



Вычисления на видеокартах

Лекция 4

- Транспонирование матрицы
- Умножение матриц
- Tensor Cores, WMMA
- Метод Штрассена
- Матрица матрицу
матрицово матрицит
матрицей в матрице

TENSOR CORES

$$A \times B = C$$



polarnick239@gmail.com

Николай Полярный

План лекции

- 1) **Транспонирование матрицы:** через локальную память (учет *bank conflicts*)
- 2) **Умножение матриц:** через локальную память
- 3) **Умножение матриц:** *Tensor Cores, WMMA*
- 4) **Умножение матриц:** оптимизации *DeepSeek*
- 5) **Умножение матриц:** метод *Штрассена*

Глава 1: Транспонирование

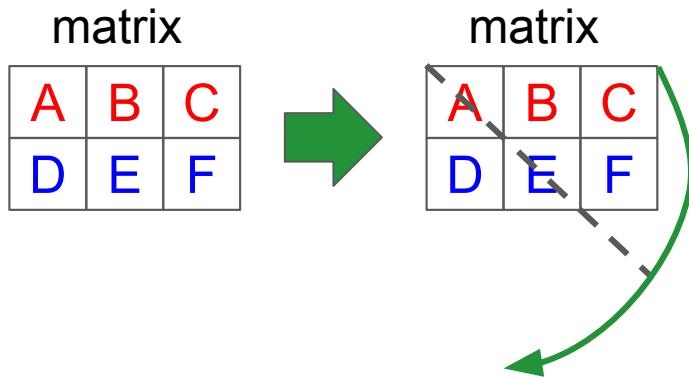
local memory, coalesced access, bank conflicts

Транспонирование матрицы

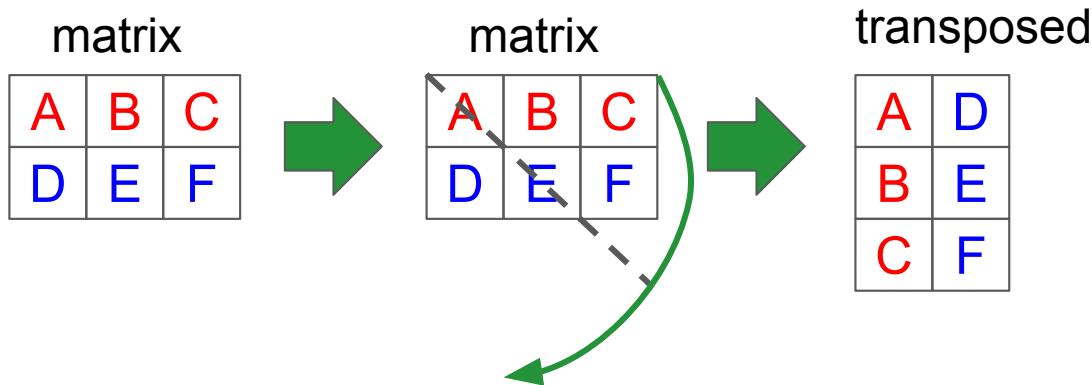
matrix

A	B	C
D	E	F

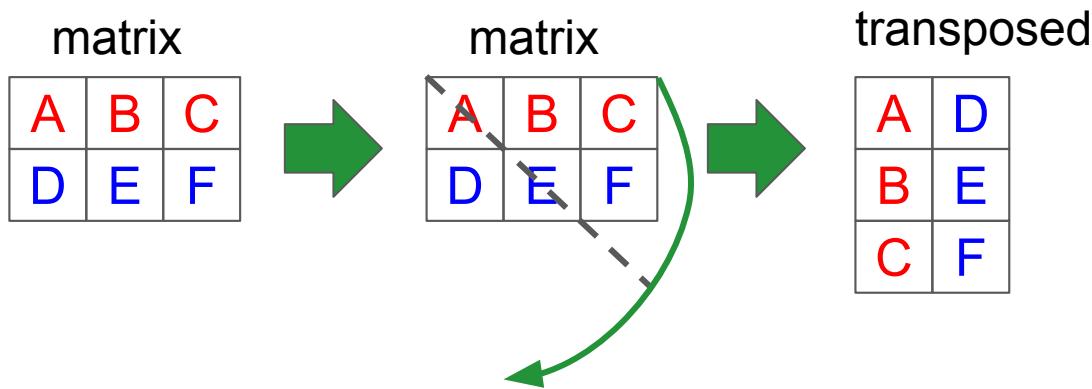
Транспонирование матрицы



Транспонирование матрицы

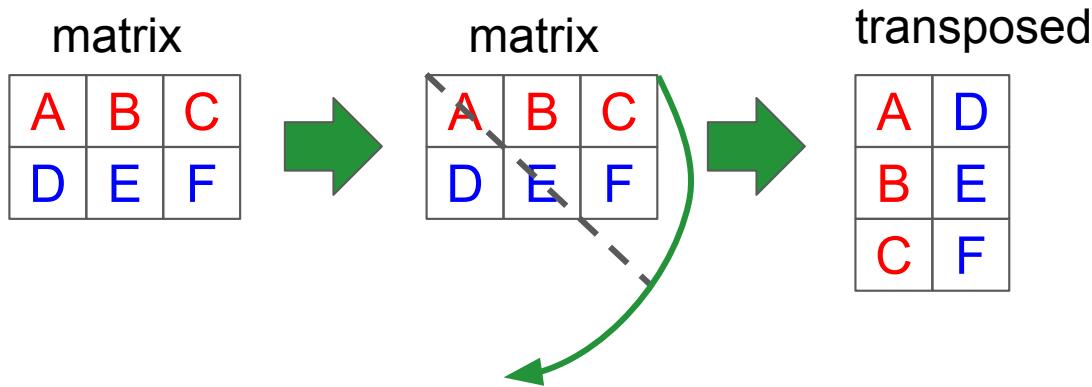


Транспонирование матрицы



Как выглядит:
- WorkRange?

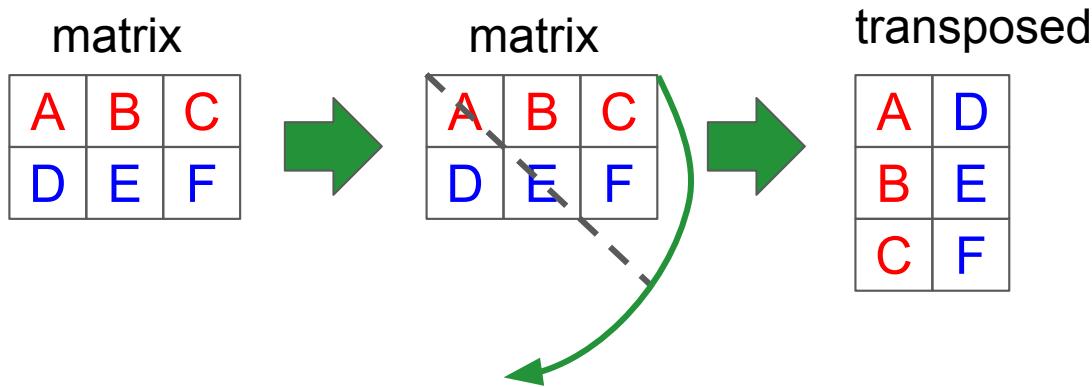
Транспонирование матрицы



Как выглядит:

- WorkRange?
- Задача WorkItem?

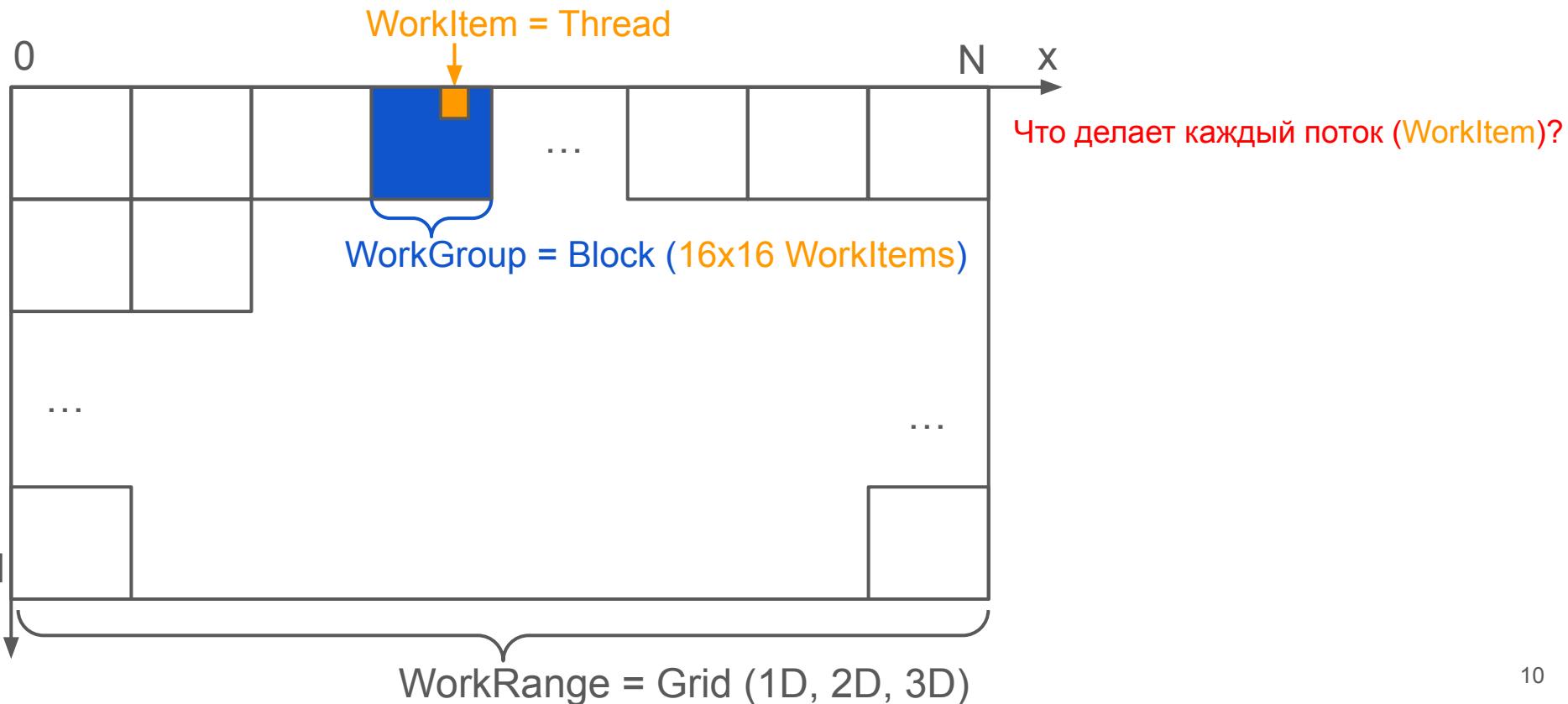
Транспонирование матрицы



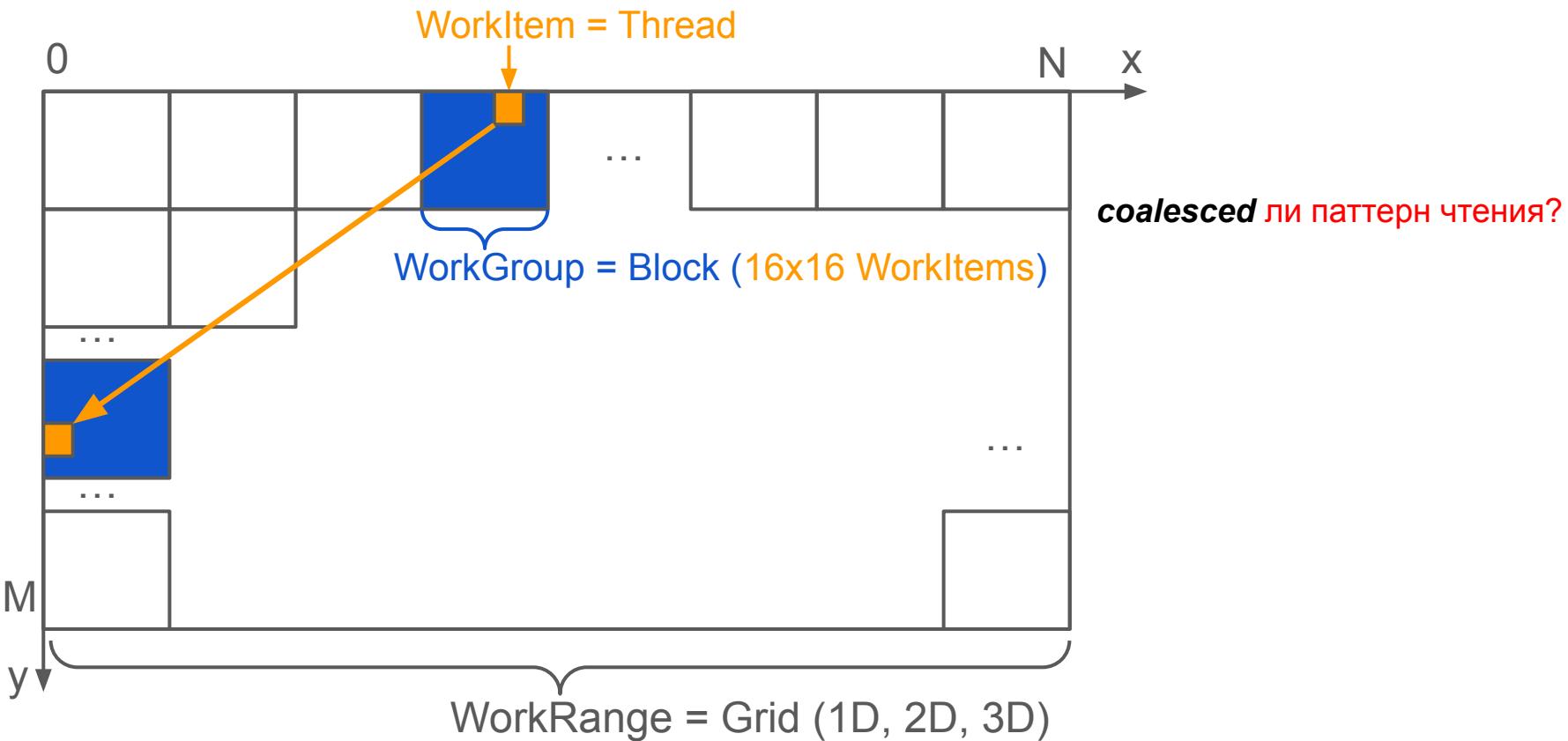
Как выглядит:

- WorkRange?
- Задача WorkItem?
- WorkGroup?

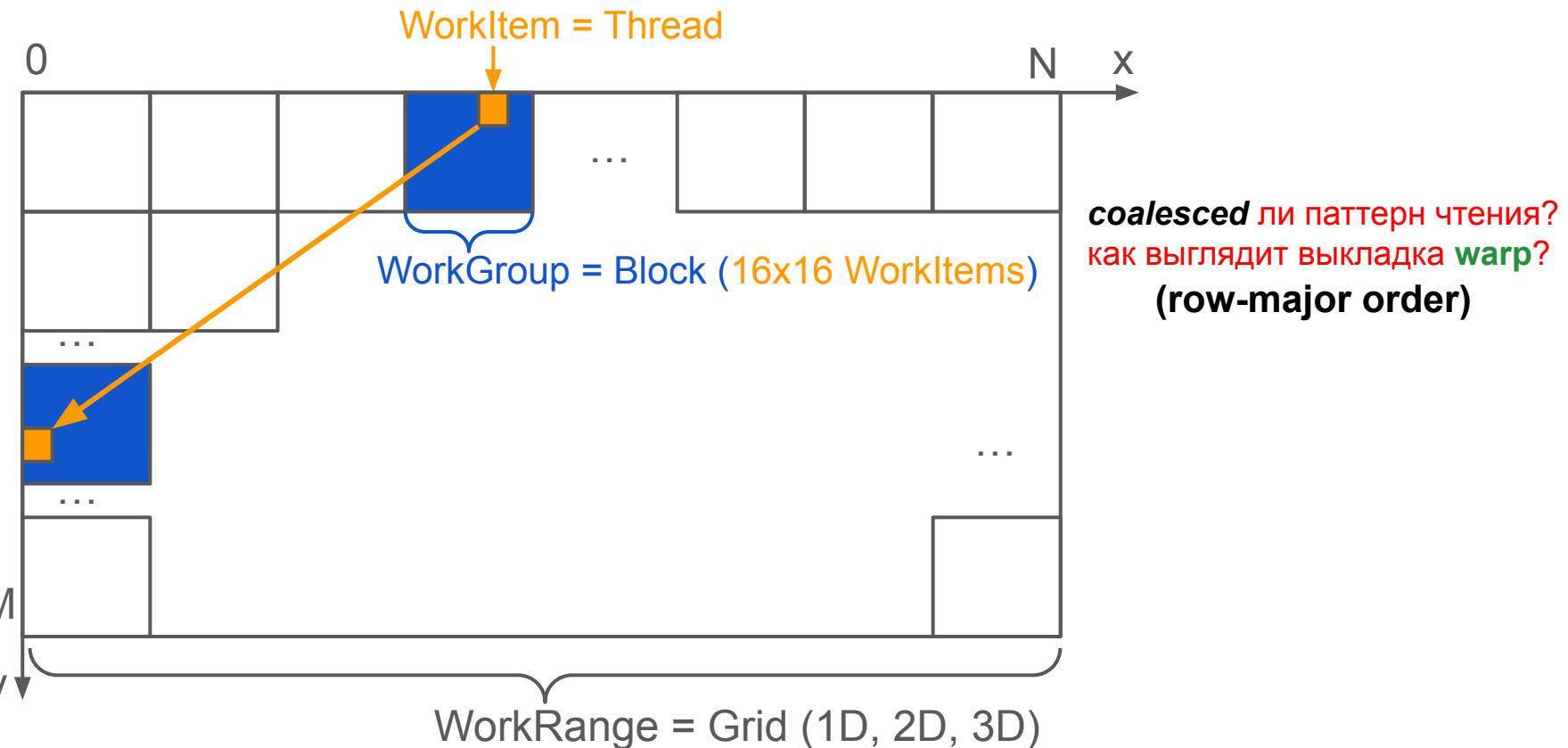
Транспонирование матрицы



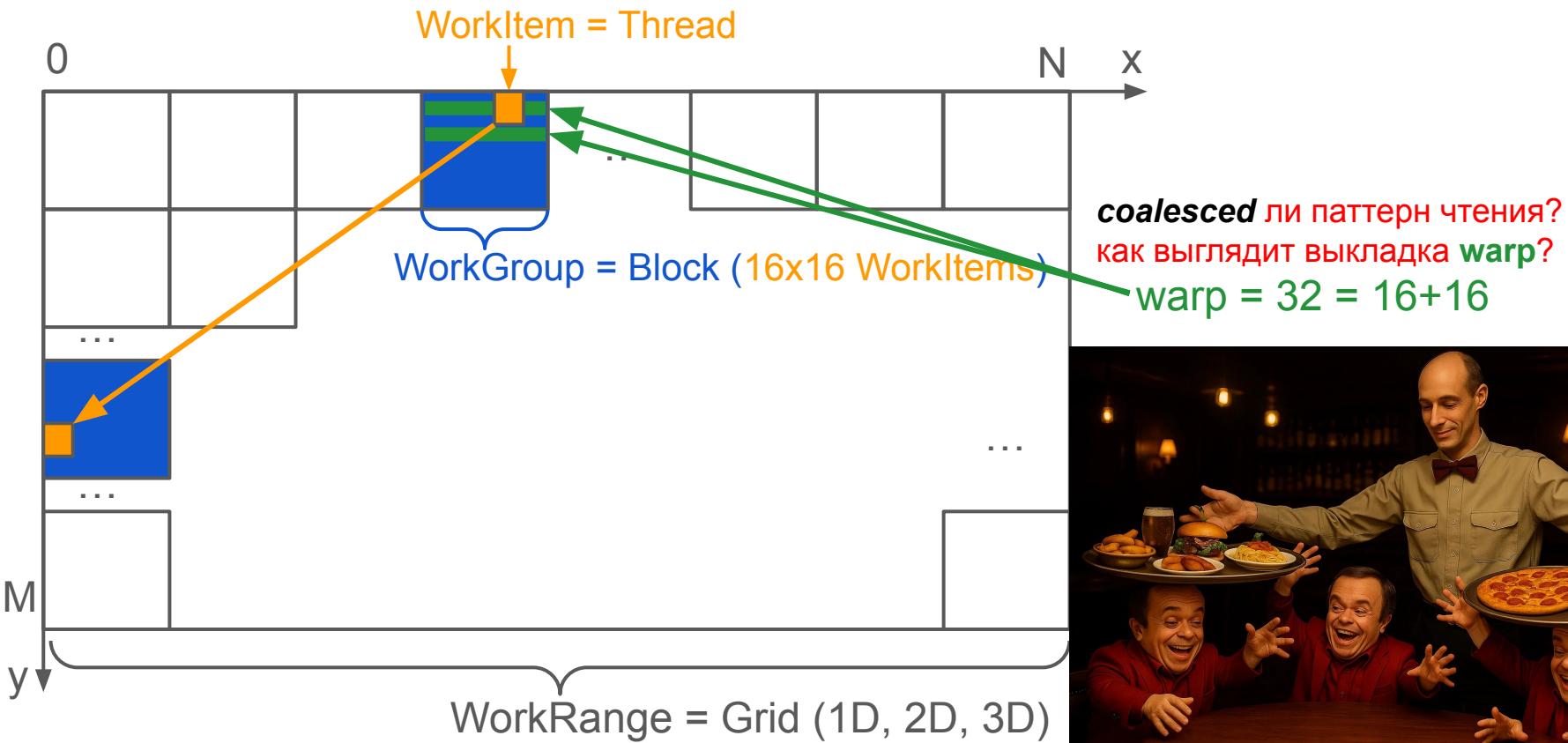
Транспонирование матрицы (*coalesced memory access*)



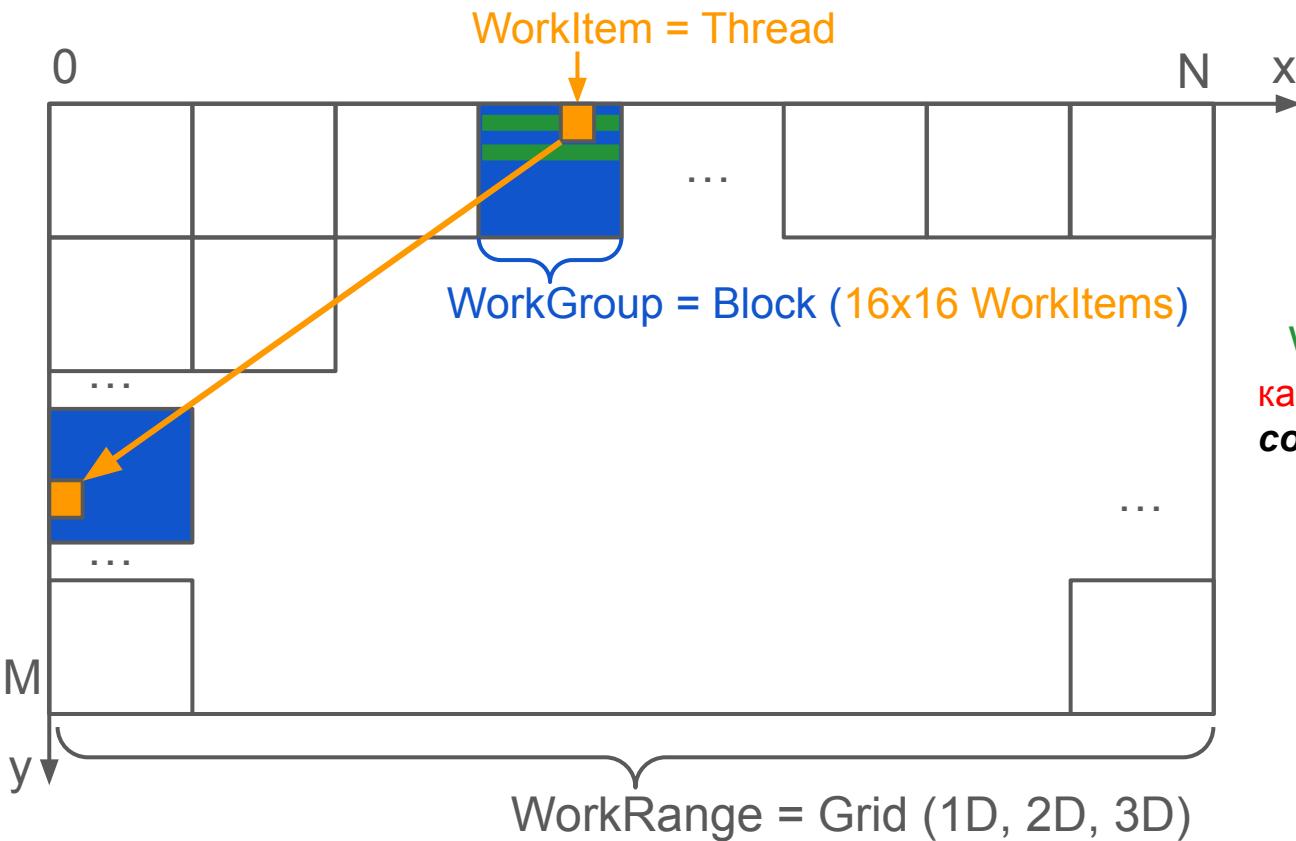
Транспонирование матрицы (*coalesced memory access*)



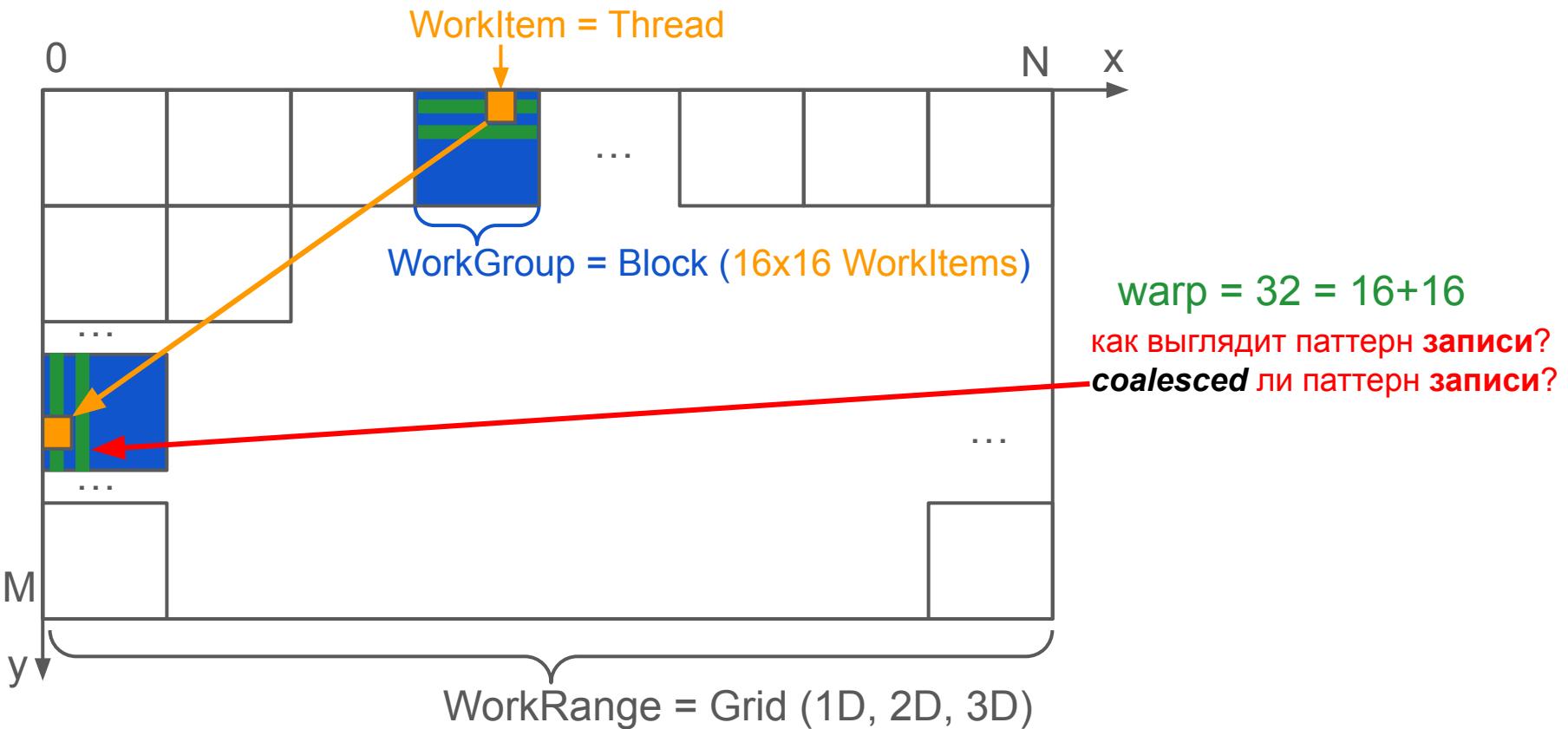
Транспонирование матрицы (*coalesced memory access*)



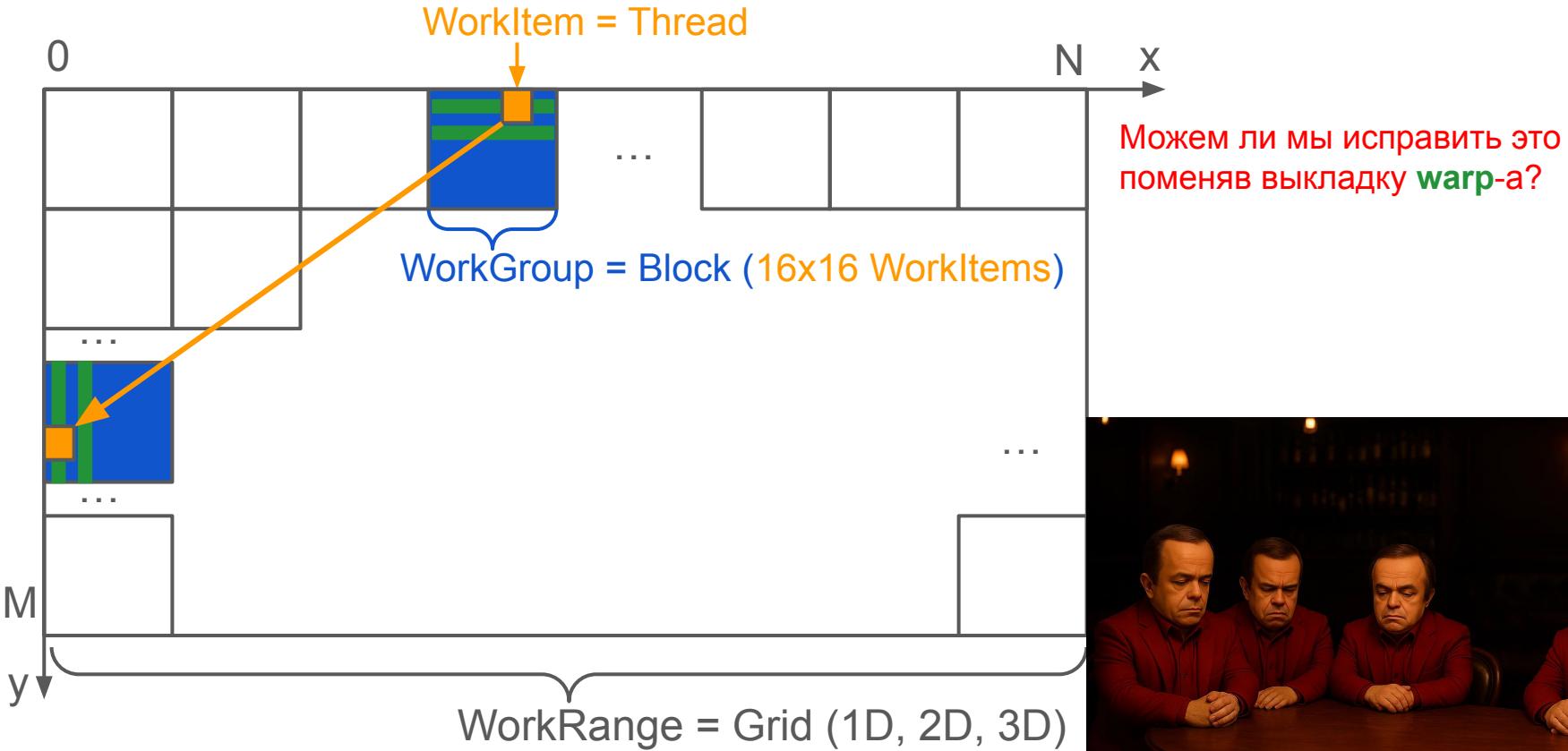
Транспонирование матрицы (*coalesced memory access*)



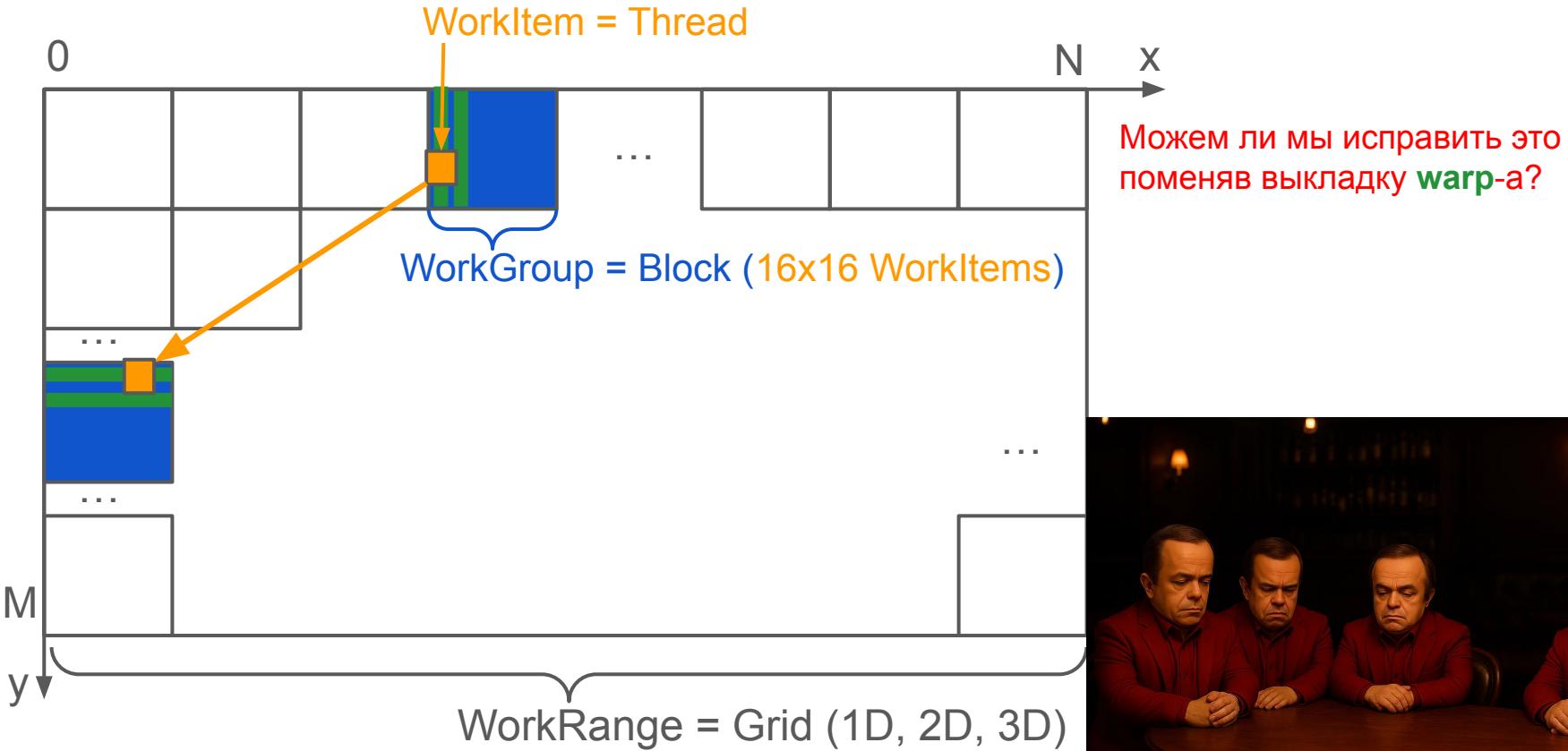
Транспонирование матрицы (*coalesced memory access*)



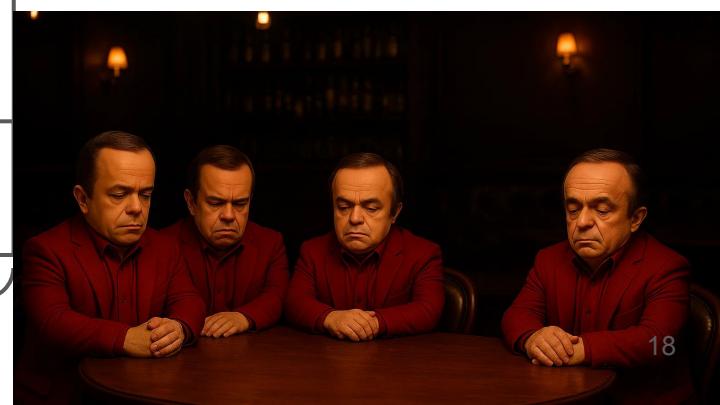
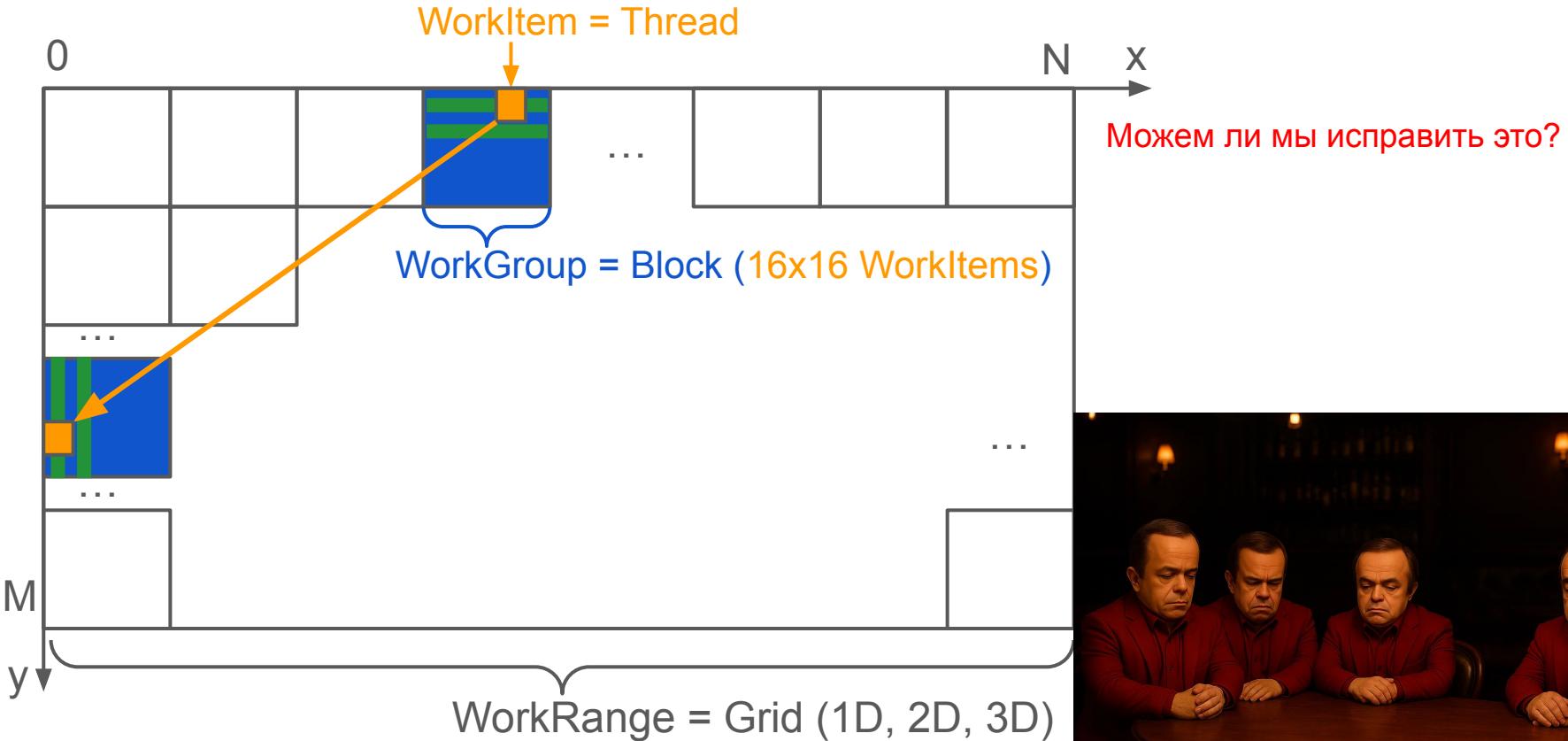
Транспонирование матрицы (*coalesced memory access*)



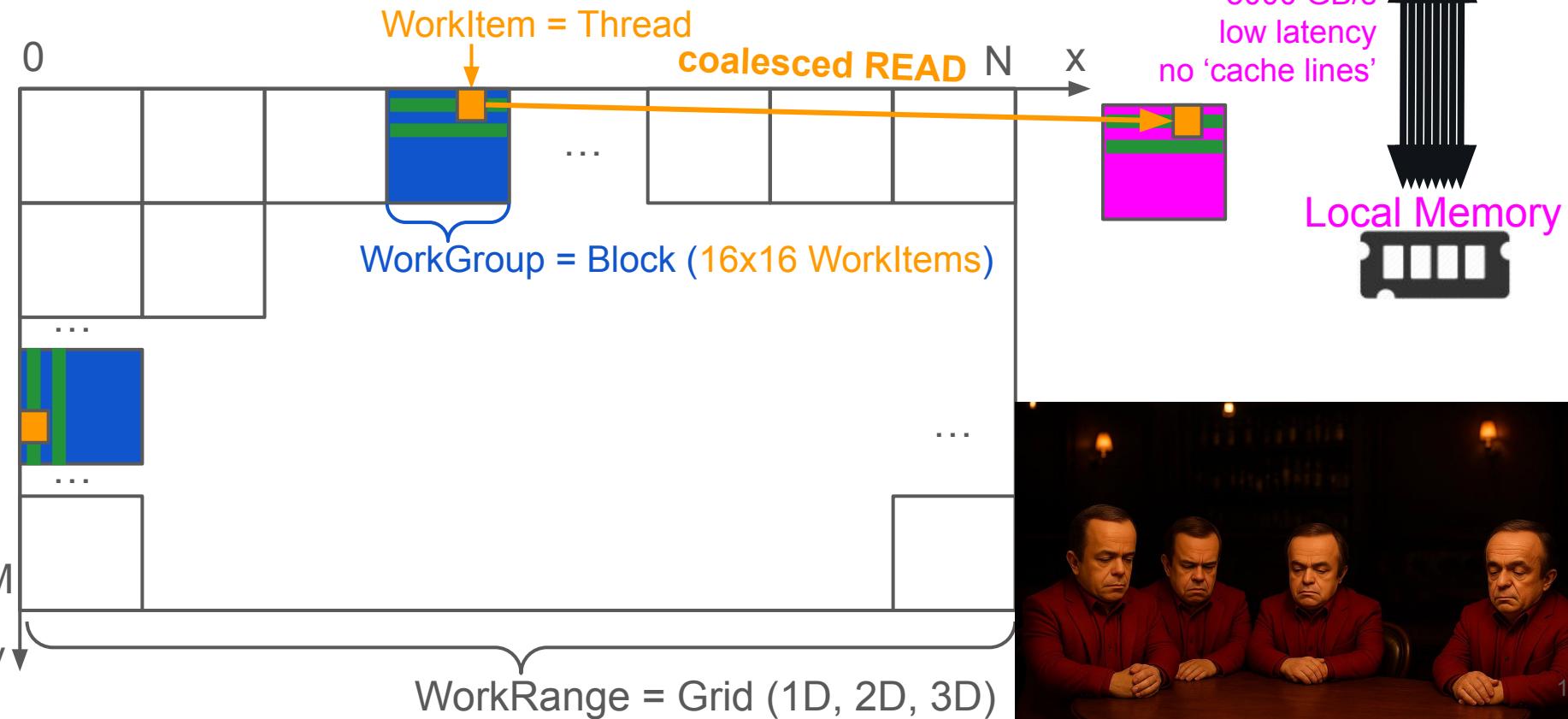
Транспонирование матрицы (*coalesced memory access*)



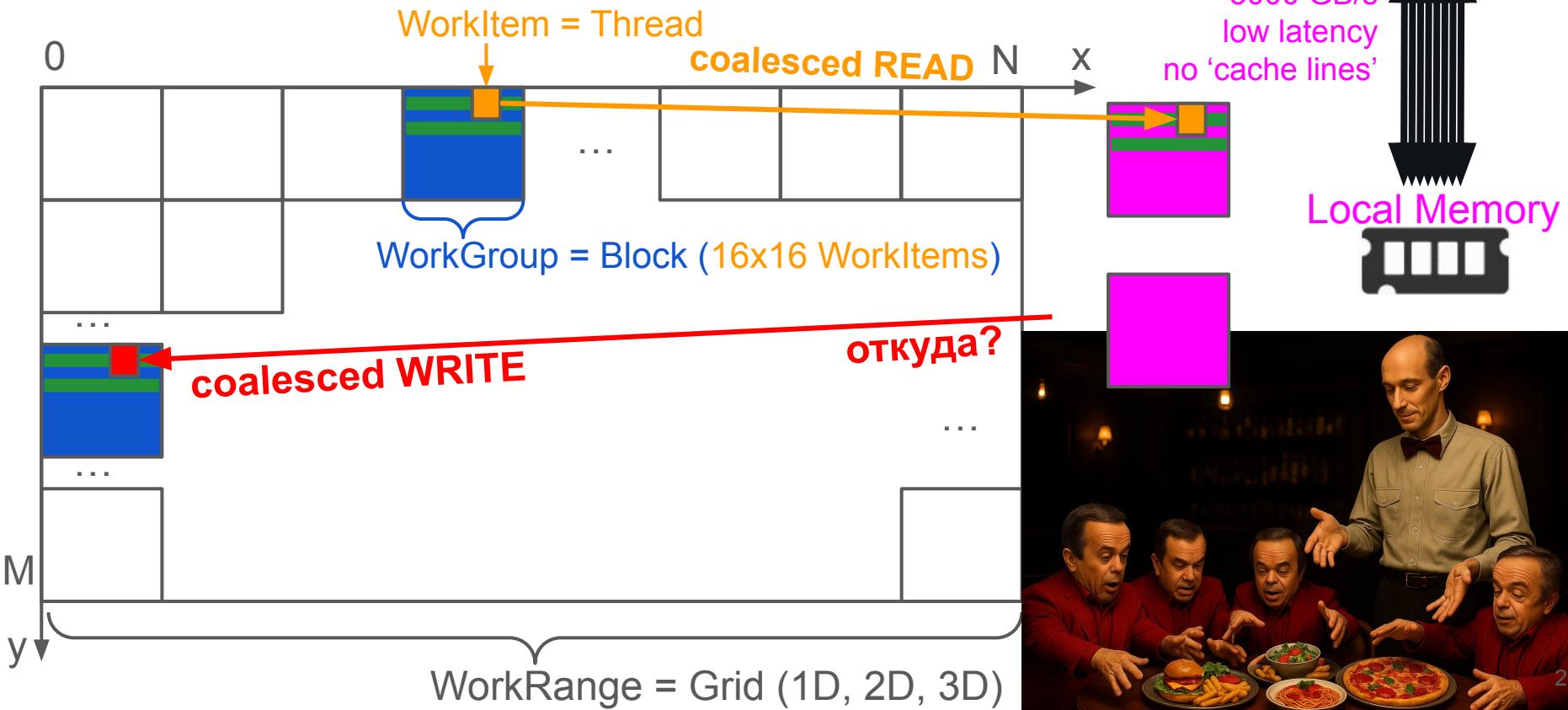
Транспонирование матрицы (*coalesced memory access*)



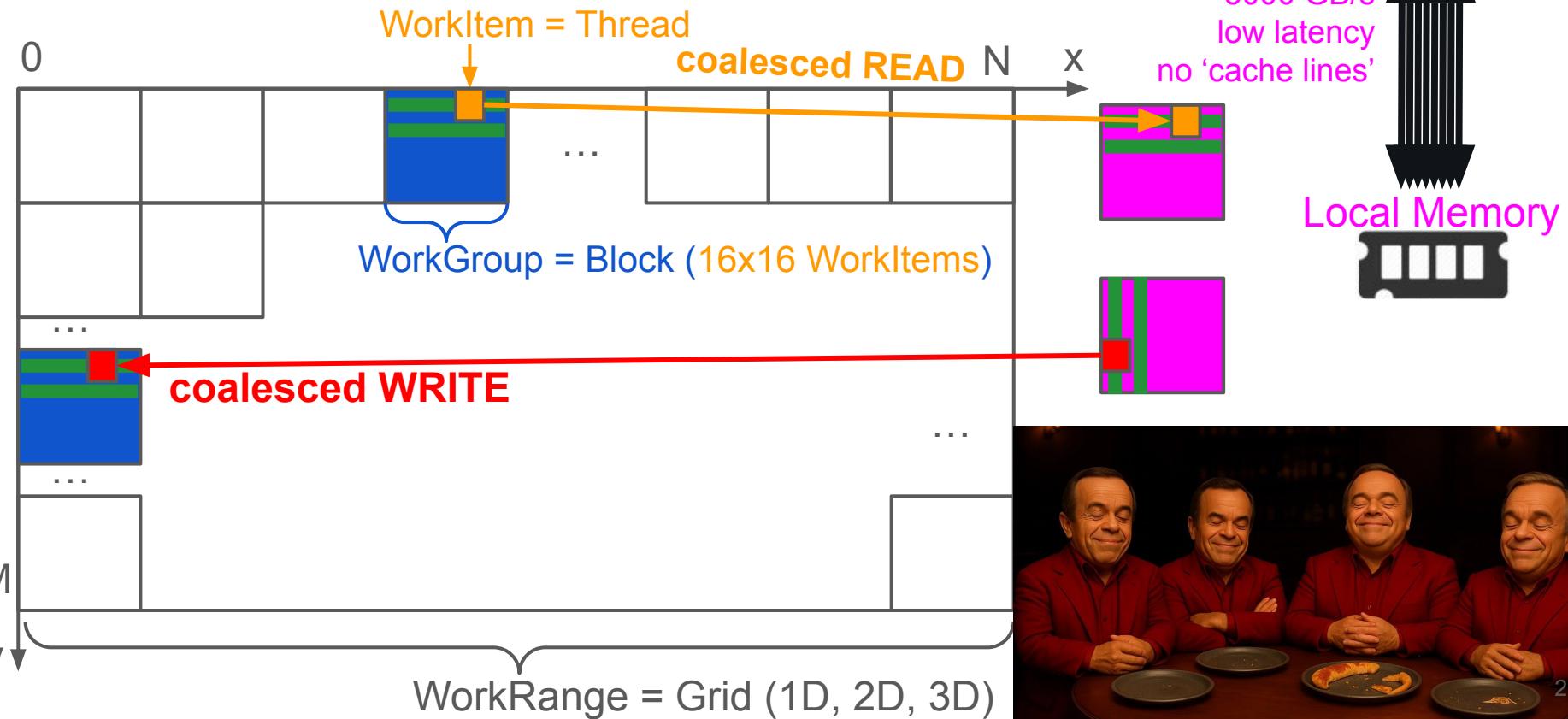
Транспонирование матрицы (coalesced)



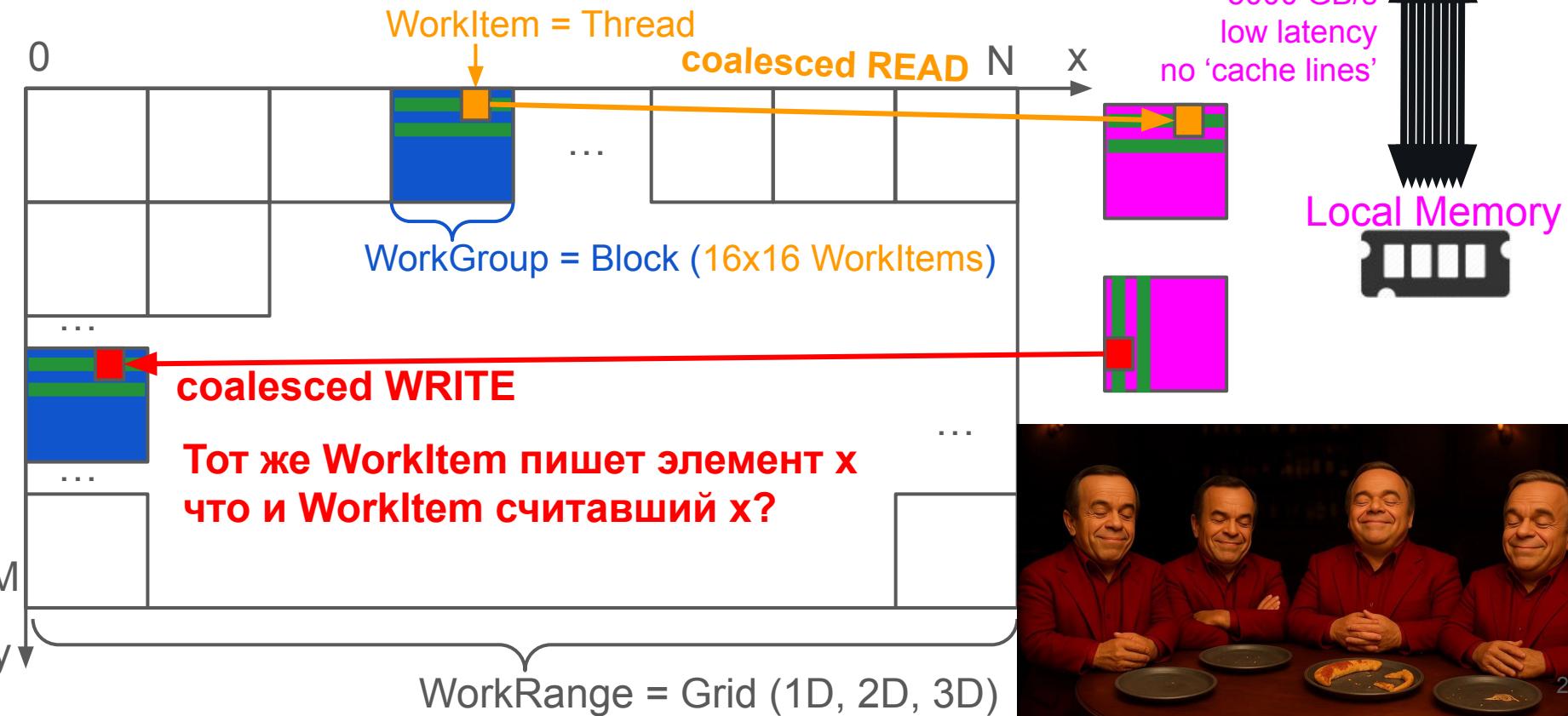
Транспонирование матрицы (coalesced)



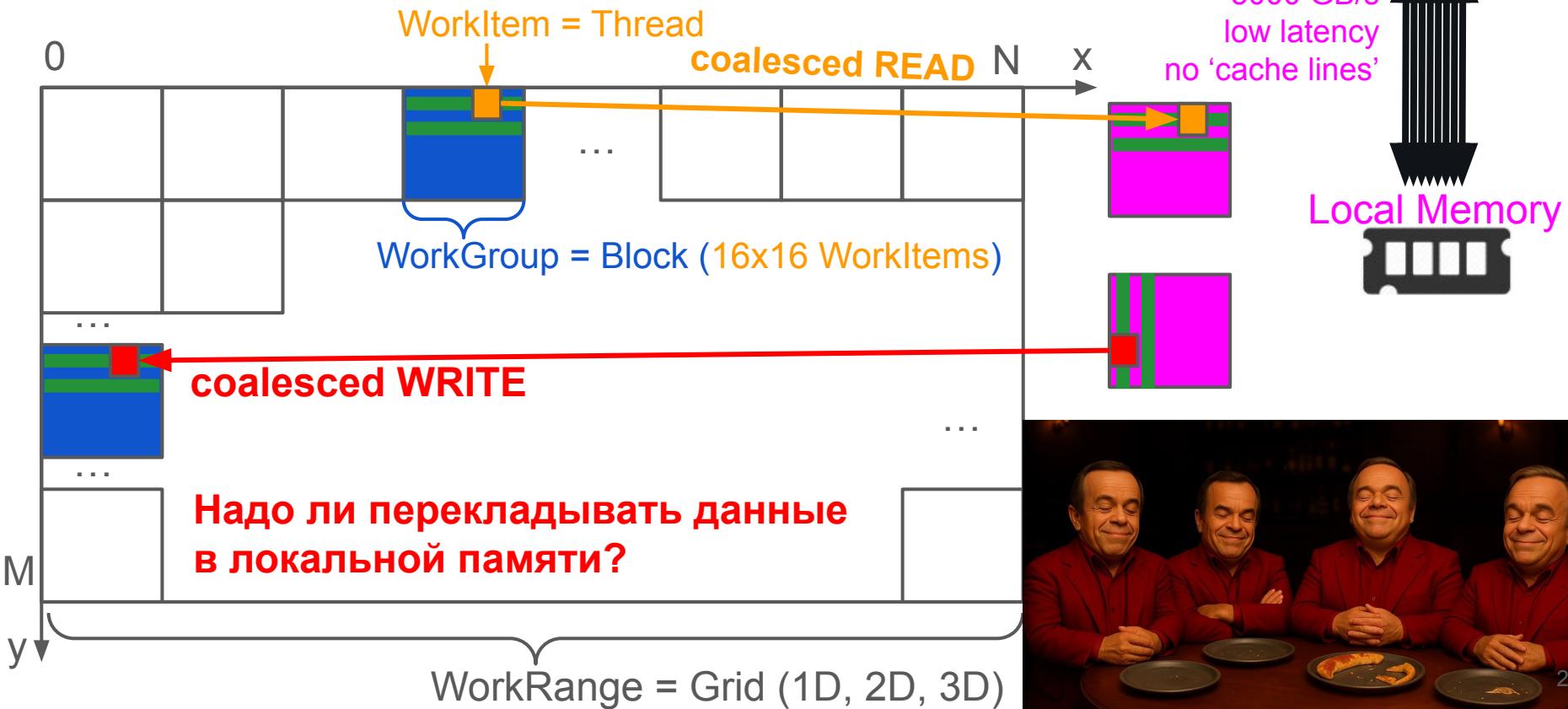
Транспонирование матрицы (coalesced)



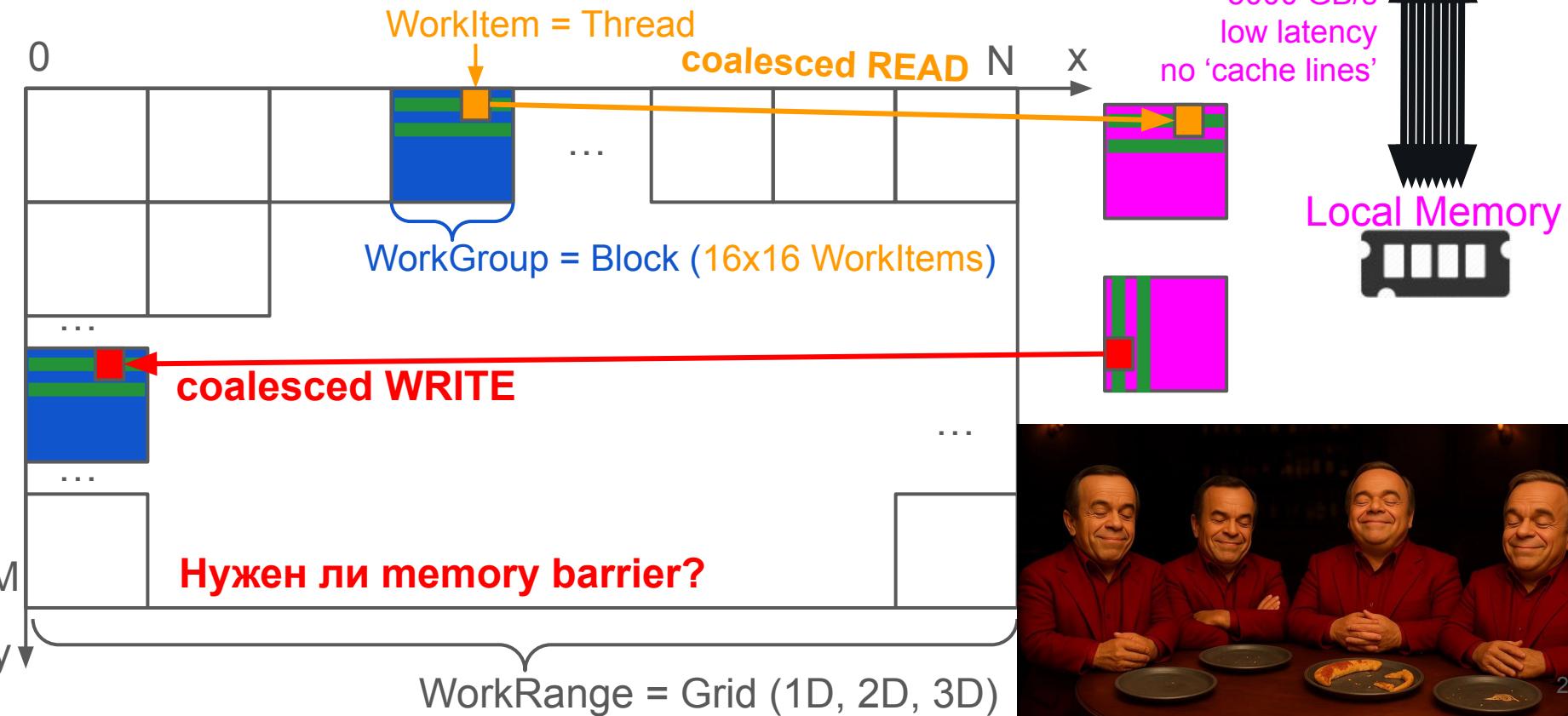
Транспонирование матрицы (coalesced)



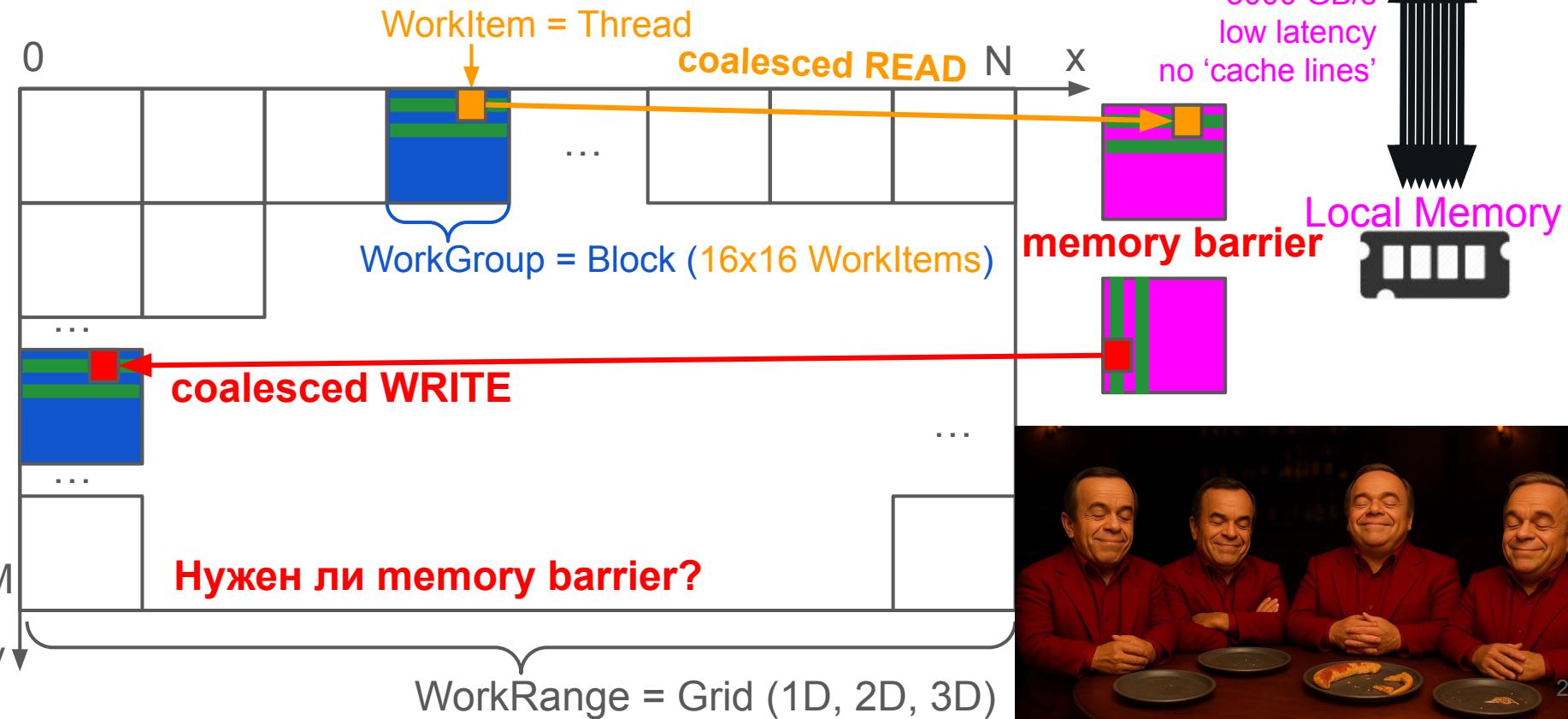
Транспонирование матрицы (coalesced)



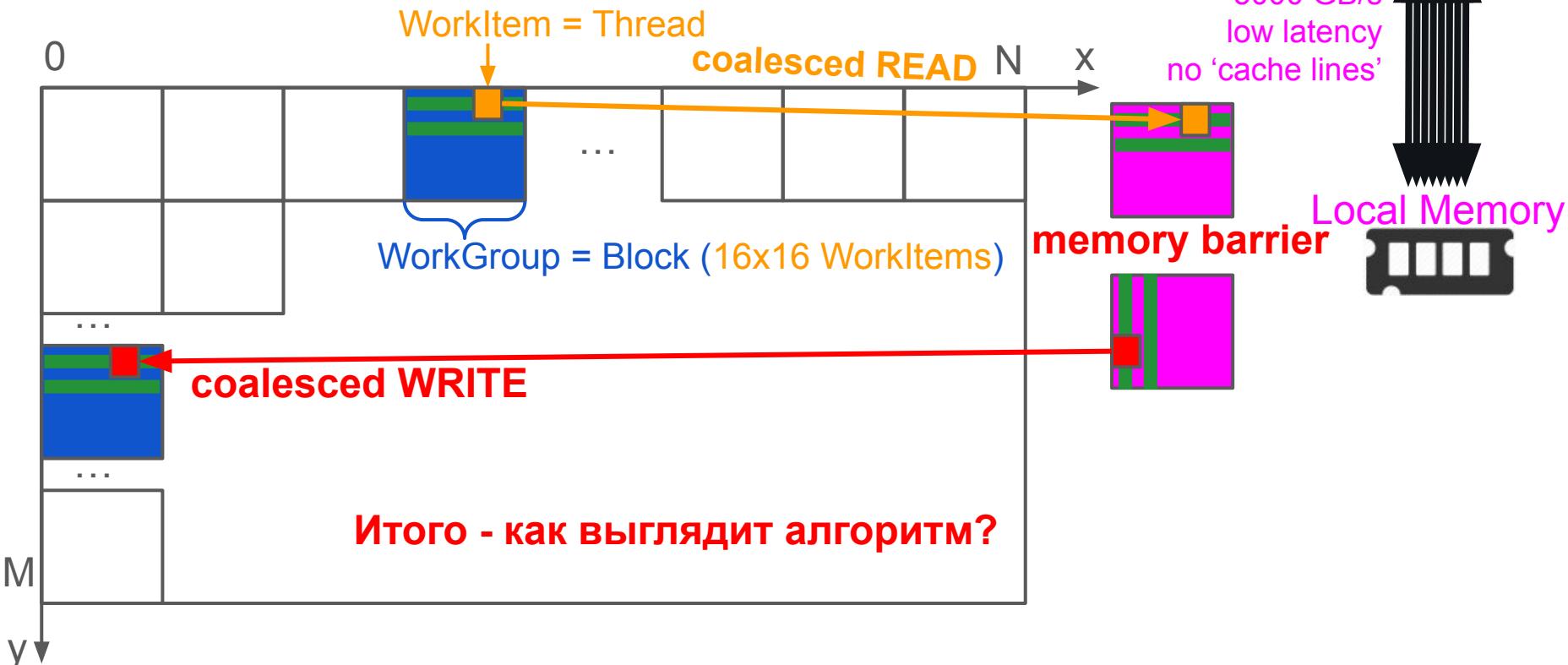
Транспонирование матрицы (coalesced)



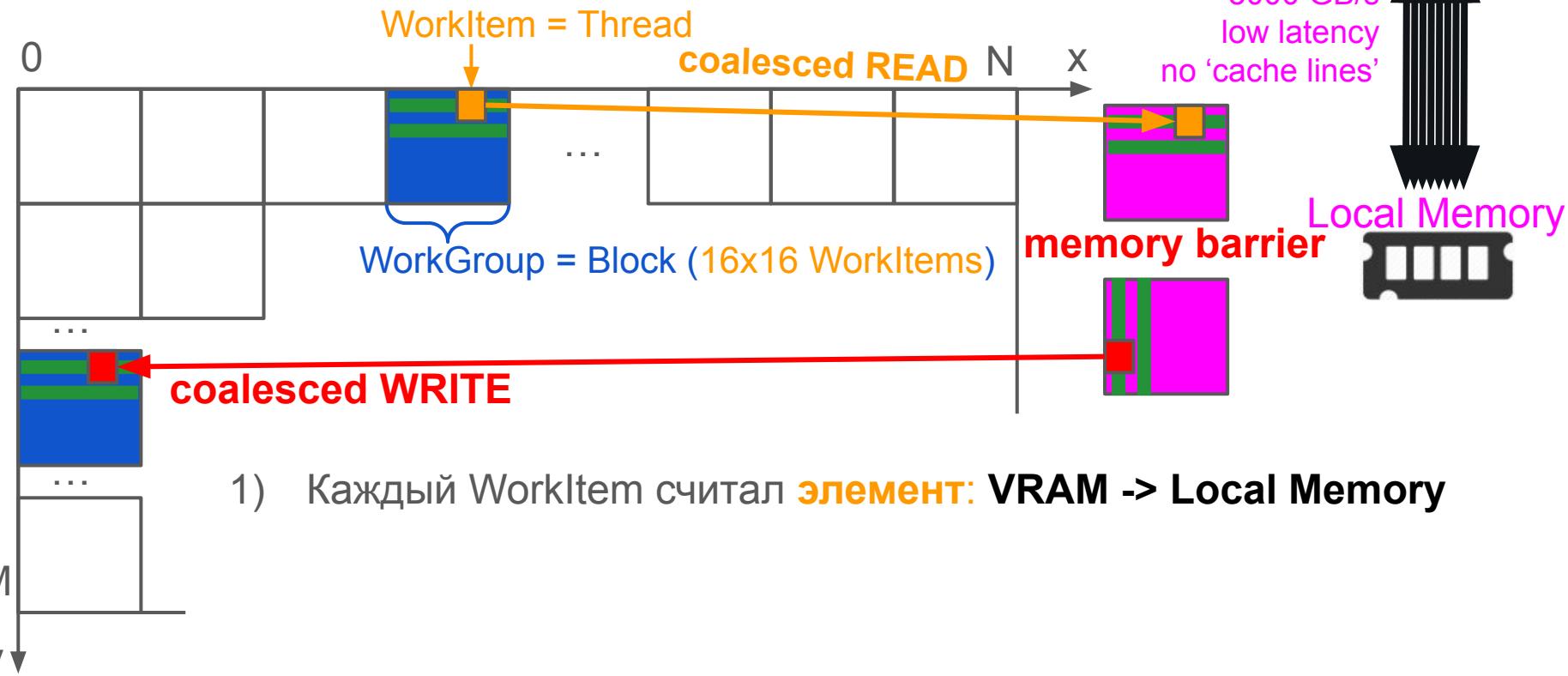
Транспонирование матрицы (coalesced)



Транспонирование матрицы (coalesced)

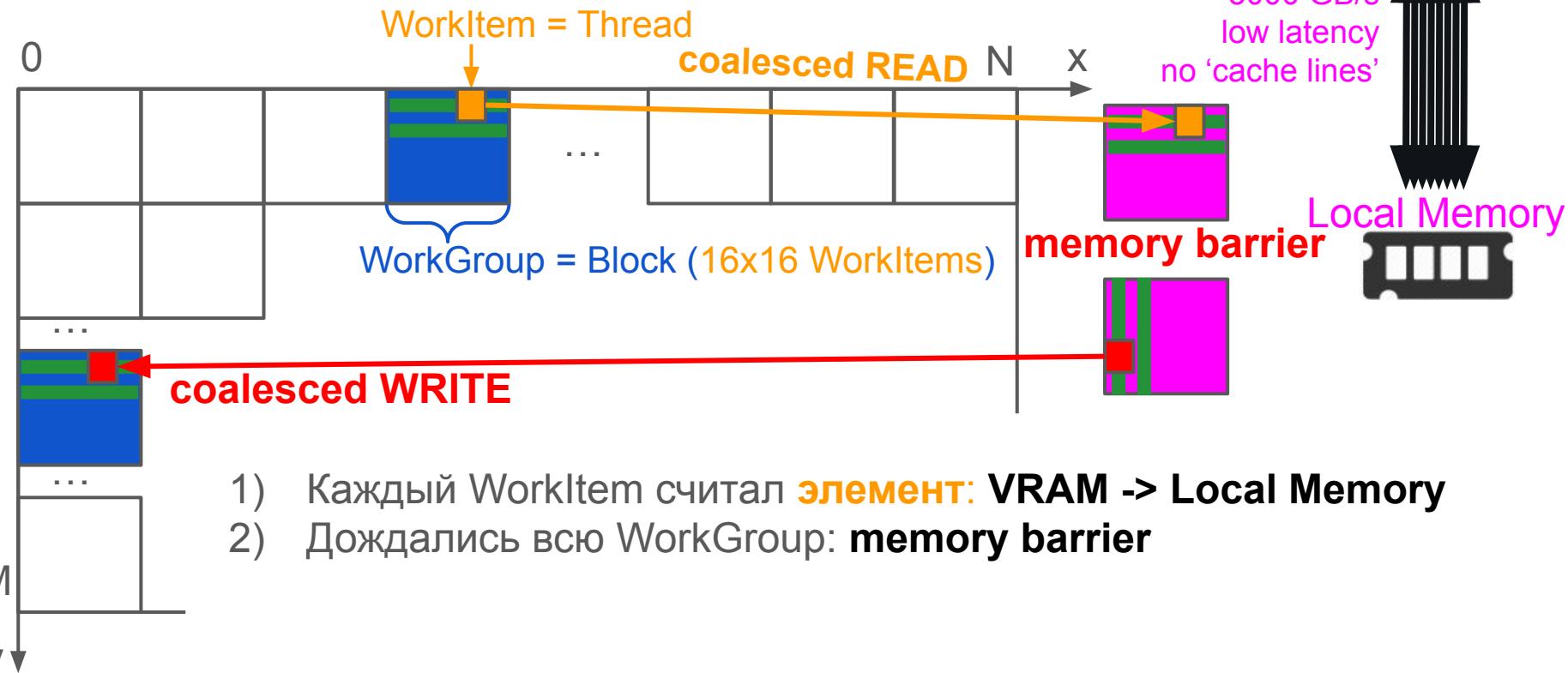


Транспонирование матрицы (coalesced)



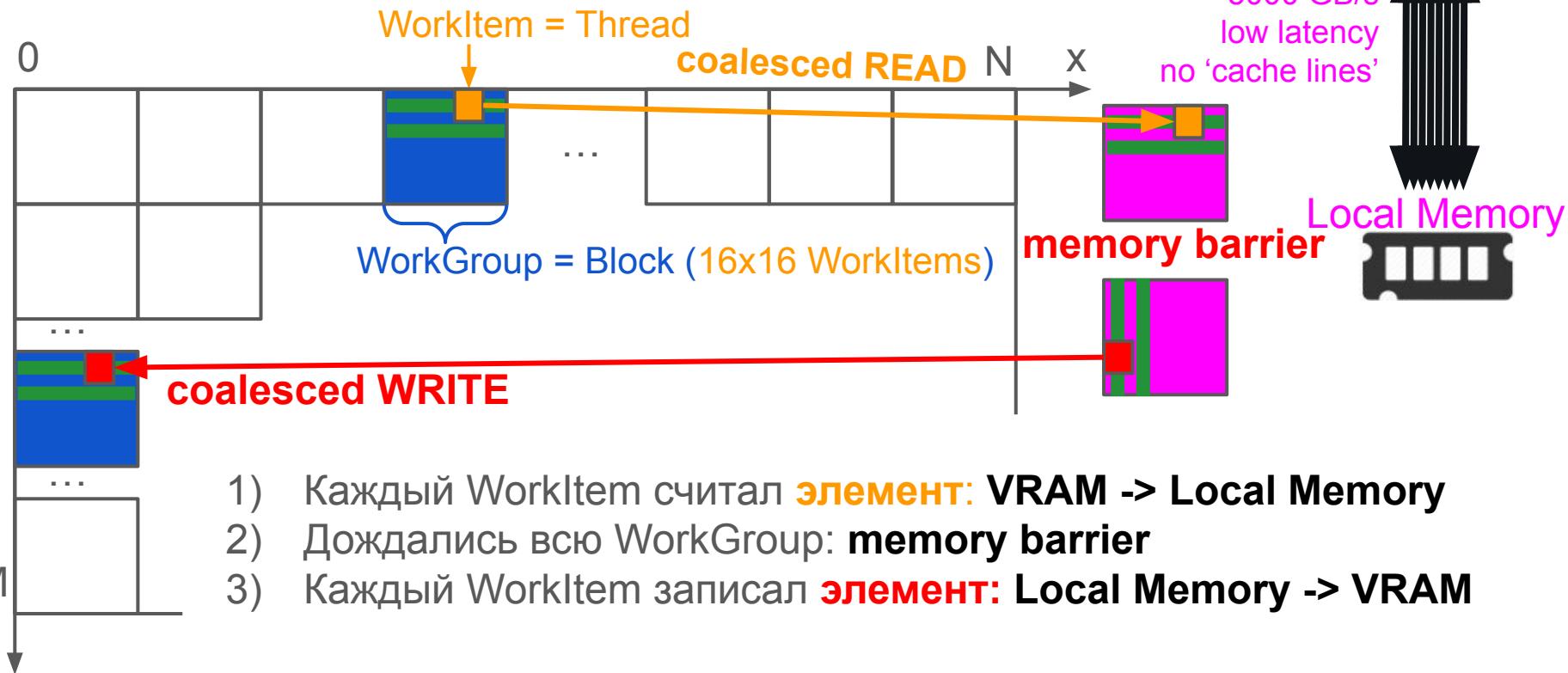
- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory

Транспонирование матрицы (coalesced)



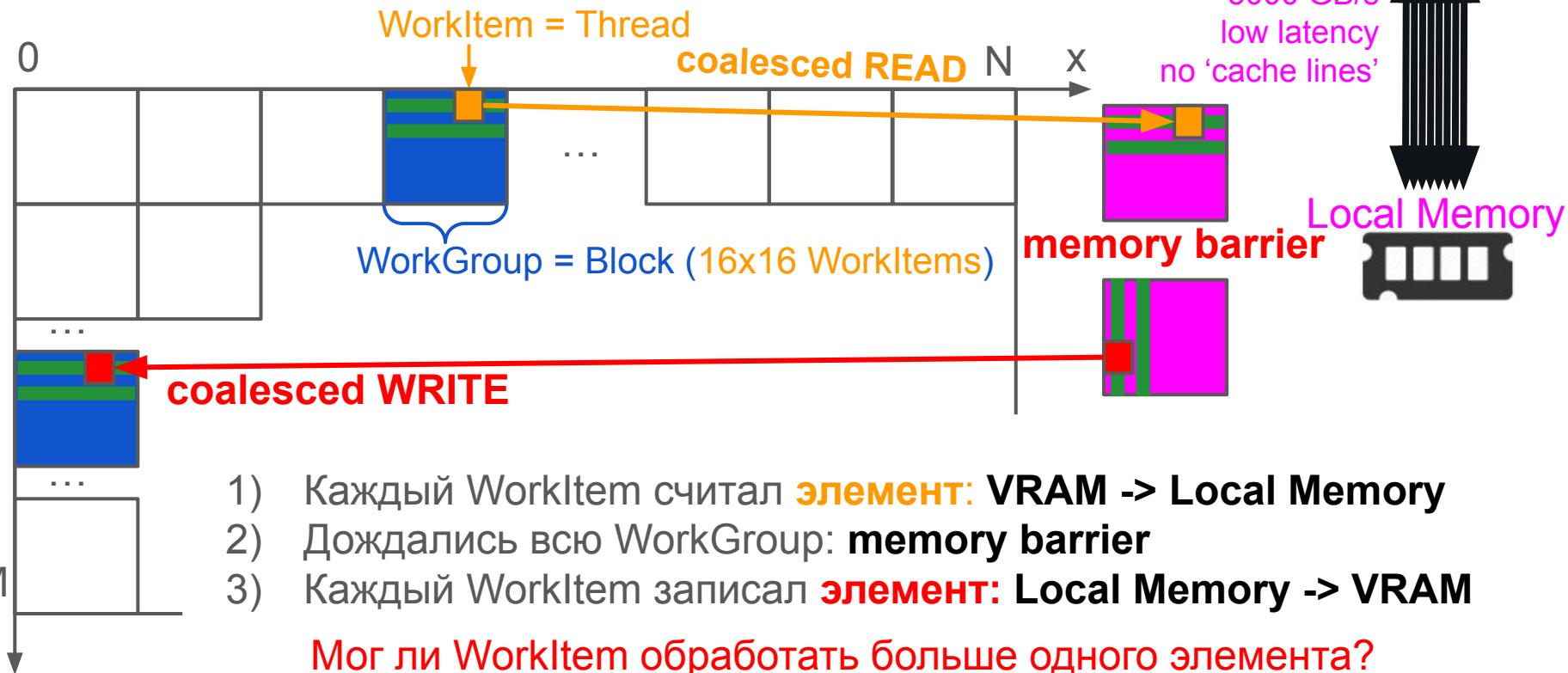
- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory
- 2) Дождались всю WorkGroup: **memory barrier**

Транспонирование матрицы (coalesced)



- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory
- 2) Дождались всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory \rightarrow VRAM

Транспонирование матрицы (coalesced)



- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory
- 2) Дождались всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory \rightarrow VRAM

Мог ли WorkItem обработать больше одного элемента?

Транспонирование

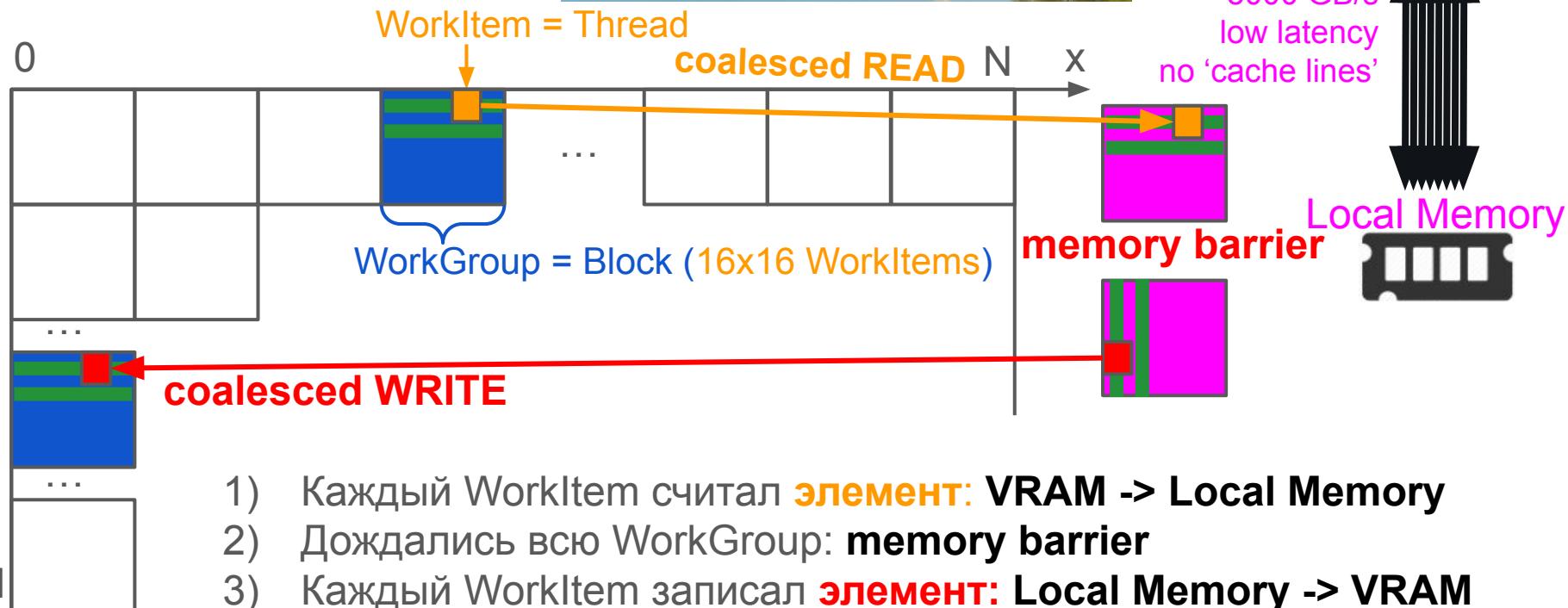


5000 GB/s
low latency
no 'cache lines'



Local Memory

memory barrier



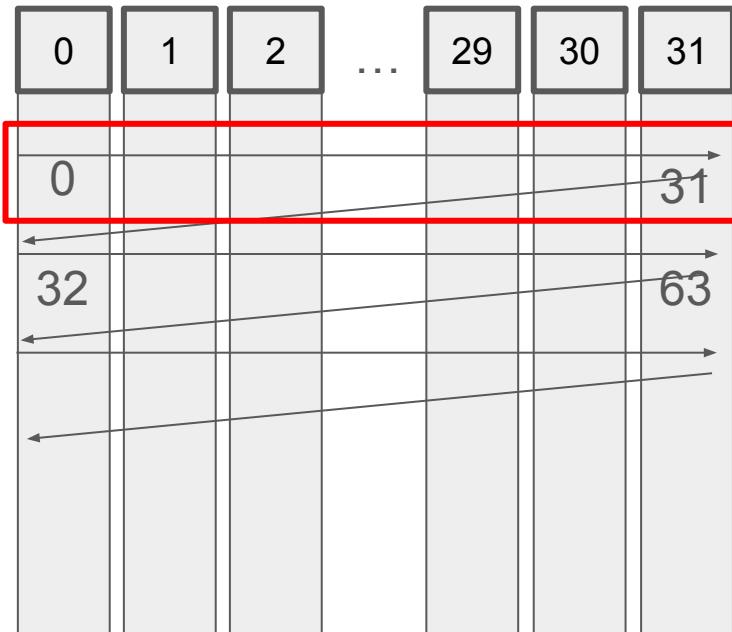
- 1) Каждый WorkItem считал **элемент**: VRAM -> Local Memory
- 2) Дождались всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory -> VRAM

Во что мы уперлись? Compute/Memory - bound?

Локальная память - bank conflicts (напоминание)

```
__local unsigned int workgroup_data[256];  
workgroup_data[get_local_id(0)] = a[index];
```

Local memory banks:



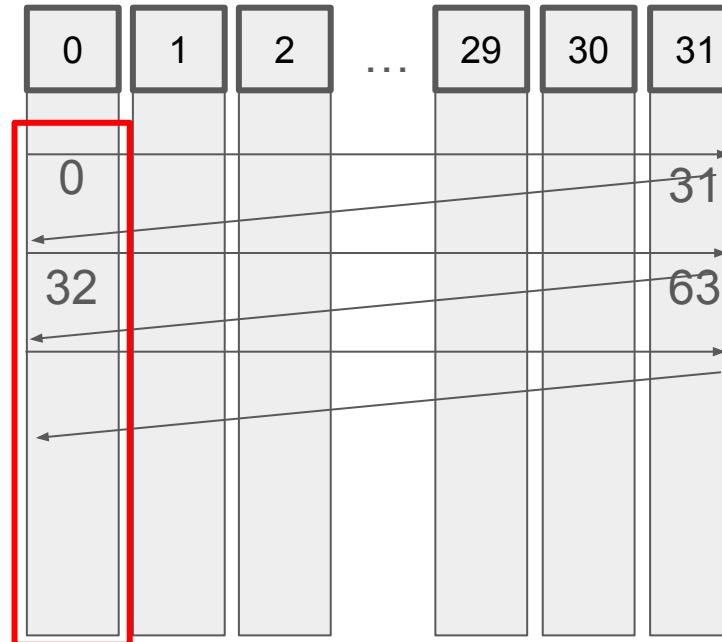
Local address space:

Локальная память - bank conflicts (напоминание)

```
__local unsigned int workgroup_data[256];  
workgroup_data[32 * get_local_id(0)] = a[index];
```

Local memory banks:

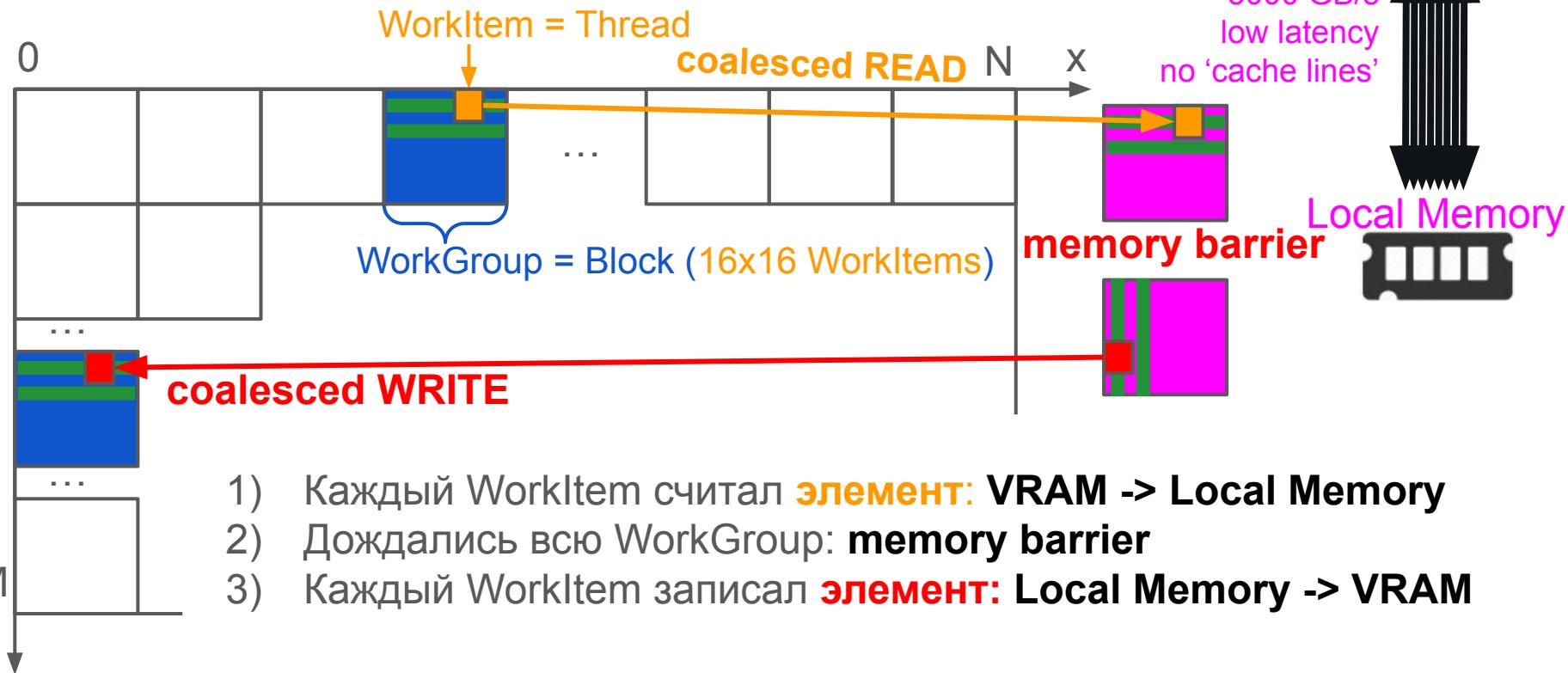
Local address space:



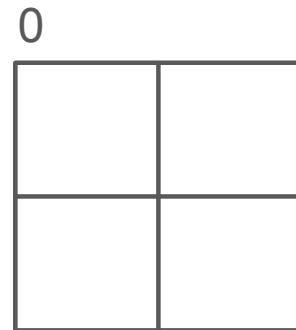
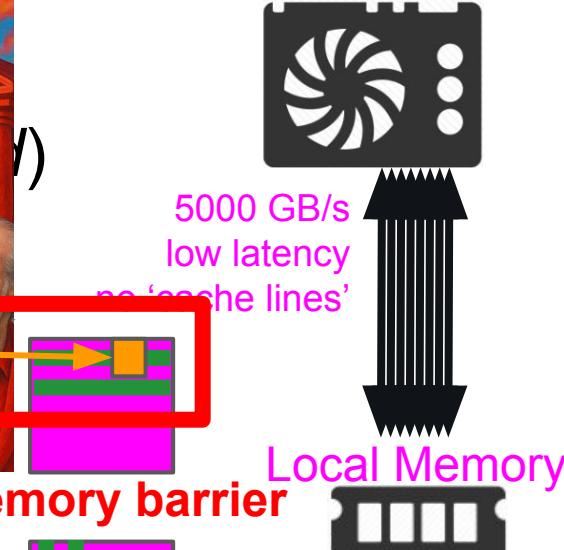
Bank conflict!



Транспонирование матрицы (coalesced)



Транспонирование



WorkItem =

WorkGroup = Block (16x16 WorkItems)

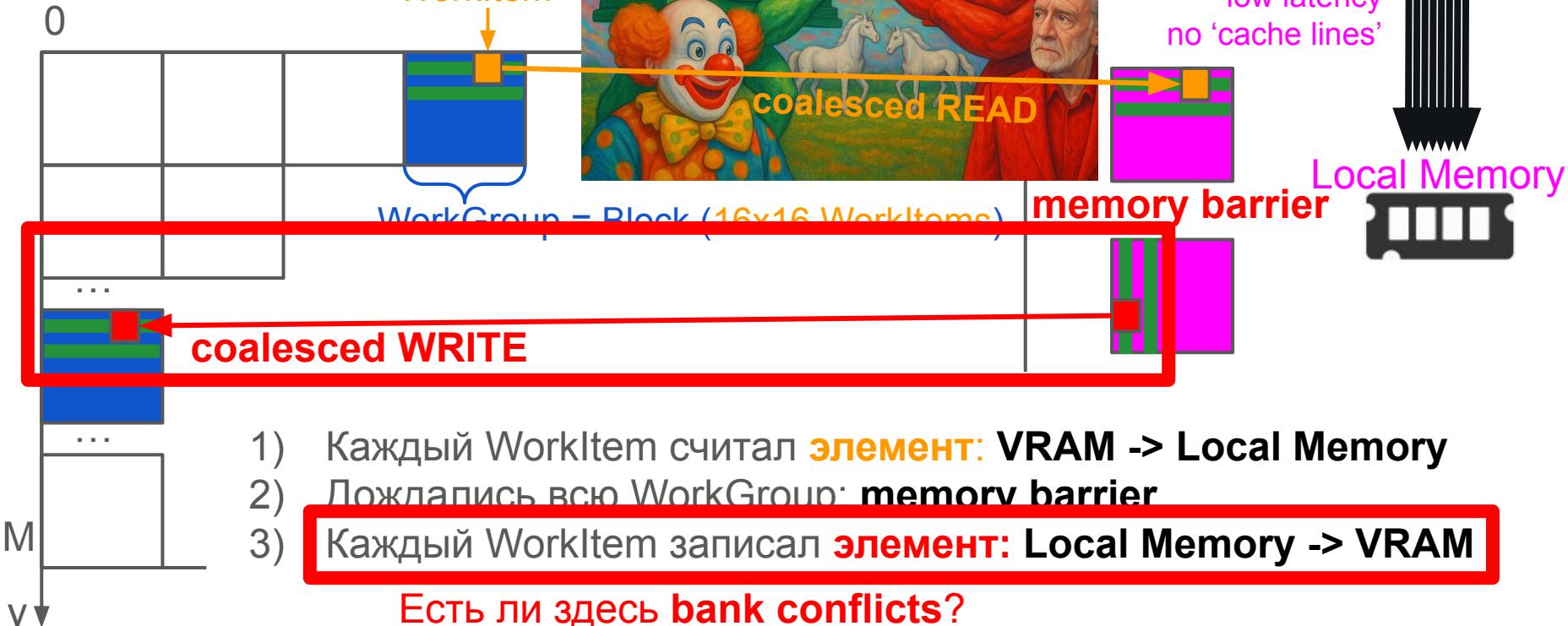
coalesced READ

coalesced WRITE

- 1) Каждый WorkItem считал **элемент**: VRAM -> Local Memory
- 2) дождались всю workGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory -> VRAM

Есть ли здесь **bank conflicts**?

Транспонирование



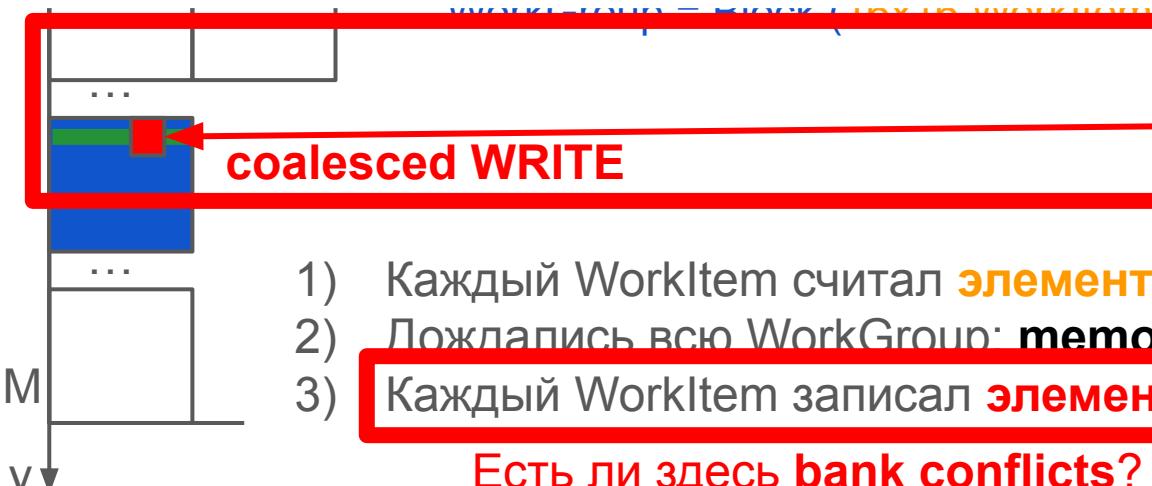


5000 GB/s
low latency
no 'cache lines'



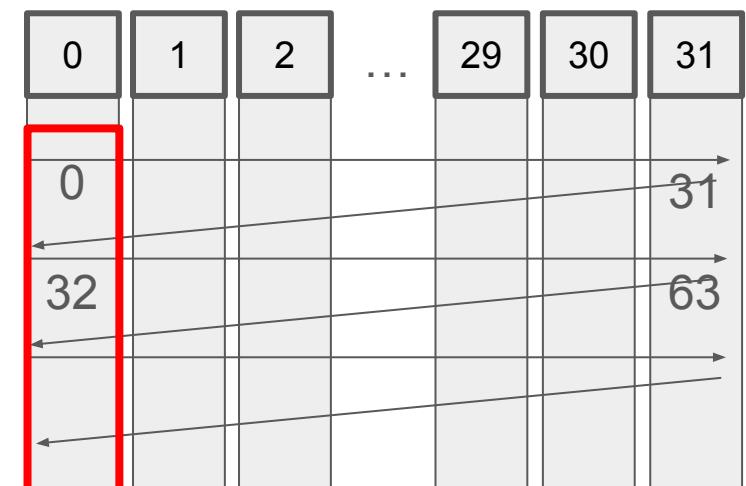
Для простоты будем считать
что WorkGroup **32 x 32!**

И Tile в Local Memory - **32 x 32!**



- 1) Каждый WorkItem считал **элемент**: VRAM -> Local Memory
- 2) Дождалась всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory -> VRAM

Есть ли здесь **bank conflicts**?

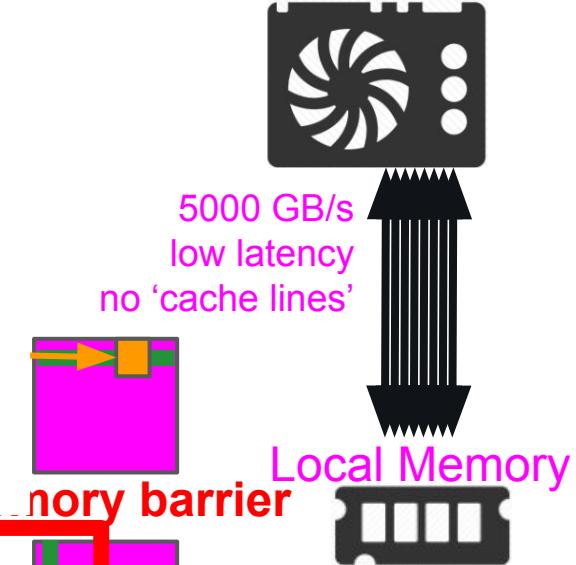


У каждого WorkItem есть свой буфер для вычислений

...
...

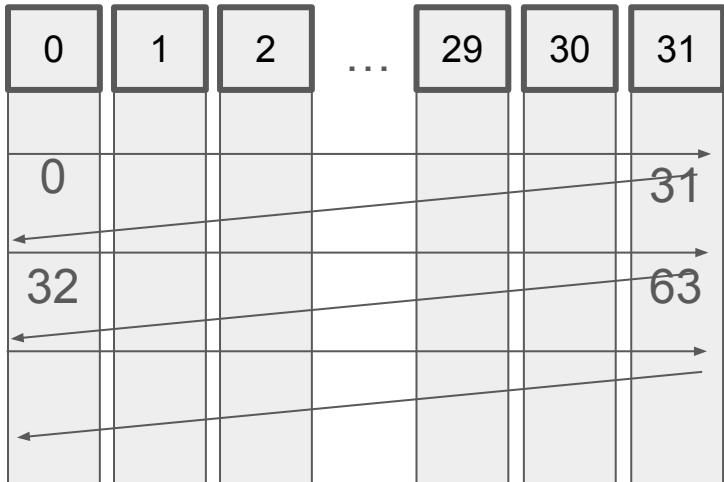
coalesced WRITE

М
у



- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory
- 2) Дождались всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory \rightarrow VRAM

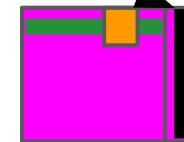
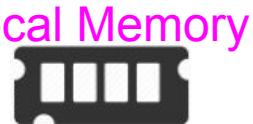
Есть ли здесь **bank conflicts**?



Что будет если
Tile - **33x32?**

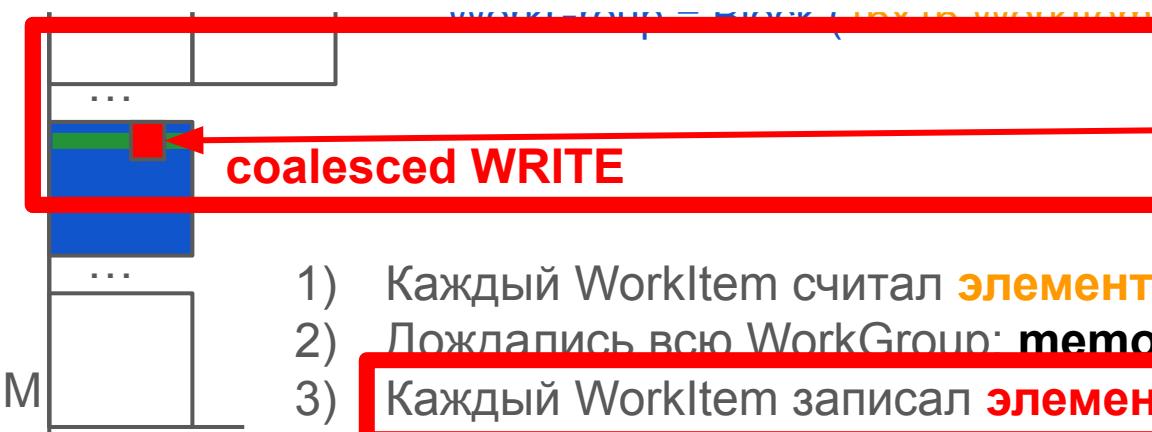


5000 GB/s
low latency
no 'cache lines'



memory barrier

coalesced WRITE



- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory
- 2) Дождалась всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory \rightarrow VRAM

Есть ли здесь **bank conflicts**?



Что будет если
Tile - **33x32?**

В какой банке второй
элемент столбика?

5000 GB/s
low latency
no 'cache lines'

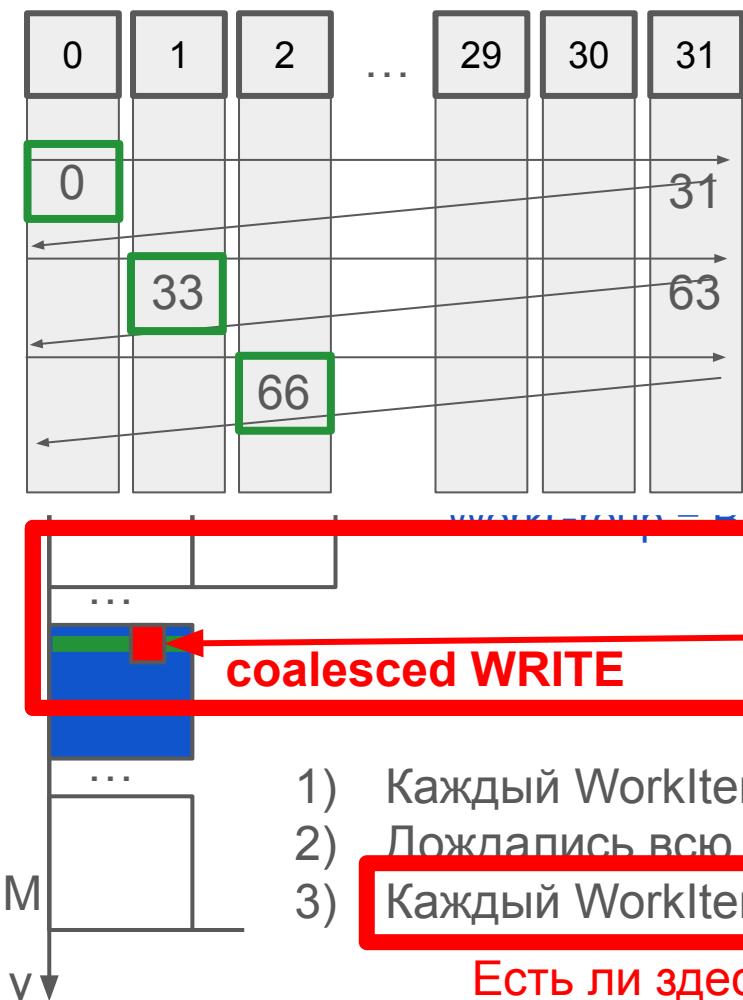


- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory
- 2) Дождалась всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory \rightarrow VRAM

Есть ли здесь **bank conflicts**?



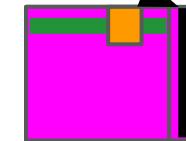
- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory
- 2) Дождалась всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory \rightarrow VRAM



Что будет если
Tile - 33x32?



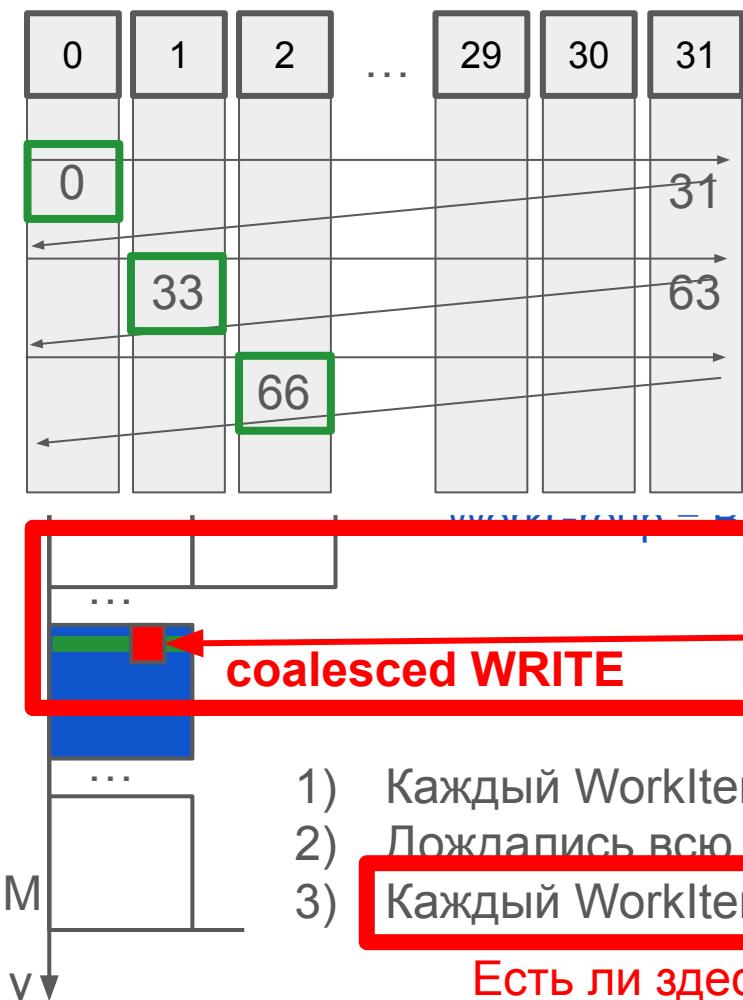
5000 GB/s
low latency
no 'cache lines'



Local Memory

memory barrier

- 1) Каждый WorkItem считал **элемент**: VRAM \rightarrow Local Memory
- 2) Дождалась всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory \rightarrow VRAM

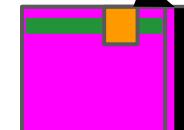


Что будет если
Tile - 33x32?

Можно ли без лишней
local memory?



5000 GB/s
low latency
no 'cache lines'



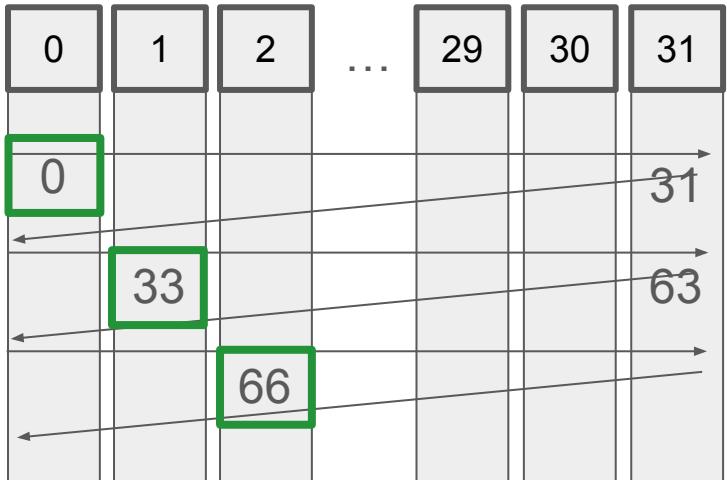
memory barrier



Local Memory

- 1) Каждый WorkItem считал **элемент**: VRAM -> Local Memory
- 2) Дождалась всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local Memory -> VRAM

Есть ли здесь **bank conflicts**?



Что будет если

Tile - 33x32?

А на что влияет
объем используемой
Local Memory?

coalesced WRITE

M
y

- 1) Каждый WorkItem считал **элемент**: VRAM ->
- 2) Дождалась всю WorkGroup: **memory barrier**
- 3) Каждый WorkItem записал **элемент**: Local M

Есть ли здесь **bank conflicts**?

SM - Streaming Multiprocessor



А на что влияет
объем используемой
Local Memory?
Occupancy!



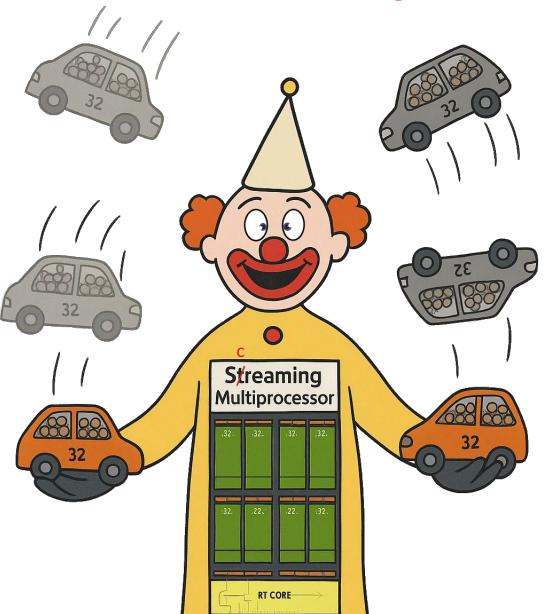
RTX 3090 (Ampere) - per SM:

- 128 KB Local Memory
 - 4 x 16384 x 32-bit registers
- = 256 KB Register File

SM - Streaming Multiprocessor



А на что влияет
объем используемой
Local Memory?
Occupancy!



RTX 3090 (Ampere) - per SM:

- **128 KB Local Memory**
- **4 x 16384 x 32-bit registers**
- = 256 KB Register File**

[DEPRECATED] Excel spreadsheet based occupancy calculator is deprecated.

Occupancy calculator in now available in Nsight Compute

CUDA Occupancy Calculator

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	8.6	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	
1.c) Select CUDA version	11,1	

2.) Enter your resource usage:	256	(Help)
Threads Per Block	256	
Registers Per Thread	32	
User Shared Memory Per Block (bytes)	2048	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:	8.6	(Help)
Active Threads per Multiprocessor	1536	
Active Warps per Multiprocessor	48	
Active Thread Blocks per Multiprocessor	6	
Occupancy of each Multiprocessor	100%	

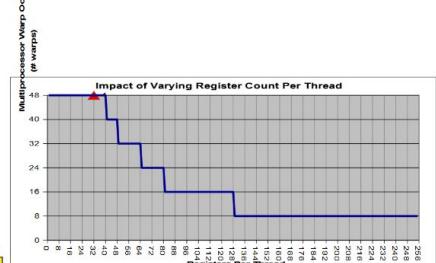
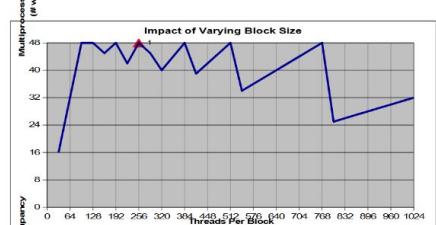
Physical Limits for GPU Compute Capability:	8.6	
Threads per Warp	32	
Max Warps per Multiprocessor	48	
Max Thread Blocks per Multiprocessor	16	
Max Threads per Multiprocessor	1536	
Maximum Thread Block Size	1024	
Registers per Multiprocessor	65538	
Max Registers per Thread Block	65538	
Max Registers per Thread	255	
Shared Memory per Multiprocessor (bytes)	65538	
Max Shared Memory per Block	256	
Register allocation unit size	warp	
Register allocation granularity	Shared Memory allocation unit size	
Shared Memory allocation unit size	128	
Warp allocation granularity	4	
Shared Memory (bytes) per Block (CUDA runtime use)	1024	

Allocated Resources	Per Block	Limit Per SM	Blocks Per SM	= Allocatable
Warp (Threads Per Block / Threads Per Warp)	8	48	6	
Registers (Warp limit per SM due to per-warp reg count)	8	64	8	
Shared Memory (Bytes)	2048	65536	32	

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block = Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	8

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit <http://developer.nvidia.com/cuda>

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



Note: SM is an abbreviation for (Streaming) Multiprocessor

SM - Streaming Multiprocessor





NVIDIA Nsight

Occupancy Calculator

Compute Capability:	8.6	Threads Per Block:	<input type="range" value="256"/> 256
Shared Memory Size Config (bytes):	102400	Registers Per Thread:	<input type="range" value="32"/> 32
Global Load Cache Mode:	L1+L2 (ca)	User Shared Memory Per Block (bytes):	<input type="range" value="2048"/> 2048
		Block Barriers:	<input type="range" value="1"/> 1

Apply Automatically



The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM. The launch parameters can be configured with the input widget to the left. This tool also provides various important statistics such as resource allocation, occupancy limiters, GPU Data, etc in the form of tables and graphs.



NVIDIA Nsight

Occupancy Calculator

Compute Capability: **8.6** Threads Per Block: **256**

Shared Memory Size Config (bytes): **102400** Registers Per Thread: **32**

Global Load Cache Mode: **L1+L2 (ca)** User Shared Memory Per Block (bytes): **2048**

Block Barriers: **1**

Apply Automatically **Apply** **Reset**

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM. The launch parameters can be configured with the input widget to the left. This tool also provides various important statistics such as resource allocation, occupancy limiters, GPU Data, etc in the form of tables and graphs.

RTX 3090 (Ampere) - Compute Capability 8.6



NVIDIA Nsight

Occupancy Calculator

Compute Capability: 8.6 Threads Per Block: 256

Shared Memory Size Config (bytes): 102400 Registers Per Thread: 32

Global Load Cache Mode: L1+L2 (ca) User Shared Memory Per Block (bytes): 2048

Block Barriers: 1

Apply Automatically

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM. The launch parameters can be configured with the input widget to the left. This tool also provides various important statistics such as resource allocation, occupancy limiters, GPU Data, etc in the form of tables and graphs.

RTX 3090 (Ampere) - Compute Capability 8.6
WorkGroup size - 256



NVIDIA Nsight

Occupancy Calculator

Compute Capability: 8.6 Threads Per Block: 256

Shared Memory Size Config (bytes): 102400 Registers Per Thread: 32

Global Load Cache Mode: L1+L2 (ca) User Shared Memory Per Block (bytes): 2048

Block Barriers: 1

Apply Automatically

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM. The launch parameters can be configured with the input widget to the left. This tool also provides various important statistics such as resource allocation, occupancy limiters, GPU Data, etc in the form of tables and graphs.

RTX 3090 (Ampere) - Compute Capability 8.6

WorkGroup size - 256

Registers Per Thread - 32



NVIDIA Nsight

Occupancy Calculator

Compute Capability: 8.6 Threads Per Block: 256

Shared Memory Size Config (bytes): 102400 Registers Per Thread: 32

Global Load Cache Mode: L1+L2 (ca) User Shared Memory Per Block (bytes): 2048

Block Barriers: 1

Apply Automatically

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM. The launch parameters can be configured with the input widget to the left. This tool also provides various important statistics such as resource allocation, occupancy limiters, GPU Data, etc in the form of tables and graphs.

RTX 3090 (Ampere) - Compute Capability 8.6

WorkGroup size - 256

Registers Per Thread - 32

Local memory Per Thread - 8 bytes



Compute Capability: 8.6 Threads Per Block: 256

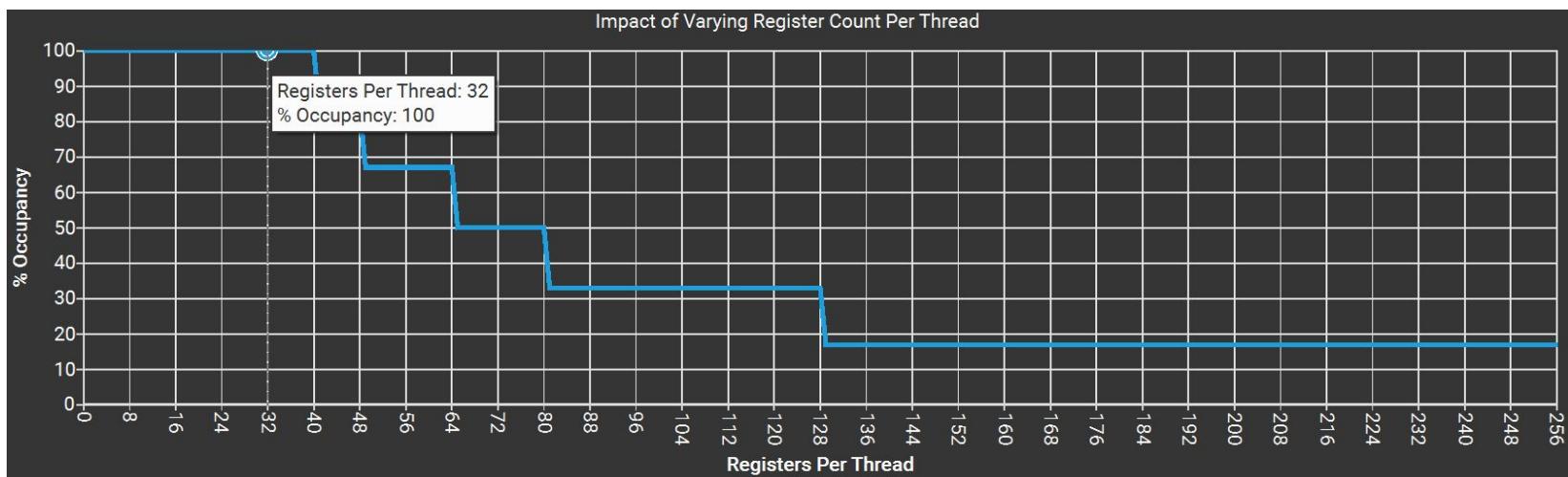
Shared Memory Size Config (bytes): 102400 Registers Per Thread: 32

Global Load Cache Mode: L1+L2 (ca) User Shared Memory Per Block (bytes): 2048

Block Barriers: 1

Apply Automatically

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM. The launch parameters can be configured with the input widget to the left. This tool also provides various important statistics such as resource allocation, occupancy limiters, GPU Data, etc in the form of tables and graphs.





NVIDIA Nsight

Occupancy Calculator

Compute Capability: 8.6 Threads Per Block: 256

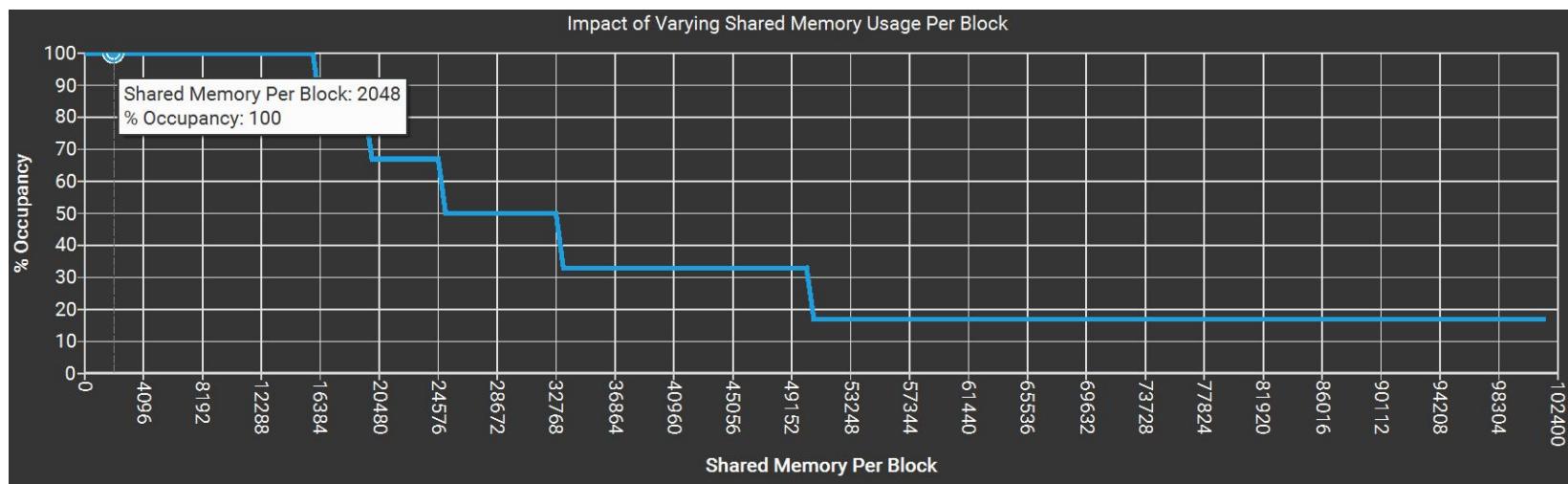
Shared Memory Size Config (bytes): 102400 Registers Per Thread: 32

Global Load Cache Mode: L1+L2 (ca) User Shared Memory Per Block (bytes): 2048

Block Barriers: 1

Apply Automatically

The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM. The launch parameters can be configured with the input widget to the left. This tool also provides various important statistics such as resource allocation, occupancy limiters, GPU Data, etc in the form of tables and graphs.



Глава 2: Умножение матриц

local memory

Все что мы видим на экране – это матрицы. Их умножение – это то, что делает компьютер. А для этого нужно уметь умножать матрицы.

Матрица – это таблица чисел, строк и столбцов. Для умножения матриц нужно, чтобы количество столбцов в первой матрице было равно количеству строк во второй. Результатом умножения будет матрица, количество строк которой – это количество строк в первой матрице, а количество столбцов – это количество столбцов во второй.

Но умножение матриц – это не что иное, как умножение строк на столбцы. Каждый элемент в результирующей матрице получается умножением соответствующих элементов из исходных матриц. Для этого нужно умножить элементы из первой строки на элементы из первой колонны, а затем сложить результаты. Итак, умножение матриц – это умножение строк на столбцы.

Но умножение матриц – это не что иное, как умножение строк на столбцы. Каждый элемент в результирующей матрице получается умножением соответствующих элементов из исходных матриц. Для этого нужно умножить элементы из первой строки на элементы из первой колонны, а затем сложить результаты. Итак, умножение матриц – это умножение строк на столбцы.

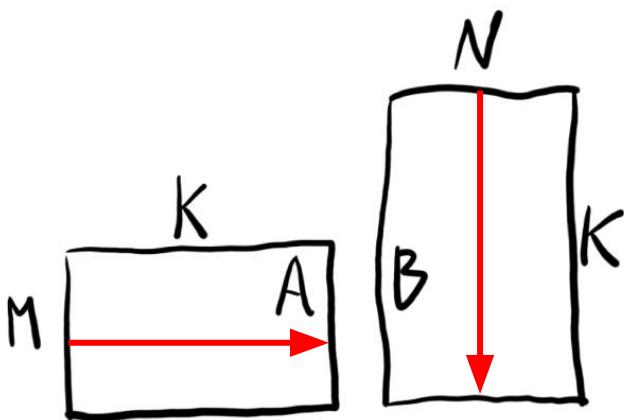
Но умножение матриц – это не что иное, как умножение строк на столбцы. Каждый элемент в результирующей матрице получается умножением соответствующих элементов из исходных матриц. Для этого нужно умножить элементы из первой строки на элементы из первой колонны, а затем сложить результаты. Итак, умножение матриц – это умножение строк на столбцы.

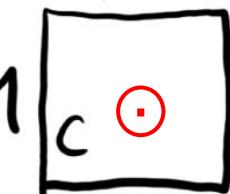
Но умножение матриц – это не что иное, как умножение строк на столбцы. Каждый элемент в результирующей матрице получается умножением соответствующих элементов из исходных матриц. Для этого нужно умножить элементы из первой строки на элементы из первой колонны, а затем сложить результаты. Итак, умножение матриц – это умножение строк на столбцы.

Но умножение матриц – это не что иное, как умножение строк на столбцы. Каждый элемент в результирующей матрице получается умножением соответствующих элементов из исходных матриц. Для этого нужно умножить элементы из первой строки на элементы из первой колонны, а затем сложить результаты. Итак, умножение матриц – это умножение строк на столбцы.

Но умножение матриц – это не что иное, как умножение строк на столбцы. Каждый элемент в результирующей матрице получается умножением соответствующих элементов из исходных матриц. Для этого нужно умножить элементы из первой строки на элементы из первой колонны, а затем сложить результаты. Итак, умножение матриц – это умножение строк на столбцы.

Умножение матриц



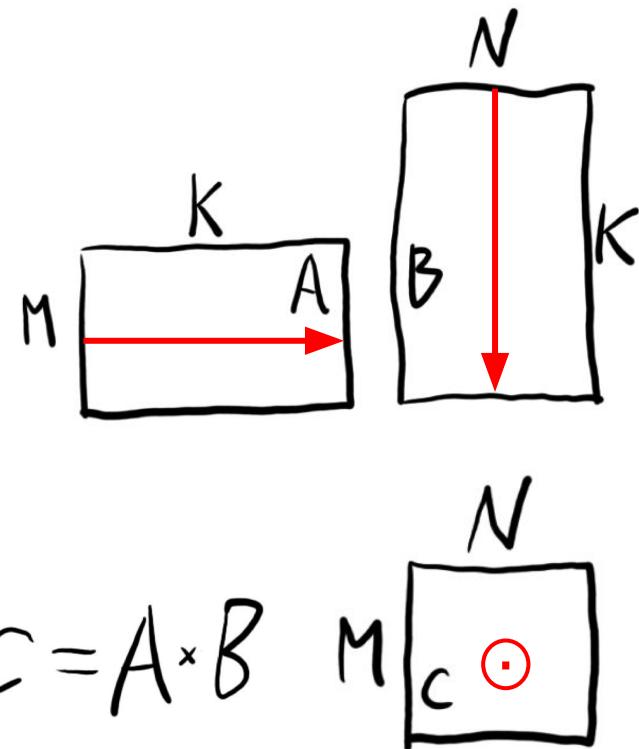
$$C = A \times B \quad M \quad N$$


Below the multiplication equation, there is a square matrix labeled 'C' in the bottom-left corner. In the top-left position of this matrix, there is a red circle containing a white dot, indicating the result of the multiplication at that specific index.

Умножение матриц

Как выглядит:

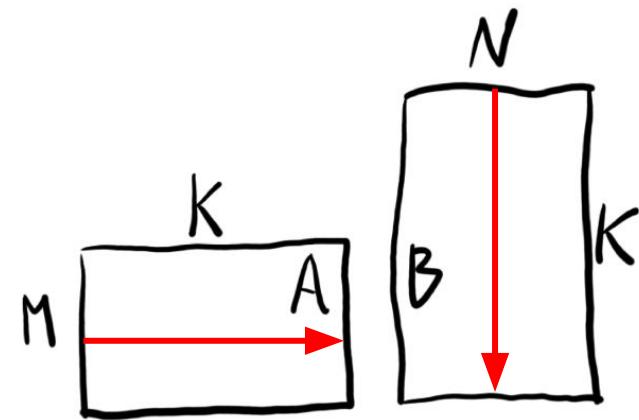
- WorkRange?
- Задача WorkItem?



Умножение матриц

Как выглядит:

- WorkRange?
- Задача WorkItem?
- WorkGroup?



$$C = A \times B \quad M \quad N$$

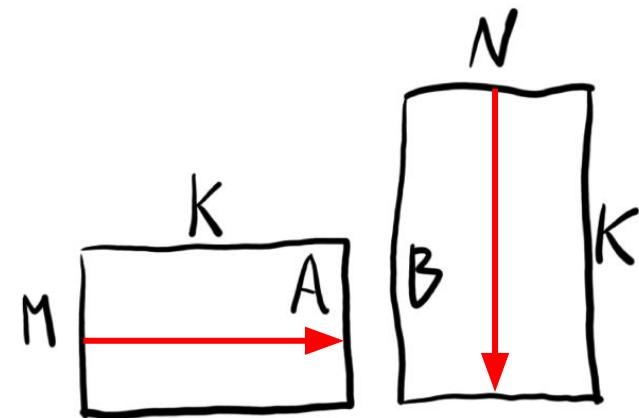
The resulting matrix C is shown as a square with dimensions M x N. The element at the intersection of the first row and first column is highlighted with a red circle containing a dot, representing the result of the first work item.

Умножение матриц

Сколько у нас вычислений?

Сколько у нас чтений/записей данных?

Какая пропорция?



$$C = A \times B \quad M \quad N$$

A square matrix C is shown with a red circle containing a dot at its center. Above the matrix is its height 'N', and to its left is its width 'M'.

Умножение матриц

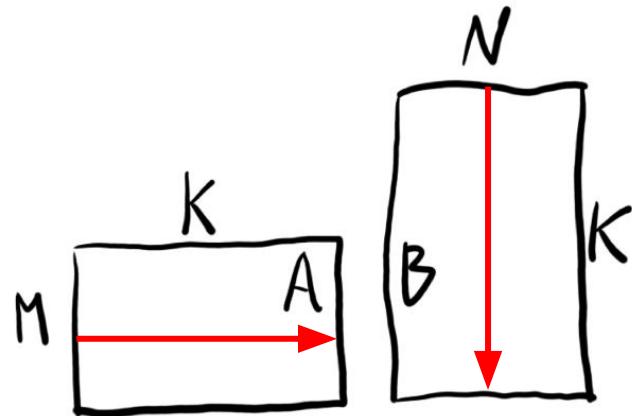
Сколько у нас вычислений?

$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей данных?

$$O(N \cdot M \cdot K)$$

Какая пропорция?



$$C = A \times B \quad M \quad N$$

Умножение матриц

Сколько у нас вычислений?

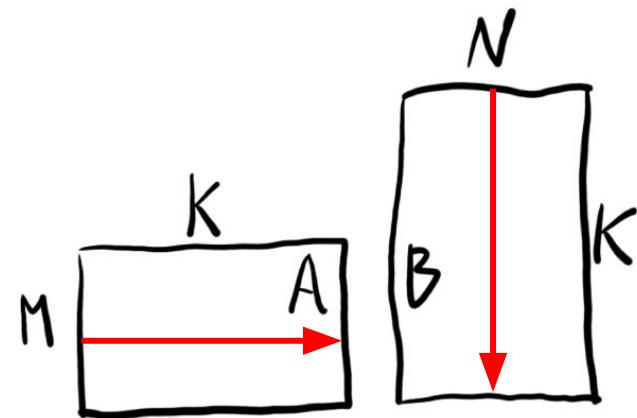
$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей данных?

$$O(N \cdot M \cdot K)$$

Какая пропорция?

1:1 Это хороший результат?



$$C = A \times B \quad M \quad N$$

Умножение матриц

Сколько у нас вычислений?

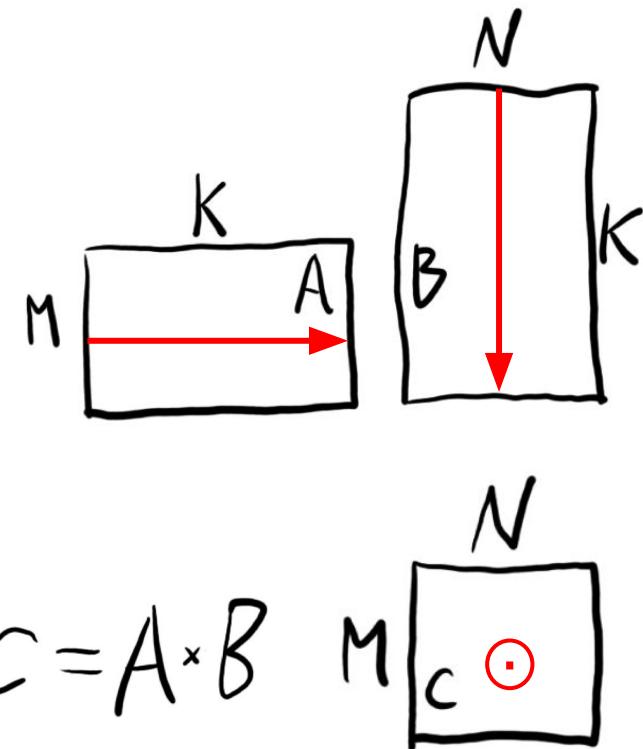
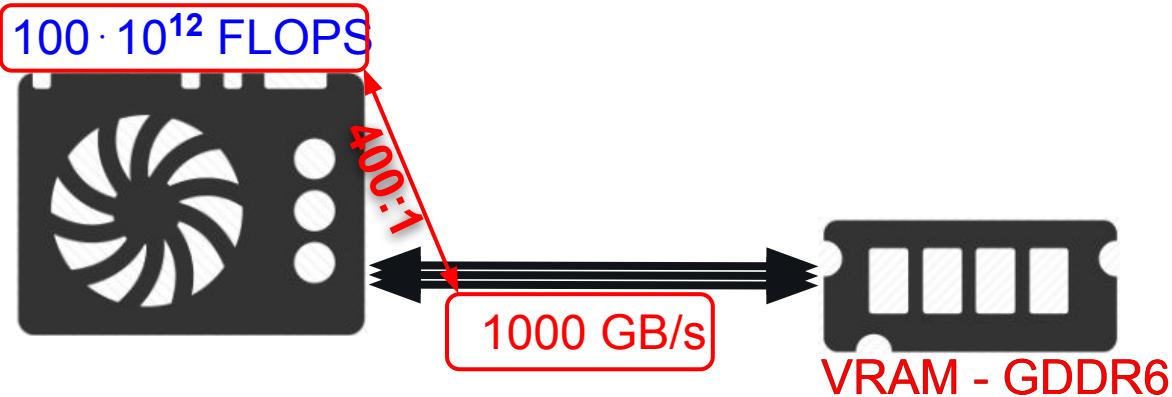
$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей данных?

$$O(N \cdot M \cdot K)$$

Какая пропорция?

1:1 Это хороший результат?



Умножение матриц

Сколько у нас вычислений?

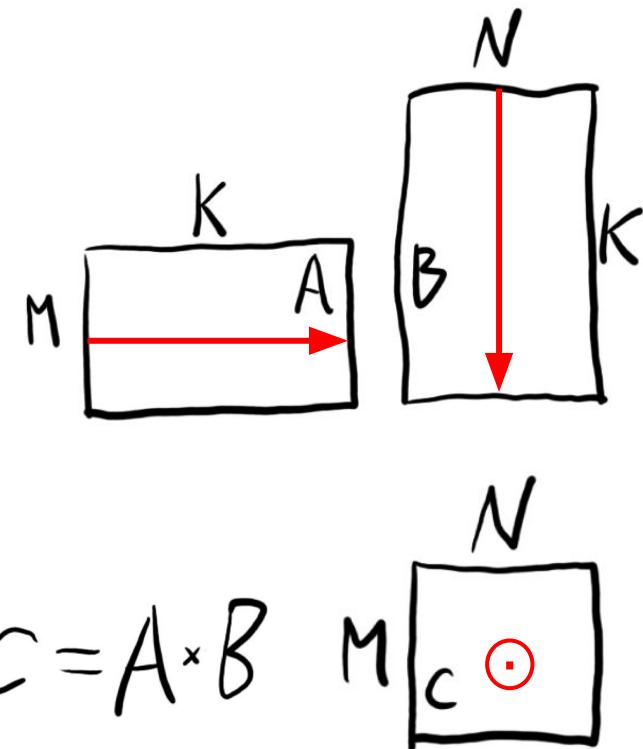
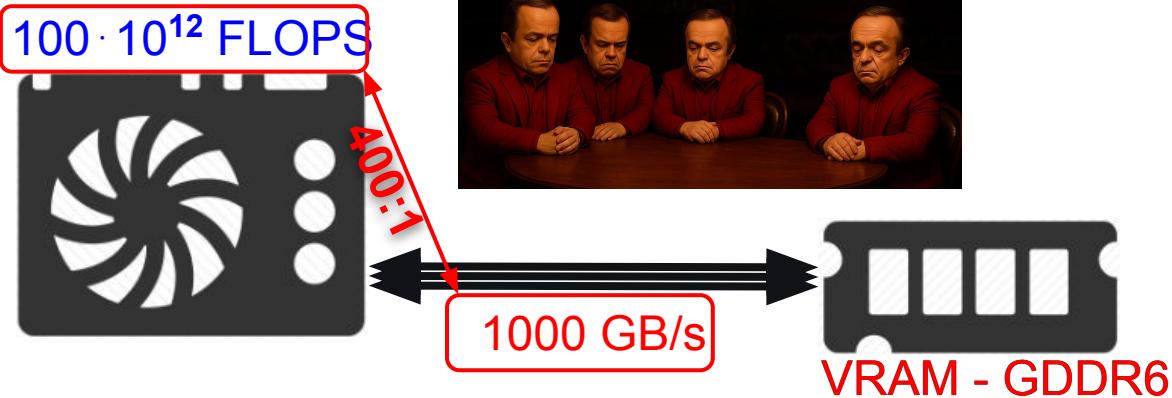
$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей данных?

$$O(N \cdot M \cdot K)$$

Какая пропорция?

1:1 **Memory-bound!**



$$C = A \times B$$

Умножение матриц

Сколько у нас вычислений?

$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей данных?

$$O(N \cdot M \cdot K)$$

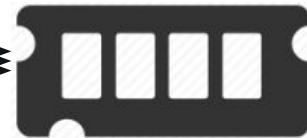
Какая пропорция?

1:1 **Memory-bound!**

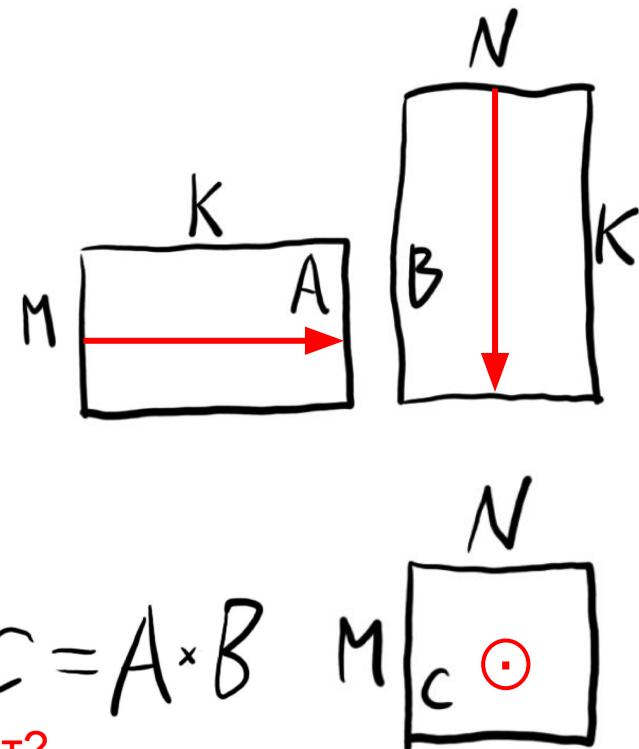
$100 \cdot 10^{12}$ FLOPS



1000 GB/s



VRAM - GDDR6



Как увеличить объем вычислений на считанный байт?

Умножение матриц

Сколько у нас вычислений?

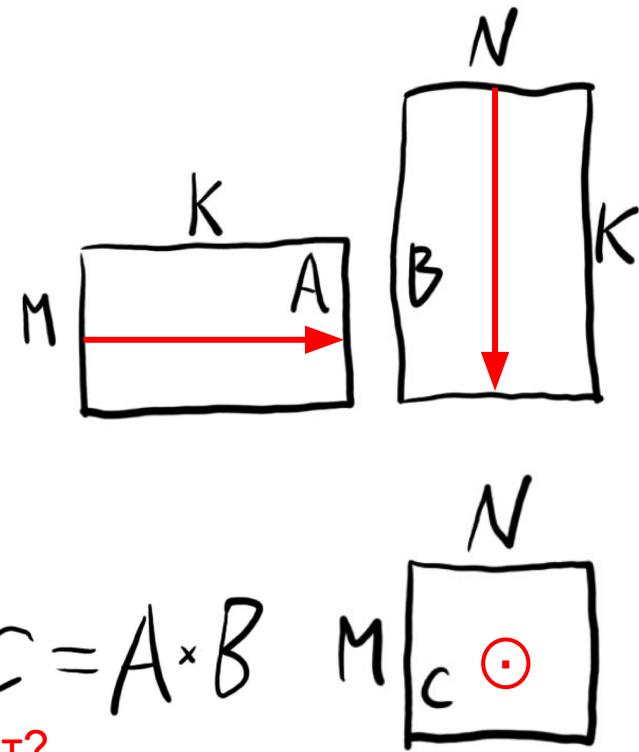
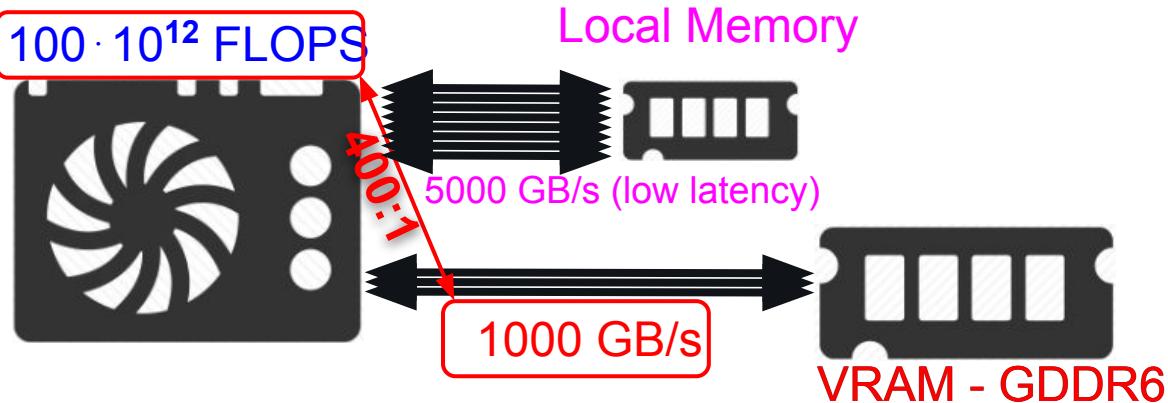
$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей данных?

$$O(N \cdot M \cdot K)$$

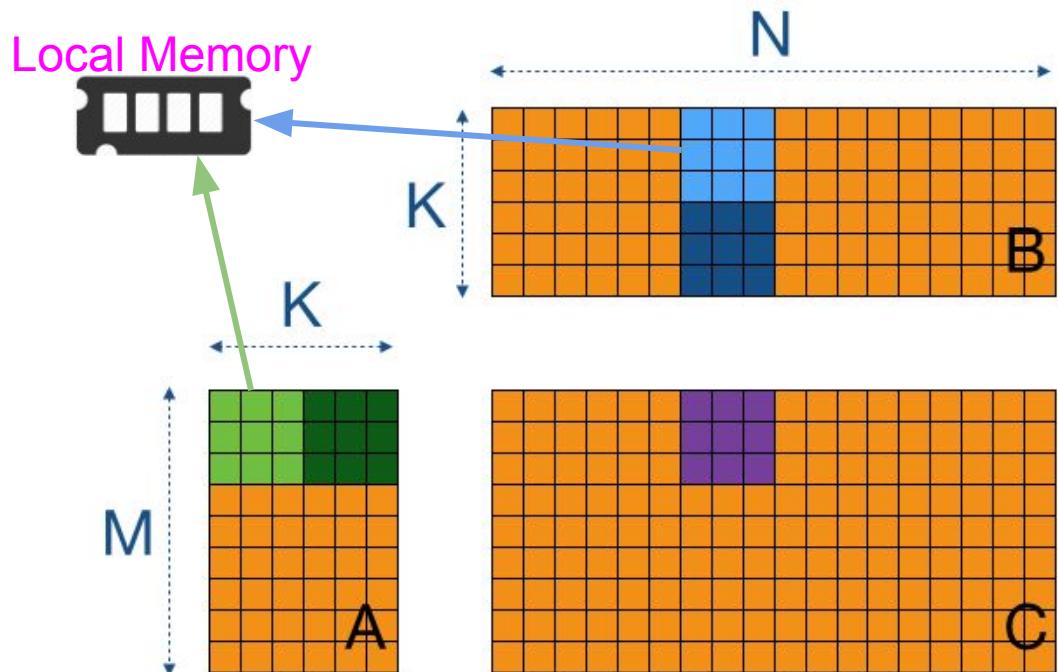
Какая пропорция?

1:1 **Memory-bound!**



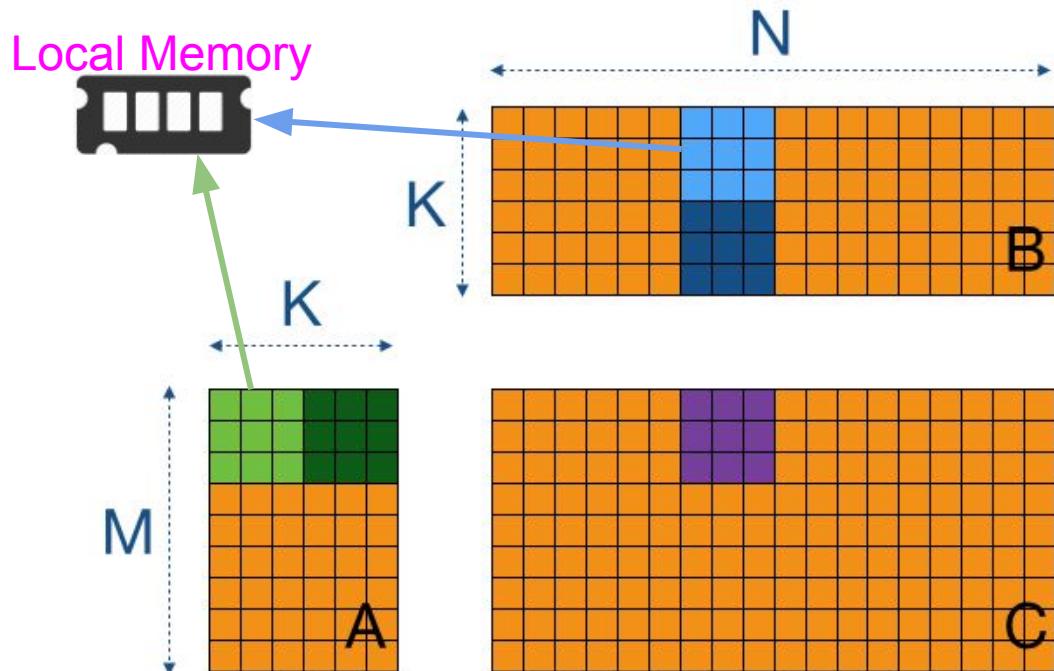
Как увеличить объем вычислений на считанный байт?

Умножение матриц



Умножение матриц

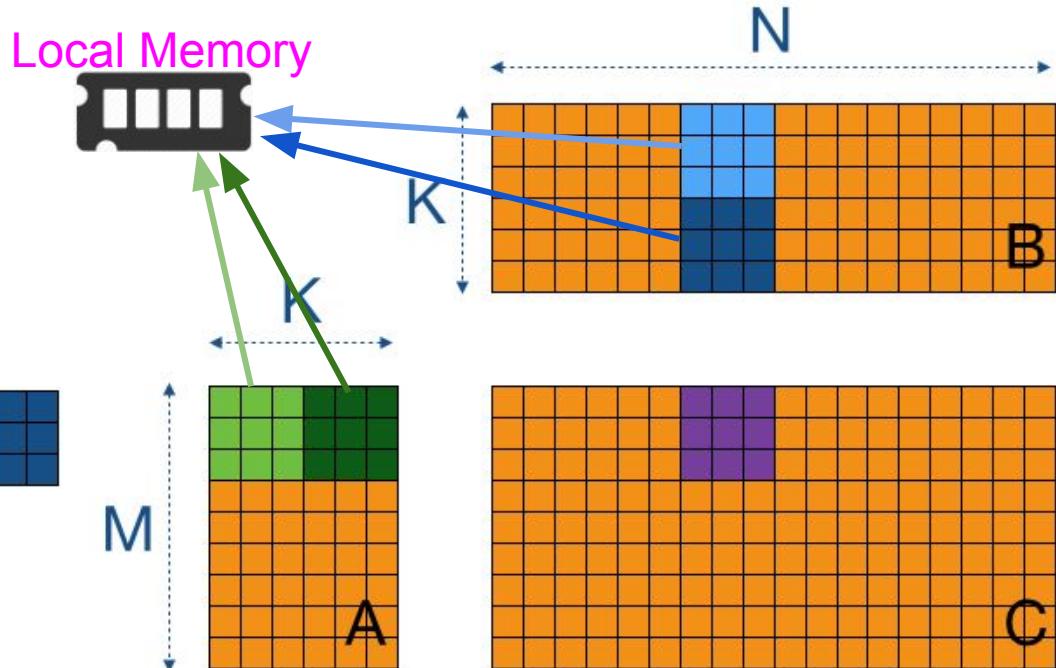
$$\begin{bmatrix} \text{purple} \\ \text{green} \\ \text{blue} \end{bmatrix} = \begin{bmatrix} \text{green} \\ \text{blue} \end{bmatrix} \times \begin{bmatrix} \text{blue} \end{bmatrix} + \dots$$



Умножение матриц

$$\begin{matrix} \text{purple} \\ \text{matrix} \end{matrix} = \begin{matrix} \text{green} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{blue} \\ \text{matrix} \end{matrix} + \begin{matrix} \text{dark green} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{dark blue} \\ \text{matrix} \end{matrix}$$

За счет чего это ускоряет?



Умножение матриц

Сколько у нас вычислений?

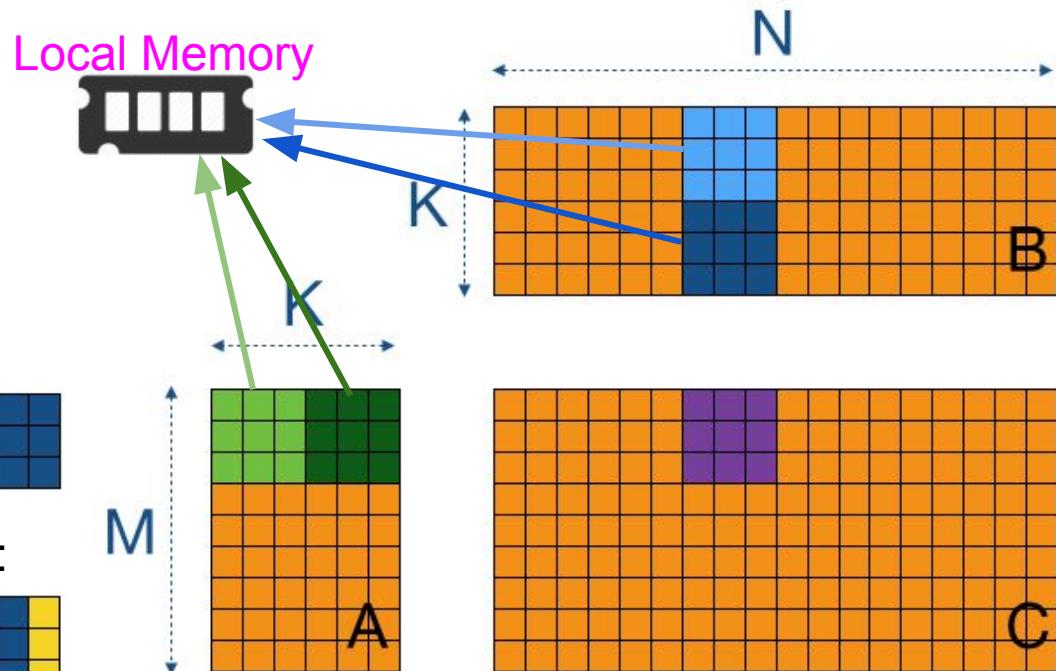
Сколько у нас чтений/записей?

Какая пропорция?

$$\begin{bmatrix} \text{purple} \\ \text{purple} \\ \text{purple} \end{bmatrix} = \begin{bmatrix} \text{green} \\ \text{green} \\ \text{green} \end{bmatrix} \times \begin{bmatrix} \text{blue} \\ \text{blue} \\ \text{blue} \end{bmatrix} + \begin{bmatrix} \text{dark green} \\ \text{dark green} \\ \text{dark green} \end{bmatrix} \times \begin{bmatrix} \text{dark blue} \\ \text{dark blue} \\ \text{dark blue} \end{bmatrix}$$

Переиспользование данных:

$$\left\{ \begin{bmatrix} \text{purple} & \text{yellow} \\ \text{purple} & \text{yellow} \end{bmatrix} \right\}_{32} = \begin{bmatrix} \text{green} & \text{yellow} \\ \text{green} & \text{yellow} \end{bmatrix} \times \begin{bmatrix} \text{blue} & \text{yellow} \\ \text{blue} & \text{yellow} \end{bmatrix} + \begin{bmatrix} \text{dark green} & \text{yellow} \\ \text{dark green} & \text{yellow} \end{bmatrix} \times \begin{bmatrix} \text{dark blue} & \text{yellow} \\ \text{dark blue} & \text{yellow} \end{bmatrix}$$



Умножение матриц

Сколько у нас вычислений?

$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей?

$$O(N \cdot M \cdot K / 32)$$

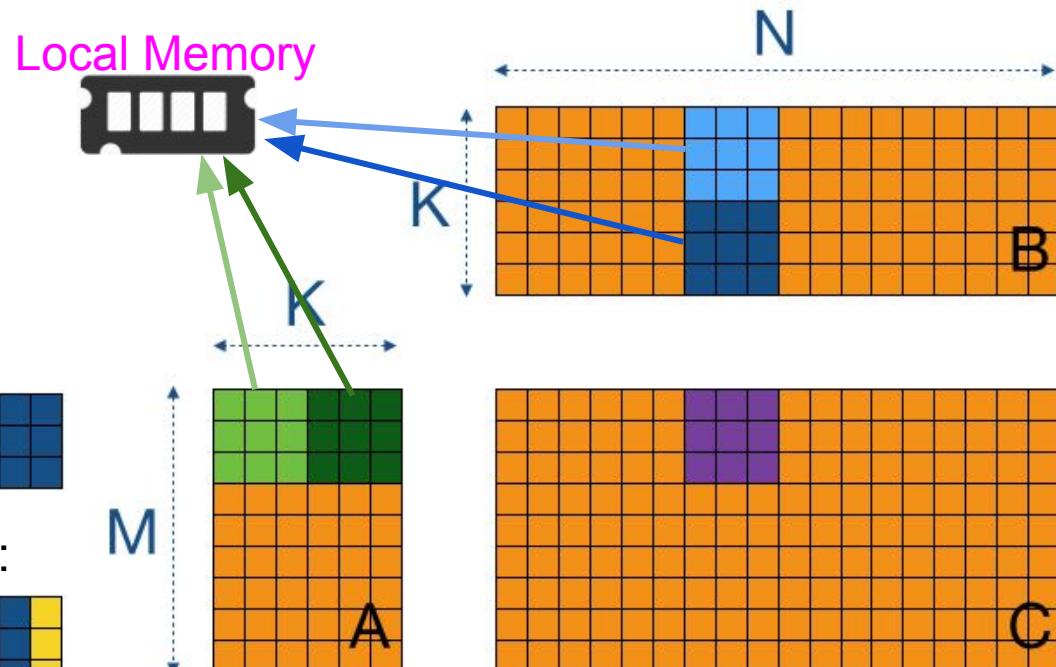
Какая пропорция?

32:1

$$\begin{matrix} \text{purple} \\ \text{matrix} \end{matrix} = \begin{matrix} \text{green} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{blue} \\ \text{matrix} \end{matrix} + \begin{matrix} \text{dark green} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{dark blue} \\ \text{matrix} \end{matrix}$$

Переиспользование данных:

$$\left\{ \begin{matrix} \text{purple} \\ \text{matrix} \end{matrix} \right. = \begin{matrix} \text{yellow} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{blue} \\ \text{matrix} \end{matrix} + \begin{matrix} \text{dark green} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{yellow} \\ \text{matrix} \end{matrix}$$



Умножение матриц

Сколько у нас вычислений?

$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей?

$$O(N \cdot M \cdot K / 32)$$

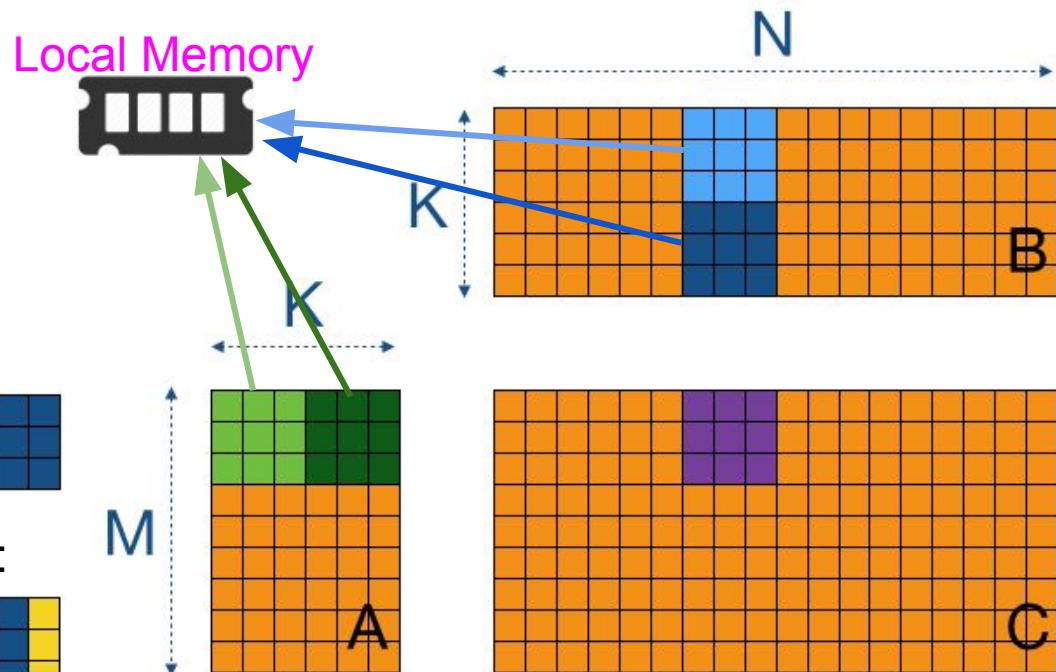
Какая пропорция?

32:1

$$\begin{matrix} \text{purple} \\ \text{matrix} \end{matrix} = \begin{matrix} \text{green} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{blue} \\ \text{matrix} \end{matrix} + \begin{matrix} \text{dark green} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{dark blue} \\ \text{matrix} \end{matrix}$$

Переиспользование данных:

$$\begin{matrix} 32 \\ \text{matrix} \end{matrix} = \begin{matrix} \text{yellow} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{blue} \\ \text{matrix} \end{matrix} + \begin{matrix} \text{dark green} \\ \text{matrix} \end{matrix} \times \begin{matrix} \text{yellow} \\ \text{matrix} \end{matrix}$$



Как пойти еще дальше? Как еще увеличить пропорцию?

Умножение матриц



Неравная битва за гигафлопсы при умножении матриц
(хорошо описанная аналитика, профилирование, оптимизация):

- AMD RDNA3 - <https://seb-v.github.io/optimization/update/2025/01/20/Fast-GPU-Matrix-multiplication.html>
- NVIDIA Kepler - <https://cnugteren.github.io/tutorial/pages/page15.html>
- <https://siboehm.com/articles/22/CUDA-MMM>

Перерыв!

Streaming
Pübiprocessor

Лицензия

© VIDIA

Глава 3: Умножение матриц

Tensor Cores, WMMA



Умножение матриц

Tensor Cores

	VRAM TB/s	FP 32 TFlops	FP 16 TFlops	FP 16 (tensor) TFlops
RTX 3090	0.93	29	29	142
RTX 4090	1.00	73	73	330
RTX 5090	1.79	105	105	419
Tesla H100	3.35	67	268 (4:1)	990 (16:1)



Умножение матриц

Tensor Cores

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} + \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} = \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 or FP32

4x4



Умножение матриц

Tensor Cores

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}_{\text{FP16 or FP32}} \times \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}_{\text{FP16 or FP32}}$$

4x4



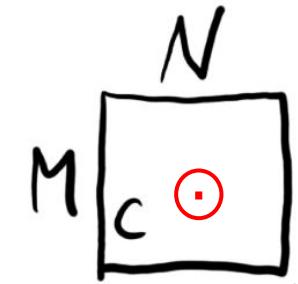
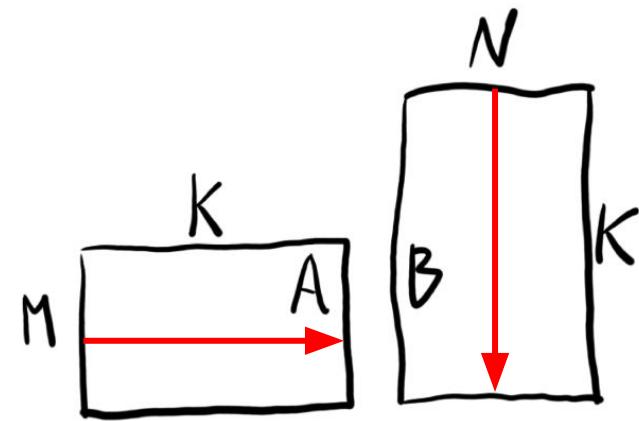
Tensor Cores (CUDA kernel)

$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,\dots} & A_{0,15} \\ A_{1,0} & A_{1,1} & A_{1,\dots} & A_{1,15} \\ A_{\dots,0} & A_{\dots,1} & A_{\dots,\dots} & A_{\dots,15} \\ A_{15,0} & A_{15,1} & A_{15,\dots} & A_{15,15} \end{pmatrix}_{\text{FP16 or FP32}} \times \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,\dots} & B_{0,15} \\ B_{1,0} & B_{1,1} & B_{1,\dots} & B_{1,15} \\ B_{\dots,0} & B_{\dots,1} & B_{\dots,\dots} & B_{\dots,15} \\ B_{15,0} & B_{15,1} & B_{15,\dots} & B_{15,15} \end{pmatrix}_{\text{FP16}} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,\dots} & C_{0,15} \\ C_{1,0} & C_{1,1} & C_{1,\dots} & C_{1,15} \\ C_{\dots,0} & C_{\dots,1} & C_{\dots,\dots} & C_{\dots,15} \\ C_{15,0} & C_{15,1} & C_{15,\dots} & C_{15,15} \end{pmatrix}_{\text{FP16 or FP32}}$$

16x16

```
69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)  
75 __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
```

$$C = \alpha A \times B + \beta C$$



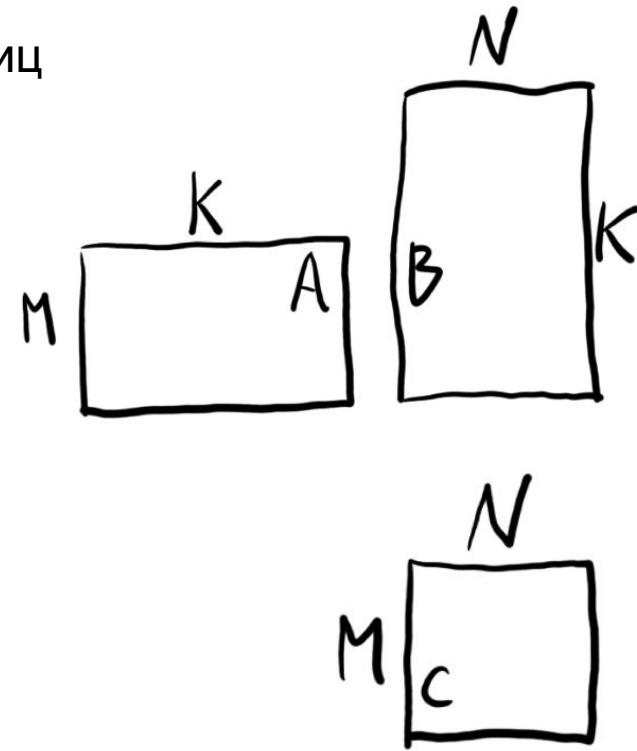
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

```

Общие на весь warp 16x16 фрагменты матриц
WMMA = Warp Matrix Multiply-Accumulate

$$C = \alpha A \times B + \beta C$$



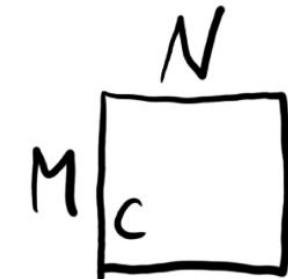
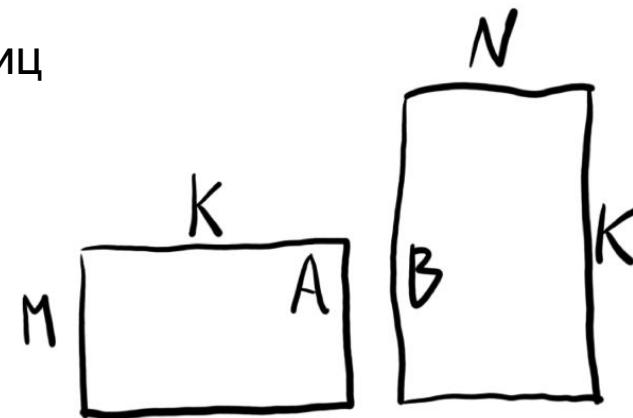
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85 // Declare the fragments
86 wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87 wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

```

Общие на весь warp 16x16 фрагменты матриц
WMMA = Warp Matrix Multiply-Accumulate

$$C = \alpha A \times B + \beta C$$



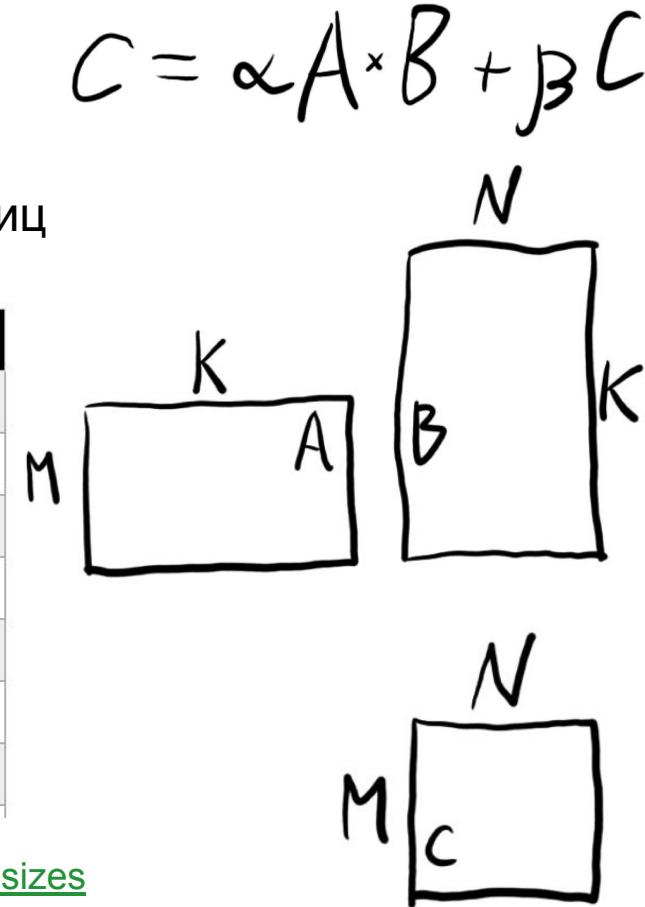
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

```

Общие на весь warp 16x16 фрагменты матриц
WMMA = Warp Matrix Multiply-Accumulate

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
<u>half</u>	<u>half</u>	float	16x16x16
half	half	float	32x8x16
half	half	float	8x32x16
half	half	<u>half</u>	16x16x16
half	half	<u>half</u>	32x8x16
half	half	<u>half</u>	8x32x16
unsigned char	unsigned char	int	16x16x16
...



<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#wmma-type-sizes>

<https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu>

```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

```

Общие на весь warp 16x16 фрагменты матриц
WMMA = Warp Matrix Multiply-Accumulate

Matrix A	Matrix B	Accumulator	Matrix Size (m-n-k)
__half	__half	float	16x16x16
__half	__half	float	32x8x16
__half	__half	float	8x32x16
__half	__half	__half	16x16x16
__half	__half	__half	32x8x16
__half	__half	__half	8x32x16
unsigned char	unsigned char	int	16x16x16
...



Почему А и В - half?
 Но С и аккумулятор - float?

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/#wmma-type-sizes>

<https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu>

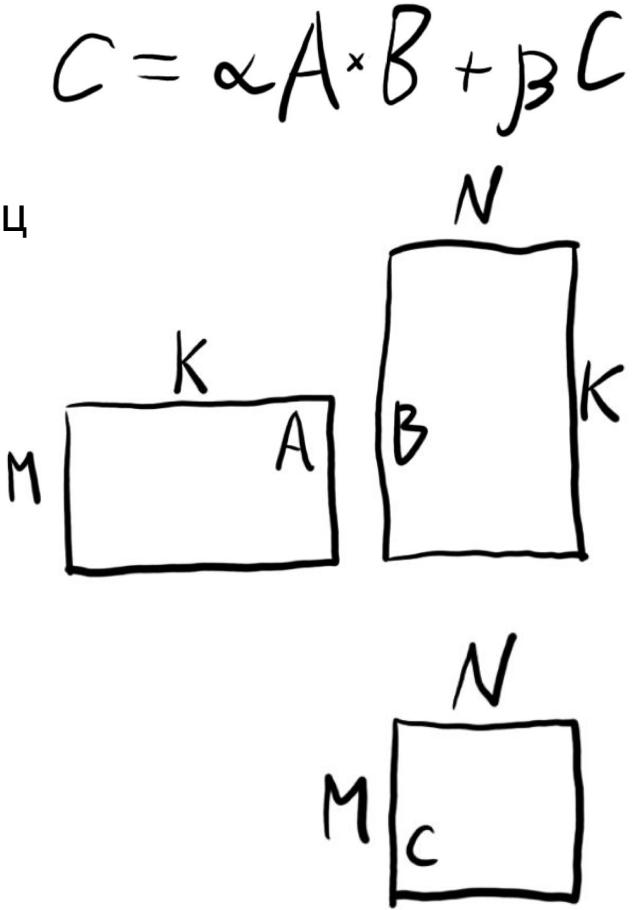
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85 // Declare the fragments
86 wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87 wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

```

Общие на весь warp 16x16 фрагменты матриц
WMMA = Warp Matrix Multiply-Accumulate

Почему А и В - half?
 Но С и аккумулятор - float?
 Подсказка:
 - А и В перемножаются
 - аккумулятор суммируется



```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

```

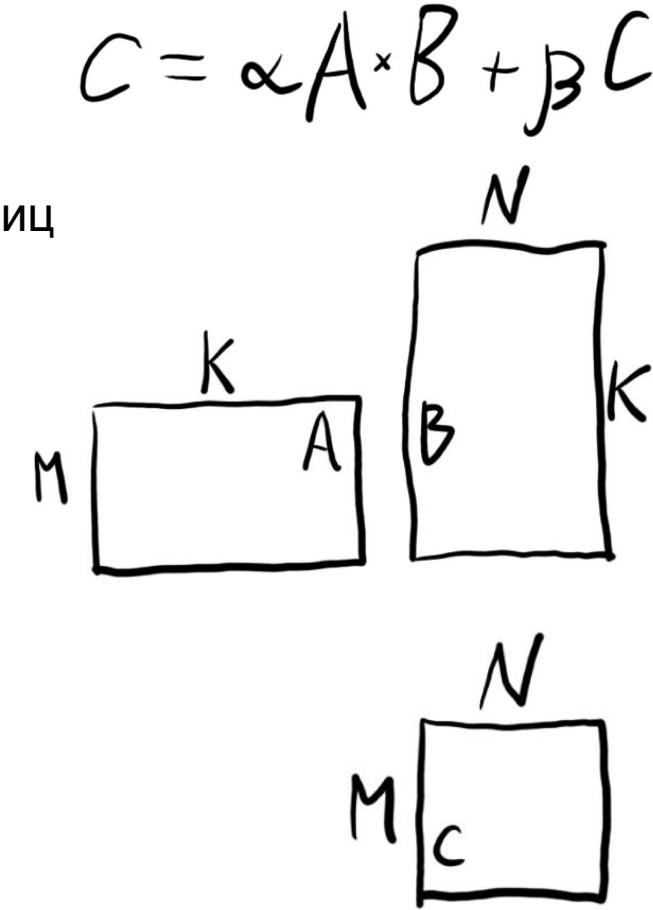
Общие на весь warp 16x16 фрагменты матриц
WMMA = Warp Matrix Multiply-Accumulate

Почему А и В - half?
 Но С и аккумулятор - float?

Что будет если
сложить огромное
 и малое число?



$$+ (1 + \text{mantissa}) \times 2^{(\text{exponent} - 127)}$$



```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

```

Общие на весь warp 16x16 фрагменты матриц
WMMA = Warp Matrix Multiply-Accumulate

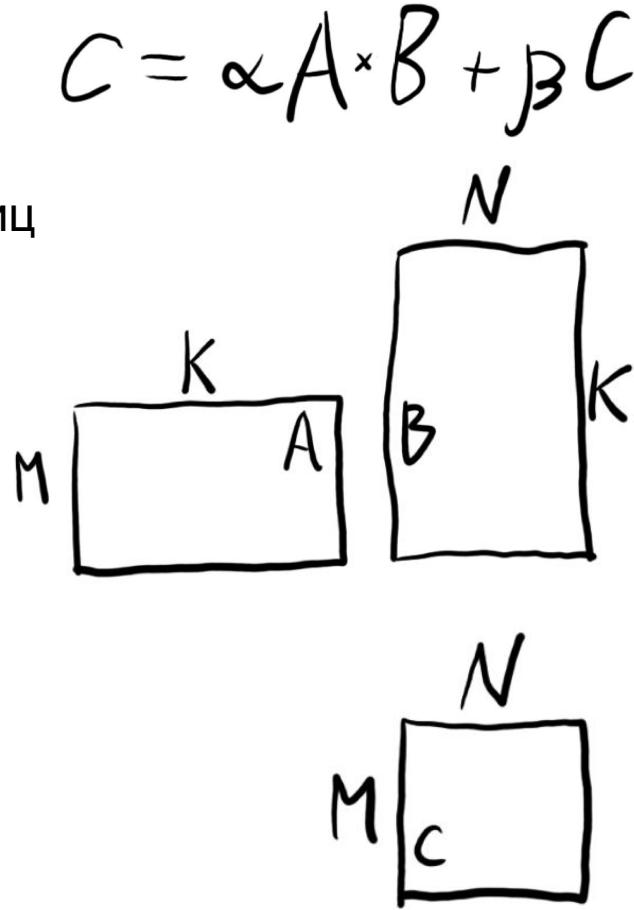
Почему А и В - half?
 Но С и аккумулятор - float?

Что будет если
сложить огромное
 и малое число?



Что будет если
умножить огромное
 и малое число?

$$\pm (1 + \text{mantissa}) \times 2^{(\text{exponent} - 127)}$$



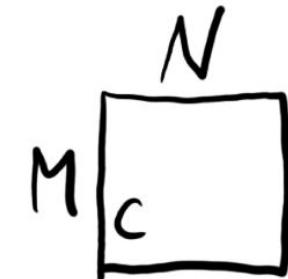
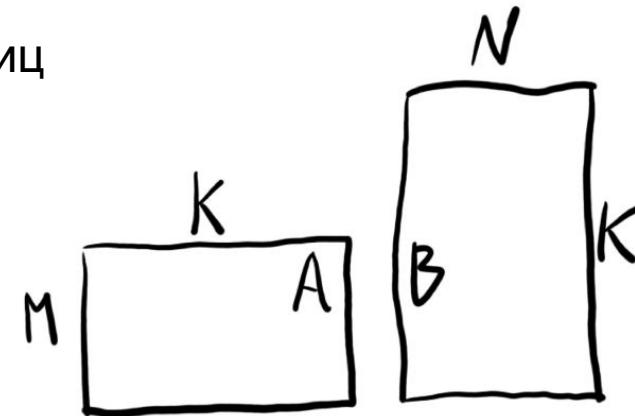
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_majorwmma::col_major

```

Общие на весь warp 16x16 фрагменты матриц
WMMA = Warp Matrix Multiply-Accumulate

$$C = \alpha A \times B + \beta C$$



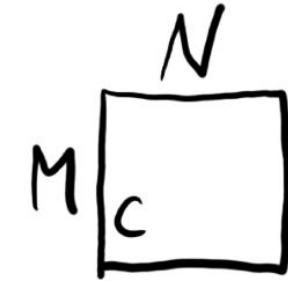
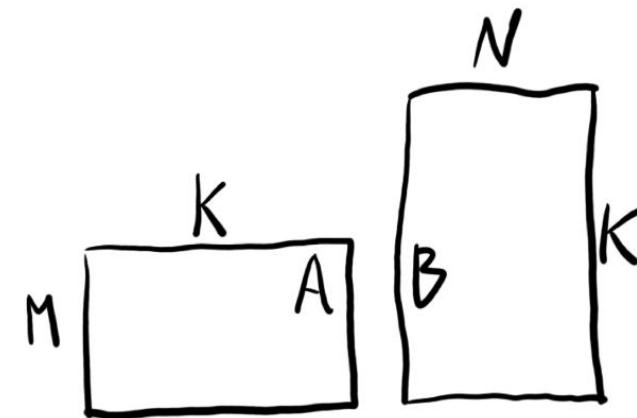
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85 // Declare the fragments
86 wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87 wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
91 wmma::fill_fragment(acc_frag, 0.0f);

```

 16x16
acc_frag

$$C = \alpha A \times B + \beta C$$



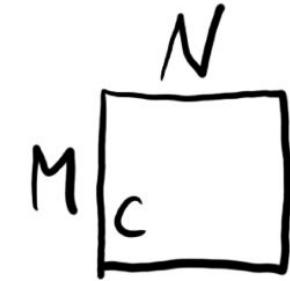
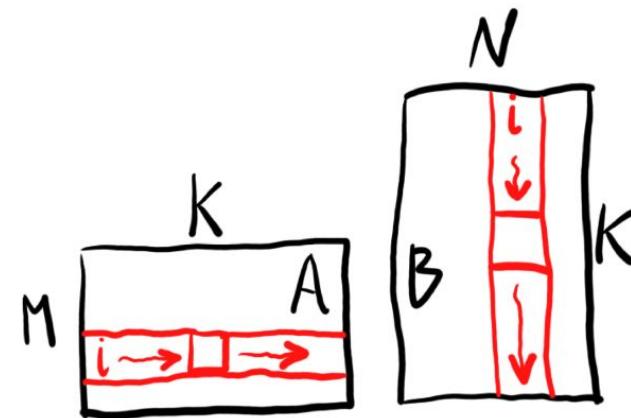
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85 // Declare the fragments
86 wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87 wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
91 wmma::fill_fragment(acc_frag, 0.0f);
94 for (int i = 0; i < K; i += WMMA_K) {

```

16x16
acc_frag

$$C = \alpha A \times B + \beta C$$



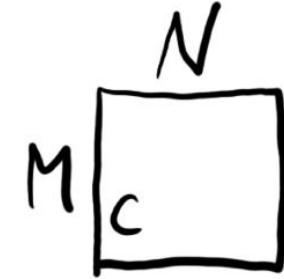
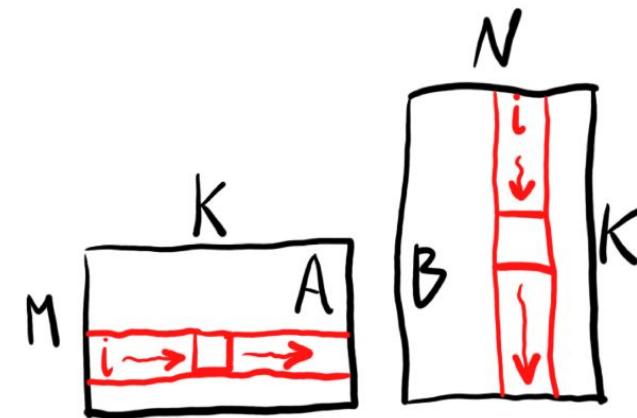
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85 // Declare the fragments
86 wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87 wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89 wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
91 wmma::fill_fragment(acc_frag, 0.0f);
94 for (int i = 0; i < K; i += WMMA_K) { acc_frag
95     int aRow = warpM * WMMA_M;
96     int aCol = i;
97
98     int bRow = i;
99     int bCol = warpN * WMMA_N;
100
101    // Bounds checking
102    if (aRow < M && aCol < K && bRow < K && bCol < N) {
103        // Load the inputs
104        wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);
105        wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);

```

16x16
acc_frag

$$C = \alpha A \times B + \beta C$$



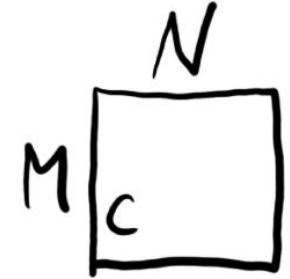
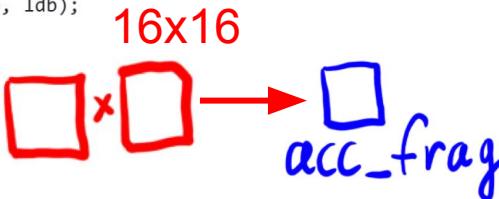
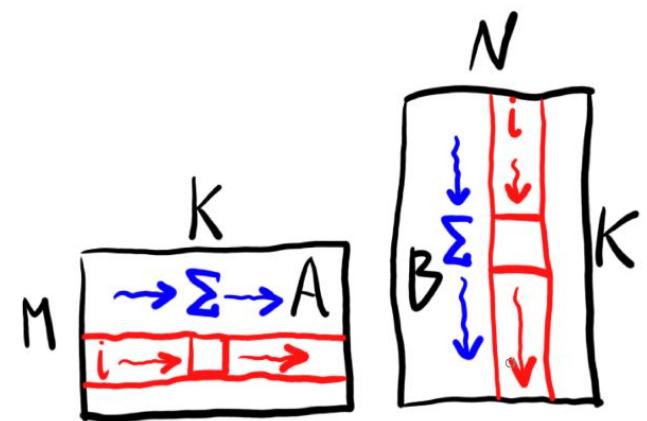
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
91     wmma::fill_fragment(acc_frag, 0.0f);
94     for (int i = 0; i < K; i += WMMA_K) { acc_frag
95         int aRow = warpM * WMMA_M;
96         int aCol = i;
97
98         int bRow = i;
99         int bCol = warpN * WMMA_N;
100
101        // Bounds checking
102        if (aRow < M && aCol < K && bRow < K && bCol < N) {
103            // Load the inputs
104            wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);
105            wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);
106
107            // Perform the matrix multiplication
108            wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
109        }
110    }
111 }

```

16x16

$$C = \alpha A \times B + \beta C$$



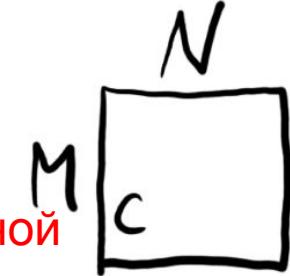
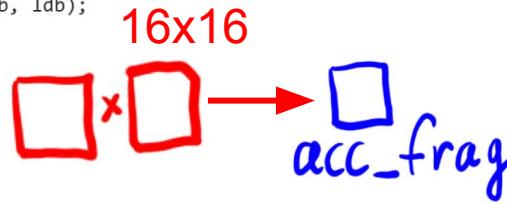
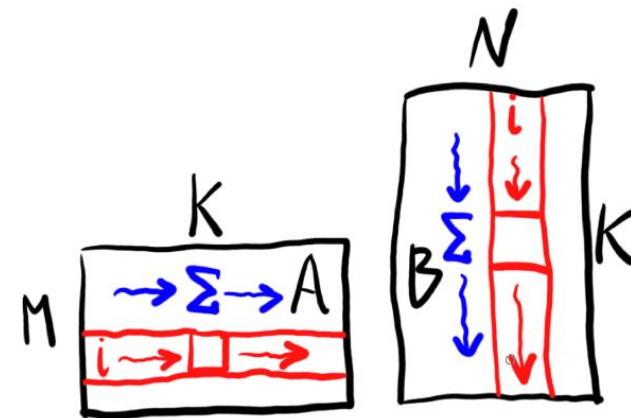
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
91     wmma::fill_fragment(acc_frag, 0.0f);
94     for (int i = 0; i < K; i += WMMA_K) { acc_frag
95         int aRow = warpM * WMMA_M;
96         int aCol = i;
97
98         int bRow = i;
99         int bCol = warpN * WMMA_N;
100
101        // Bounds checking
102        if (aRow < M && aCol < K && bRow < K && bCol < N) {
103            // Load the inputs
104            wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);
105            wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);
106
107            // Perform the matrix multiplication
108            wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
109        }
110    }

```

16x16
acc_frag

$$C = \alpha A \times B + \beta C$$



Чем этот код отличается от классического умножения в локальной памяти?

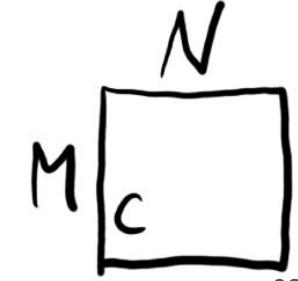
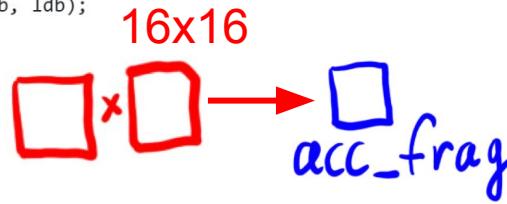
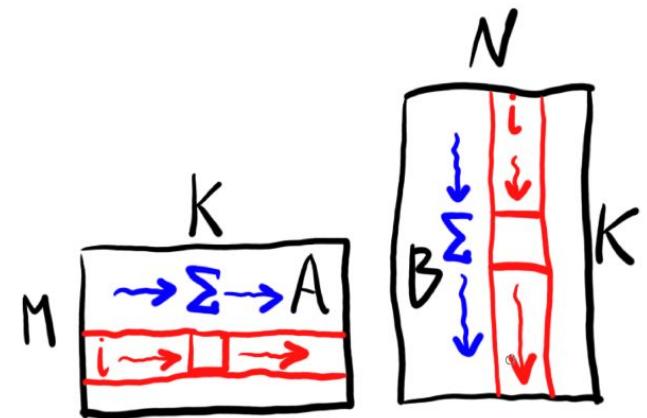
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
91     wmma::fill_fragment(acc_frag, 0.0f);
94     for (int i = 0; i < K; i += WMMA_K) { acc_frag
95         int aRow = warpM * WMMA_M;
96         int aCol = i;
97
98         int bRow = i;
99         int bCol = warpN * WMMA_N;
100
101        // Bounds checking
102        if (aRow < M && aCol < K && bRow < K && bCol < N) {
103            // Load the inputs
104            wmma::load_matrix_sync(a_frag, a + aRow + aCol * lda, lda);
105            wmma::load_matrix_sync(b_frag, b + bRow + bCol * ldb, ldb);
106
107            // Perform the matrix multiplication
108            wmma::mma_sync(acc_frag, a_frag, b_frag, acc_frag);
109        }
110    }

```

16x16

$$C = \alpha A \times B + \beta C$$



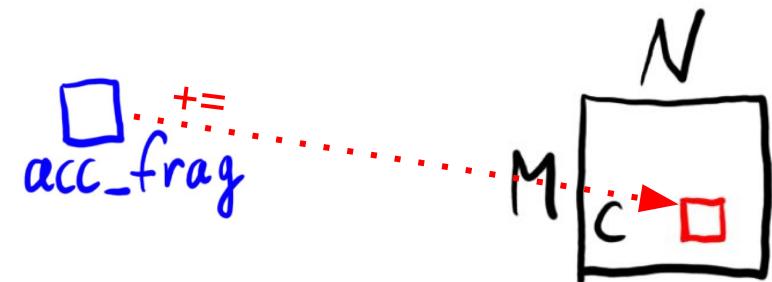
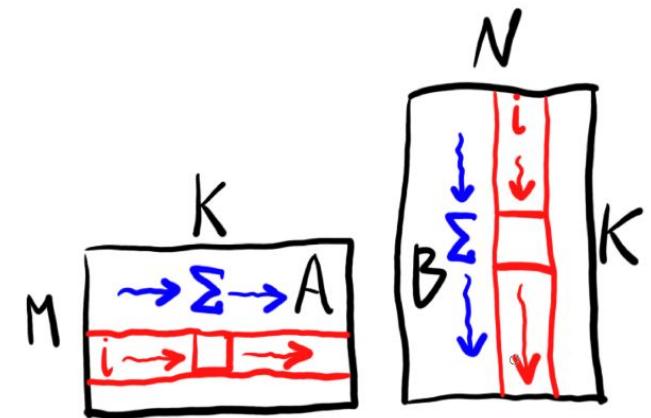
Что нам осталось сделать?

```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
90
91     .....

```

$$C = \alpha A \times B + \beta C$$



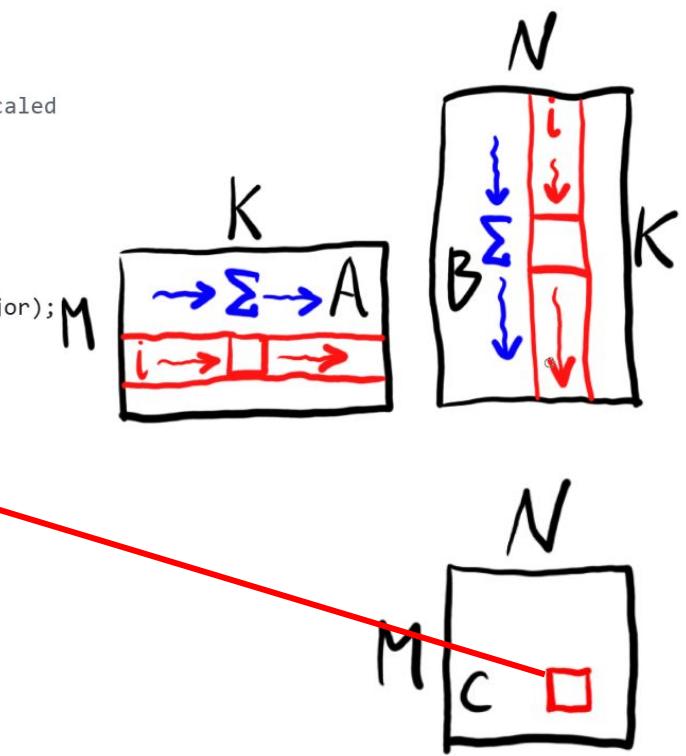
```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;

.....
113     // Load in the current value of c, scale it by beta, and add this our result scaled
114     int cRow = warpM * WMMA_M;
115     int cCol = warpN * WMMA_N;

117     if (cRow < M && cCol < N) {
118         wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc, ldc, wmma::mem_col_major); ←
```

$$C = \alpha A \times B + \beta C$$

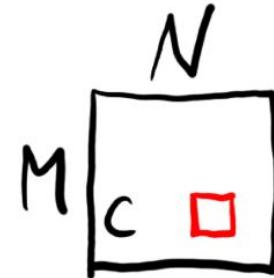
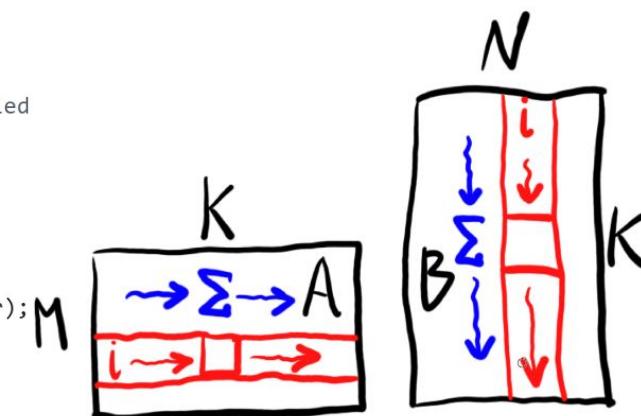


```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
90
91     .....
92
113     // Load in the current value of c, scale it by beta, and add this our result scaled
114     int cRow = warpM * WMMA_M;
115     int cCol = warpN * WMMA_N;
116
117     if (cRow < M && cCol < N) {
118         wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc, ldc, wmma::mem_col_major);
119
120 #pragma unroll
121     for(int i=0; i < c_frag.num_elements; i++) {
122         c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
123     }
}

```

$$C = \alpha A \times B + \beta C$$

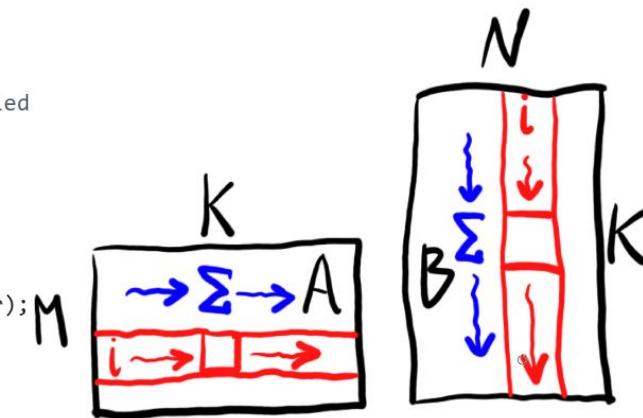


```

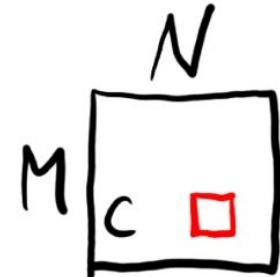
69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
90
91     .....
92
113     // Load in the current value of c, scale it by beta, and add this our result scaled
114     int cRow = warpM * WMMA_M;
115     int cCol = warpN * WMMA_N;
116
117     if (cRow < M && cCol < N) {
118         wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc, ldc, wmma::mem_col_major);
119
120 #pragma unroll
121     for(int i=0; i < c_frag.num_elements; i++) {
122         c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
123     }
}

```

$$C = \alpha A \times B + \beta C$$



А не будет тормозить?
Почему не спец. функция в железе?

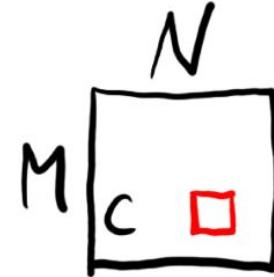
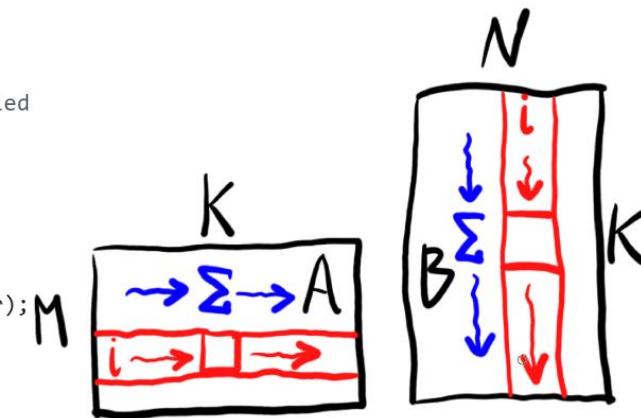


```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
90
91     .....
92
113     // Load in the current value of c, scale it by beta, and add this our result scaled
114     int cRow = warpM * WMMA_M;
115     int cCol = warpN * WMMA_N;
116
117     if (cRow < M && cCol < N) {
118         wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc, ldc, wmma::mem_col_major);
119
120 #pragma unroll
121     for(int i=0; i < c_frag.num_elements; i++) {
122         c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
123     }
}

```

$$C = \alpha A \times B + \beta C$$



А не будет тормозить?

Почему не спец. функция в железе?

А почему бы не добавить к С в отдельном кернеле?

```

69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments

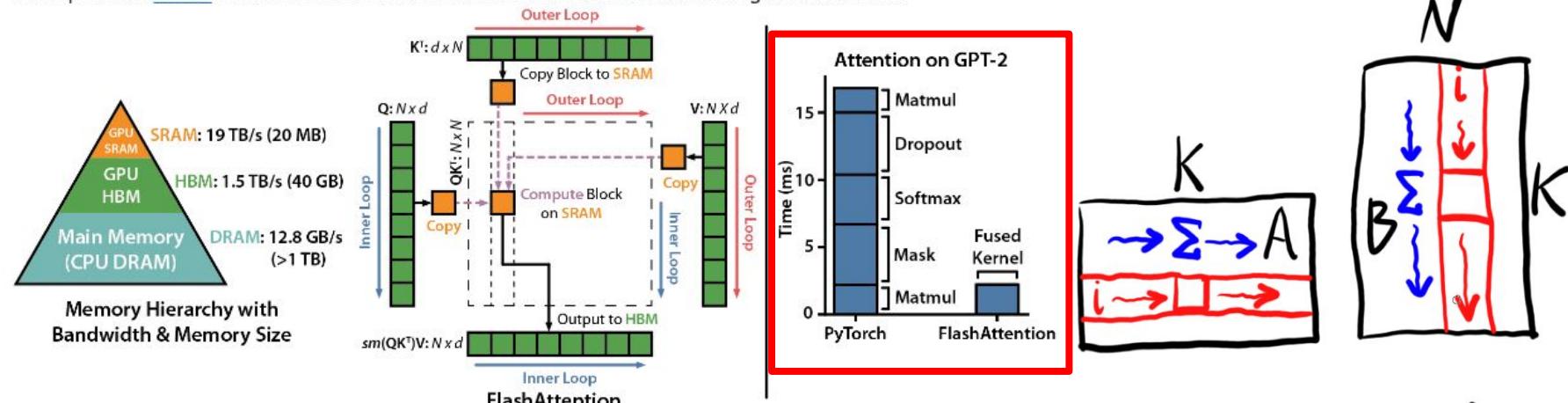
```

FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

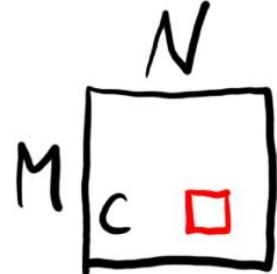
Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, Christopher Ré

Paper: <https://arxiv.org/abs/2205.14135> <https://github.com/Dao-AILab/flash-attention>

IEEE Spectrum [article](#) about our submission to the MLPerf 2.0 benchmark using FlashAttention.



А не будет тормозить?
Почему не спец. функция в железе?
А почему бы не добавить к С в отдельном кернеле?

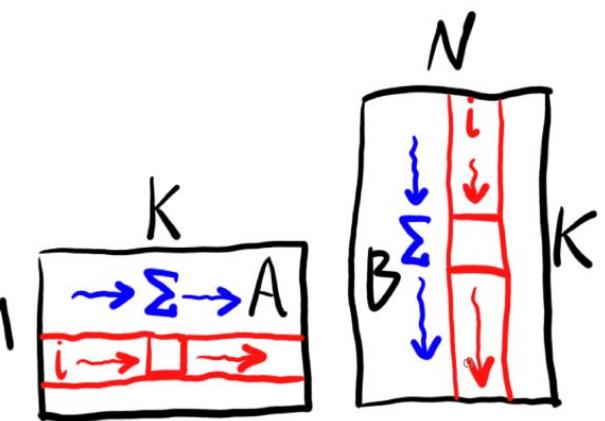


```

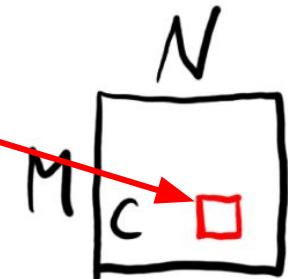
69 // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75 _global_ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85     // Declare the fragments
86     wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87     wmma::fragment<wmma::matrix_b, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> b_frag;
88     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> acc_frag;
89     wmma::fragment<wmma::accumulator, WMMA_M, WMMA_N, WMMA_K, float> c_frag;
90
91     .....
92
113     // Load in the current value of c, scale it by beta, and add this our result scaled
114     int cRow = warpM * WMMA_M;
115     int cCol = warpN * WMMA_N;
116
117     if (cRow < M && cCol < N) {
118         wmma::load_matrix_sync(c_frag, c + cRow + cCol * ldc, ldc, wmma::mem_col_major);
119
120 #pragma unroll
121         for(int i=0; i < c_frag.num_elements; i++) {
122             c_frag.x[i] = alpha * acc_frag.x[i] + beta * c_frag.x[i];
123         }
124
125         // Store the output
126         wmma::store_matrix_sync(c + cRow + cCol * ldc, c_frag, ldc, wmma::mem_col_major);
127     }
128 }

```

$$C = \alpha A \times B + \beta C$$



acc_frag



Умножение матриц - Tensor Cores, WMMA

Сколько у нас вычислений?

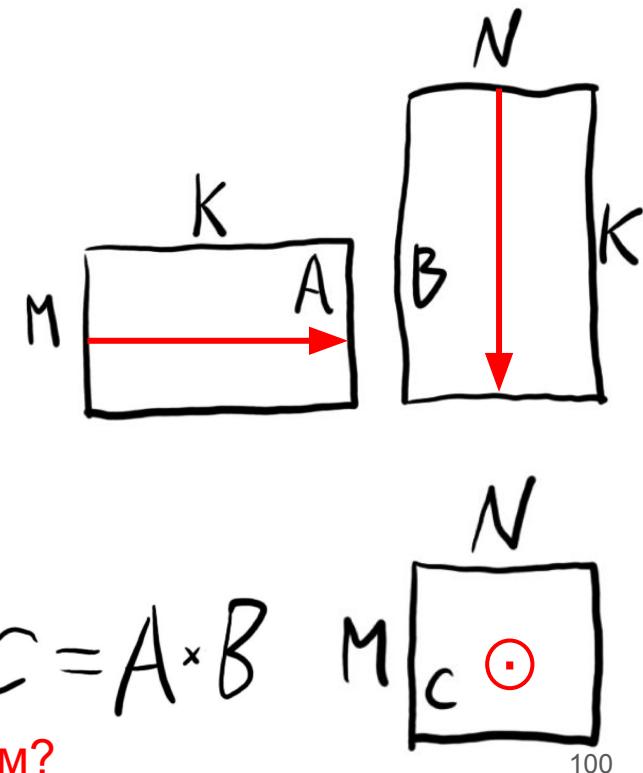
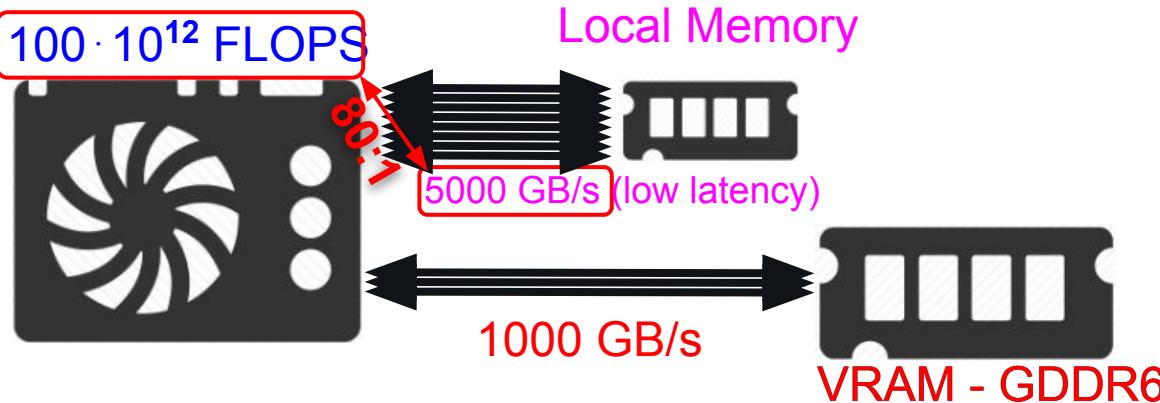
$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей данных?

$$O(N \cdot M \cdot K)$$

Какая пропорция?

1:1 **Memory-bound!**



Какая у нас теперь пропорция вычислений к чтениям?

Умножение матриц - Tensor Cores, WMMA

Сколько у нас вычислений?

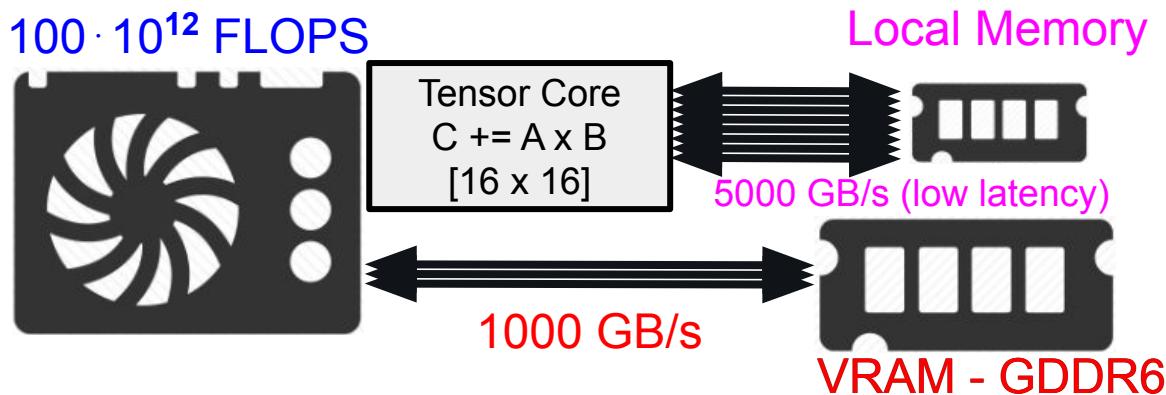
$$O(N \cdot M \cdot K)$$

Сколько у нас чтений/записей данных?

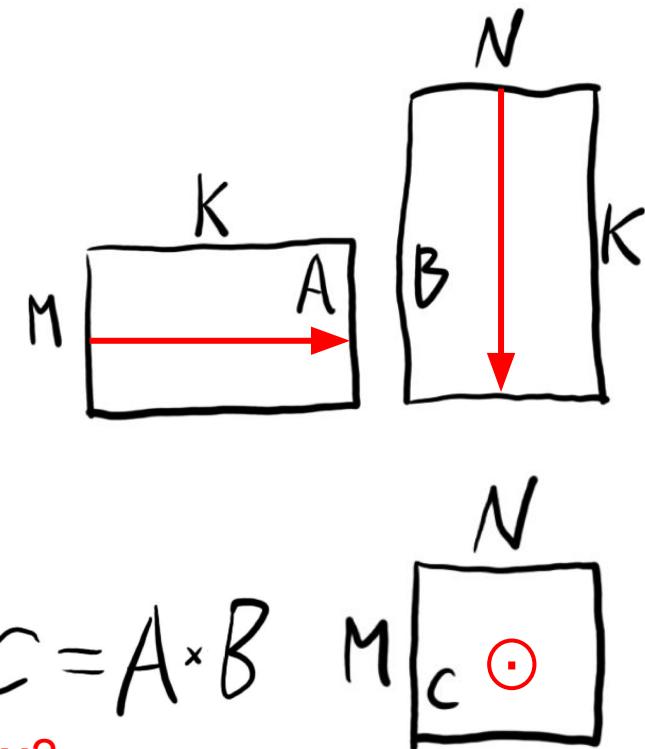
$$O(N \cdot M \cdot K)$$

Какая пропорция?

1:1 **Memory-bound!**



Какая у нас теперь пропорция вычислений к чтениям?



Умножение матриц

Сколько у нас вычислений?

$$O(N \cdot M \cdot K)$$

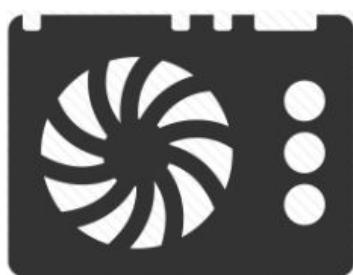
Сколько у нас чтений/записей данных?

$$O(N \cdot M \cdot K)$$

Какая пропорция?

1:1 **Memory-bound!**

$100 \cdot 10^{12}$ FLOPS



Tensor Core
 $C += A \times B$
[16 x 16]

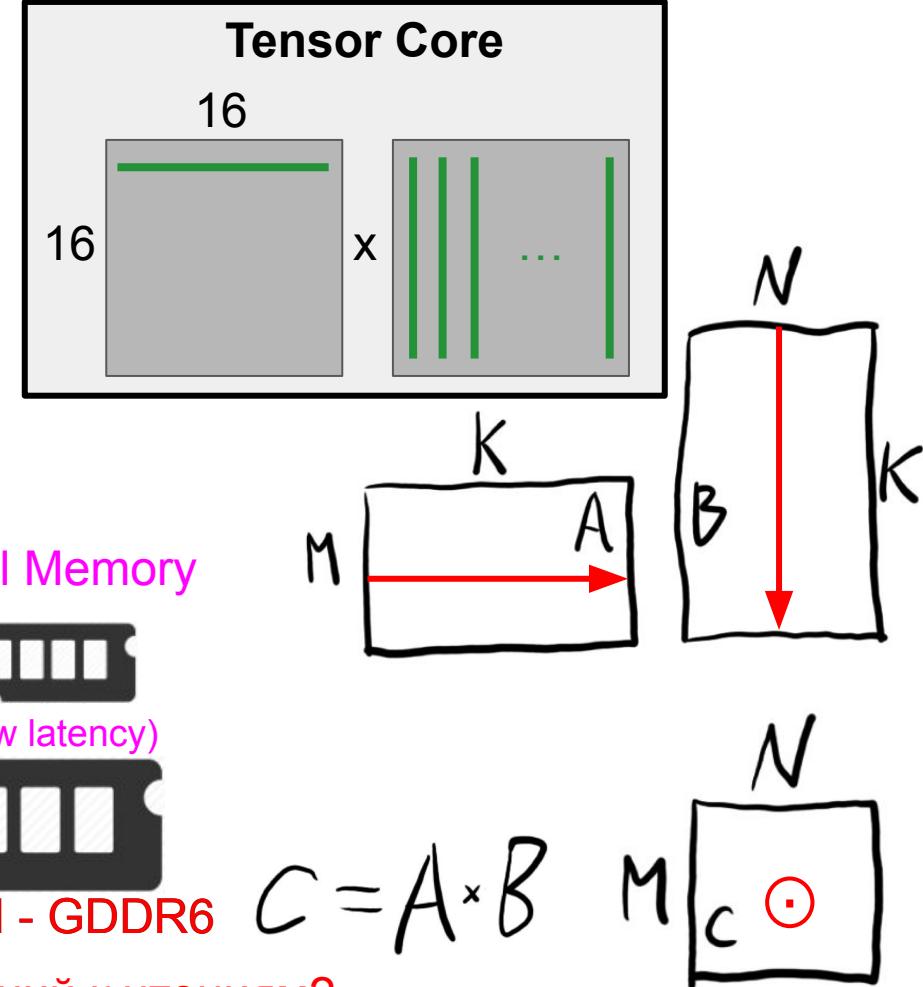
Local Memory

5000 GB/s (low latency)

1000 GB/s



VRAM - GDDR6



Какая у нас теперь пропорция вычислений к чтениям?

Умножение матриц

Tensor Cores

<https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/>

<https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu>

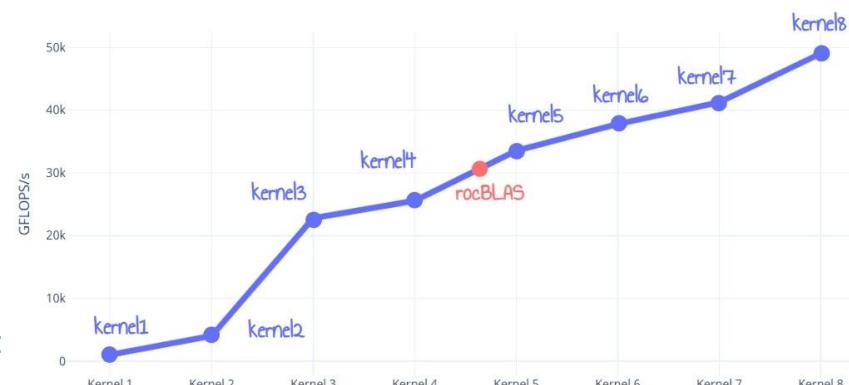
https://github.com/NVIDIA/cutlass/blob/main/examples/00_basic_gemm/basic_gemm.cu

<https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda>

<https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

Неравная битва за гигафлопсы при умножении матриц
(хорошо описанная аналитика, профилирование, оптимизация):

- AMD RDNA3 - <https://seb-v.github.io/optimization/update/2025/01/20/Fast-GPU-Matrix-multiplication.html>
- NVIDIA Kepler - <https://cnugteren.github.io/tutorial/pages/page15.html>
- <https://siboehm.com/articles/22/CUDA-MMM>



Умножение матриц

Tensor Cores

<https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/>

<https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu>

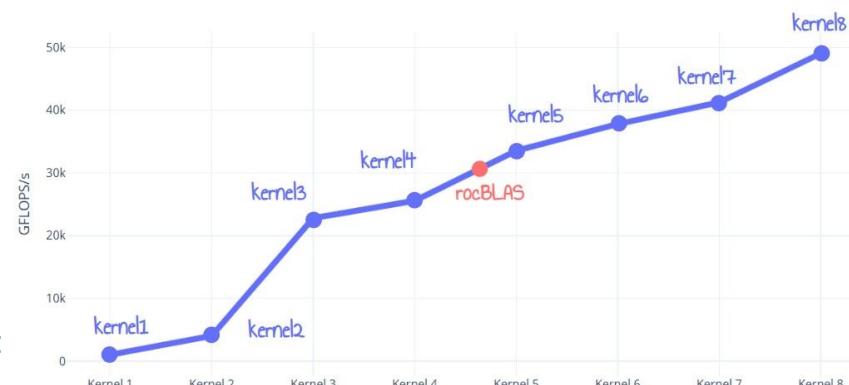
https://github.com/NVIDIA/cutlass/blob/main/examples/00_basic_gemm/basic_gemm.cu

<https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda>

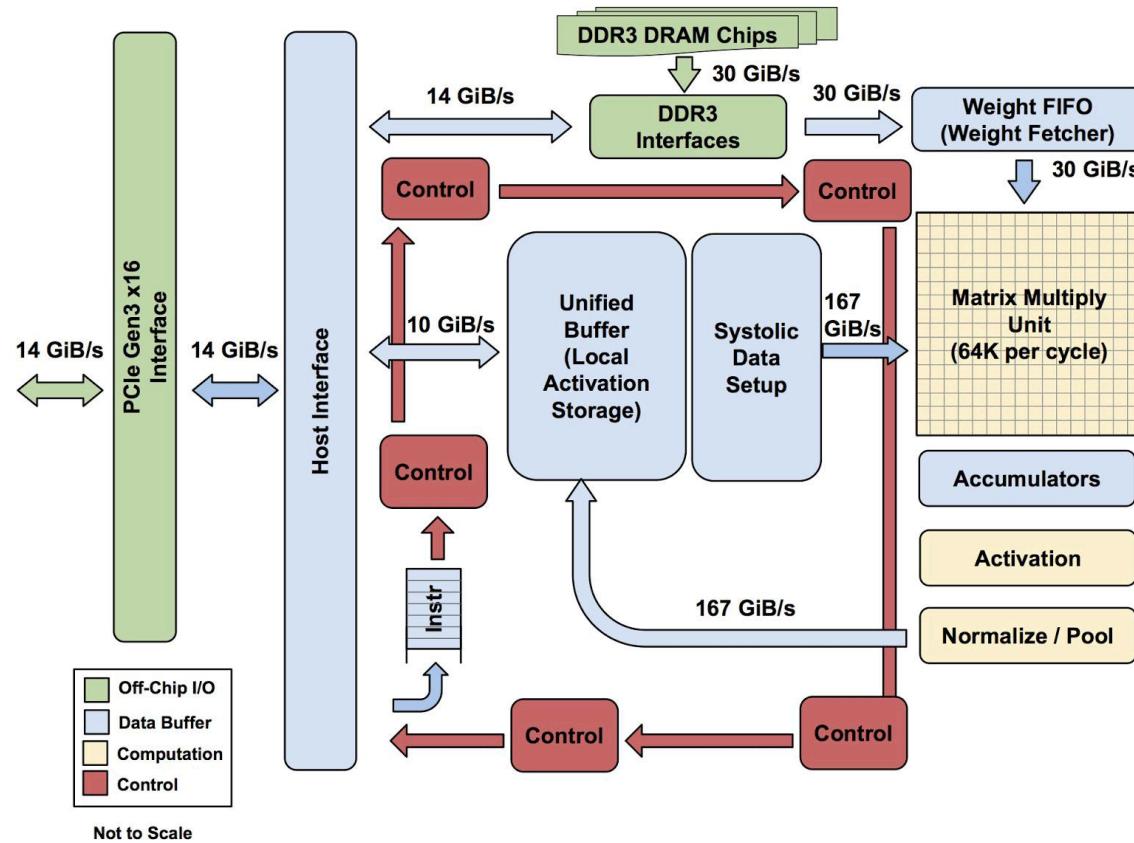
<https://docs.nvidia.com/cuda/cuda-c-programming-guide/>

Неравная битва за гигафлопсы при умножении матриц
(хорошо описанная аналитика, профилирование, оптимизация):

- AMD RDNA3 - <https://seb-v.github.io/optimization/update/2025/01/20/Fast-GPU-Matrix-multiplication.html>
- NVIDIA Kepler - <https://cnugteren.github.io/tutorial/pages/page15.html>
- <https://siboehm.com/articles/22/CUDA-MMM>

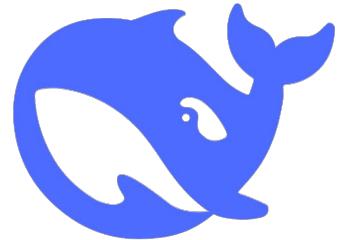


Tensor Processor Unit - TPU (Google)



Глава 4: Умножение матриц

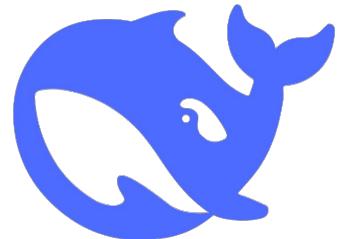
DeepSeek



DeepSeek: x2 ускорение обучения (fp16 → fp8)



$$\pm (1 + \text{mantissa}) \times 2^{(\text{exponent} - 127)}$$



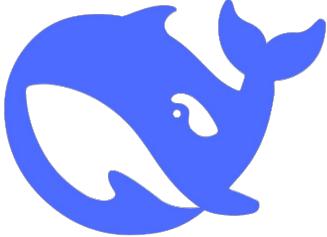
DeepSeek: x2 ускорение обучения (fp16 → fp8)

	sign	exponent						mantissa								
FP16	0	0	1	1	0	1	1	0	0	1	0	1	0	1	1	= 0.395264

BF16	0	0	1	1	1	1	1	0	1	1	0	0	1	0	1	= 0.394531
------	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	------------

FP8 E4M3	0	0	1	0	1	1	0	1								= 0.40625
----------	---	---	---	---	---	---	---	---	--	--	--	--	--	--	--	-----------

FP8 E5M2	0	0	1	1	0	1	1	0								= 0.375
----------	---	---	---	---	---	---	---	---	--	--	--	--	--	--	--	---------

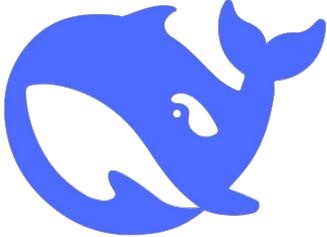


DeepSeek: x2 ускорение обучения (fp16 → fp8)



NVIDIA H100 (почти то же что и H800*):

- Memory bandwidth: **3.35 TB/sec**
- FP 32: **67 TFlops**
- FP 16: **268 TFlops**
- FP 32 (tensor): **495 TFlops**
- FP 16 (tensor): **990 TFlops**
- FP 8 (tensor): **1979 TFlops**



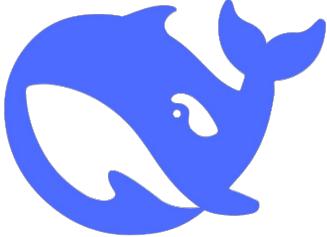
DeepSeek: x2 ускорение обучения (fp16 → fp8)



NVIDIA H100 (почти то же что и H800*):

- Memory bandwidth: **3.35 TB/sec**
- FP 32: **67 TFlops**
- FP 16: **268 TFlops**
- FP 32 (tensor): **495 TFlops**
- FP 16 (tensor): **990 TFlops**
- FP 8 (tensor): **1979 TFlops**

x2



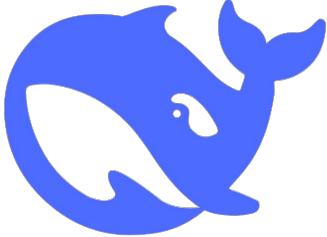
DeepSeek: x2 ускорение обучения (fp16 → fp8)



NVIDIA H100 (почти то же что и H800*):

- Memory bandwidth: **3.35 TB/sec** Хватит ли пропускной способности памяти чтобы насытить ALU (tensor cores)?
 - FP 32: **67 TFlops**
 - FP 16: **268 TFlops**
 - FP 32 (tensor): **495 TFlops**
 - FP 16 (tensor): **990 TFlops**
 - FP 8 (tensor): **1979 TFlops**
- x2

*H800 - почти H100, но соответствует регуляциям экспорта из США в Китай (400 GBs NVlink вместо 600 GB/s + 10% медленнее + нет FP64)



DeepSeek: x2 ускорение обучения (fp16 → fp8)

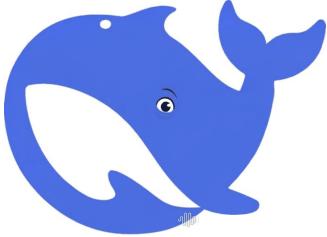


NVIDIA H100 (почти то же что и H800*):

- Memory bandwidth: **3.35 TB/sec**
- FP 32: **67 TFlops**
- FP 16: **268 TFlops**
- FP 32 (tensor): **495 TFlops**
- FP 16 (tensor): **990 TFlops**
- FP 8 (tensor): **1979 TFlops**

Какие риски?
Почему так не делают все?

*H800 - почти H100, но соответствует регуляциям экспорта из США в Китай (400 GBs NVlink вместо 600 GB/s + 10% медленнее + нет FP64)

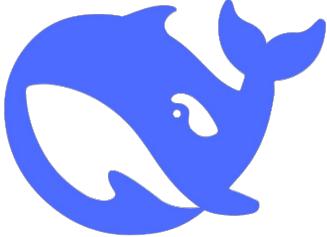


DeepSeek: fine-grained fp8 quantization



NVIDIA H100 (почти то же что и H800*):

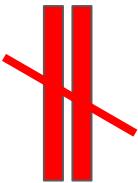
- Memory bandwidth: **3.35 TB/sec**
- FP 32: **67 TFlops**
- FP 16: **268 TFlops**
- FP 32 (tensor): **495 TFlops**
- FP 16 (tensor): **990 TFlops**
- FP 8 (tensor): **1979 TFlops**
- DeepGEMM достиг **1550 TFlops** на H800!



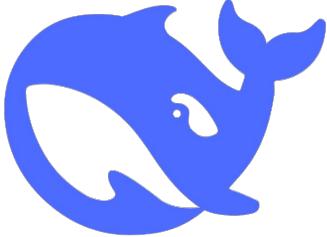
DeepSeek: fine-grained fp8 quantization



fp32



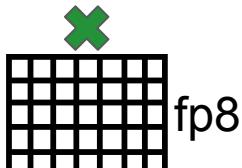
fp8

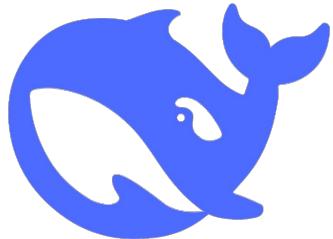


DeepSeek: fine-grained fp8 quantization

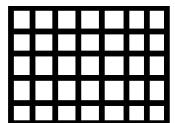


Scaling Factor **fp32**





DeepSeek: fine-grained fp8 quantization



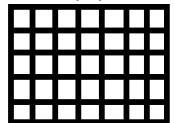
fp32 Input Values



Scaling Factor

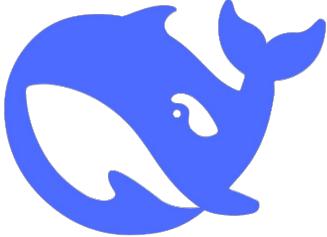
fp32

Каким его выбрать?

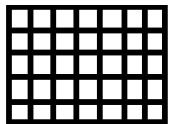


fp8

Input Values / Scaling Factor



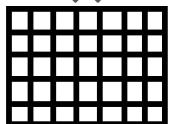
DeepSeek: fine-grained fp8 quantization



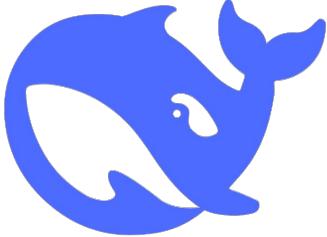
fp32 Input Values



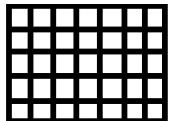
Scaling Factor **fp32** $\text{absmax}(\text{Input Values}) / 448$



fp8 Input Values / Scaling Factor



DeepSeek: fine-grained fp8 quantization

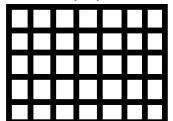


fp32 Input Values

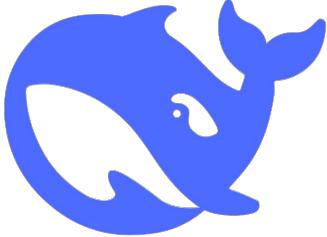


Scaling Factor

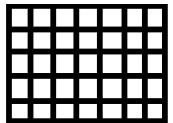
fp32 $\text{absmax}(\text{Input Values}) / 448$ Что это за волшебное число? 🦄



fp8 Input Values / Scaling Factor



DeepSeek: fine-grained fp8 quantization

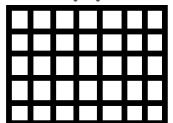


fp32 Input Values



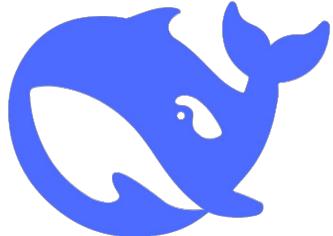
Scaling Factor

$$\text{fp32} \quad \text{absmax}(\text{Input Values}) / 448 = \text{FP8_MAX}$$



fp8

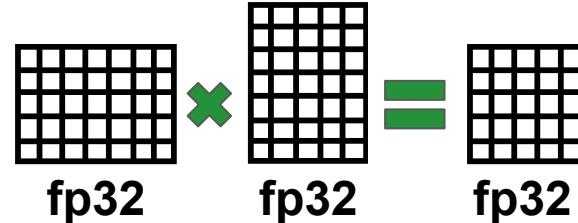
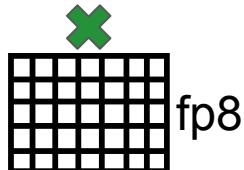
$$\text{Input Values} / \text{Scaling Factor} \in [-448; +448]$$

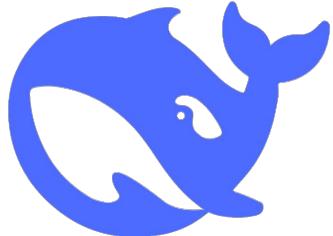


DeepSeek: fine-grained fp8 quantization

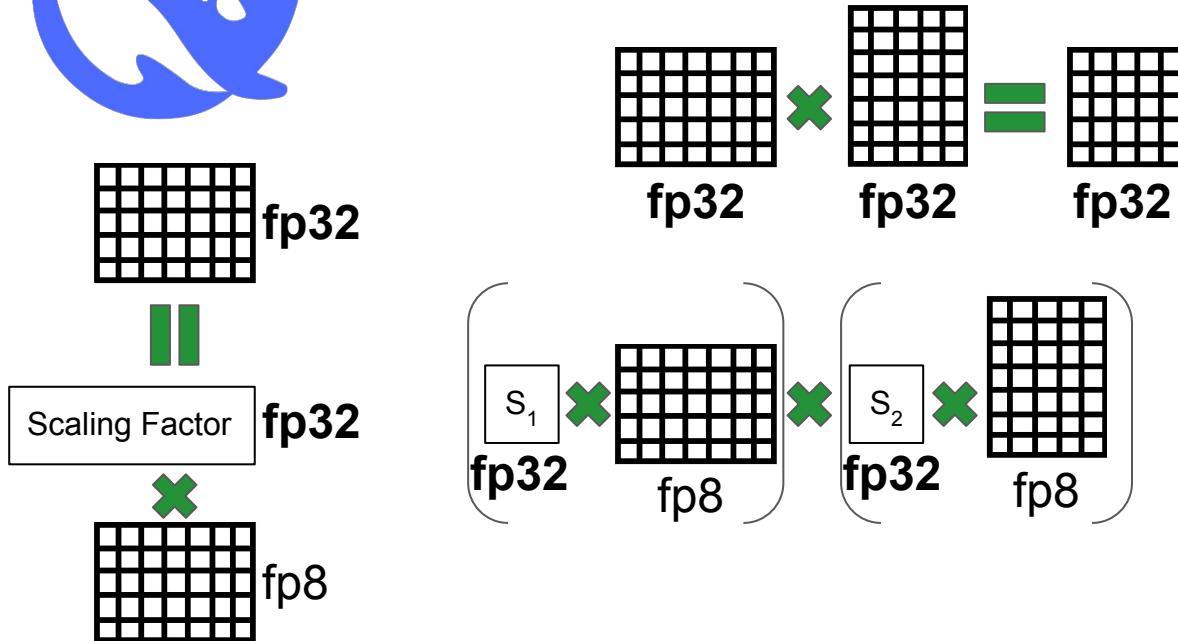


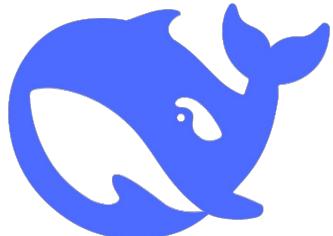
Scaling Factor **fp32**



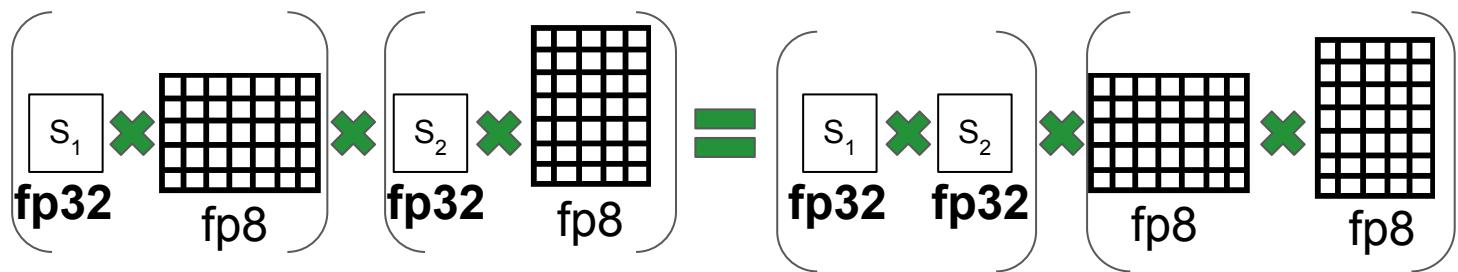
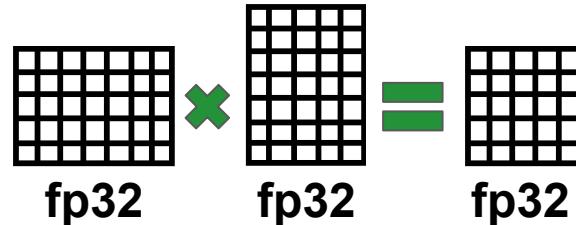
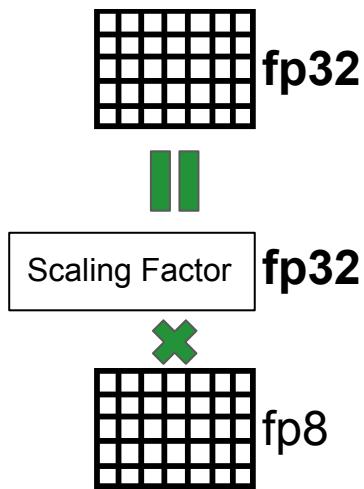


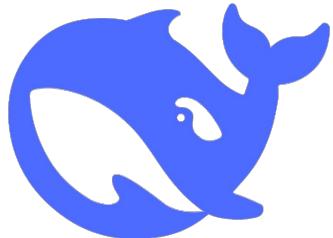
DeepSeek: fine-grained fp8 quantization



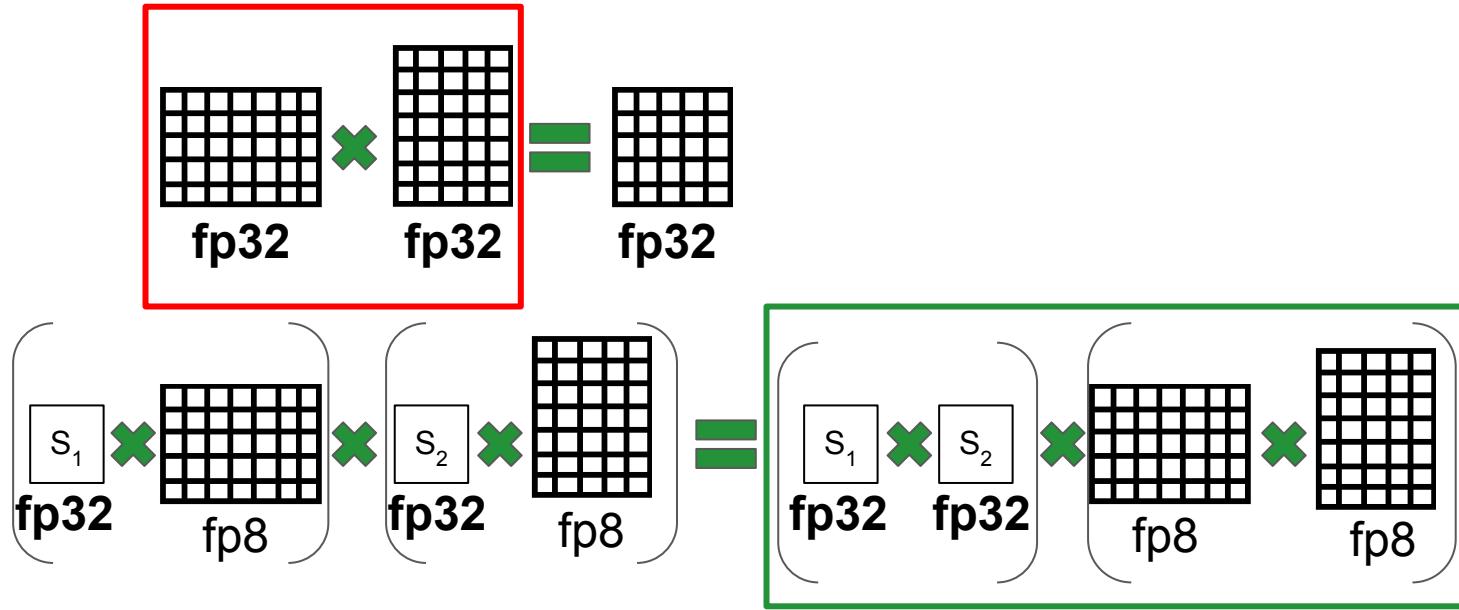
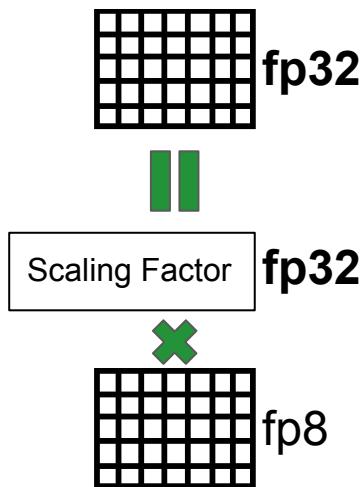


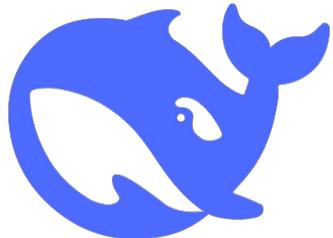
DeepSeek: fine-grained fp8 quantization



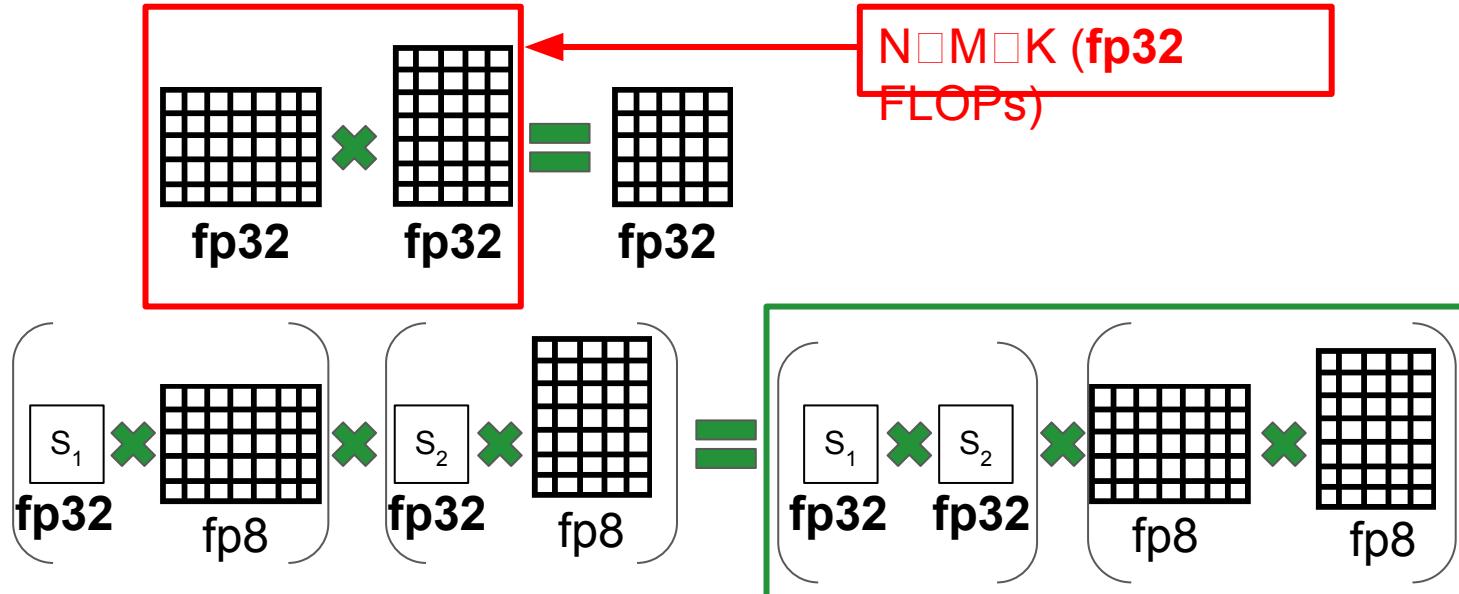
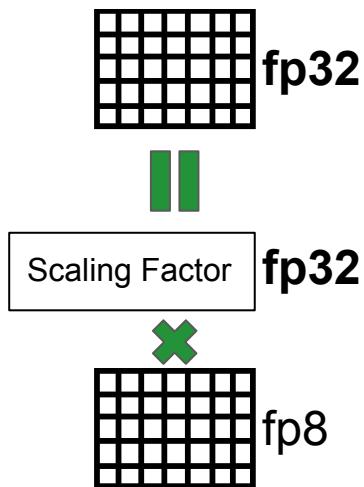


DeepSeek: fine-grained fp8 quantization





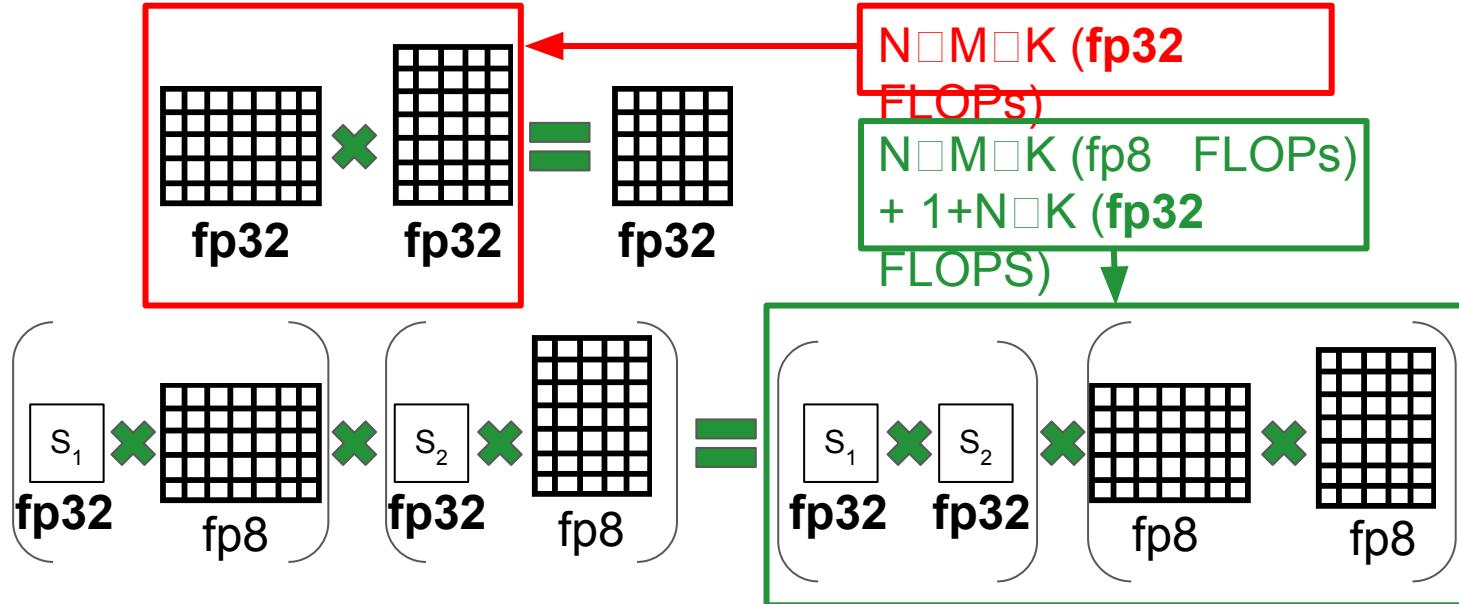
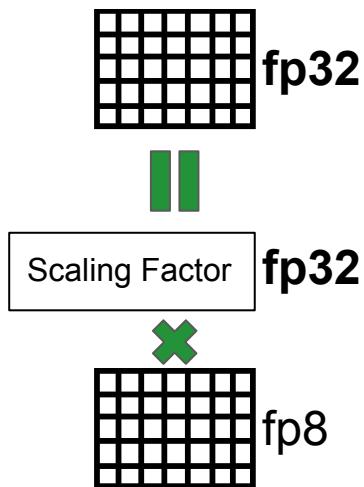
DeepSeek: fine-grained fp8 quantization



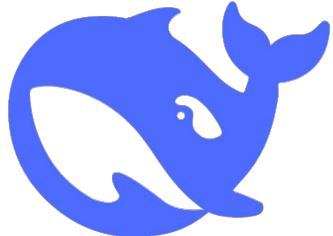
Насколько быстрее?



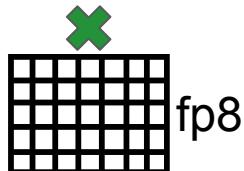
DeepSeek: fine-grained fp8 quantization



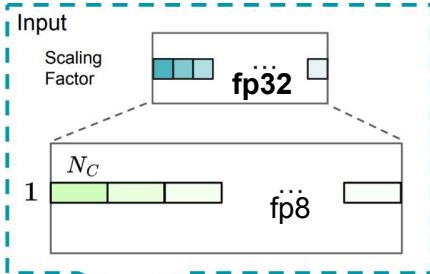
Насколько быстрее?

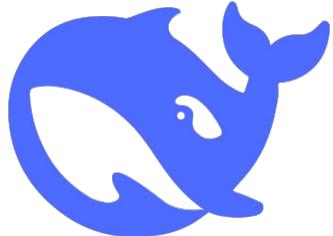


Scaling Factor **fp32**

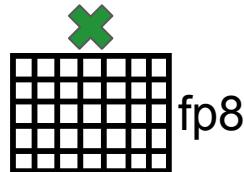


DeepSeek: fine-grained fp8 quantization

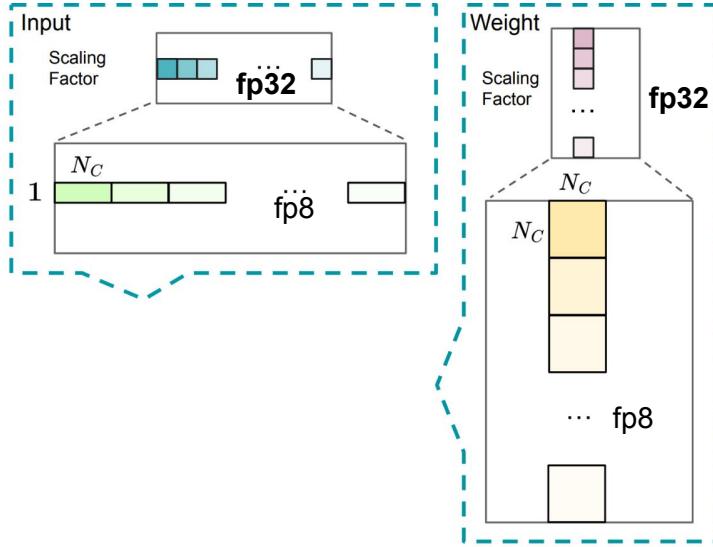


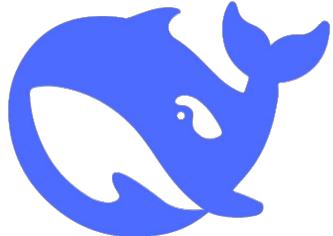


Scaling Factor **fp32**

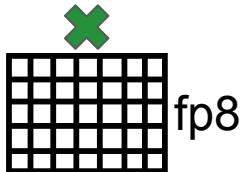


DeepSeek: fine-grained fp8 quantization

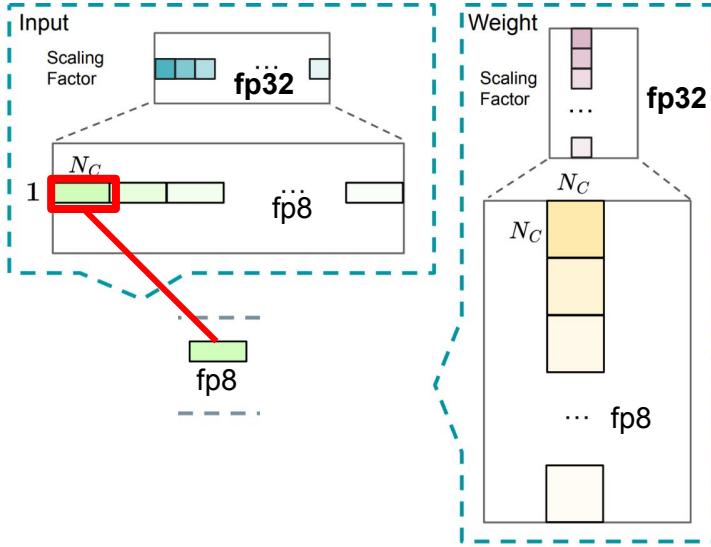




Scaling Factor **fp32**

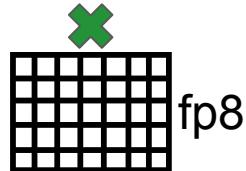


DeepSeek: fine-grained fp8 quantization

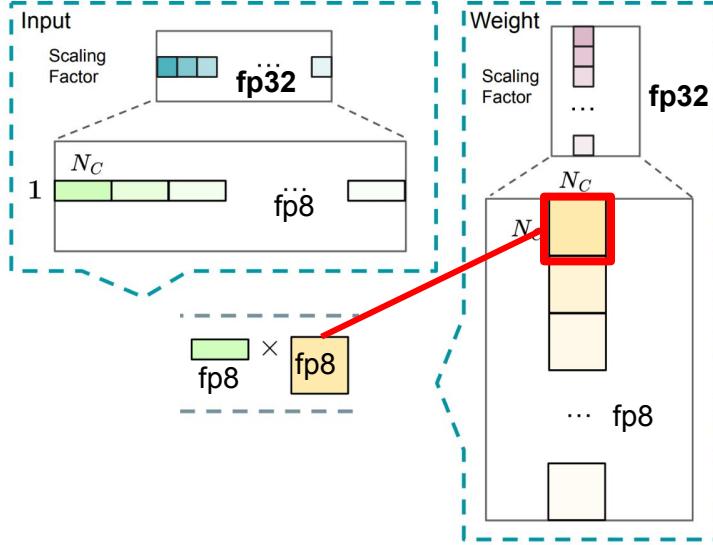


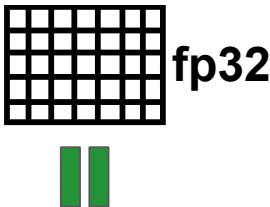


Scaling Factor **fp32**

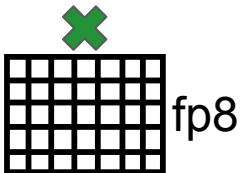


DeepSeek: fine-grained fp8 quantization

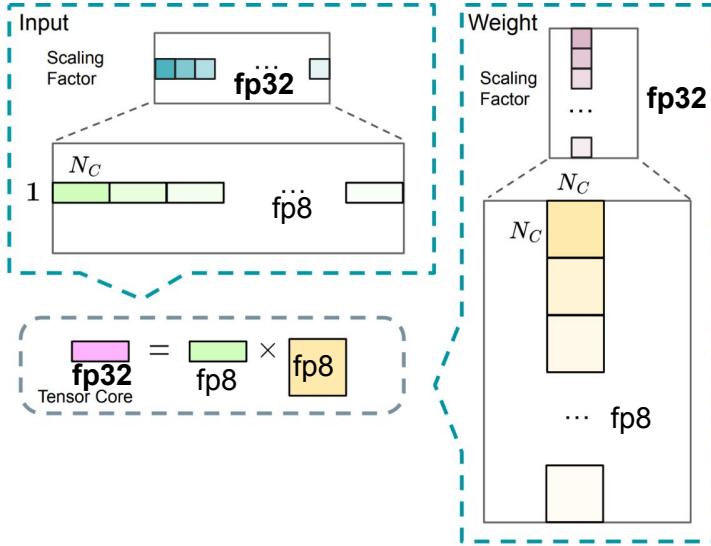


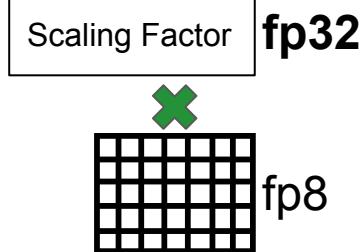
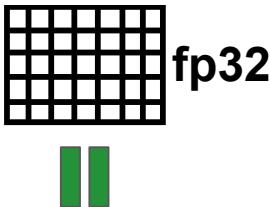


Scaling Factor **fp32**

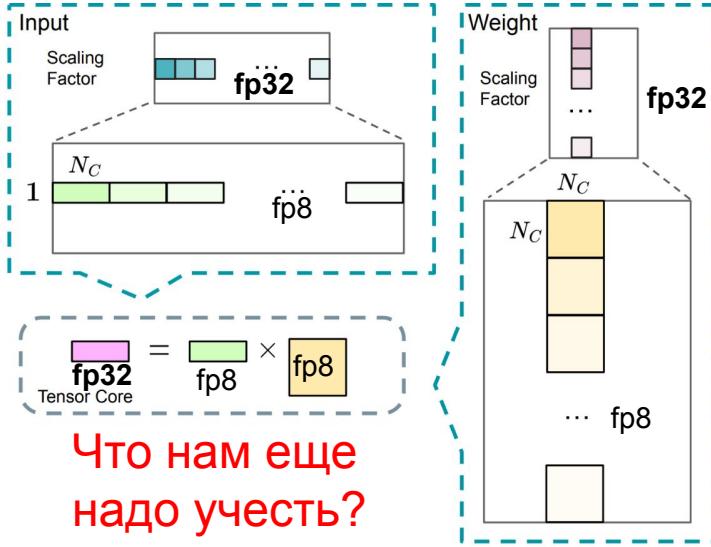


DeepSeek: fine-grained fp8 quantization

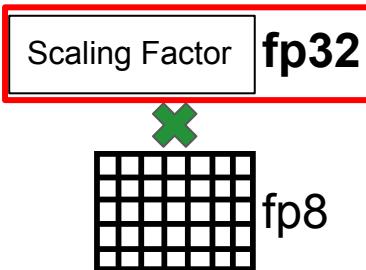
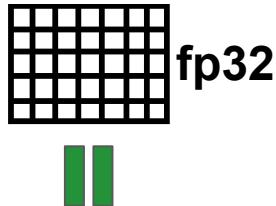
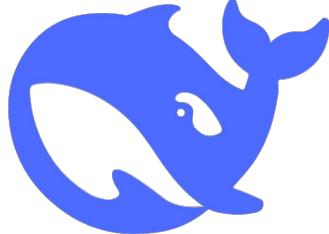




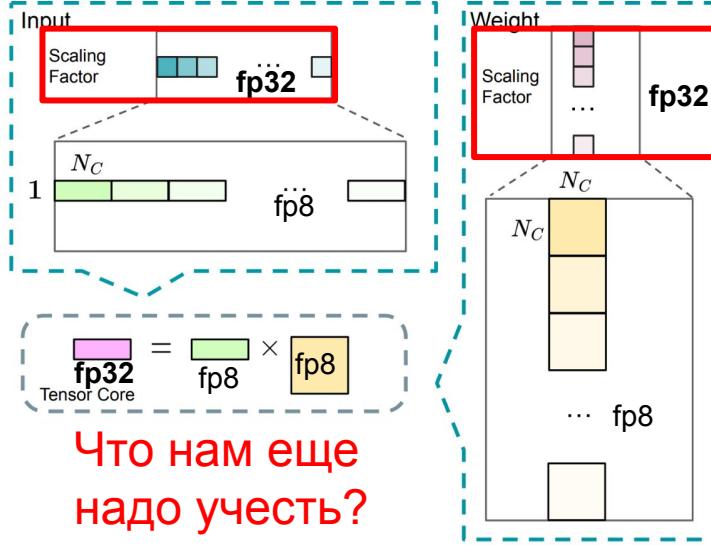
DeepSeek: fine-grained fp8 quantization



Что нам еще
надо учить?



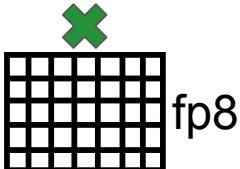
DeepSeek: fine-grained fp8 quantization



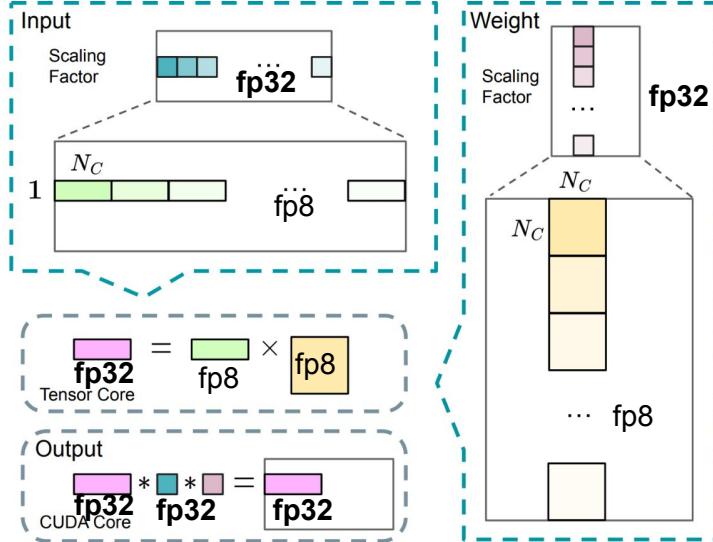
Что нам еще
надо учить?



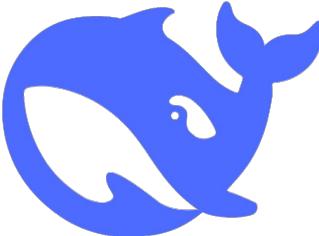
Scaling Factor **fp32**



DeepSeek: fine-grained fp8 quantization



DeepSeek: fine-grained fp8 quantization



fp32

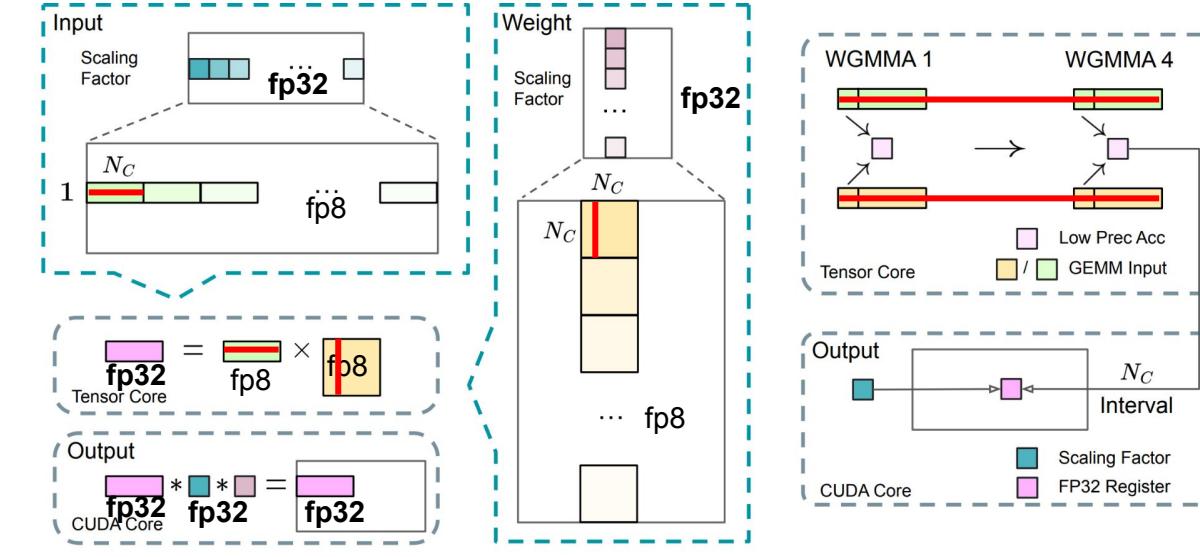
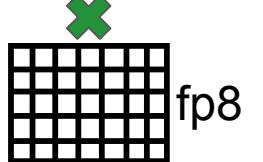


Figure 7 | (a) We propose a fine-grained quantization method to mitigate quantization errors caused by feature outliers; for illustration simplicity, only Fprop is illustrated. (b) In conjunction with our quantization strategy, we improve the FP8 GEMM precision by promoting to CUDA Cores at an interval of $N_C = 128$ elements MMA for the high-precision accumulation.

DeepSeek: fine-grained fp8 quantization



fp32

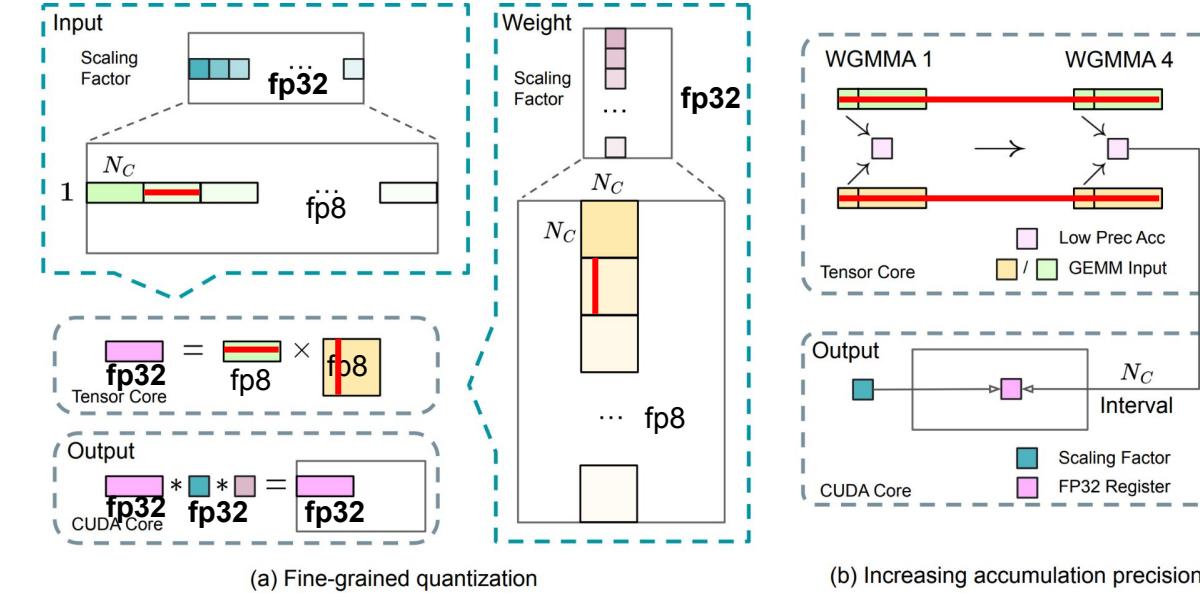
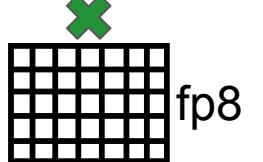
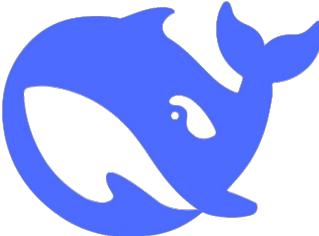


Figure 7 | (a) We propose a fine-grained quantization method to mitigate quantization errors caused by feature outliers; for illustration simplicity, only Fprop is illustrated. (b) In conjunction with our quantization strategy, we improve the FP8 GEMM precision by promoting to CUDA Cores at an interval of $N_C = 128$ elements MMA for the high-precision accumulation.

DeepSeek: fine-grained fp8 quantization



fp32

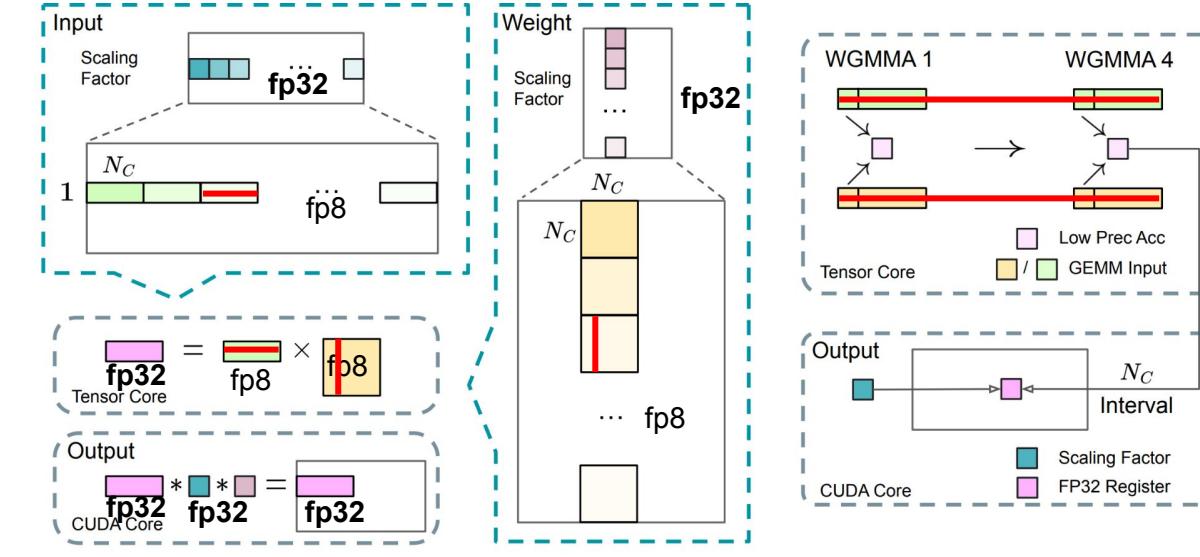
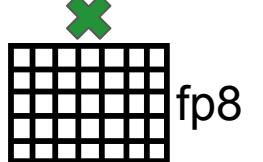
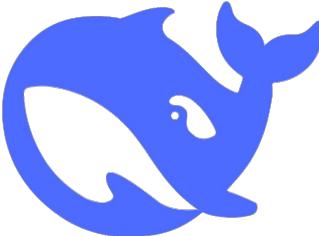
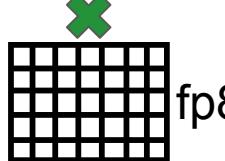


Figure 7 | (a) We propose a fine-grained quantization method to mitigate quantization errors caused by feature outliers; for illustration simplicity, only Fprop is illustrated. (b) In conjunction with our quantization strategy, we improve the FP8 GEMM precision by promoting to CUDA Cores at an interval of $N_C = 128$ elements MMA for the high-precision accumulation.

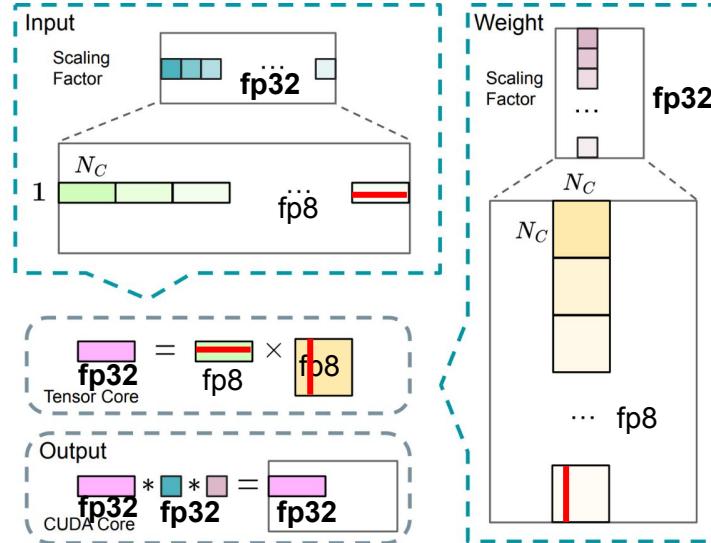
DeepSeek: fine-grained fp8 quantization



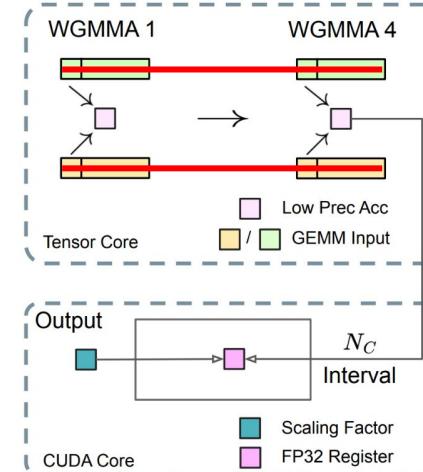
fp32



fp8

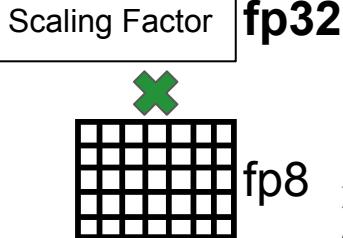
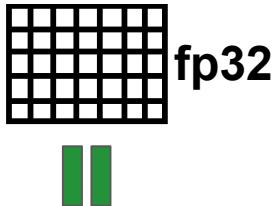
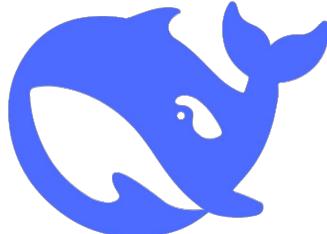


(a) Fine-grained quantization

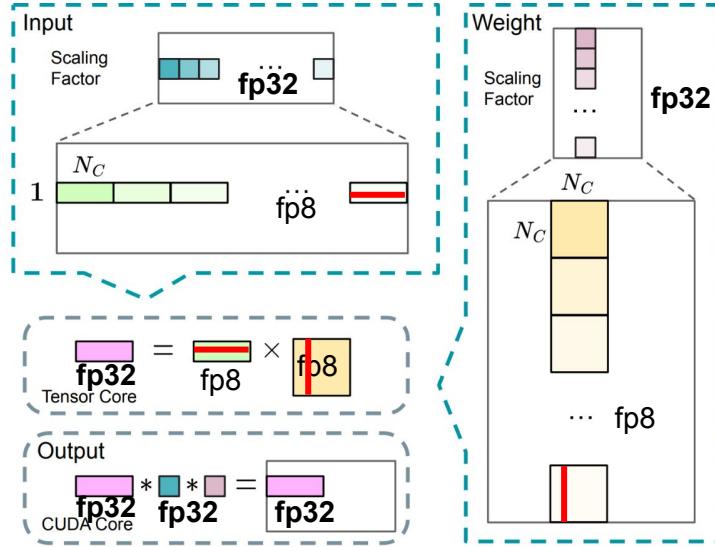


(b) Increasing accumulation precision

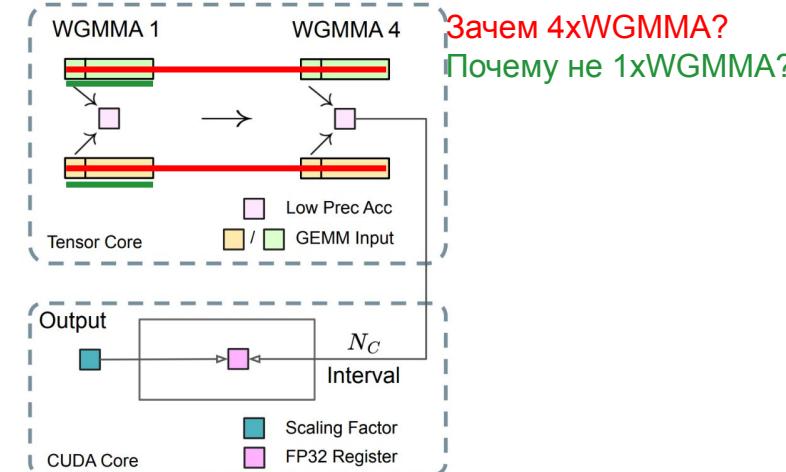
Figure 7 | (a) We propose a fine-grained quantization method to mitigate quantization errors caused by feature outliers; for illustration simplicity, only Fprop is illustrated. (b) In conjunction with our quantization strategy, we improve the FP8 GEMM precision by promoting to CUDA Cores at an interval of $N_C = 128$ elements MMA for the high-precision accumulation.



DeepSeek: fine-grained fp8 quantization



(a) Fine-grained quantization

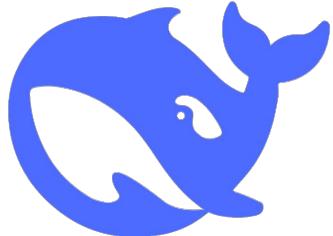


(b) Increasing accumulation precision

Зачем 4xWGMMA?

Почему не 1xWGMMA?

Figure 7 | (a) We propose a fine-grained quantization method to mitigate quantization errors caused by feature outliers; for illustration simplicity, only Fprop is illustrated. (b) In conjunction with our quantization strategy, we improve the FP8 GEMM precision by promoting to CUDA Cores at an interval of $N_C = 128$ elements MMA for the high-precision accumulation.

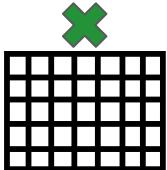


fp32



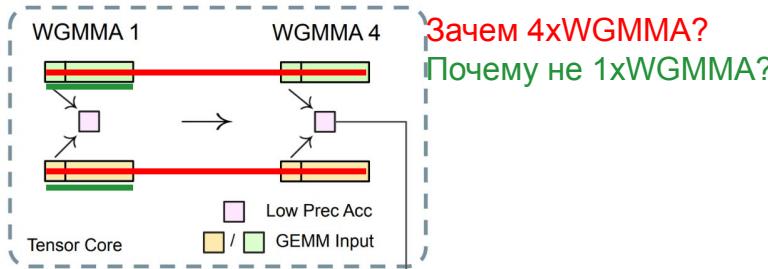
Scaling Factor

fp32

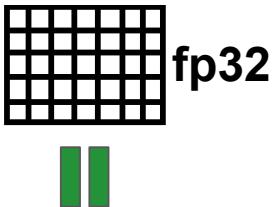
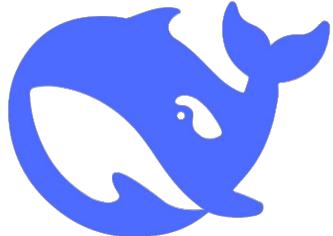


fp8

DeepSeek: fine-grained fp8 quantization



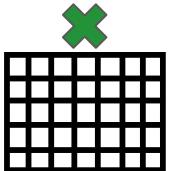
It is worth noting that this modification reduces the WGMMA (Warpgroup-level Matrix Multiply-Accumulate) instruction issue rate for a single warpgroup. However, on the H800 architecture, it is typical for two WGMMA to persist concurrently: while one warpgroup performs the promotion operation, the other is able to execute the MMA operation. This design enables overlapping of the two operations, maintaining high utilization of Tensor Cores. Based on our experiments, setting $N_C = 128$ elements, equivalent to 4 WGMAs, represents the minimal accumulation interval that can significantly improve precision without introducing substantial overhead.



fp32

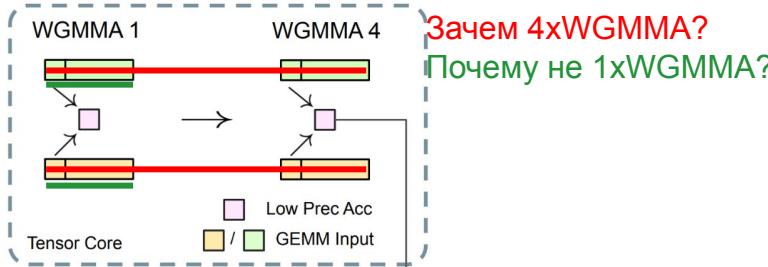


fp32



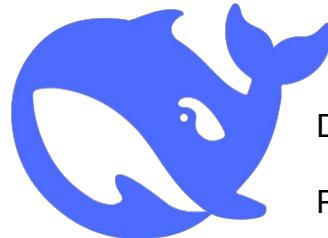
fp8

DeepSeek: fine-grained fp8 quantization



It is worth noting that this modification reduces the WGMMA (Warpgroup-level Matrix Multiply-Accumulate) instruction issue rate for a single warpgroup. However, on the H800 architecture, it is typical for two WGMMA to persist concurrently: while one warpgroup performs the promotion operation, the other is able to execute the MMA operation. This design enables overlapping of the two operations, maintaining high utilization of Tensor Cores. Based on our experiments, setting $N_C = 128$ elements, equivalent to 4 WGMAs, represents the minimal accumulation interval that can significantly improve precision without introducing substantial overhead.

Не знаю почему нужно делать 4xWGMMA + PROMOTION
вместо 4x(WGMMA + PROMOTION)



DeepSeek: x2 ускорение обучения (fp16 → fp8)

DeepSeek-V3 Technical Report - <https://arxiv.org/abs/2412.19437>

Репозиторий - <https://github.com/deepseek-ai/DeepGEMM> (GEMM - General Matrix Multiplications)

[Fast Matrix-Multiplication with WGMMA on NVIDIA® Hopper™ GPUs](#)

https://docs.nvidia.com/deeplearning/transformer-engine/user-guide/examples/fp8_primer.html

Глава 5: Умножение матриц

Метод Штассена

Важнейшим оператором в языке Python является умножение матриц. Для этого есть специальный класс `Matrix`, который определяет метод `__mul__`. Важно отметить, что умножение матриц не коммутативно, то есть $A \cdot B \neq B \cdot A$.

Метод `__mul__` имеет следующий формат:

```
def __mul__(self, other):
    if not isinstance(other, Matrix):
        raise ValueError("Multiplication is only defined for Matrix objects")
    else:
        return Matrix._matrix_multiply(self, other)
```

Метод `_matrix_multiply` имеет следующий формат:

```
def _matrix_multiply(A, B):
    if A.n != B.m:
        raise ValueError("Incompatible dimensions for multiplication")
    else:
        m = A.m
        n = B.n
        p = A.n
        C = Matrix(m, n)
        for i in range(m):
            for j in range(n):
                C[i][j] = sum(A[i][k] * B[k][j] for k in range(p))
        return C
```

Метод `sum` в данном контексте означает вычисление суммы элементов в строке i матрицы B . Для этого используется цикл `for k in range(p)`. Важно отметить, что в данном коде используется явное суммирование, что может привести к медленному выполнению для больших матриц.

Для улучшения производительности можно использовать рекурсивный метод Штассена. Важно отметить, что для этого необходимо изменить метод `__mul__` следующим образом:

```
def __mul__(self, other):
    if not isinstance(other, Matrix):
        raise ValueError("Multiplication is only defined for Matrix objects")
    else:
        if self.n == 1:
            return other
        else:
            return Matrix._matrix_multiply(self, other)
```

Метод `_matrix_multiply` имеет следующий формат:

```
def _matrix_multiply(A, B):
    if A.n == 1:
        return B
    else:
        m = A.m
        n = B.n
        p = A.n
        C = Matrix(m, n)
        for i in range(m):
            for j in range(n):
                C[i][j] = sum(A[i][k] * B[k][j] for k in range(p))
        return C
```

В данном коде метод `sum` используется для вычисления суммы элементов в строке i матрицы B . Важно отметить, что в данном коде используется явное суммирование, что может привести к медленному выполнению для больших матриц.

Умножение матриц - Метод Штрассена

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

Умножение матриц - Метод Штрассена

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

$p_1 = a(f - h)$
 $p_3 = (c + d)e$
 $p_5 = (a + d)(e + h)$
 $p_7 = (a - c)(e + f)$

$$p_2 = (a + b)h$$

$$p_4 = d(g - e)$$

$$p_6 = (b - d)(g + h)$$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}$$

A B C

Умножение матриц - Метод Штрассена

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

$p_1 = a(f - h)$
 $p_3 = (c + d)e$
 $p_5 = (a + d)(e + h)$
 $p_7 = (a - c)(e + f)$

$p_2 = (a + b)h$
 $p_4 = d(g - e)$
 $p_6 = (b - d)(g + h)$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}$$

A B C

Получили рекуррентную асимптотику $T(N)=7T(N/2)+8O(N^2) \Rightarrow T(N)=O(N^{\log 7})=O(N^{2.8})$

Умножение матриц - Метод Штрассена

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

A B C

$p_1 = a(f - h)$
 $p_3 = (c + d)e$
 $p_5 = (a + d)(e + h)$
 $p_7 = (a - c)(e + f)$

$p_2 = (a + b)h$
 $p_4 = d(g - e)$
 $p_6 = (b - d)(g + h)$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p_5 + p_4 - p_2 + p_6 & p_1 + p_2 \\ p_3 + p_4 & p_1 + p_5 - p_3 - p_7 \end{bmatrix}$$

A B C

Получили рекуррентную асимптотику $T(N)=7T(N/2)+8O(N^2) \Rightarrow T(N)=O(N^{\log 7})=O(N^{2.8})$

Метод Виноградова - тоже $O(N^{2.8})$

<https://www.geeksforgeeks.org/strassens-matrix-multiplication/>



ВОПРОСЫ?



[@UnicornGlade](https://t.me/UnicornGlade)

[@PolarNick239](https://twitter.com/PolarNick239)

polarnick239@gmail.com

Николай Полярный