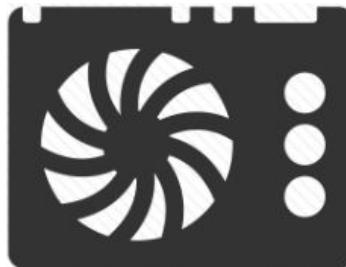


Где быстрее сложить два массива чисел?  
И во что мы упираемся - в память или в ALUs?  
(ALUs = Arithmetic Logic Units)

40 GB/s



$100 \cdot 10^{12}$  FLOPS



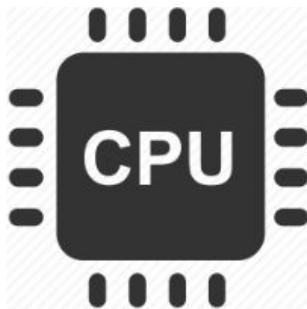
1000 GB/s



VRAM - GDDR6 или HBM (high bandwidth)

# Архитектура

$100 \cdot 10^9$  FLOPS



$100 \cdot 10^{12}$  FLOPS



Где быстрее сложить два массива чисел?  
И во что мы упираемся - в память или в ALUs?  
А что если данные находятся в **RAM**?

40 GB/s

RAM - DDR5

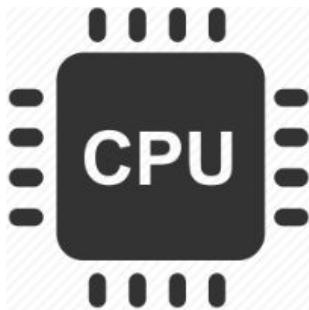
PCI-E 5.0 x16  
16 GB/s

1000 GB/s

VRAM - GDDR6

# Архитектура

$100 \cdot 10^9$  FLOPS



$100 \cdot 10^{12}$  FLOPS



Где быстрее сложить два массива чисел?  
И во что мы упираемся - в память или в ALUs?  
А что если данные находятся в **RAM**?

40 GB/s

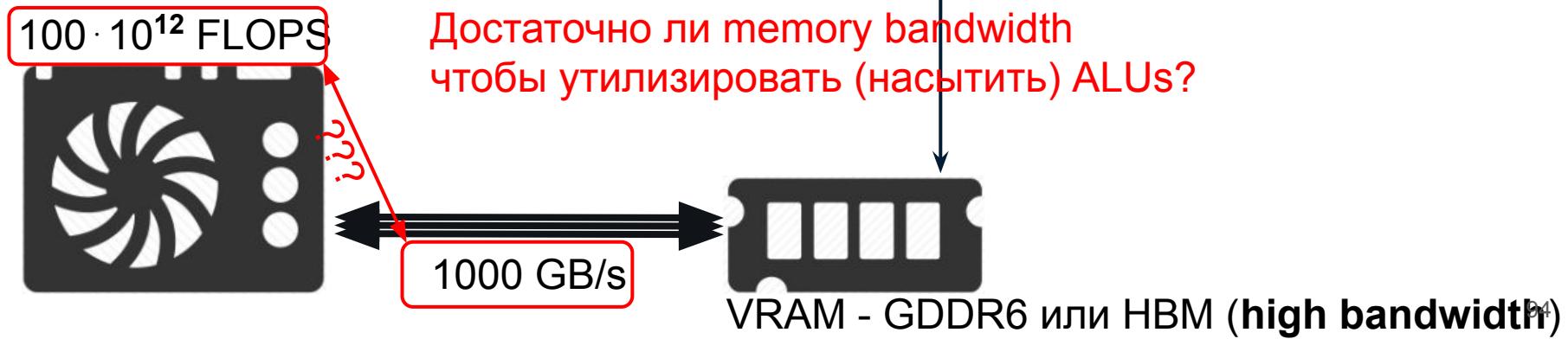
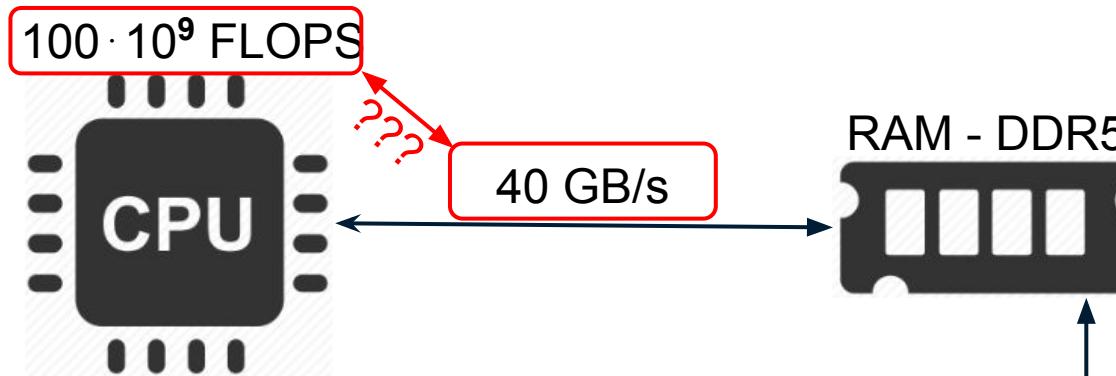
RAM - DDR5

PCI-E 5.0 x16  
16 GB/s

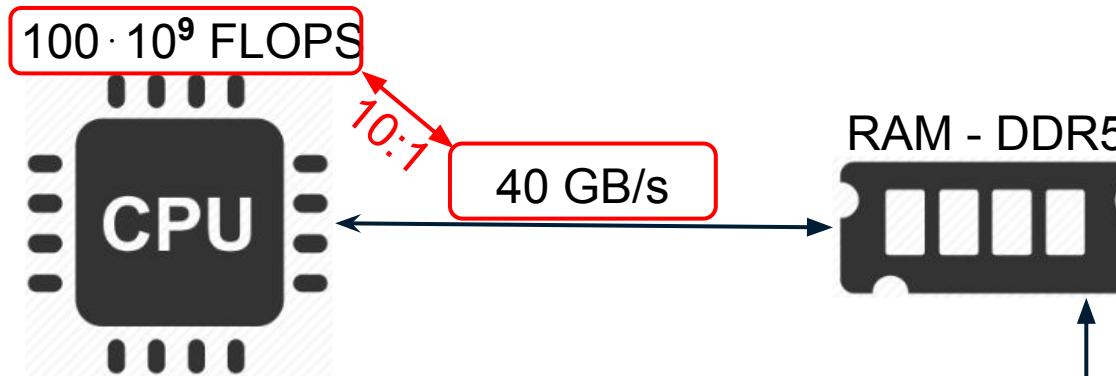
1000 GB/s

VRAM - GDDR6

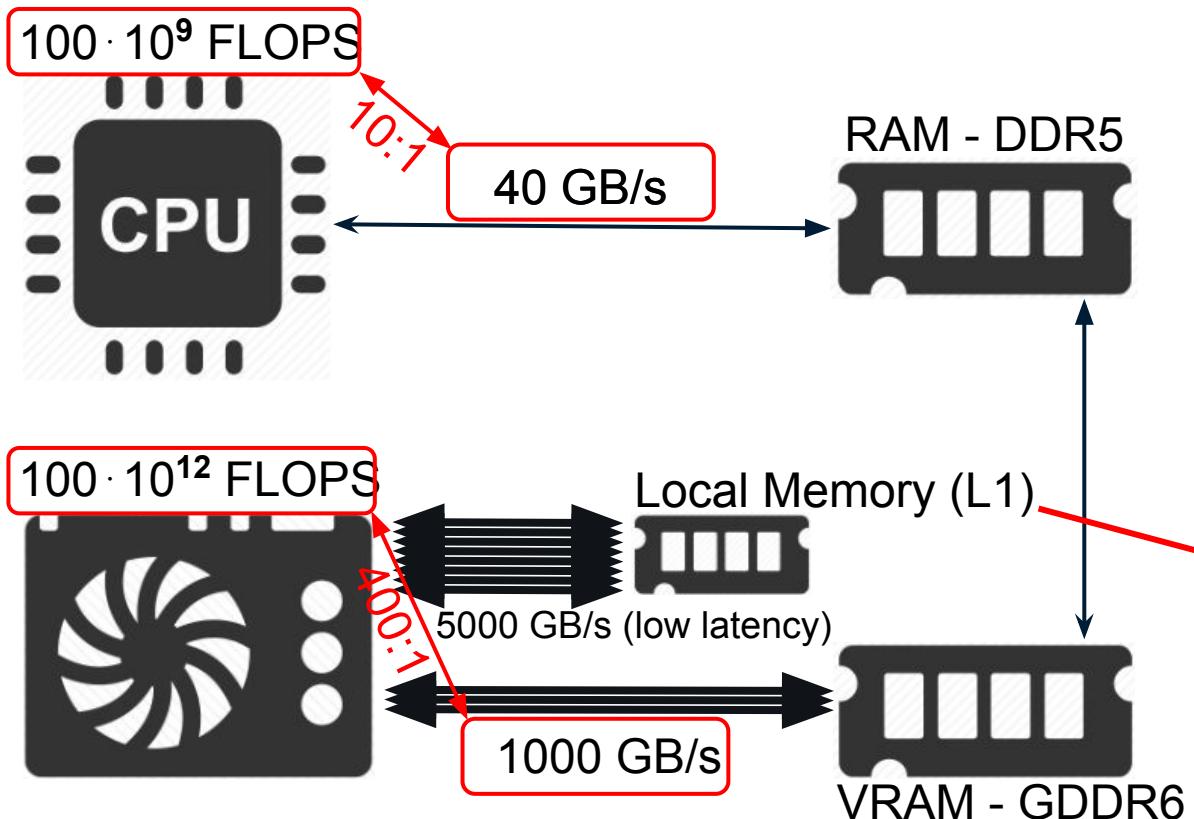
# Архитектура



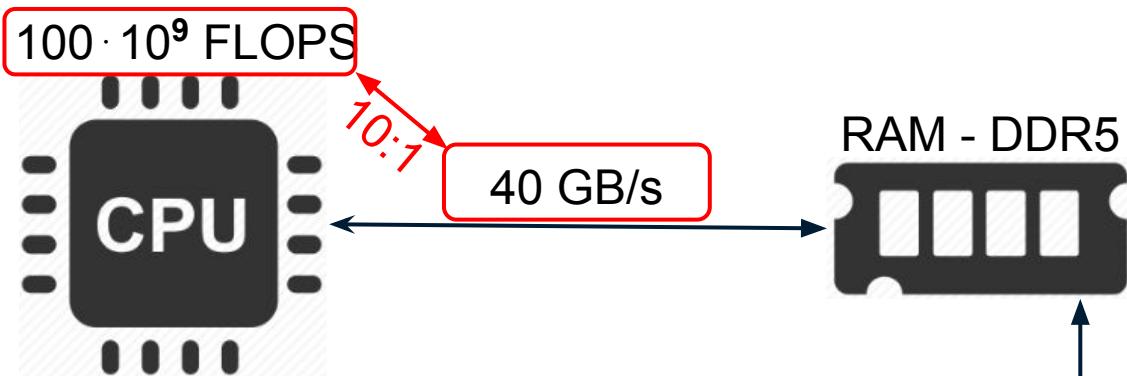
# Архитектура



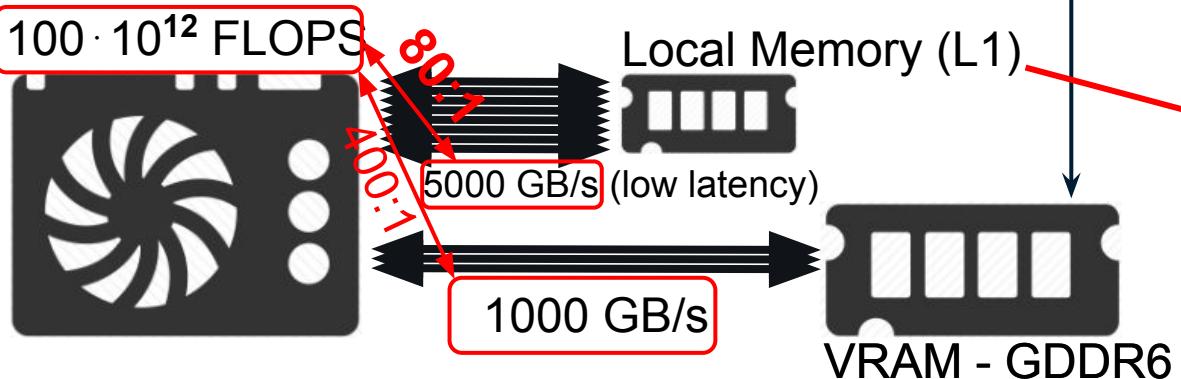
# Архитектура



# Архитектура

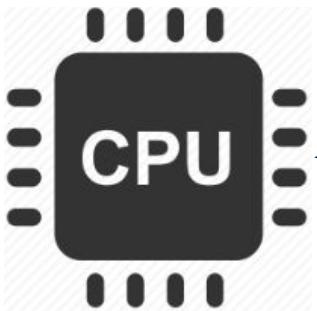


Зачем же так много ALUs?



# Архитектура

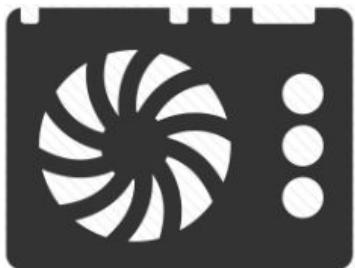
$100 \cdot 10^9$  FLOPS



40 GB/s

RAM - DDR5

$100 \cdot 10^{12}$  FLOPS



Можно ли ее использовать  
так же как VRAM?

Local Memory (L1)

5000 GB/s (low latency)

1000 GB/s

VRAM - GDDR6



L2 Cache



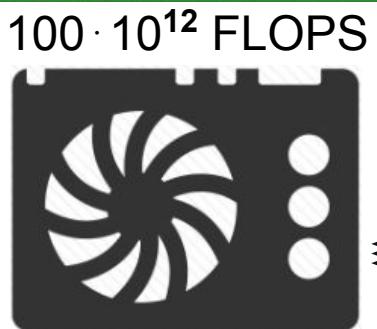
SM - Streaming Multiprocessor

SM

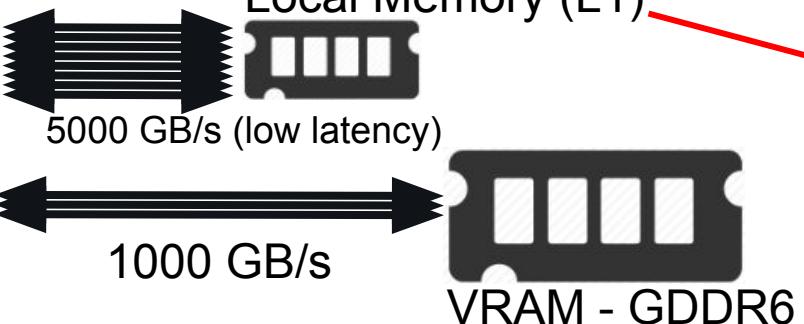


High-Speed Hub

VLink – Two x8 Links

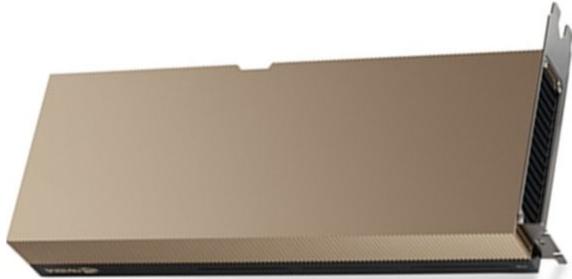


$100 \cdot 10^{12}$  FLOPS



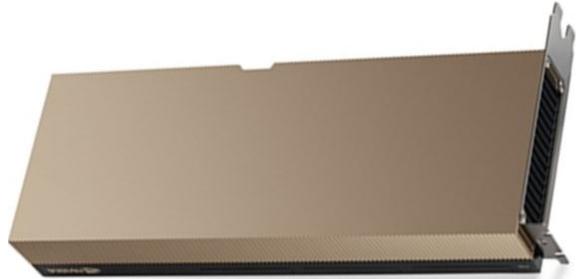
# Архитектура GPU

- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)



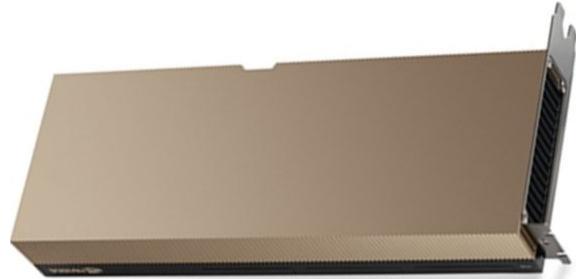
# Архитектура GPU

- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра



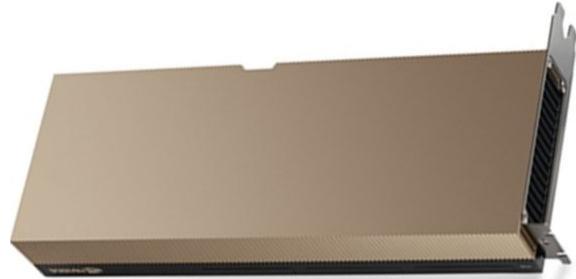
# Архитектура GPU

- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)

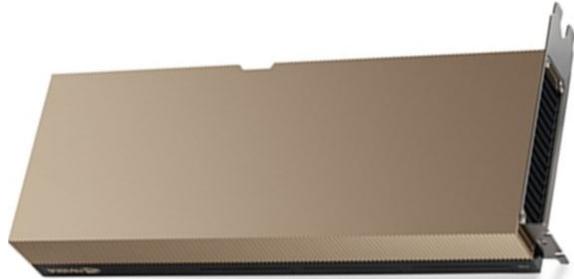


# Архитектура GPU

- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)

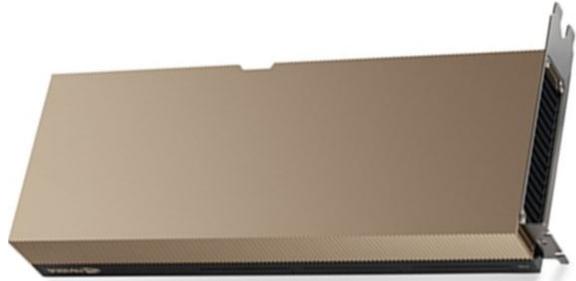


# Архитектура GPU



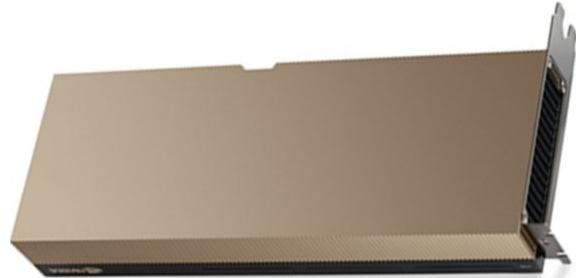
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:

# Архитектура GPU



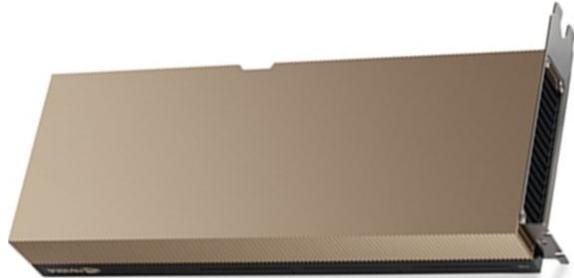
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)

# Архитектура GPU



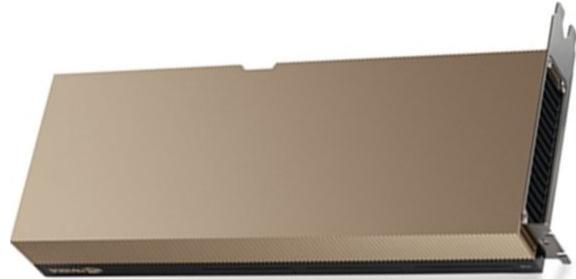
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверх-SMT/Hyper-Threading** (при высокой **occupancy**)

# Архитектура GPU



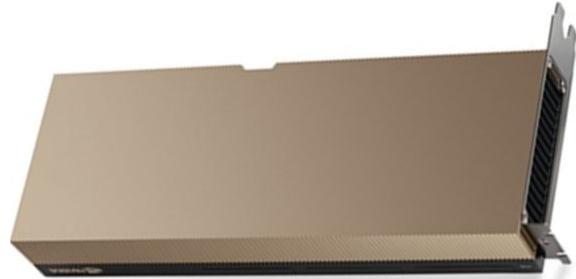
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверх-SMT/Hyper-Threading** (при высокой **occupancy**)
  - транзакции разбиты на 128-байтные **cache line**

# Архитектура GPU



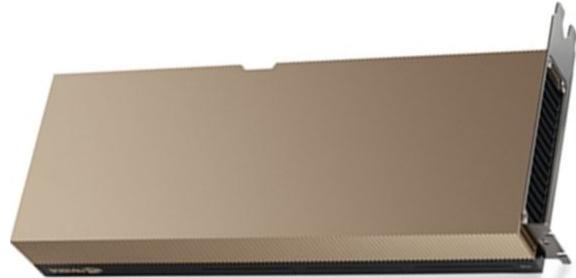
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверхx-SMT/Hyper-Threading** (при высокой **occupancy**)
  - транзакции разбиты на 128-байтные **cache line**
  - но при **un-coalesced** memory access pattern - малая проп. способность

# Архитектура GPU



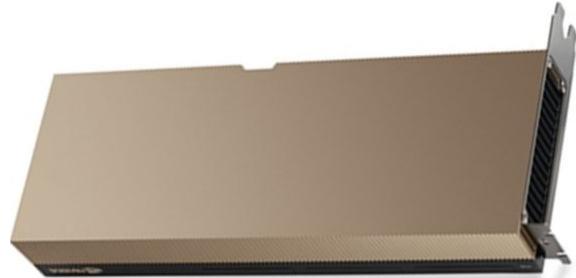
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверхx-SMT/Hyper-Threading** (при высокой **occupancy**)
  - транзакции разбиты на 128-байтные **cache line**
  - но при **un-coalesced** memory access pattern - малая проп. способность
- 3) Локальная память (**local/shared memory**):

# Архитектура GPU



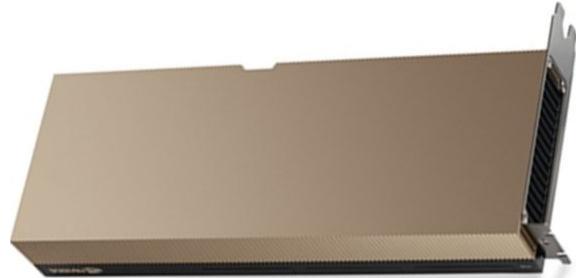
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверх-SMT/Hyper-Threading** (при высокой **occupancy**)
  - транзакции разбиты на 128-байтные **cache line**
  - но при **un-coalesced** memory access pattern - малая проп. способность
- 3) Локальная память (**local/shared memory**):
  - **титаническая** пропускная способность + низкая задержка

# Архитектура GPU



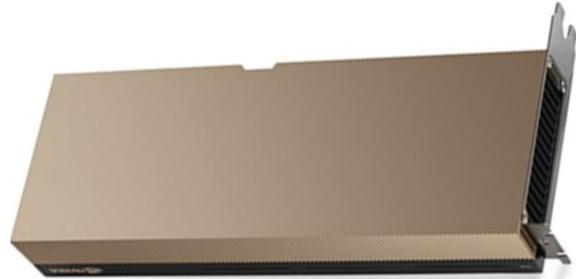
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверх-SMT/Hyper-Threading** (при высокой **occupancy**)
  - транзакции разбиты на 128-байтные **cache line**
  - но при **un-coalesced** memory access pattern - малая проп. способность
- 3) Локальная память (**local/shared memory**):
  - **титаническая** пропускная способность + низкая задержка
  - нет проблемы с coalesced-паттерном

# Архитектура GPU



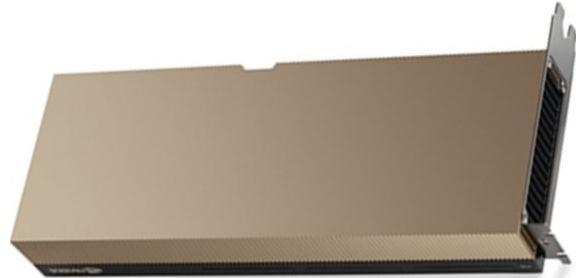
- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверх-SMT/Hyper-Threading** (при высокой **occupancy**)
  - транзакции разбиты на 128-байтные **cache line**
  - но при **un-coalesced** memory access pattern - малая проп. способность
- 3) Локальная память (**local/shared memory**):
  - **тиганическая** пропускная способность + низкая задержка
  - нет проблемы с coalesced-паттерном
  - **но?**

# Архитектура GPU

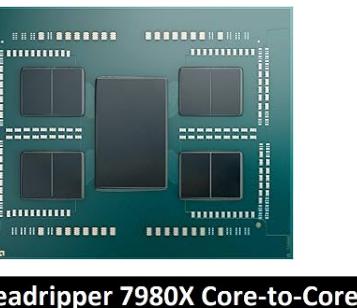


- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверх-SMT/Hyper-Threading** (при высокой **occupancy**)
  - транзакции разбиты на 128-байтные **cache line**
  - но при **un-coalesced** memory access pattern - малая проп. способность
- 3) Локальная память (**local/shared memory**):
  - **титаническая** пропускная способность + низкая задержка
  - нет проблемы с coalesced-паттерном
  - но **небольшая + локальная** для каждого **warp WorkGroup=Block**

# Архитектура GPU

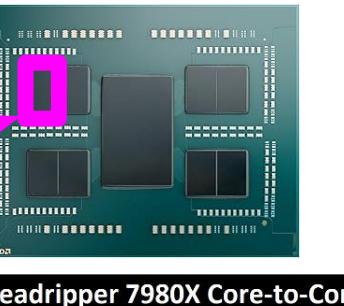


- 1) Десятки тысяч ядер (много **FLOPs**):
  - сгруппированы по 32 в **warp**-ы (по 64 в **wavefront**-ы у AMD)
  - но слабые ядра
  - но у warp-а общий **instruction pointer** (опасность **code divergence**)
  - активен **warp** который не ждет данных из **VRAM** (при высокой **occupancy**)
- 2) Огромная пропускная способность **VRAM**:
  - но большая задержка (**latency**)
  - скрывается за счет **сверх-SMT/Hyper-Threading** (при высокой **occupancy**)
  - транзакции разбиты на 128-байтные **cache line**
  - но при **un-coalesced** memory access pattern - малая проп. способность
- 3) Локальная память (**local/shared memory**):
  - **титаническая** пропускная способность + низкая задержка
  - нет проблемы с coalesced-паттерном      **Позже мы обсудим bank conflicts**
  - но **небольшая** + **локальная** для каждого **warp WorkGroup=Block**  
**И в этом секрет успеха! Рецепт к масштабируемости!**



88.8	91.1	91.1	89.7	89.6	91.8	91.9	90.7	90.6	90.7	90.5	93.1	92.9	98.4	93.5	92.0	92.2	94.2	94.4	98.4	93.4	94.0	94.8
88.7	91.1	91.1	89.8	89.8	91.8	91.9	90.8	90.5	90.7	90.9	92.7	93.1	99.5	93.5	92.0	92.0	94.0	94.4	99.5	93.4	94.8	94.8
89.3	91.5	91.5	90.2	90.3	92.4	92.5	91.1	91.0	91.2	91.1	93.6	93.5	93.9	93.9	97.5	92.5	94.8	94.7	93.9	94.0	95.4	95.3
89.3	91.6	91.5	90.2	90.3	92.4	92.4	91.0	91.1	91.1	93.4	93.5	94.0	93.9	92.4	92.5	94.8	94.9	93.9	93.9	95.3	95.4	

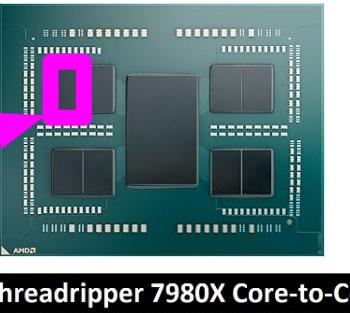
64 ядра = 8 x Чиплетов (по 8 ядер)



## AMD Ryzen Threadripper 7980X Core-to-Core Latency

	X	7.4	20.1	20.1	93.8	93.8	94.6	94.6
	7.4 x	20.1	20.1	93.9	93.8	94.5	94.5	94.6
	20.1	20.1 x	7.4	94.3	94.3	94.5	95.0	95.1
	20.1	20.1	7.4 x	94.4	94.4	94.4	95.2	95.1
	93.8	93.9	94.3	94.4 x	7.4	18.1	18.1	18.1
	93.8	93.8	94.5	94.4	7.4 x	18.0	18.0	18.1
	94.6	94.5	95.0	95.2	18.1	18.0 x	18.0	7.4
	94.6	94.6	95.1	95.1	18.1	18.1	18.1	7.4 x

Но почему задержка маленькая  
группами по 16 ядер?  
64 ядра = 8 x Чиплетов (по 8 ядер)

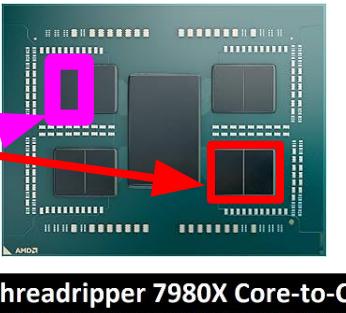


X	7.4	20.1	20.1	93.8	93.8	94.6	94.6
7.4 x							
20.1		x					
20.1		20.1	x				
93.8	93.9	94.3	94.4 x		7.4	18.1	18.1
93.8	93.8	94.5	94.4	7.4 x		18.0	18.1
94.6	94.5	95.0	95.2	18.1	18.0 x		7.4
94.6	94.6	95.1	95.1	18.1	18.1	18.1	7.4 x

AMD Ryzen Threadripper 7980X Core-to-Core Latency

	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7	CPU8	CPU9	CPU10	CPU11	CPU12	CPU13	CPU14	CPU15	CPU16	CPU17	CPU18	CPU19	CPU20	CPU21	CPU22	CPU23	CPU24	CPU25	CPU26	CPU27	CPU28	CPU29	CPU30	CPU31	CPU32	CPU33	CPU34	CPU35	CPU36	CPU37	CPU38	CPU39	CPU40	CPU41	CPU42	CPU43	CPU44	CPU45	CPU46	CPU47	CPU48	CPU49	CPU50	CPU51	CPU52	CPU53	CPU54	CPU55	CPU56	CPU57	CPU58	CPU59	CPU60	CPU61	CPU62	CPU63	CPU64	CPU65	CPU66	CPU67	CPU68	CPU69	CPU70	CPU71	CPU72	CPU73	CPU74	CPU75	CPU76	CPU77	CPU78	CPU79	CPU80	CPU81	CPU82	CPU83	CPU84	CPU85	CPU86	CPU87	CPU88	CPU89	CPU90	CPU91	CPU92	CPU93	CPU94	CPU95	CPU96	CPU97	CPU98	CPU99	CPU100	CPU101	CPU102	CPU103	CPU104	CPU105	CPU106	CPU107	CPU108	CPU109	CPU110	CPU111	CPU112	CPU113	CPU114	CPU115	CPU116	CPU117	CPU118	CPU119	CPU120	CPU121	CPU122	CPU123	CPU124	CPU125	CPU126	CPU127	CPU128	CPU129	CPU130	CPU131	CPU132	CPU133	CPU134	CPU135	CPU136	CPU137	CPU138	CPU139	CPU140	CPU141	CPU142	CPU143	CPU144	CPU145	CPU146	CPU147	CPU148	CPU149	CPU150	CPU151	CPU152	CPU153	CPU154	CPU155	CPU156	CPU157	CPU158	CPU159	CPU160	CPU161	CPU162	CPU163	CPU164	CPU165	CPU166	CPU167	CPU168	CPU169	CPU170	CPU171	CPU172	CPU173	CPU174	CPU175	CPU176	CPU177	CPU178	CPU179	CPU180	CPU181	CPU182	CPU183	CPU184	CPU185	CPU186	CPU187	CPU188	CPU189	CPU190	CPU191	CPU192	CPU193	CPU194	CPU195	CPU196	CPU197	CPU198	CPU199	CPU200	CPU201	CPU202	CPU203	CPU204	CPU205	CPU206	CPU207	CPU208	CPU209	CPU210	CPU211	CPU212	CPU213	CPU214	CPU215	CPU216	CPU217	CPU218	CPU219	CPU220	CPU221	CPU222	CPU223	CPU224	CPU225	CPU226	CPU227	CPU228	CPU229	CPU230	CPU231	CPU232	CPU233	CPU234	CPU235	CPU236	CPU237	CPU238	CPU239	CPU240	CPU241	CPU242	CPU243	CPU244	CPU245	CPU246	CPU247	CPU248	CPU249	CPU250	CPU251	CPU252	CPU253	CPU254	CPU255	CPU256	CPU257	CPU258	CPU259	CPU260	CPU261	CPU262	CPU263	CPU264	CPU265	CPU266	CPU267	CPU268	CPU269	CPU270	CPU271	CPU272	CPU273	CPU274	CPU275	CPU276	CPU277	CPU278	CPU279	CPU280	CPU281	CPU282	CPU283	CPU284	CPU285	CPU286	CPU287	CPU288	CPU289	CPU290	CPU291	CPU292	CPU293	CPU294	CPU295	CPU296	CPU297	CPU298	CPU299	CPU300	CPU301	CPU302	CPU303	CPU304	CPU305	CPU306	CPU307	CPU308	CPU309	CPU310	CPU311	CPU312	CPU313	CPU314	CPU315	CPU316	CPU317	CPU318	CPU319	CPU320	CPU321	CPU322	CPU323	CPU324	CPU325	CPU326	CPU327	CPU328	CPU329	CPU330	CPU331	CPU332	CPU333	CPU334	CPU335	CPU336	CPU337	CPU338	CPU339	CPU340	CPU341	CPU342	CPU343	CPU344	CPU345	CPU346	CPU347	CPU348	CPU349	CPU350	CPU351	CPU352	CPU353	CPU354	CPU355	CPU356	CPU357	CPU358	CPU359	CPU360	CPU361	CPU362	CPU363	CPU364	CPU365	CPU366	CPU367	CPU368	CPU369	CPU370	CPU371	CPU372	CPU373	CPU374	CPU375	CPU376	CPU377	CPU378	CPU379	CPU380	CPU381	CPU382	CPU383	CPU384	CPU385	CPU386	CPU387	CPU388	CPU389	CPU390	CPU391	CPU392	CPU393	CPU394	CPU395	CPU396	CPU397	CPU398	CPU399	CPU400	CPU401	CPU402	CPU403	CPU404	CPU405	CPU406	CPU407	CPU408	CPU409	CPU410	CPU411	CPU412	CPU413	CPU414	CPU415	CPU416	CPU417	CPU418	CPU419	CPU420	CPU421	CPU422	CPU423	CPU424	CPU425	CPU426	CPU427	CPU428	CPU429	CPU430	CPU431	CPU432	CPU433	CPU434	CPU435	CPU436	CPU437	CPU438	CPU439	CPU440	CPU441	CPU442	CPU443	CPU444	CPU445	CPU446	CPU447	CPU448	CPU449	CPU450	CPU451	CPU452	CPU453	CPU454	CPU455	CPU456	CPU457	CPU458	CPU459	CPU460	CPU461	CPU462	CPU463	CPU464	CPU465	CPU466	CPU467	CPU468	CPU469	CPU470	CPU471	CPU472	CPU473	CPU474	CPU475	CPU476	CPU477	CPU478	CPU479	CPU480	CPU481	CPU482	CPU483	CPU484	CPU485	CPU486	CPU487	CPU488	CPU489	CPU490	CPU491	CPU492	CPU493	CPU494	CPU495	CPU496	CPU497	CPU498	CPU499	CPU500	CPU501	CPU502	CPU503	CPU504	CPU505	CPU506	CPU507	CPU508	CPU509	CPU510	CPU511	CPU512	CPU513	CPU514	CPU515	CPU516	CPU517	CPU518	CPU519	CPU520	CPU521	CPU522	CPU523	CPU524	CPU525	CPU526	CPU527	CPU528	CPU529	CPU530	CPU531	CPU532	CPU533	CPU534	CPU535	CPU536	CPU537	CPU538	CPU539	CPU540	CPU541	CPU542	CPU543	CPU544	CPU545	CPU546	CPU547	CPU548	CPU549	CPU550	CPU551	CPU552	CPU553	CPU554	CPU555	CPU556	CPU557	CPU558	CPU559	CPU560	CPU561	CPU562	CPU563	CPU564	CPU565	CPU566	CPU567	CPU568	CPU569	CPU570	CPU571	CPU572	CPU573	CPU574	CPU575	CPU576	CPU577	CPU578	CPU579	CPU580	CPU581	CPU582	CPU583	CPU584	CPU585	CPU586	CPU587	CPU588	CPU589	CPU590	CPU591	CPU592	CPU593	CPU594	CPU595	CPU596	CPU597	CPU598	CPU599	CPU600	CPU601	CPU602	CPU603	CPU604	CPU605	CPU606	CPU607	CPU608	CPU609	CPU610	CPU611	CPU612	CPU613	CPU614	CPU615	CPU616	CPU617	CPU618	CPU619	CPU620	CPU621	CPU622	CPU623	CPU624	CPU625	CPU626	CPU627	CPU628	CPU629	CPU630	CPU631	CPU632	CPU633	CPU634	CPU635	CPU636	CPU637	CPU638	CPU639	CPU640	CPU641	CPU642	CPU643	CPU644	CPU645	CPU646	CPU647	CPU648	CPU649	CPU650	CPU651	CPU652	CPU653	CPU654	CPU655	CPU656	CPU657	CPU658	CPU659	CPU660	CPU661	CPU662	CPU663	CPU664	CPU665	CPU666	CPU667	CPU668	CPU669	CPU670	CPU671	CPU672	CPU673	CPU674	CPU675	CPU676	CPU677	CPU678	CPU679	CPU680	CPU681	CPU682	CPU683	CPU684	CPU685	CPU686	CPU687	CPU688	CPU689	CPU690	CPU691	CPU692	CPU693	CPU694	CPU695	CPU696	CPU697	CPU698	CPU699	CPU700	CPU701	CPU702	CPU703	CPU704	CPU705	CPU706	CPU707	CPU708	CPU709	CPU710	CPU711	CPU712	CPU713	CPU714	CPU715	CPU716	CPU717	CPU718	CPU719	CPU720	CPU721	CPU722	CPU723	CPU724	CPU725	CPU726	CPU727	CPU728	CPU729	CPU730	CPU731	CPU732	CPU733	CPU734	CPU735	CPU736	CPU737	CPU738	CPU739	CPU740	CPU741	CPU742	CPU743	CPU744	CPU745	CPU746	CPU747	CPU748	CPU749	CPU750	CPU751	CPU752	CPU753	CPU754	CPU755	CPU756	CPU757	CPU758	CPU759	CPU760	CPU761	CPU762	CPU763	CPU764	CPU765	CPU766	CPU767	CPU768	CPU769	CPU770	CPU771	CPU772	CPU773	CPU774	CPU775	CPU776	CPU777	CPU778	CPU779	CPU780	CPU781	CPU782	CPU783	CPU784	CPU785	CPU786	CPU787	CPU788	CPU789	CPU790	CPU791	CPU792	CPU793	CPU794	CPU795	CPU796	CPU797	CPU798	CPU799	CPU800	CPU801	CPU802	CPU803	CPU804	CPU805	CPU806	CPU807	CPU808	CPU809	CPU810	CPU811	CPU812	CPU813	CPU814	CPU815	CPU816	CPU817	CPU818	CPU819	CPU820	CPU821	CPU822	CPU823	CPU824	CPU825	CPU826	CPU827	CPU828	CPU829	CPU830	CPU831	CPU832	CPU833	CPU834	CPU835	CPU836	CPU837	CPU838	CPU839	CPU840	CPU841	CPU842	CPU843	CPU844	CPU845	CPU846	CPU847	CPU848	CPU849	CPU850	CPU851	CPU852	CPU853	CPU854	CPU855	CPU856	CPU857	CPU858	CPU859	CPU860	CPU861	CPU862	CPU863	CPU864	CPU865	CPU866	CPU867	CPU868	CPU869	CPU870	CPU871	CPU872	CPU873	CPU874	CPU875	CPU876	CPU877	CPU878	CPU879	CPU880	CPU881	CPU882	CPU883	CPU884	CPU885	CPU886	CPU887	CPU888	CPU889	CPU890	CPU891	CPU892	CPU893	CPU894	CPU895	CPU896	CPU897	CPU898	CPU899	CPU900	CPU901	CPU902	CPU903	CPU904	CPU905	CPU906	CPU907	CPU908	CPU909	CPU910	CPU911	CPU912	CPU913	CPU914	CPU915	CPU916	CPU917	CPU918	CPU919	CPU920	CPU921	CPU922	CPU923	CPU924	CPU925	CPU926	CPU927	CPU928	CPU929	CPU930	CPU931	CPU932	CPU933	CPU934	CPU935	CPU936	CPU937	CPU938	CPU939	CPU940	CPU941	CPU942	CPU943	CPU944	CPU945	CPU946	CPU947	CPU948	CPU949	CPU950	CPU951	CPU952	CPU953	CPU954	CPU955	CPU956	CPU957	CPU958	CPU959	CPU960	CPU961	CPU962	CPU963	CPU964	CPU965	CPU966	CPU967	CPU968	CPU969	CPU970	CPU971	CPU972	CPU973	CPU974	CPU975	CPU976	CPU977	CPU978	CPU979	CPU980	CPU981	CPU982	CPU983	CPU984	CPU985	CPU986	CPU987	CPU988	CPU989	CPU990	CPU991	CPU992	CPU993	CPU994	CPU995	CPU996	CPU997	CPU998	CPU999	CPU1000	CPU1001	CPU1002	CPU1003	CPU1004	CPU1005	CPU1006	CPU1007	CPU1008	CPU1009	CPU1010	CPU1011	CPU1012	CPU1013	CPU1014	CPU1015	CPU1016	CPU1017	CPU1018	CPU1019	CPU1020	CPU1021	CPU1022	CPU1023	CPU1024	CPU1025	CPU1026	CPU1027	CPU1028	CPU1029	CPU1030	CPU1031	CPU1032	CPU1033	CPU1034	CPU1035	CPU1036	CPU1037	CPU1038	CPU1039	CPU1040	CPU1041	CPU1042	CPU1043	CPU1044	CPU1045	CPU1046	CPU1047	CPU1048	CPU1049	CPU1050	CPU1051	CPU1052	CPU1053	CPU1054	CPU1055	CPU1056	CPU1057	CPU1058	CPU1059	CPU1060	CPU1061	CPU1062	CPU1063	CPU1064	CPU1065	CPU1066	CPU1067	CPU1068	CPU1069	CPU1070	CPU1071	CPU1072	CPU1073	CPU1074	CPU1075	CPU1076	CPU1077	CPU1078	CPU1079	CPU1080	CPU1081	CPU1082	CPU1083	CPU1084	CPU1085	CPU1086	CPU1087	CPU1088	CPU1089	CPU1090	CPU1091	CPU1092	CPU1093	CPU1094	CPU1095	CPU1096	CPU1097	CPU1098	CPU1099	CPU1100	CPU1101	CPU1102	CPU1103	CPU1104	CPU1105	CPU1106	CPU1107	CPU1108	CPU1109	CPU1110	CPU1111	CPU1112	CPU1113	CPU1114	CPU1115	CPU1116	CPU1117	CPU1118	CPU1119	CPU1120	CPU1121	CPU1122	CPU1123	CPU1124	CPU1125	CPU1126	CPU1127	CPU1128	CPU1129	CPU1130	CPU1131	CPU1132	CPU1133	CPU1134	CPU1135	CPU1136	CPU1137	CPU1138	CPU1139	CPU1140	CPU1141	CPU1142	CPU1143	CPU1144	CPU1145	CPU1146	CPU1147	CPU1148	CPU1149	CPU1150	CPU1151	CPU1152	CPU1153	CPU1154	CPU1155	CPU1156	CPU1157	CPU1158	CPU1159	CPU1160	CPU1161	CPU1162	CPU1163	CPU1164	CPU1165	CPU1166	CPU1167	CPU1168	CPU1169	CPU1170	CPU1171	CPU1172	CPU1173	CPU1174	CPU1175	CPU1176	CPU1177	CPU1178	CPU1179	CPU1180	CPU1181	CPU1182	CPU1183	CPU1184	CPU1185	CPU1186	CPU1187	CPU1188	CPU1189	CPU1190	CPU1191	CPU1192	CPU1193	CPU1194	CPU1195	CPU1196	CPU1197	CPU1198	CPU1199	CPU1200	CPU1201	CPU1202	CPU1203	CPU1204	CPU1205	CPU1206	CPU1207	CPU1208	CPU1209	CPU1210	CPU1211	CPU1212	CPU1213	CPU1214	CPU1215	CPU1216	CPU1217	CPU1218	CPU1219	CPU1220	CPU1221	CPU1222	CPU1223	CPU1224	CPU1225	CPU1226	CPU1227	CPU1228	CPU1229	CPU1230	CPU1231	CPU1232	CPU1233	CPU1234

Но почему задержка маленькая  
группами по 16 ядер?  
64 ядра = 8 x Чиплетов (по 8 ядер)



X	7.4	20.1	20.1	93.8	93.8	94.6	94.6
7.4 x							
20.1		x					
20.1		20.1	x				
93.8	93.9	94.3	94.4	x	7.4	18.1	18.1
93.8	93.8	94.5	94.4	7.4 x		18.0	18.1
94.6	94.5	95.0	95.2	18.1	18.0 x		7.4
94.6	94.6	95.1	95.1	18.1	18.1	7.4 x	

AMD Ryzen Threadripper 7980X Core-to-Core Latency

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6	CPU7	CPU8	CPU9	CPU10	CPU11	CPU12	CPU13	CPU14	CPU15	CPU16	CPU17	CPU18	CPU19	CPU20	CPU21	CPU22	CPU23	CPU24	CPU25	CPU26	CPU27	CPU28	CPU29	CPU30	CPU31	CPU32	CPU33	CPU34	CPU35	CPU36	CPU37	CPU38	CPU39	CPU40	CPU41	CPU42	CPU43	CPU44	CPU45	CPU46	CPU47	CPU48	CPU49	CPU50	CPU51	CPU52	CPU53	CPU54	CPU55	CPU56	CPU57	CPU58	CPU59	CPU60	CPU61	CPU62	CPU63	CPU64	CPU65	CPU66	CPU67	CPU68	CPU69	CPU70	CPU71	CPU72	CPU73	CPU74	CPU75	CPU76	CPU77	CPU78	CPU79	CPU80	CPU81	CPU82	CPU83	CPU84	CPU85	CPU86	CPU87	CPU88	CPU89	CPU90	CPU91	CPU92	CPU93	CPU94	CPU95	CPU96	CPU97	CPU98	CPU99	CPU100	CPU101	CPU102	CPU103	CPU104	CPU105	CPU106	CPU107	CPU108	CPU109	CPU110	CPU111	CPU112	CPU113	CPU114	CPU115	CPU116	CPU117	CPU118	CPU119	CPU120	CPU121	CPU122	CPU123	CPU124	CPU125	CPU126	CPU127	CPU128	CPU129	CPU130	CPU131	CPU132	CPU133	CPU134	CPU135	CPU136	CPU137	CPU138	CPU139	CPU140	CPU141	CPU142	CPU143	CPU144	CPU145	CPU146	CPU147	CPU148	CPU149	CPU150	CPU151	CPU152	CPU153	CPU154	CPU155	CPU156	CPU157	CPU158	CPU159	CPU160	CPU161	CPU162	CPU163	CPU164	CPU165	CPU166	CPU167	CPU168	CPU169	CPU170	CPU171	CPU172	CPU173	CPU174	CPU175	CPU176	CPU177	CPU178	CPU179	CPU180	CPU181	CPU182	CPU183	CPU184	CPU185	CPU186	CPU187	CPU188	CPU189	CPU190	CPU191	CPU192	CPU193	CPU194	CPU195	CPU196	CPU197	CPU198	CPU199	CPU200	CPU201	CPU202	CPU203	CPU204	CPU205	CPU206	CPU207	CPU208	CPU209	CPU210	CPU211	CPU212	CPU213	CPU214	CPU215	CPU216	CPU217	CPU218	CPU219	CPU220	CPU221	CPU222	CPU223	CPU224	CPU225	CPU226	CPU227	CPU228	CPU229	CPU230	CPU231	CPU232	CPU233	CPU234	CPU235	CPU236	CPU237	CPU238	CPU239	CPU240	CPU241	CPU242	CPU243	CPU244	CPU245	CPU246	CPU247	CPU248	CPU249	CPU250	CPU251	CPU252	CPU253	CPU254	CPU255	CPU256	CPU257	CPU258	CPU259	CPU260	CPU261	CPU262	CPU263	CPU264	CPU265	CPU266	CPU267	CPU268	CPU269	CPU270	CPU271	CPU272	CPU273	CPU274	CPU275	CPU276	CPU277	CPU278	CPU279	CPU280	CPU281	CPU282	CPU283	CPU284	CPU285	CPU286	CPU287	CPU288	CPU289	CPU290	CPU291	CPU292	CPU293	CPU294	CPU295	CPU296	CPU297	CPU298	CPU299	CPU300	CPU301	CPU302	CPU303	CPU304	CPU305	CPU306	CPU307	CPU308	CPU309	CPU310	CPU311	CPU312	CPU313	CPU314	CPU315	CPU316	CPU317	CPU318	CPU319	CPU320	CPU321	CPU322	CPU323	CPU324	CPU325	CPU326	CPU327	CPU328	CPU329	CPU330	CPU331	CPU332	CPU333	CPU334	CPU335	CPU336	CPU337	CPU338	CPU339	CPU340	CPU341	CPU342	CPU343	CPU344	CPU345	CPU346	CPU347	CPU348	CPU349	CPU350	CPU351	CPU352	CPU353	CPU354	CPU355	CPU356	CPU357	CPU358	CPU359	CPU360	CPU361	CPU362	CPU363	CPU364	CPU365	CPU366	CPU367	CPU368	CPU369	CPU370	CPU371	CPU372	CPU373	CPU374	CPU375	CPU376	CPU377	CPU378	CPU379	CPU380	CPU381	CPU382	CPU383	CPU384	CPU385	CPU386	CPU387	CPU388	CPU389	CPU390	CPU391	CPU392	CPU393	CPU394	CPU395	CPU396	CPU397	CPU398	CPU399	CPU400	CPU401	CPU402	CPU403	CPU404	CPU405	CPU406	CPU407	CPU408	CPU409	CPU410	CPU411	CPU412	CPU413	CPU414	CPU415	CPU416	CPU417	CPU418	CPU419	CPU420	CPU421	CPU422	CPU423	CPU424	CPU425	CPU426	CPU427	CPU428	CPU429	CPU430	CPU431	CPU432	CPU433	CPU434	CPU435	CPU436	CPU437	CPU438	CPU439	CPU440	CPU441	CPU442	CPU443	CPU444	CPU445	CPU446	CPU447	CPU448	CPU449	CPU450	CPU451	CPU452	CPU453	CPU454	CPU455	CPU456	CPU457	CPU458	CPU459	CPU460	CPU461	CPU462	CPU463	CPU464	CPU465	CPU466	CPU467	CPU468	CPU469	CPU470	CPU471	CPU472	CPU473	CPU474	CPU475	CPU476	CPU477	CPU478	CPU479	CPU480	CPU481	CPU482	CPU483	CPU484	CPU485	CPU486	CPU487	CPU488	CPU489	CPU490	CPU491	CPU492	CPU493	CPU494	CPU495	CPU496	CPU497	CPU498	CPU499	CPU500	CPU501	CPU502	CPU503	CPU504	CPU505	CPU506	CPU507	CPU508	CPU509	CPU510	CPU511	CPU512	CPU513	CPU514	CPU515	CPU516	CPU517	CPU518	CPU519	CPU520	CPU521	CPU522	CPU523	CPU524	CPU525	CPU526	CPU527	CPU528	CPU529	CPU530	CPU531	CPU532	CPU533	CPU534	CPU535	CPU536	CPU537	CPU538	CPU539	CPU540	CPU541	CPU542	CPU543	CPU544	CPU545	CPU546	CPU547	CPU548	CPU549	CPU550	CPU551	CPU552	CPU553	CPU554	CPU555	CPU556	CPU557	CPU558	CPU559	CPU560	CPU561	CPU562	CPU563	CPU564	CPU565	CPU566	CPU567	CPU568	CPU569	CPU570	CPU571	CPU572	CPU573	CPU574	CPU575	CPU576	CPU577	CPU578	CPU579	CPU580	CPU581	CPU582	CPU583	CPU584	CPU585	CPU586	CPU587	CPU588	CPU589	CPU590	CPU591	CPU592	CPU593	CPU594	CPU595	CPU596	CPU597	CPU598	CPU599	CPU600	CPU601	CPU602	CPU603	CPU604	CPU605	CPU606	CPU607	CPU608	CPU609	CPU610	CPU611	CPU612	CPU613	CPU614	CPU615	CPU616	CPU617	CPU618	CPU619	CPU620	CPU621	CPU622	CPU623	CPU624	CPU625	CPU626	CPU627	CPU628	CPU629	CPU630	CPU631	CPU632	CPU633	CPU634	CPU635	CPU636	CPU637	CPU638	CPU639	CPU640	CPU641	CPU642	CPU643	CPU644	CPU645	CPU646	CPU647	CPU648	CPU649	CPU650	CPU651	CPU652	CPU653	CPU654	CPU655	CPU656	CPU657	CPU658	CPU659	CPU660	CPU661	CPU662	CPU663	CPU664	CPU665	CPU666	CPU667	CPU668	CPU669	CPU670	CPU671	CPU672	CPU673	CPU674	CPU675	CPU676	CPU677	CPU678	CPU679	CPU680	CPU681	CPU682	CPU683	CPU684	CPU685	CPU686	CPU687	CPU688	CPU689	CPU690	CPU691	CPU692	CPU693	CPU694	CPU695	CPU696	CPU697	CPU698	CPU699	CPU700	CPU701	CPU702	CPU703	CPU704	CPU705	CPU706	CPU707	CPU708	CPU709	CPU710	CPU711	CPU712	CPU713	CPU714	CPU715	CPU716	CPU717	CPU718	CPU719	CPU720	CPU721	CPU722	CPU723	CPU724	CPU725	CPU726	CPU727	CPU728	CPU729	CPU730	CPU731	CPU732	CPU733	CPU734	CPU735	CPU736	CPU737	CPU738	CPU739	CPU740	CPU741	CPU742	CPU743	CPU744	CPU745	CPU746	CPU747	CPU748	CPU749	CPU750	CPU751	CPU752	CPU753	CPU754	CPU755	CPU756	CPU757	CPU758	CPU759	CPU760	CPU761	CPU762	CPU763	CPU764	CPU765	CPU766	CPU767	CPU768	CPU769	CPU770	CPU771	CPU772	CPU773	CPU774	CPU775	CPU776	CPU777	CPU778	CPU779	CPU780	CPU781	CPU782	CPU783	CPU784	CPU785	CPU786	CPU787	CPU788	CPU789	CPU790	CPU791	CPU792	CPU793	CPU794	CPU795	CPU796	CPU797	CPU798	CPU799	CPU800	CPU801	CPU802	CPU803	CPU804	CPU805	CPU806	CPU807	CPU808	CPU809	CPU810	CPU811	CPU812	CPU813	CPU814	CPU815	CPU816	CPU817	CPU818	CPU819	CPU820	CPU821	CPU822	CPU823	CPU824	CPU825	CPU826	CPU827	CPU828	CPU829	CPU830	CPU831	CPU832	CPU833	CPU834	CPU835	CPU836	CPU837	CPU838	CPU839	CPU840	CPU841	CPU842	CPU843	CPU844	CPU845	CPU846	CPU847	CPU848	CPU849	CPU850	CPU851	CPU852	CPU853	CPU854	CPU855	CPU856	CPU857	CPU858	CPU859	CPU860	CPU861	CPU862	CPU863	CPU864	CPU865	CPU866	CPU867	CPU868	CPU869	CPU870	CPU871	CPU872	CPU873	CPU874	CPU875	CPU876	CPU877	CPU878	CPU879	CPU880	CPU881	CPU882	CPU883	CPU884	CPU885	CPU886	CPU887	CPU888	CPU889	CPU890	CPU891	CPU892	CPU893	CPU894	CPU895	CPU896	CPU897	CPU898	CPU899	CPU900	CPU901	CPU902	CPU903	CPU904	CPU905	CPU906	CPU907	CPU908	CPU909	CPU910	CPU911	CPU912	CPU913	CPU914	CPU915	CPU916	CPU917	CPU918	CPU919	CPU920	CPU921	CPU922	CPU923	CPU924	CPU925	CPU926	CPU927	CPU928	CPU929	CPU930	CPU931	CPU932	CPU933	CPU934	CPU935	CPU936	CPU937	CPU938	CPU939	CPU940	CPU941	CPU942	CPU943	CPU944	CPU945	CPU946	CPU947	CPU948	CPU949	CPU950	CPU951	CPU952	CPU953	CPU954	CPU955	CPU956	CPU957	CPU958	CPU959	CPU960	CPU961	CPU962	CPU963	CPU964	CPU965	CPU966	CPU967	CPU968	CPU969	CPU970	CPU971	CPU972	CPU973	CPU974	CPU975	CPU976	CPU977	CPU978	CPU979	CPU980	CPU981	CPU982	CPU983	CPU984	CPU985	CPU986	CPU987	CPU988	CPU989	CPU990	CPU991	CPU992	CPU993	CPU994	CPU995	CPU996	CPU997	CPU998	CPU999	CPU1000	CPU1001	CPU1002	CPU1003	CPU1004	CPU1005	CPU1006	CPU1007	CPU1008	CPU1009	CPU1010	CPU1011	CPU1012	CPU1013	CPU1014	CPU1015	CPU1016	CPU1017	CPU1018	CPU1019	CPU1020	CPU1021	CPU1022	CPU1023	CPU1024	CPU1025	CPU1026	CPU1027	CPU1028	CPU1029	CPU1030	CPU1031	CPU1032	CPU1033	CPU1034	CPU1035	CPU1036	CPU1037	CPU1038	CPU1039	CPU1040	CPU1041	CPU1042	CPU1043	CPU1044	CPU1045	CPU1046	CPU1047	CPU1048	CPU1049	CPU1050	CPU1051	CPU1052	CPU1053	CPU1054	CPU1055	CPU1056	CPU1057	CPU1058	CPU1059	CPU1060	CPU1061	CPU1062	CPU1063	CPU1064	CPU1065	CPU1066	CPU1067	CPU1068	CPU1069	CPU1070	CPU1071	CPU1072	CPU1073	CPU1074	CPU1075	CPU1076	CPU1077	CPU1078	CPU1079	CPU1080	CPU1081	CPU1082	CPU1083	CPU1084	CPU1085	CPU1086	CPU1087	CPU1088	CPU1089	CPU1090	CPU1091	CPU1092	CPU1093	CPU1094	CPU1095	CPU1096	CPU1097	CPU1098	CPU1099	CPU1100	CPU1101	CPU1102	CPU1103	CPU1104	CPU1105	CPU1106	CPU1107	CPU1108	CPU1109	CPU1110	CPU1111	CPU1112	CPU1113	CPU1114	CPU1115	CPU1116	CPU1117	CPU1118	CPU1119	CPU1120	CPU1121	CPU1122	CPU1123	CPU1124	CPU1125	CPU1126	CPU1127	CPU1128	CPU1129	CPU1130	CPU1131	CPU1132	CPU1133	CPU1134	CPU1135	CPU1136	CPU1137	CPU1138	CPU1139	CPU1140	CPU1141	CPU1142	CPU1143	CPU1144	CPU1145	CPU1146	CPU1147	CPU1148	CPU1149	CPU1150	CPU1151	CPU1152	CPU1153	CPU1154	CPU1155	CPU1156	CPU1157	CPU1158	CPU1159	CPU1160	CPU1161	CPU1162	CPU1163	CPU1164	CPU1165	CPU1166	CPU1167	CPU1168	CPU1169	CPU1170	CPU1171	CPU1172	CPU1173	CPU1174	CPU1175	CPU1176	CPU1177	CPU1178	CPU1179	CPU1180	CPU1181	CPU1182	CPU1183	CPU1184	CPU1185	CPU1186	CPU1187	CPU1188	CPU1189	CPU1190	CPU1191	CPU1192	CPU1193	CPU1194	CPU1195	CPU1196	CPU1197	CPU1198	CPU1199	CPU1200	CPU1201	CPU1202	CPU1203	CPU1204	CPU1205	CPU1206	CPU1207	CPU1208	CPU1209	CPU1210	CPU1211	CPU1212	CPU1213	CPU1214	CPU1215	CPU1216	CPU1217	CPU1218	CPU1219	CPU1220	CPU1221	CPU1222	CPU1223	CPU1224	CPU1225	CPU1226	CPU1227	CPU1228	CPU1229	CPU1230	CPU1231	CPU1232	CPU1233



Перерыв!

Streaming  
Pübiprocessor

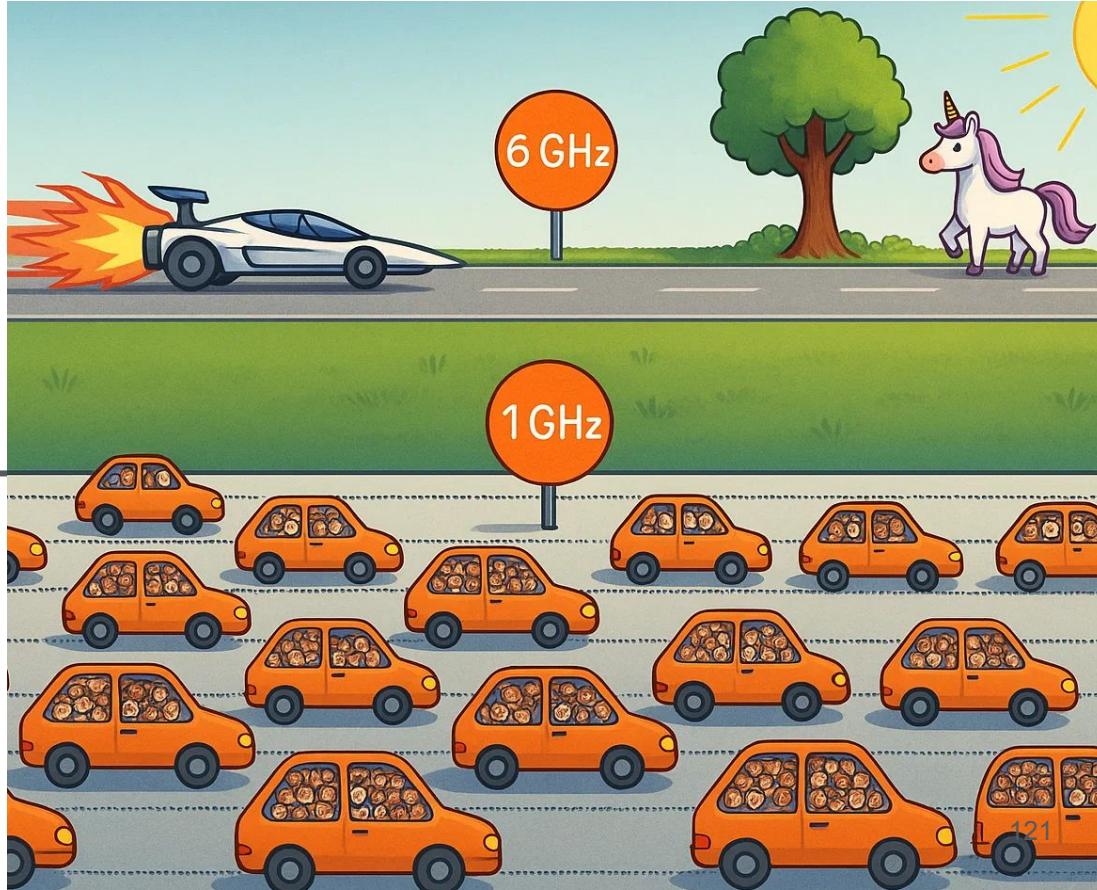
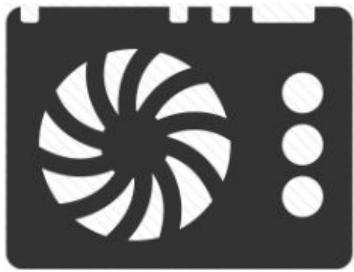
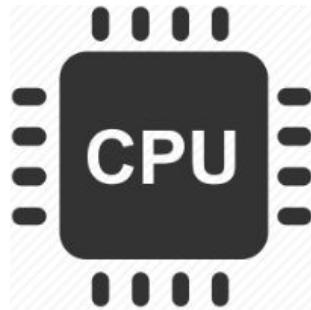
Лицензия

© VIDIA

# Глава 4: Модель вычислений массового параллелизма

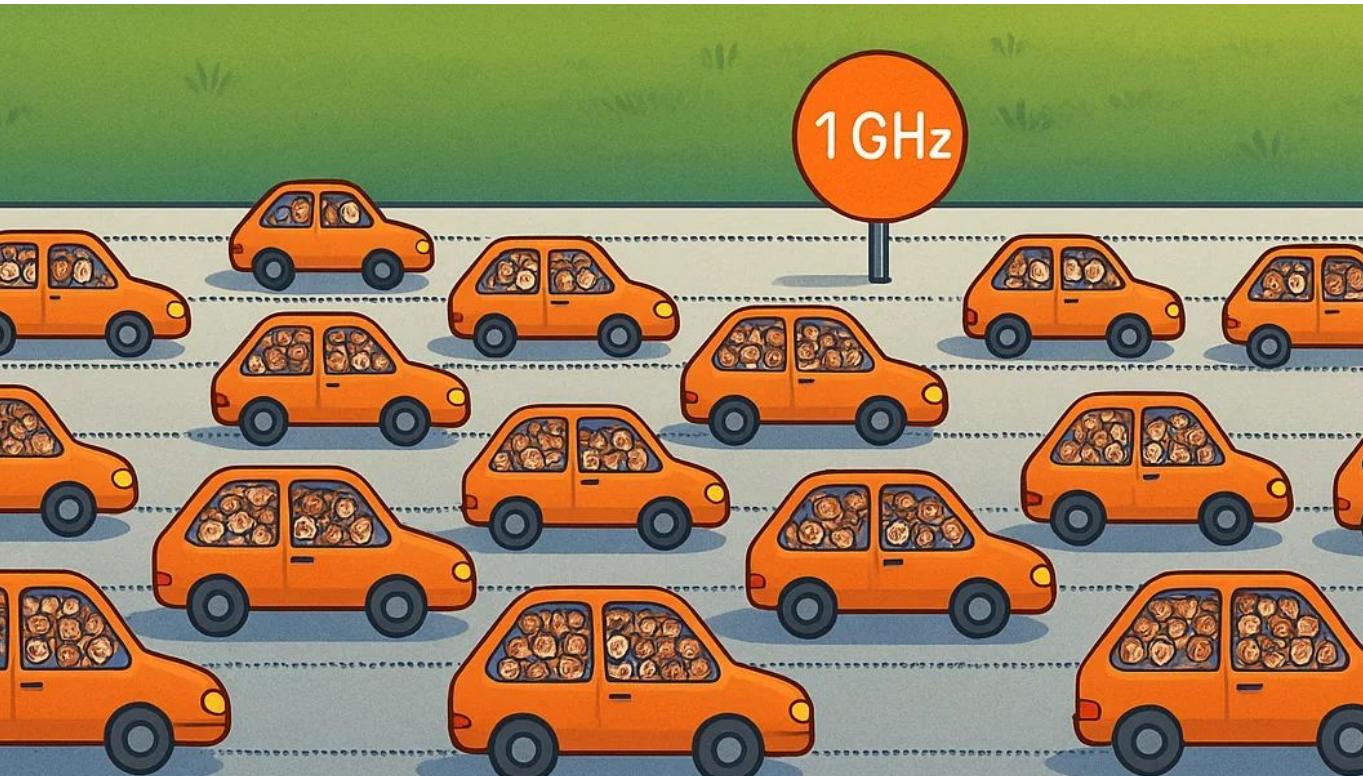


# Модель вычислений массового параллелизма



# Модель вычислений массового параллелизма

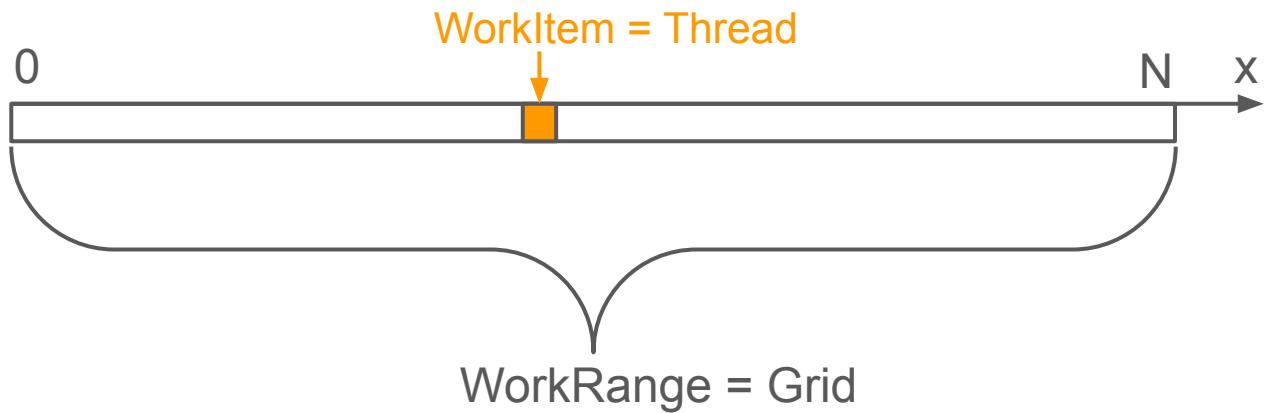
SM - Streaming Multiprocessor



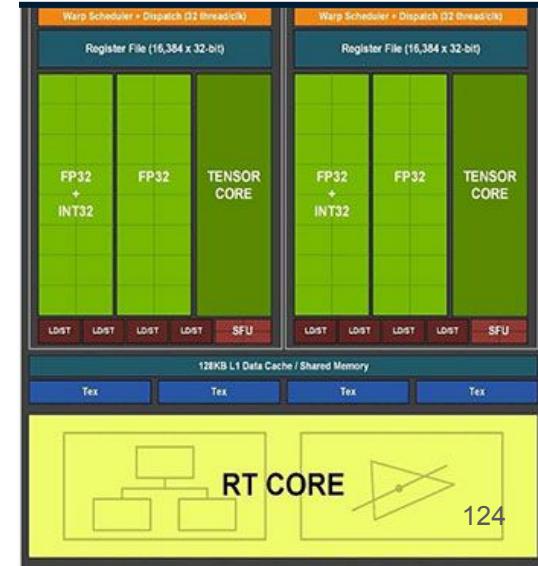
# Модель вычислений массового параллелизма



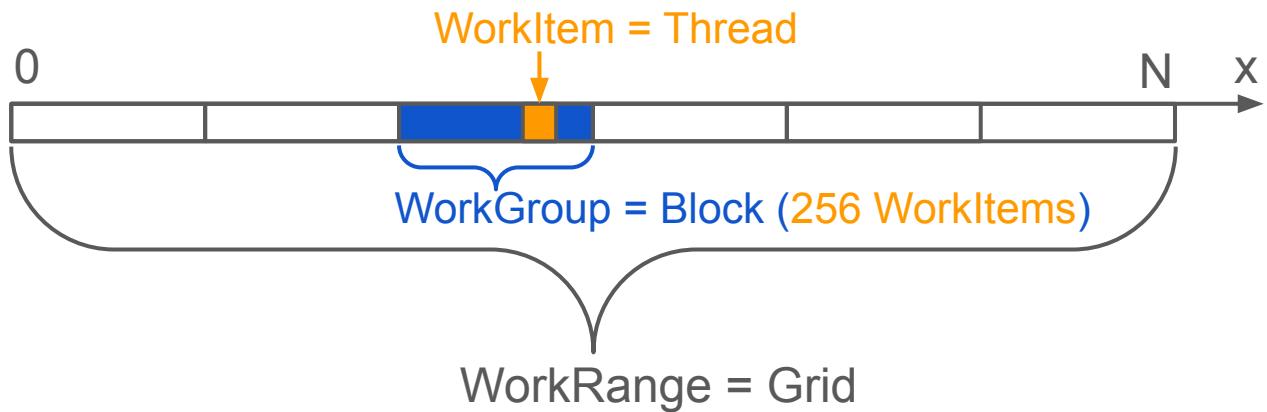
# Модель вычислений массового параллелизма



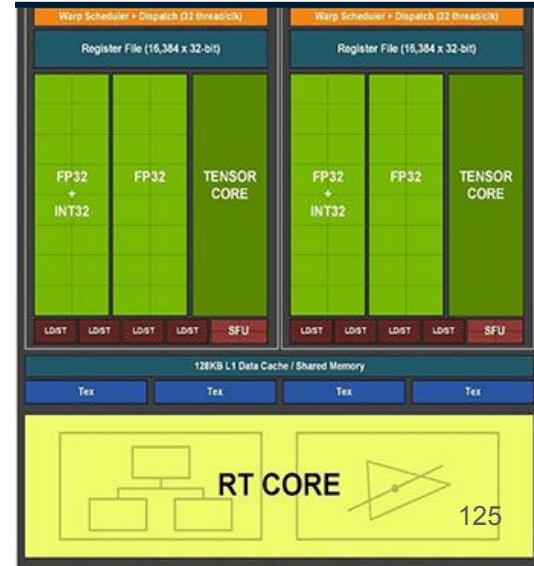
SM - Streaming Multiprocessor



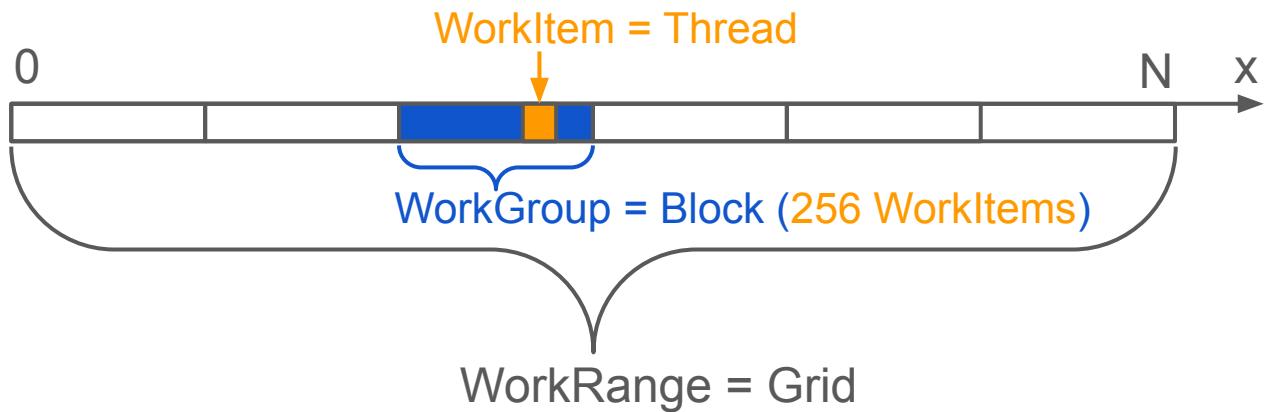
# Модель вычислений массового параллелизма



Много SM  
SM - Streaming Multiprocessor



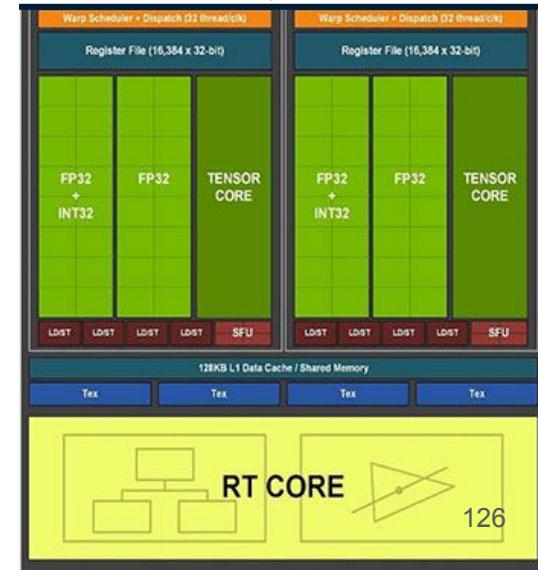
# Модель вычислений массового параллелизма



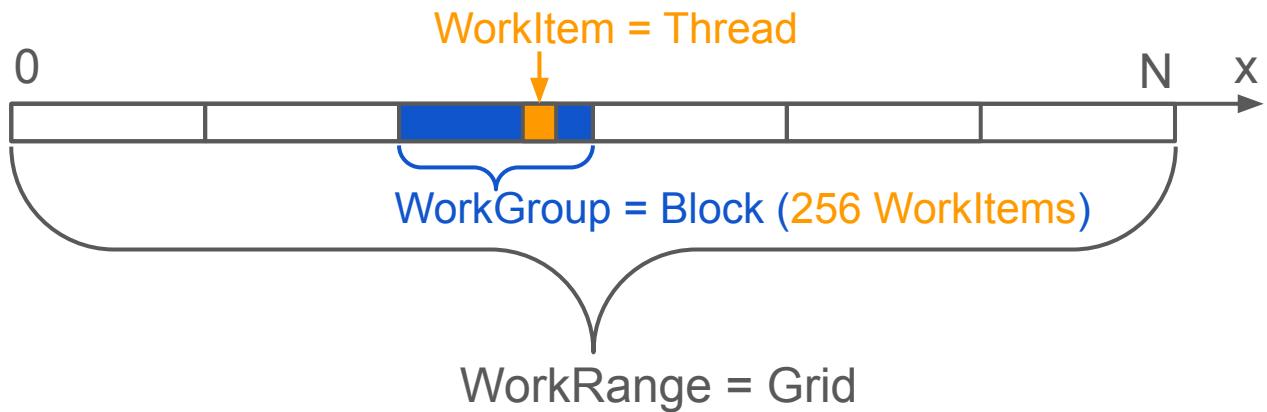
Но ведь в одном warp 32 потока!  
(у **AMD**: 64 потока в wavefront)  
Как так?



SM - Streaming Multiprocessor



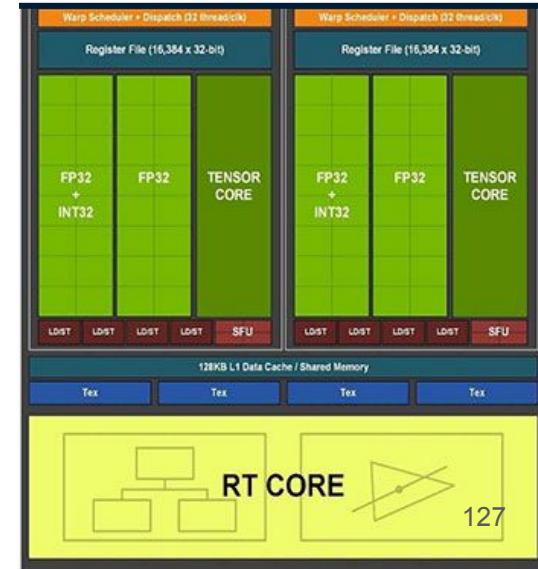
# Модель вычислений массового параллелизма



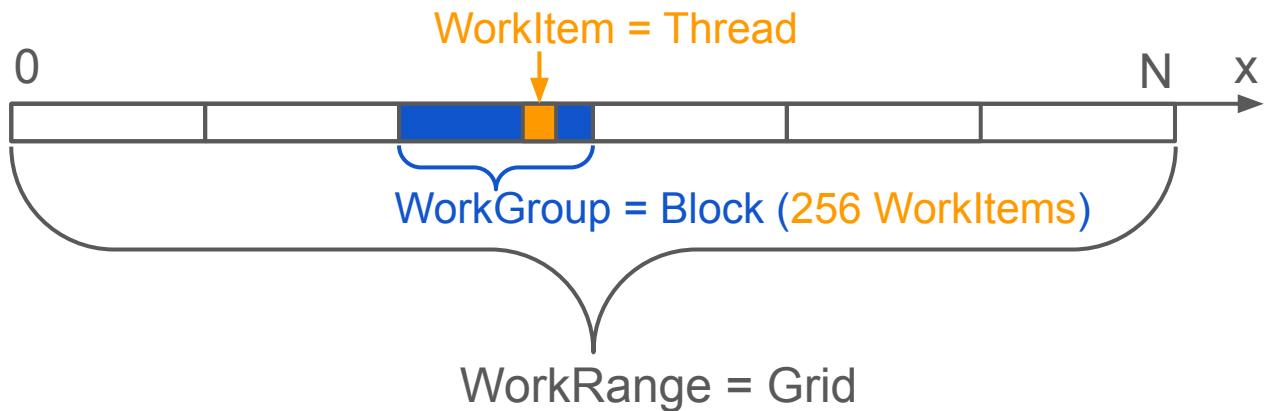
WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)



Много SM  
SM - Streaming Multiprocessor



# Модель вычислений массового параллелизма

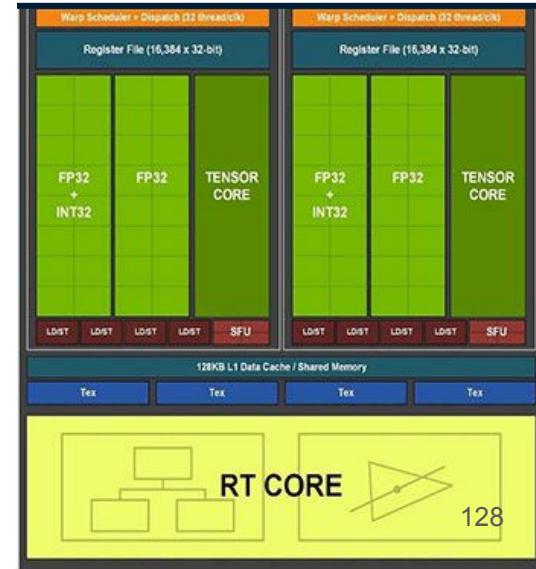


WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)

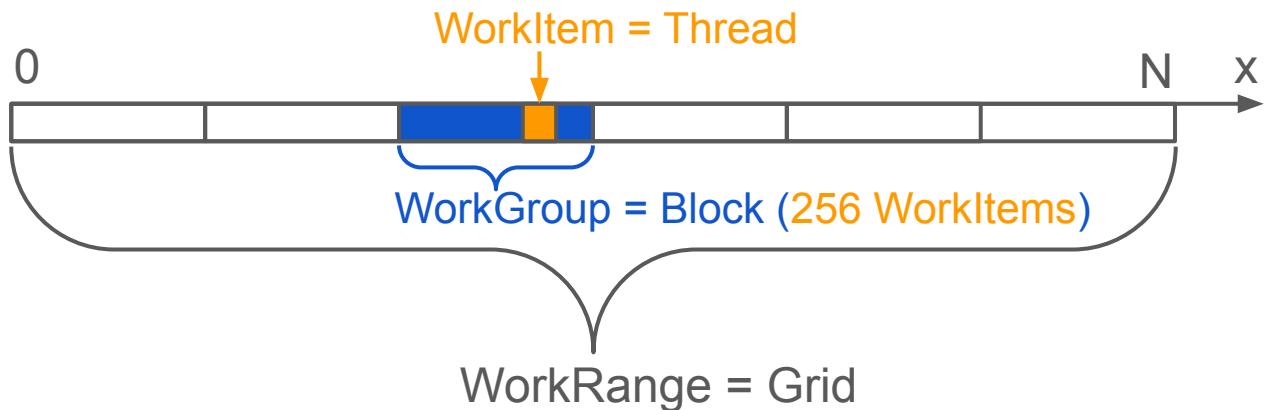
Зачем нам контроль над **WorkGroups**?  
Зачем на уровне API знать про warps?



Много SM  
SM - Streaming Multiprocessor



# Модель вычислений массового параллелизма



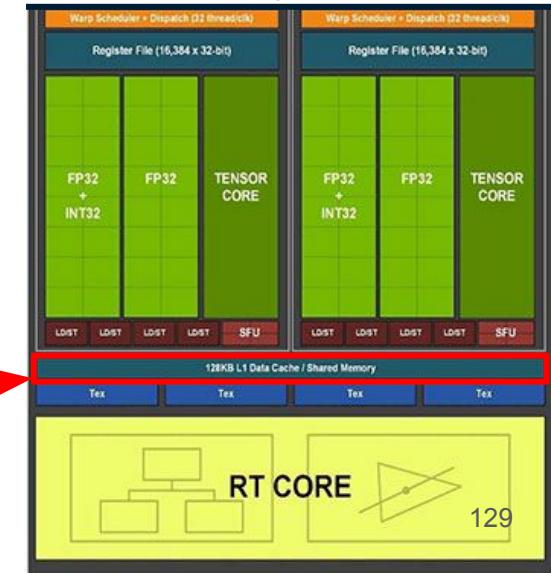
WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)

- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)



Много SM

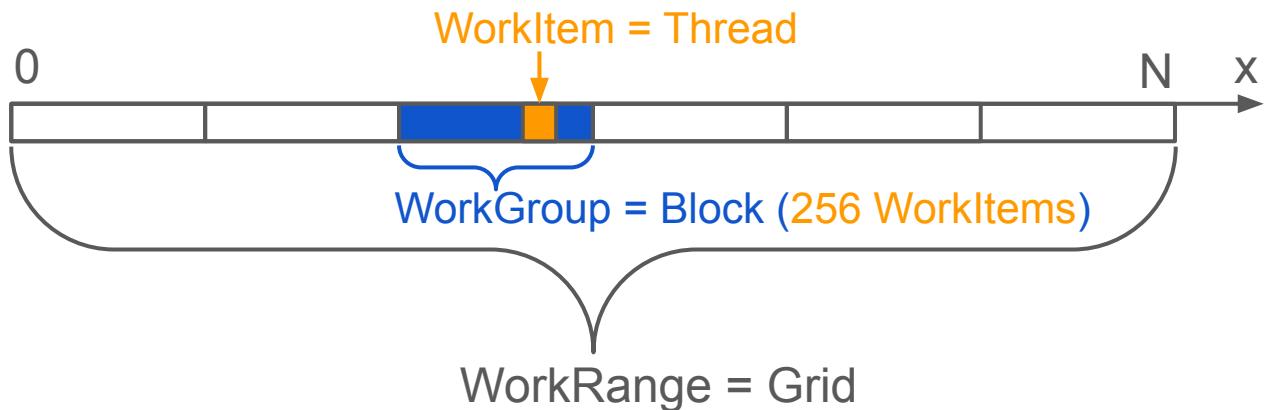
SM - Streaming Multiprocessor



RT CORE

129

# Модель вычислений массового параллелизма



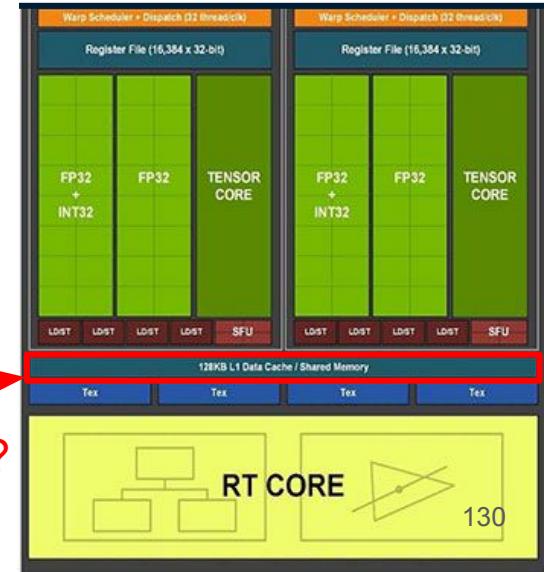
WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)

- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)

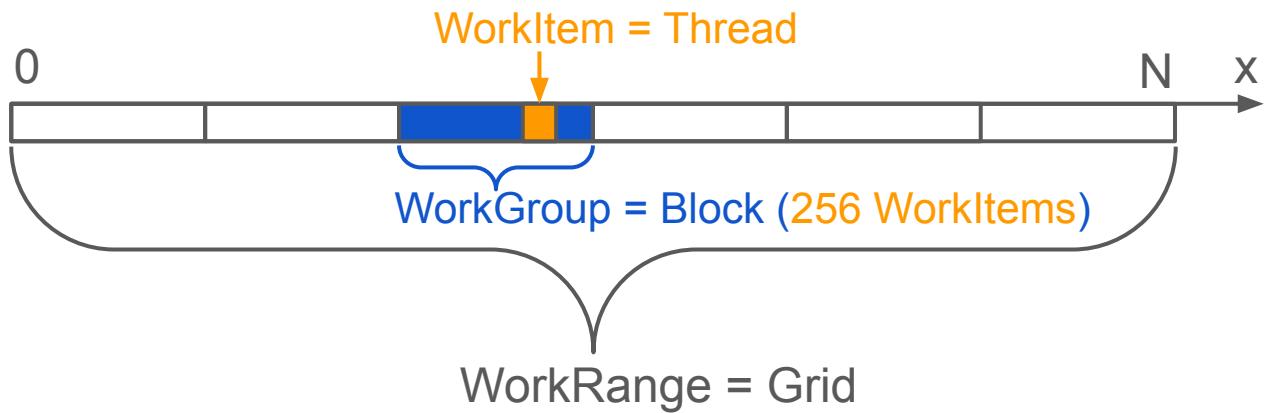
Могут ли warp-ы одной WorkGroup исполняться на разных SM?



SM - Streaming Multiprocessor



# Модель вычислений массового параллелизма



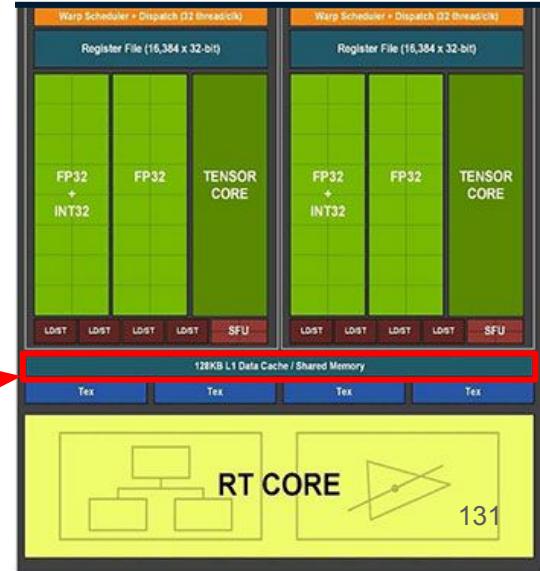
WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)

- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)

А если один такой поток записал в Local Memory  
число - увидит ли другой поток этой WorkGroup?



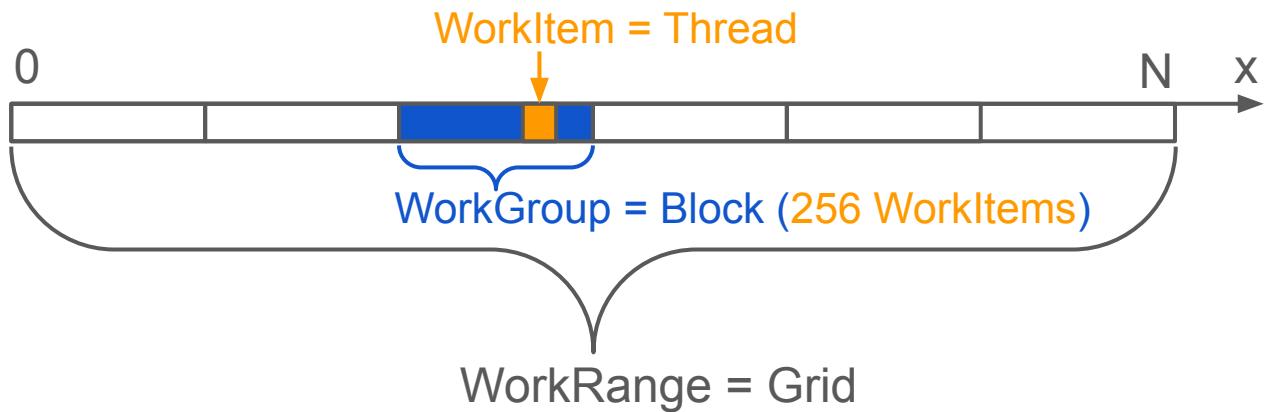
Много SM  
SM - Streaming Multiprocessor



RT CORE

131

# Модель вычислений массового параллелизма



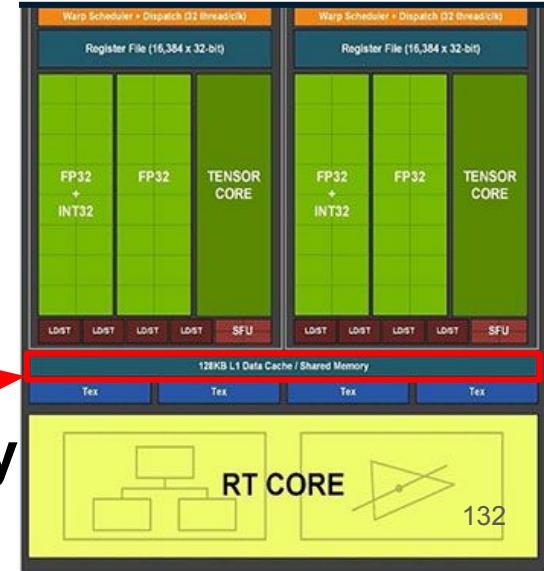
WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)

- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)

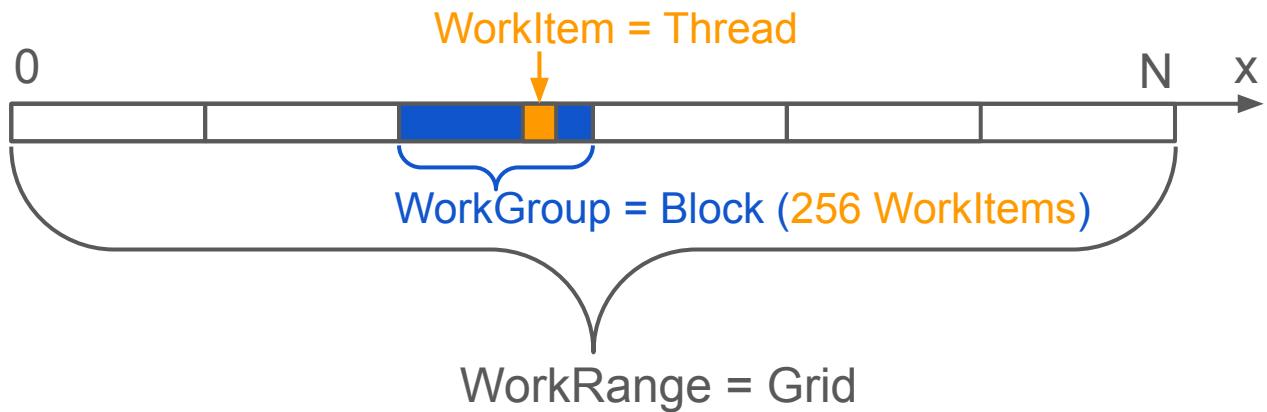
А если один такой поток записал в Local Memory memory число - увидит ли другой поток этой WorkGroup? barrier!



SM - Streaming Multiprocessor

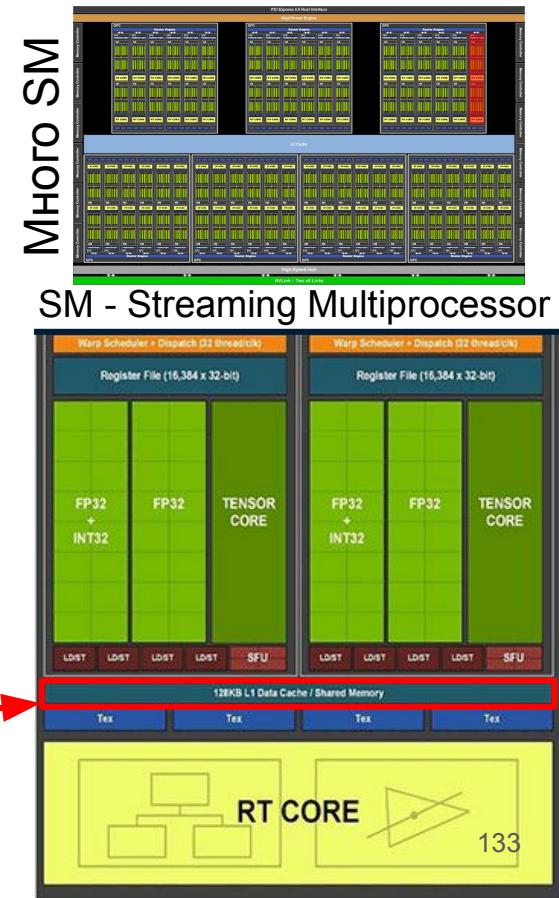


# Модель вычислений массового параллелизма



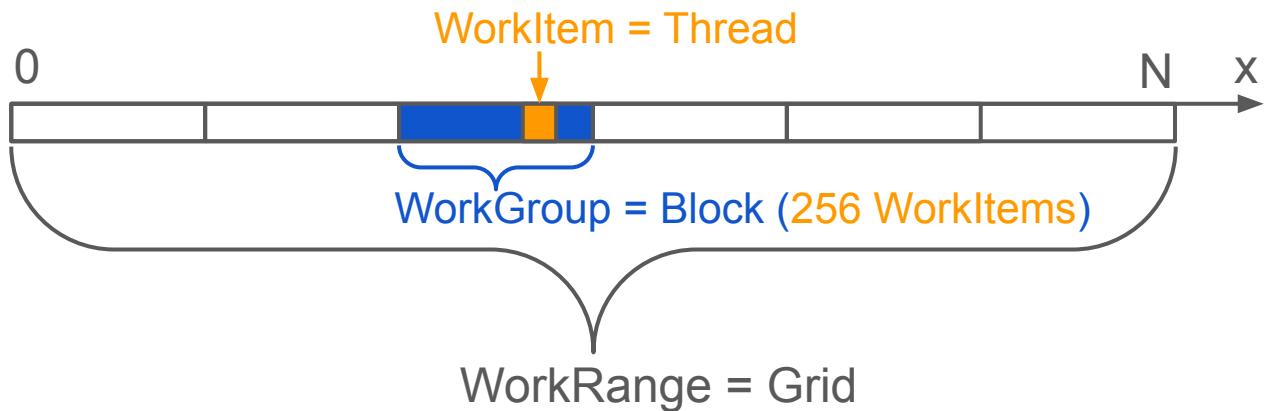
WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)

- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)



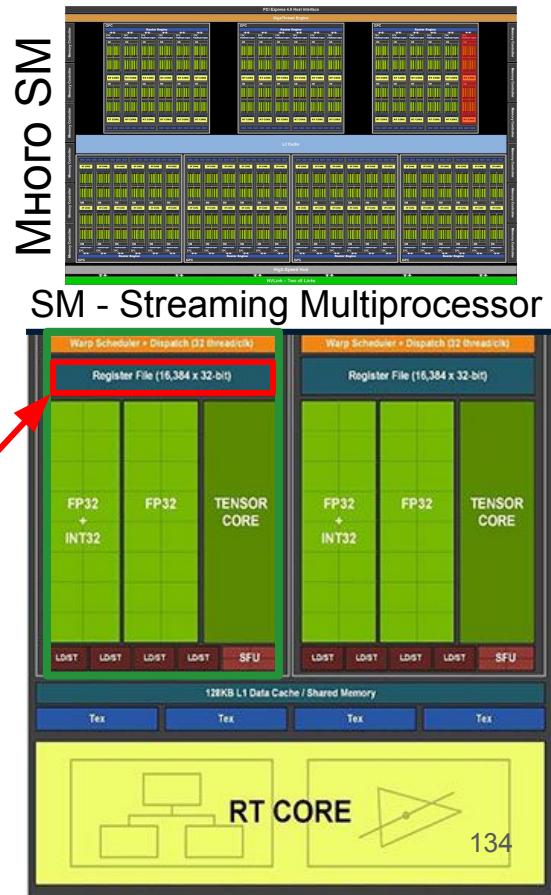
Могут ли потоки одного warp-а общаться еще эффективнее?

# Модель вычислений массового параллелизма

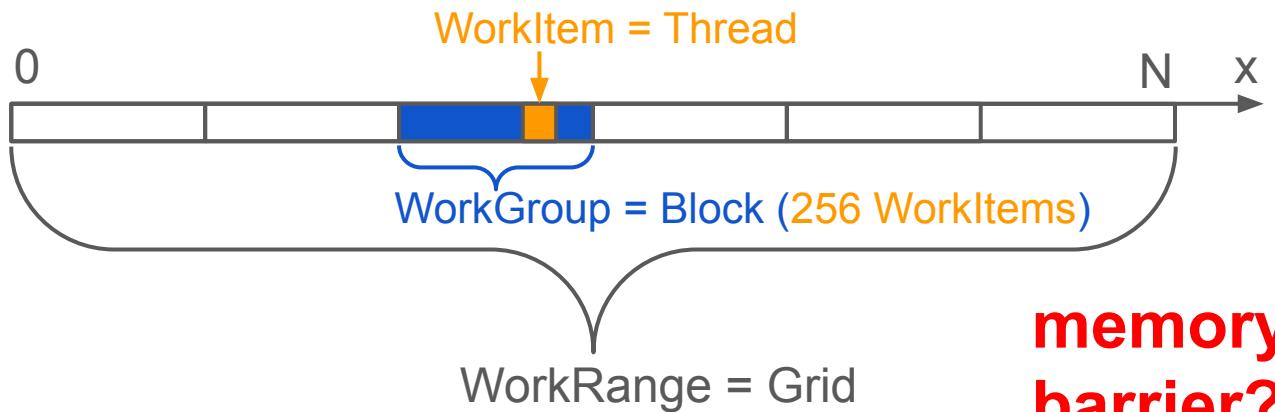


WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)

- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)
- WorkItems в рамках одного warp-а могут подглядывать друг другу в регистры (shuffle instr.)



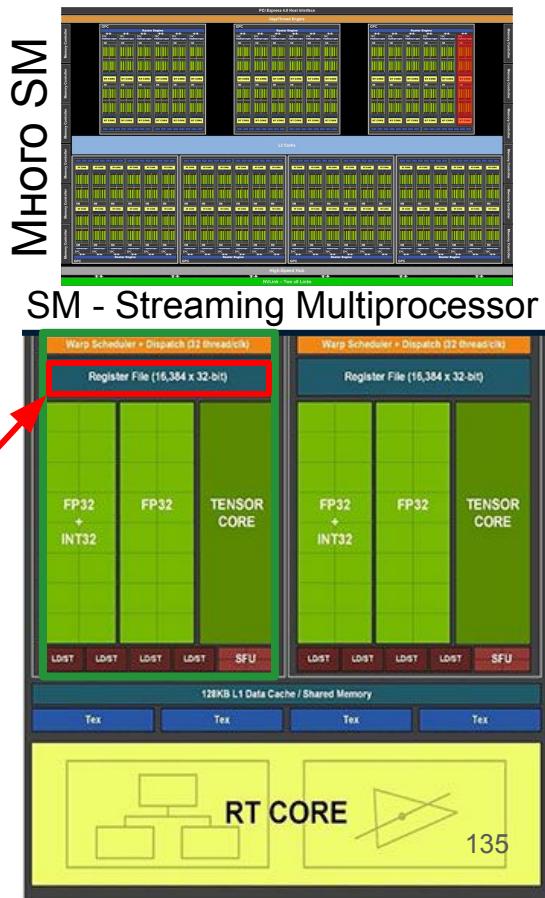
# Модель вычислений массового параллелизма



WorkGroup = Block (256 WorkItems) = 8 x Warps (32 threads)

**memory  
barrier?**

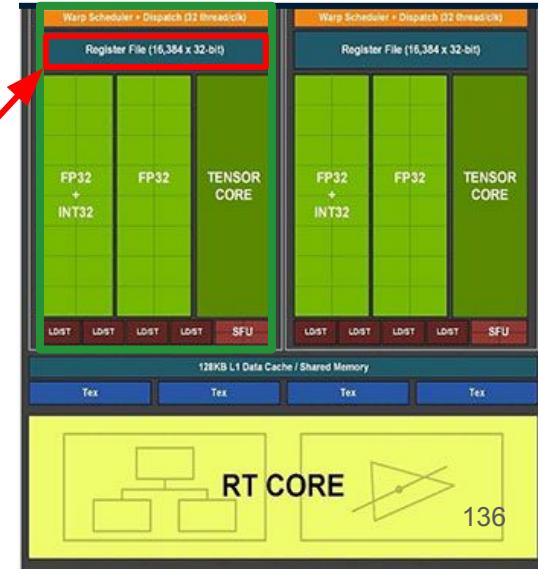
- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)
- WorkItems в рамках одного warp-а могут подглядывать друг другу в регистры (shuffle instr.)



**dFdx(...), dFdy(...)** — return the partial derivative of an argument  
with respect to x or y

- **WorkItems** коммуницируют через VRAM
- **WorkItems** в рамках одной **WorkGroup** общаются эффективнее - через **Shared/Local Memory (L1)**
- **WorkItems** в рамках одного warp-а могут подглядывать друг другу в **регистры (*shuffle instr.*)**

SM - Streaming Multiprocessor

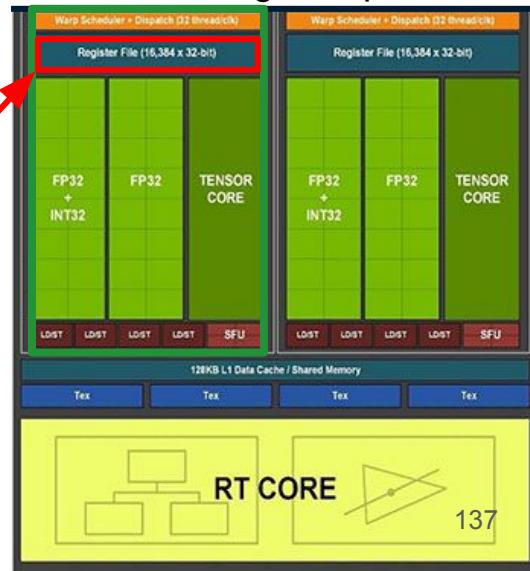


**dFdx(...), dFdy(...)** — return the partial derivative of an argument with respect to x or y

$$\text{dFdx}(p(x,y)) = p(x+1,y) - p(x,y)$$

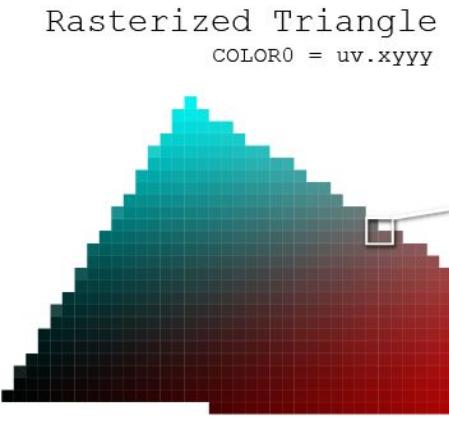
$$\text{dFdy}(p(x,y)) = p(x,y+1) - p(x,y)$$

SM - Streaming Multiprocessor



- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)
- WorkItems в рамках одного warp-а могут подглядывать друг другу в регистры (*shuffle instr.*)

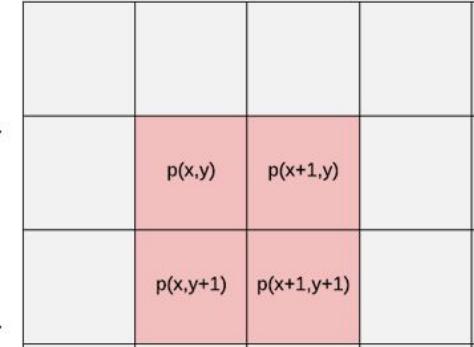
$dFdx(\dots), dFdy(\dots)$  — return the partial derivative of an argument with respect to x or y



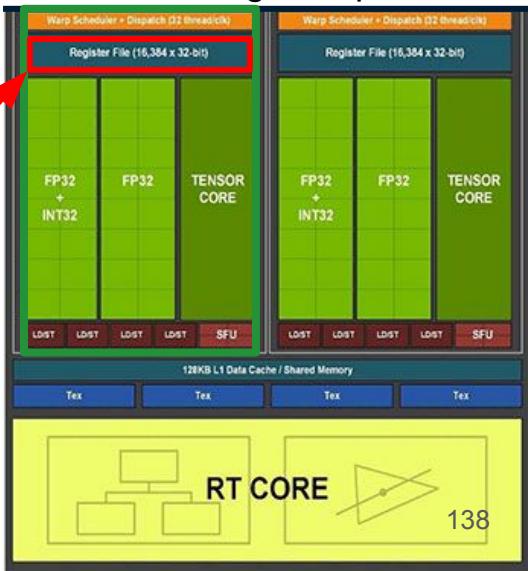
Derivatives for bottom-right fragment (high-precision)

Shading Quad (2x2 screen pixels)  
 $dFdy(p(x,y)) = p(x,y+1) - p(x,y)$

$$\begin{aligned}
 & uv = 0.498, 0.489 & uv = 0.533, 0.487 \\
 & ddy(uv.x) = 0.533 - 0.542 & = -0.009 \\
 & & ddy(uv.y) = 0.487 - 0.433 \\
 & uv = 0.507, 0.436 & = 0.054 \\
 & uv = 0.542, 0.433 & \\
 & ddx(uv.x) = 0.542 - 0.507 & = 0.035 \\
 & & ddx(uv.y) = 0.433 - 0.436 \\
 & & = -0.003
 \end{aligned}$$

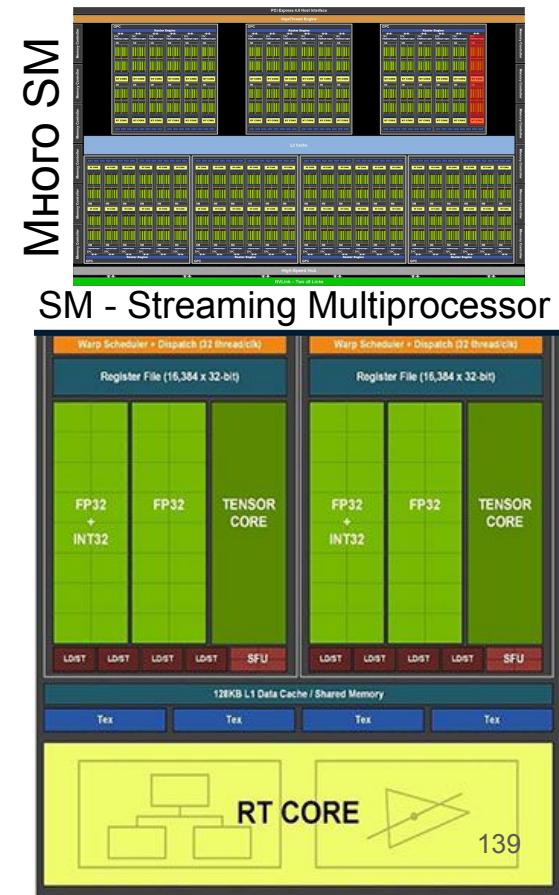
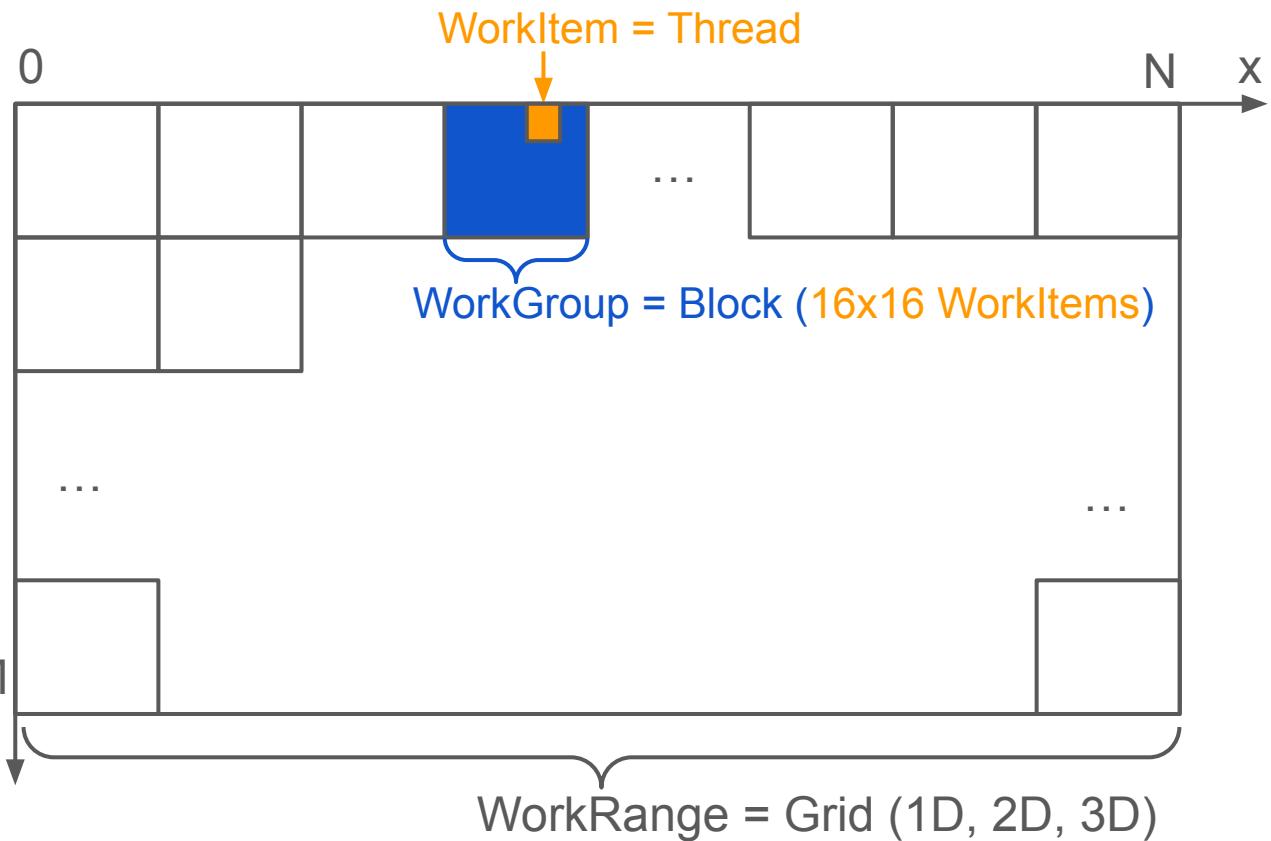


SM - Streaming Multiprocessor

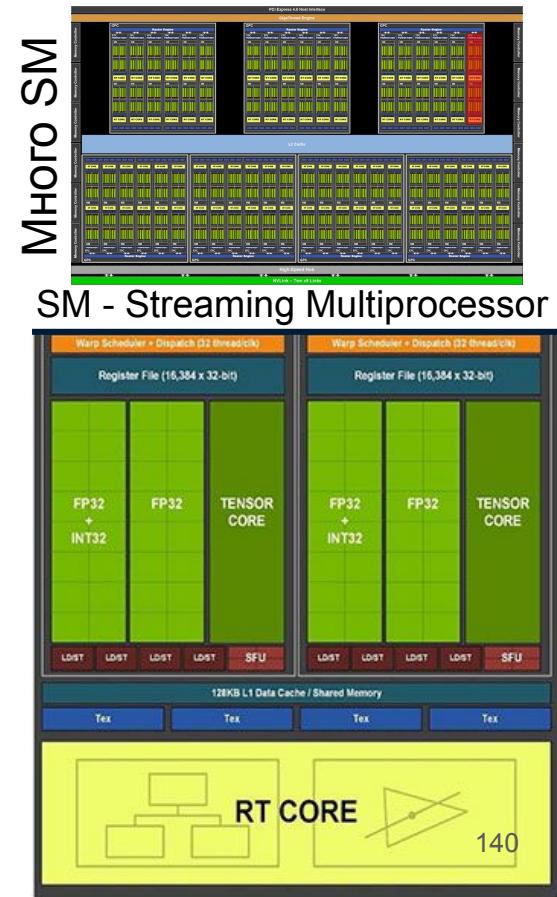
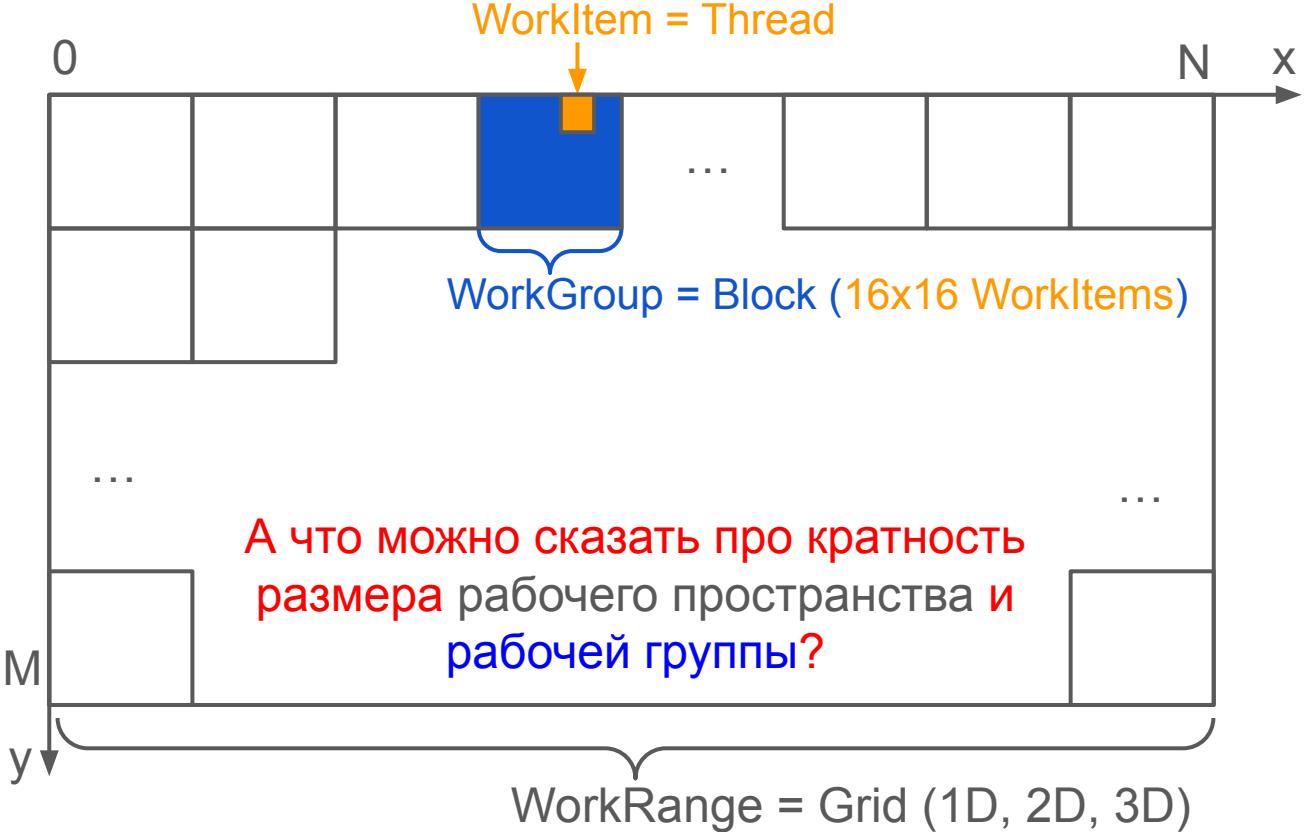


- WorkItems коммуницируют через VRAM
- WorkItems в рамках одной WorkGroup общаются эффективнее - через Shared/Local Memory (L1)
- WorkItems в рамках одного warp-а МОГУТ подглядывать друг другу в регистры (shuffle instr.)

# Модель вычислений массового параллелизма



# Модель вычислений массового параллелизма



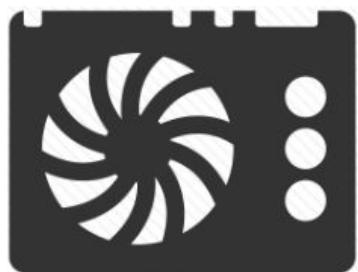
# Глава 5: Профилирование и оптимизация



МНОГО вычислений? (**ALU**)

МНОГО читать+писать? (**VRAM**)

$100 \cdot 10^{12}$  FLOPS



1000 GB/s



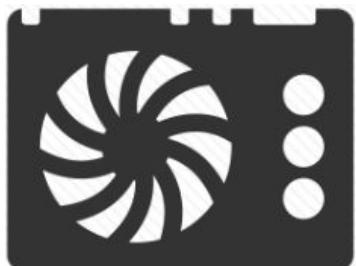
VRAM - GDDR6

Чего больше?  
Как сравнить яблоки и  
апельсины?

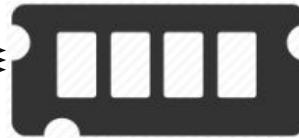
МНОГО вычислений? (**ALU**)

МНОГО читать+писать? (**VRAM**)

$100 \cdot 10^{12}$  FLOPS



1000 GB/s



VRAM - GDDR6

Чего больше?

МНОГО вычислений? (**ALU**)

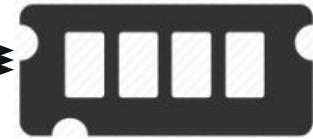
МНОГО читать+писать? (**VRAM**)

$100 \cdot 10^{12}$  FLOPS



400:1

1000 GB/s



VRAM - GDDR6

Если **ALU:VRAM** > 400:1

Чего больше?  
Какая пропорция  
**ALU:VRAM?**

Если **ALU:VRAM** < 400:1

МНОГО вычислений? (**ALU**)

МНОГО читать+писать? (**VRAM**)

$100 \cdot 10^{12}$  FLOPS



400:1

1000 GB/s



VRAM - GDDR6

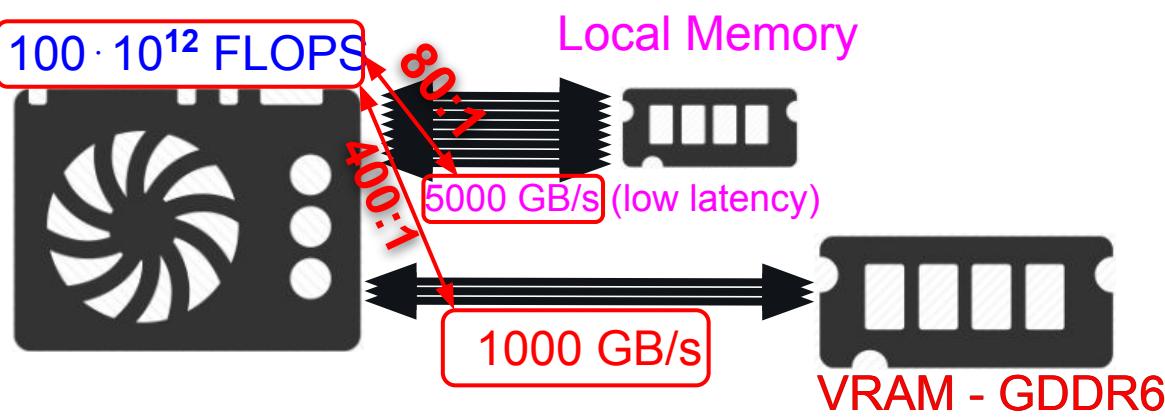
Если **ALU:VRAM** > 400:1

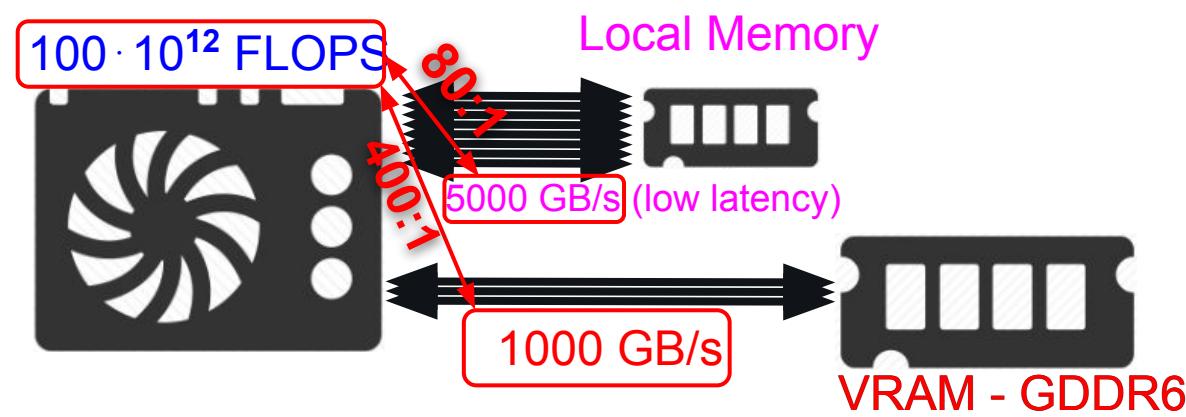
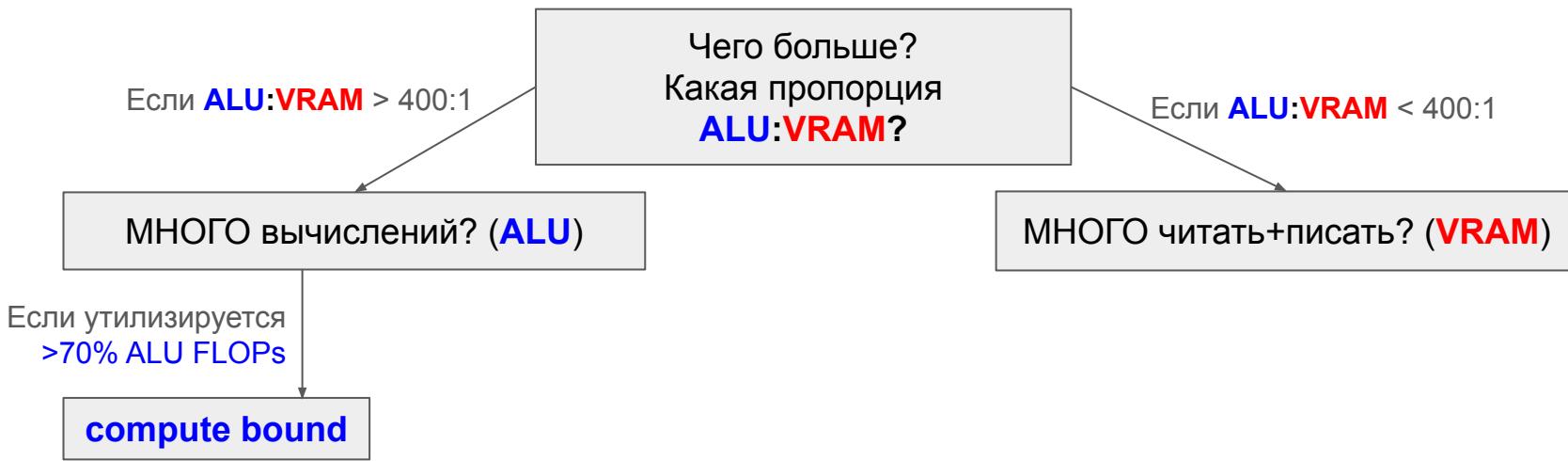
Чего больше?  
Какая пропорция  
**ALU:VRAM**?

Если **ALU:VRAM** < 400:1

МНОГО вычислений? (**ALU**)

МНОГО читать+писать? (**VRAM**)





Чего больше?  
Какая пропорция  
**ALU:VRAM?**

Если **ALU:VRAM** > 400:1

МНОГО вычислений? (**ALU**)

Если утилизируется  
>70% ALU FLOPs

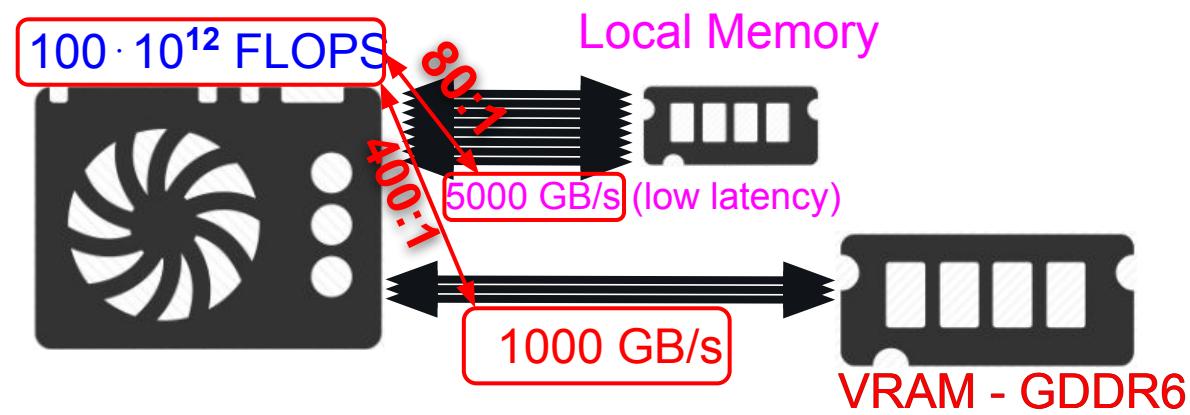
**compute bound**

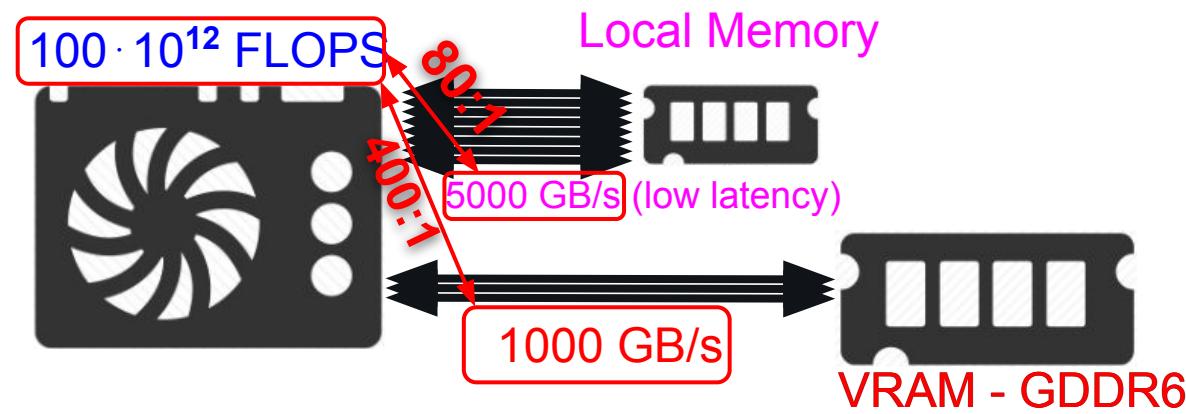
Устранить  
**code divergence**

RECURSION  
Here we go again  
RECURSION  
Here we go again  
RECURSION  
Here we go again

Если **ALU:VRAM** < 400:1

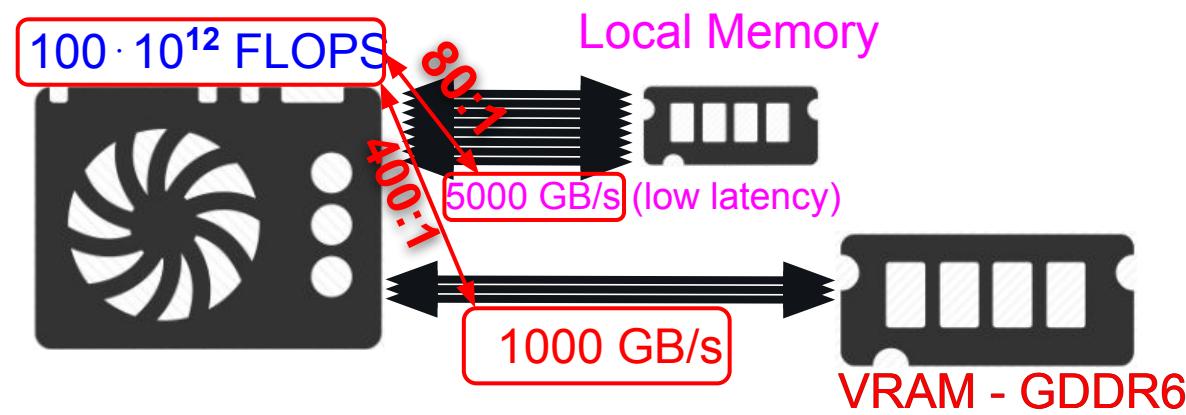
МНОГО читать+писать? (**VRAM**)

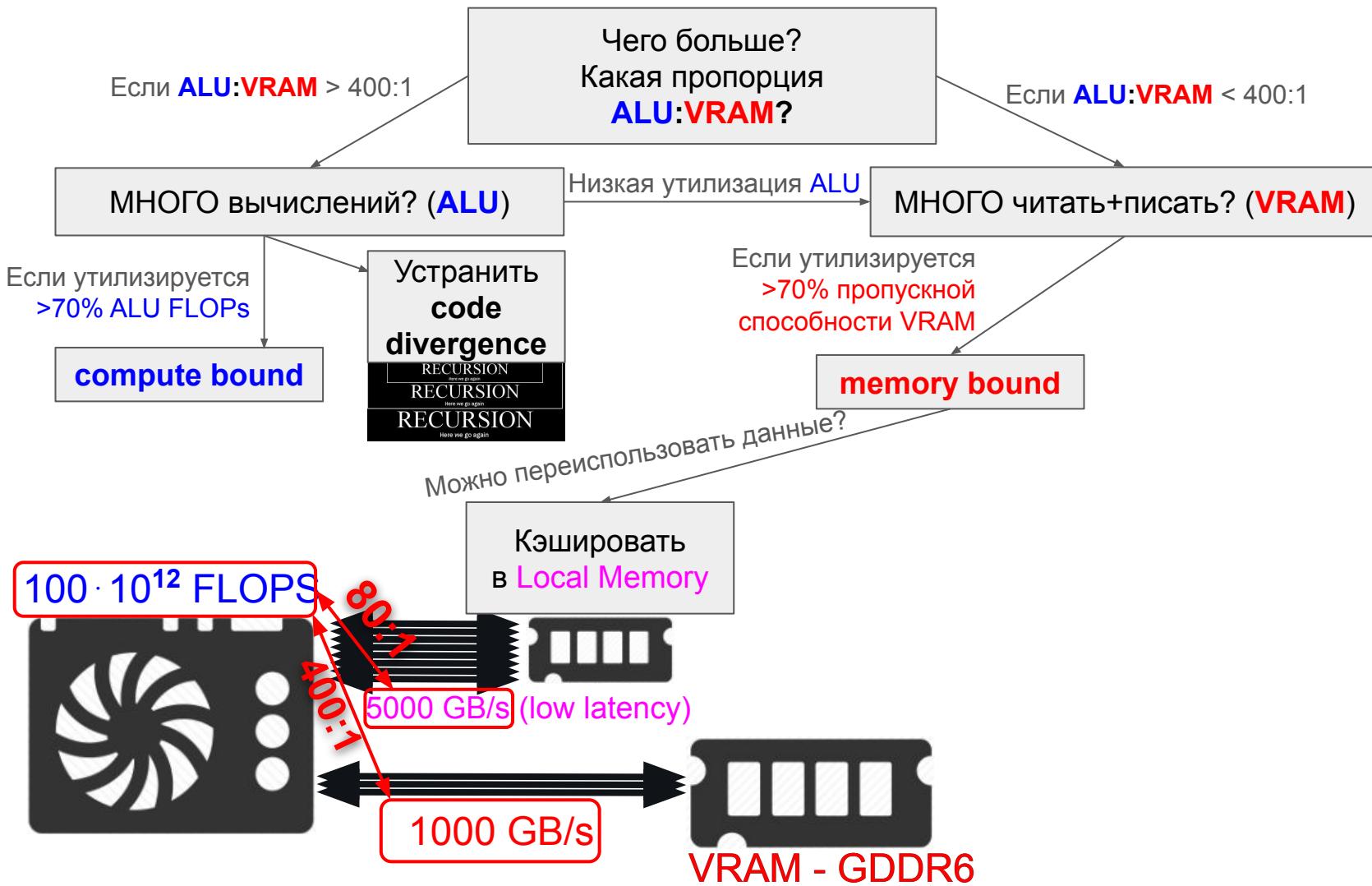


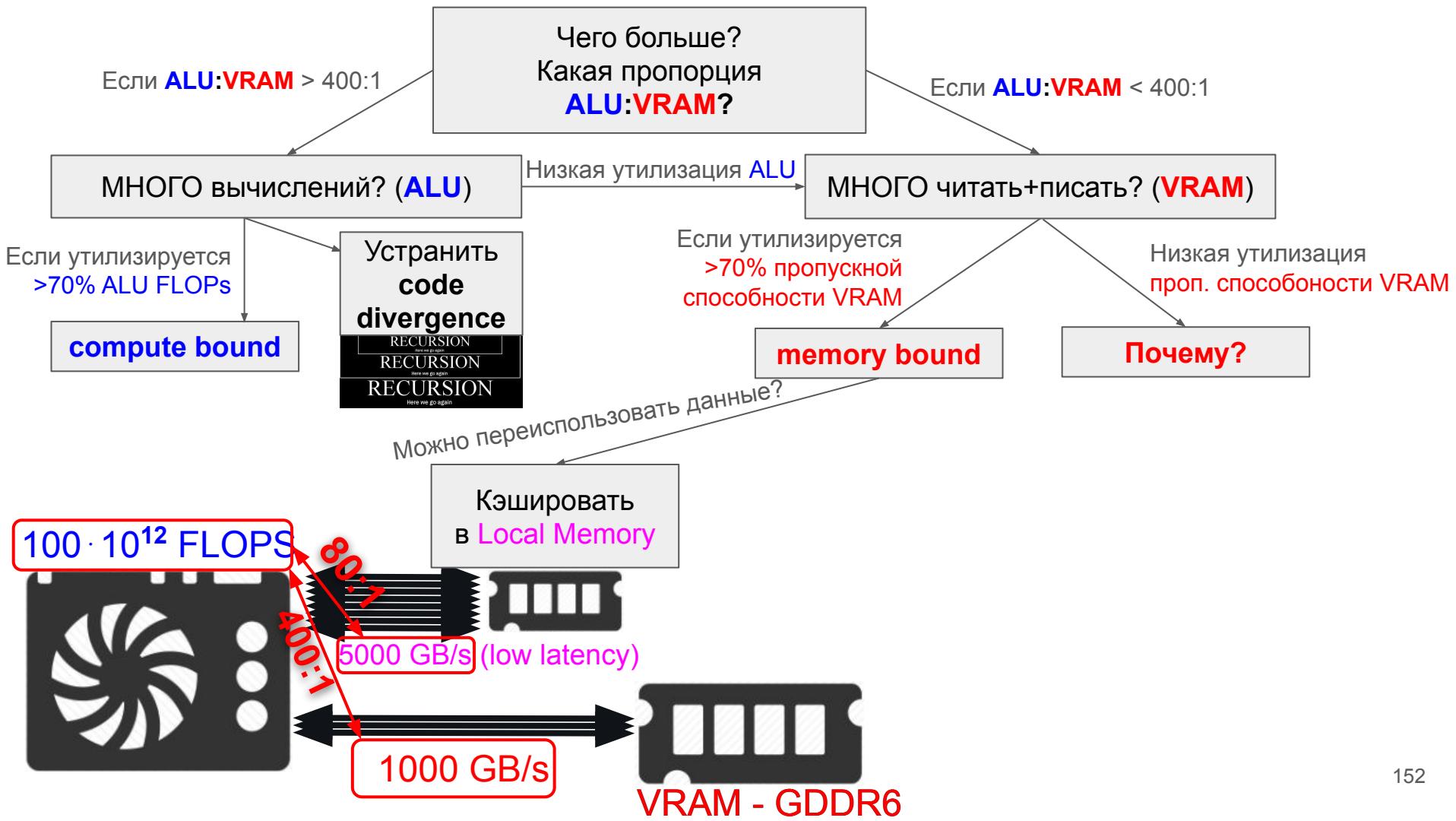


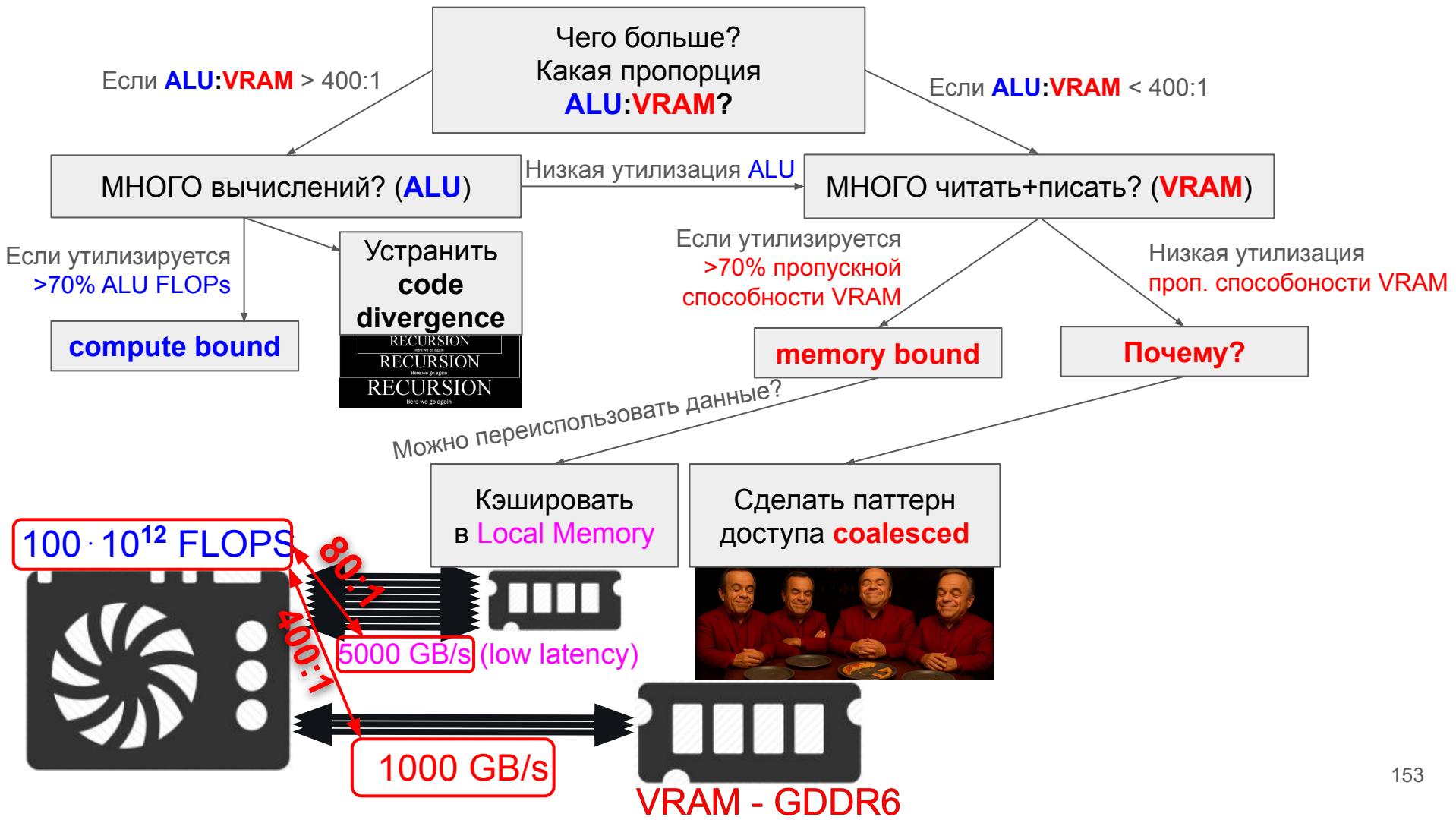


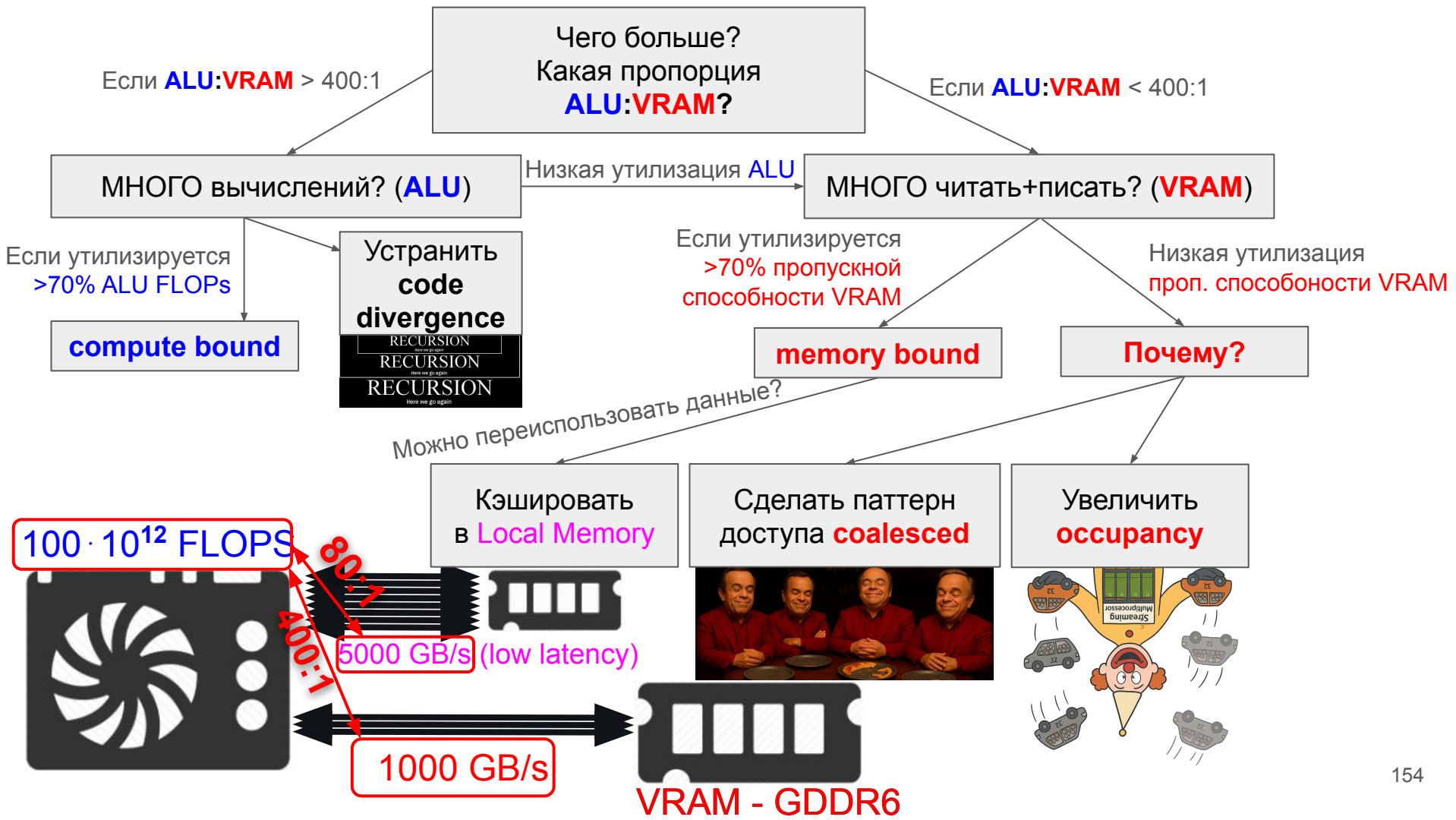
**Можно ли что-то сделать?**











Чего больше?  
Какая пропорция  
**ALU:VRAM?**

Если **ALU:VRAM** > 400:1

Если **ALU:VRAM** < 400:1

МНОГО вычислений? (**ALU**)

Низкая утилизация **ALU**

МНОГО читать+писать? (**VRAM**)

Если утилизируется  
>70% ALU FLOPs

Устранить  
**code divergence**

Если утилизируется  
>70% пропускной  
способности VRAM

**compute bound**

Низкая утилизация  
проп. способности VRAM

**memory bound**

**Почему?**

Можно переиспользовать данные?

Кэшировать  
в Local Memory

Сделать паттерн  
доступа **coalesced**

Увеличить  
**occupancy**

Устранить  
**code divergence**

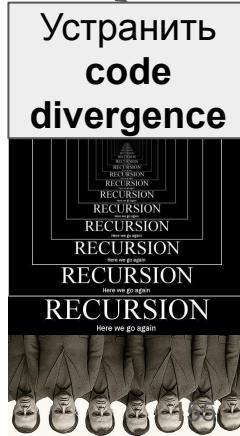
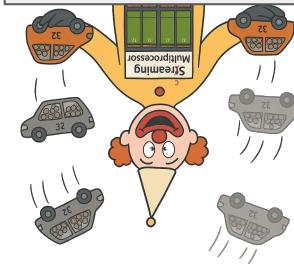
**100 · 10<sup>12</sup> FLOPS**

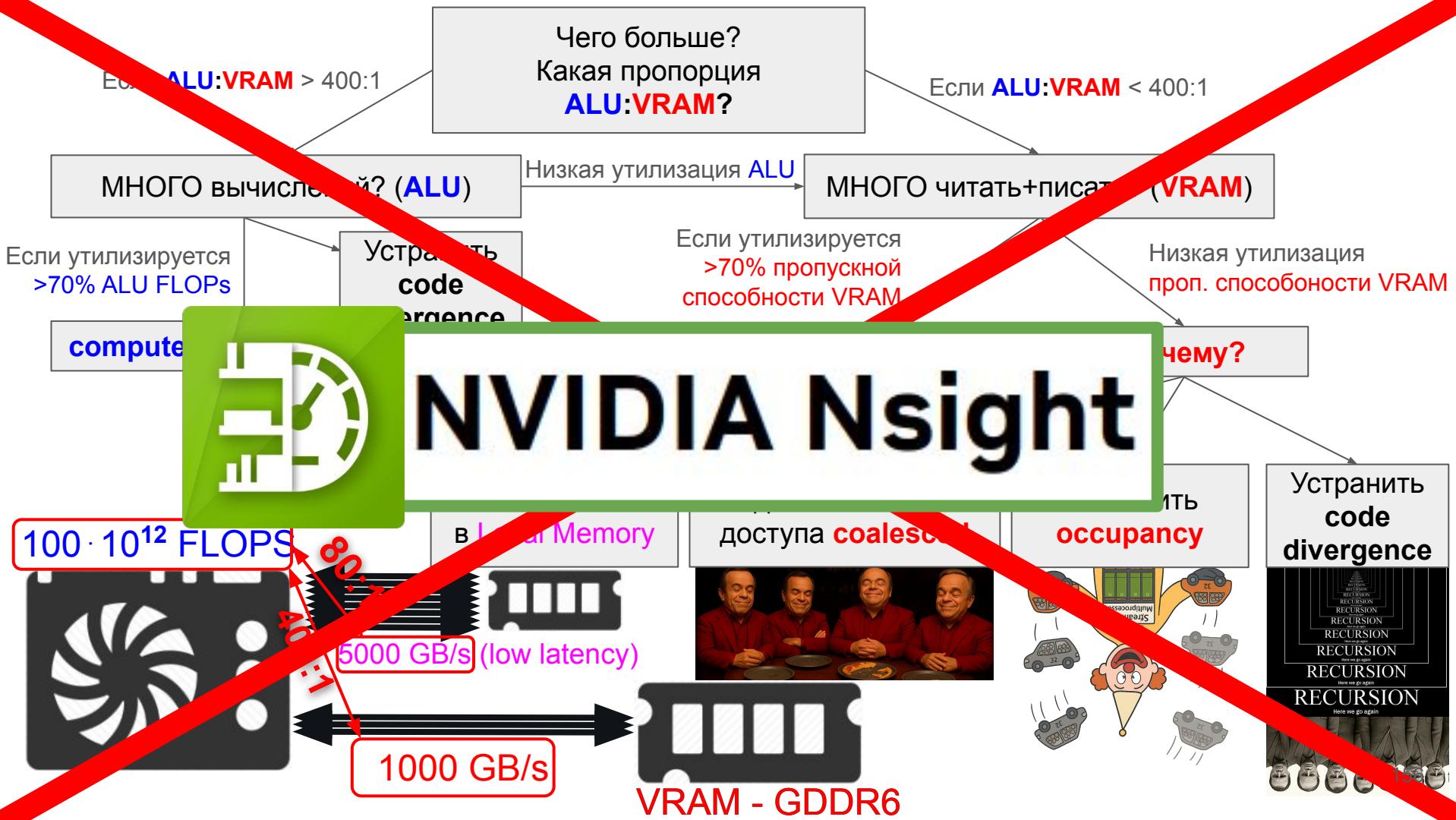
80:1

400:1

5000 GB/s (low latency)

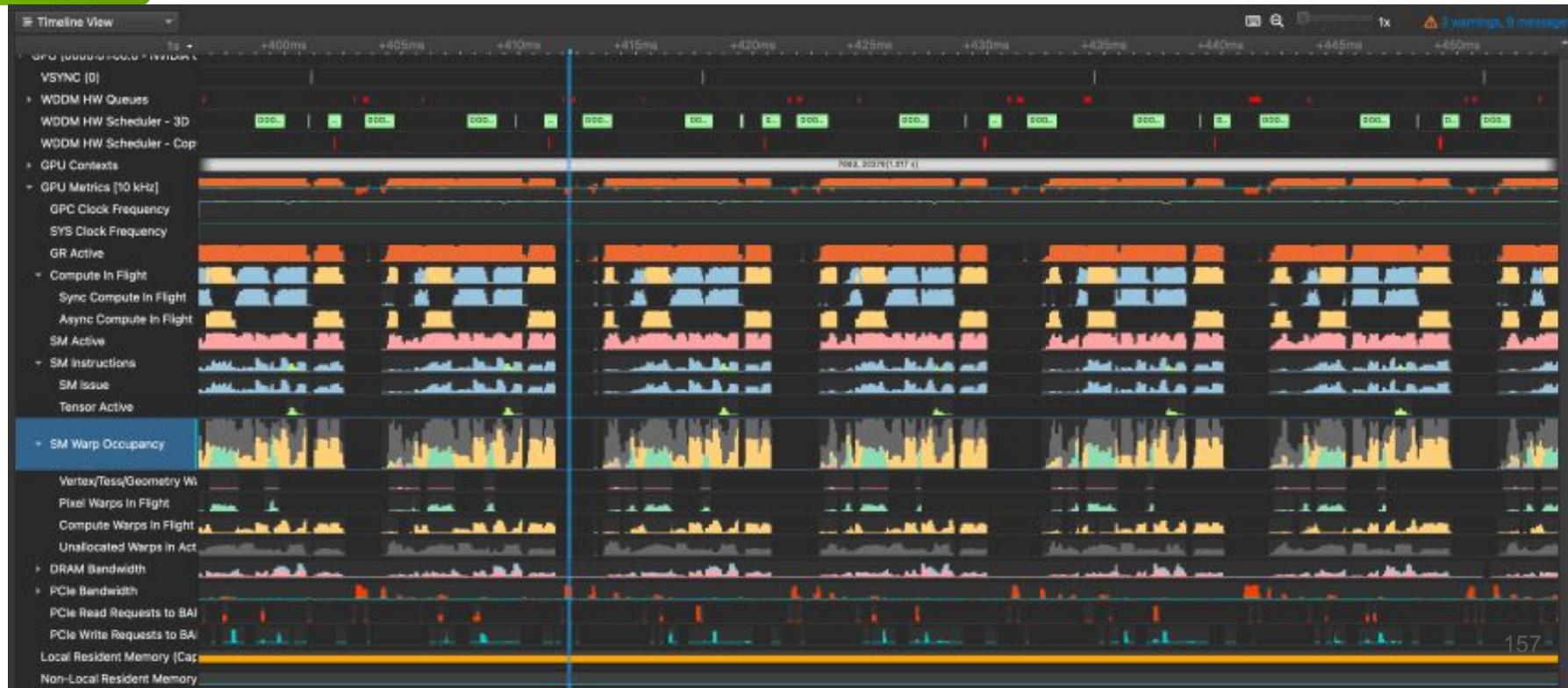
1000 GB/s

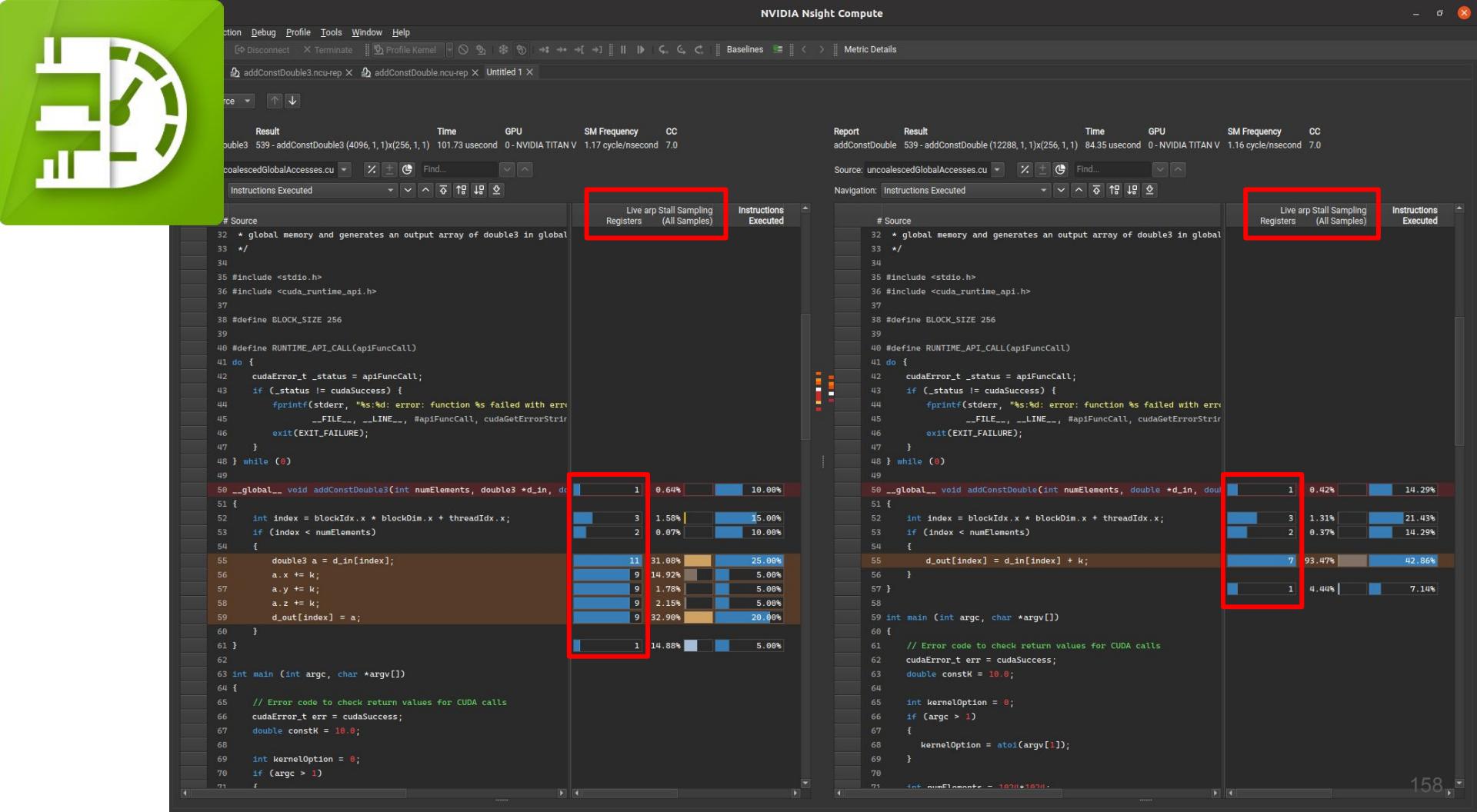


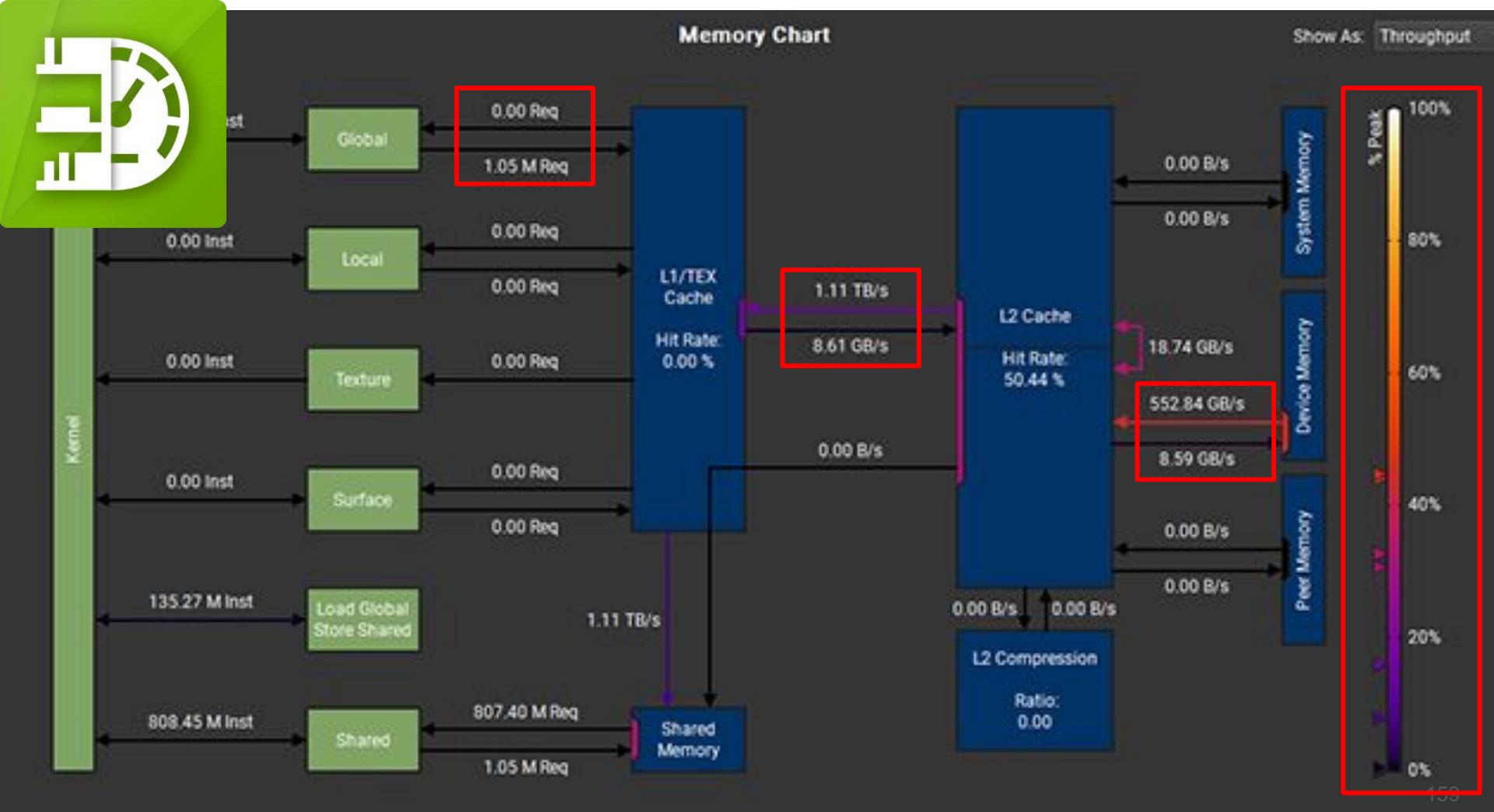


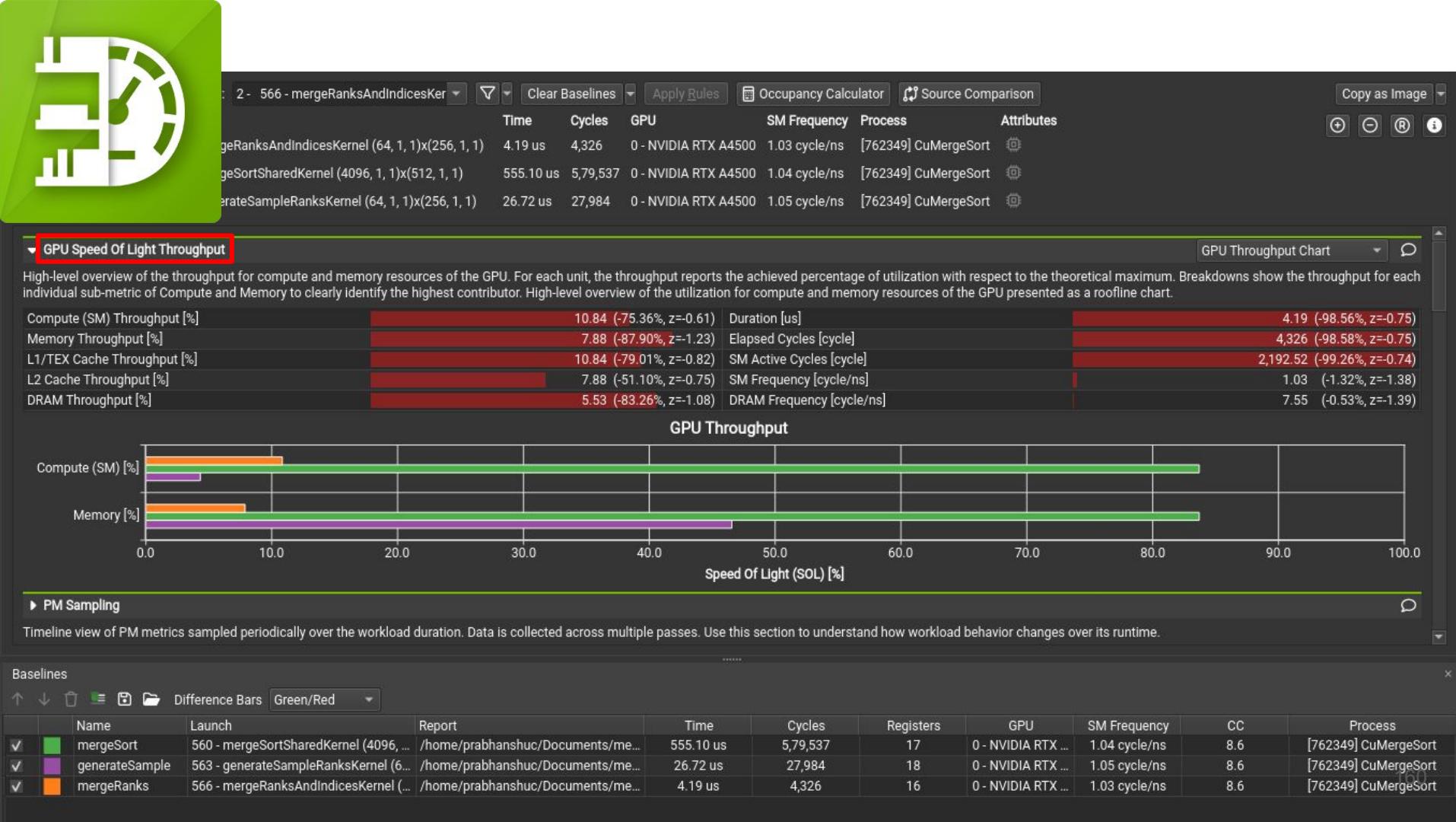


# NVIDIA Nsight









# Глава 6: Как выглядит код

OpenMP, OpenCL, CUDA, Vulkan (GLSL)

Пример: A + B

A [10    34    12    34    54    113 ... 1]

+  
B [32    12    57    12    14    126 ... 5]

||  
C [42    46    69    46    68    239 ... 6]

# Пример: A + B

A  10 34 12 34 54 113 ... 1

+  32 12 57 12 14 126 ... 5

||  42 46 69 46 68 239 ... 6

```
void solveCPU(int[] a, int[] b, int c[], int n) {  
    for (int i = 0; i < n; ++i) {  
        int sum = a[i] + b[i];  
        c[i] = sum;  
    }  
}
```

# Пример: A + B

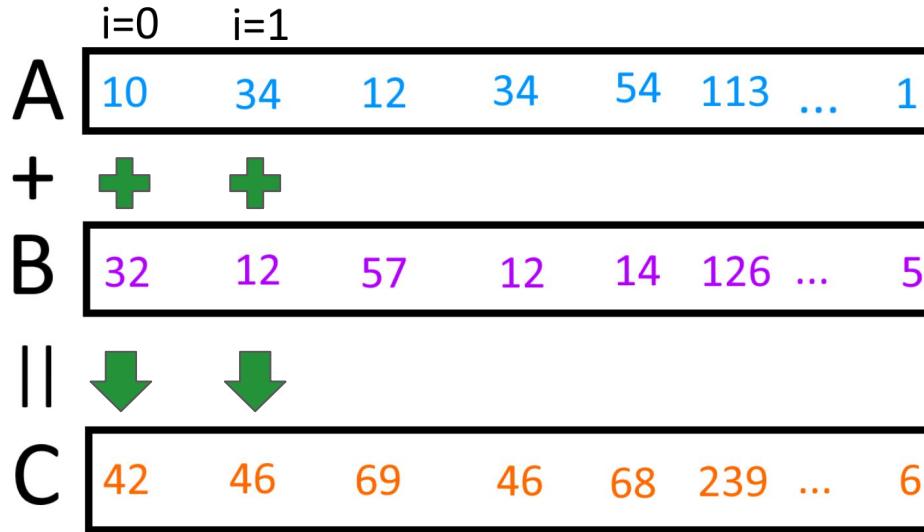
i=0  
A 

+ 

|| 

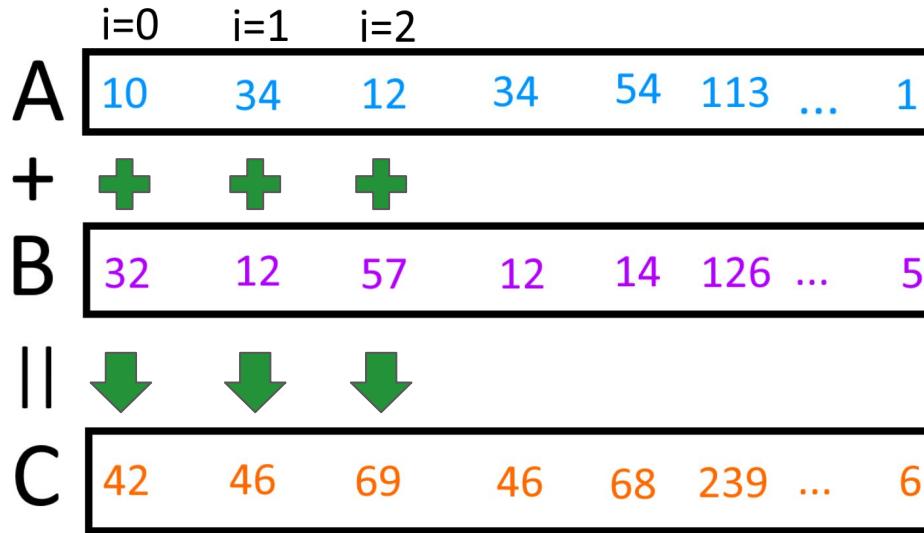
```
void solveCPU(int[] a, int[] b, int c[], int n) {  
    for (int i = 0; i < n; ++i) {  
        int sum = a[i] + b[i];  
        c[i] = sum;  
    }  
}
```

# Пример: A + B



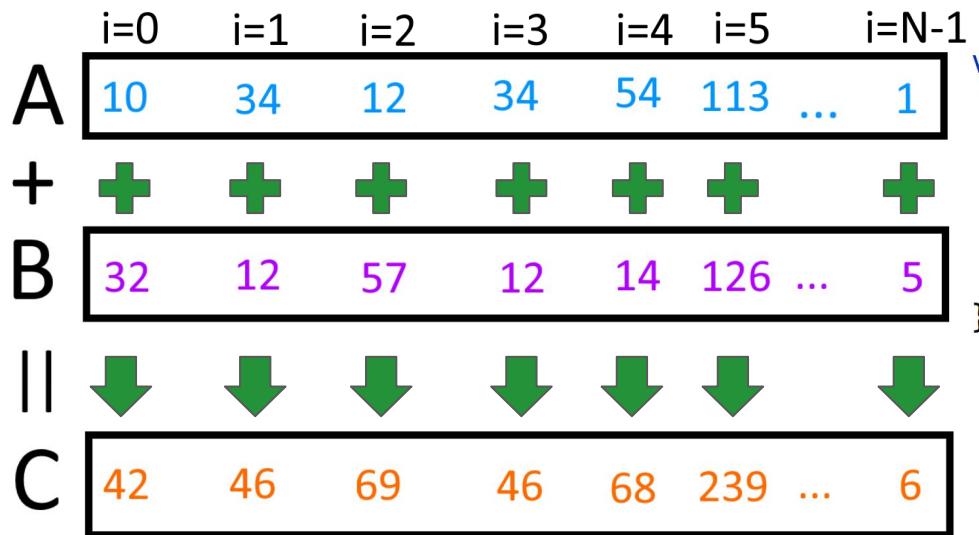
```
void solveCPU(int[] a, int[] b, int c[], int n) {  
    for (int i = 0; i < n; ++i) {  
        int sum = a[i] + b[i];  
        c[i] = sum;  
    }  
}
```

# Пример: A + B



```
void solveCPU(int[] a, int[] b, int c[], int n) {  
    for (int i = 0; i < n; ++i) {  
        int sum = a[i] + b[i];  
        c[i] = sum;  
    }  
}
```

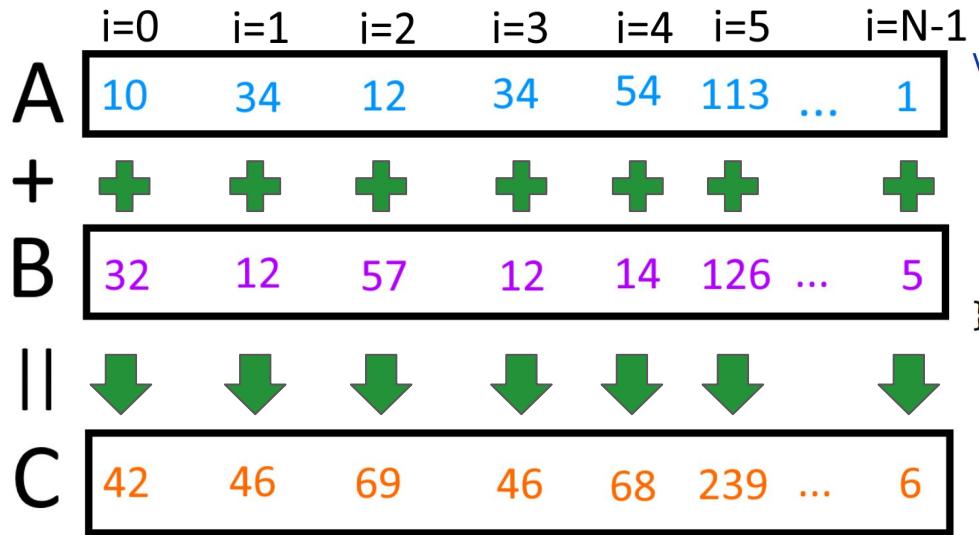
# Пример: A + B



```
void solveCPU(int[] a, int[] b, int c[], int n) {  
    for (int i = 0; i < n; ++i) {  
        int sum = a[i] + b[i];  
        c[i] = sum;  
    }  
}
```

Какая  
асимптотика?

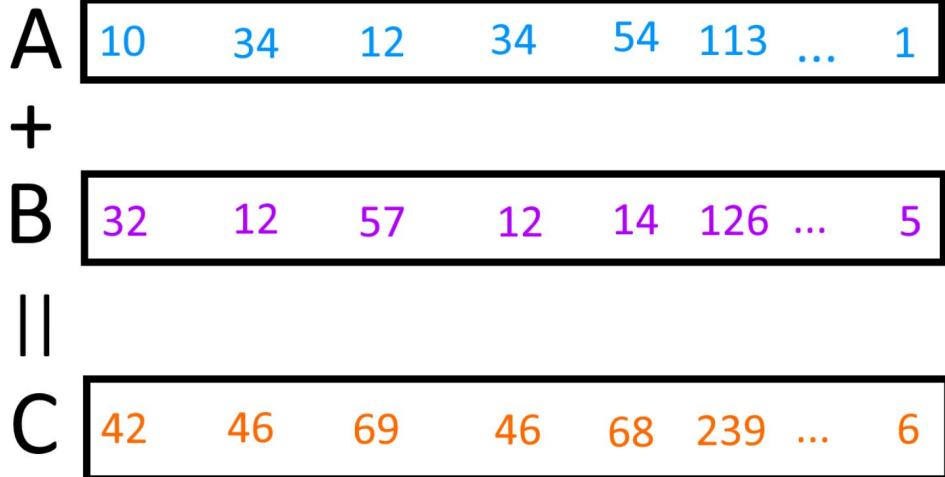
# Пример: A + B



```
void solveCPU(int[] a, int[] b, int c[], int n) {  
    for (int i = 0; i < n; ++i) {  
        int sum = a[i] + b[i];  
        c[i] = sum;  
    }  
}
```

**O(N)**

# Пример: A + B



**NVIDIA RTX 4090 - 16 тысяч ядер!**



# Пример: A + B

A [10 34 12 34 54 113 ... 1]

+ [32 12 57 12 14 126 ... 5]

|| [42 46 69 46 68 239 ... 6]

~~void solveCPU(int[] a, int[] b, int c[], int n) {  
 for (int i = 0; i < n; ++i) {  
 int sum = a[i] + b[i];  
 c[i] = sum;  
 }  
}~~

void solveGPU(int[] a, int[] b, int c[], int n) {  
 const int i = get\_global\_id(0);  
 c[i] = b[i] + a[i];**Супер многопоточно!**  
}



NVIDIA RTX 4090 - 16 тысяч ядер!



# Пример: A + B

A [10 34 12 34 54 113 ... 1]

+ [32 12 57 12 14 126 ... 5]

|| [42 46 69 46 68 239 ... 6]

~~void solveCPU(int[] a, int[] b, int c[], int n) {  
 for (int i = 0; i < n; ++i) {  
 int sum = a[i] + b[i];  
 c[i] = sum;  
 }  
}~~

~~void solveGPU(int[] a, int[] b, int c[], int n) {  
 const int i = get\_global\_id(0);  
 c[i] = b[i] + a[i];  
}~~

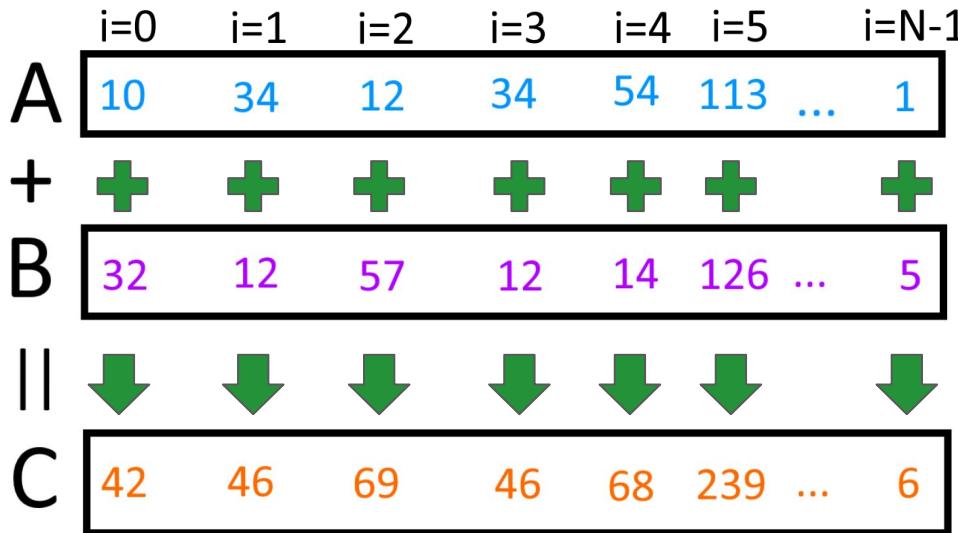
O(N)  
**Какая асимптотика?**



NVIDIA RTX 4090 - 16 тысяч ядер!



# Пример: A + B



~~void solveCPU(int[] a, int[] b, int c[], int n) {  
 for (int i = 0; i < n; ++i) {  
 int sum = a[i] + b[i];  
 c[i] = sum;  
 }  
}~~ O(N)

void solveGPU(int[] a, int[] b, int c[], int n) {  
 const int i = get\_global\_id(0);  
 c[i] = b[i] + a[i];  
}

O(N / 16384)



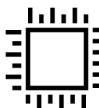
NVIDIA RTX 4090 - 16 тысяч ядер!



## Пример: A + B (N=100.000.000)

```
for (size_t i = 0; i < n; ++i) {  
    cs[i] = as[i] + bs[i];  
}
```

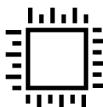
# Пример: A + B (N=100.000.000)



**CPU - Intel 13700K**

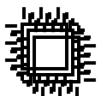
```
for (size_t i = 0; i < n; ++i) {
    cs[i] = as[i] + bs[i];
}
a + b median time: 0.068 sec (+-0.00481664)
a + b median RAM bandwidth: 16.4351 GB/s
```

# Пример: A + B (N=100.000.000)



**CPU - Intel 13700K**

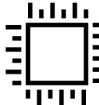
```
for (size_t i = 0; i < n; ++i) {
    cs[i] = as[i] + bs[i];
}
a + b median time: 0.068 sec (+-0.00481664)
a + b median RAM bandwidth: 16.4351 GB/s
```



**CPU - Intel 13700K**

```
#pragma omp parallel for
for (ptrdiff_t i = 0; i < n; ++i) {
    cs[i] = as[i] + bs[i];
}
a + b median time: 0.047 sec (+-0.00153623)
a + b median RAM bandwidth: 23.7784 GB/s
```

# Пример: A + B (N=100.000.000)



**CPU - Intel 13700K**

```
for (size_t i = 0; i < n; ++i) {  
    cs[i] = as[i] + bs[i];  
}  
a + b median time: 0.068 sec (+-0.00481664)  
a + b median RAM bandwidth: 16.4351 GB/s
```



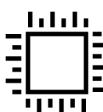
**CPU - Intel 13700K**

```
#pragma omp parallel for  
for (ptrdiff_t i = 0; i < n; ++i) {  
    cs[i] = as[i] + bs[i];  
}  
a + b median time: 0.047 sec (+-0.00153623)  
a + b median RAM bandwidth: 23.7784 GB/s
```

**Почему разница такая  
незначительная?**

**Как проверить гипотезу?  
Как спровоцировать  
значительную разницу?**

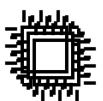
Пример: **100\*cos(A + B)** (N=100.000.000)



### CPU - Intel 13700K

```
for (size_t i = 0; i < n; ++i) {  
    cs[i] = 100.0*cos( Left: as[i] + bs[i]);  
}
```

median time: 1.178 sec (+-0.0235009)  
median RAM bandwidth: 0.948716 GB/s



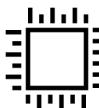
### CPU - Intel 13700K

```
#pragma omp parallel for  
for (ptrdiff_t i = 0; i < n; ++i) {  
    cs[i] = 100*cos( Left: as[i] + bs[i]);  
}
```

median time: 0.109 sec (+-0.0121980)  
median RAM bandwidth: 10.2531 GB/s

x10.9

# Пример: A + B (N=100.000.000)



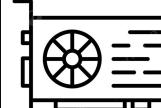
## CPU - Intel 13700K

```
for (size_t i = 0; i < n; ++i) {  
    cs[i] = as[i] + bs[i];  
}  
  
a + b median time: 0.068 sec (+-0.00481664)  
a + b median RAM bandwidth: 16.4351 GB/s
```



## CPU - Intel 13700K

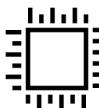
```
#pragma omp parallel for  
  
for (ptrdiff_t i = 0; i < n; ++i) {  
    cs[i] = as[i] + bs[i];  
}  
  
a + b median time: 0.047 sec (+-0.00153623)  
a + b median RAM bandwidth: 23.7784 GB/s
```



## GPU - NVIDIA RTX 4090

```
--kernel void aplusb(__global const float* a,  
                      __global const float* b,  
                      __global      float* c,  
                      unsigned int n)  
  
{  
    const unsigned int index = get_global_id(dimindx: 0);  
    if (index >= n)  
        return;  
  
    c[index] = a[index] + b[index];  
}  
  
a + b median time: 0.002 sec (+-0.000632456)  
a + b median VRAM bandwidth: 558.794 GB/s
```

# Пример: A + B (N=100.000.000)



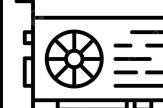
**CPU - Intel 13700K**

```
for (size_t i = 0; i < n; ++i) {  
    cs[i] = as[i] + bs[i];  
}  
a + b median time: 0.068 sec (+-0.00481664)  
a + b median RAM bandwidth: 16.4351 GB/s
```



**CPU - Intel 13700K**

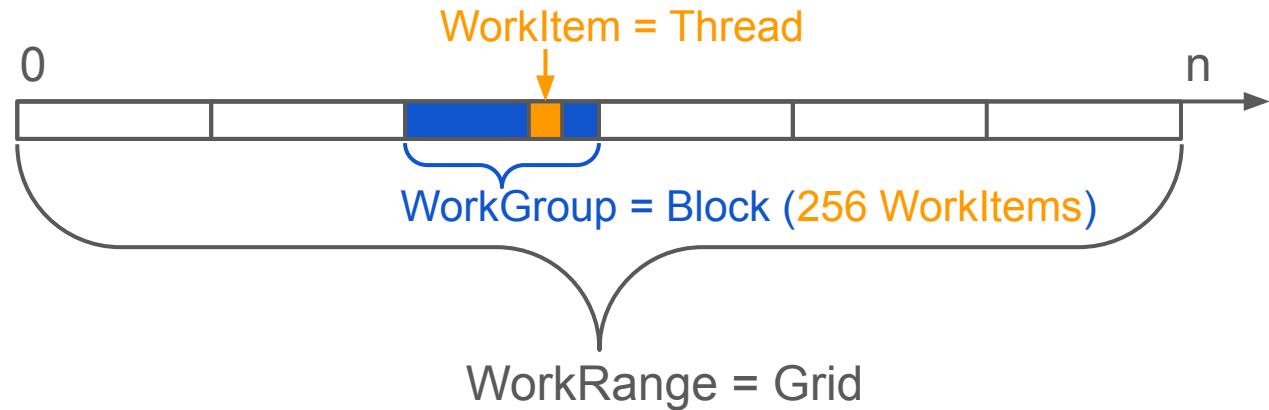
```
#pragma omp parallel for  
for (ptrdiff_t i = 0; i < n; ++i) {  
    cs[i] = as[i] + bs[i];  
}  
a + b median time: 0.047 sec (+-0.00153623)  
a + b median RAM bandwidth: 23.7784 GB/s
```



**GPU - NVIDIA RTX 4090**

```
--kernel void aplusb(__global const float* a,  
                      __global const float* b,  
                      __global      float* c,  
                      unsigned int n)  
{  
    const unsigned int index = get_global_id(dimindx: 0);  
    if (index >= n)  
        return;  
  
    c[index] = a[index] + b[index];  
}  
x23.5  
a + b median time: 0.002 sec (+-0.000632456)  
a + b median VRAM bandwidth: 558.794 GB/s
```

# Пример: A + B (N=100.000.000)

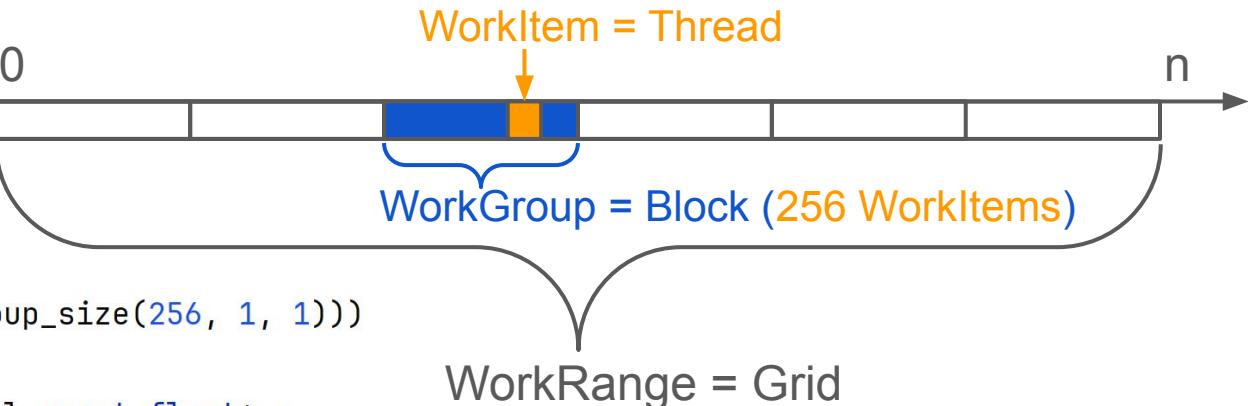


```
1 #ifdef __CLION_IDE__  
2 #include <libgpu/opencl/cl/clionDefines.cl> Приимер: A + B (N=100.000.000)  
3 #endif
```

```
4  
5 #line 5
```

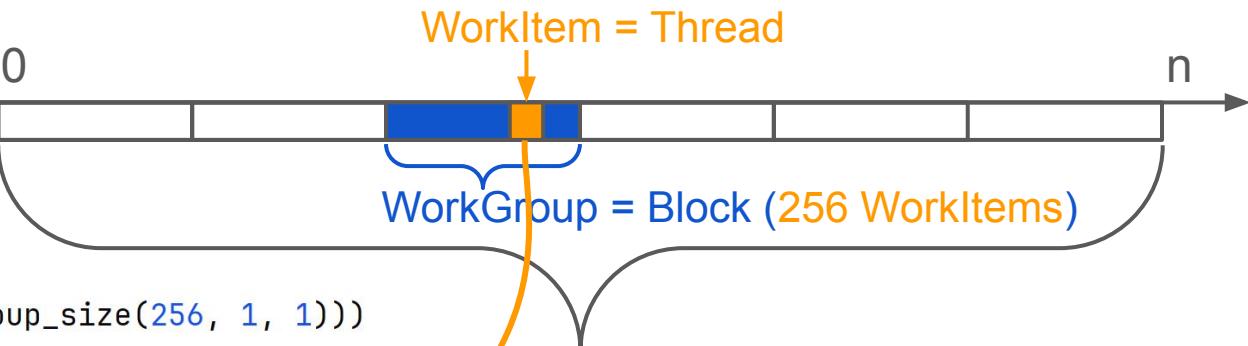


```
6  
7  
8  
9  
10 __attribute__((reqd_work_group_size(256, 1, 1)))  
11  
12 __kernel void aplusb(__global const float* a,  
13                         __global const float* b,  
14                         __global      float* c,  
15                         unsigned int n)  
16 {  
17     const unsigned int index = get_global_id(dimindx: 0);  
18     if (index >= n)  
19         return;  
20  
21     c[index] = a[index] + b[index];  
22 }
```



```
1 #ifdef __CLION_IDE__  
2 #include <libgpu/opencl/cl/clionDefines.cl> Приимер: A + B (N=100.000.000)  
3 #endif
```

```
4  
5 #line 5
```



```
10 __attribute__((reqd_work_group_size(256, 1, 1)))  
11  
12 __kernel void aplusb(__global const float* a,  
13                      __global const float* b,  
14                      __global      float* c,  
15                      unsigned int n)  
16 {  
17     const unsigned int index = get_global_id(dimindx: 0);  
18     if (index >= n)  
19         return;  
20  
21     c[index] = a[index] + b[index];  
22 }
```

```
1 #ifdef __CLION_IDE__  
2 #include <libgpu/opencl/cl/clionDefines.cl> Приимер: A + B (N=100.000.000)  
3 #endif
```

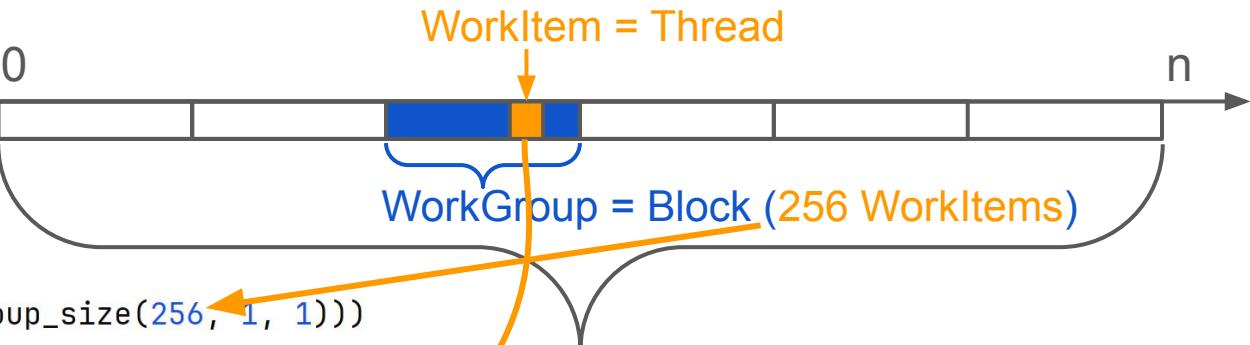
```
4  
5 #line 5
```



```
6  
7  
8  
9  
10 __attribute__((reqd_work_group_size(256, 1, 1)))
```

```
11  
12 __kernel void aplusb(__global const float* a,  
13  
14             __global const float* b,  
15             __global      float* c,  
16             unsigned int n)
```

```
17 {  
18     const unsigned int index = get_global_id(dimindx: 0);  
19     if (index >= n)  
20         return;  
21  
22     c[index] = a[index] + b[index];  
23 }
```



```
1 #ifdef __CLION_IDE__  
2 #include <libgpu/opencl/cl/clionDefines.cl> Приимер: A + B (N=100.000.000)  
3 #endif
```

```
4  
5 #line 5
```

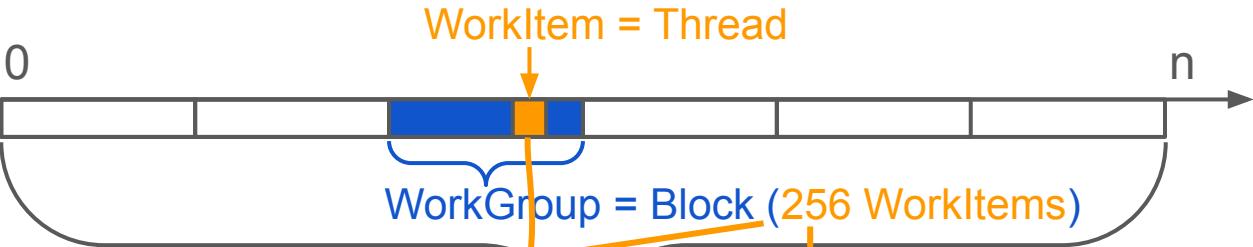


```
6  
7  
8  
9  
10 __attribute__((reqd_work_group_size(256, 1, 1)))
```

```
11  
12 __kernel void aplusb(__global const float* a,  
13  
14             __global const float* b,  
15             __global      float* c,  
16             unsigned int n)
```

```
17 {  
18     const unsigned int index = get_global_id(dimindx: 0);  
19     if (index >= n)  
20         return;
```

```
21     c[index] = a[index] + b[index];  
22 }
```



HOST-side (CPU, C++)

```
gpu::WorkSize workSize(GROUP_SIZE, n);
```

```
ocl_aplusb.exec(workSize, a_gpu, b_gpu, c_gpu, n);
```

```
1 #ifdef __CLION_IDE__  
2 #include <libgpu/opencl/cl/clion_defines.cl> Приимер: A + B (N=100.000.000)  
3 #endif
```

```
4  
5 #line 5
```



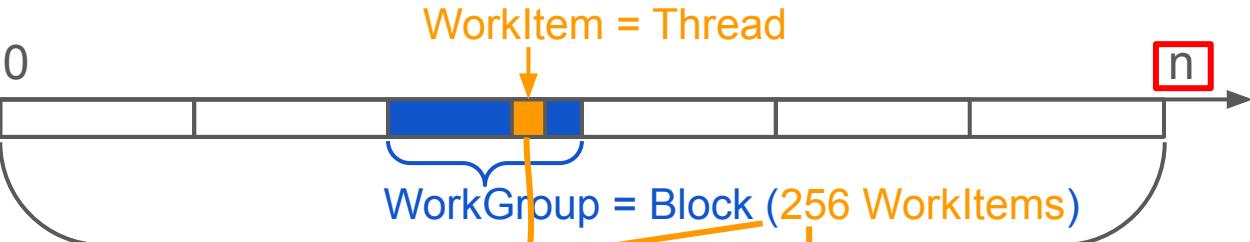
```
6  
7  
8  
9  
10 __attribute__((reqd_work_group_size(256, 1, 1)))
```

```
11  
12 __kernel void aplusb(__global const float* a,  
13  
14             __global const float* b,  
15             __global      float* c,  
16             unsigned int n)
```

```
17 {  
18     const unsigned int index = get_global_id(dimindx: 0);
```

```
19     if (index >= n)  
20         return;
```

```
21  
22     c[index] = a[index] + b[index];
```



А где задается размер рабочего пространства?

HOST-side (CPU, C++)

```
gpu::WorkSize workSize(GROUP_SIZE, n);
```

```
ocl_aplusb.exec(workSize, a_gpu, b_gpu, c_gpu, n);
```

```
1 #ifdef __CLION_IDE__  
2 #include <libgpu/opencl/cl/clionDefines.cl> Приимер: A + B (N=100.000.000)  
3 #endif
```

```
4  
5 #line 5
```



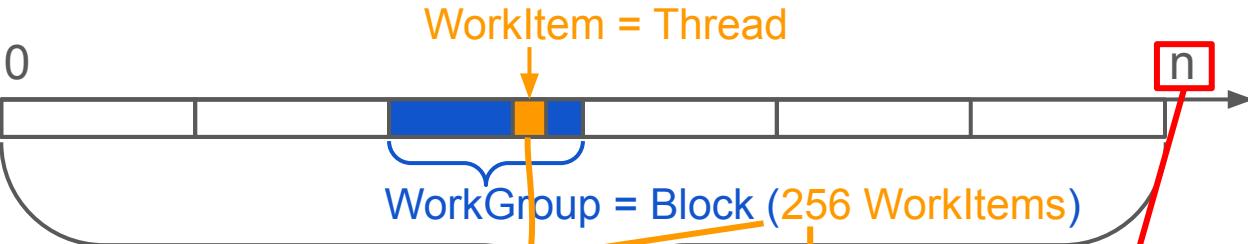
```
10 __attribute__((reqd_work_group_size(256, 1, 1)))
```

```
11 __kernel void aplusb(__global const float* a,  
12                      __global const float* b,  
13                      __global      float* c,  
14                      unsigned int n)
```

```
15 {  
16     const unsigned int index = get_global_id(dimindx: 0);
```

```
17     if (index >= n)  
18         return;
```

```
21     c[index] = a[index] + b[index];  
22 }
```



HOST-side (CPU, C++)

```
gpu::WorkSize workSize(GROUP_SIZE, n);
```

```
ocl_aplusb.exec(workSize, a_gpu, b_gpu, c_gpu, n);
```

```
1 #ifdef __CLION_IDE__  
2 #include <libgpu/opencl/cl/clionDefines.cl> Приимер: A + B (N=100.000.000)  
3 #endif
```

```
4  
5 #line 5
```



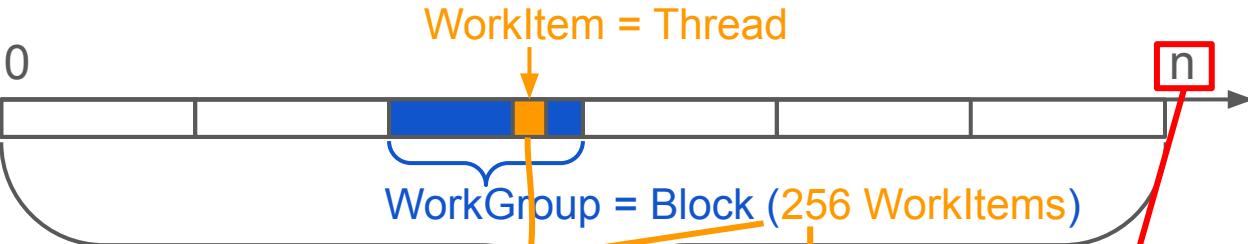
```
10 __attribute__((reqd_work_group_size(256, 1, 1)))
```

```
11 __kernel void aplusb(__global const float* a,  
12                      __global const float* b,  
13                      __global      float* c,  
14                      unsigned int n)
```

```
15 {  
16     const unsigned int index = get_global_id(dimindx: 0);
```

```
17     if (index >= n)  
18         return;
```

```
21     c[index] = a[index] + b[index];
```



А где задается размер рабочего пространства?



HOST-side (CPU, C++)

```
gpu::WorkSize workSize(GROUP_SIZE, n);
```

```
ocl_aplusb.exec(workSize, a_gpu, b_gpu, c_gpu, n);
```

```
1 #ifdef __CLION_IDE__  
2 #include <libgpu/opencl/cl/clionDefines.cl> Приимер: A + B (N=100.000.000)  
3 #endif
```

```
4  
5 #line 5
```



```
10 __attribute__((reqd_work_group_size(256, 1, 1)))
```

```
11 __kernel void aplusb(__global const float* a,
```

```
12           __global const float* b,
```

```
13           __global       float* c,
```

```
14           unsigned int n)
```

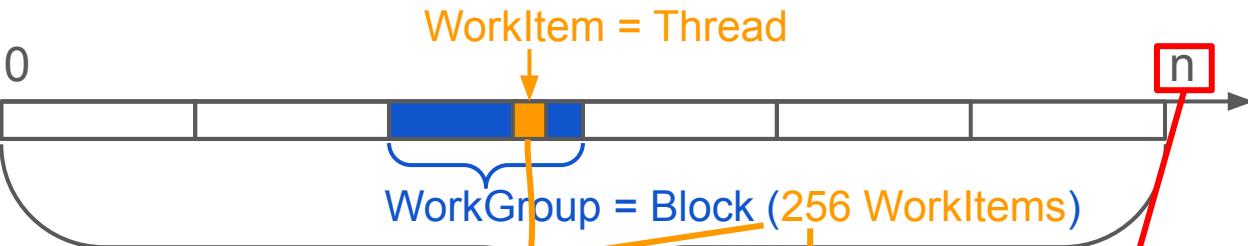
```
15 {
```

```
16     const unsigned int index = get_global_id(dimindx: 0);
```

```
17     if (index >= n)
```

```
18       return;
```

```
21     c[index] = a[index] + b[index];
```



А где задается размер рабочего пространства?

HOST-side (CPU, C++)

```
gpu::WorkSize workSize(GROUP_SIZE, n);
```

```
ocl_aplusb.exec(workSize, a_gpu, b_gpu, c_gpu, n);
```



```
1 #ifdef __CLION_IDE__
```

```
2 #include <libgpu/opencl/cl/clion_defines.cl>
```

```
3 #endif
```

```
4
```

```
5 #line 5
```



```
10 __attribute__((reqd_work_group_size(256, 1, 1)))
```

```
11
```

```
12 __kernel void aplusb(__global const float* a,
```

```
13           __global const float* b,
```

```
14           __global      float* c,
```

```
15           unsigned int n)
```

```
16 {
```

```
17     const unsigned int index = get_global_id(dimindx: 0)
```

```
18     if (index >= n)
```

```
19         return;
```

```
20
```

```
21     c[index] = a[index] + b[index];
```

```
22 }
```

# Пример: A + B (N=100.000.000)



```
blockIdx.x * blockDim.x + threadIdx.x;
```

```
1 __global__ void aplusb(const float* a,
```

```
2           const float* b,
```

```
3           float* c,
```

```
4           unsigned int n)
```

```
5 {
```

```
6     const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
```

```
7     if (index >= n)
```

```
8         return;
```

```
9
```

```
10    c[index] = a[index] + b[index];
```

```
11 }
```

189

# Пример: A + B (N=100.000.000)



```
1  __global__ void aplusb(const float* a,
2                          const float* b,
3                          float* c,
4                          unsigned int n)
5  {
6      const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
7      if (index >= n)          WorkGroup = Block (256 WorkItems)
8          return;
9
10     c[index] = a[index] + b[index];
11 }
```

## HOST-side (CPU, C++)

## Пример: A + B (N=100.000.000)

```
12
13 void cuda_aplusb(const gpu::WorkSize &workSize,
14                     const float* a, const float* b, float* c, unsigned int n,
15                     cudaStream_t stream)
16 {
17     const unsigned int blockSize = 256;
18     const unsigned int gridSize = (n + blockSize - 1) / blockSize;
19     aplusb<<<gridSize, blockSize, 0, stream>>>(a, b, c, n);
20     CUDA_CHECK_KERNEL(stream);
21 }
```



```
1 __global__ void aplusb(const float* a,
2                         const float* b,
3                         float* c,
4                         unsigned int n)
5 {
6     const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
7     if (index >= n)          WorkGroup = Block (256 WorkItems)
8         return;
9
10    c[index] = a[index] + b[index];
11 }
```

## HOST-side (CPU, C++)

## Пример: A + B (N=100.000.000)

```
12
13 void cuda_aplusb(const gpu::WorkSize &workSize,
14                     const float* a, const float* b, float* c, unsigned int n,
15                     cudaStream_t stream)
16 {
17     const unsigned int blockSize = 256;
18     const unsigned int gridSize = (n + blockSize - 1) / blockSize;
19     aplusb<<<gridSize, blockSize, 0, stream>>>(a, b, c, n);
20     CUDA_CHECK_KERNEL(stream);
21 }
```



А где задается  
размер рабочего  
пространства?

```
1 __global__ void aplusb(const float* a,
2                         const float* b,
3                         float* c,
4                         unsigned int n)
5 {
6     const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
7     if (index >= n)          WorkGroup = Block (256 WorkItems)
8         return;
9
10    c[index] = a[index] + b[index];
11 }
```

## HOST-side (CPU, C++)

## Пример: A + B (N=100.000.000)

```
12
13 void cuda_aplusb(const gpu::WorkSize &workSize,
14                     const float* a, const float* b, float* c, unsigned int n,
15                     cudaStream_t stream)
16 {
17     const unsigned int blockSize = 256;
18     const unsigned int gridSize = (n + blockSize - 1) / blockSize;
19     aplusb<< gridSize blockSize 0, stream>>>(a, b, c, n);
20     CUDA_CHECK_KERNEL(stream);
21 }
```



А где задается  
размер рабочего  
пространства?

```
1 __global__ void aplusb(const float* a,
2                         const float* b,
3                         float* c,
4                         unsigned int n)
5 {
6     const unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
7     if (index >= n)          WorkGroup = Block (256 WorkItems)
8         return;
9
10    c[index] = a[index] + b[index];
11 }
```

```
1 #version 450
2
3 #include <libgpu/vulkan/vk/common.vk>
4
5 layout (std430, binding = 0) readonly buffer AsIn { uint as[]; };
6 layout (std430, binding = 1) readonly buffer BsIn { uint bs[]; };
7 layout (std430, binding = 2) writeonly buffer CsOut { uint cs[]; };
8
9 layout (push_constant) uniform PushConstants {
10     uint n;
11 } params;
12
13 layout (local_size_x = 256) in;
14
15 void main()
16 {
17     const uint index = gl_GlobalInvocationID.x;
18     if (index >= params.n)
19         return;
20
21     cs[index] = as[index] + bs[index];
22 }
```

# Пример: A + B (N=100.000.000)

## GLSL (Graphics Library Shading Language)



```
__kernel void aplusb_matrix(__global const uint* a,
                           __global const uint* b,
                           __global      uint* c,
                           unsigned int width,
                           unsigned int height)
```

```
{
```

```
    const unsigned int index = get_global_id(0);
```

```
    for (int k = 0; k < width; ++k) {
        unsigned int col = k;
        unsigned int row = index;
        c[row * width + col] = a[row * width + col]
                               + b[row * width + col];
    }
}
```

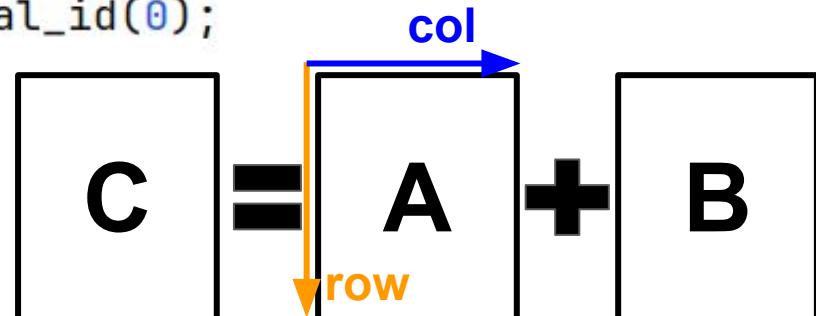
$$\boxed{\mathbf{C}} = \boxed{\mathbf{A}} + \boxed{\mathbf{B}}$$

```
__kernel void aplusb_matrix(__global const uint* a,  
                           __global const uint* b,  
                           __global      uint* c,  
                           unsigned int width,  
                           unsigned int height)
```

```
{
```

```
    const unsigned int index = get_global_id(0);
```

```
    for (int k = 0; k < width; ++k) {  
        unsigned int col = k;  
        unsigned int row = index;  
        c[row * width + col] = a[row * width + col]  
                            + b[row * width + col];  
    }
```

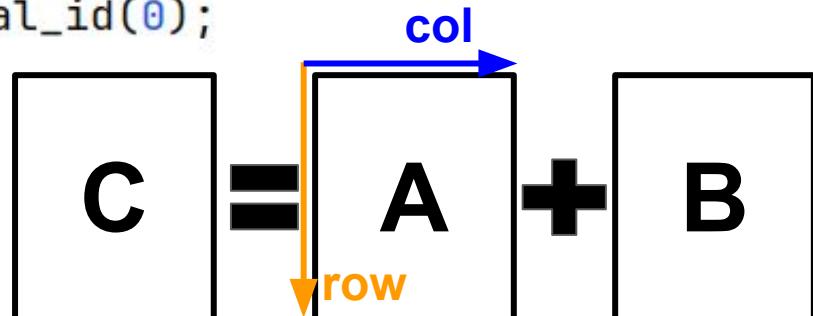


```

__kernel void aplusb_matrix(__global const uint* a,
                           __global const uint* b,
                           __global      uint* c,
                           unsigned int width,
                           unsigned int height)
{
    const unsigned int index = get_global_id(0);

    for (int k = 0; k < width; ++k) {
        unsigned int col = k;
        unsigned int row = index;
        c[row * width + col] = a[row * width + col]
                               + b[row * width + col];
    }
}

```



*coalesced memory access?*

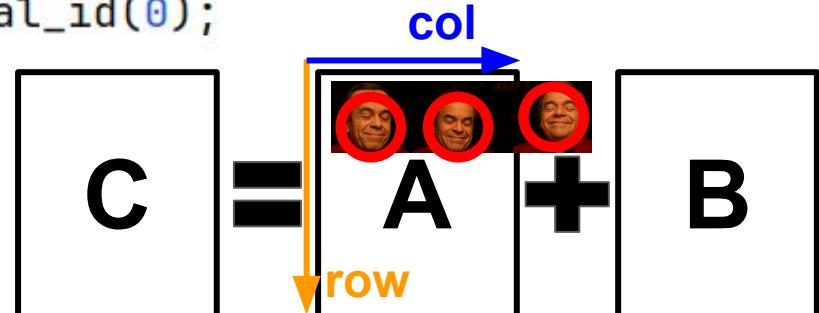
```
__kernel void aplusb_matrix(__global const uint* a,  
                           __global const uint* b,  
                           __global      uint* c,  
                           unsigned int width,  
                           unsigned int height)
```

{

```
const unsigned int index = get_global_id(0);
```

```
for (int k = 0; k < width; ++k) {  
    unsigned int col = k;  
    unsigned int row = index;  
    c[row * width + col] = a[row * width + col]  
        + b[row * width + col];  
}
```

Как выглядит warp?



*coalesced memory access?*

}

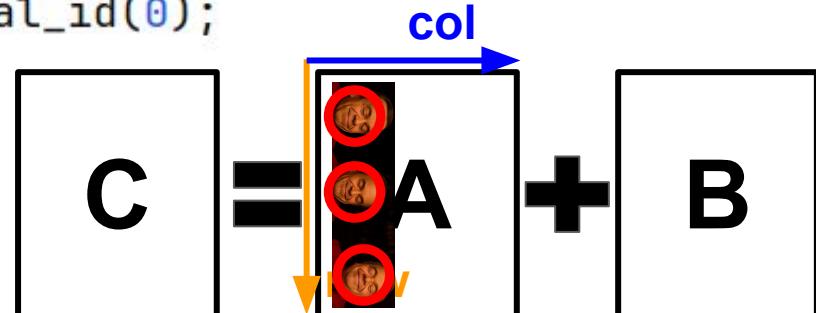
```
__kernel void aplusb_matrix(__global const uint* a,
                           __global const uint* b,
                           __global      uint* c,
                           unsigned int width,
                           unsigned int height)
```

{

```
const unsigned int index = get_global_id(0);
```

```
for (int k = 0; k < width; ++k) {
    unsigned int col = k;
    unsigned int row = index;
    c[row * width + col] = a[row * width + col]
                           + b[row * width + col];
}
```

Как выглядит warp?



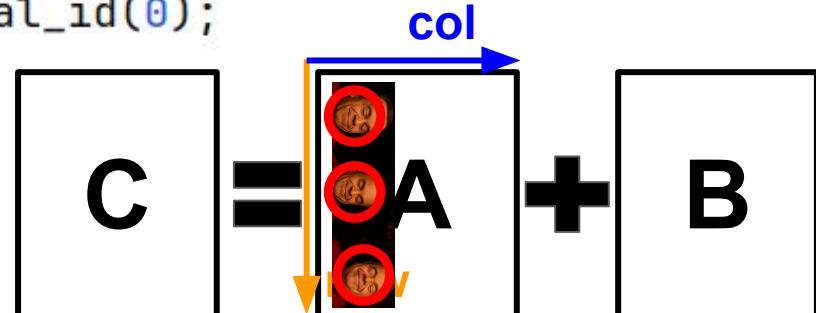
coalesced memory access?

```
__kernel void aplusb_matrix(__global const uint* a,  
                           __global const uint* b,  
                           __global      uint* c,  
                           unsigned int width,  
                           unsigned int height)
```

**HE coalesced memory access!!!**

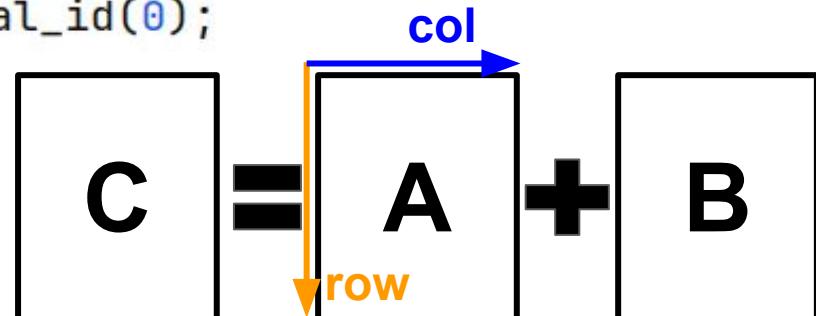
```
const unsigned int index = get_global_id(0);
```

```
for (int k = 0; k < width; ++k) {  
    unsigned int col = k;  
    unsigned int row = index;  
    c[row * width + col] = a[row * width + col]  
        + b[row * width + col];  
}
```



```
__kernel void aplusb_matrix(__global const uint* a,
                           __global const uint* b,
                           __global      uint* c,
                           unsigned int width,
                           unsigned int height)
{
    const unsigned int index = get_global_id(0);

    for (int k = 0; k < height; ++k) {
        unsigned int col = index;
        unsigned int row = k;
        c[row * width + col] = a[row * width + col]
                               + b[row * width + col];
    }
}
```

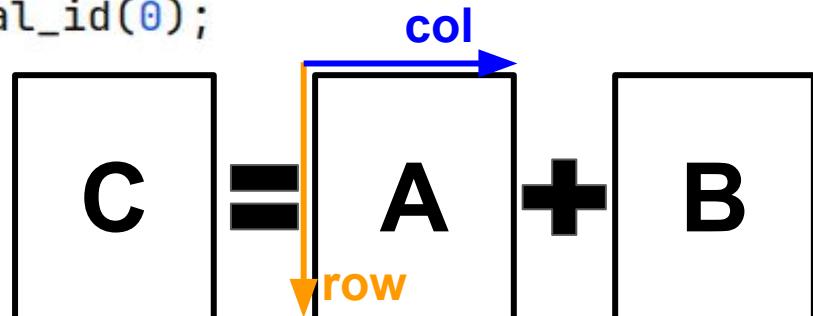


```

__kernel void aplusb_matrix(__global const uint* a,
                           __global const uint* b,
                           __global      uint* c,
                           unsigned int width,
                           unsigned int height)
{
    const unsigned int index = get_global_id(0);

    for (int k = 0; k < height; ++k) {
        unsigned int col = index;
        unsigned int row = k;
        c[row * width + col] = a[row * width + col]
                               + b[row * width + col];
    }
}

```



*coalesced memory access?*

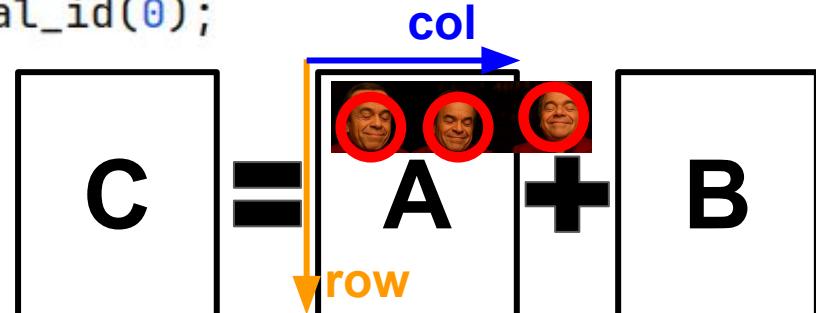
```
__kernel void aplusb_matrix(__global const uint* a,
                           __global const uint* b,
                           __global      uint* c,
                           unsigned int width,
                           unsigned int height)
```

{

```
const unsigned int index = get_global_id(0);
```

```
for (int k = 0; k < height; ++k) {
    unsigned int col = index;
    unsigned int row = k;
    c[row * width + col] = a[row * width + col]
                           + b[row * width + col];
}
```

Как выглядит warp?



*coalesced memory access?*

```
__kernel void aplusb_matrix(__global const uint* a,  
                           __global const uint* b,  
                           __global      uint* c,  
                           unsigned int width,  
                           unsigned int height)
```

{

```
const unsigned int index = get_global_id(0);
```

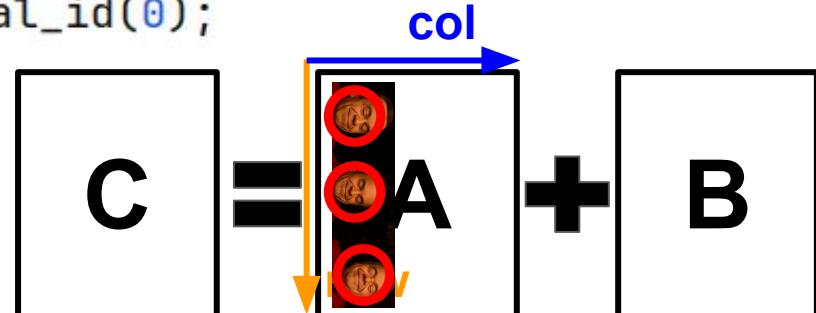
```
for (int k = 0; k < height; ++k) {  
    unsigned int col = index;
```

```
    unsigned int row = k;
```

```
    c[row * width + col] = a[row * width + col]  
                           + b[row * width + col];
```

}

Как выглядит warp?



coalesced memory access?

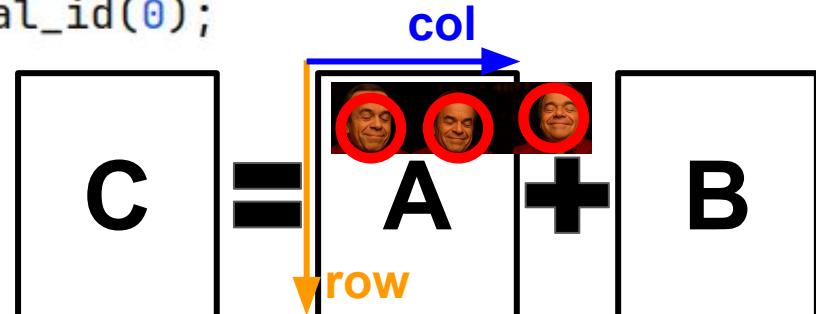
```
__kernel void aplusb_matrix(__global const uint* a,  
                           __global const uint* b,  
                           __global      uint* c,  
                           unsigned int width,  
                           unsigned int height)
```

```
{
```

*coalesced memory access!!!*

```
const unsigned int index = get_global_id(0);
```

```
for (int k = 0; k < height; ++k) {  
    unsigned int col = index;  
    unsigned int row = k;  
    c[row * width + col] = a[row * width + col]  
        + b[row * width + col];  
}
```





Вопросы?