CS Space
Клуб технологий и науки

Вычисления на видеокартах

Vulkan

OpenCL

NVIDIA CUDA

Лекция 4

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

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* (и VK_KHR_cooperative_matrix)
4) **Умножение матриц:** оптимизации *DeepSeek*
5) **Умножение матриц:** метод *Штрассена*

Глава 1: Транспонирование

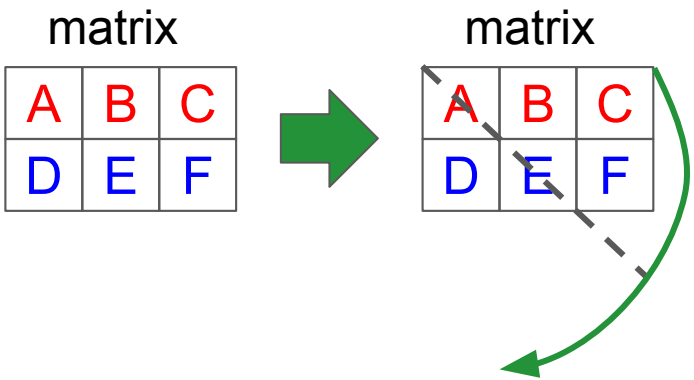
local memory, coalesced access, bank conflicts

Транспонирование матрицы

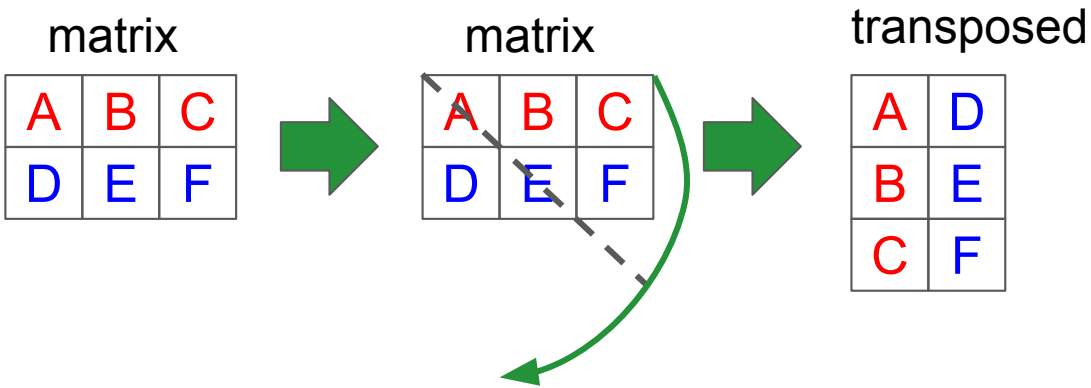
matrix

A	B	C
D	E	F

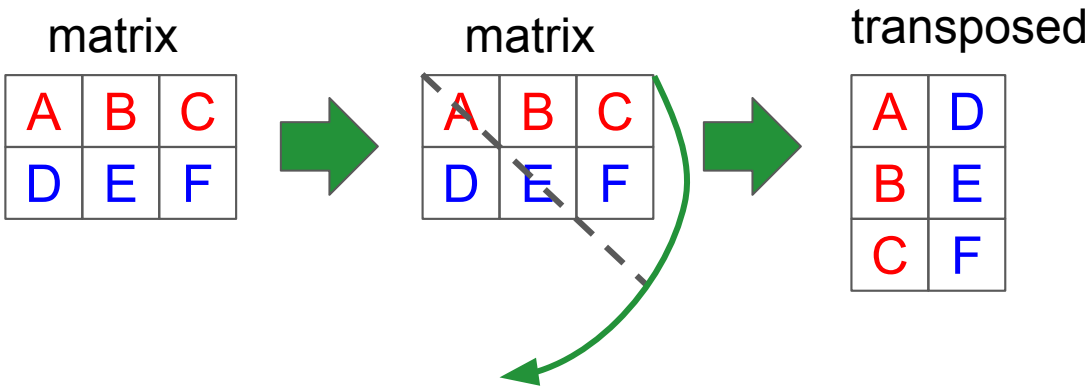
Транспонирование матрицы



Транспонирование матрицы



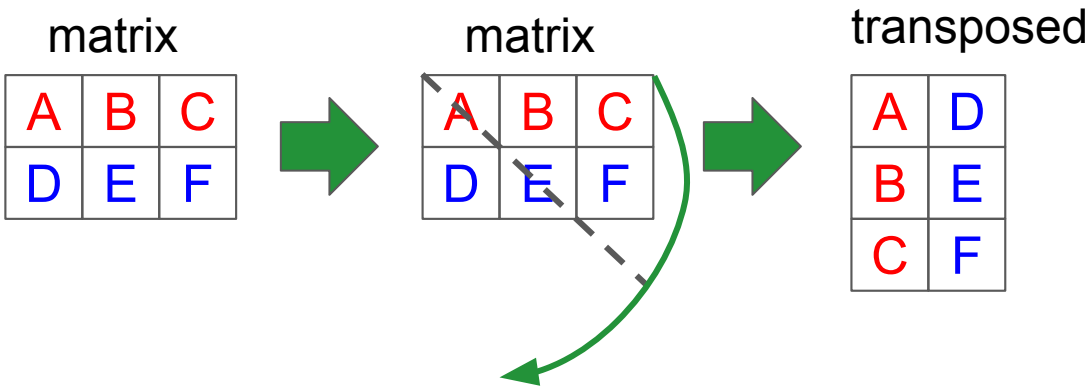
Транспонирование матрицы



Как выглядит:

- WorkRange?

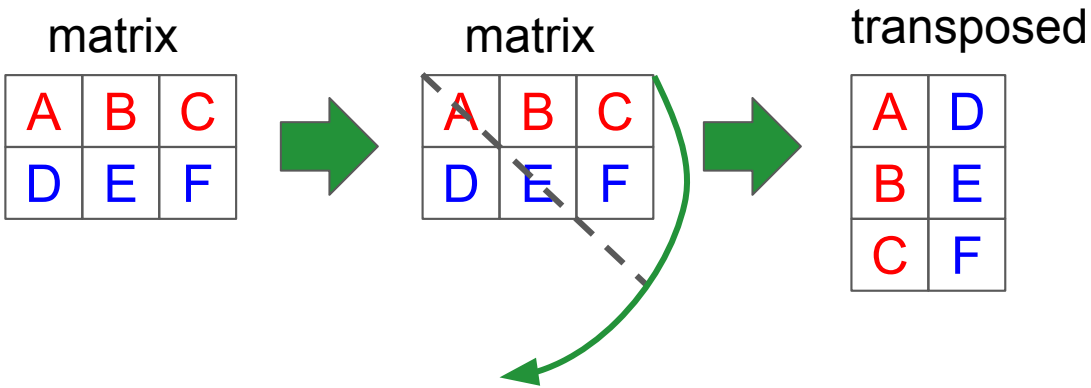
Транспонирование матрицы



Как выглядит:

- WorkRange?
- Задача WorkItem?

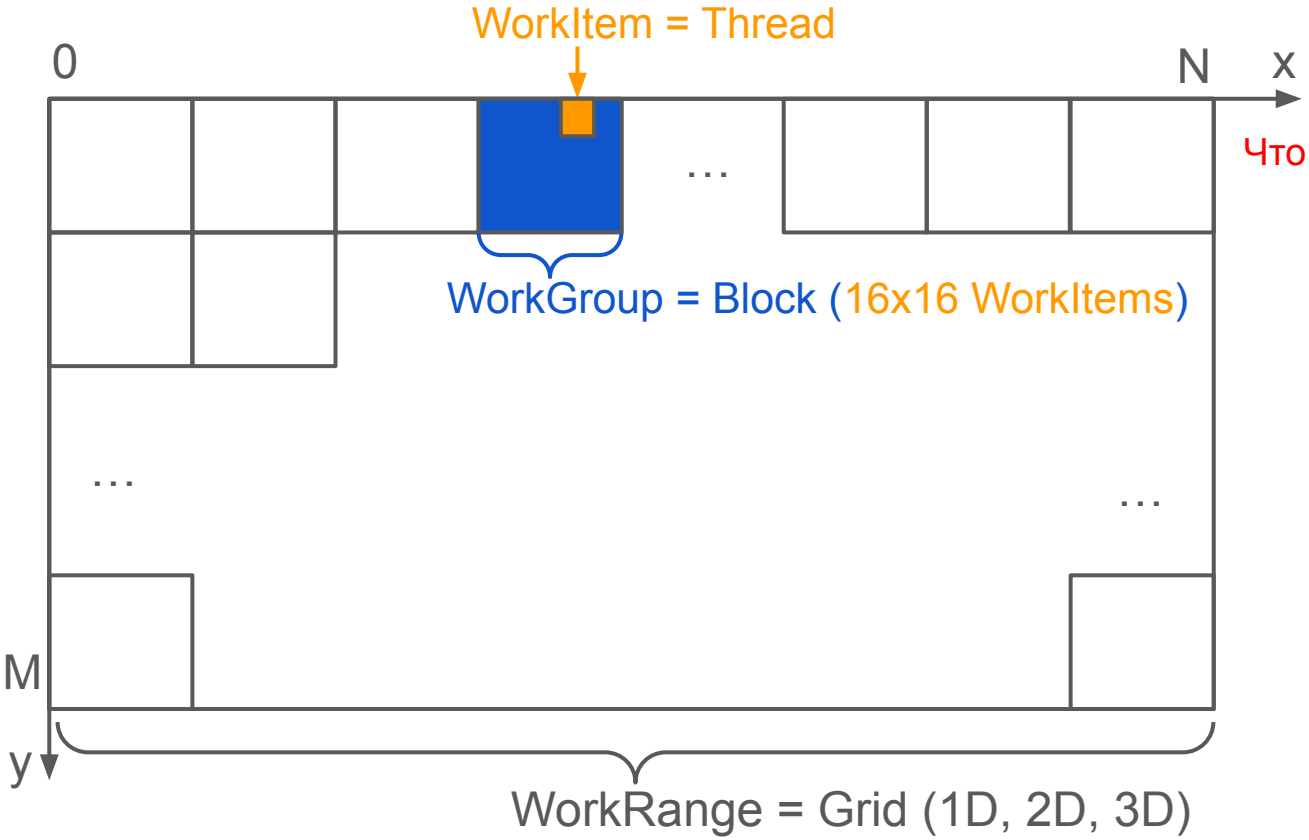
Транспонирование матрицы



Как выглядит:

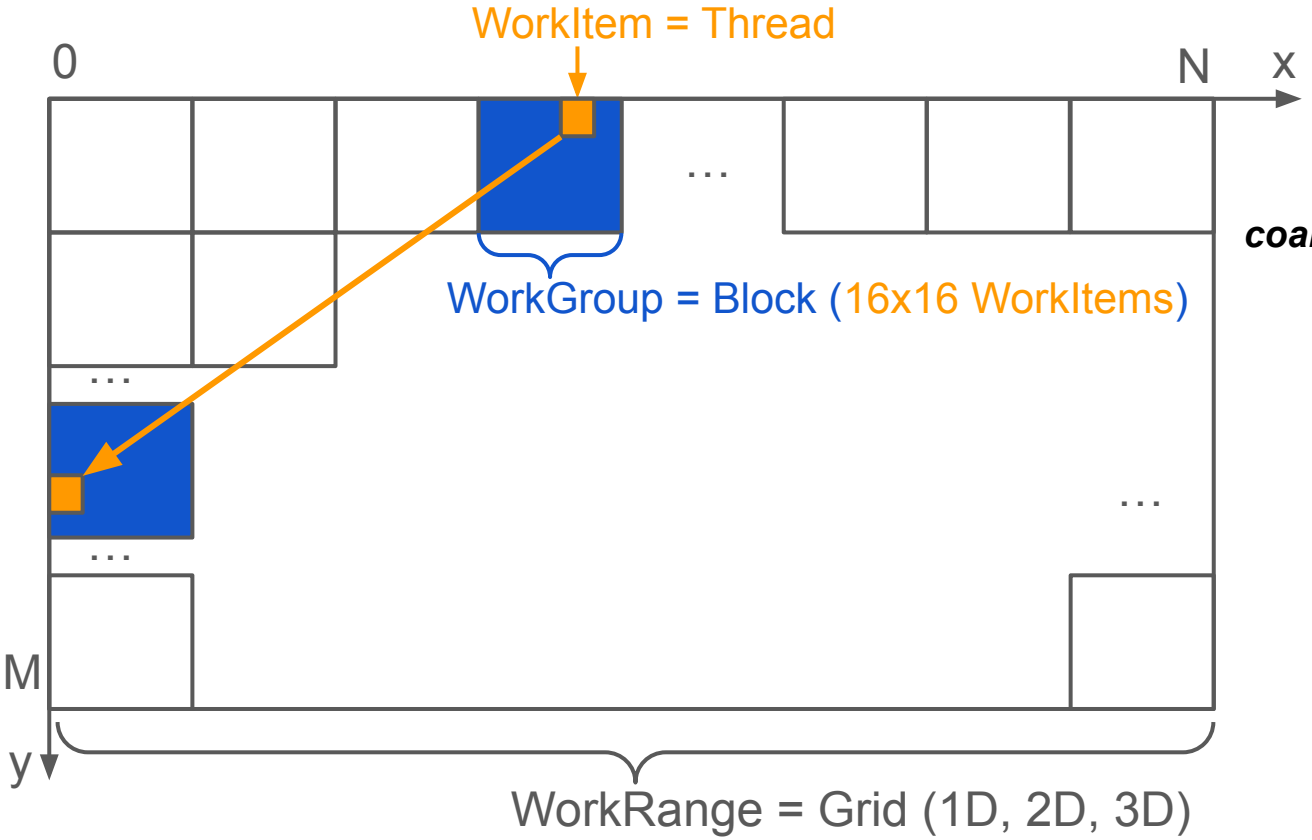
- WorkRange?
- Задача WorkItem?
- WorkGroup?

Транспонирование матрицы



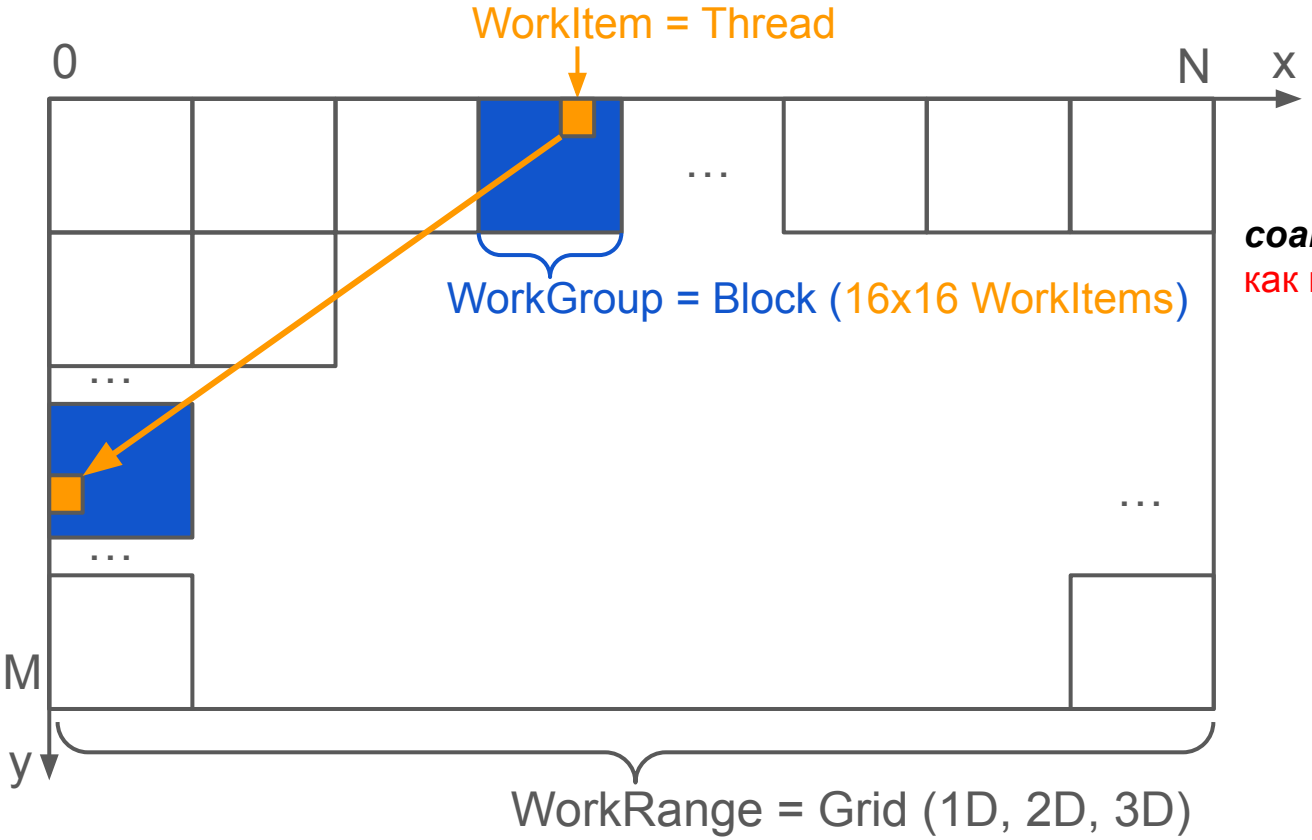
Что делает каждый поток (WorkItem)?

Транспонирование матрицы (*coalesced memory access*)



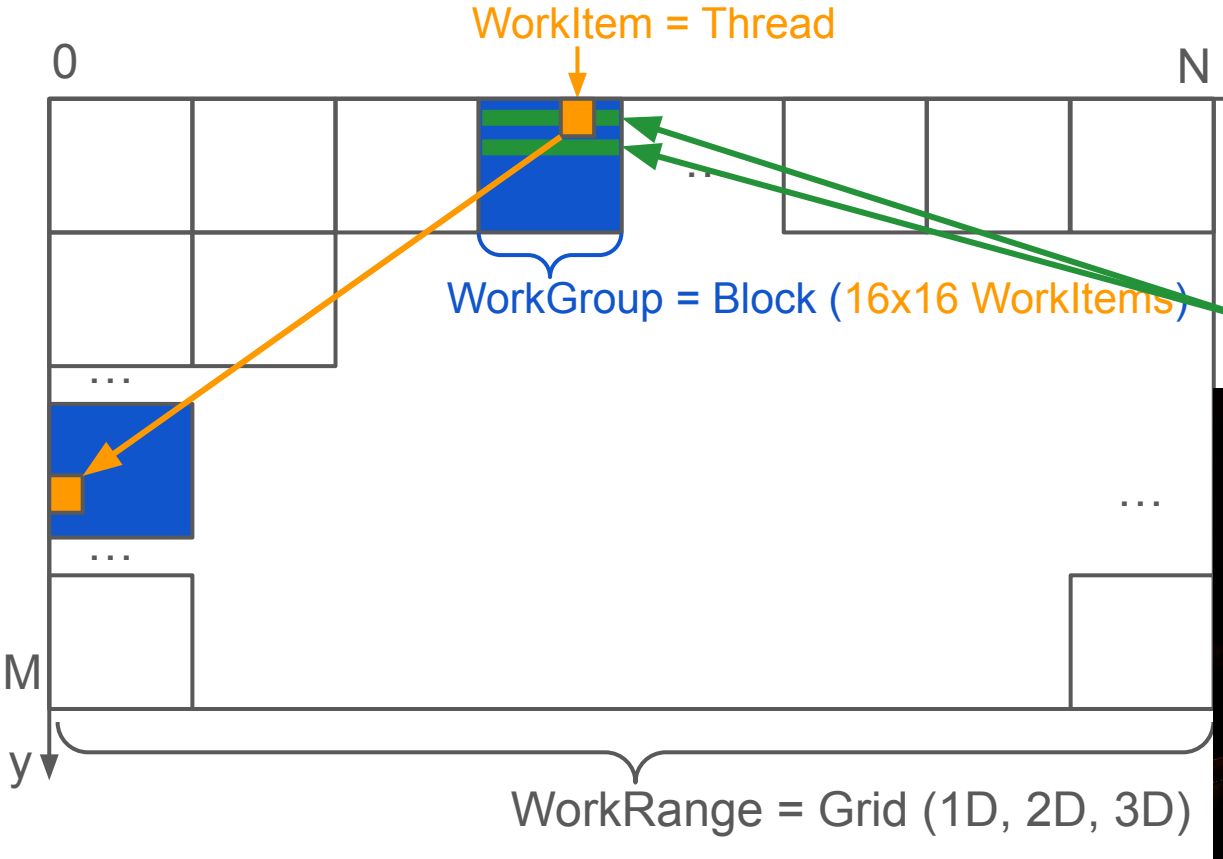
coalesced ли паттерн чтения?

Транспонирование матрицы (*coalesced memory access*)



coalesced ли паттерн чтения?
как выглядит выкладка **warp**?
(row-major order)

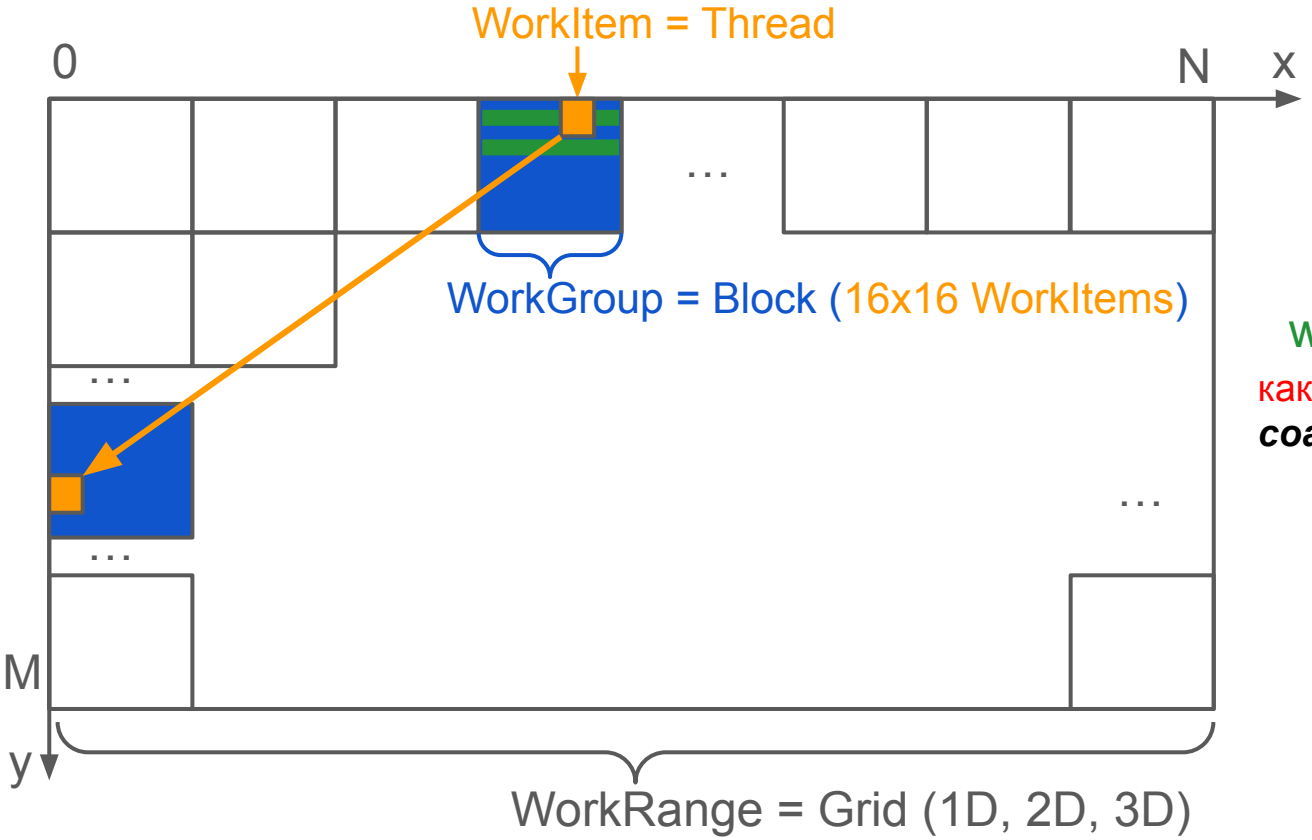
Транспонирование матрицы (*coalesced memory access*)



coalesced ли паттерн чтения?
как выглядит выкладка **warp**?

warp = 32 = 16+16

Транспонирование матрицы (*coalesced memory access*)

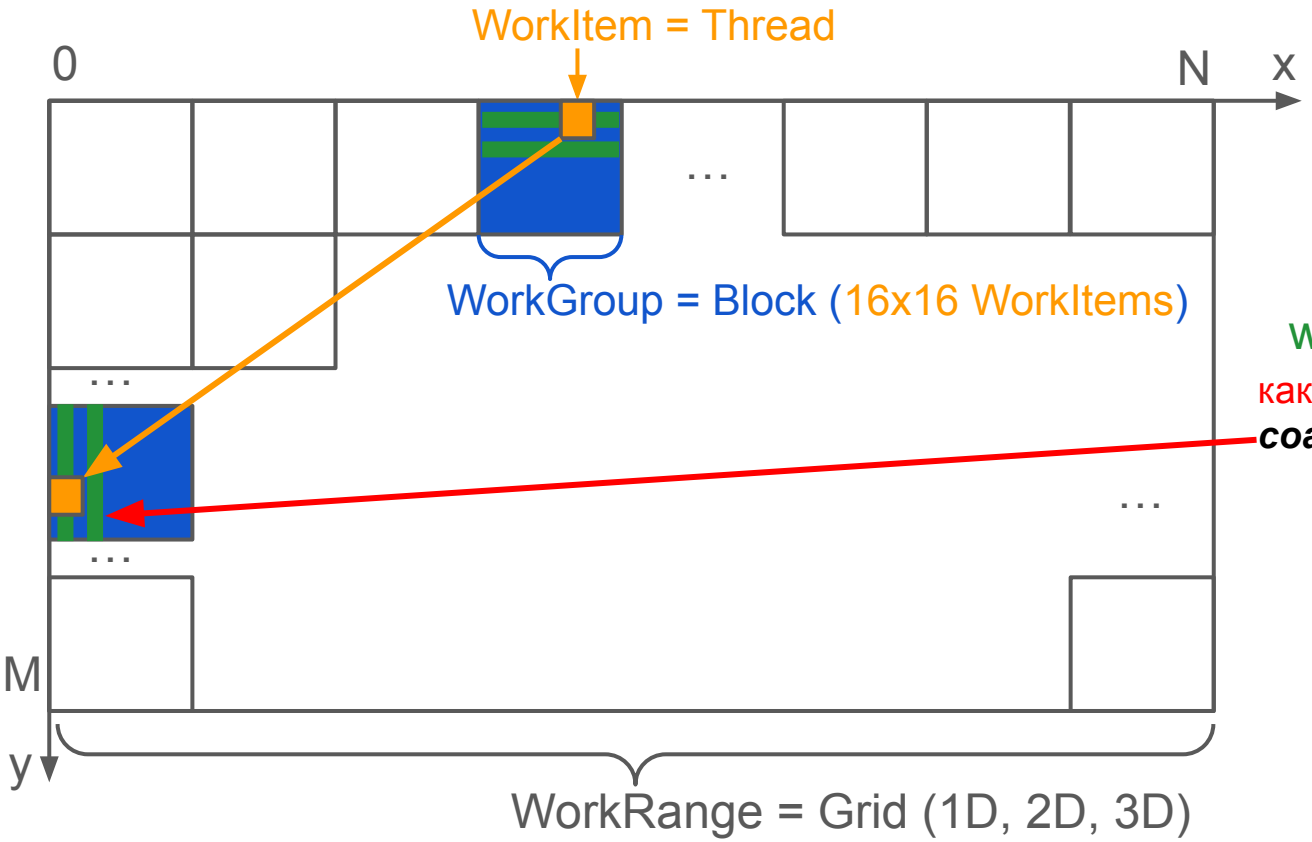


warp = 32 = 16+16

как выглядит паттерн **записи**?

coalesced ли паттерн **записи**?

Транспонирование матрицы (*coalesced memory access*)

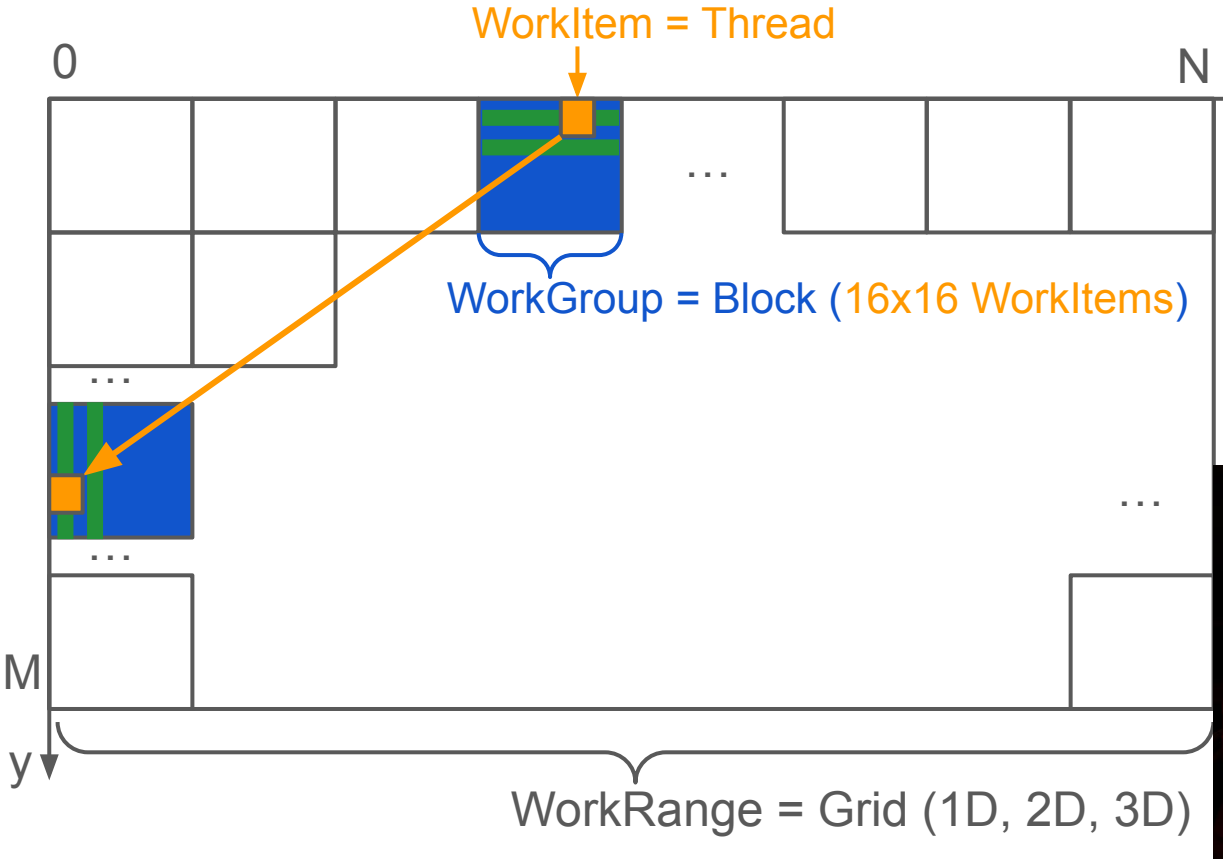


warp = 32 = 16+16

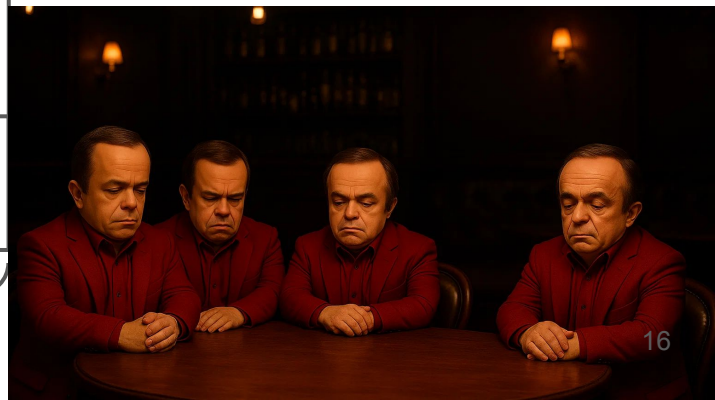
как выглядит паттерн **записи**?

coalesced ли паттерн **записи**?

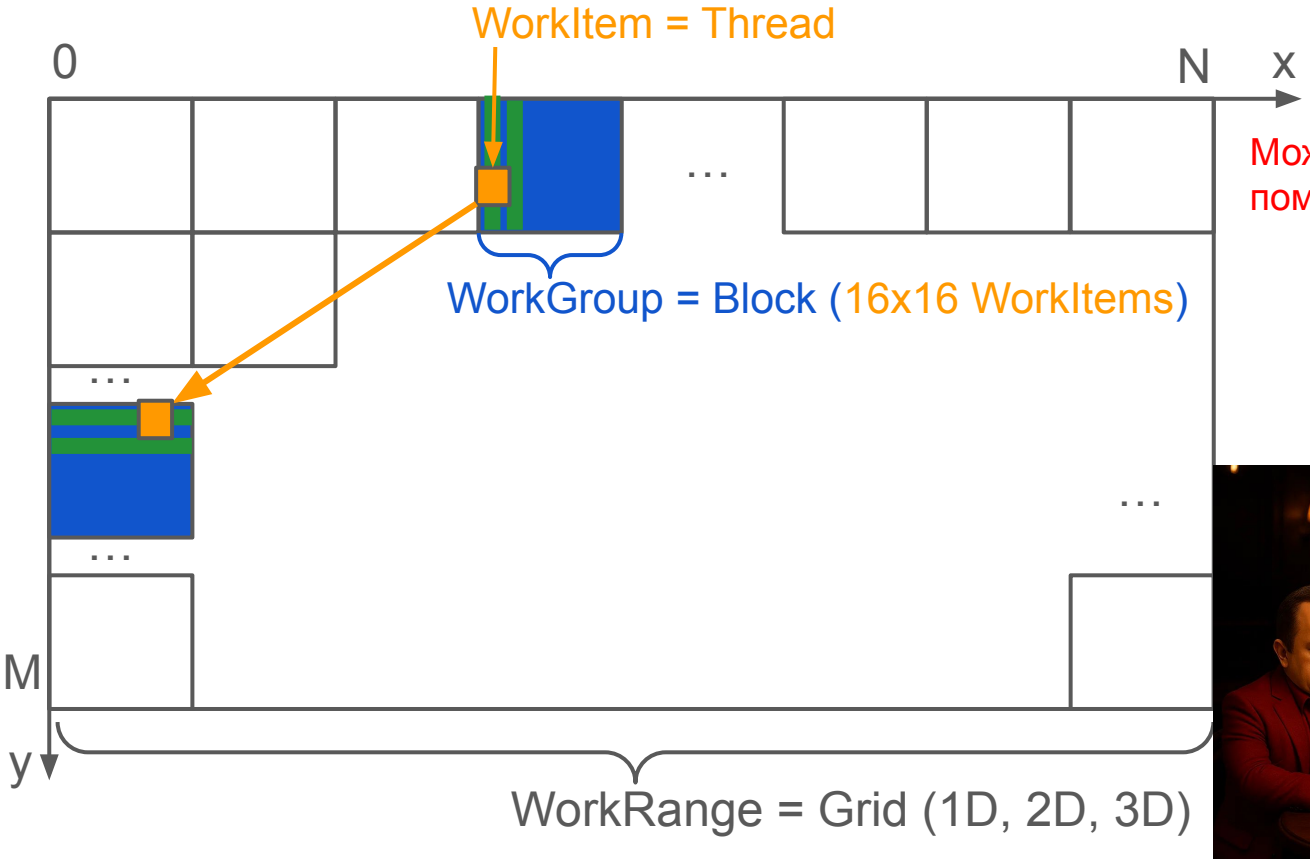
Транспонирование матрицы (*coalesced memory access*)



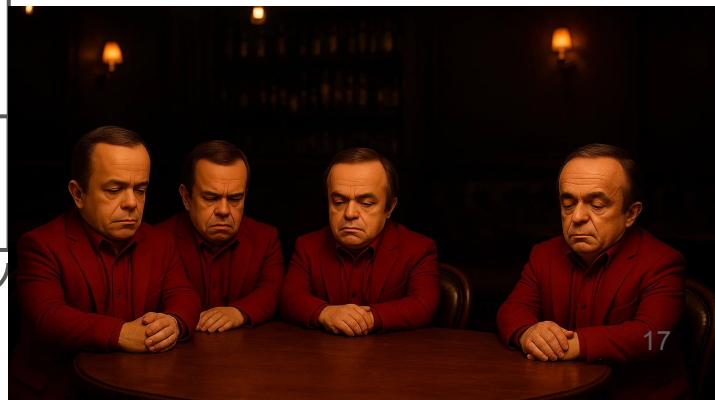
Можем ли мы исправить это поменяв выкладку **warp**-а?



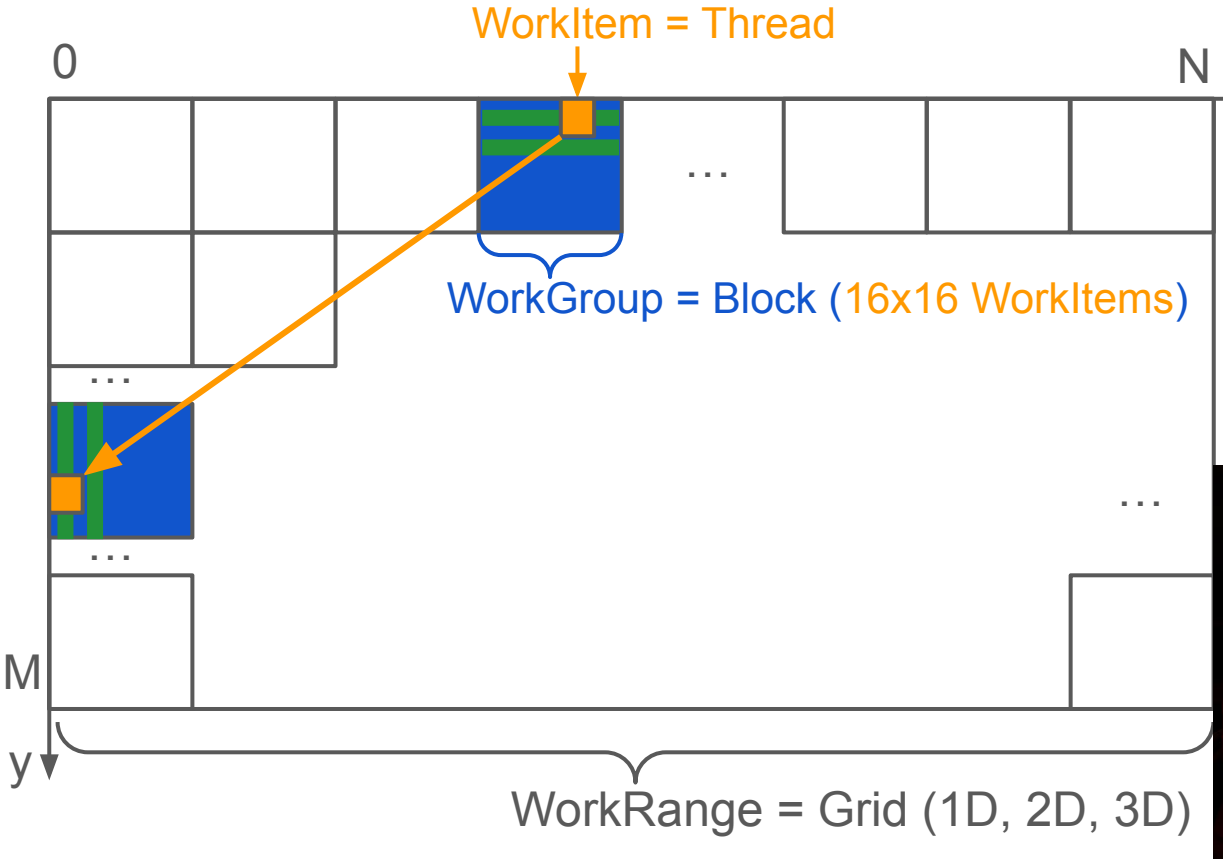
Транспонирование матрицы (*coalesced memory access*)



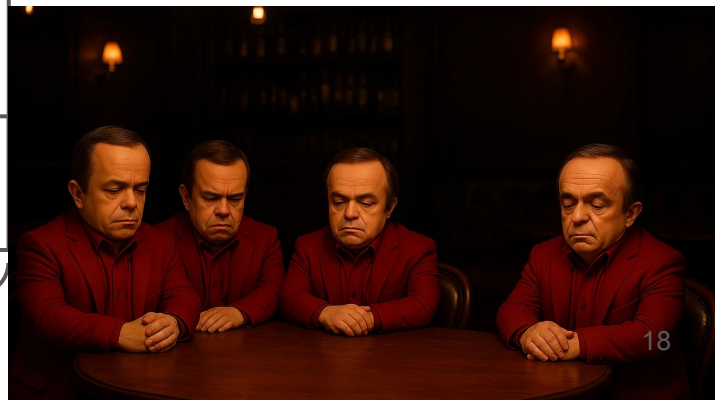
Можем ли мы исправить это поменяв выкладку **warp**-а?



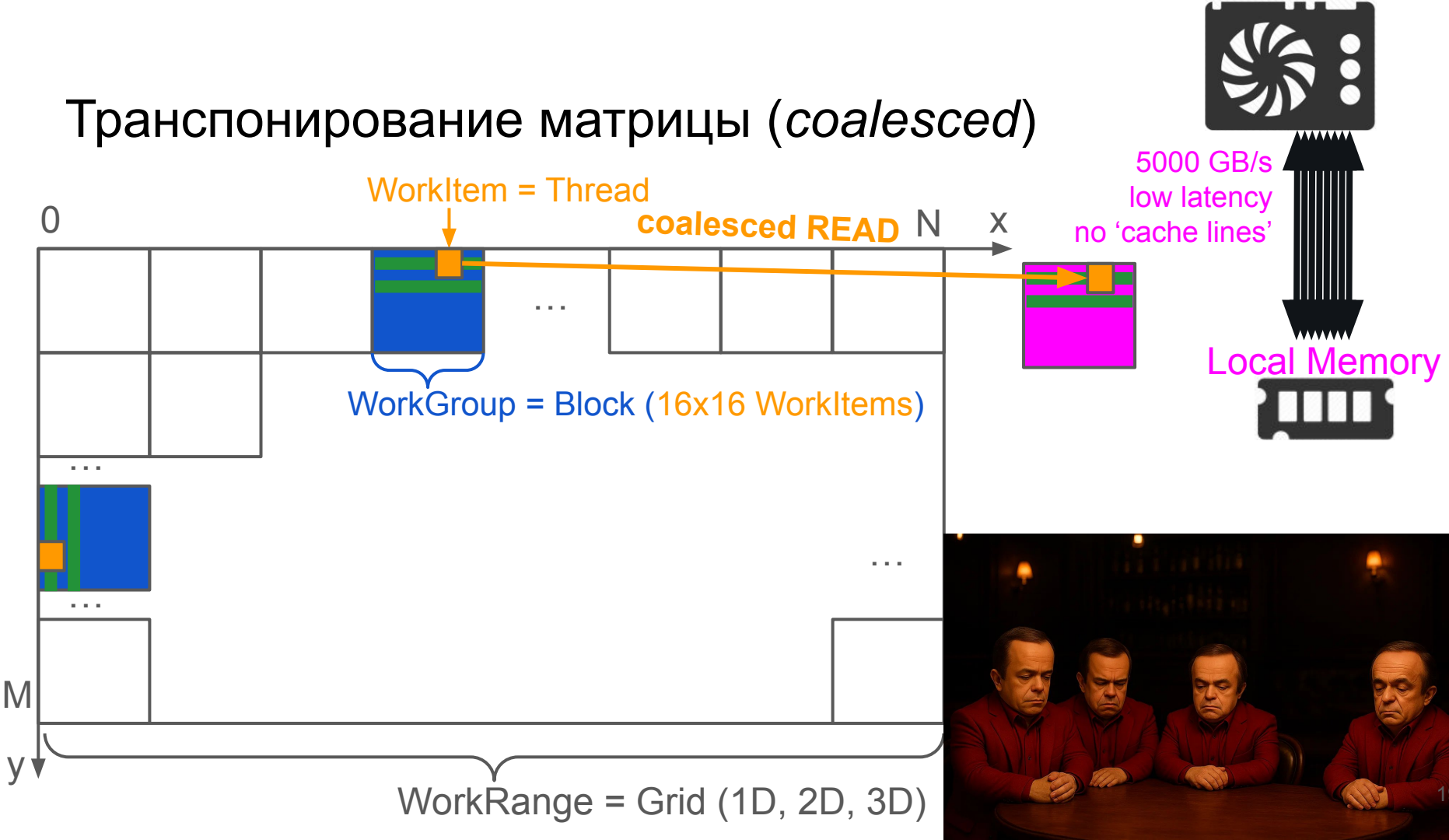
Транспонирование матрицы (*coalesced memory access*)



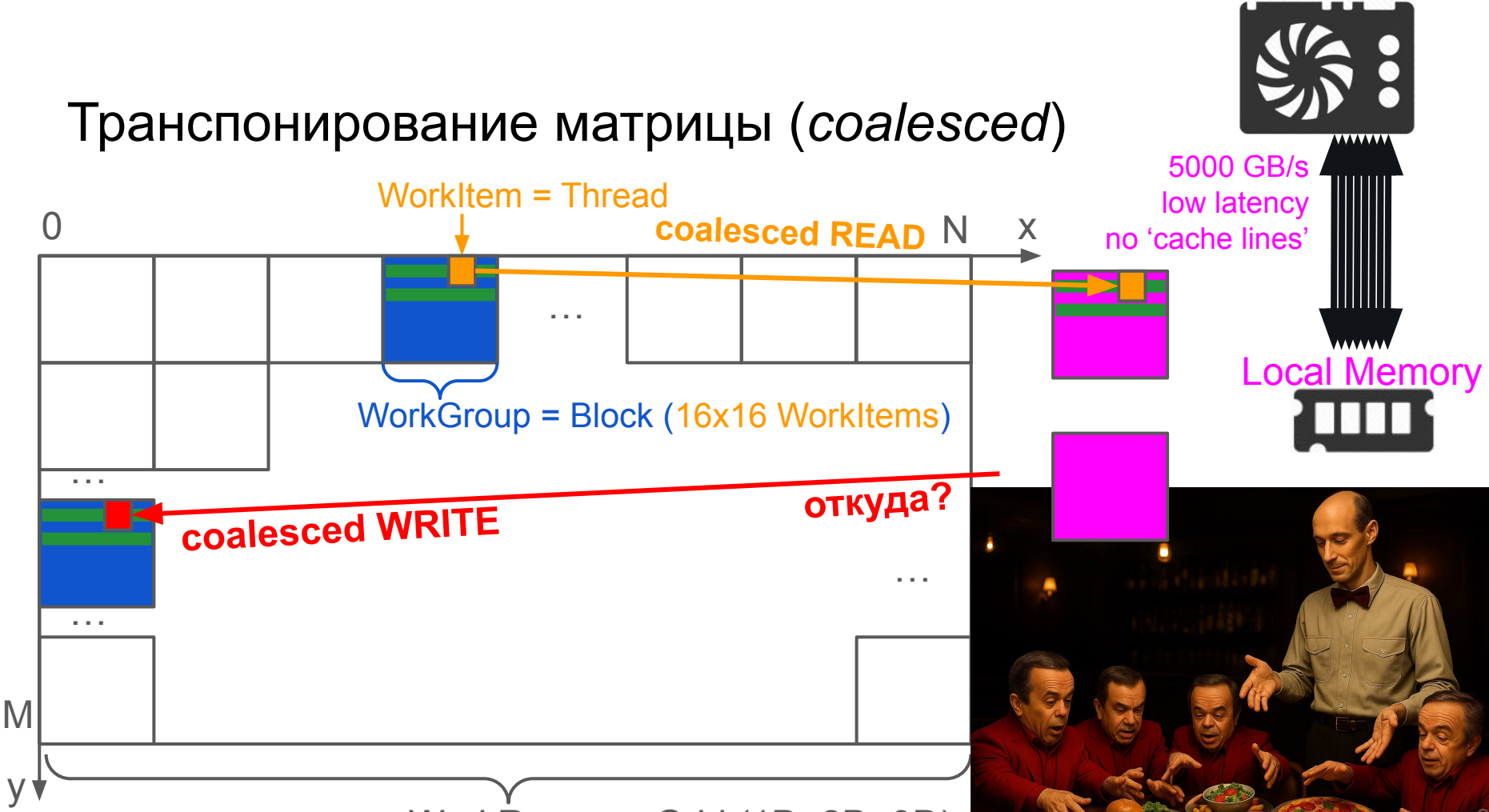
Можем ли мы исправить это?



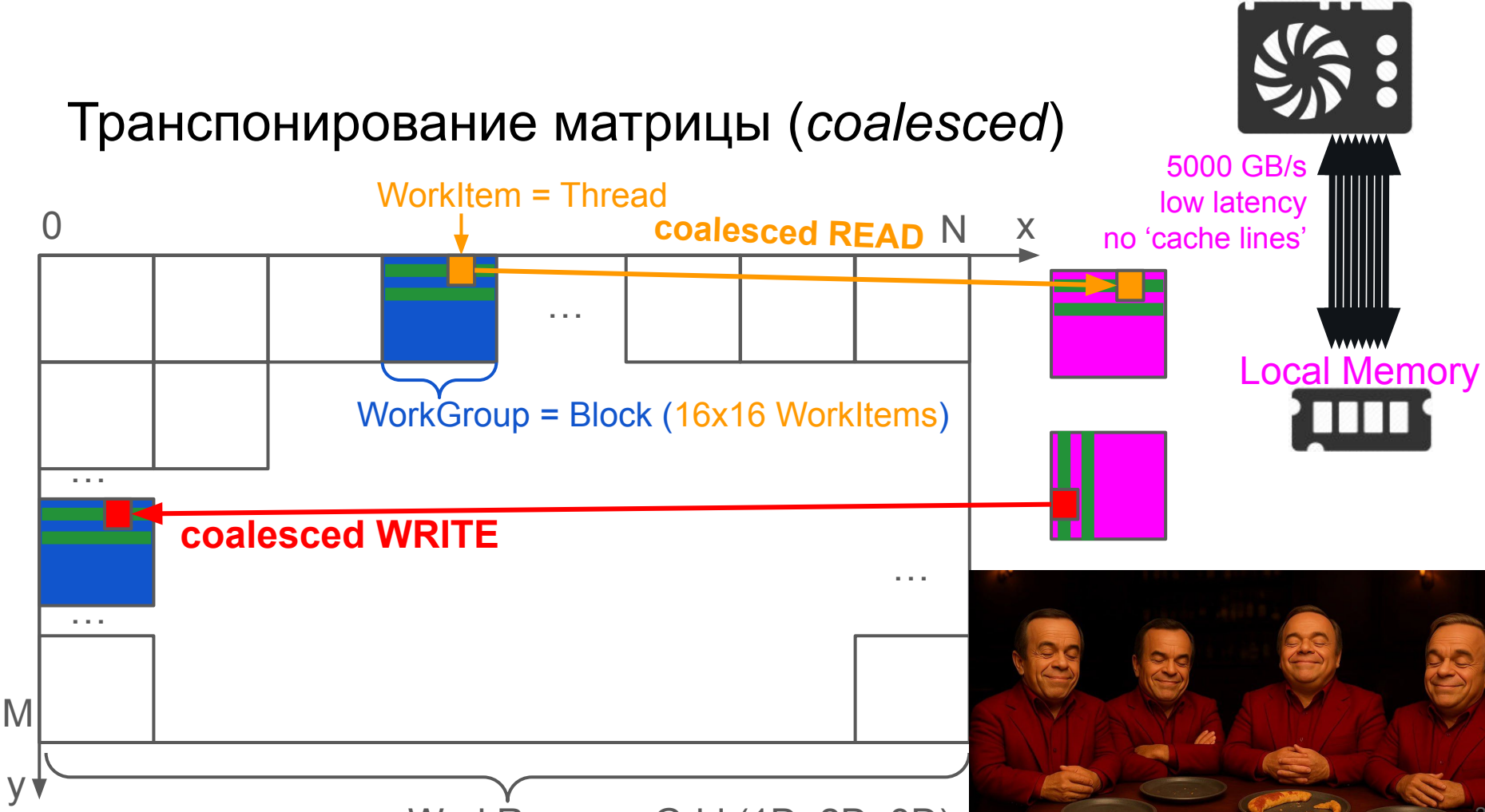
Транспонирование матрицы (*coalesced*)



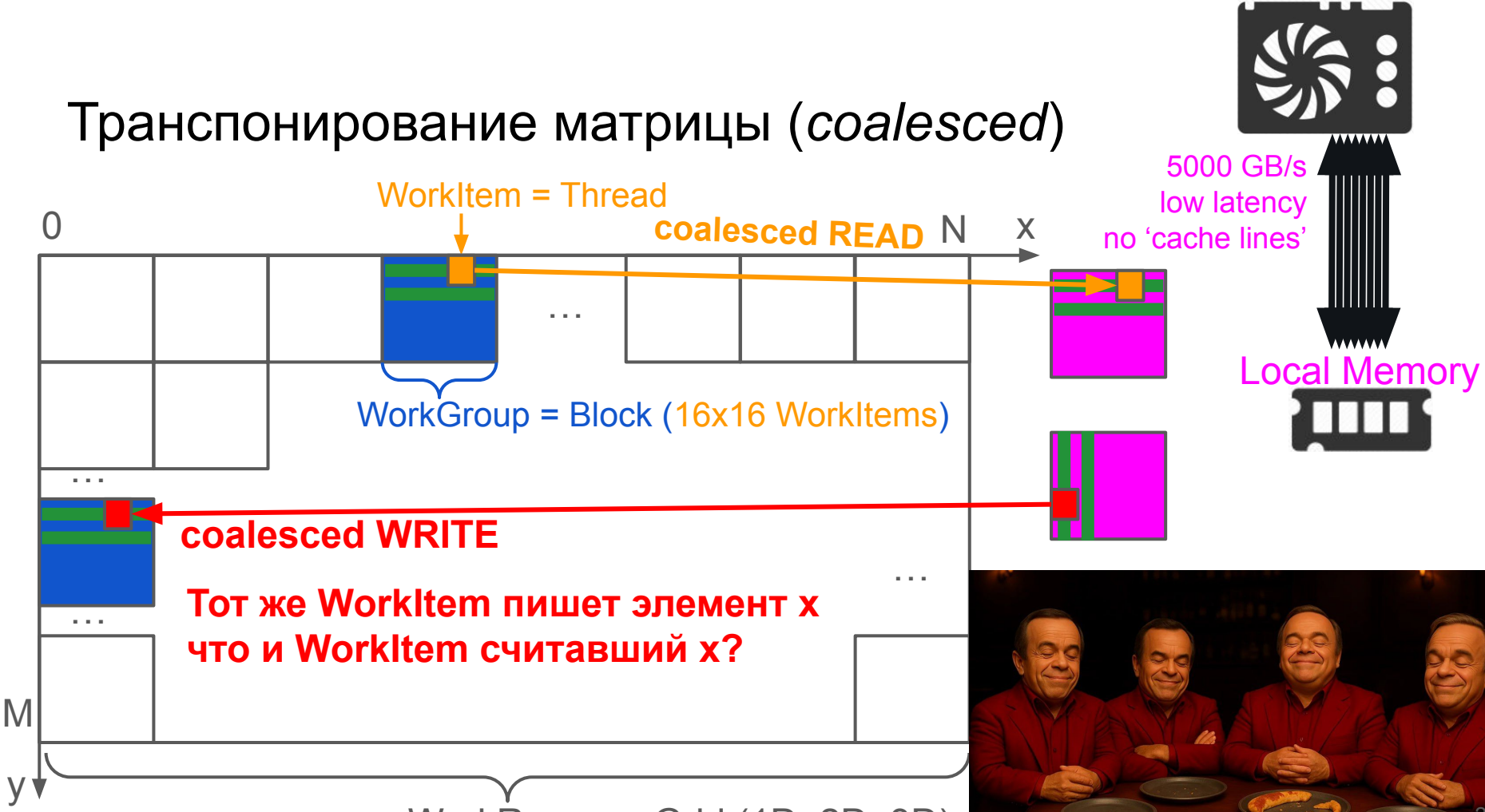
Транспонирование матрицы (*coalesced*)



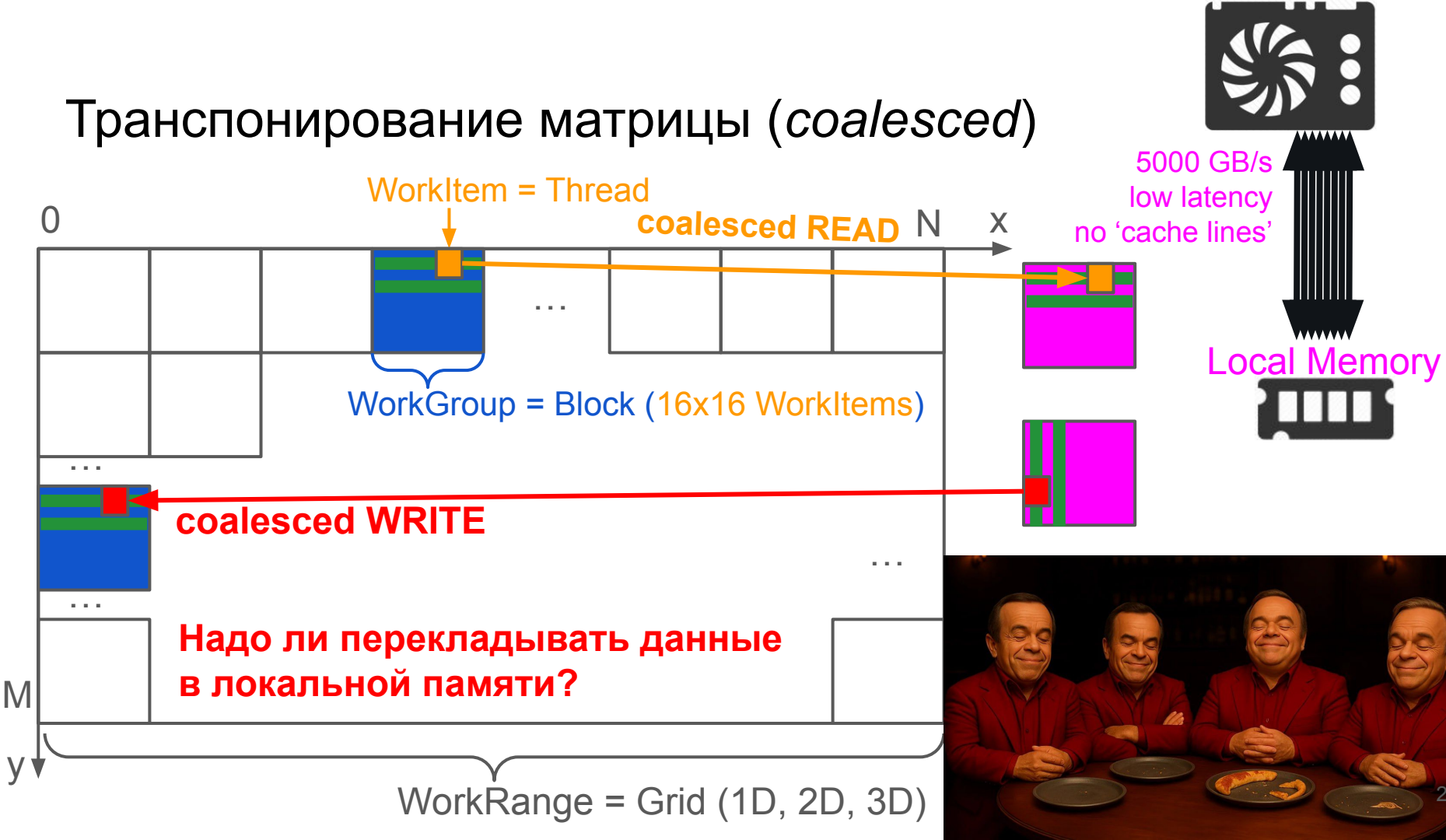
Транспонирование матрицы (*coalesced*)



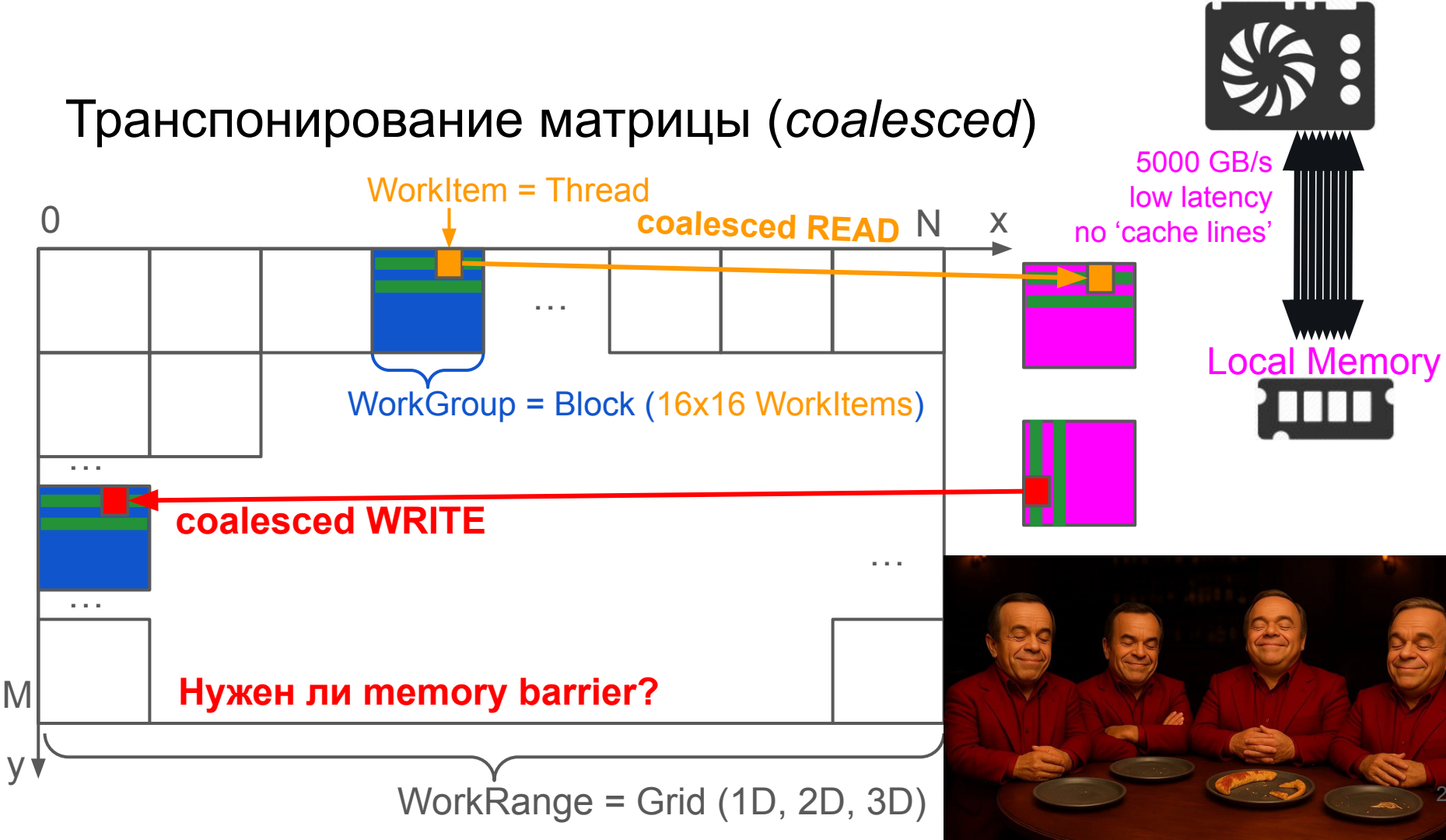
Транспонирование матрицы (*coalesced*)



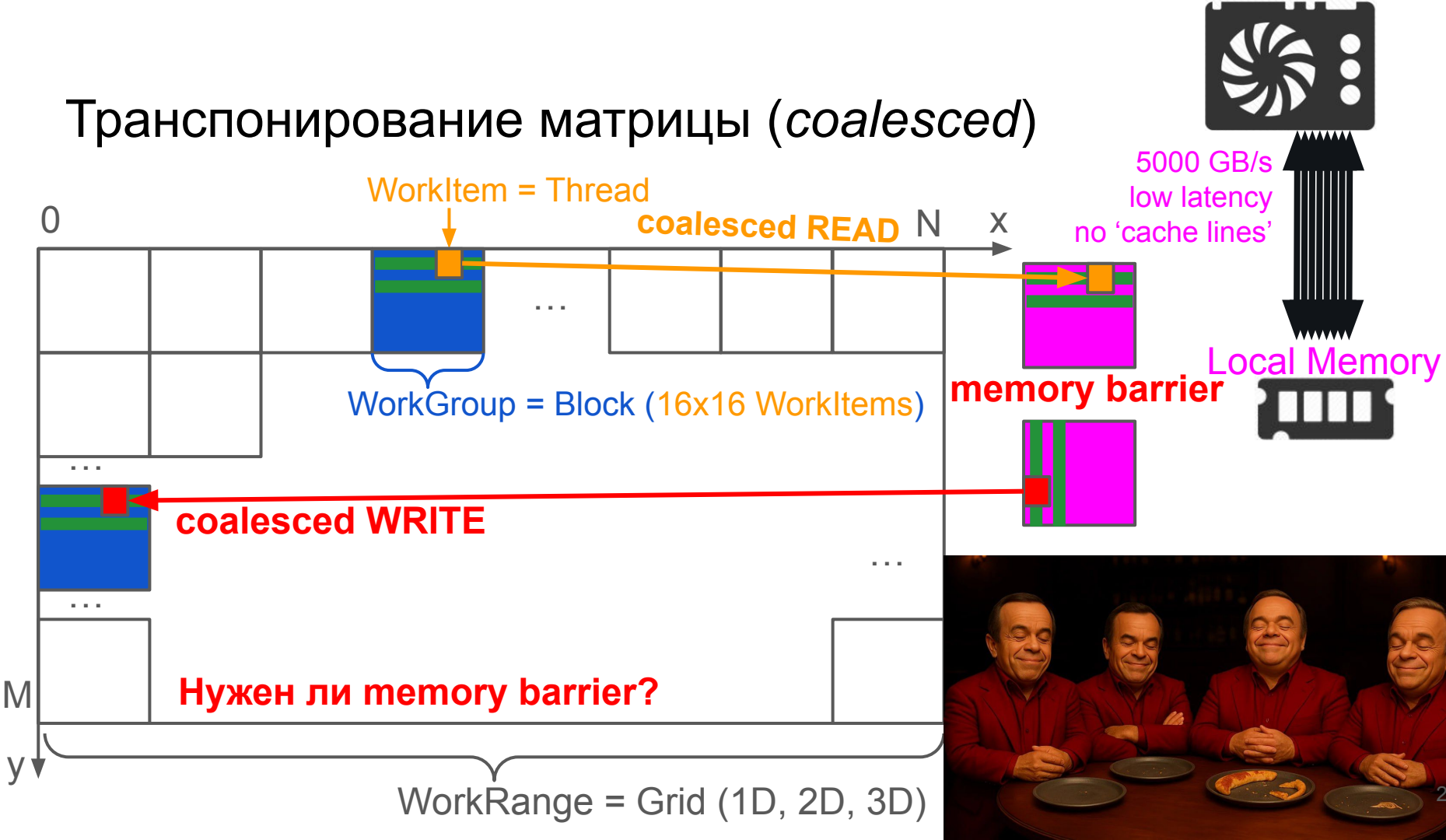
Транспонирование матрицы (*coalesced*)



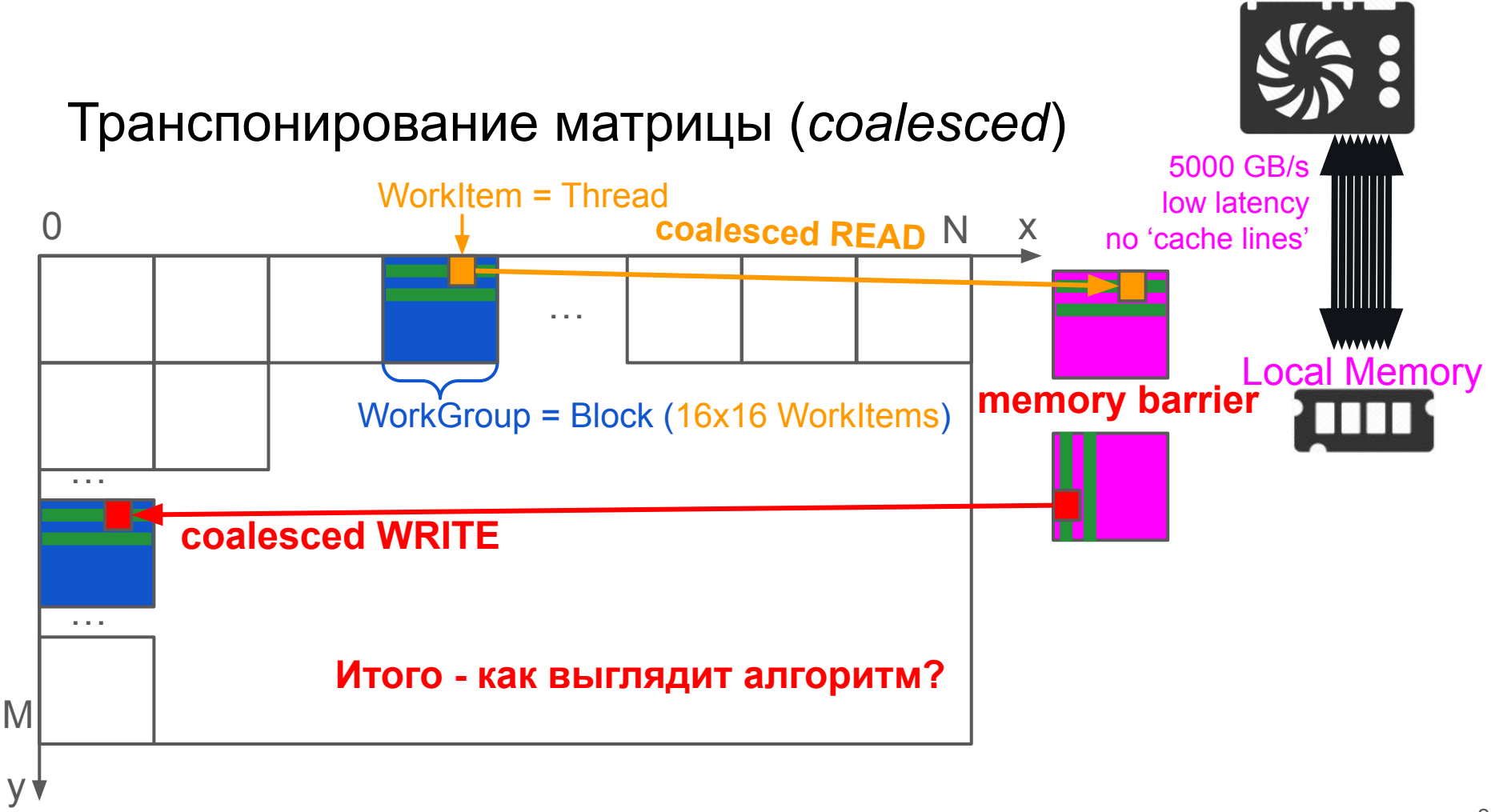
Транспонирование матрицы (*coalesced*)



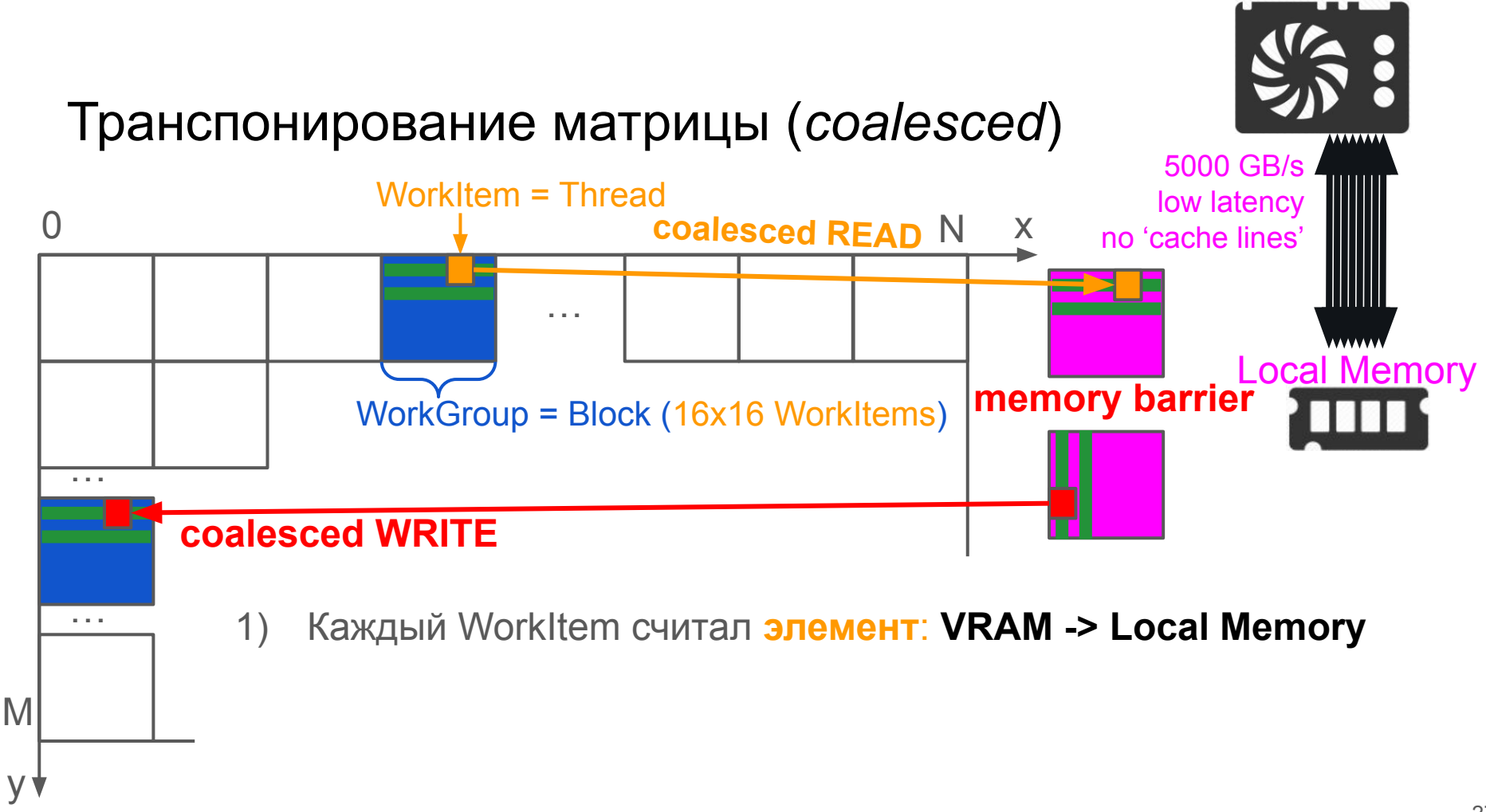
Транспонирование матрицы (*coalesced*)



Транспонирование матрицы (*coalesced*)

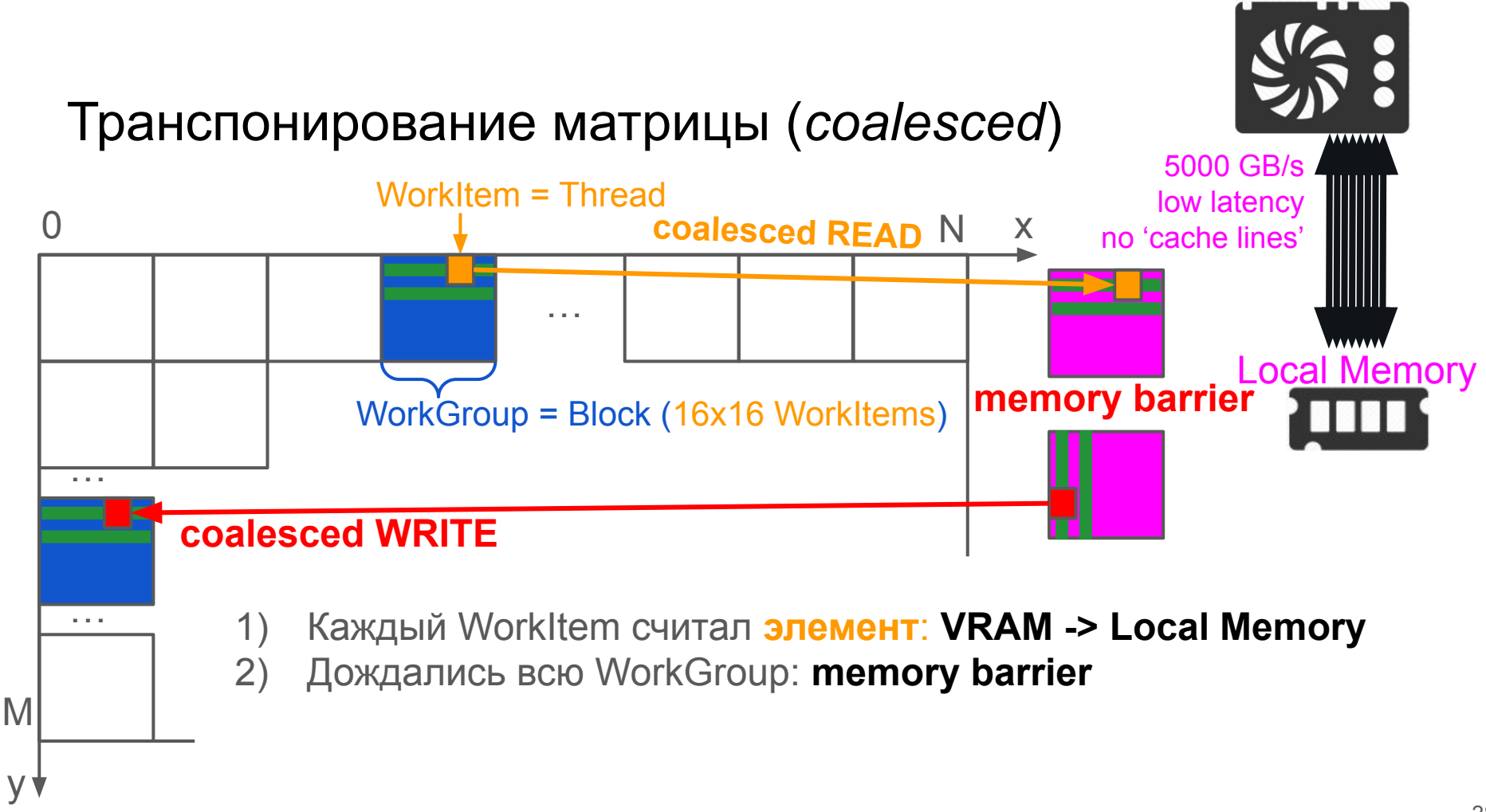


Транспонирование матрицы (*coalesced*)



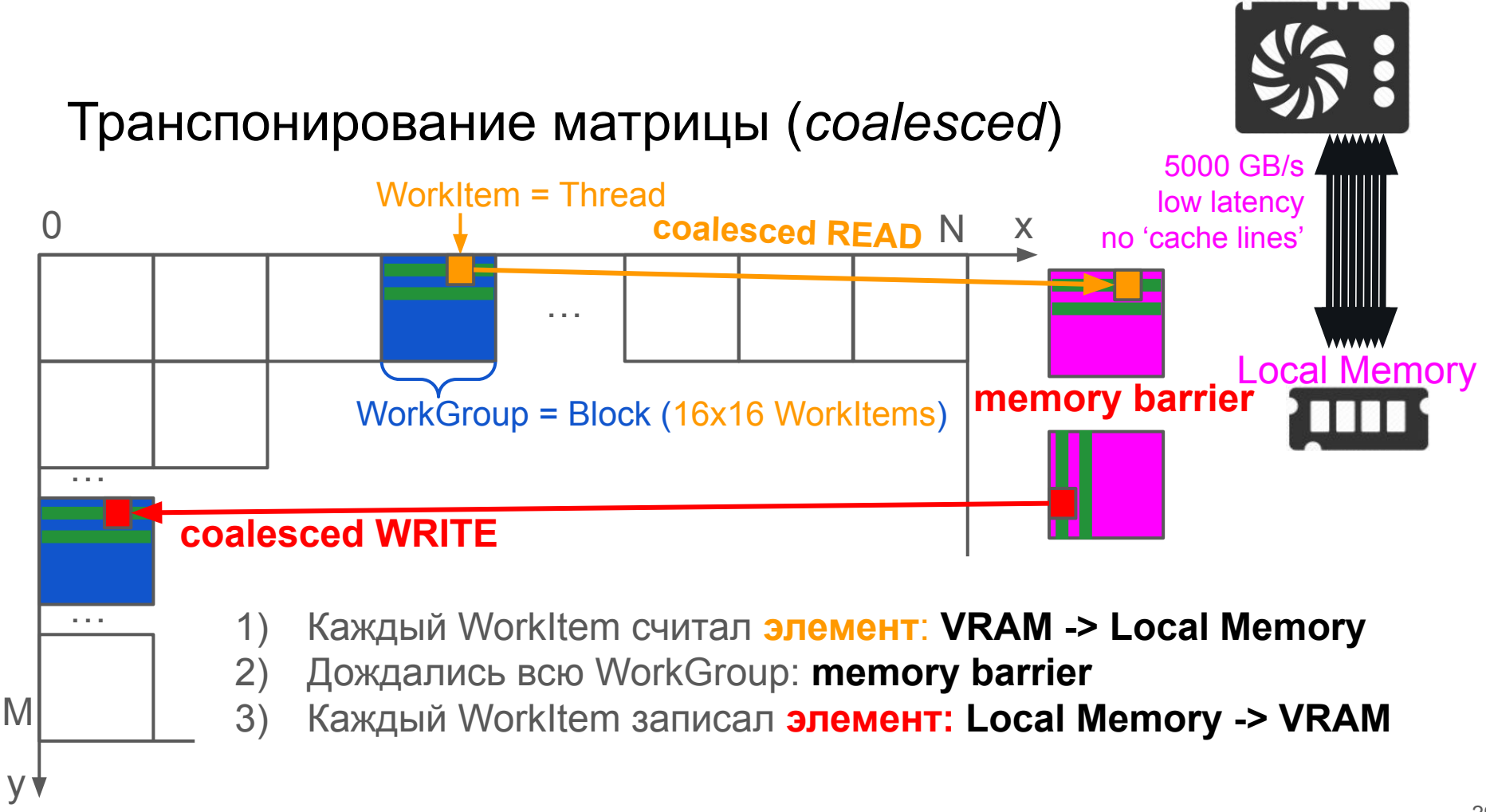
1) Каждый WorkItem считал **элемент**: **VRAM -> Local Memory**

Транспонирование матрицы (*coalesced*)



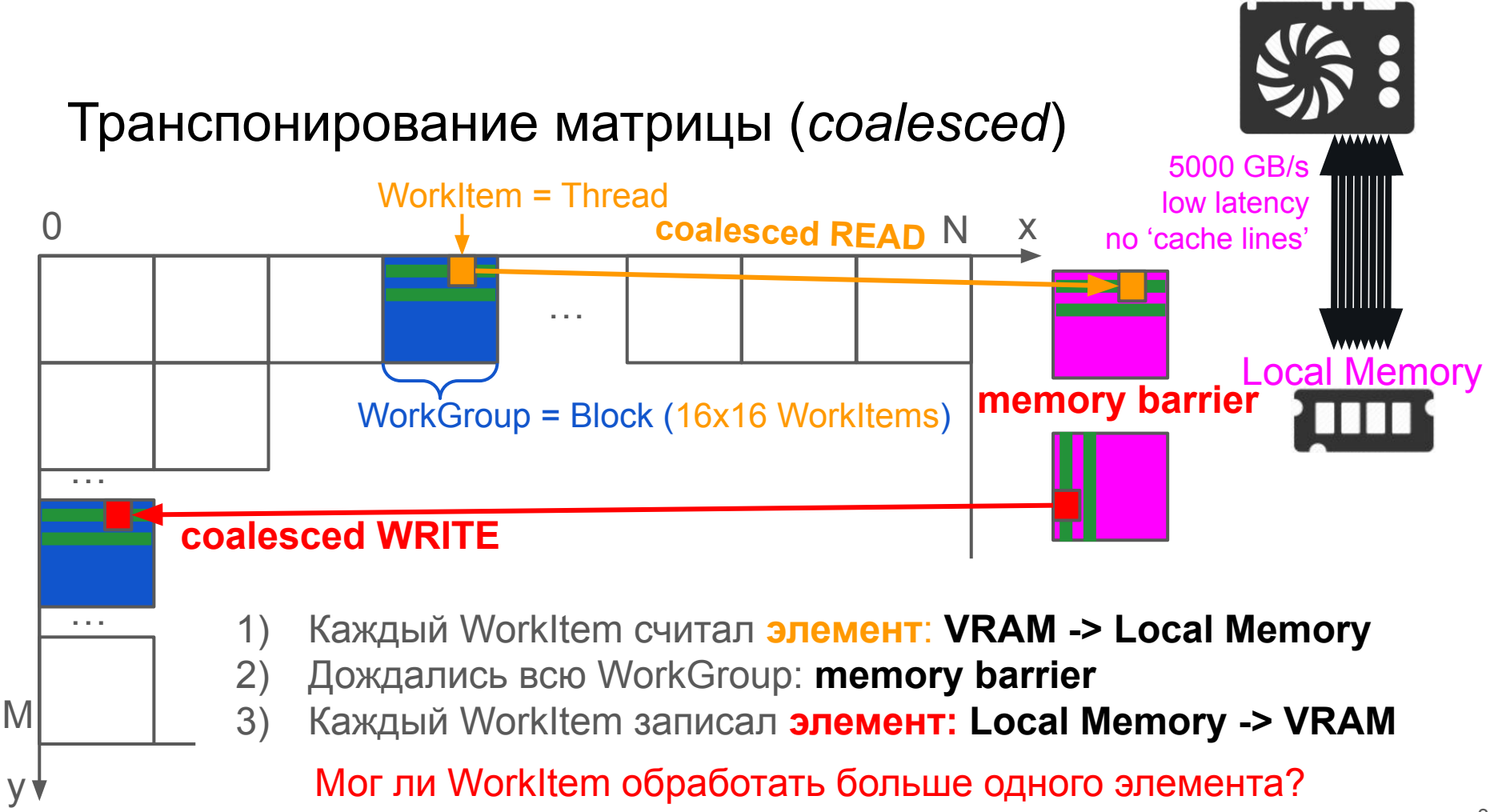
1) Каждый WorkItem считал **элемент**: **VRAM -> Local Memory**
2) Дождались всю WorkGroup: **memory barrier**

Транспонирование матрицы (*coalesced*)



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

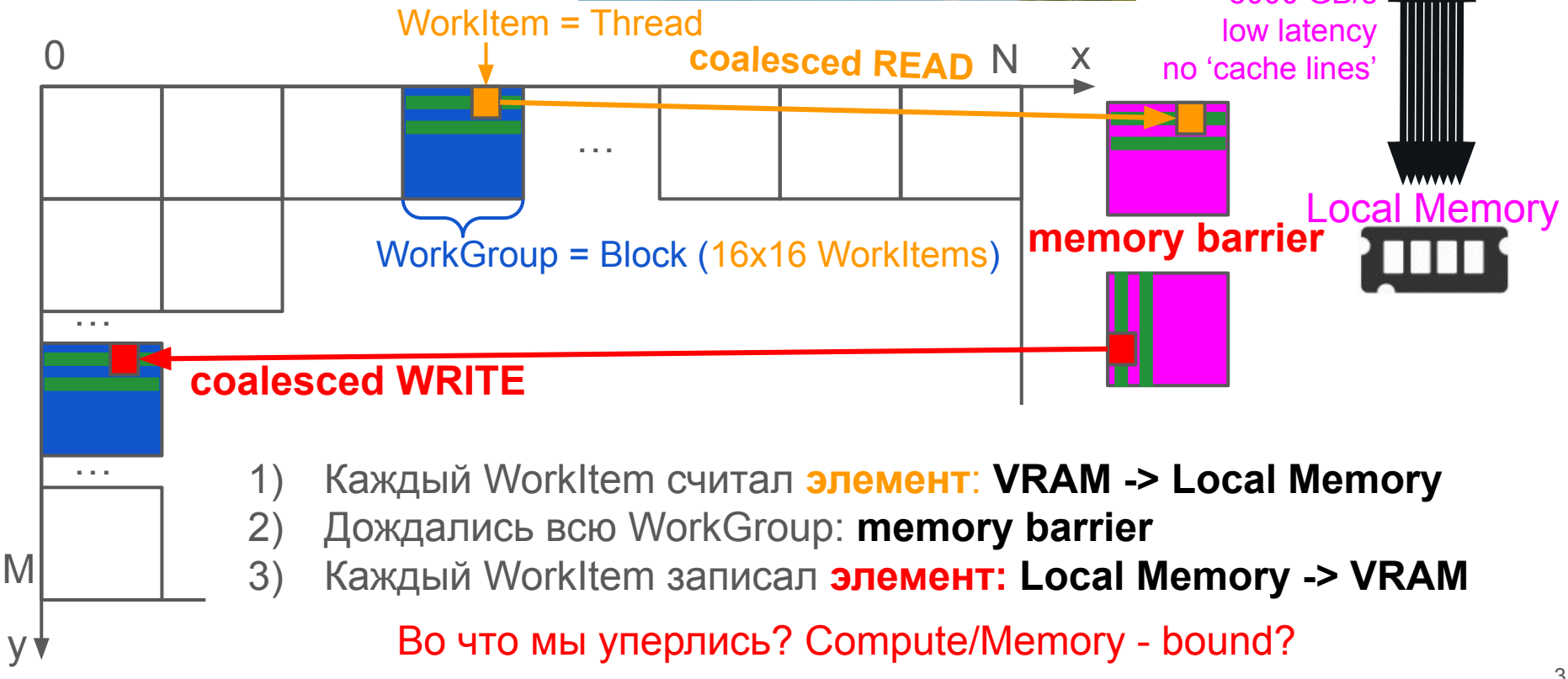
Транспонирование матрицы (*coalesced*)



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

Мог ли WorkItem обработать больше одного элемента?

Транспонирование



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

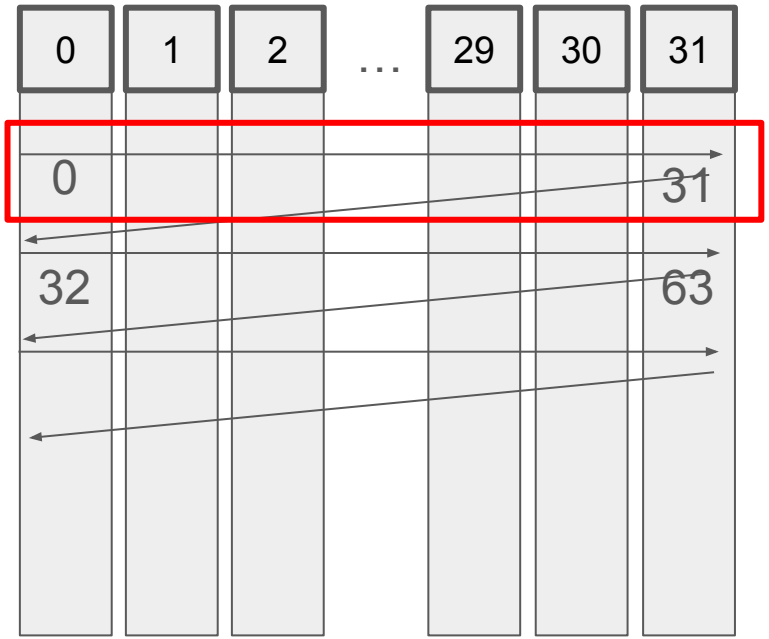
Во что мы уперлись? Compute/Memory - bound?

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:



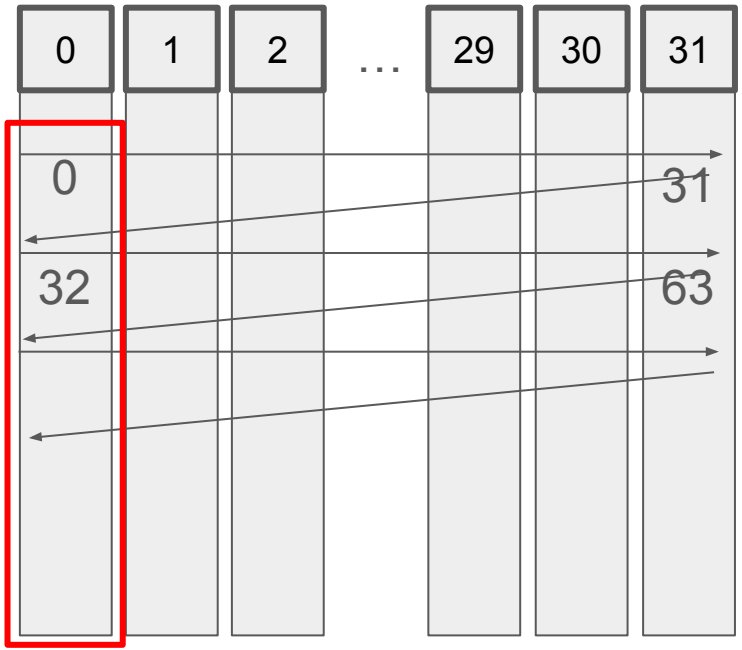
Локальная память - bank conflicts (напоминание)

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

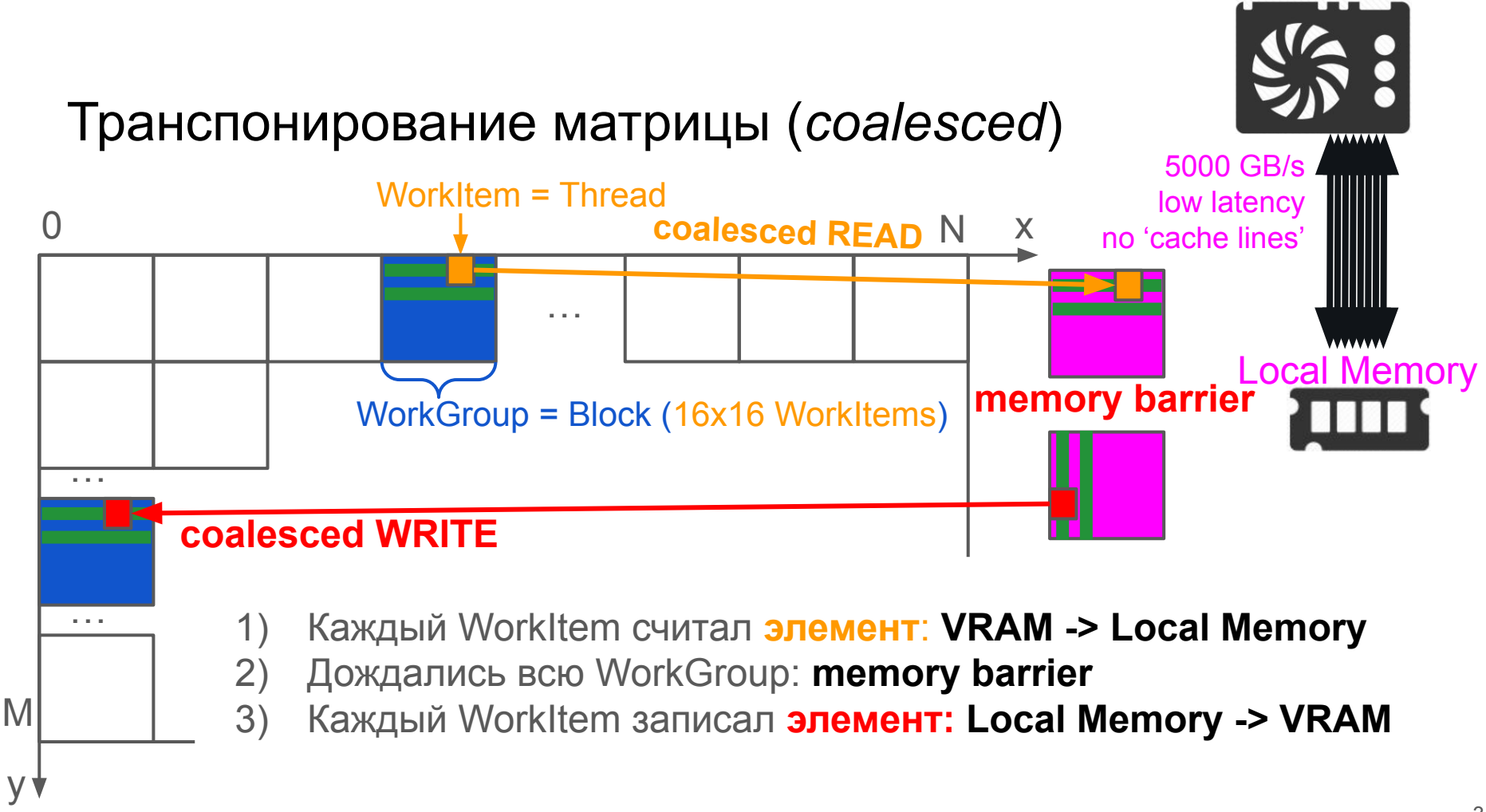
Bank conflict!

Local memory banks:

Local address space:



Транспонирование матрицы (*coalesced*)



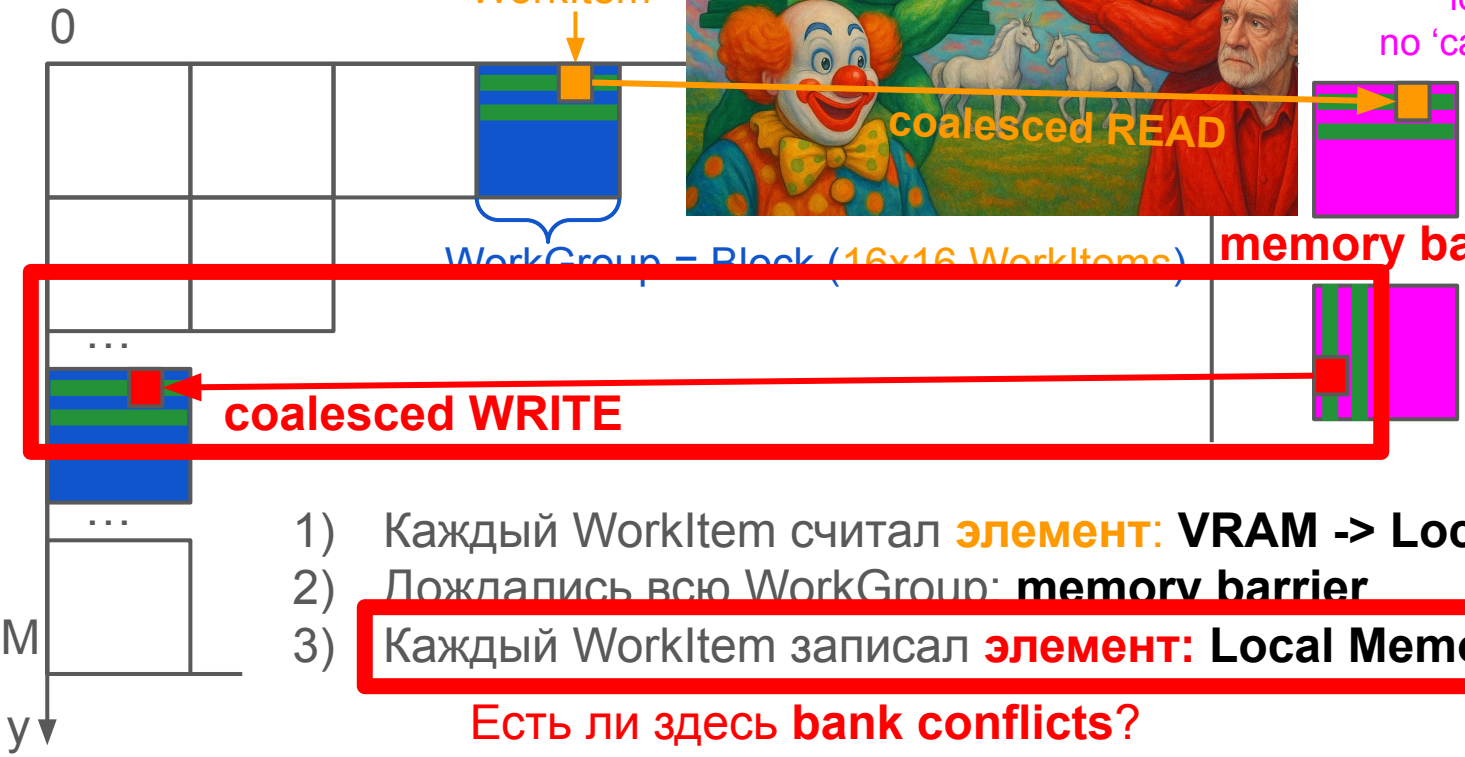
1) Каждый WorkItem считал **элемент**: **VRAM -> Local Memory**
2) Дождались всю WorkGroup: **memory barrier**
3) Каждый WorkItem записал **элемент: Local Memory -> VRAM**

Транспонирование

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**?

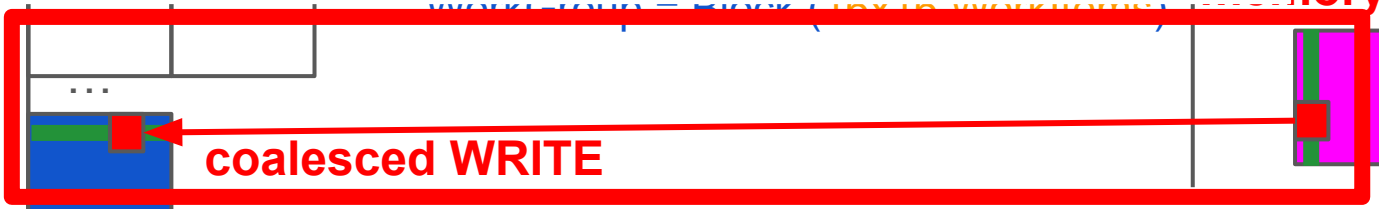
Транспонирование



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

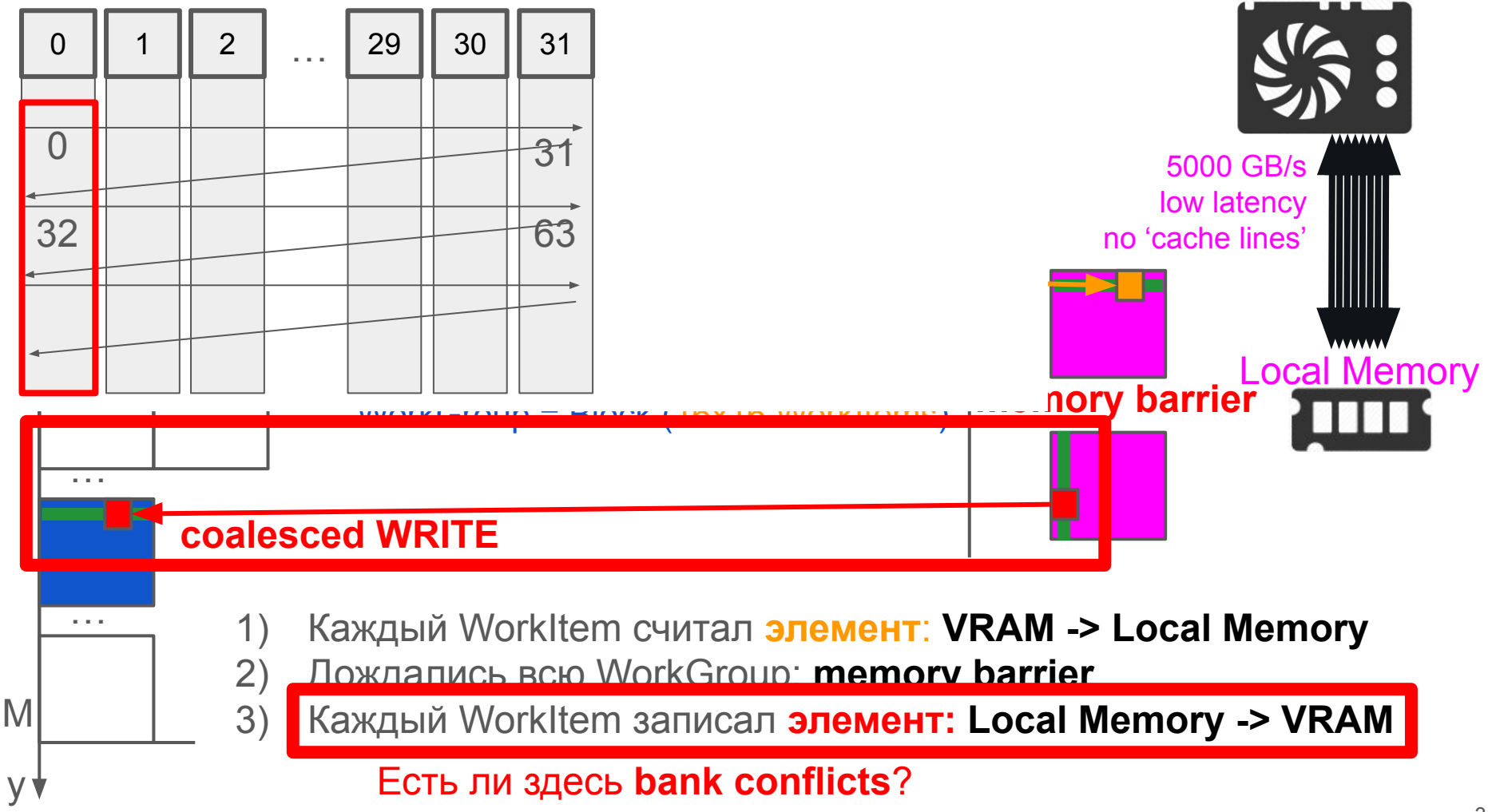
Есть ли здесь **bank conflicts**?

Для простоты будем считать
что WorkGroup **32 x 32!**
И Tile в Local Memory - **32 x 32**!



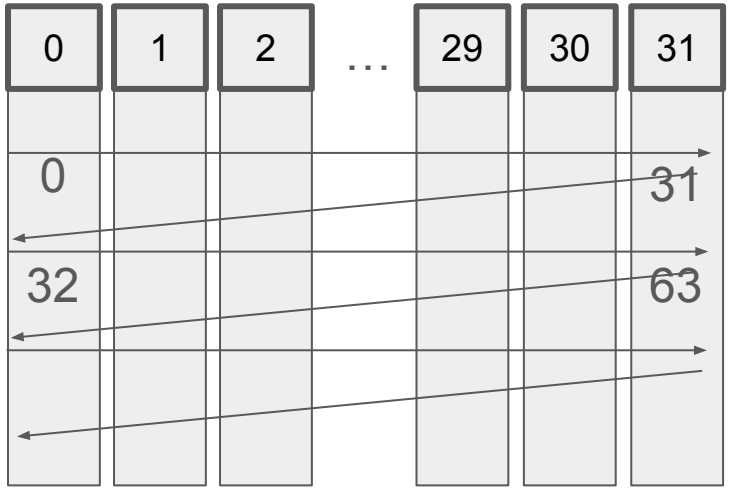
1) Каждый WorkItem считал **элемент**: **VRAM -> Local Memory**
2) Дождались всю WorkGroup: **memory barrier**
3) Каждый WorkItem записал **элемент: Local Memory -> VRAM**

Есть ли здесь **bank conflicts**?

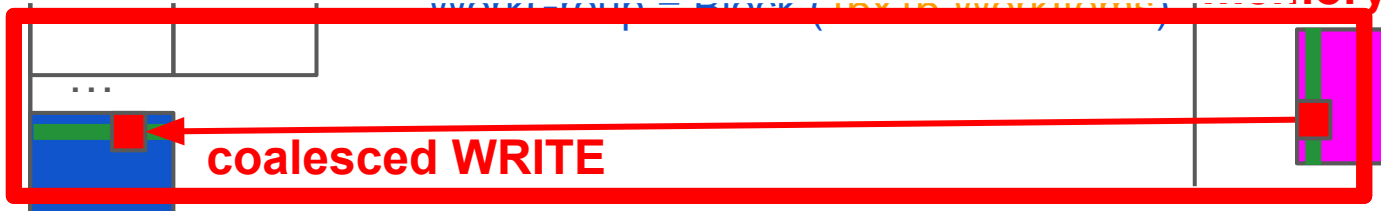


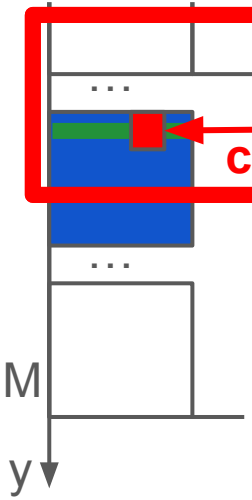
1) Каждый WorkItem считал **элемент**: **VRAM -> Local Memory**
2) Дождались всю WorkGroup: **memory barrier**
3) Каждый WorkItem записал **элемент: Local Memory -> VRAM**

Есть ли здесь **bank conflicts**?



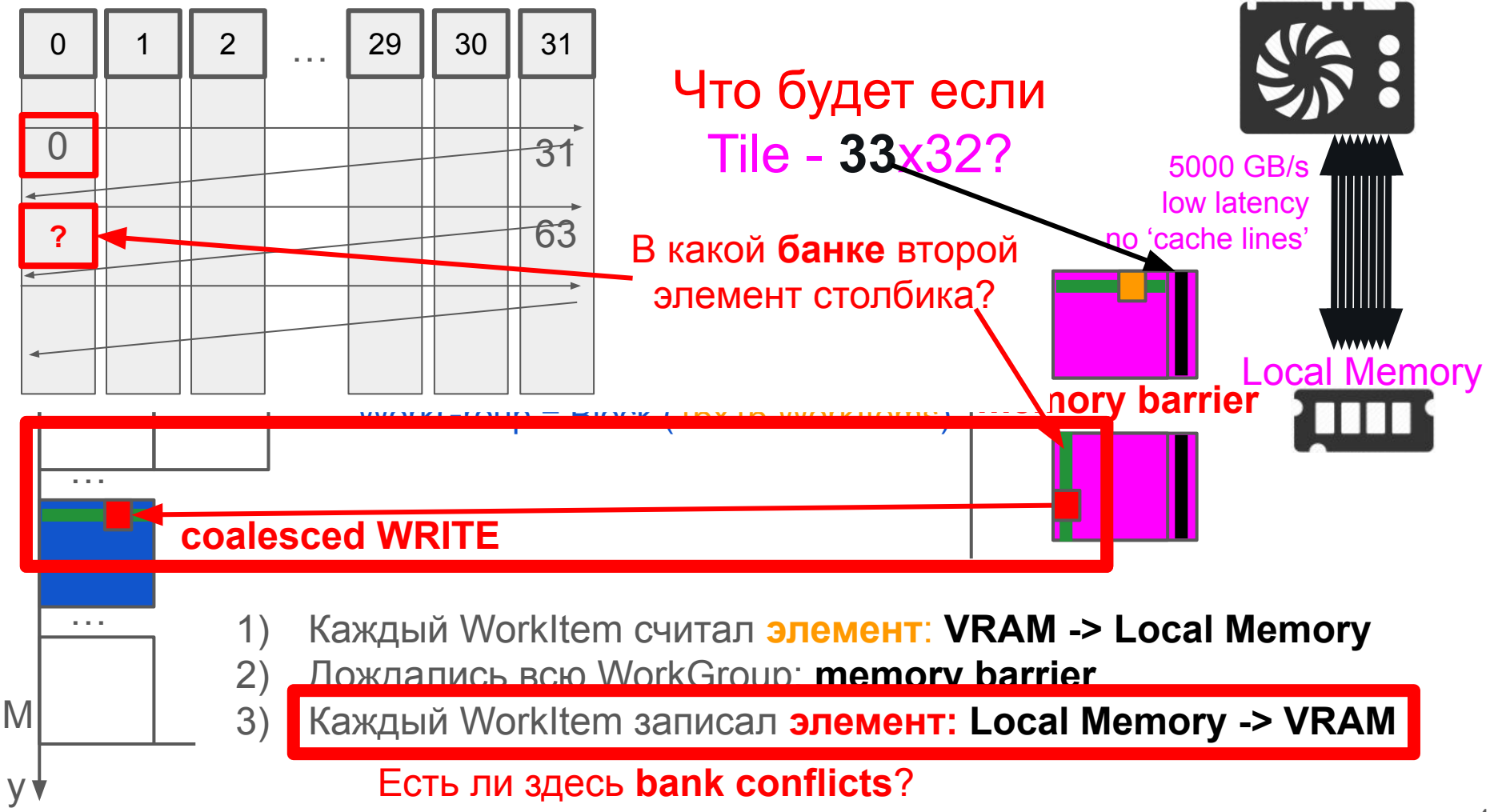
Что будет если
Tile - **33**x32?





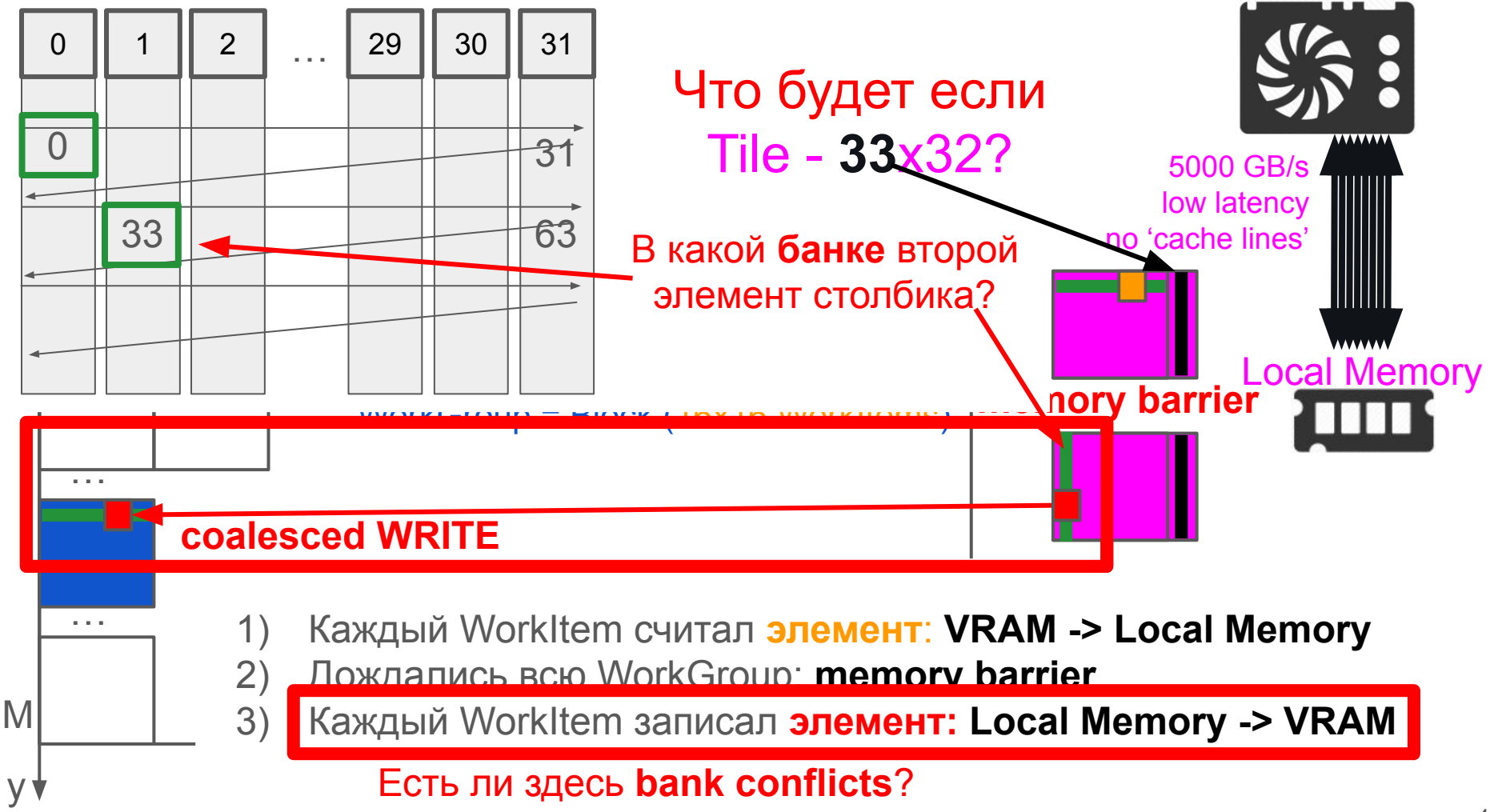
1) Каждый WorkItem считал **элемент**: **VRAM -> Local Memory**
2) Дождались всю WorkGroup: **memory barrier**
3) Каждый WorkItem записал **элемент: Local Memory -> VRAM**

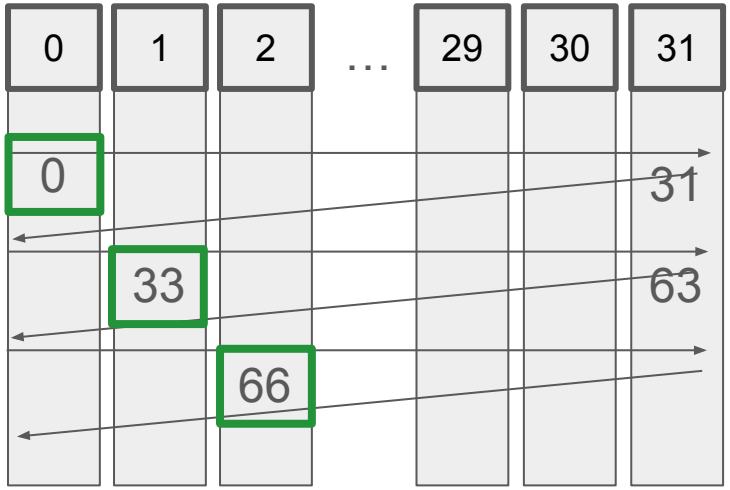
Есть ли здесь **bank conflicts**?



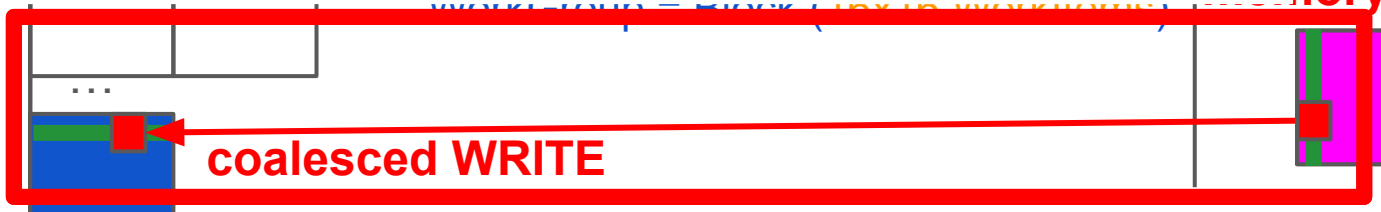
1) Каждый WorkItem считал элемент: **VRAM -> Local Memory**
2) Дождались всю WorkGroup: **memory barrier**
3) Каждый WorkItem записал элемент: **Local Memory -> VRAM**

Есть ли здесь **bank conflicts**?


0
1
2
…
29
30
31
0
31
33
63
Что будет если
Tile - 33x32?
В какой банке второй
элемент столбика?
5000 GB/s
low latency
no ‘cache lines’
Local Memory
nory barrier
coalesced WRITE
…
…
M
y
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?

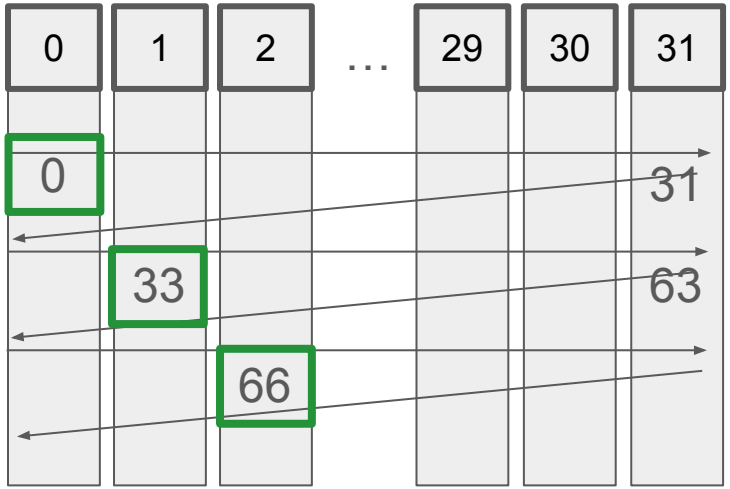


Что будет если
Tile - **33**x32?

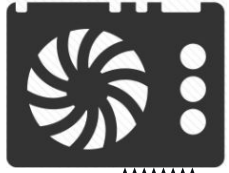


1) Каждый WorkItem считал **элемент**: **VRAM -> Local Memory**
2) Дождались всю WorkGroup: **memory barrier**
3) Каждый WorkItem записал **элемент: Local Memory -> VRAM**

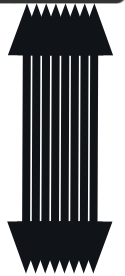
Есть ли здесь **bank conflicts**?



Что будет если
Tile - **33**x32?



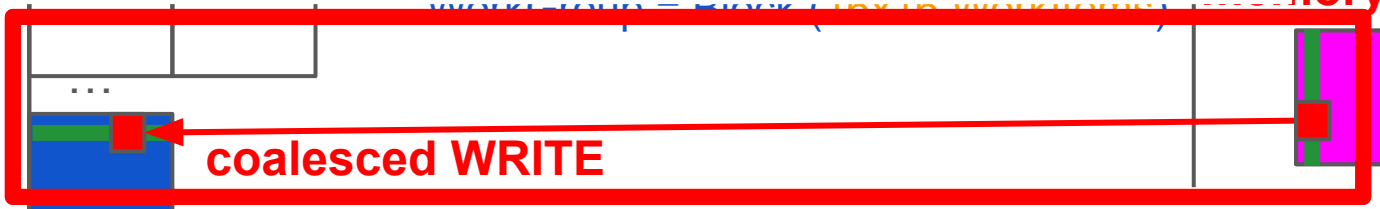
5000 GB/s
low latency
no ‘cache lines’



Можно ли без лишней
local memory?

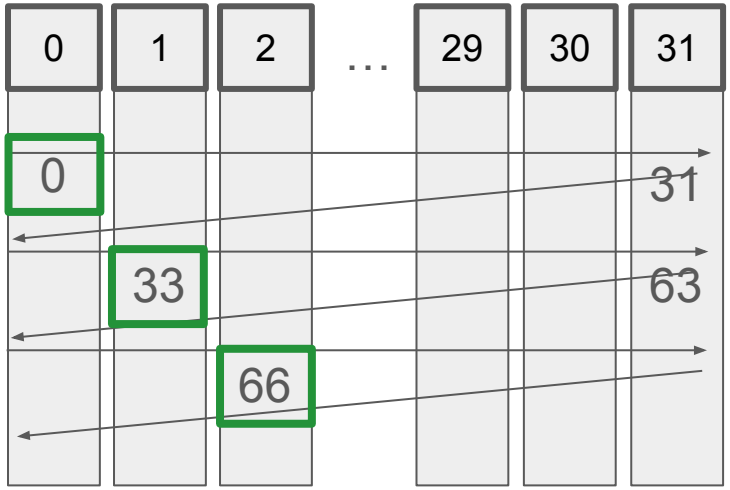
Local Memory

nory barrier



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

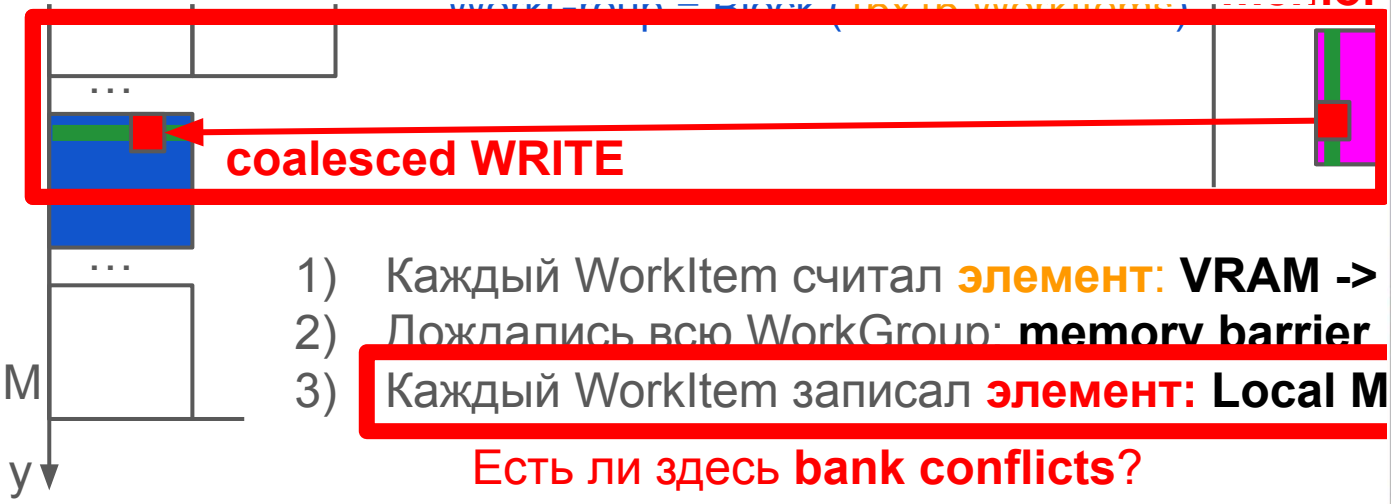
Есть ли здесь **bank conflicts**?



Что будет если
Tile - **33**x32?

А на что влияет
объем используемой
Local Memory?

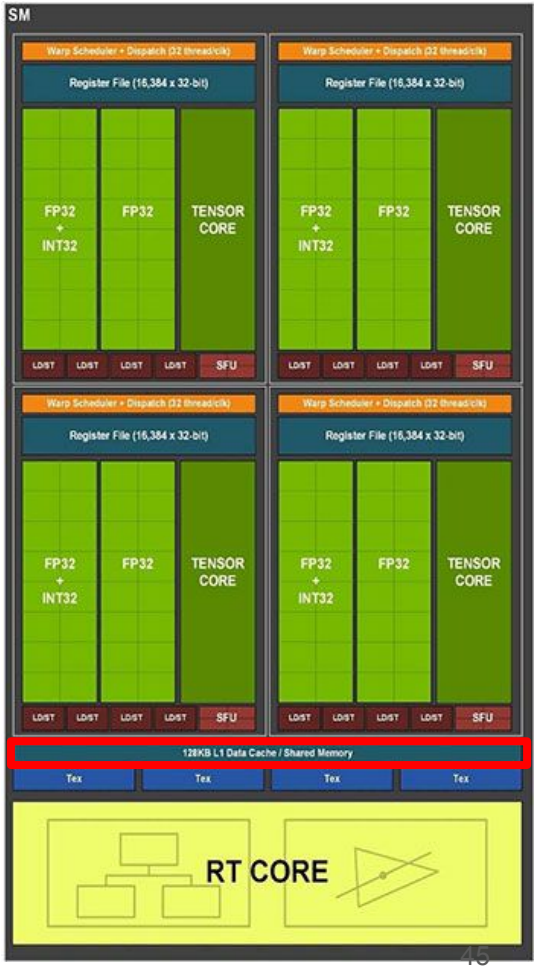
SM - Streaming Multiprocessor



1) Каждый WorkItem считал **элемент: VRAM ->**
2) Дождались 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?

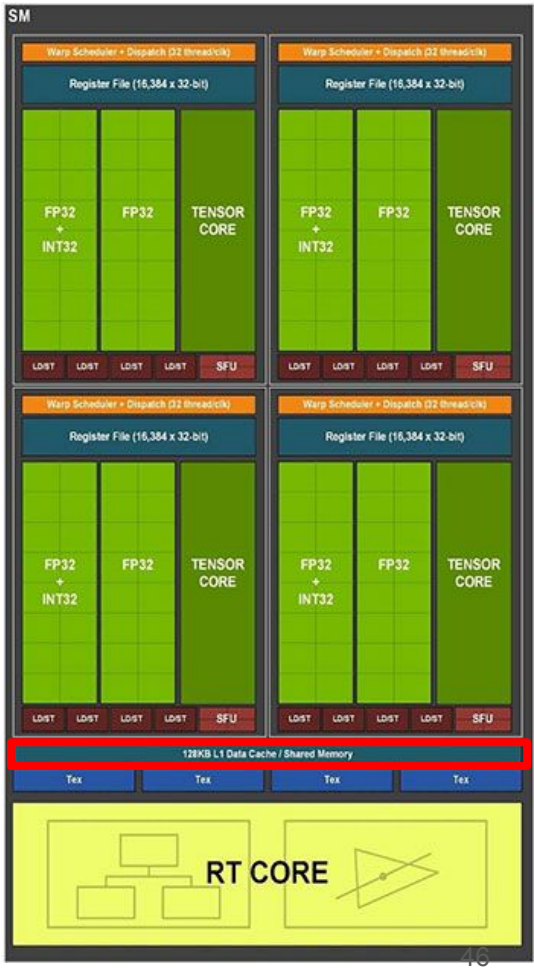
Occupancy!

RTX 3090 (Ampere) - per **SM**:

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

А на что влияет
объем используемой
Local Memory?
Occupancy!

SM - Streaming Multiprocessor



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:		
Threads Per Block	256	(Help)
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:		
Active Threads per Multiprocessor	1536	(Help)
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	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	128
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

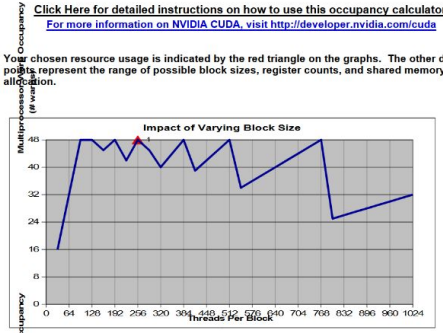
Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	8	48	6
Registers (Warp limit per SM due to per-warp reg count)	8	64	8
Shared Memory (Bytes)	2048	65536	32

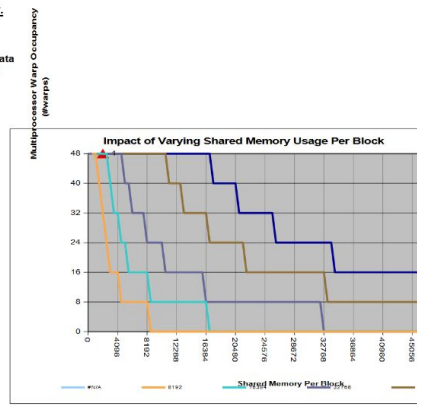
Note: SM is an abbreviation for (Streaming) Multiprocessor

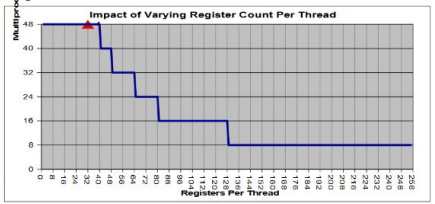
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	6	8	48

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.



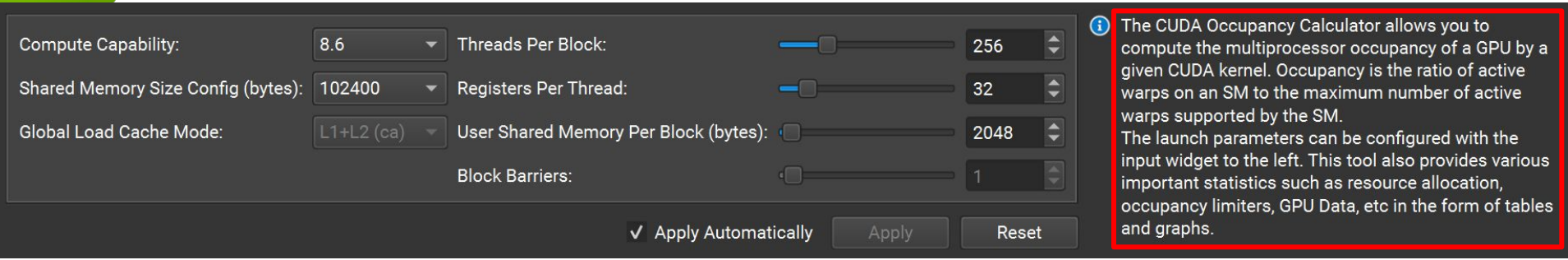




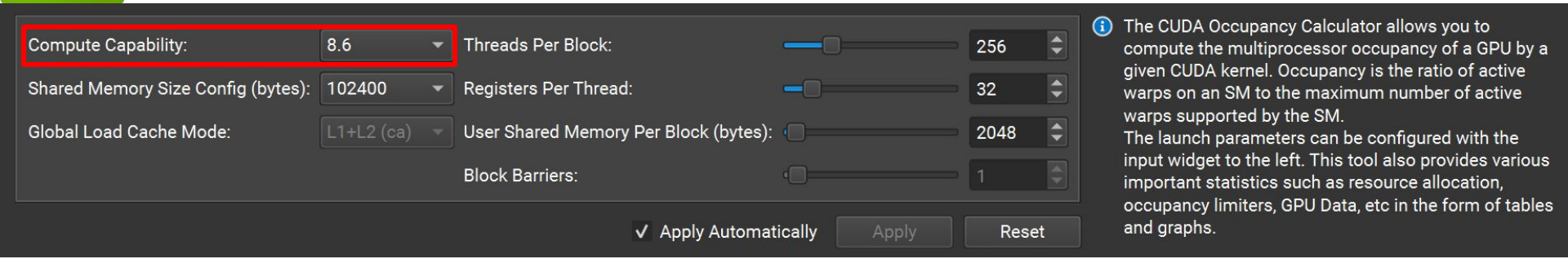
SM - Streaming Multiprocessor

NVIDIA Nsight

Occupancy Calculator

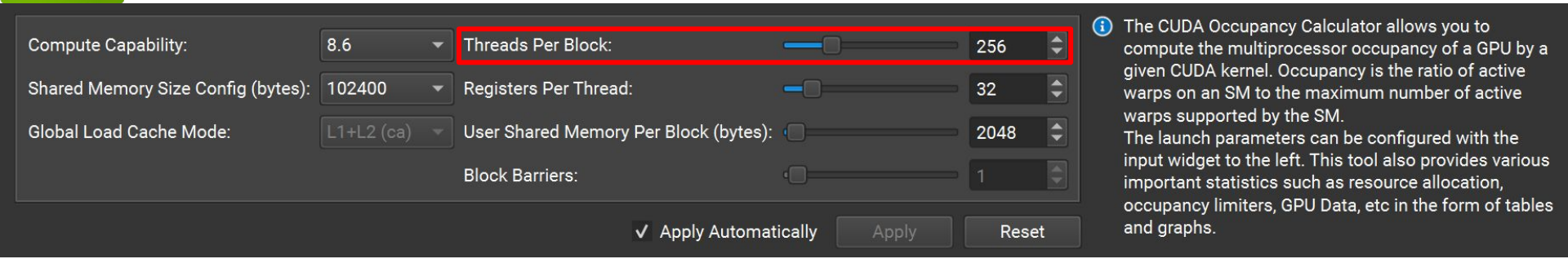


NVIDIA Nsight Occupancy Calculator



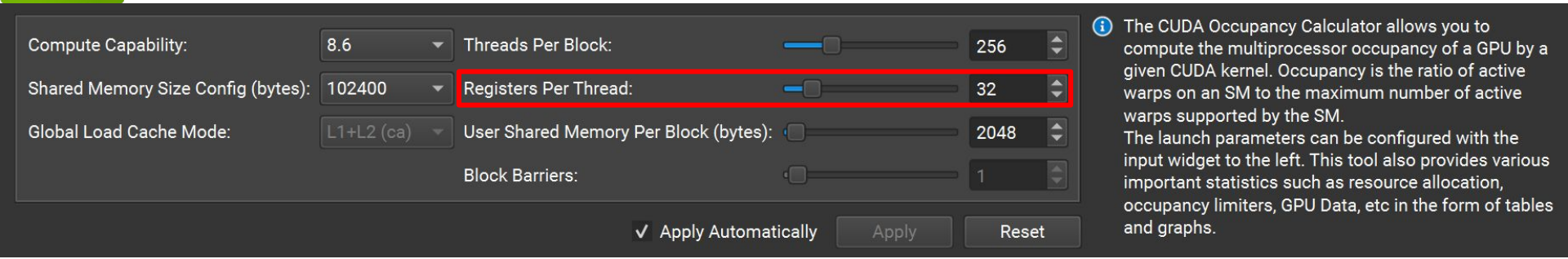
RTX 3090 (Ampere) - Compute Capability **8.6**

NVIDIA Nsight Occupancy Calculator



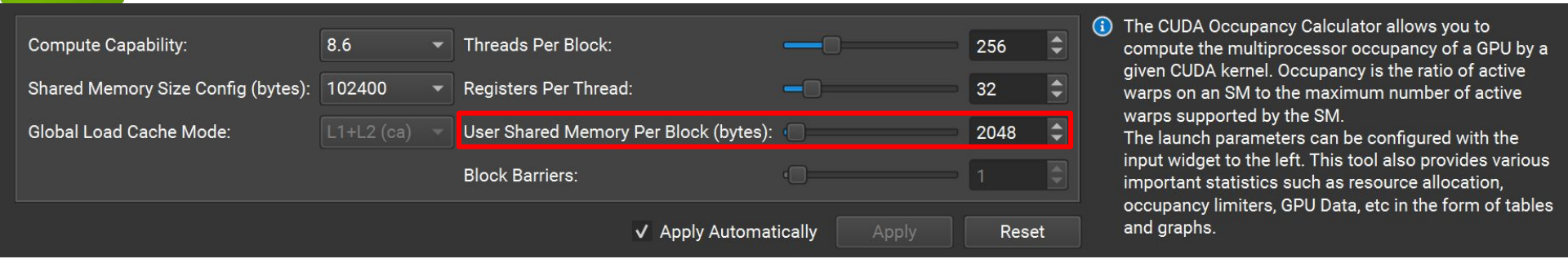
RTX 3090 (Ampere) - Compute Capability **8.6**
WorkGroup size - 256

NVIDIA Nsight Occupancy Calculator



RTX 3090 (Ampere) - Compute Capability **8.6**
WorkGroup size - 256
Registers Per Thread - 32

NVIDIA Nsight Occupancy Calculator



RTX 3090 (Ampere) - Compute Capability **8.6**
WorkGroup size - 256
Registers Per Thread - 32
Local memory Per Thread - 8 bytes

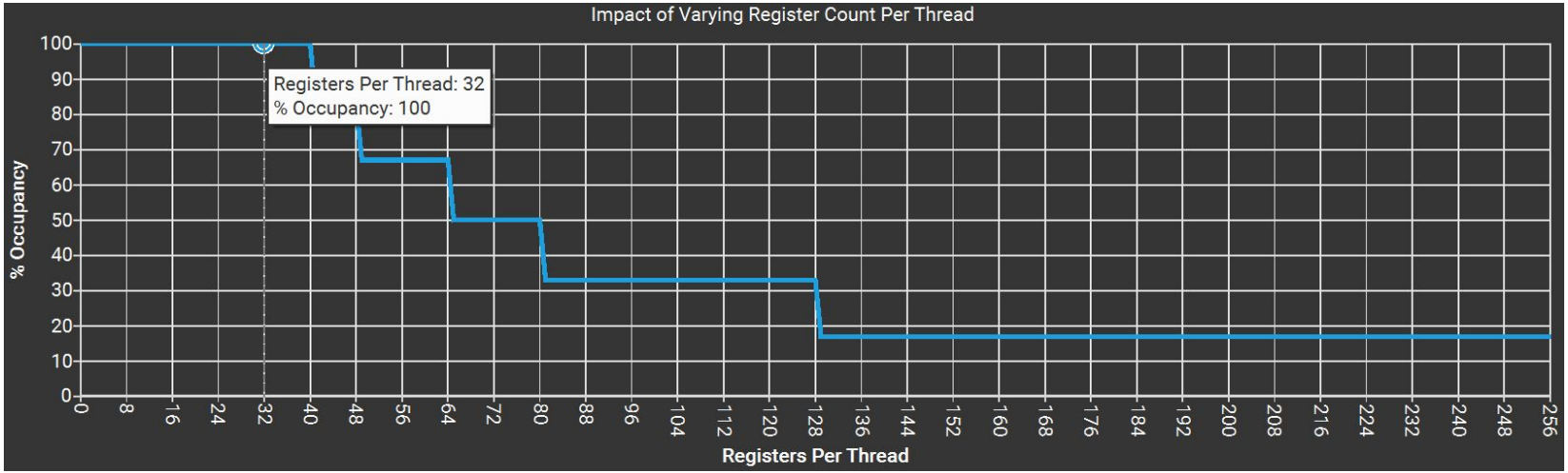
NVIDIA Nsight

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.

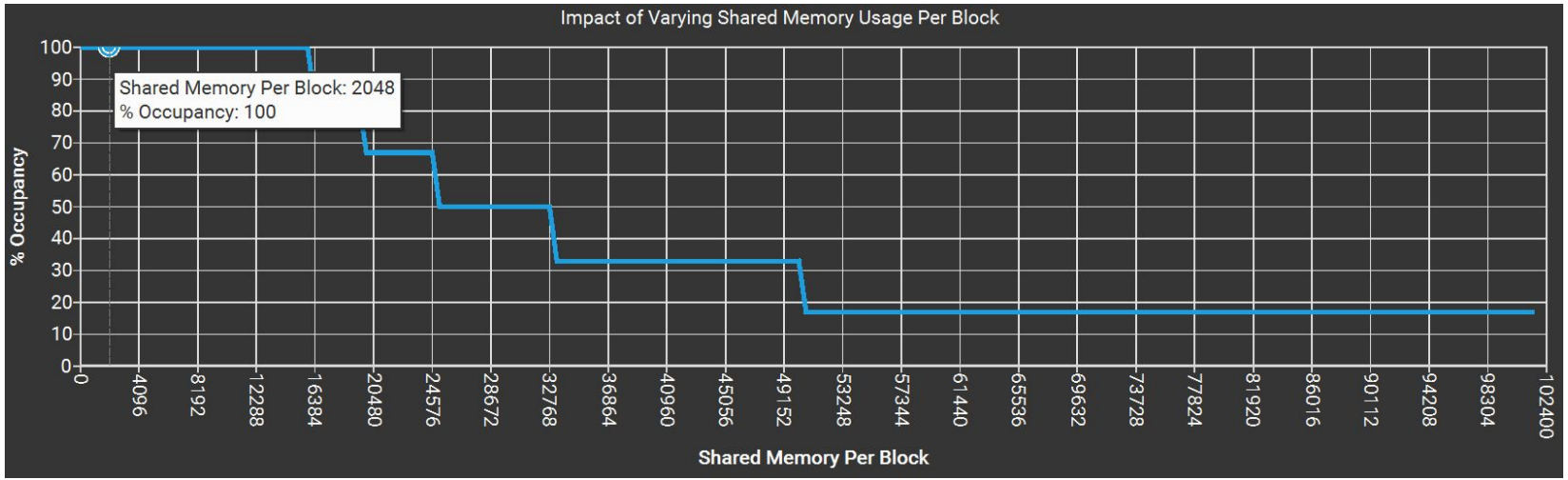


NVIDIA Nsight Occupancy Calculator

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

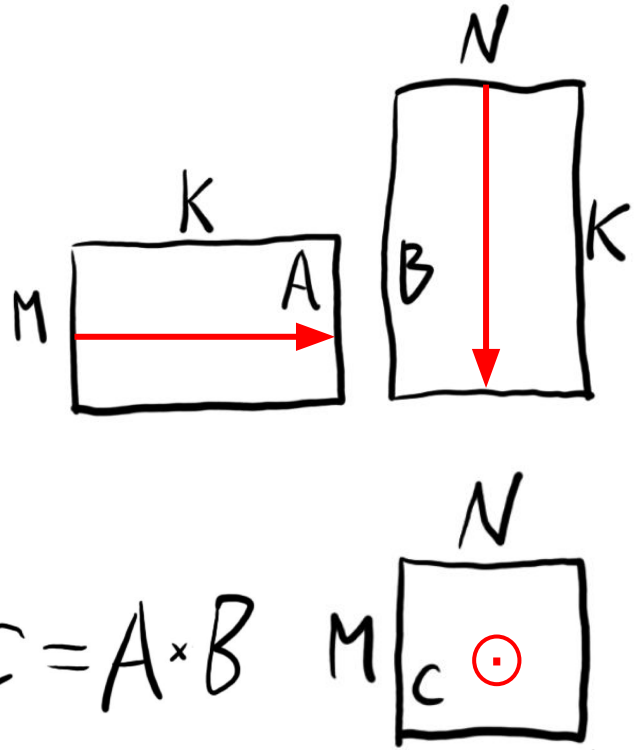
The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA kernel. Occupancy is the ratio of active warps on an SM to the maximum number of active warps supported by the SM.
The launch parameters can be configured with the input widget to the left. This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



Глава 2: Умножение матриц

local memory

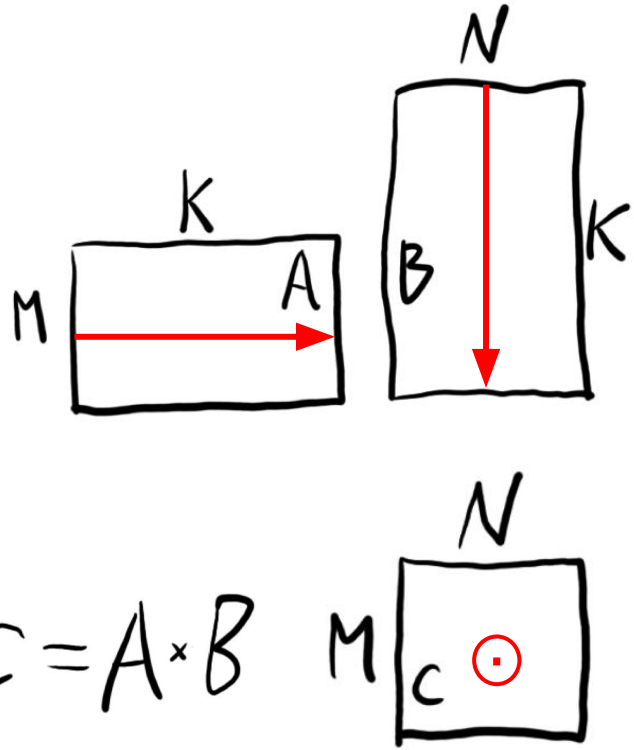
Умножение матриц



Умножение матриц

Как выглядит:

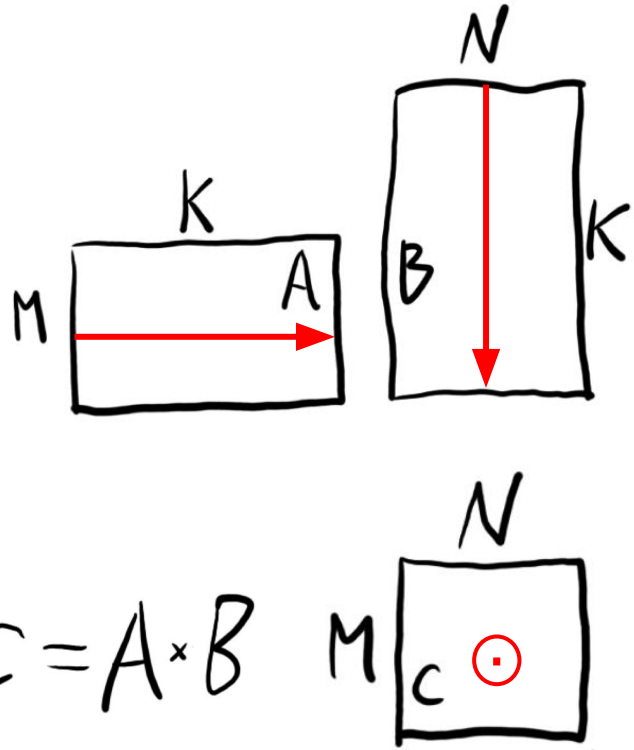
- WorkRange?
- Задача WorkItem?



Умножение матриц

Как выглядит:

- WorkRange?
- Задача WorkItem?
- WorkGroup?

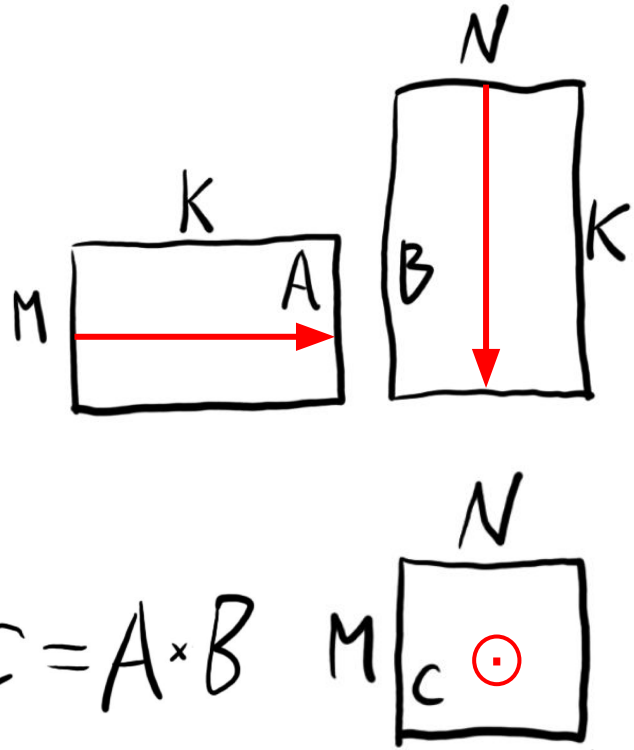


Умножение матриц

Сколько у нас вычислений?

Сколько у нас чтений/записей данных?

Какая пропорция?



Умножение матриц

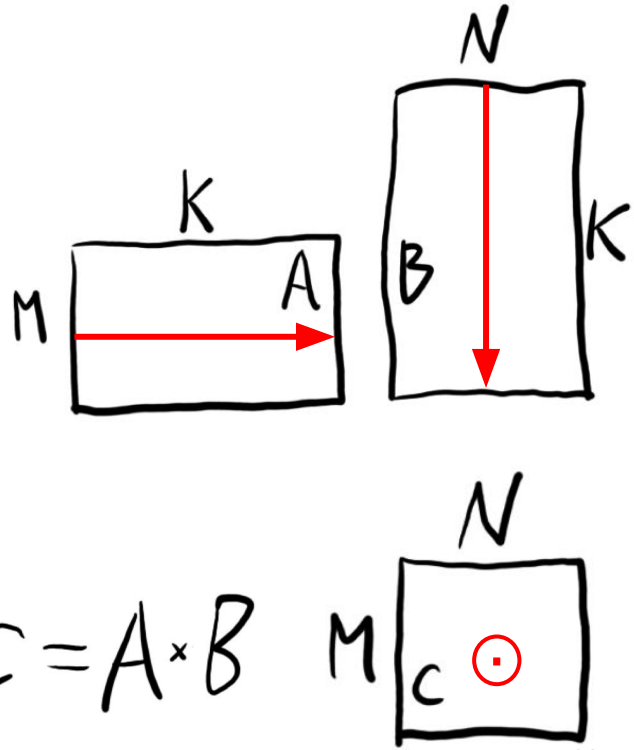
Сколько у нас вычислений?

O(N□M□K)

Сколько у нас чтений/записей данных?

O(N□M□K)

Какая пропорция?



Умножение матриц

Сколько у нас вычислений?

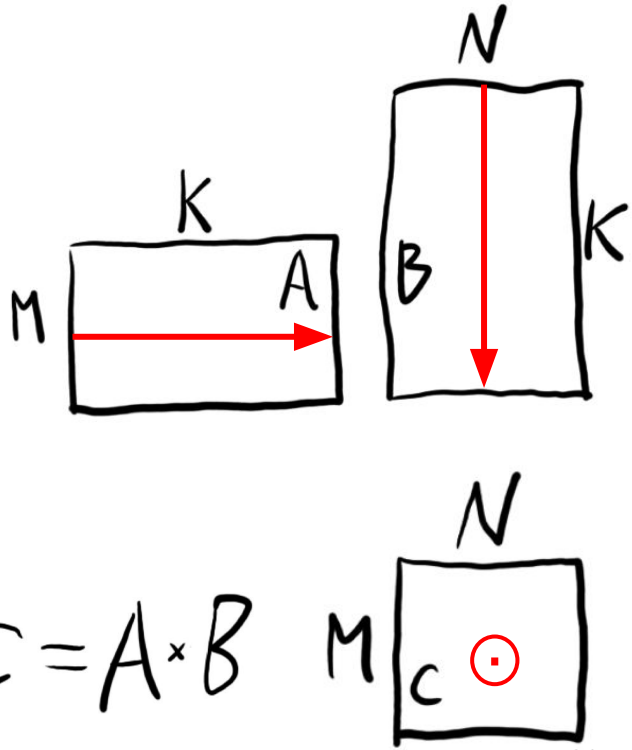
O(N□M□K)

Сколько у нас чтений/записей данных?

O(N□M□K)

Какая пропорция?

1:1 Это хороший результат?



Умножение матриц

Сколько у нас вычислений?

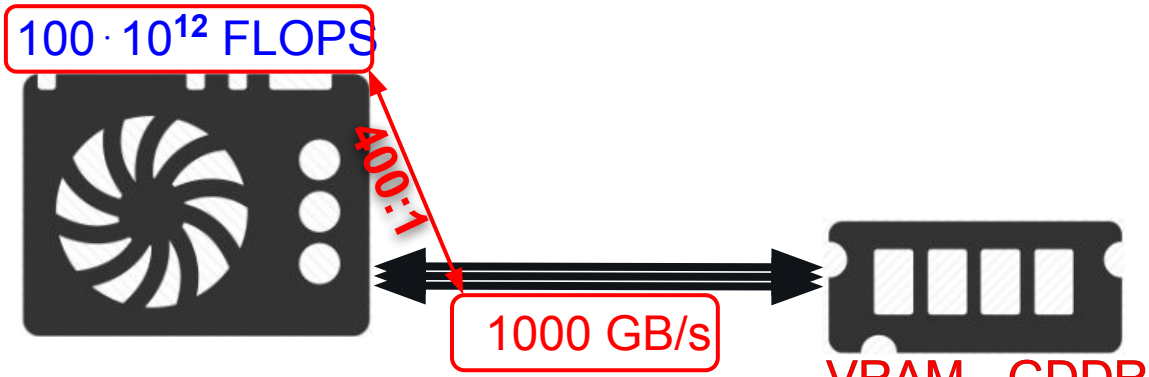
O(N□M□K)

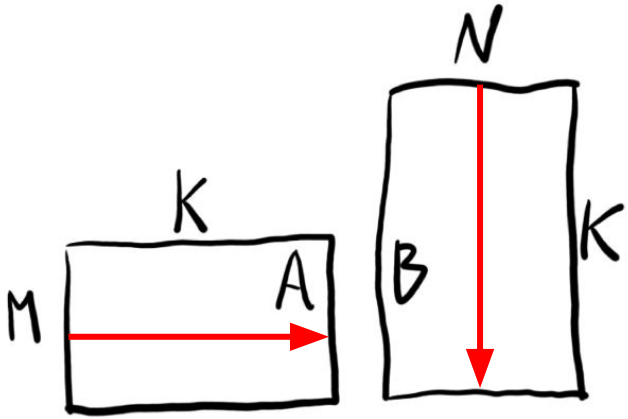
Сколько у нас чтений/записей данных?

O(N□M□K)

Какая пропорция?

1:1 Это хороший результат?





Умножение матриц

Сколько у нас вычислений?

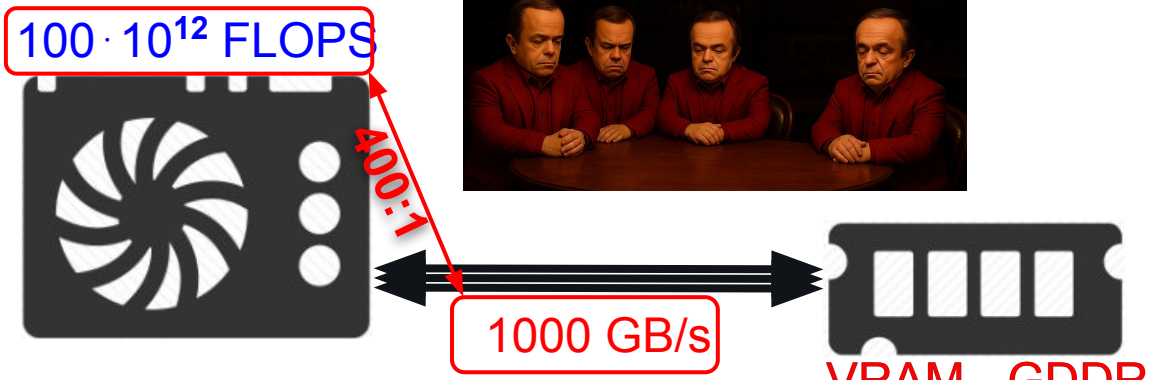
O(N□M□K)

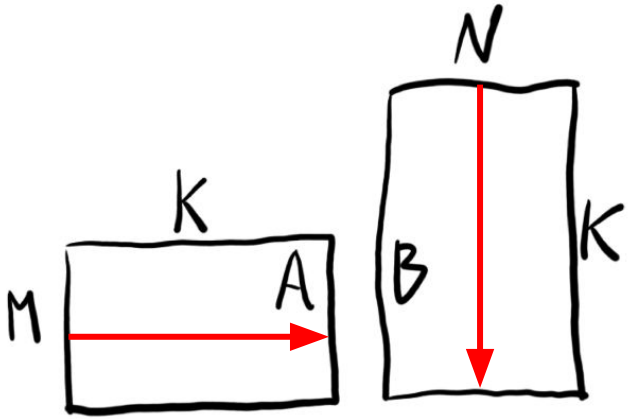
Сколько у нас чтений/записей данных?

O(N□M□K)

Какая пропорция?

1:1 **Memory-bound!**





Умножение матриц

Сколько у нас вычислений?

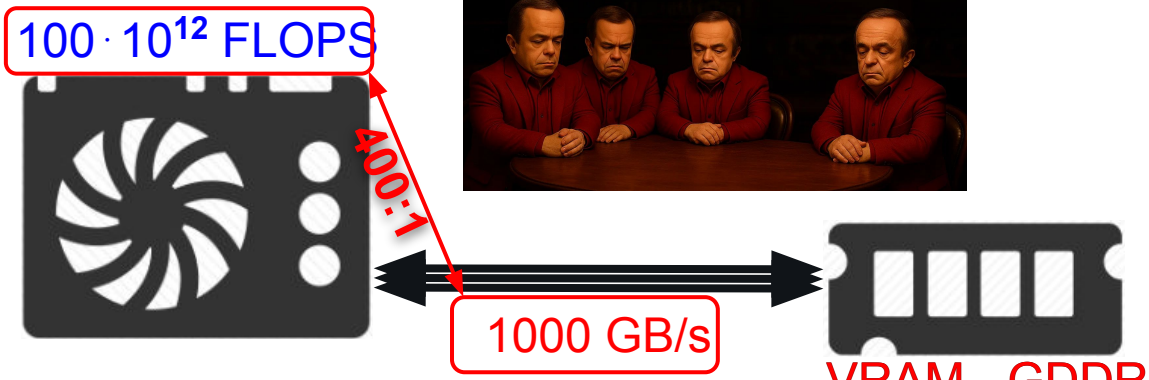
$O(N \cdot M \cdot K)$

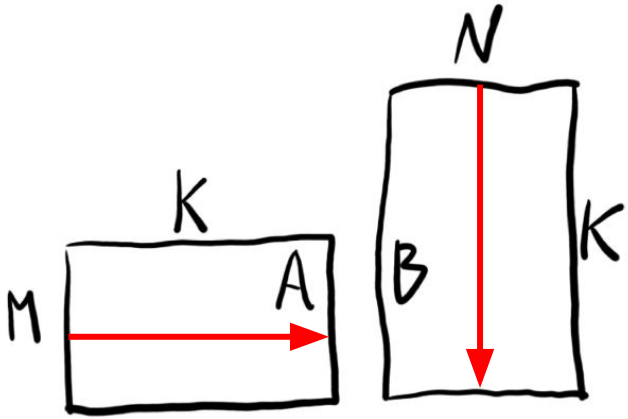
Сколько у нас чтений/записей данных?

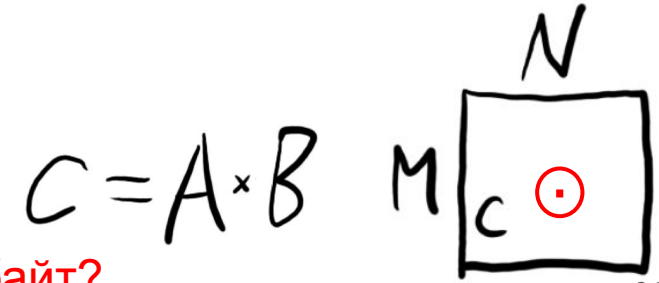
$O(N \cdot M \cdot K)$

Какая пропорция?

1:1 **Memory-bound!**







Как увеличить объем вычислений на считанный байт?

Умножение матриц

Сколько у нас вычислений?

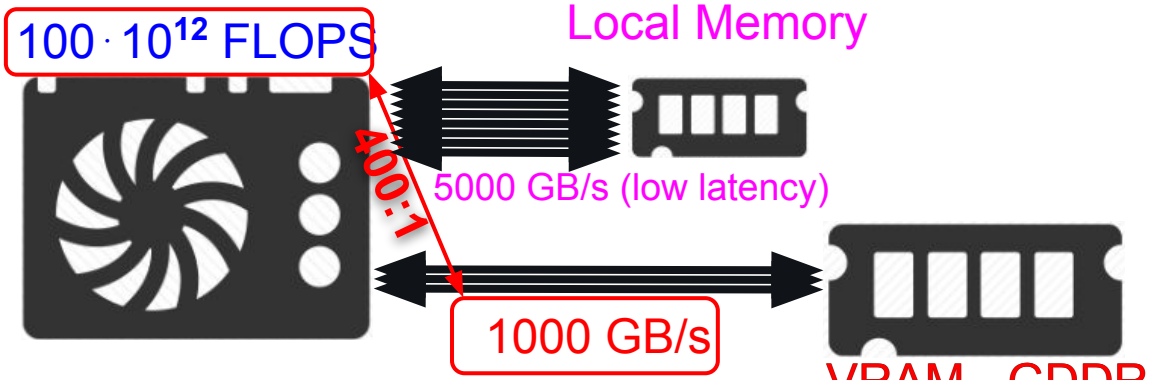
O(N□M□K)

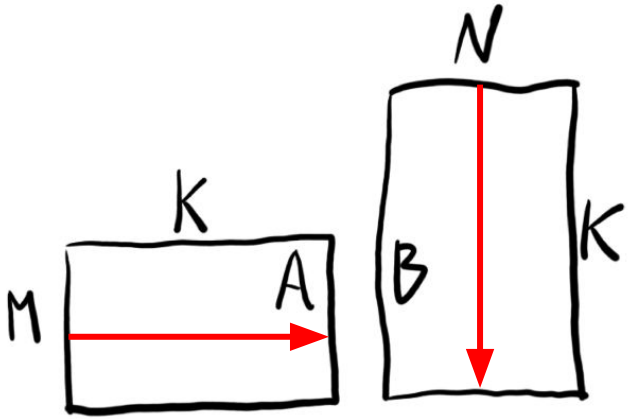
Сколько у нас чтений/записей данных?

O(N□M□K)

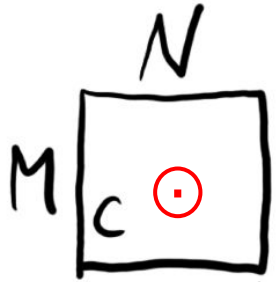
Какая пропорция?

1:1 **Memory-bound!**



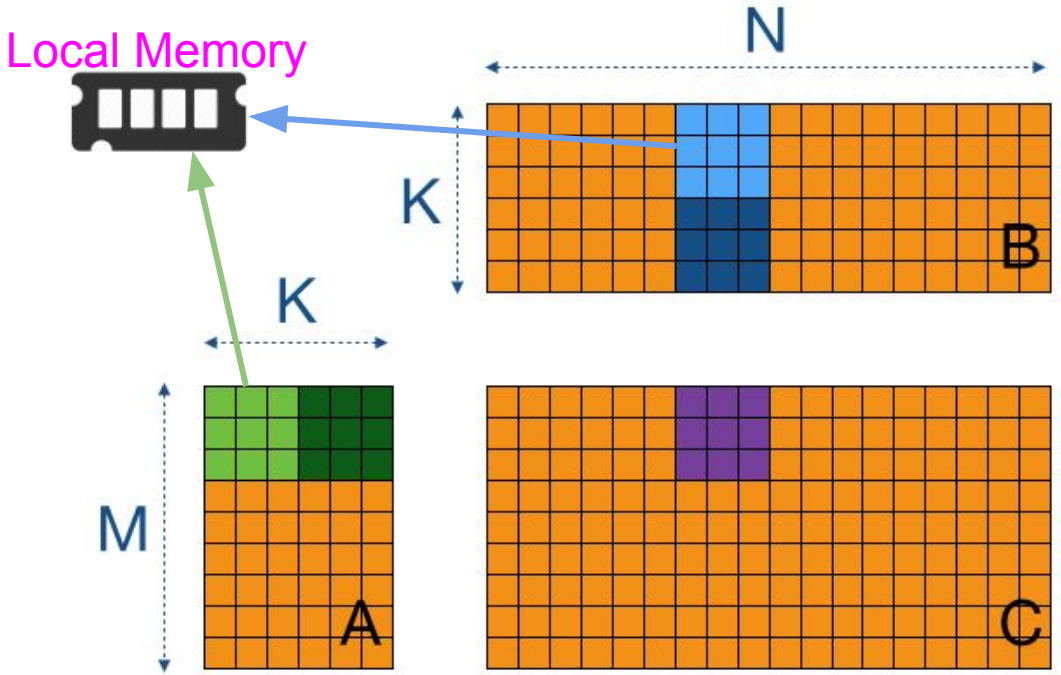


$C = A \times B$

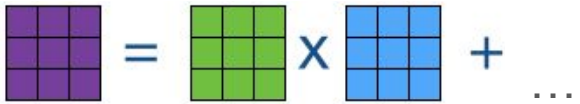


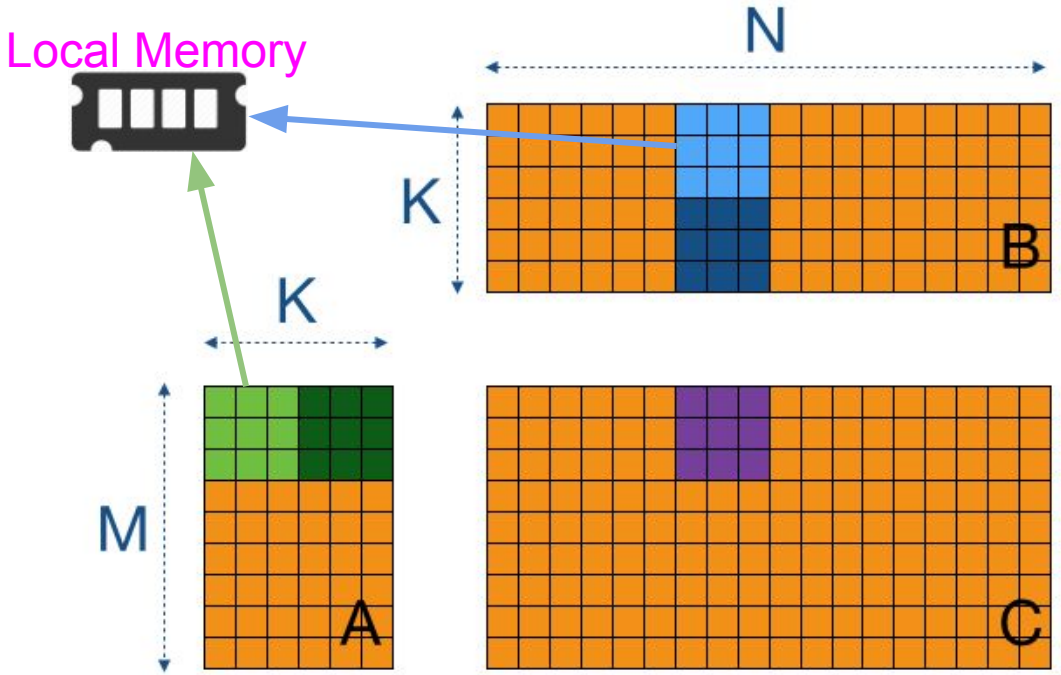
Как увеличить объем вычислений на считанный байт?

Умножение матриц

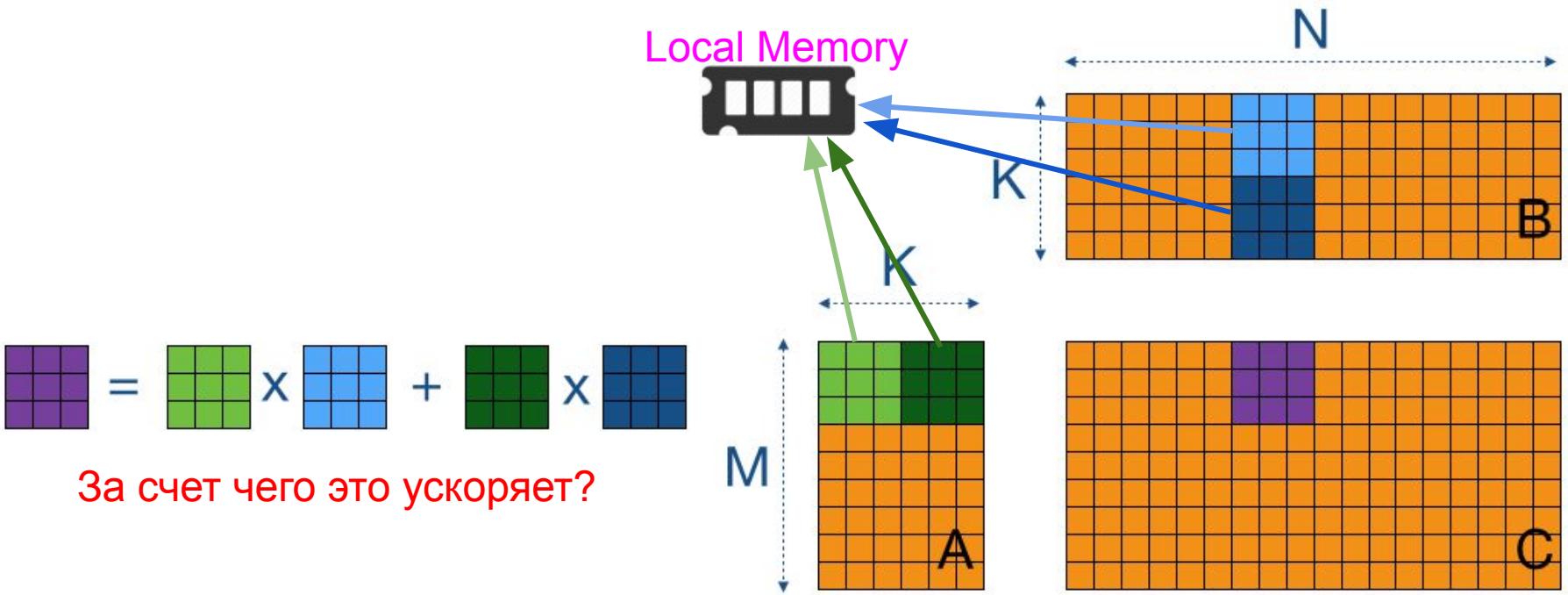


Умножение матриц





Умножение матриц



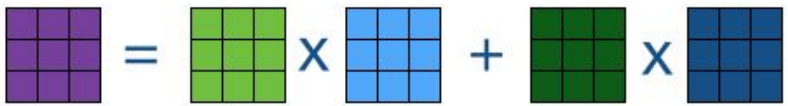
За счет чего это ускоряет?

Умножение матриц

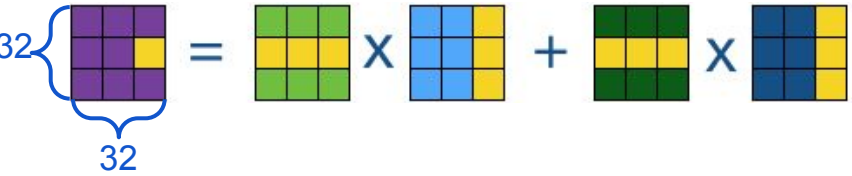
Сколько у нас вычислений?

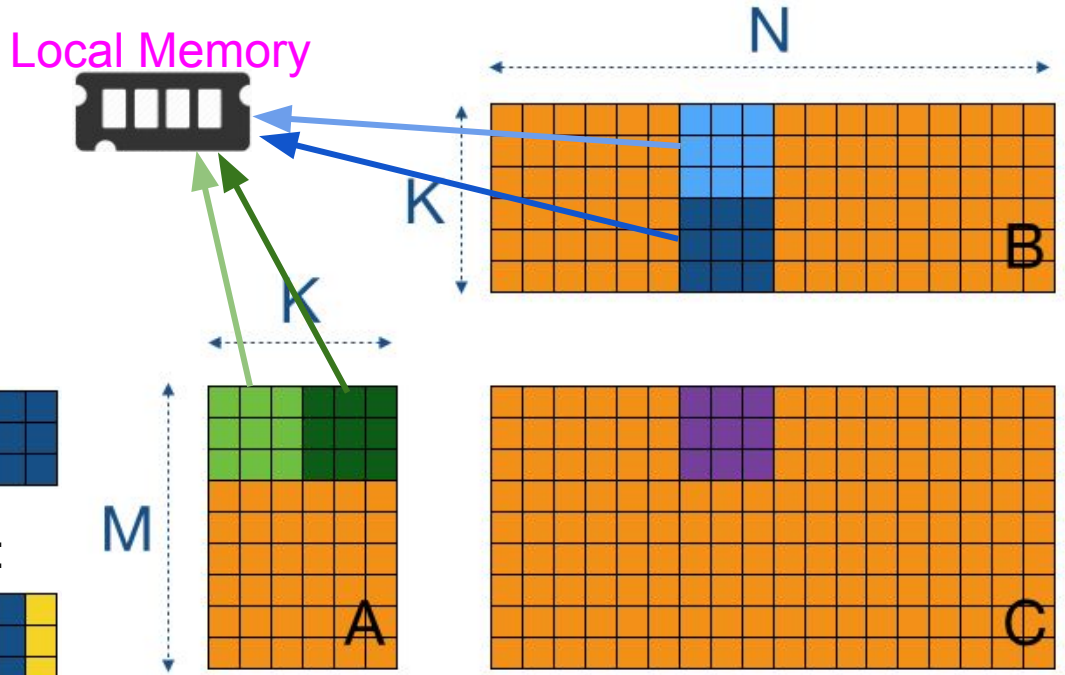
Сколько у нас чтений/записей?

Какая пропорция?



Переиспользование данных:





Умножение матриц

Сколько у нас вычислений?

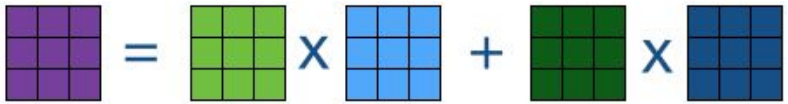
O(N□M□K)

Сколько у нас чтений/записей?

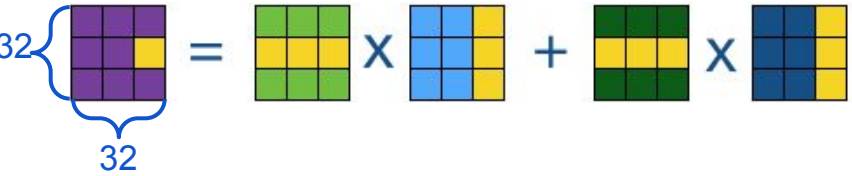
O(N□M□K / **32**)

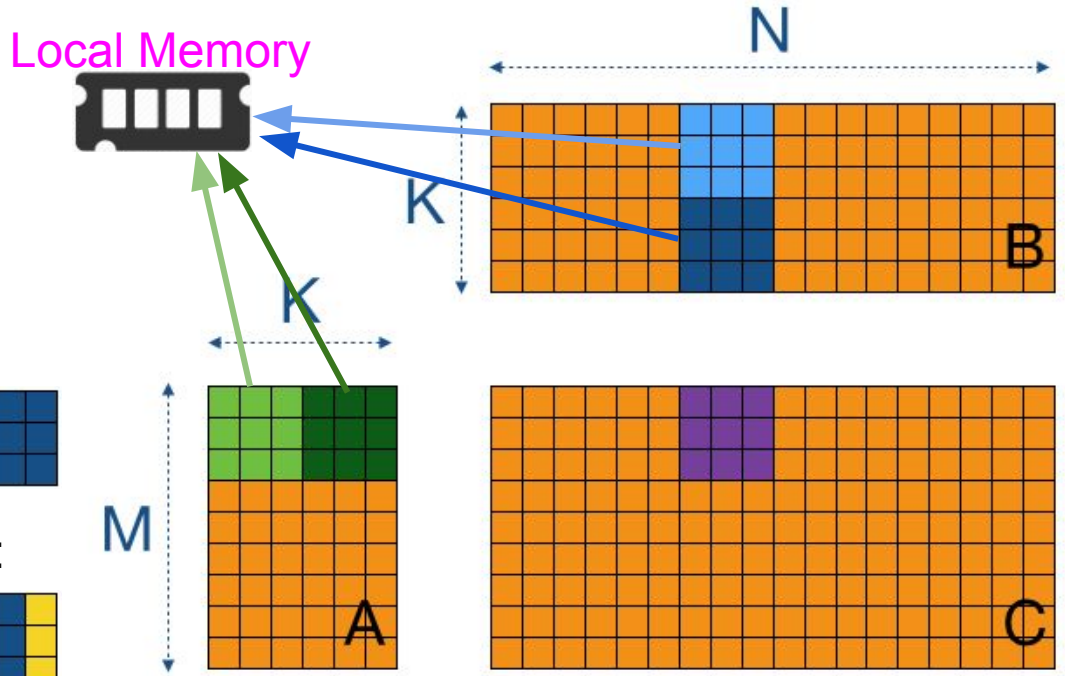
Какая пропорция?

32:1



Переиспользование данных:





Умножение матриц

Сколько у нас вычислений?

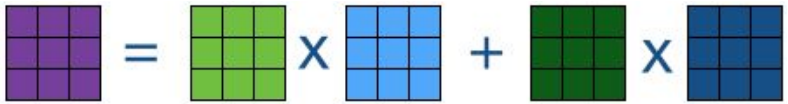
O(N□M□K)

Сколько у нас чтений/записей?

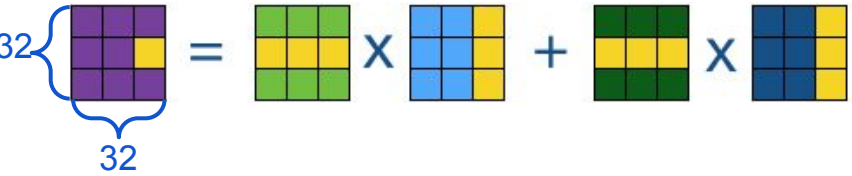
O(N□M□K / 32)

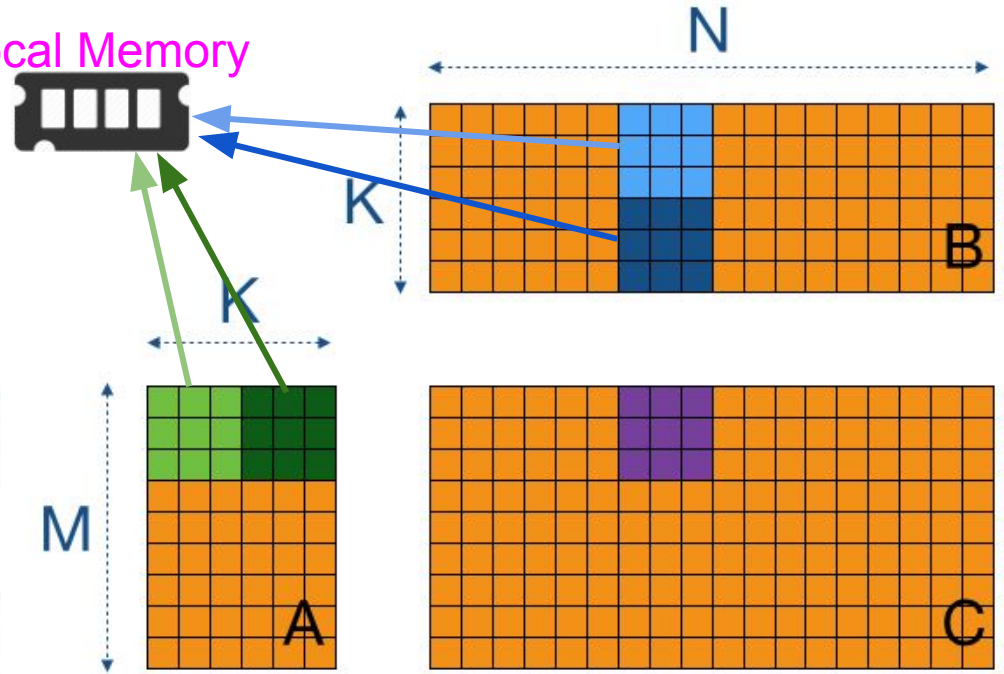
Какая пропорция?

32:1



Переиспользование данных:





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

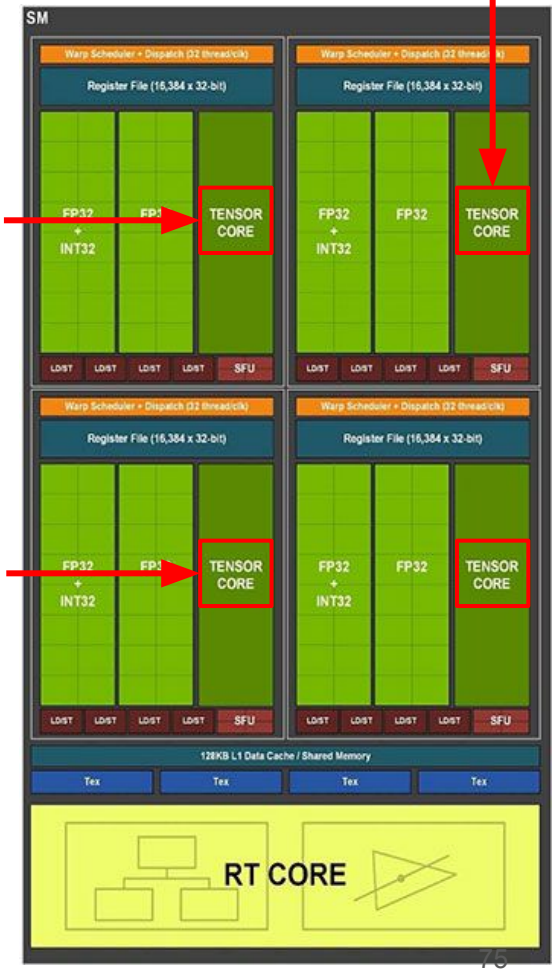

Перерыв!
Streaming
Pubiprocestor
VIDIA

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

4x4

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

FP16 or FP32 — FP16 — FP16 — FP16 or FP32

Умножение матриц

Tensor Cores

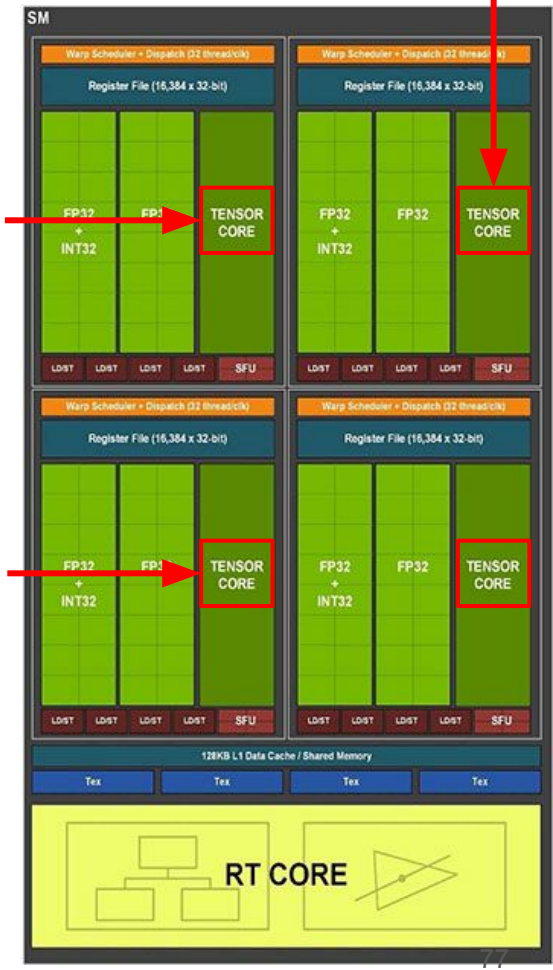
4x4

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

Tensor Cores (CUDA kernel)

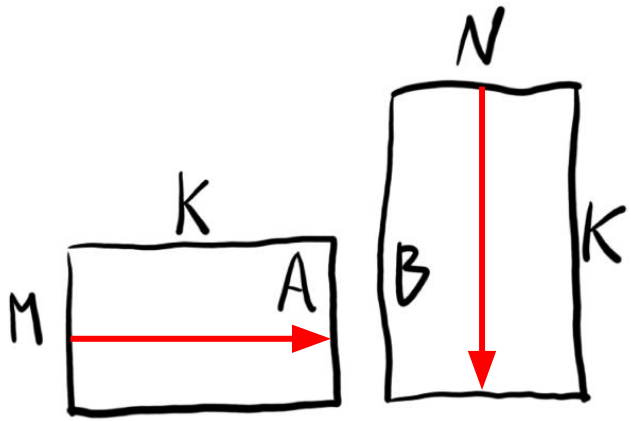
16x16

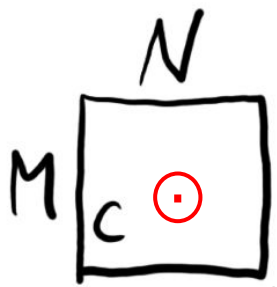
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,\dots} & A_{0,15} \\ A_{1,0} & A_{1,1} & A_{1,\dots} & A_{1,15} \\ A_{\dots,0} & A_{\dots,1} & A_{\dots,\dots} & A_{\dots,15} \\ A_{15,0} & A_{15,1} & A_{15,\dots} & A_{15,15} \end{pmatrix} \begin{matrix} \\ \\ \\ \\ \text{FP16 or FP32} \end{matrix} \begin{pmatrix}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



```
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) {
```

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



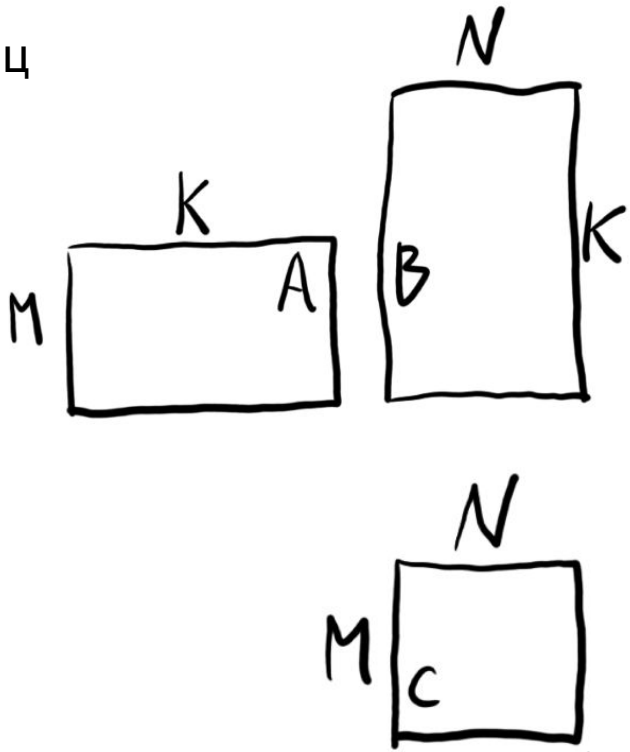


https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
```

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

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

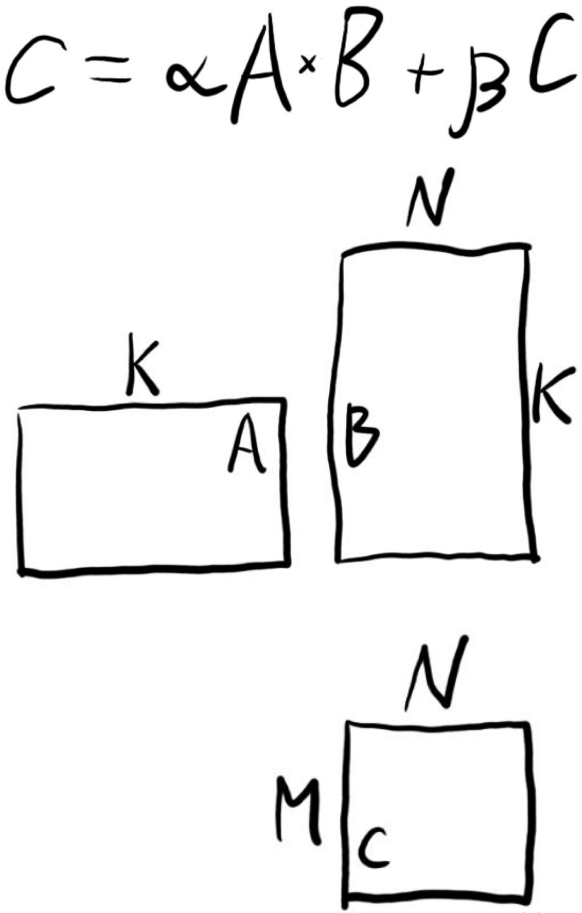


https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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



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

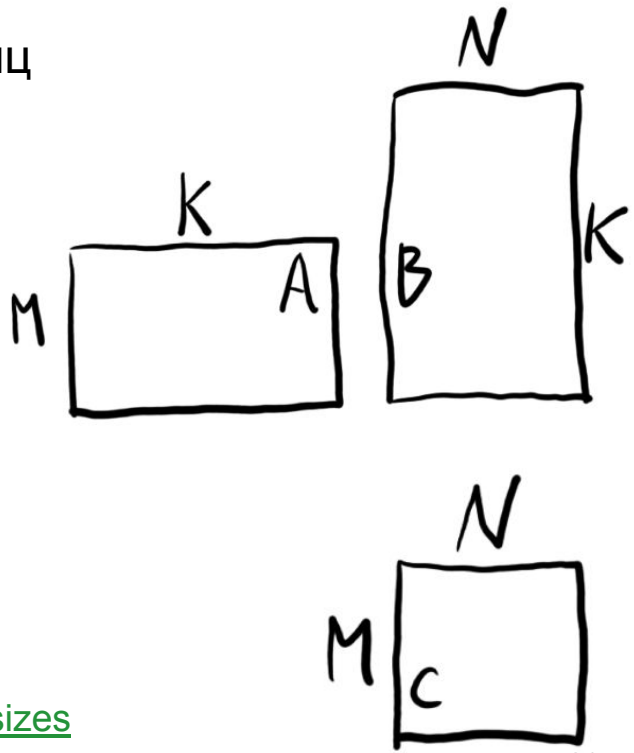
```
69    // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75    __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85       // Declare the fragments
86       wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87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

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

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
...	...	...	...



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

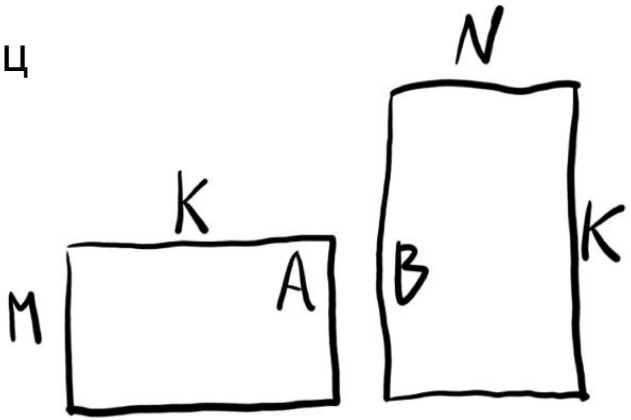
```
69    // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75    __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85       // Declare the fragments
86       wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87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

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

Общие на весь **warp** 16x16 фрагменты матриц

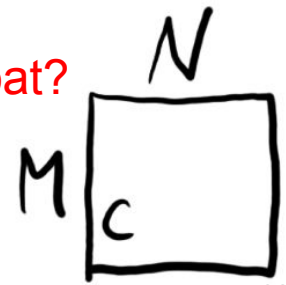
WMMA = Warp Matrix Multiply-Accumulate

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 A и B - half?

Но C и аккумулятор - float?



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

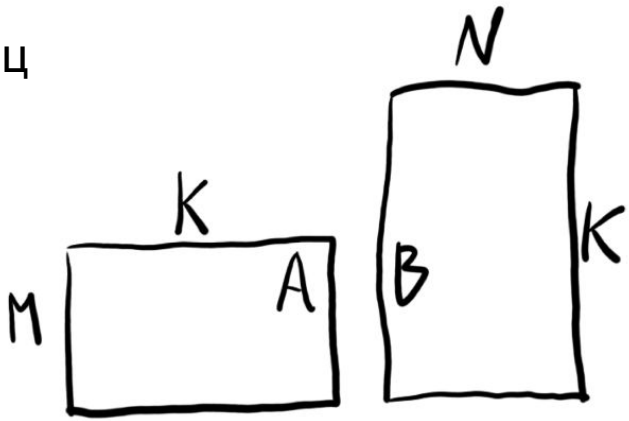
https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

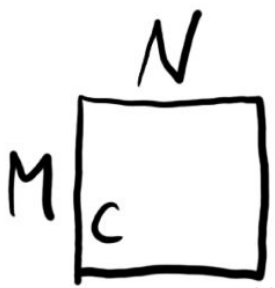
```
69    // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75    __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85       // Declare the fragments
86       wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87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

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

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

Почему A и B - half?
Но C и аккумулятор - float?
Подсказка:
- A и B перемножаются
- аккумулятор суммируется



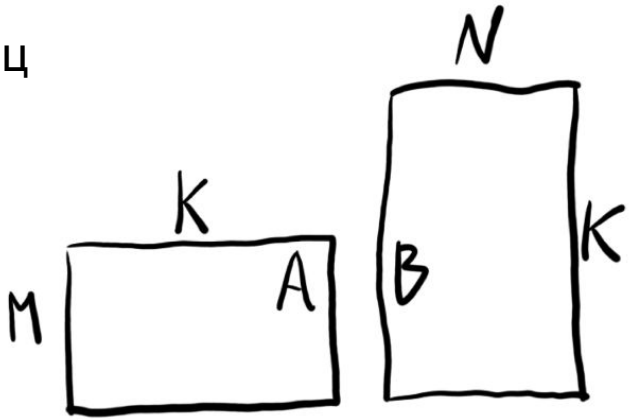


https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
```

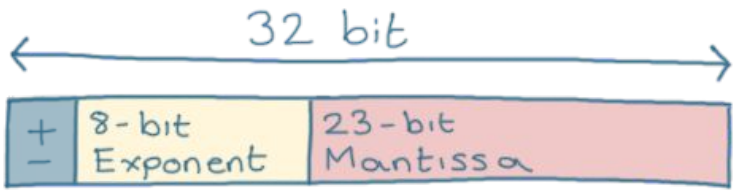
$$C = \alpha A \times B + \beta C$$

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

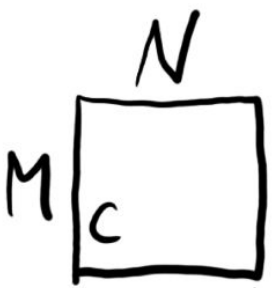


Почему A и B - half?
Но C и аккумулятор - float?

Что будет если **сложить** огромное и малое число?



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

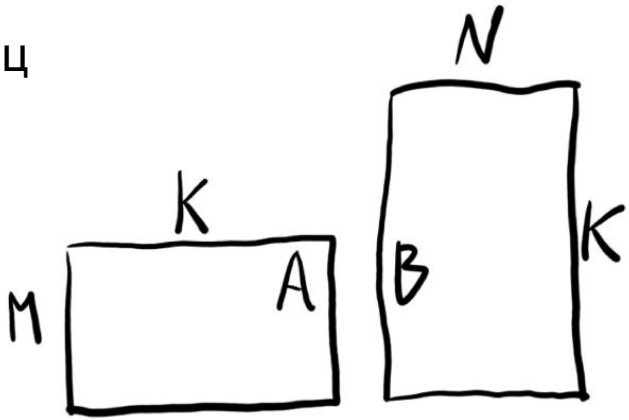


https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
```

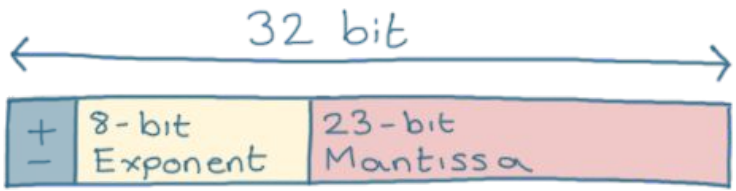
$$C = \alpha A \times B + \beta C$$

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



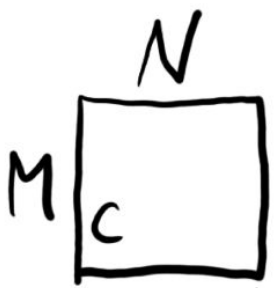
Почему A и B - half?
Но C и аккумулятор - float?

Что будет если **сложить** огромное и малое число?



Что будет если **умножить** огромное и малое число?

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

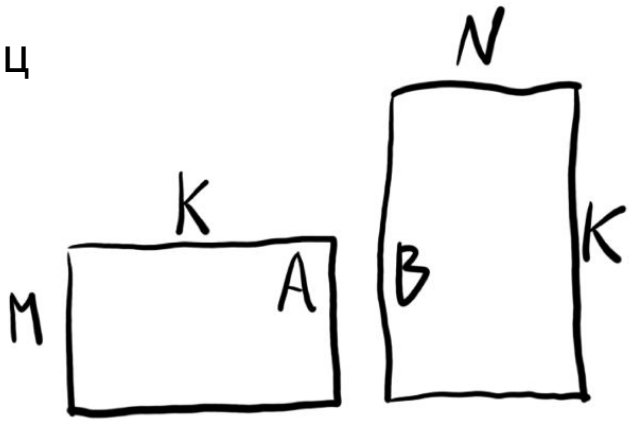


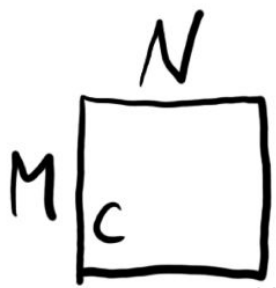
https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
```

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

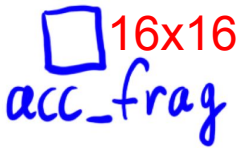
Общие на весь **warp** 16x16 фрагменты матриц
WMMA = **Warp** Matrix Multiply-Accumulate

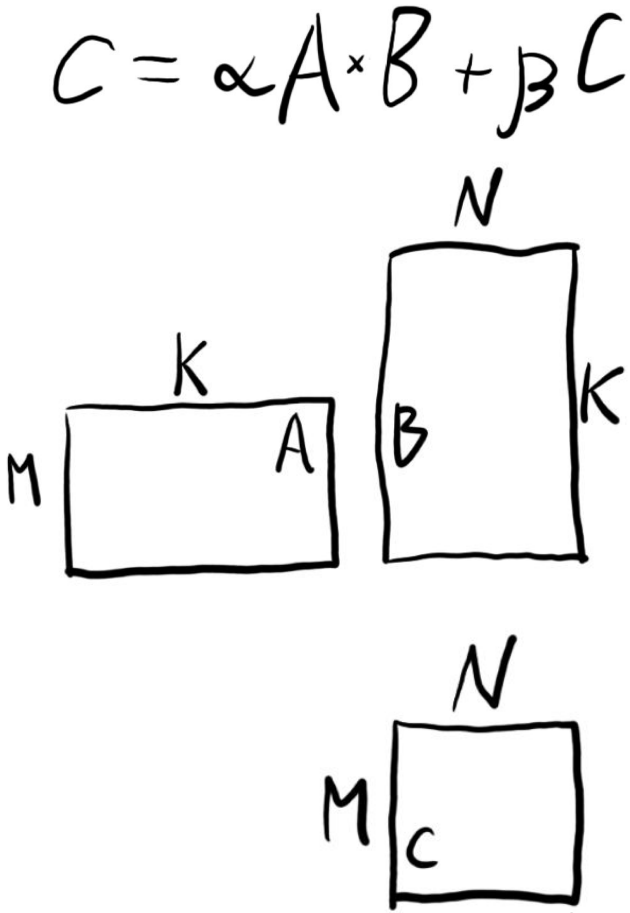




https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
91       wmma::fragment_fragment(acc_frag, 0.0f);
```



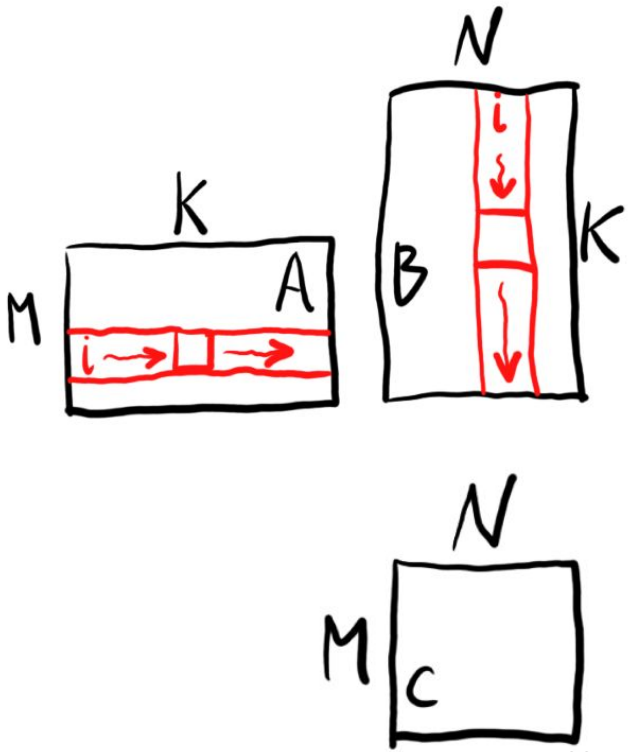


https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
91     wmma::fill_fragment(acc_frag, 0.0f);
94     for (int i = 0; i < K; i += WMMA_K) {
```

16x16 acc_frag

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

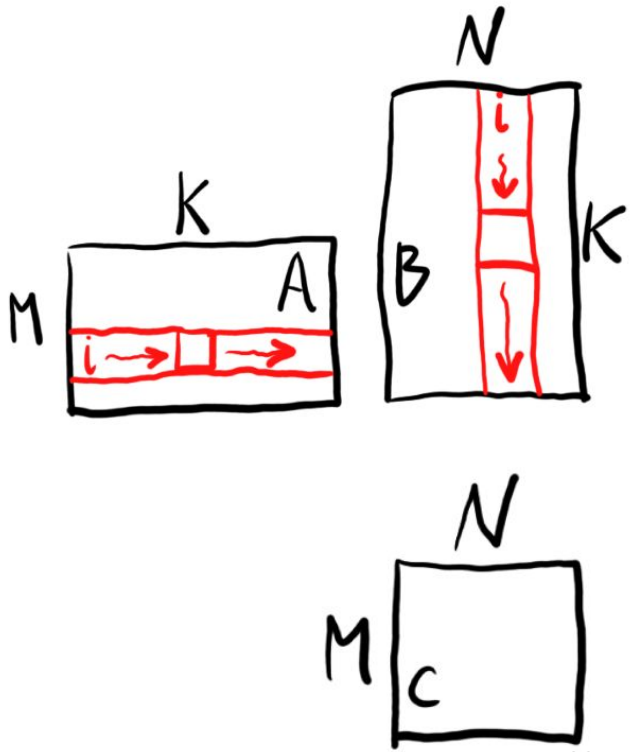


https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
91     wmma::fill_fragment(acc_frag, 0.0f);
94     for (int i = 0; i < K; i += WMMA_K) {
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);
```

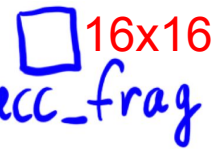
16x16 acc_frag

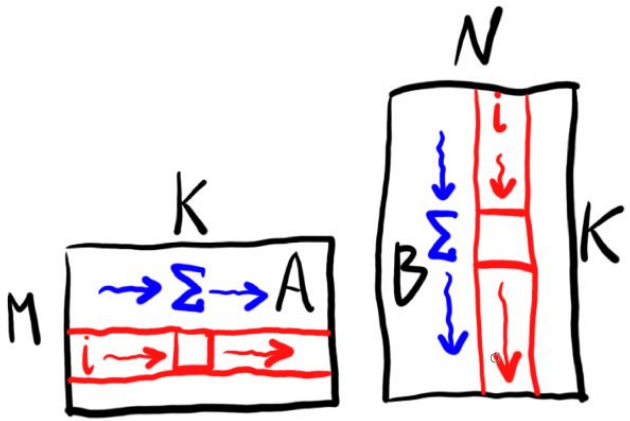
$C = \alpha A \times B + \beta C$

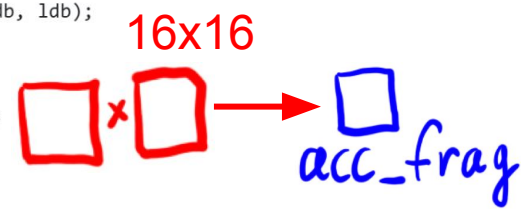


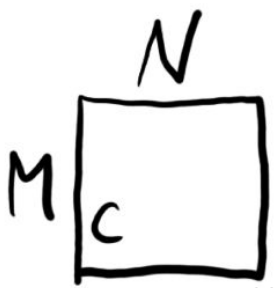
https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
91     wmma::fill_fragment(acc_frag, 0.0f);
94     for (int i = 0; i < K; i += WMMA_K) {
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    }
```



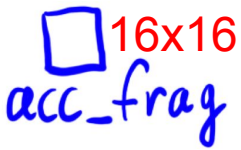


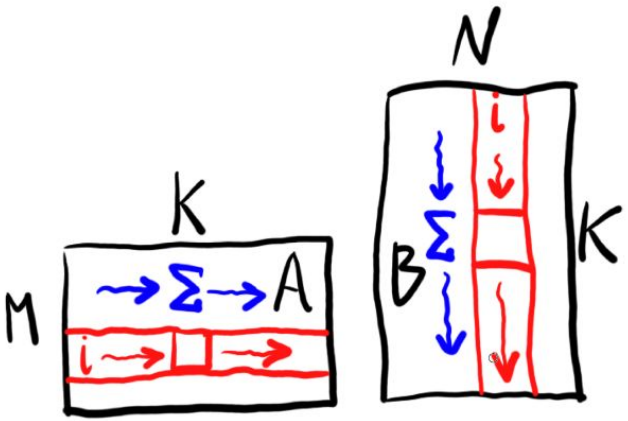


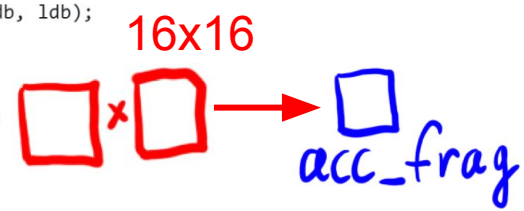


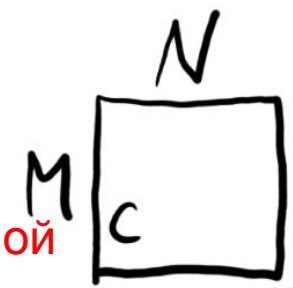
https://github.com/NVIDIA-developer-blog/code-samples/blob/master/posts/tensor-cores/simpleTensorCoreGEMM.cu

```
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;
91     wmma::fill_fragment(acc_frag, 0.0f);
94     for (int i = 0; i < K; i += WMMA_K) {
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 }
```





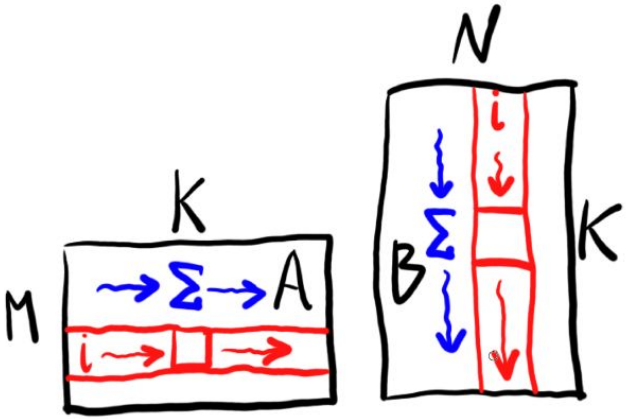


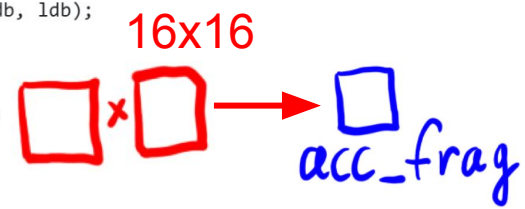


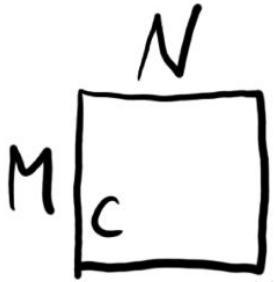
Чем этот код отличается от классического умножения в локальной памяти?

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;
91     wmma::fill_fragment(acc_frag, 0.0f);
94     for (int i = 0; i < K; i += WMMA_K) {
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 }
```





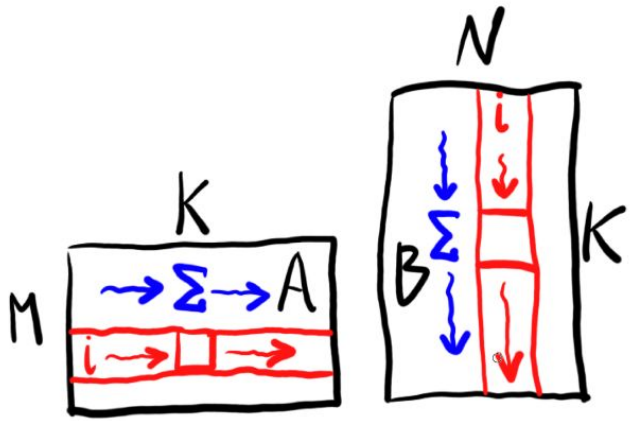


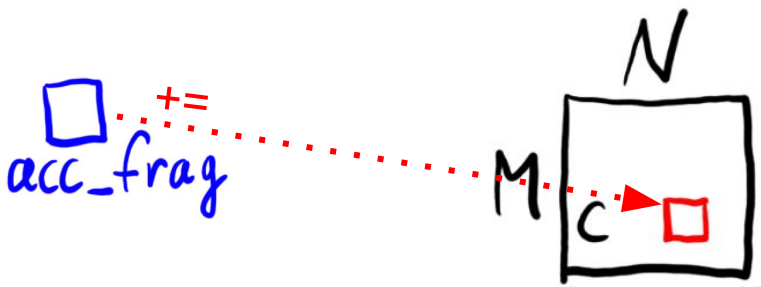
Что нам осталось сделать?

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;
         .........
```

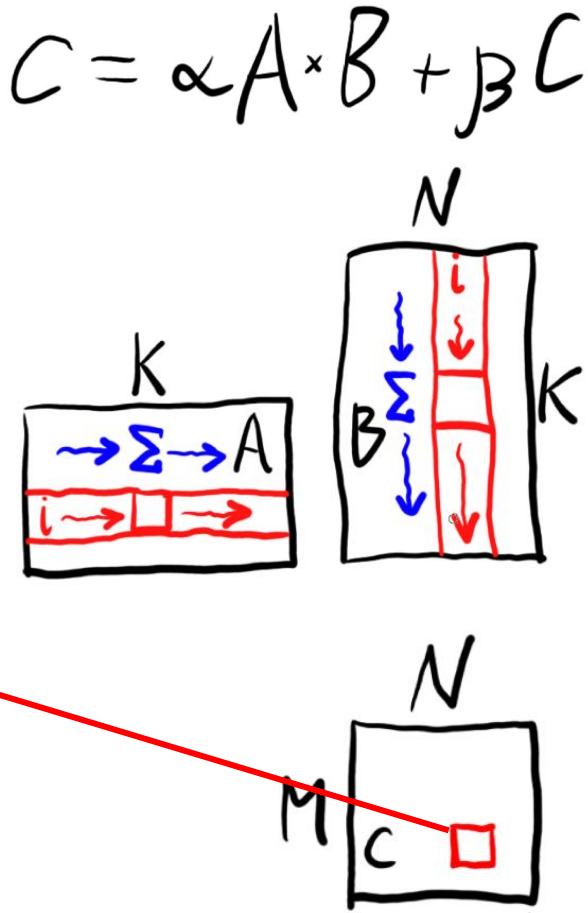
$$C = \alpha A \times B + \beta C$$





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

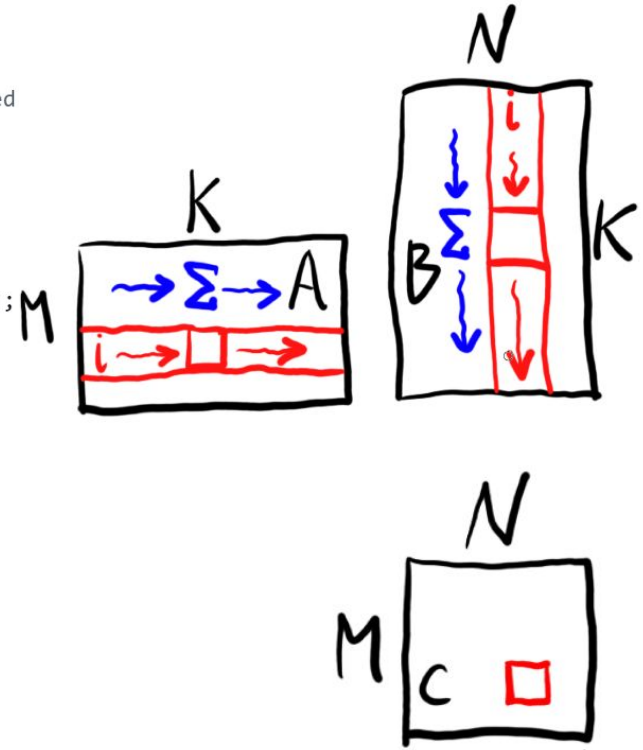
```
69    // Performs an MxNxK GEMM (C=alpha*A*B + beta*C)
75    __global__ void wmma_example(half *a, half *b, float *c, int M, int N, int K, float alpha, float beta) {
85       // Declare the fragments
86       wmma::fragment<wmma::matrix_a, WMMA_M, WMMA_N, WMMA_K, half, wmma::col_major> a_frag;
87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



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;
         .........
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         }
```

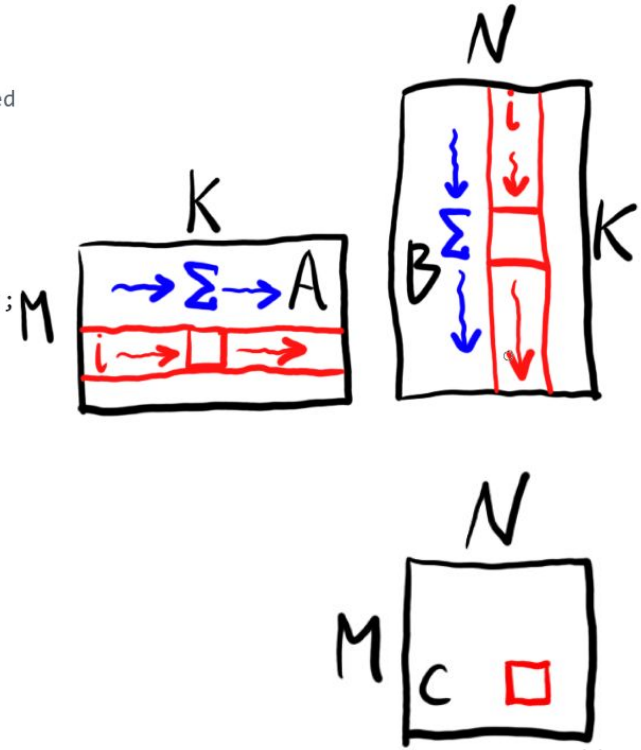
$$C = \alpha A \times B + \beta C$$



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;
         .........
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         }
```

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

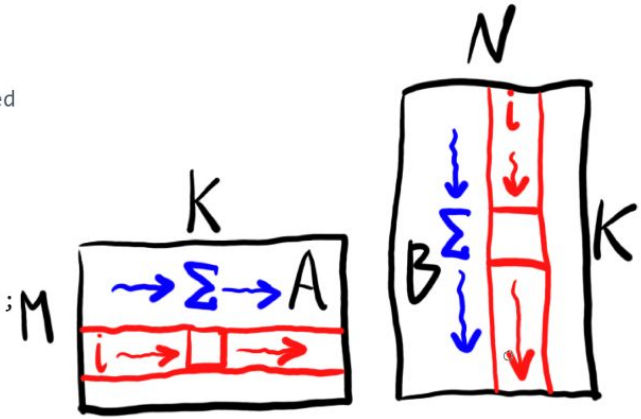


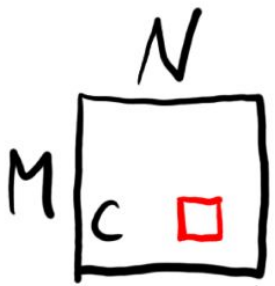
А не будет тормозить?
Почему не спец. функция в железе?

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;
         .........
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 C в отдельном кернеле?

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

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

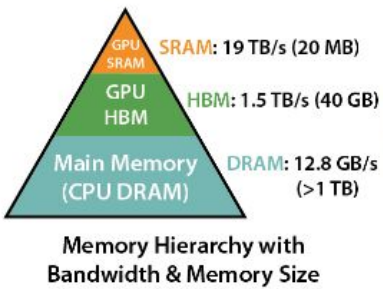
FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness

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

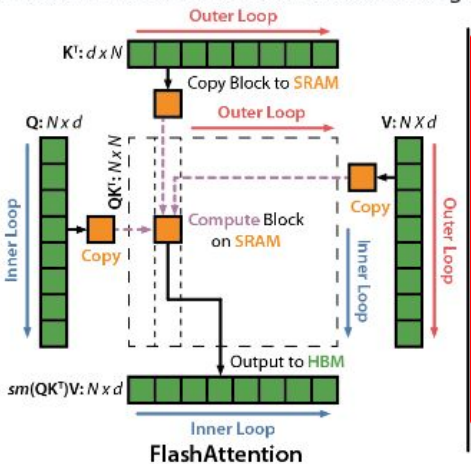
Paper: https://arxiv.org/abs/2205.14135 https://github.com/Dao-AILab/flash-attention

IEEE Spectrum article about our submission to the MLPerf 2.0 benchmark using FlashAttention.

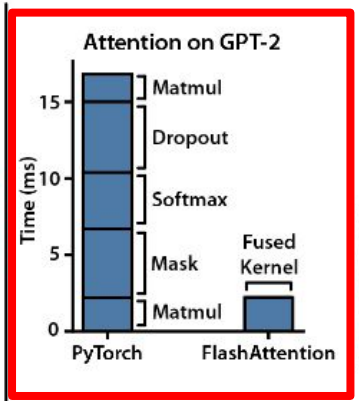
$$C = \alpha A \times B + \beta C$$

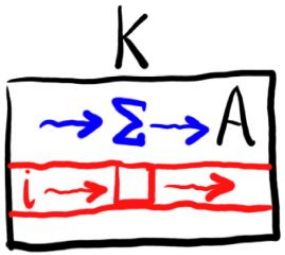


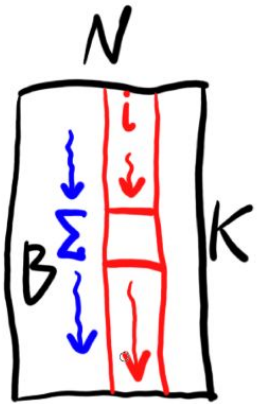
Memory Hierarchy with Bandwidth & Memory Size



FlashAttention



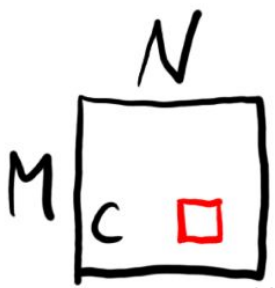




А не будет тормозить?

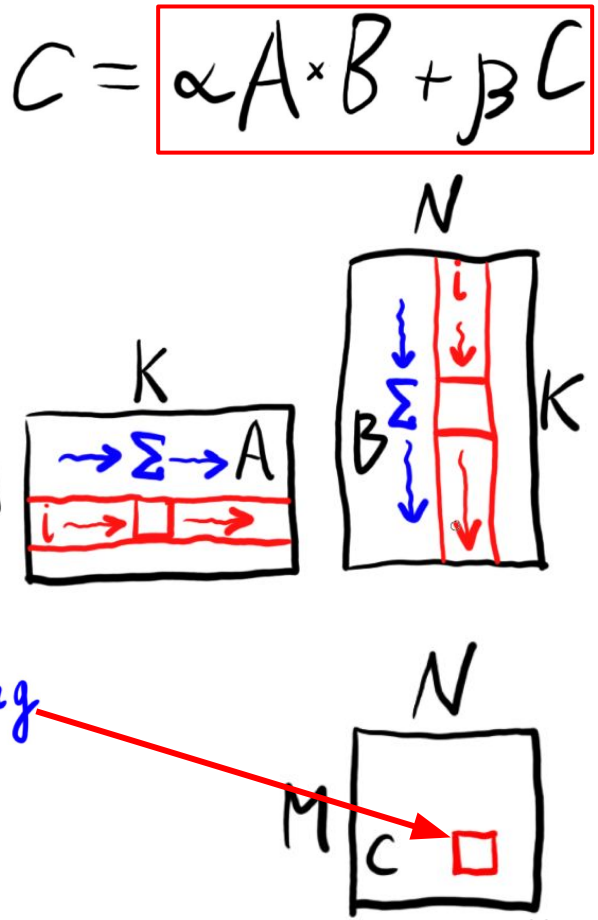
Почему не спец. функция в железе?

А почему бы не добавить к C в отдельном кернеле?



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;
         .........
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   }
```



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

Умножение матриц - Tensor Cores, WMMA

Сколько у нас вычислений?

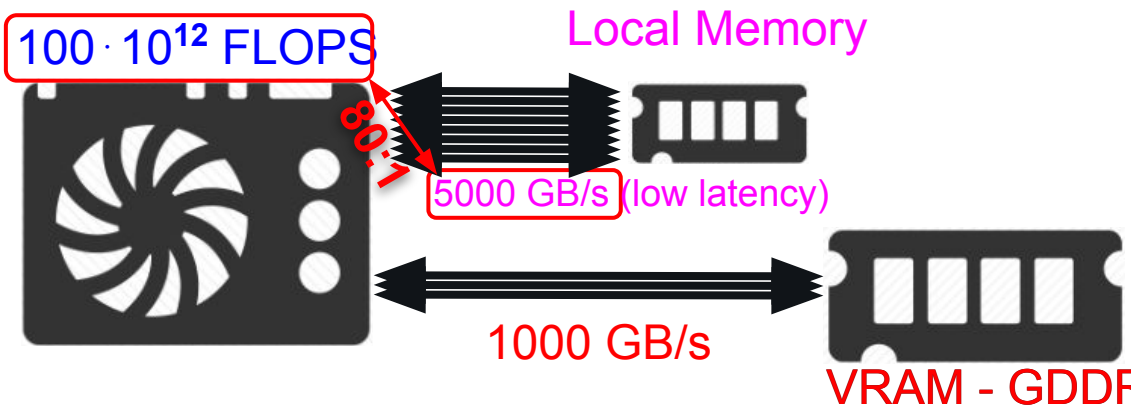
O(N□M□K)

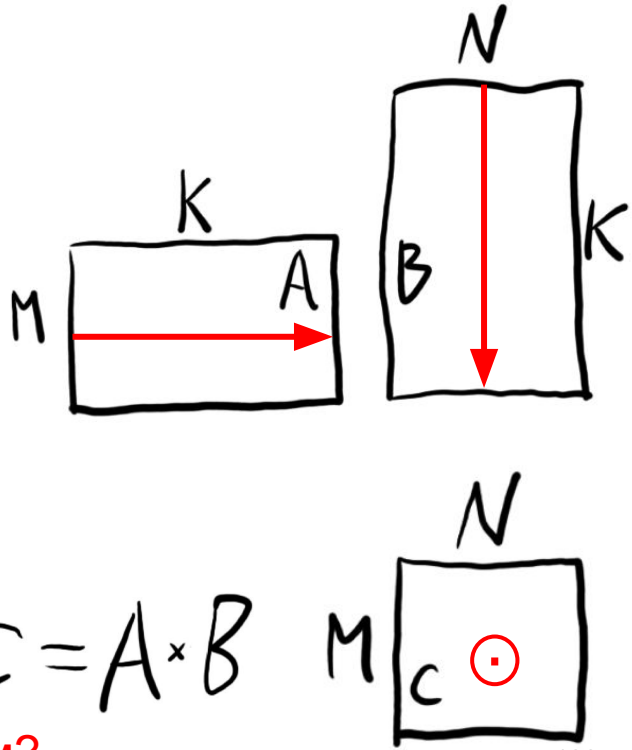
Сколько у нас чтений/записей данных?

O(N□M□K)

Какая пропорция?

1:1 **Memory-bound!**





Какая у нас теперь пропорция вычислений к чтениям?

Умножение матриц - Tensor Cores, WMMA

Сколько у нас вычислений?

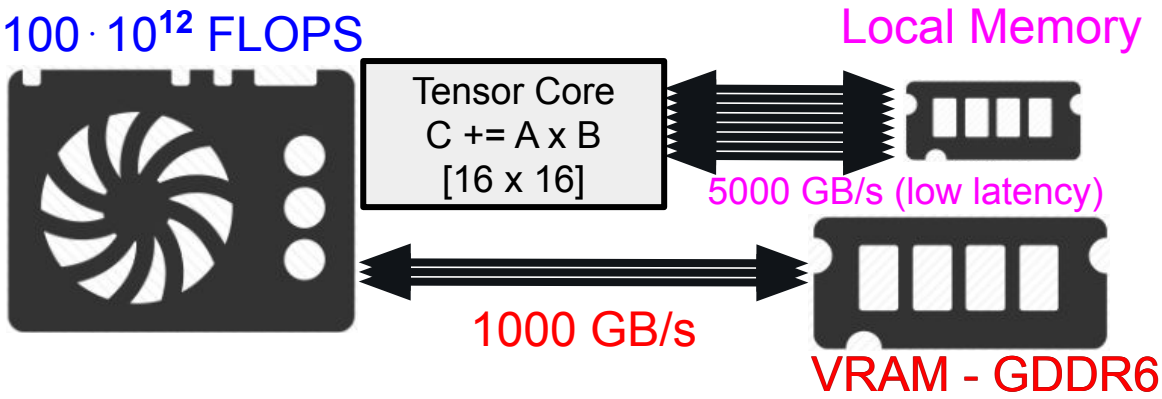
O(N□M□K)

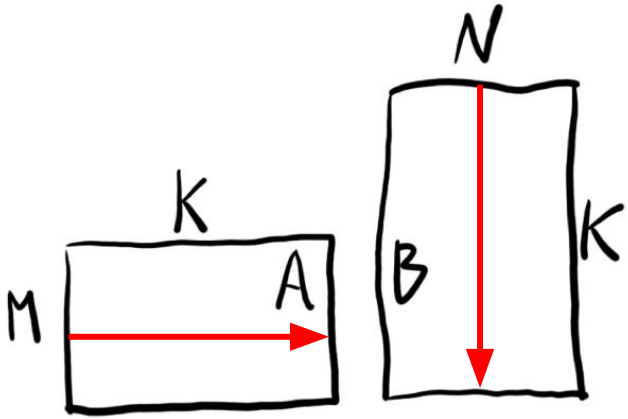
Сколько у нас чтений/записей данных?

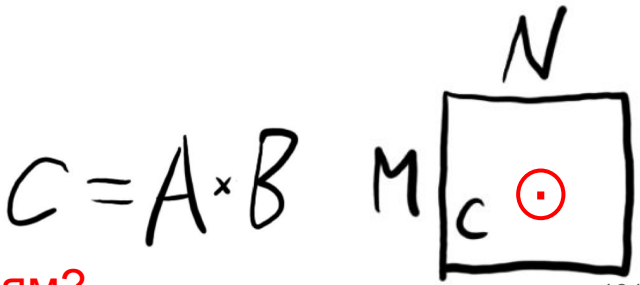
O(N□M□K)

Какая пропорция?

1:1 **Memory-bound!**







Какая у нас теперь пропорция вычислений к чтениям?

Умножение матриц

Сколько у нас вычислений?

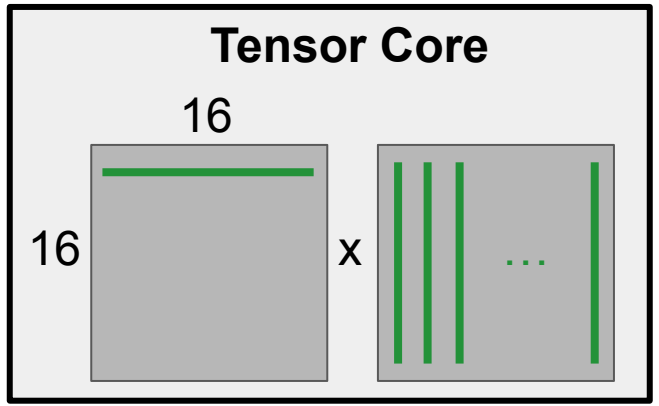
O(N□M□K)

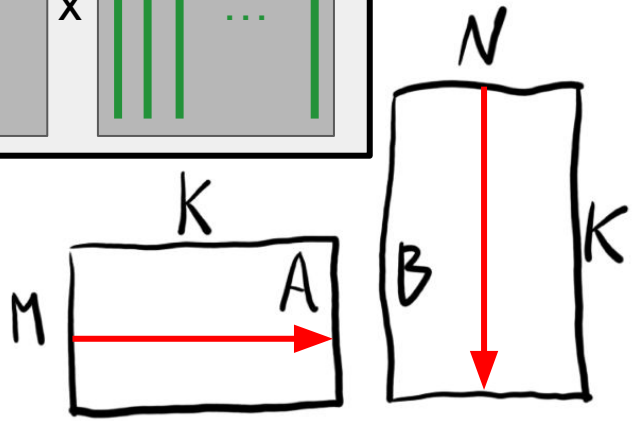
Сколько у нас чтений/записей данных?

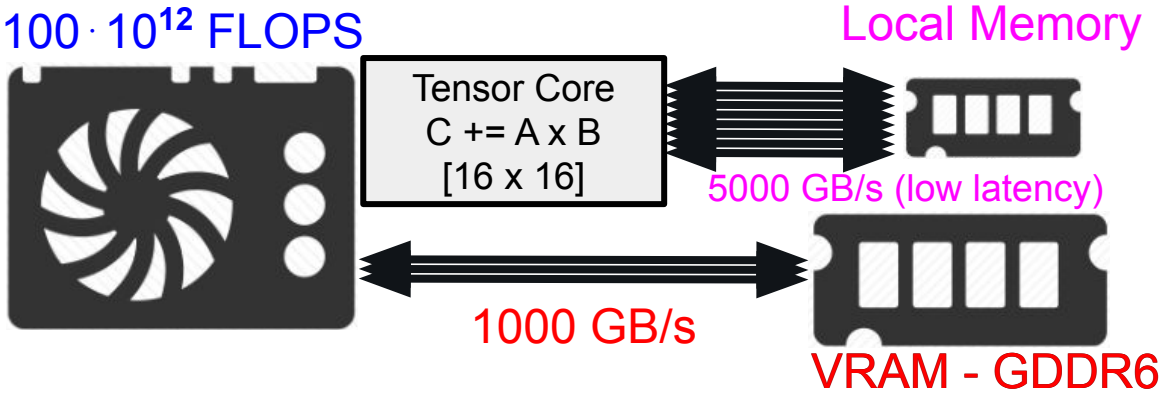
O(N□M□K)

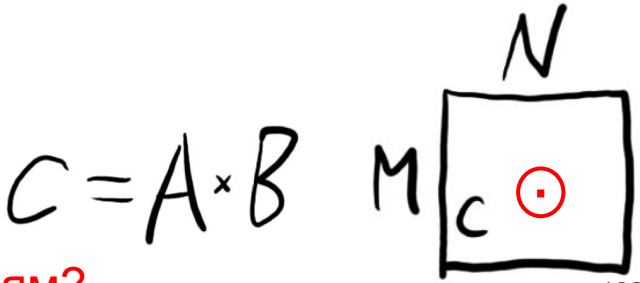
Какая пропорция?

1:1 **Memory-bound!**









Какая у нас теперь пропорция вычислений к чтениям?

secret techniques

✓ **cutlass name trick**

Mogball committed on Jul 9

23 ■■■■ python/tutorials/gluon/01-attention-forward.py

```
@@ -834,13 +834,20 @@ def _attn_fwd_correction(config, chnls, descs, M, STAGE: gl.constexpr):
834 834             config, prog, s1_tmem, M, corr1_consumer, epi_producer, o_consumer)
835 835
836 836
837     - @gluon.jit(do_not_specialize=["Z"])
838     - def gluon_attn(sm_scale, M, Z, H, N_CTX,  #
839     -                desc_q, desc_k, desc_v, desc_o,  #
840     -                BLOCK_M: gl.constexpr, BLOCK_N: gl.constexpr, HEAD_DIM: gl.constexpr,  #
841     -                GROUP_SIZE_N: gl.constexpr, NUM_SMS: gl.constexpr,  #
842     -                STAGE: gl.constexpr, dtype: gl.constexpr,  #
843     -                num_warps: gl.constexpr):
    837 + def attention_repr(specialization):
    838 +     name = "gluon_attention"
    839 +     # Up to 150 TFLOPS faster for fp8!
    840 +     if specialization.constants["dtype"] == gl.float8e5:
    841 +         name = "cutlass_" + name
```

peterbell10 on Jul 9 — Contributor

very cool... did you check if other names change the scheduling (e.g. because of non-determinism or code alignment) or if it's literally just special cased for cutlass.

Mogball on Jul 9 — Collaborator, Author

> it's literally just special cased for cutlass.

Yup

😀 46 😕 8 👀 5

https://github.com/triton-lang/triton/pull/7298/commits/a5e23d8e7e64b8a11af3edc1705407d91084b01d

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/

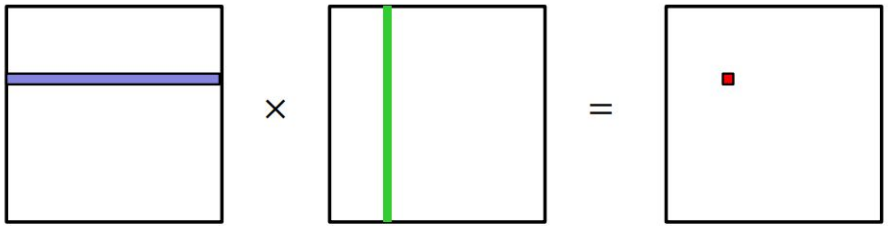
Неравная битва за гигафлопсы при умножении матриц
(хорошо описанная аналитика, профилирование, оптимизация):

- 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

Scalar Loop

- Each invocation computes one element of the result matrix
- Lots of redundant loads
- Example of what NOT to do

```
uint i = gl_GlobalInvocationID.y;
uint j = gl_GlobalInvocationID.x;
float16_t C = inputC.x[sC * i + j];

for (uint k = 0; k < K; ++k) {
  float16_t A = inputA.x[sA * i + k];
  float16_t B = inputB.x[sB * k + j];
  C += A*B;
}

outputD.x[sD * i + j] = C;
```



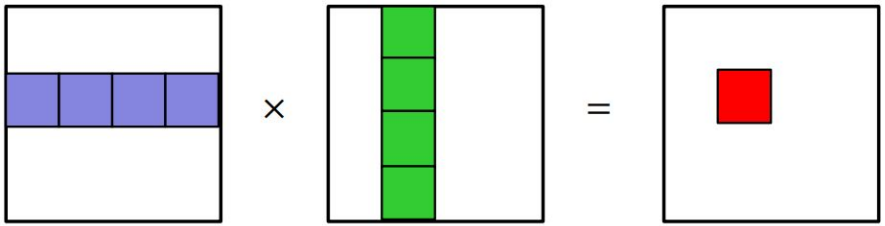
GPU	TFLOPS
RTX 2070	0.232
RTX TITAN	0.528

https://www.khronos.org/assets/uploads/developers/presentations/Cooperative_Matrix_May22.pdf

Simple Cooperative Multiply

- A bit more coordinate calculation, but still a simple sum($A_{ik}B_{kj}$)
- Still very memory-limited, but improved

```
lM = 16; lN = 8; lK = 8;
fcoopmatNV<16, gl_ScopeSubgroup, lM, lK> matA;
fcoopmatNV<16, gl_ScopeSubgroup, lK, lN> matB;
fcoopmatNV<16, gl_ScopeSubgroup, lM, lN> matC;

uvec2 matrixID = uvec2(gl_WorkGroupID);
uint cRow = lM * matrixID.y;
uint cCol = lN * matrixID.x;

coopMatLoadNV matC, inputC.x, sC * cRow + cCol, sC, false);

for (uint k = 0; k < K; k += lK) {
  uint aRow = lM * matrixID.y;
  uint aCol = k;
  coopMatLoadNV matA, inputA.x, sA * aRow + aCol, sA, false);

  uint bRow = k;
  uint bCol = lN * matrixID.x;
  coopMatLoadNV matB, inputB.x, sB * bRow + bCol, sB, false);

  matC = coopMatMulAddNV(matA, matB, matO);
}

coopMatStoreNV(matC, outputD.x, sD * cRow + cCol, sD, false);
```



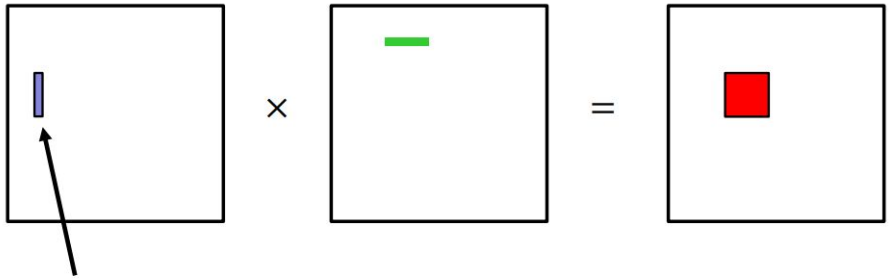
GPU	TFLOPS
RTX 2070	2.00
RTX TITAN	3.70

https://www.khronos.org/assets/uploads/developers/presentations/Cooperative_Matrix_May22.pdf

Tiled Scalar Multiply

- **Illustrates a way to tile the multiply**
  - Load row, load column, outer product
  - Name of the game is to load once, then perform as many multiplies as possible
  - Limited by register file size
  - Maybe 8x8 tile size per invocation

```
float16_t C[C_ROWS][C_COLS];
uvec2 tileID = uvec2(gl_WorkGroupID.xy);
uvec2 inv = uvec2(gl_LocalInvocationID.xy);
// load C
...
// iterate through K dimension and accumulate
for (uint k = 0; k < K; ++k) {
  float16_t A[C_ROWS];
  for (uint i = 0; i < C_ROWS; ++i) {
    uint gi = TILE_M * tileID.y + (C_ROWS * inv.y + i);
    uint gk = k;
    A[i] = inputA.x[sA * gi + gk];
  }
  float16_t B;
  for (uint j = 0; j < C_COLS; ++j) {
    uint gk = k;
    uint gj = TILE_N * tileID.x + (C_COLS * inv.x + j);
    B = inputB.x[sB * gk + gj];
    for (uint i = 0; i < C_ROWS; ++i) {
      C[i][j] = A[i] * B + C[i][j];
    }
  }
}
// store C
...
```



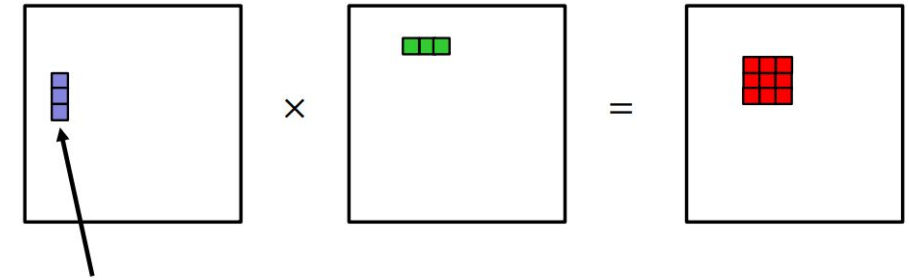
GPU	TFLOPS
RTX 2070	1.45
RTX TITAN	2.79

https://www.khronos.org/assets/uploads/developers/presentations/Cooperative_Matrix_May22.pdf

Tiled Cooperative Matrix Multiply

- Apply tiling approach to coop matrices
- Effectively 32x the register file, since matrices are spread over a subgroup
  - 112x112x16 multiply per outer loop iter
  - Takes advantage of cheap communication within subgroups to maximize reuse
- Shader code is no more complex than what you'd write for scalar

```
fcoopmatNV<16, gl_ScopeSubgroup, lM, lN> matC[C_ROWS][C_COLS];
uvec2 tileID = uvec2(gl_WorkGroupID.xy);
// load matC
...
for (uint k = 0; k < K; k += TILE_K) {
  fcoopmatNV<16, gl_ScopeSubgroup, lM, lK> matA[C_ROWS];
  for (uint i = 0; i < C_ROWS; ++i) {
    uint gi = TILE_M * tileID.y + lM * i;
    uint gk = k;
    coopMatLoadNV(matA[i], inputA.x, sA * gi + gk, sA, false);
  }
  fcoopmatNV<16, gl_ScopeSubgroup, lK, lN> matB;
  for (uint j = 0; j < C_COLS; ++j) {
    uint gk = k;
    uint gj = TILE_N * tileID.x + lN * j;
    coopMatLoadNV(matB, inputB.x, sB * gk + gj, sB, false);
    for (uint i = 0; i < C_ROWS; ++i) {
      matC[i][j] = coopMatMulAddNV(matA[i], matB, matC[i][j]);
    }
  }
}
// store matC
...
```

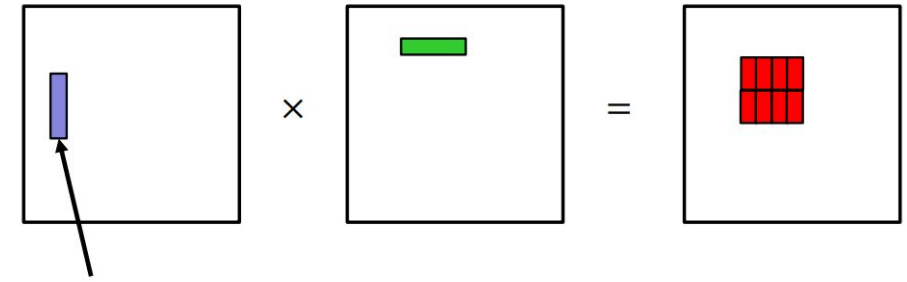


GPU	TFLOPS
RTX 2070	21.8
RTX TITAN	42.1

https://www.khronos.org/assets/uploads/developers/presentations/Cooperative_Matrix_May22.pdf

Staging Through Shared Memory

- **Subgroups cooperate to copy data from buffer to shared, then load out of shared**
  - Overlap buffer loads with matrix math (i.e. pipeline load for next iteration)
- **Example (FP16): 8 subgroups split a 256x256 tile into 8 128x64 tiles (K=32)**
  - Cooperate to copy A block (256x32) and B block (32x256) into shared memory
  - Then each subgroup loads the portions it needs from shared memory
  - Only 128B of buffer loads per thread per K iteration overlapping with 8K FMADs
- **Example (INT8): 8 subgroups split a 128x256 tile into 8 64x64 tiles (K=64)**
  - Accumulator is INT32, requires 2x register file, hurts latency hiding
  - Double math rate, half tile size = 4x harder to hide latency, much farther from SOL



GPU	INT8 TOPS
RTX 2070	66.0
RTX TITAN	~130

GPU	FP16 TFLOPS
RTX 2070	49.0
RTX TITAN	~100

https://www.khronos.org/assets/uploads/developers/presentations/Cooperative_Matrix_May22.pdf

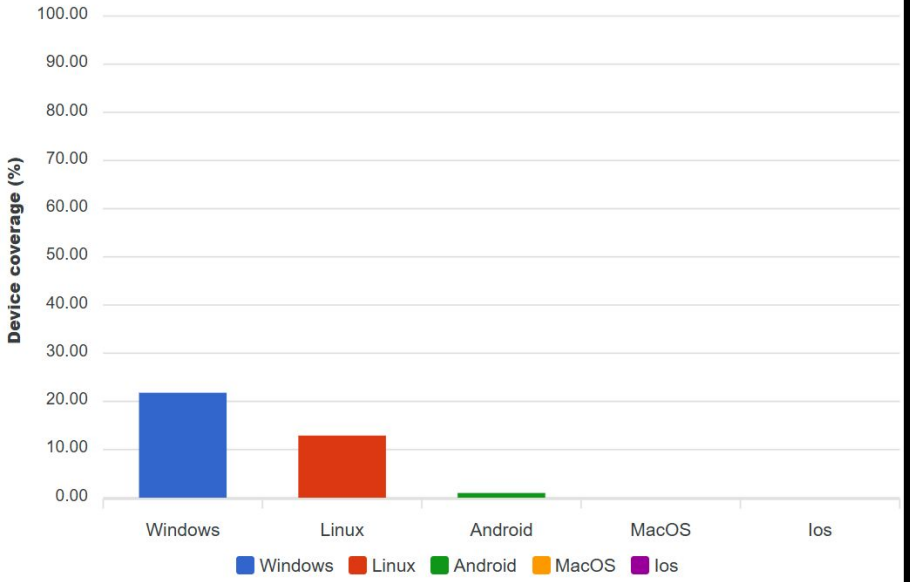
Device coverage for VK_KHR_cooperative_matrix

Vulkan registry manpage

This extension has additional features and properties

Extension was first submitted at 2023-06-27

Vulkan extension: VK_KHR_cooperative_matrix



Platform	Coverage
Windows	21.79%
Linux	12.9%
Android	0.98%

Listing first known driver version support for VK_KHR_cooperative_matrix

This extension has additional features and properties

Extension was first submitted at 2023-06-27

All platforms | Windows | Linux | Android | macOS | iOS

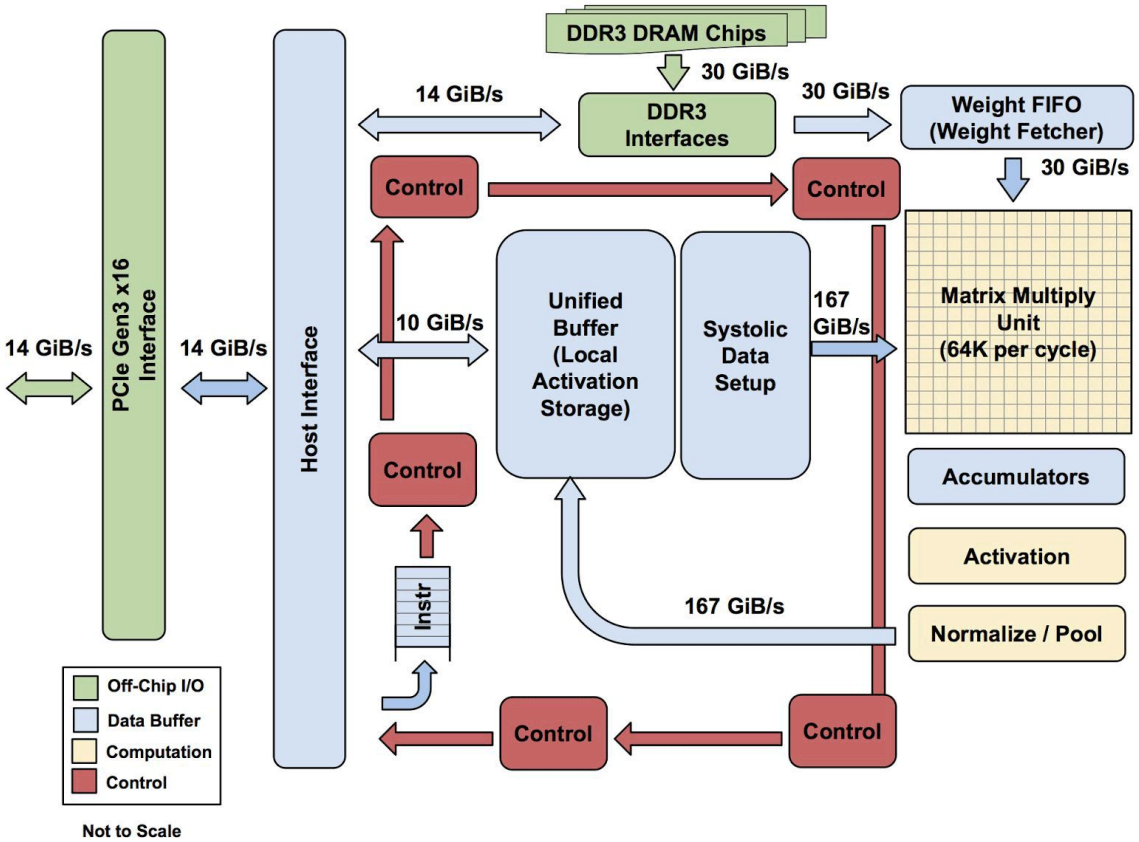
Show 50 entries

amd	Type to filter	Type to filter		
Device	Vendor	Driver [?]	Date	Compare
AMD Radeon(TM) 740M	AMD	2.0.302	2025-09-16	Add
AMD Radeon RX 5500	AMD	2.0.310	2025-08-26	Add
AMD Radeon Pro 5500M	AMD	2.0.317	2025-08-13	Add
AMD Radeon(TM) 860M Graphics	AMD	2.0.331	2025-07-23	Add

intel	Type to filter	Type to filter		
Device	Vendor	Driver [?]	Date	Compare
Intel(R) Iris(R) Xe Graphics (ADL GT2)	INTEL	6400.4099	2025-07-09	Add
Intel(R) Arc(TM) 130V GPU (8GB)	INTEL	101.6314	2025-07-06	Add
Intel(R) Arc(TM) 130V GPU (16GB)	INTEL	101.6556	2025-06-23	Add
Intel(R) Arc(TM) 140T GPU (32GB)	INTEL	101.6554	2025-05-31	Add
Intel(R) Arc(TM) B570 Graphics	INTEL	101.6790	2025-05-19	Add

https://vulkan.gpuinfo.org/displayextensiondetail.php?extension=VK_KHR_cooperative_matrix

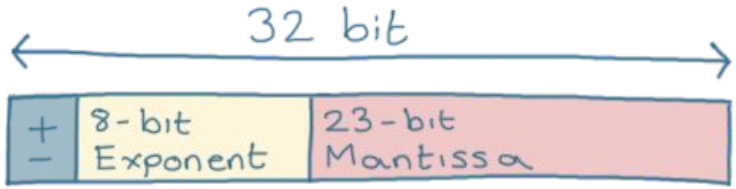
Tensor Processor Unit - TPU (Google)



Глава 4: Умножение матриц

DeepSeek

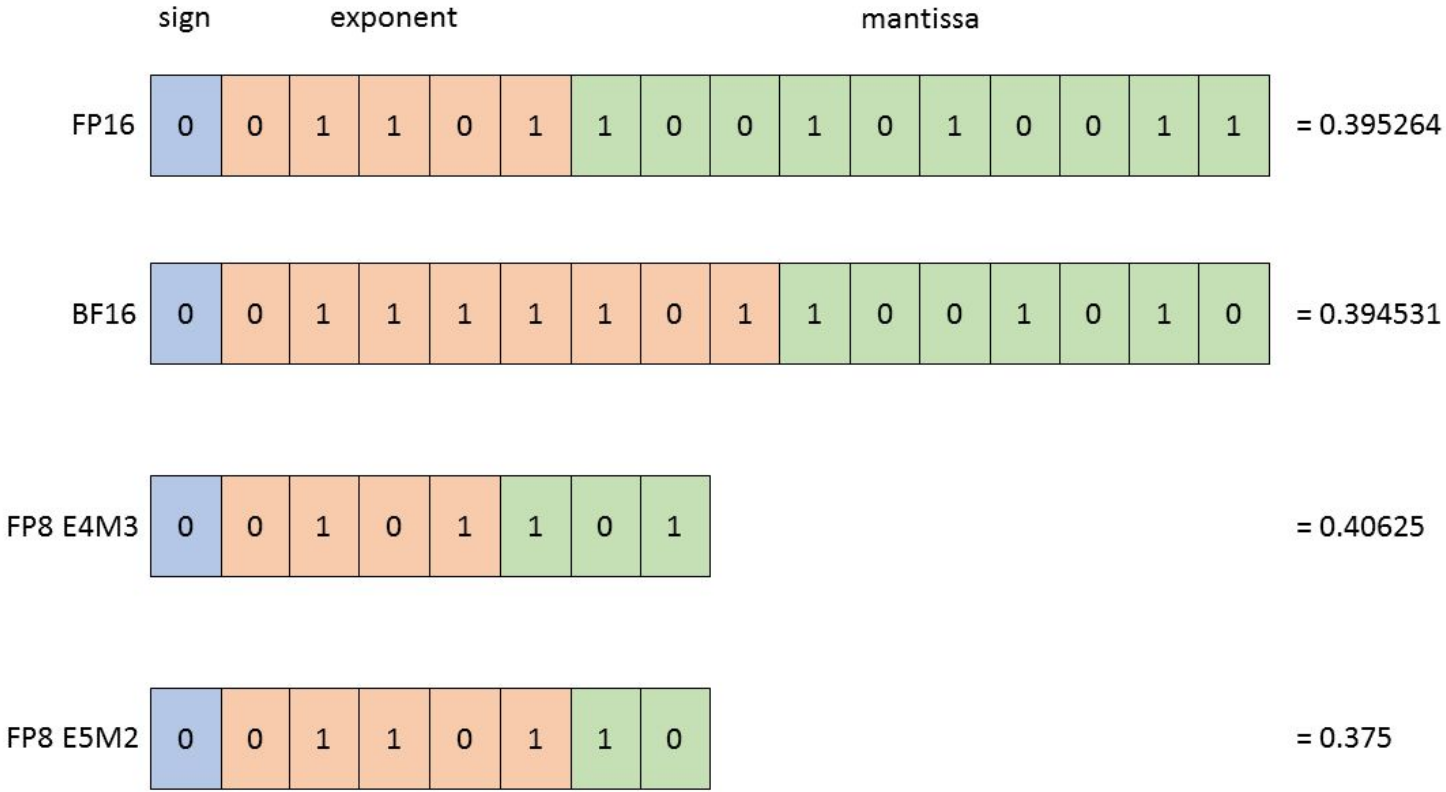
DeepSeek: x2 ускорение обучения (fp16 → fp8)



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

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

DeepSeek: x2 ускорение обучения (fp16 → fp8)



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

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**

*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**

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**

x2

Хватит ли пропускной способности памяти чтобы насытить ALU (tensor cores)?

*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**

x2

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*):

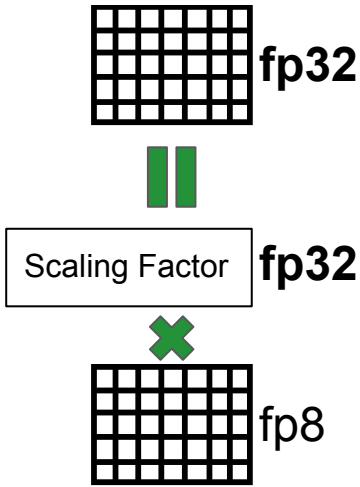
- Memory bandwidth: **3.35 TB/sec**
- FP 32: **67 TFlops**
- FP 16: **268 TFlops**
- FP 32 (tensor): **495 TFlops**
- FP 16 (tensor): **990 TFlops**
- FP 8 (tensor): **1979 TFlops**
- DeepGEMM достиг **1550 TFlops** на H800!

DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437
Репозиторий - https://github.com/deepseek-ai/DeepGEMM (GEMM - General Matrix Multiplications)

DeepSeek: fine-grained fp8 quantization

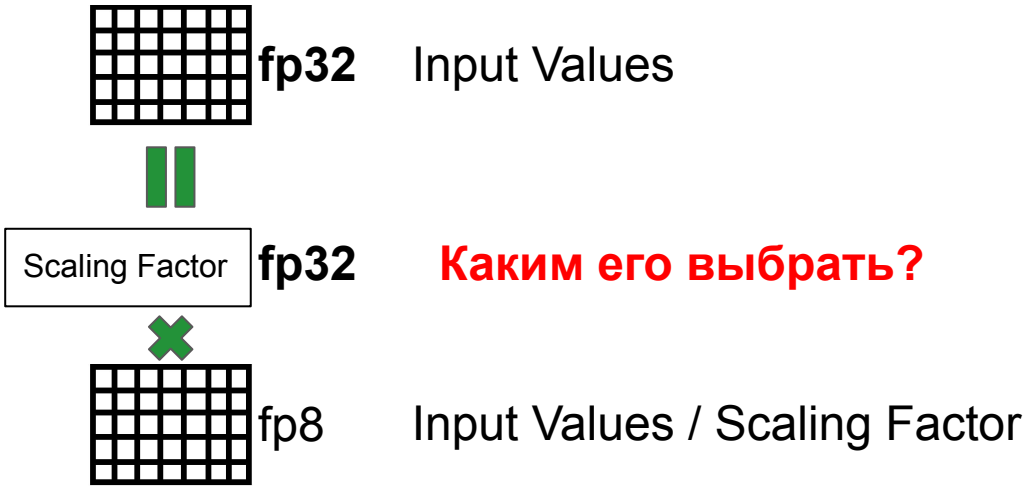
DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

DeepSeek: fine-grained fp8 quantization



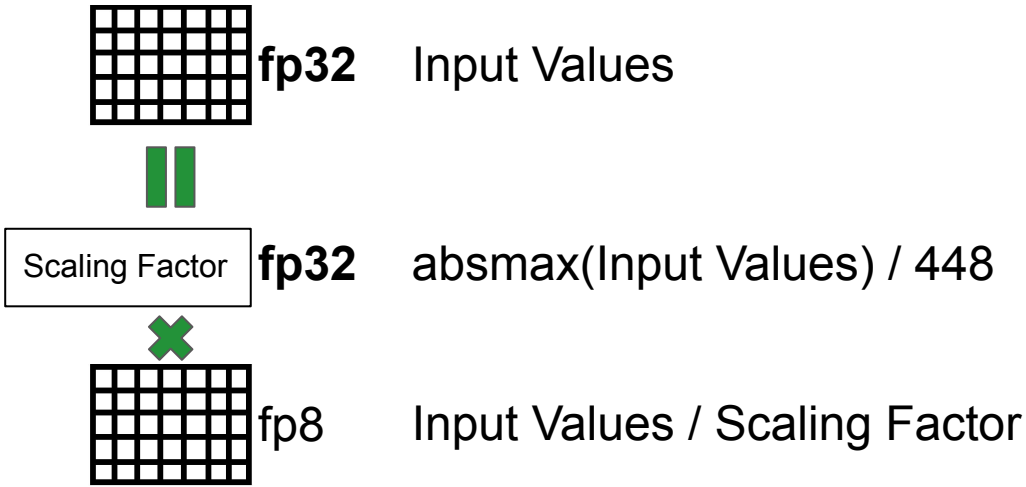
DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

DeepSeek: fine-grained fp8 quantization



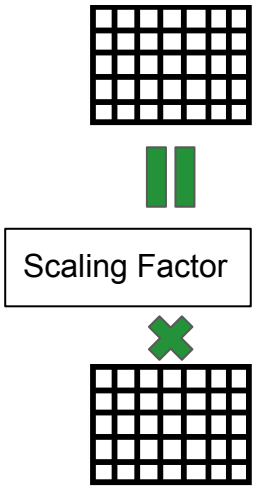
DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

DeepSeek: fine-grained fp8 quantization



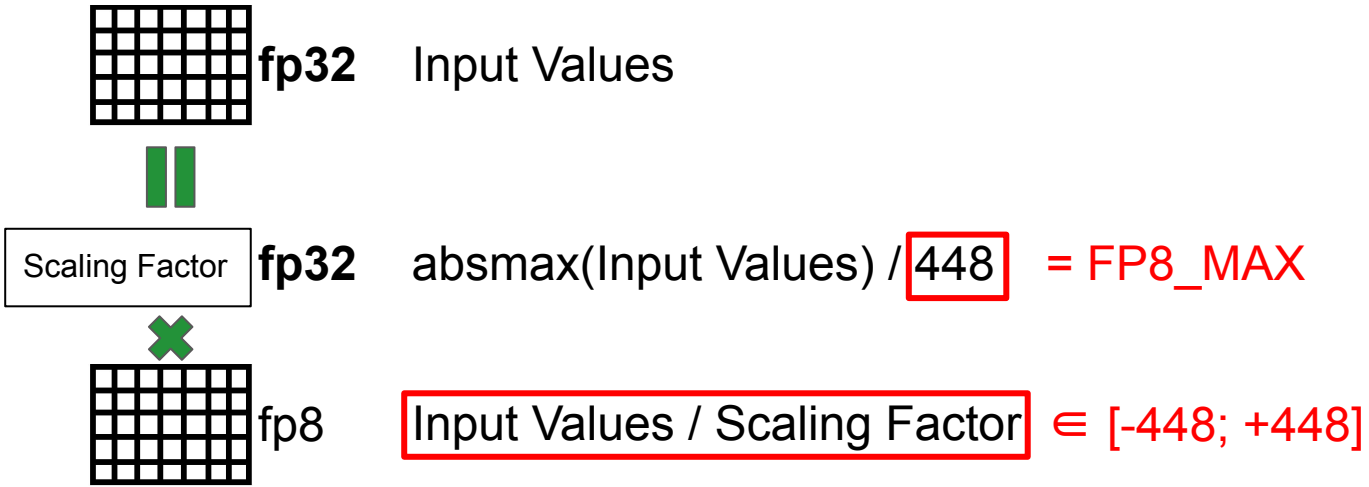
DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

DeepSeek: fine-grained fp8 quantization



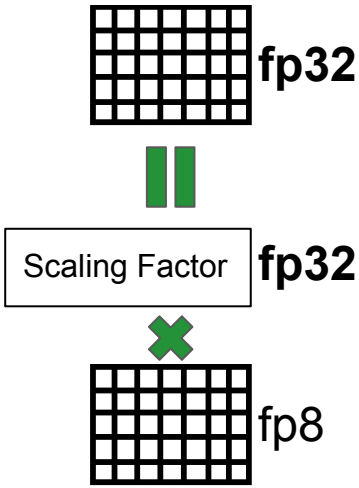
DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

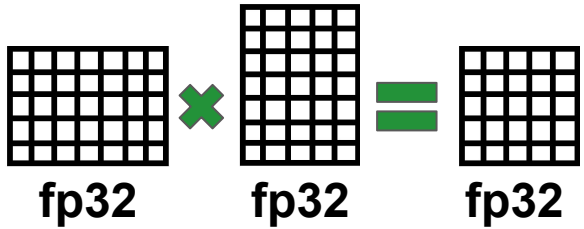
DeepSeek: fine-grained fp8 quantization



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

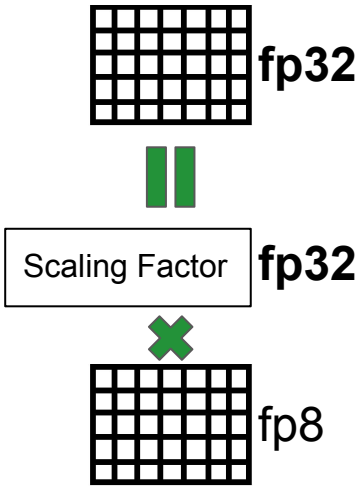
DeepSeek: fine-grained fp8 quantization

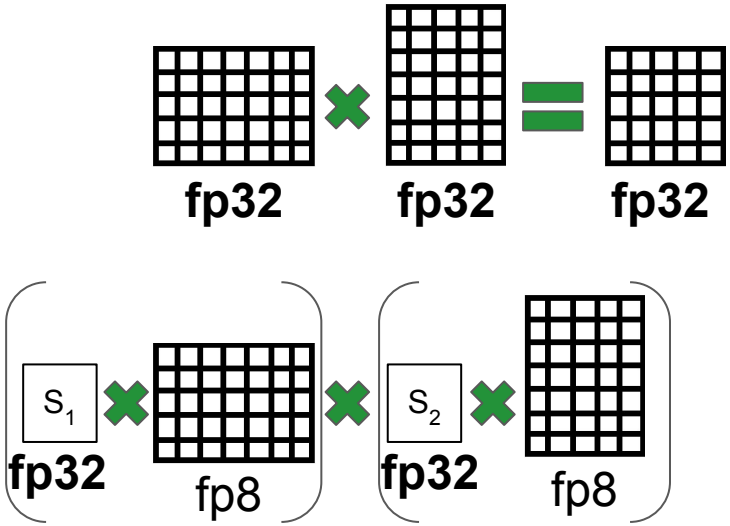




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

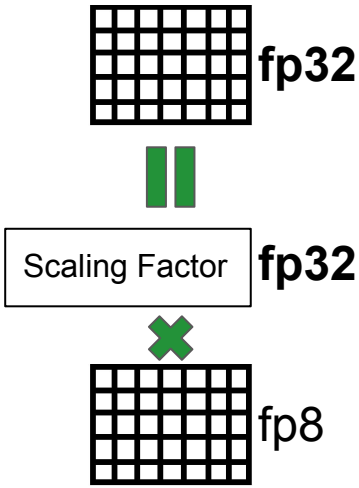
DeepSeek: fine-grained fp8 quantization

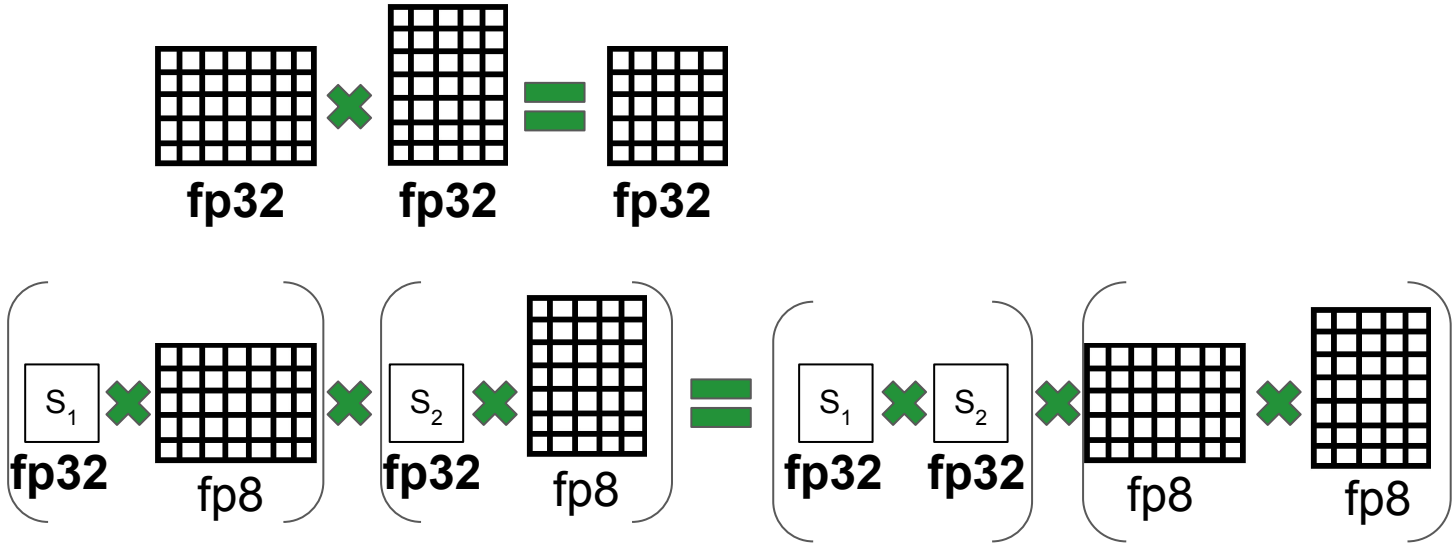




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

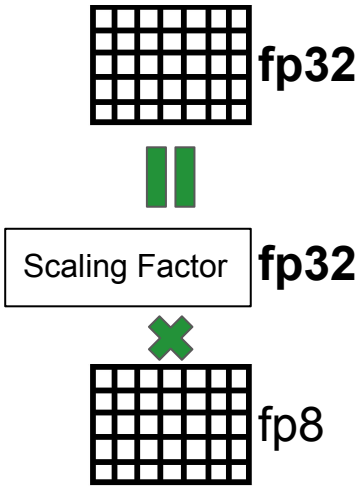
DeepSeek: fine-grained fp8 quantization

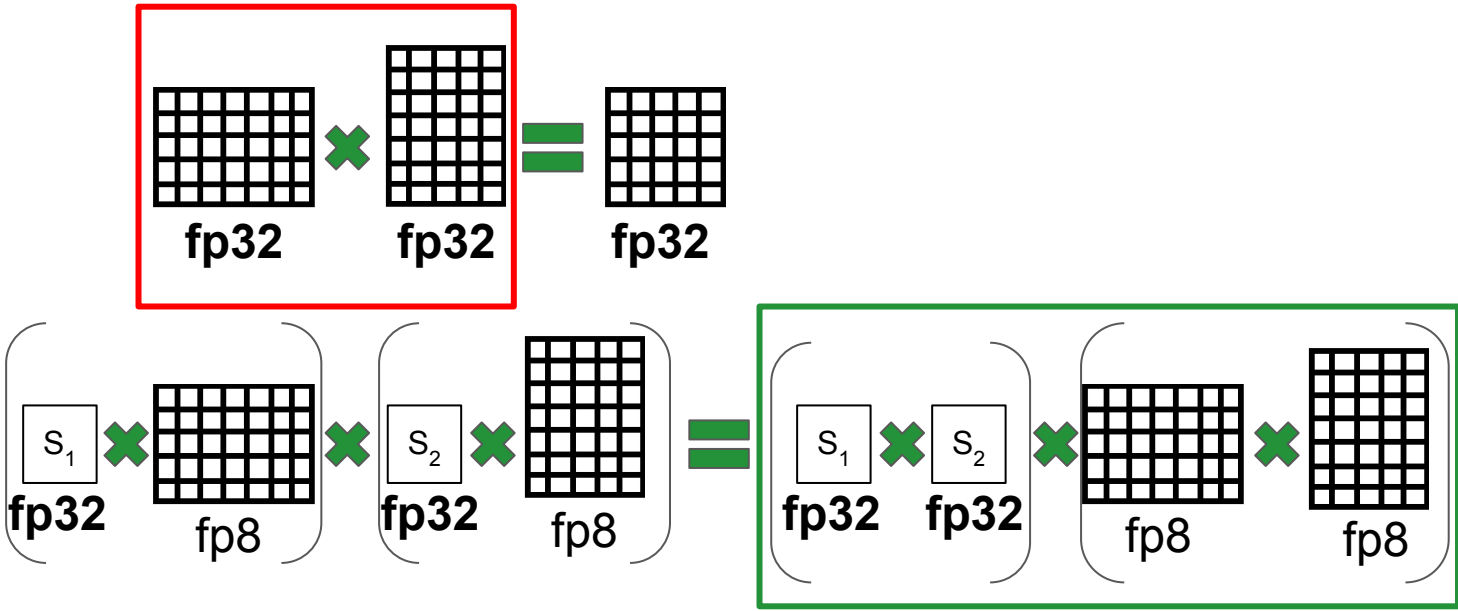




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

DeepSeek: fine-grained fp8 quantization

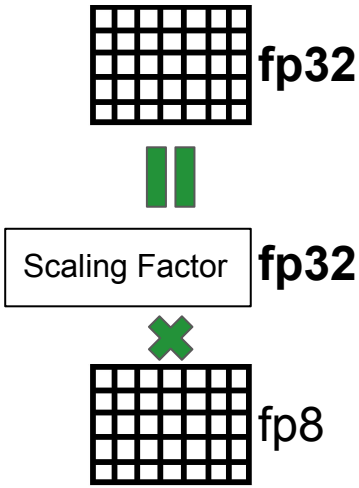


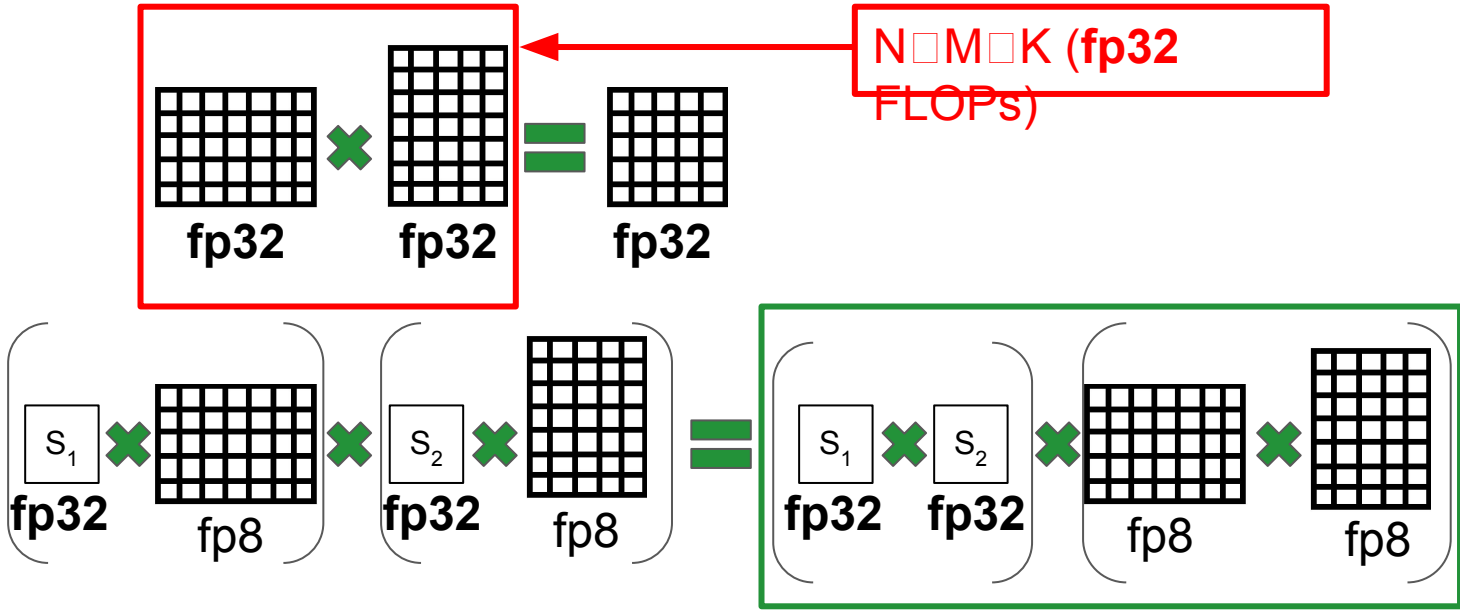


Насколько быстрее?

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

DeepSeek: fine-grained fp8 quantization

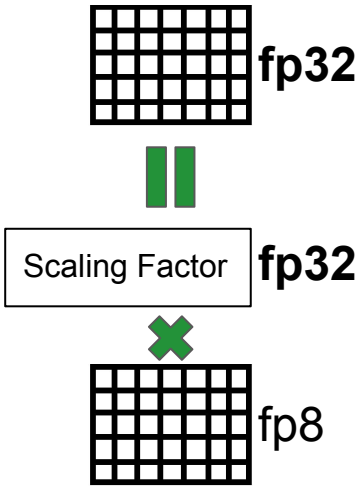


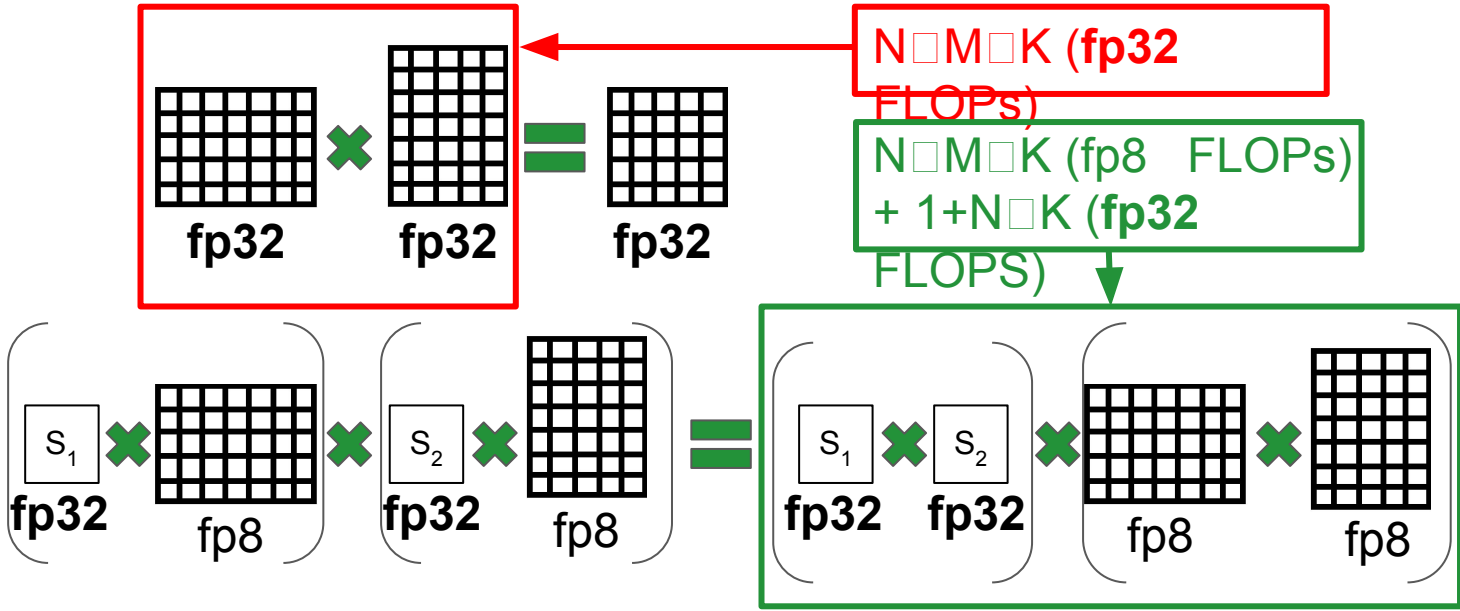


Насколько быстрее?

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

DeepSeek: fine-grained fp8 quantization

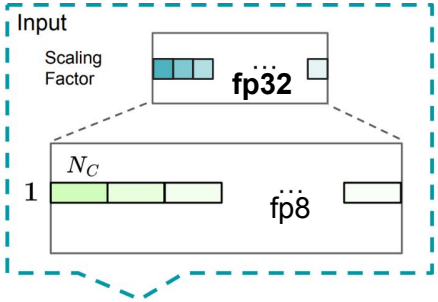


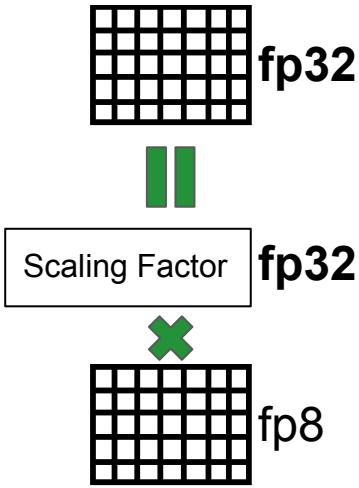


Насколько быстрее?

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

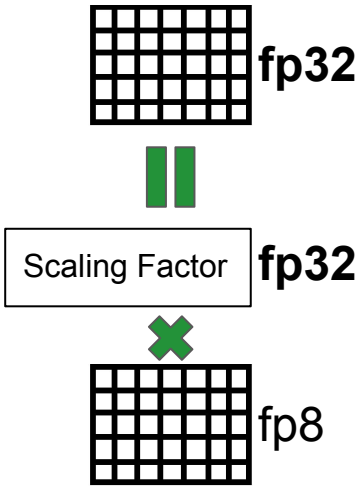
DeepSeek: fine-grained fp8 quantization

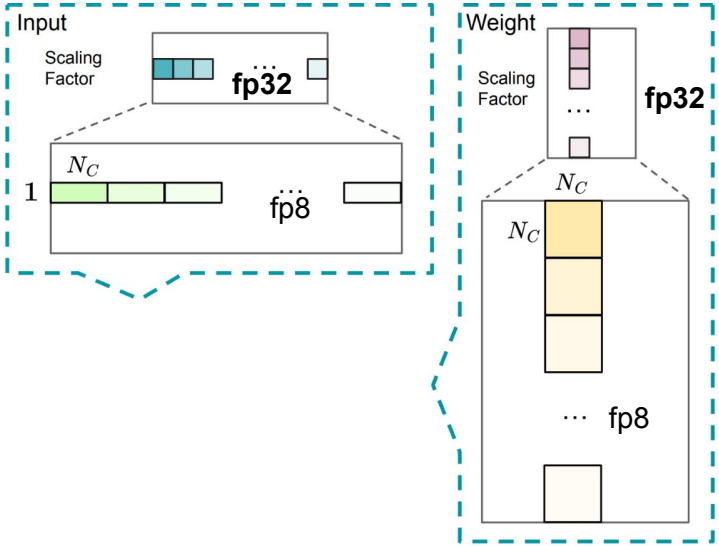




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

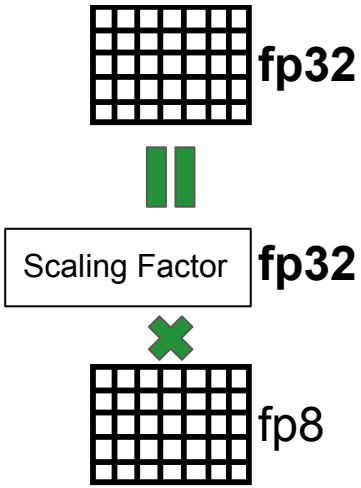
DeepSeek: fine-grained fp8 quantization

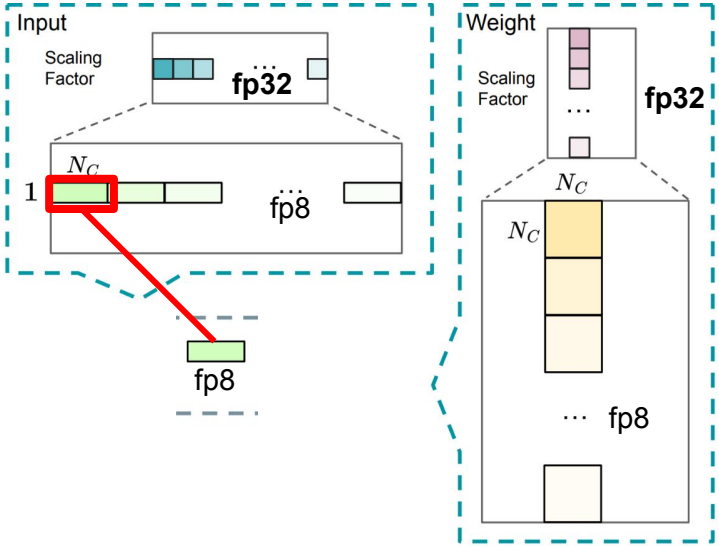




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

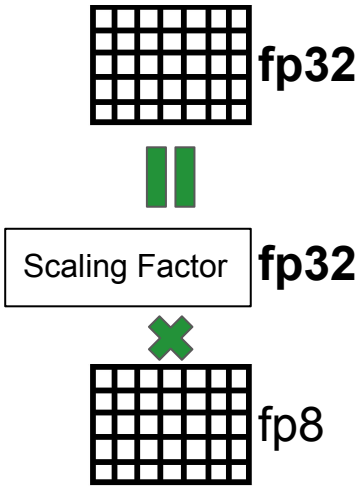
DeepSeek: fine-grained fp8 quantization

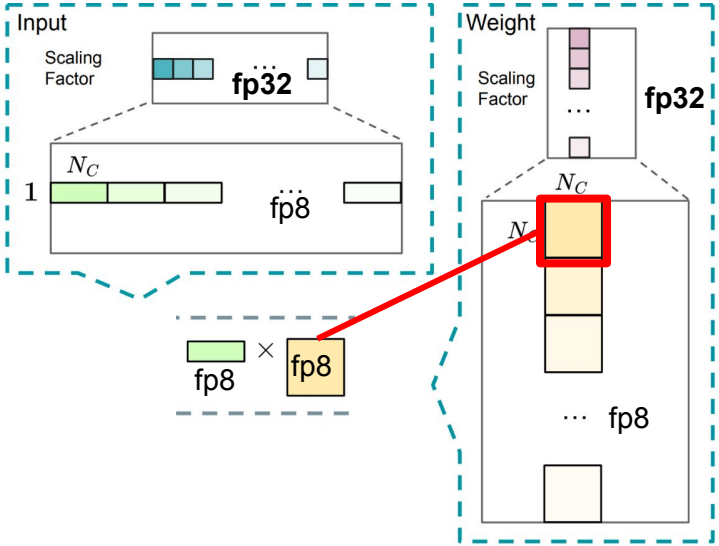




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

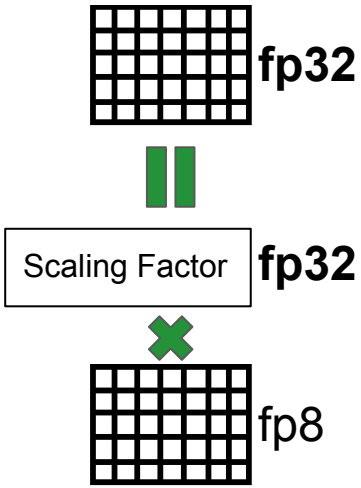
DeepSeek: fine-grained fp8 quantization

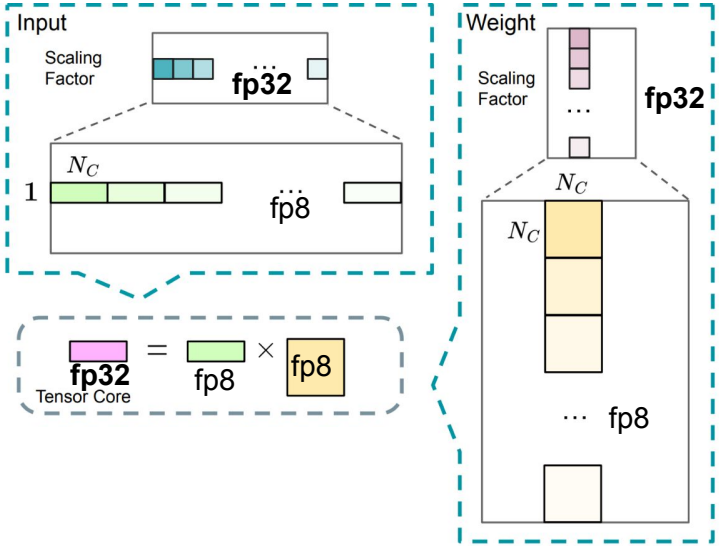




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

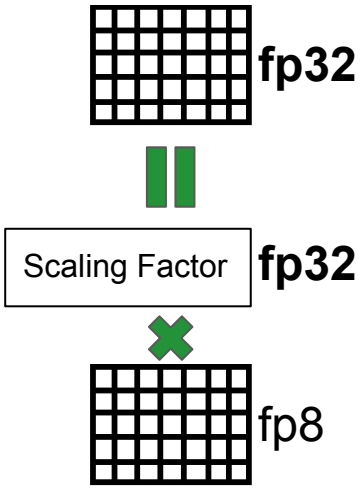
DeepSeek: fine-grained fp8 quantization

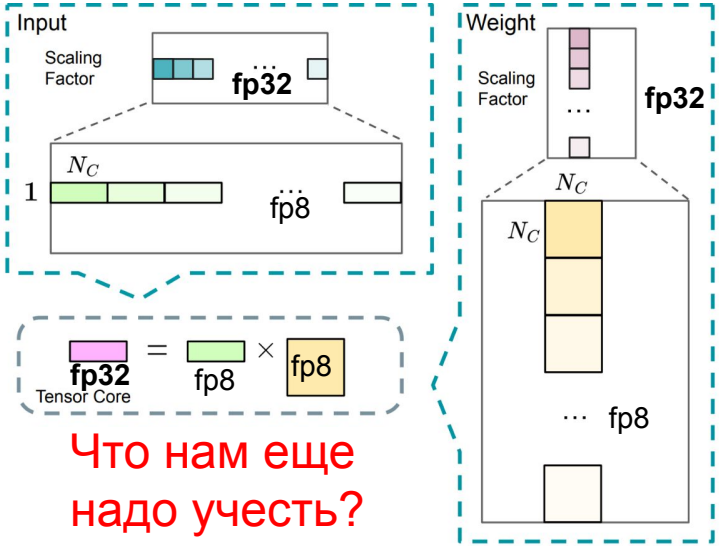




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

DeepSeek: fine-grained fp8 quantization

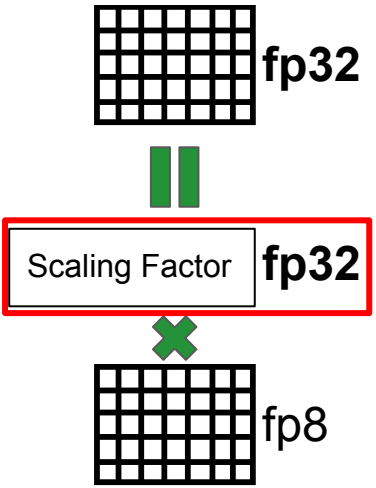


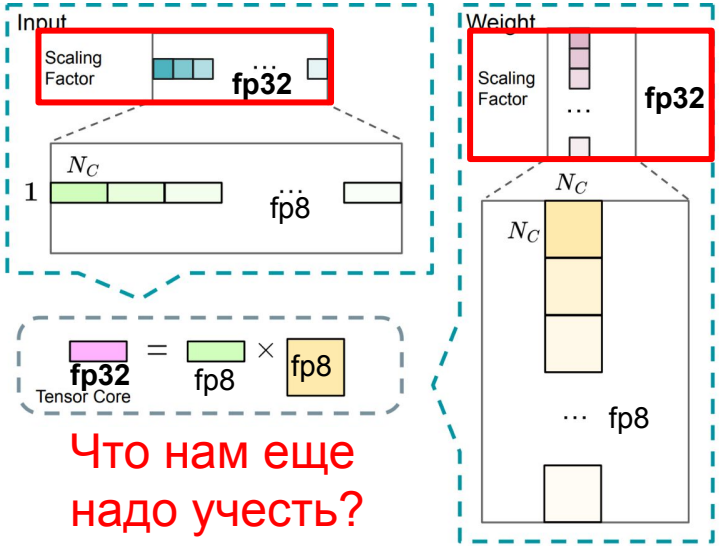


Что нам еще надо учесть?

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

DeepSeek: fine-grained fp8 quantization

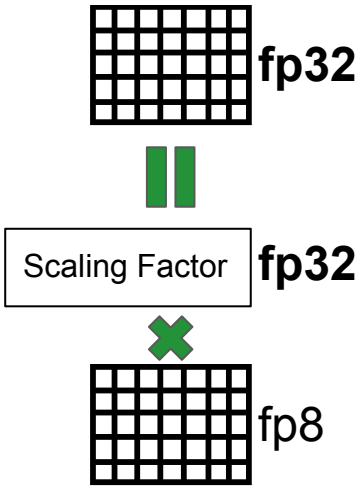


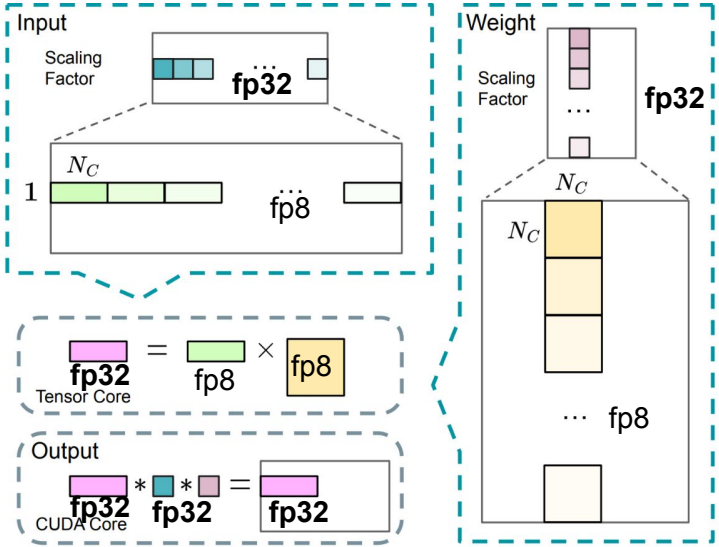


Что нам еще надо учесть?

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

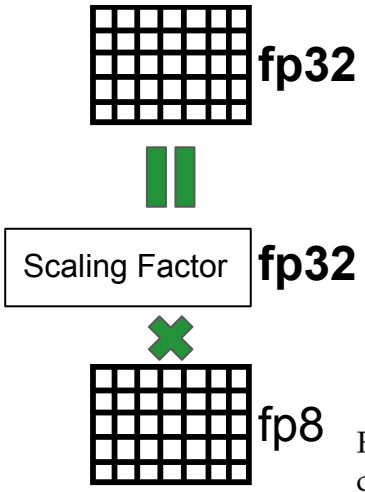
DeepSeek: fine-grained fp8 quantization

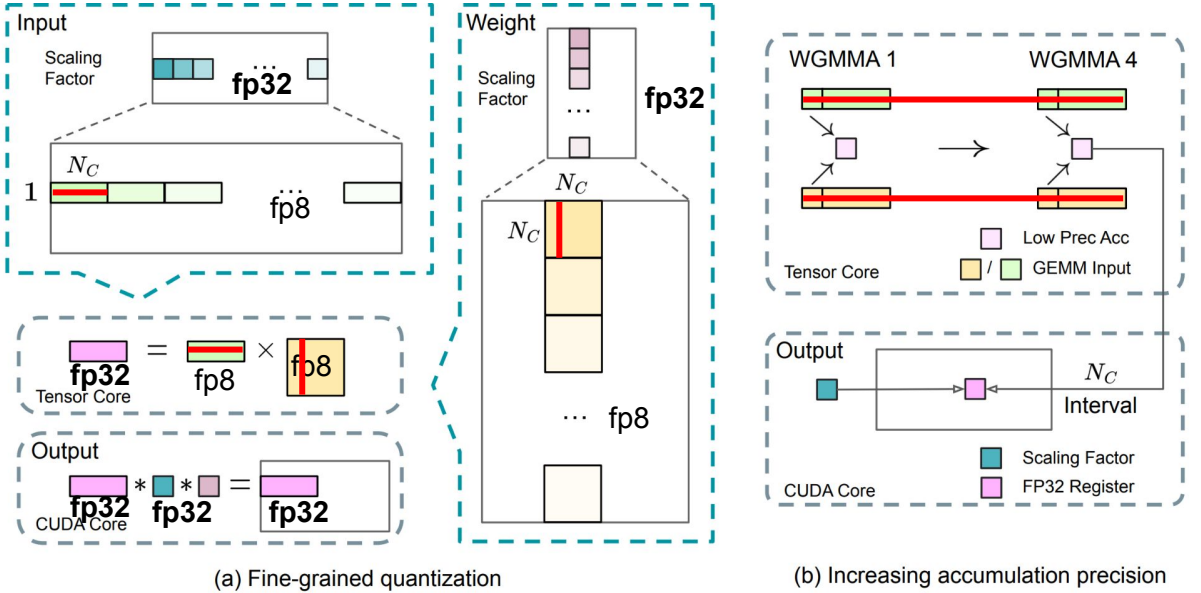




DeepSeek-V3 Technical Report - https://arxiv.org/abs/2412.19437

DeepSeek: fine-grained fp8 quantization





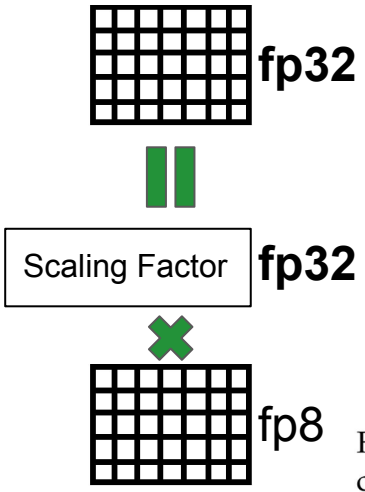
(a) Fine-grained quantization

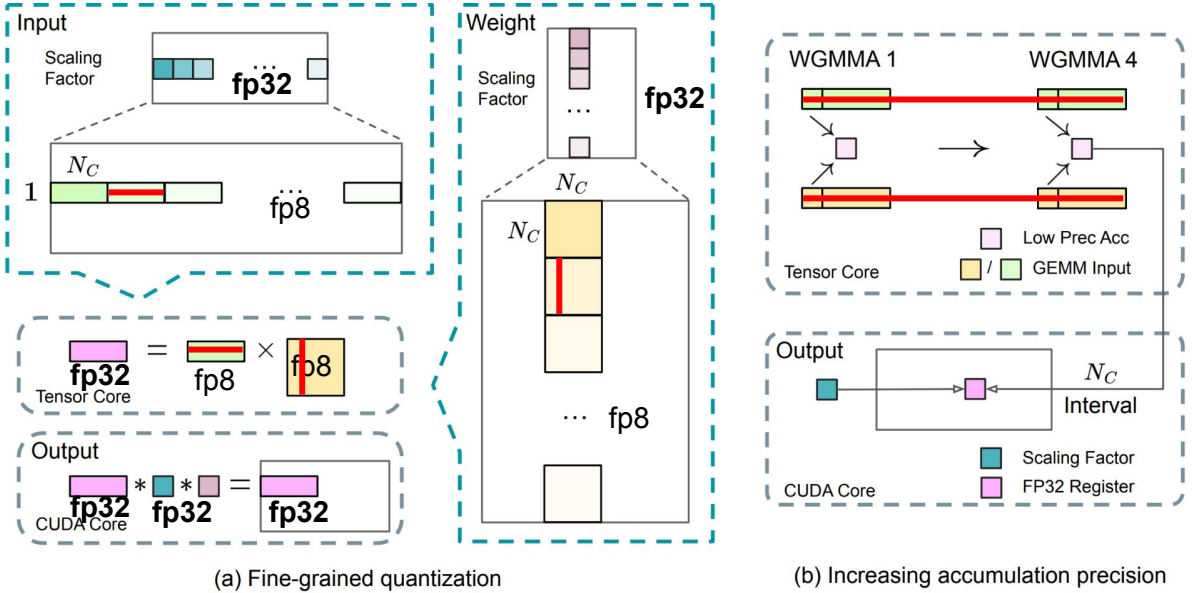
(b) Increasing accumulation precision

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

DeepSeek: fine-grained fp8 quantization





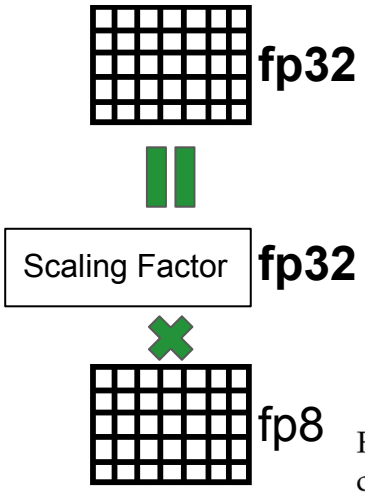
(a) Fine-grained quantization

(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-V3 Technical Report - https://arxiv.org/abs/2412.19437

DeepSeek: fine-grained fp8 quantization



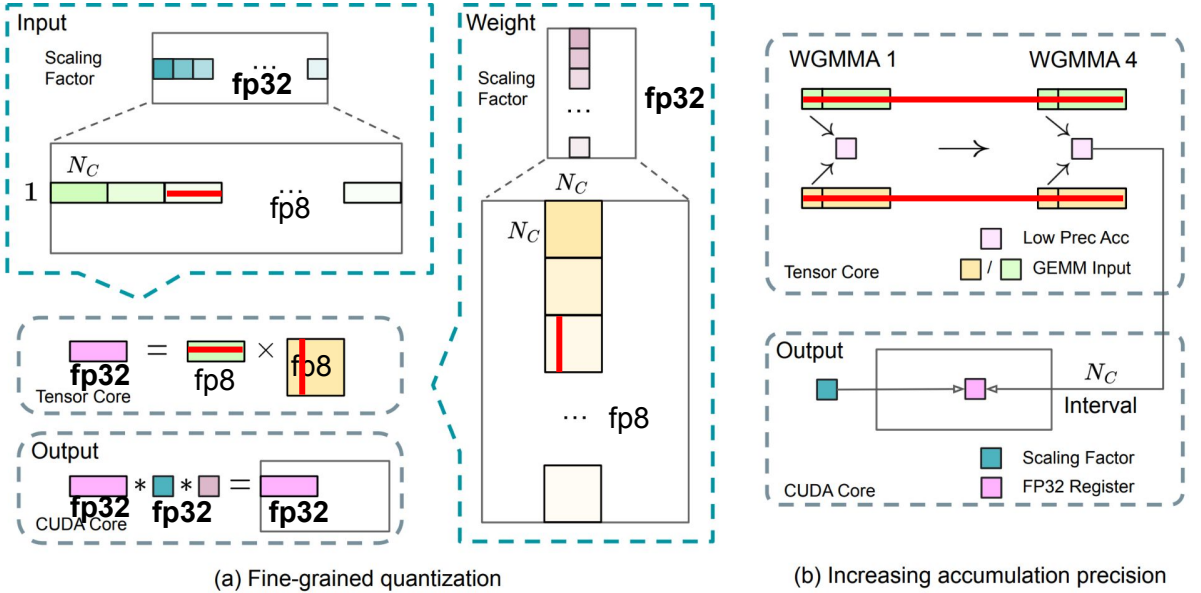
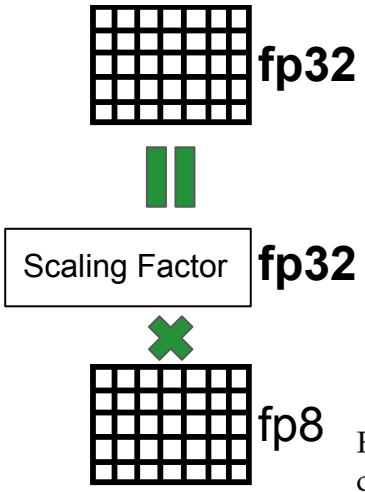
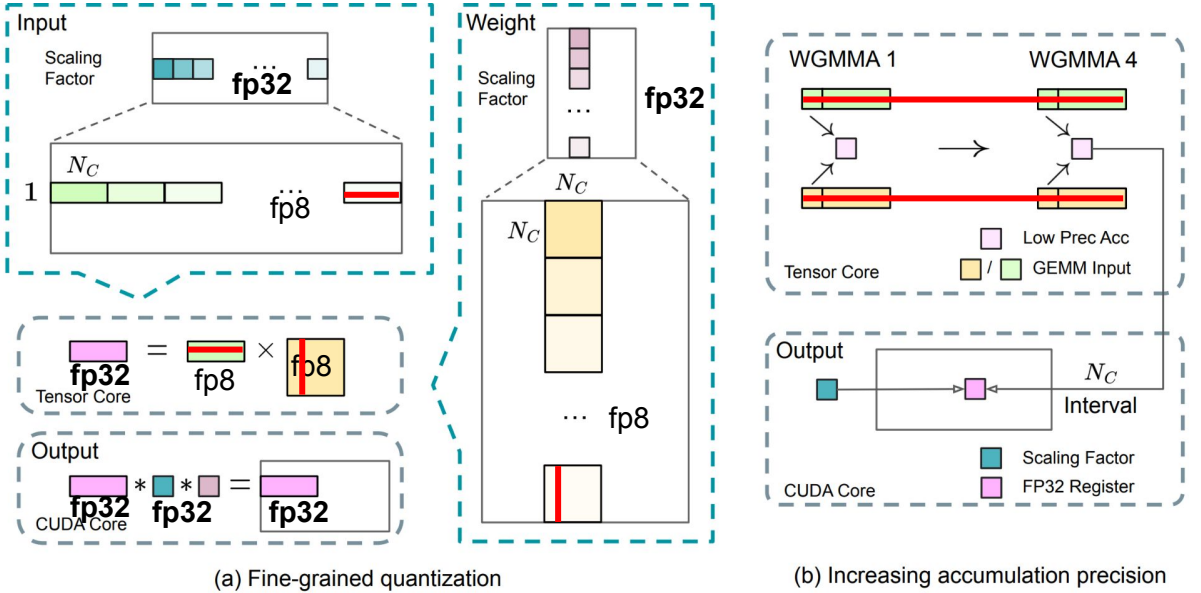


Figure 7 | (a) We propose a fine-grained quantization method to mitigate quantization errors caused by feature outliers; for illustration simplicity, only `Fprop` is illustrated. (b) In conjunction with our quantization strategy, we improve the FP8 GEMM precision by promoting to CUDA Cores at an interval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

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

DeepSeek: fine-grained fp8 quantization





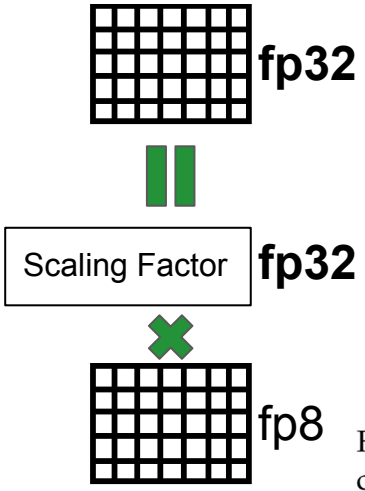
(a) Fine-grained quantization

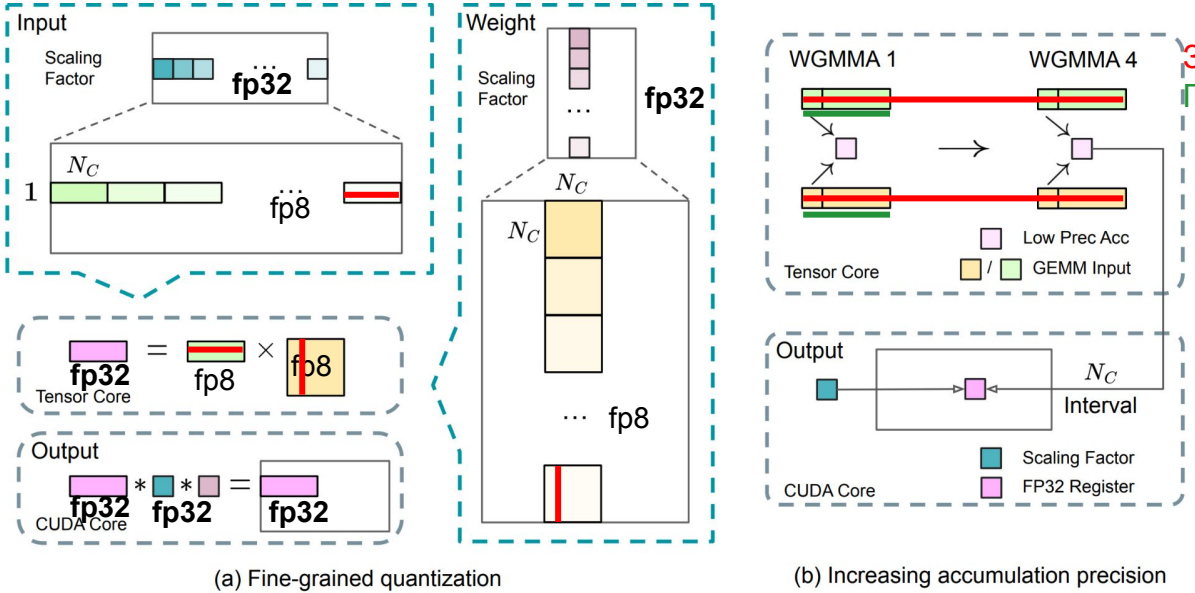
(b) Increasing accumulation precision

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

DeepSeek: fine-grained fp8 quantization



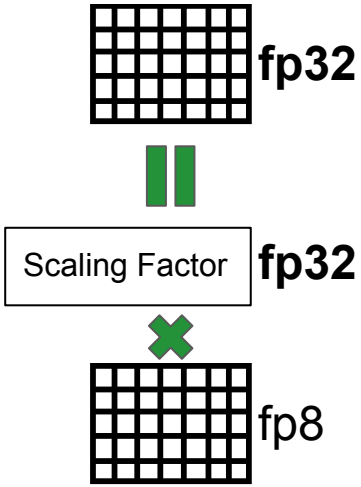


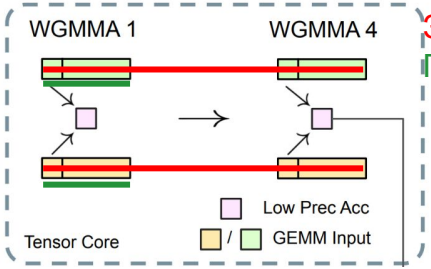
Зачем 4xWGMMA?
Почему не 1xWGMMA?

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.

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

DeepSeek: fine-grained fp8 quantization



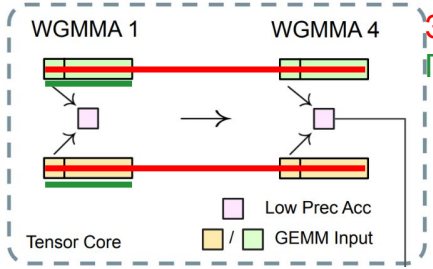


Зачем 4xWGMMA?
Почему не 1xWGMMA?

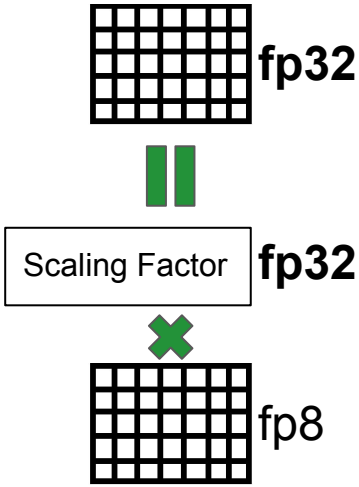
It is worth noting that this modification reduces the WGMMA (Warpgroup-level Matrix Multiply-Accumulate) instruction issue rate for a single warpgroup. However, on the H800 architecture, it is typical for two WGMMA to persist concurrently: while one warpgroup performs the promotion operation, the other 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 WGMMAs, represents the minimal accumulation interval that can significantly improve precision without introducing substantial overhead.

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

DeepSeek: fine-grained fp8 quantization



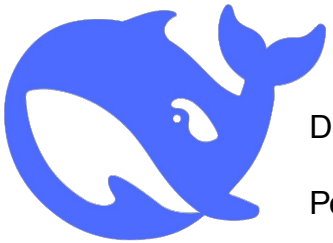
Зачем 4xWGMMA?
Почему не 1xWGMMA?



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 WGMMAs, 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-V3 Technical Report - https://arxiv.org/abs/2412.19437



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

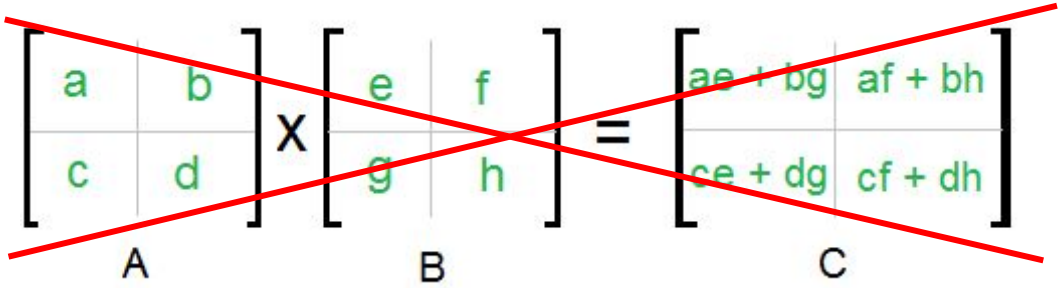
Глава 5: Умножение матриц

Метод Штрассена

Умножение матриц - Метод Штрассена

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

Умножение матриц - Метод Штрассена

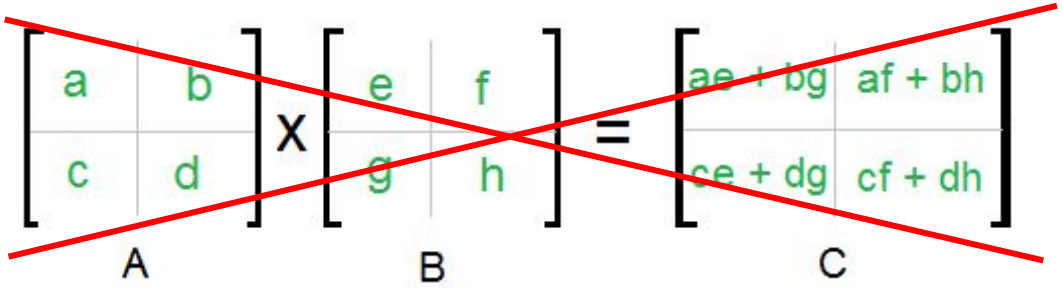


p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

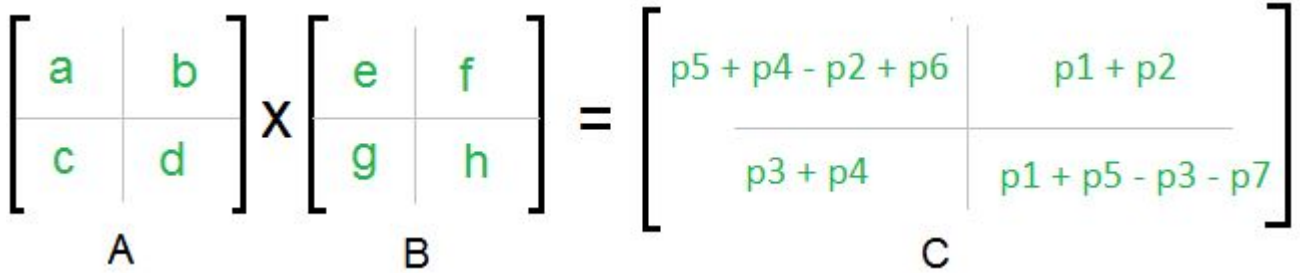
$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \times \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} p5 + p4 - p2 + p6 & p1 + p2 \\ p3 + p4 & p1 + p5 - p3 - p7 \end{bmatrix}$$

A B C

Умножение матриц - Метод Штрассена

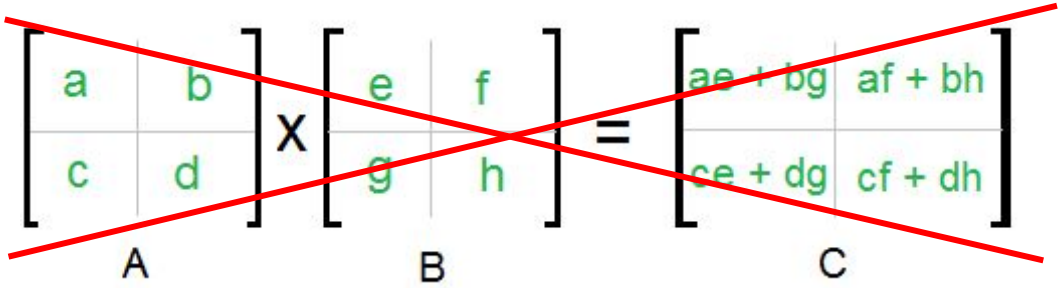


p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)

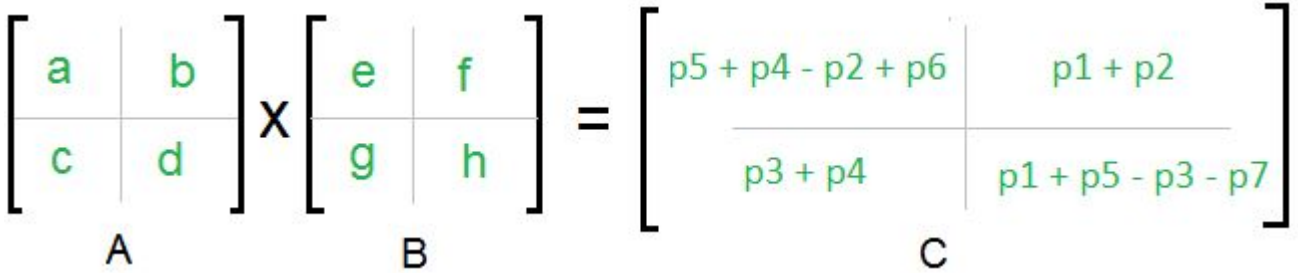


Получили рекуррентную асимптотику T(N)=**7**T(N/2)+8O(N^2) =>
T(N)=O(N^log7)=**O(N^2.8)**

Умножение матриц - Метод Штрассена

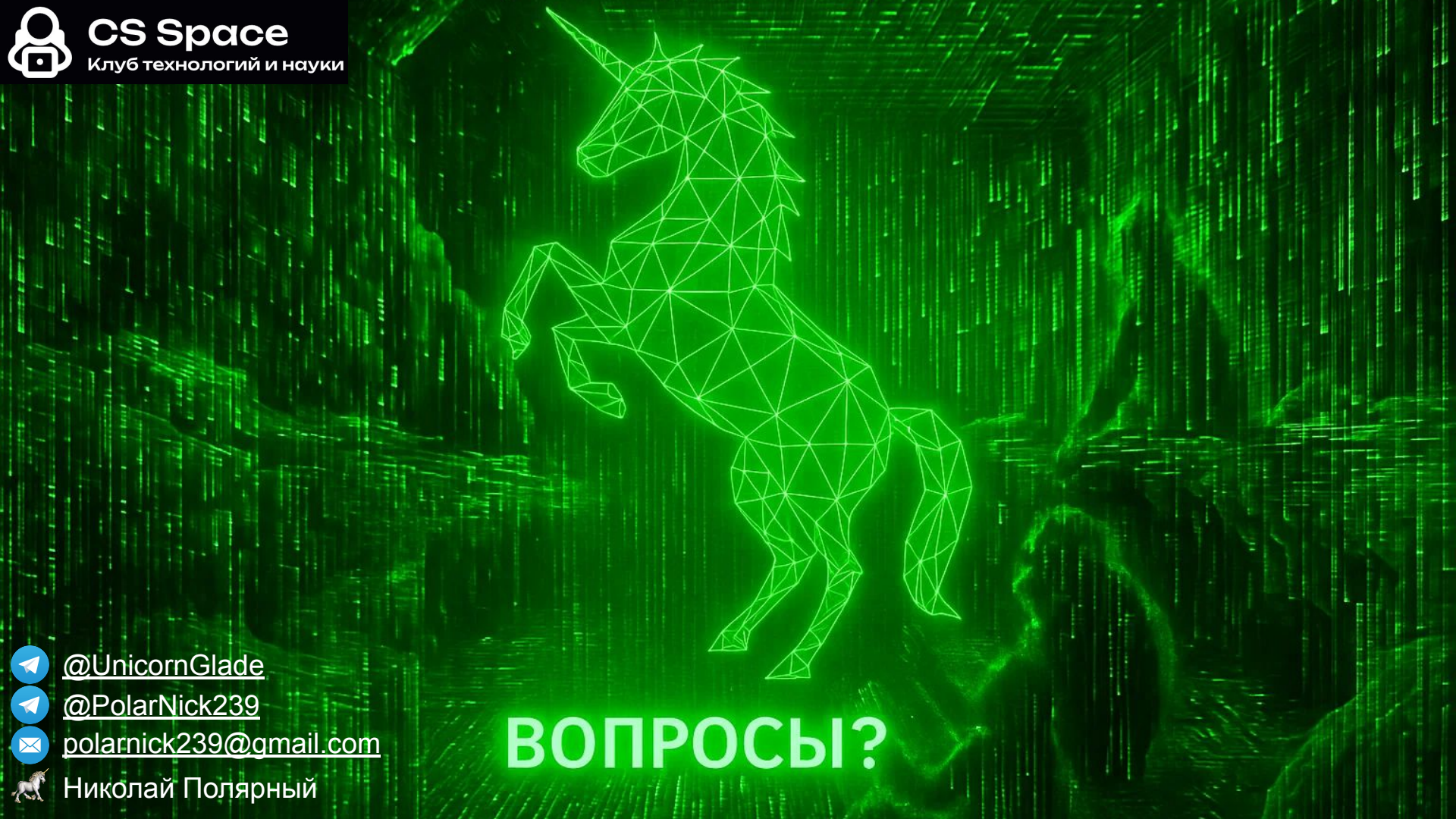


p1 = a(f - h)
p2 = (a + b)h
p3 = (c + d)e
p4 = d(g - e)
p5 = (a + d)(e + h)
p6 = (b - d)(g + h)
p7 = (a - c)(e + f)



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

Метод Виноградова - тоже **O(N^2.8)**
https://www.geeksforgeeks.org/strassens-matrix-multiplication/


CS Space
Клуб технологий и науки
@UnicornGlade
@PolarNick239
polarnick239@gmail.com
Николай Полярный
ВОПРОСЫ?