Efficient models: GPU

Egor Shvetsov

Today's plan:

1. Previous RECAP
2. Data Types
3. GPU Architecture
4. How to compute on GPU
5. Triton Language

>>> RECAP

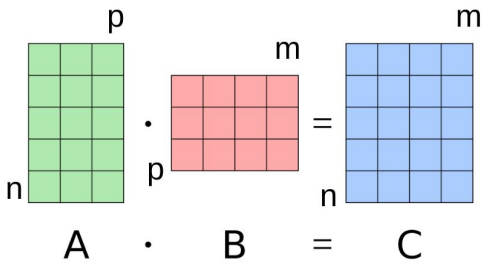
Ресурсы

Memory
FLOPs
MAC
Latency
Energy

FLOPs - number of floating point operations?

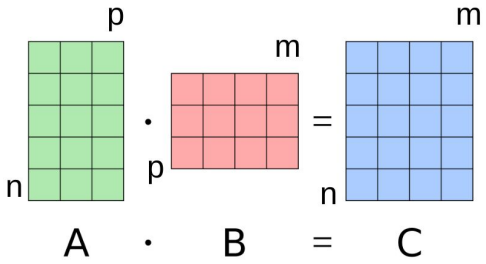
How much is it for a matrix multiplication?

$$F[(n \times p) \times (p \times m)] = ?$$



Сколько флопов уходит на умножение матриц?

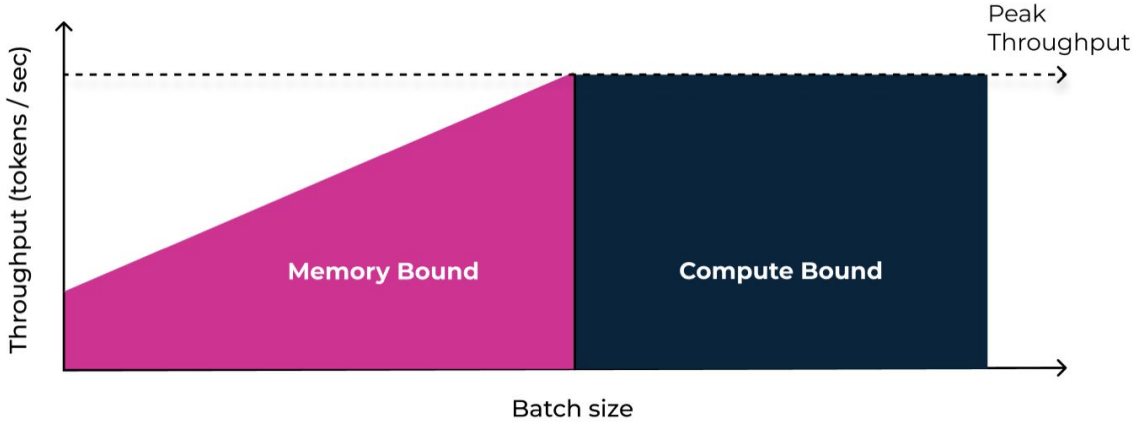
- 2npm сложений и lmn умножений, всего 2npm флопов.
- Это 8 умножений для матриц размера 2×2.
- Алгоритм Штрассена делает это за 7 умножений.
- На практике не используется из-за архитектуры «железа».
- На самом деле ответ pmn, а не 2pmn потому что fma. (Fused multiply–add)



Memory
FLOPs
MAC
Latency
Energy

Multiply–accumulate (MAC) or multiply-add (MAD) operation is a common step that computes the product of two numbers and adds that product to an accumulator. The hardware unit that performs the operation is known as a multiplier–accumulator (MAC unit).

Memory
FLOPs
MAC
Latency
Energy



Memory
FLOPs
MAC
Latency
Energy

Memory limited layers - слои которые почти не ограничены в compute bound:

- Normalization (scaling)
- Batch Normalization
- Activations
- Pooling
- Else ... (Your ideas) ?

source: memory-Limited Layers

model	gpu	task	energy	throughput	response_length	latency	arc	hellaswag	truthfulqa	parameters
MetaAI/Llama-7B	A100	chat	335.26	27.47	60.65	1.99	51.11	77.74	34.08	7
MetaAI/Llama-13B	A100	chat	631.57	23.2	76.47	2.97	56.31	80.86	39.9	13
tatsu-lab/alpaca-7B	A100	chat	684.17	28	132.85	4.63	52.65	76.91	39.55	7
RWKV/rwkv-raven-7b	A100	chat	735.77	61.52	214.78	3.08	39.42	66.45	38.54	7
Neutralzz/BiLLa-7B-SFT	A100	chat	881.44	33.57	161.81	4.79	27.73	26.04	49.05	7
H2OAI/H2OGPT-oasst1-7B	A100	chat	951.08	33.2	212.46	6.39	36.86	61.55	37.94	7
FreedomIntelligence/phoenix-inst-chat-7b	A100	chat	1030.89	55.84	240.34	4.12	44.97	63.22	47.08	7
StabilityAI/stablelm-tuned-alpha-7b	A100	chat	1124.4	45.42	243.88	5.21	31.91	53.59	40.22	7
databricks/dolly-v2-12B	A100	chat	1242.92	22.11	153.76	7.94	42.15	71.83	33.37	12
Salesforce/xgen-7b-8k-inst	A100	chat	1246.09	49.54	275.27	5.6	46.67	74.85	41.89	7
BAIR/koala-7b	A100	chat	1345.55	26.56	261.07	9.62	47.1	73.7	46	7
togethercomputer/RedPajama-INCITE-7B-Chat	A100	chat	1457.86	27.49	274.39	9.54	42.15	70.84	36.1	7
LMSys/vicuna-7B	A100	chat	1531.7	27.26	284.92	10.3	53.5	77.53	49	7
project-baize/baize-v2-7B	A100	chat	1588.27	31.59	326.3	10.17	48.46	75	41.66	7
LMSys/fastchat-t5-3b-v1.0	A100	chat	1802.98	19.29	314.3	20.68	35.92	46.36	48.79	3
nomic-ai/gpt4all-13b-snoozy	A100	chat	2004.73	26.57	247.8	9.28	56.06	78.69	48.36	13
OpenAssistant/oasst-sft-1-pythia-12b	A100	chat	2060.03	25.4	259.59	9.93	45.56	69.93	39.19	12
BAIR/koala-13b	A100	chat	2144.83	21.21	265.57	12.14	52.9	77.54	50.09	13
openaccess-ai-collective/manticore-13b-ch	A100	chat	2189.25	26.28	288.82	10.95	58.7	81.96	48.86	13
Camel-AI/CAMEL-13B-Combined-Data	A100	chat	2526.46	24.5	291.92	13.17	55.55	79.3	47.33	13
LMSys/vicuna-13B	A100	chat	2600.84	21.61	280.74	12.74	52.9	80.12	51.82	13

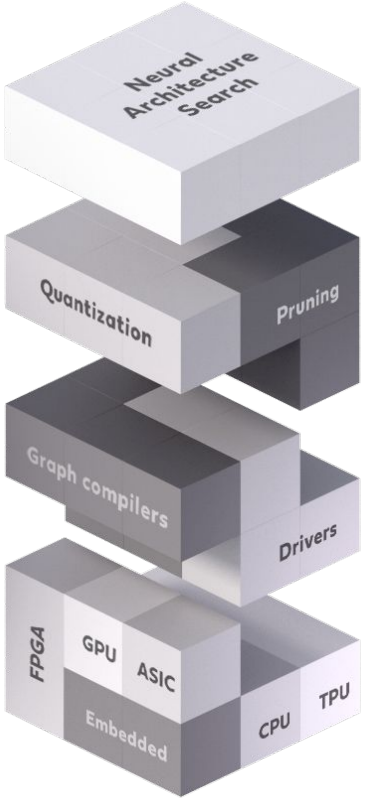
How much energy consumes LLama-7B in terms of 60W a light bulb:

- 3 seconds of work
- 5 seconds of work
- 60 seconds on work

Brain consumes 20 joules energy per second.

source: ml.energy

Software meets Hardware



HARDWARE	SOFTWARE	ALGORITHMIC
• GPU • TPU • CPU • FPGA • Ascend • etc.	• CUDA • Graph Compilers • MKL - DNN • etc.	• PRUNING • QUANTIZATION • NAS • etc.

A majority of machine learning architectures lack explicit consideration for software or hardware properties during development, resulting in computational inefficiency.

Furthermore, many models concentrate exclusively on achieving optimal results rather than accounting for practical implementation issues.

Image source: deci.ai

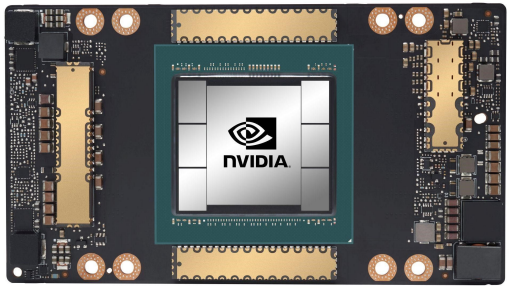
Support of low precision computations in GPU

- Float16
- Float32
- Bfloat16

Will be discussed in future series …

Supported by hardware and software for most modern GPU

Not supported by hardware or software



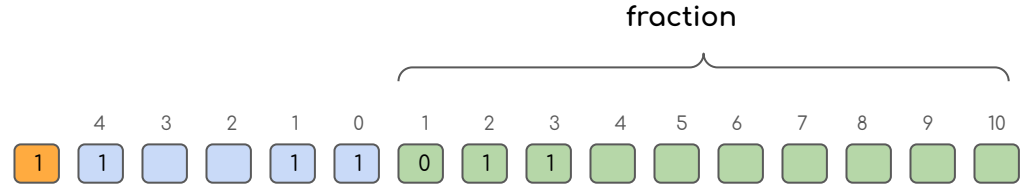
Supported by some hardware only

>>> Data formats

Int16/Int8/Int4

Range of values: $[-2^{n-1}, 2^{n-1} - 1]$

Float32 and Float16
(Floating Point)



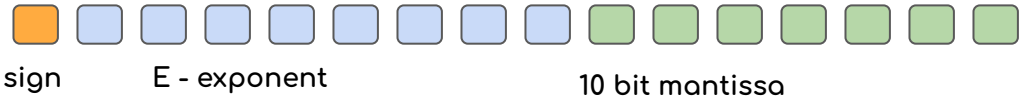
$$\mathbf{v} = (-1)^1(1+f)2^{e-bias} \quad bias = 15$$

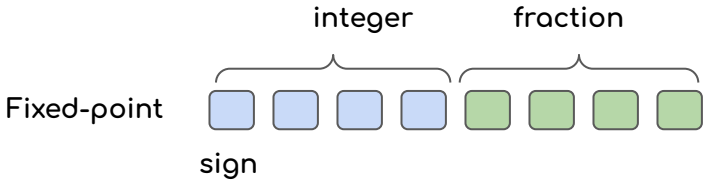
$$f = 2^{-2} + 2^{-3} = \frac{1}{4} + \frac{1}{8} = 0.375$$

$$e = 2^0 + 2^1 + 2^4 = 19$$

$$\mathbf{v} = (-1)^1(1+0.375)2^{19-15} = 22$$

Bfloat16





MORE:

TensorFloat-32 19 bit (supported in A100) : 1 bit sign, 8 bit exponent, 10 bit —mantissa.

E4M3 (8bit) 1 bit sign, 4 bit exponent, 3 bit mantissa
E5M2 (8bit) 1 bit sign, 5 bit exponent, 2 bit mantissa
E2M1 (4bit) 1 bit sign, 2 bit exponent, 1 bit mantissa
E3M0 (4bit) 1 bit sign, 3 bit exponent

So, Float8 and int8 or Float4 and int4 use the same number of bits.
Why do we care about int4 or int8 then?

E4M3 (8bit) 1 bit sign, 4 bit exponent, 3 bit mantissa
E5M2 (8bit) 1 bit sign, 5 bit exponent, 2 bit mantissa
E2M1 (4bit) 1 bit sign, 2 bit exponent, 1 bit mantissa
E3M0 (4bit) 1 bit sign, 3 bit exponent

Int16/Int8/Int4

sign 3 bits

Range of values: $[-2^{n-1}, 2^{n-1} - 1]$

Ресурсы

Как устроены процессоры?

Никто не расскажет, потому что это коммерческая тайна.

Никто не расскажет, потому что никто не знает устройства полностью.

Описание архитектуры Hopper — 71 страница.

Описание низкоуровневого набора команд PTX для ускорителей Nvidia — 598 страниц.

80 миллиардов транзисторов в H100.

Попробуем разобраться

- GPU состоит из нескольких потоковых мультипроцессоров (**Streaming Multiprocessor - SM**), каждый из которых имеет несколько вычислительных ядер.
- Есть г**лобальная память вне чипа**, которая представлена HBM или DRAM. Она расположена далеко от SM на чипе и **имеет большую задержку.**
- Есть кэш **L2 вне чипа** и кэш **L1 на чипе**.
- В каждом **SM есть небольшое количество настраиваемой разделяемой памяти (shared memory)**. Она общая между ядрами. Обычно потоки внутри блока загружают часть данных в **shared memory** и многократно используют её, чтобы не загружать данные снова из глобальной памяти **(global memory)**.
- **Каждый SM имеет большое количество регистров**, которые распределяются между потоками в зависимости от их потребностей. **Например, Nvidia H100 имеет 65 536 регистров на SM.**
- **Чтобы запустить ядро на GPU, мы создаем сетку потоков - тредов.** Сетка состоит из одного или нескольких блоков потоков.
- **GPU назначает один или несколько блоков на выполнение в SM в зависимости от доступных ресурсов. Все потоки одного блока назначаются и выполняются на одном и том же SM.** Это делается для эффективного использования локальности данных и синхронизации между потоками.
- **Потоки, назначенные на SM, дополнительно группируются по 32, эта группа называется warp.** Все потоки внутри warp-а выполняют одну и ту же инструкцию одновременно, но на разных частях данных (SIMT). (Хотя в новых поколениях GPU также поддерживается независимое планирование потоков.)

Ну вот как бы и все ...

Jacquard Loom -> Threads and Warps

Buzzwords:

- SM - Streaming Multiprocessor
- Shared Memory
- Threads - потоки
- Блоки
- Global Memory
- Регистры
- Warps
- SIMT - Single instruction, multiple threads
- L1, L2 caches

Latency optimized

CPU может сложить два числа гораздо быстрее, чем GPU.

В последовательном выполнении нескольких таких операций CPU действительно окажется быстрее.

Throughput optimized

Однако, когда речь идёт о миллионах или миллиардах подобных вычислений, GPU справляется с ними гораздо быстрее, чем CPU, благодаря своему огромному параллелизму.

SIMD - Single Instruction Multiple Data

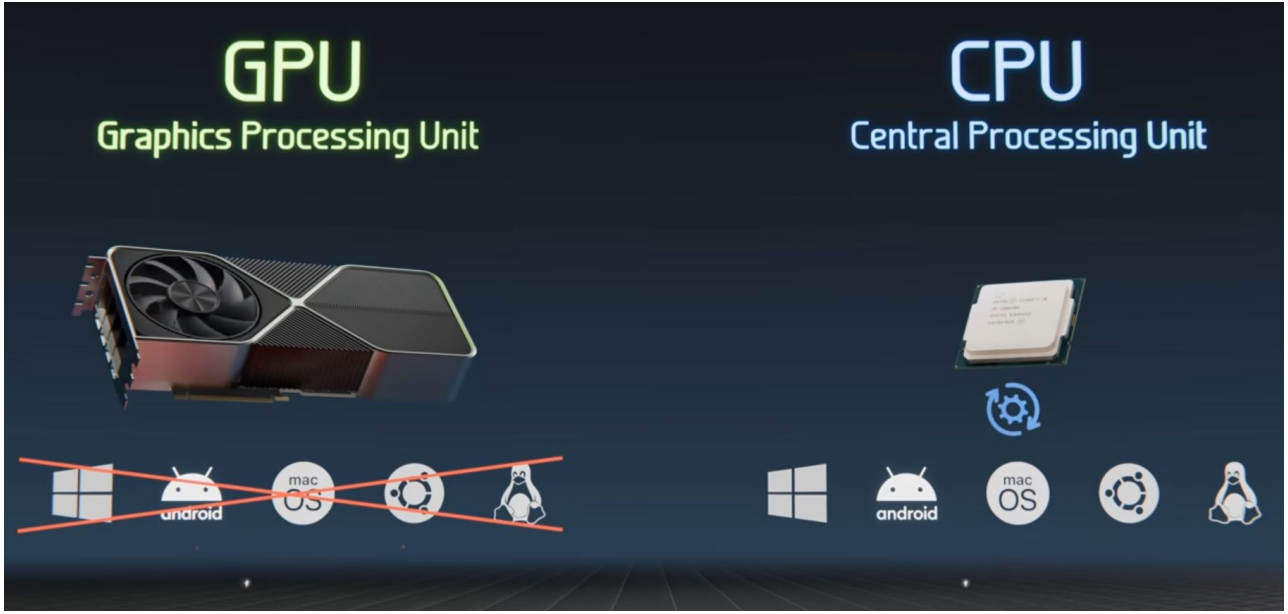
StMp - Single Truck Multiple Products

FLOPS - Число операций с плавающей точкой в секунду. (Почему не MAC ???)

Nvidia Ampere A100 пропускная способность 19,5 TFLOPS при 32-битной точности.

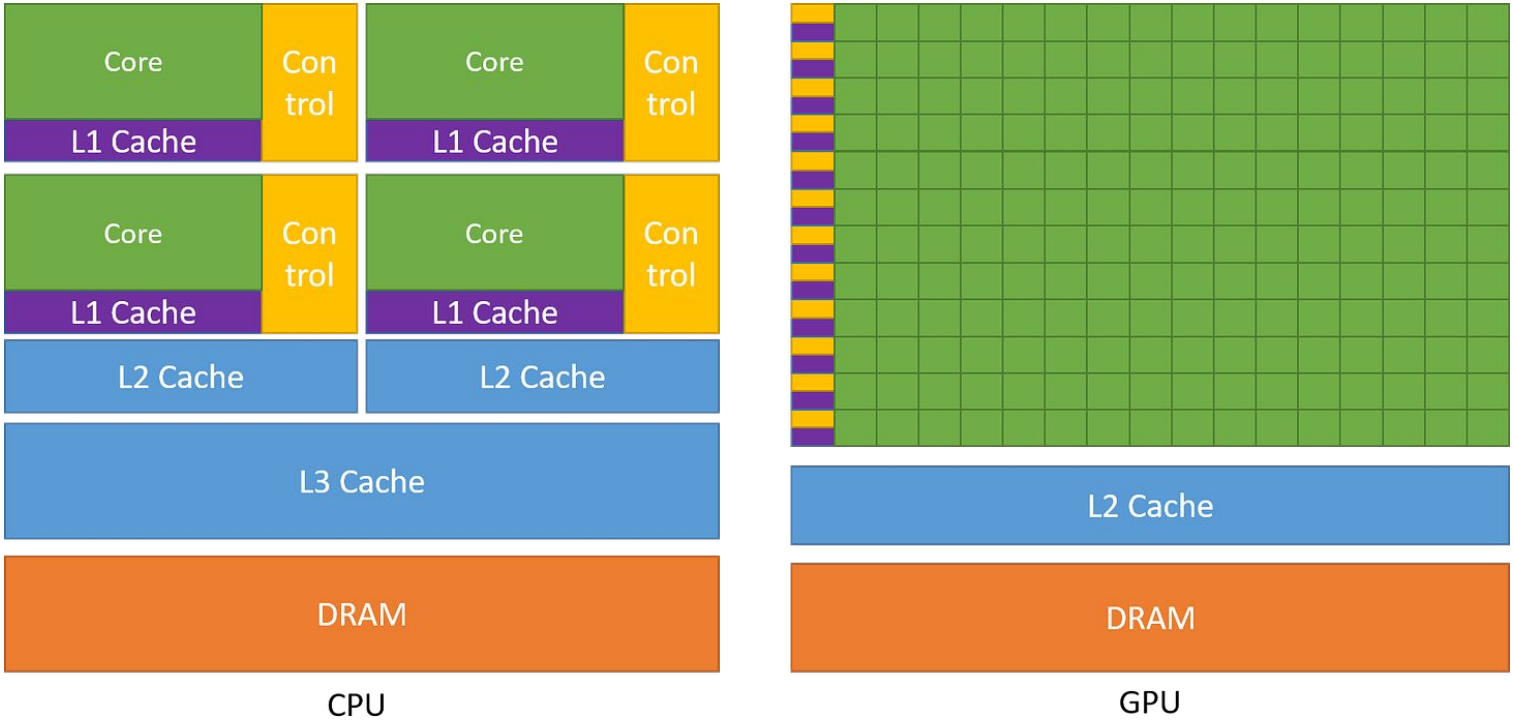
Процессор Intel 0,66 24-ядерна - 0,66 TFLOPS при той же 32-битной точности (данные за 2021 год).

source


GPU
Graphics Processing Unit
CPU
Central Processing Unit
android
mac
OS
android
mac
OS

source

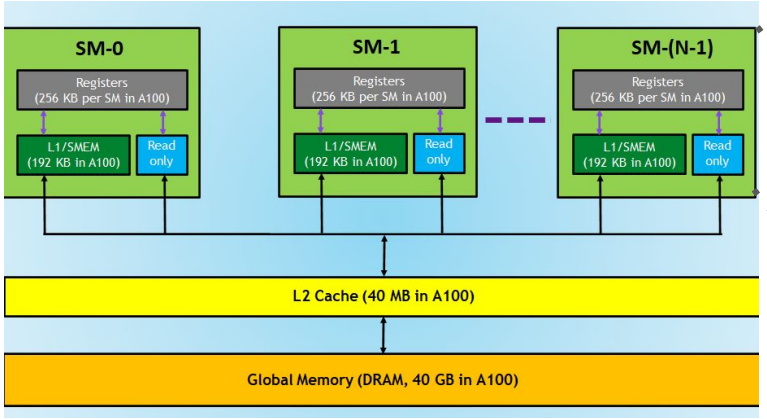
Архитектура GPU

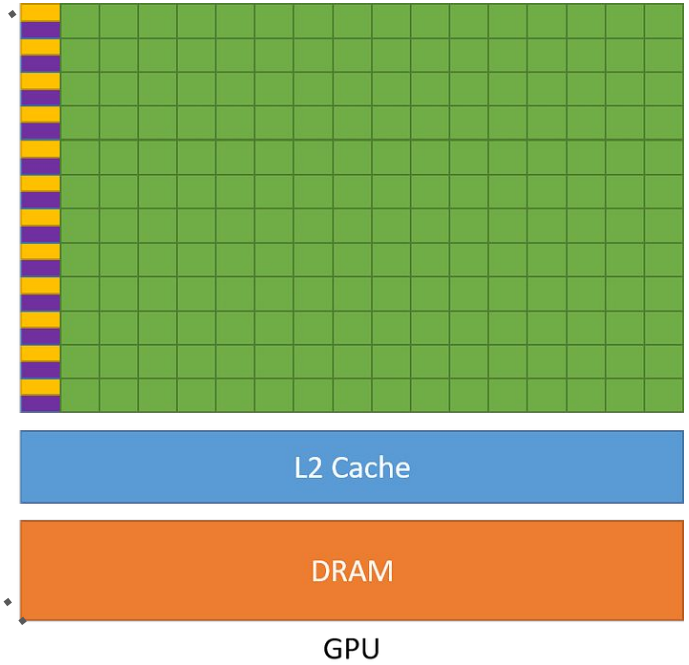


CPU: большие кэши и большое количество блоков управления.

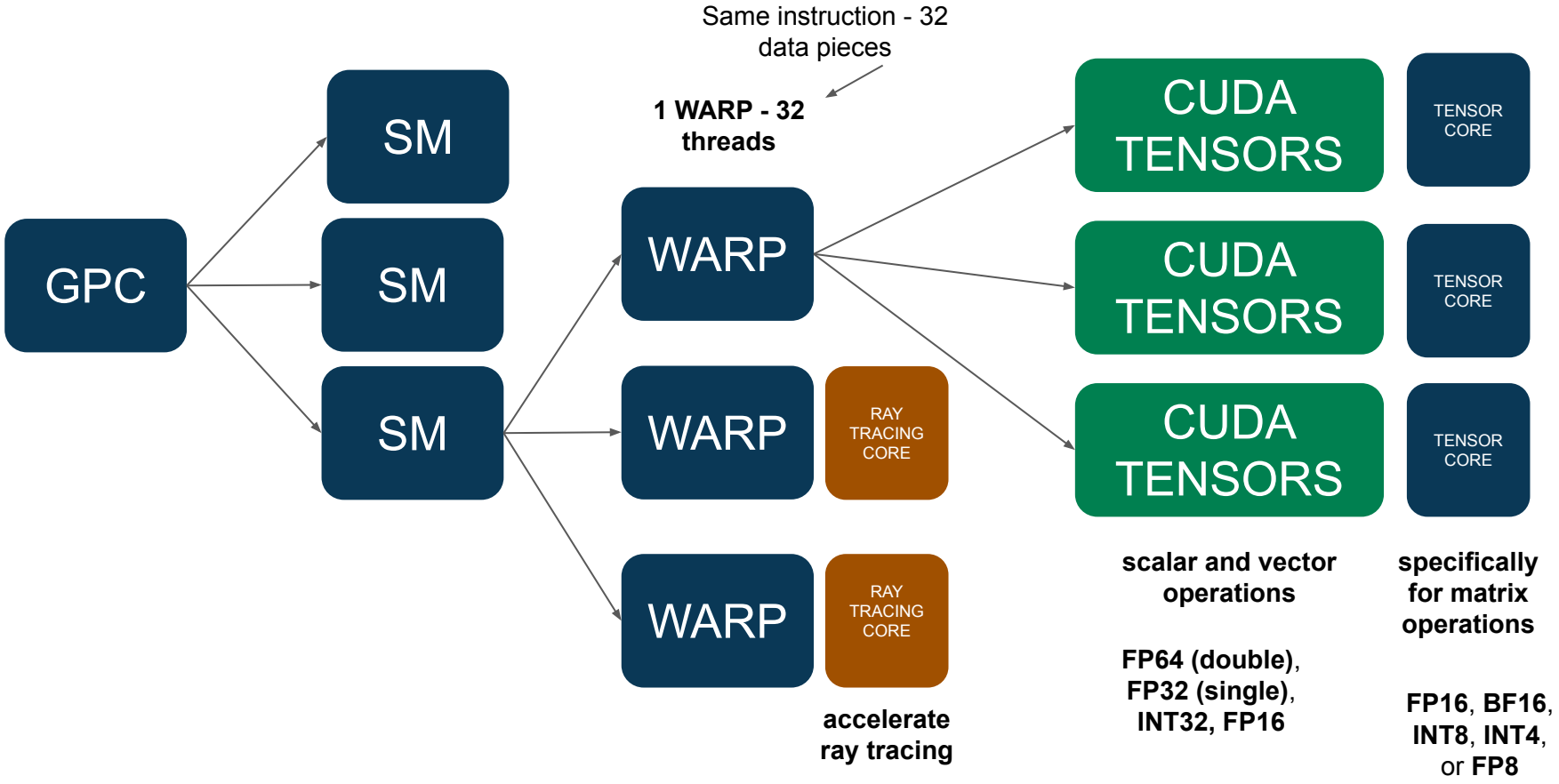
GPU: большое количество ALU для максимизации своей вычислительной мощности и пропускной способности. Мало кэшей и блоков управления.

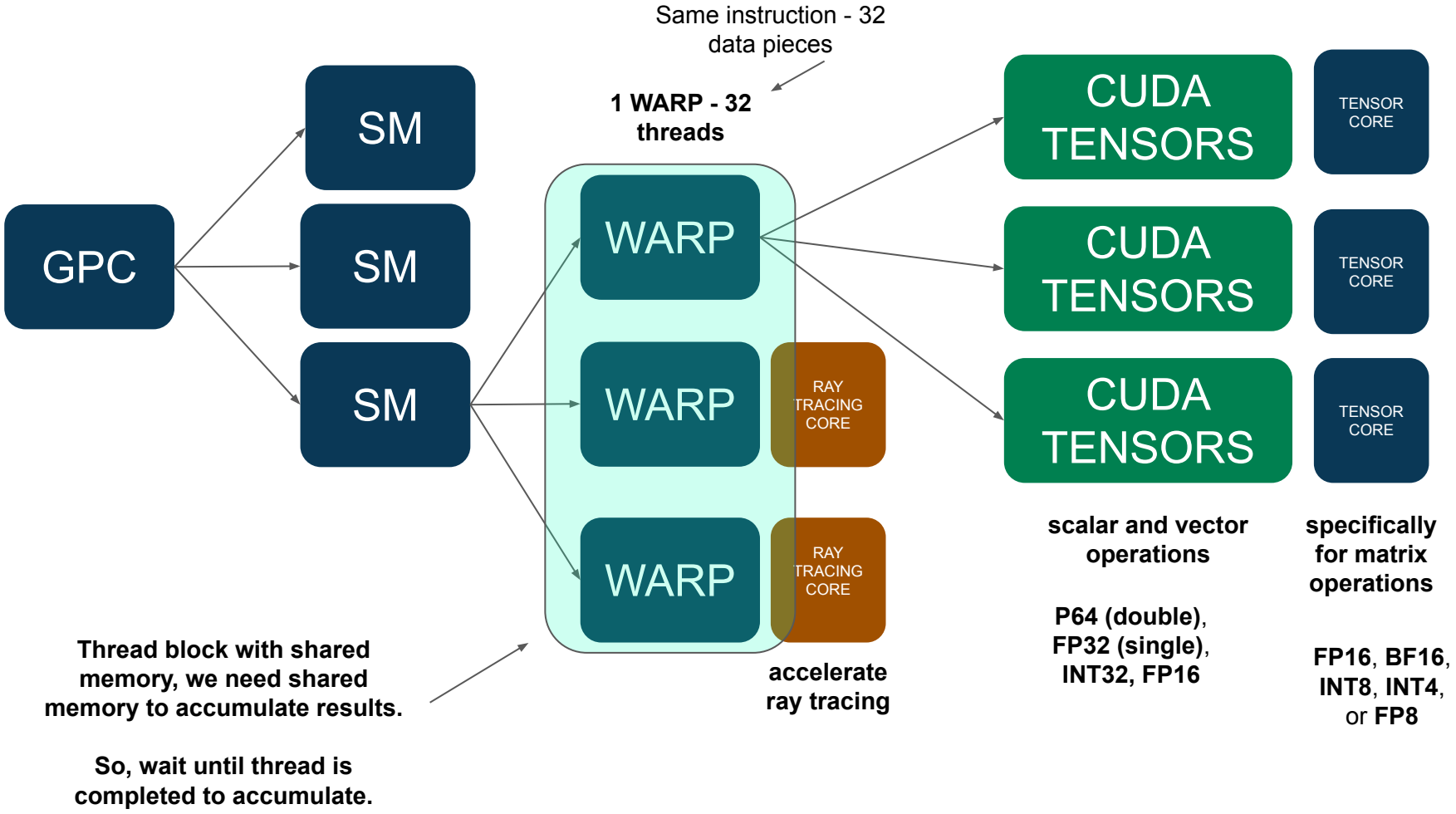
Архитектура GPU





- GPU состоит из массива потоковых мультипроцессоров - Streaming Multiprocessor (SM).
- Каждый SM, в свою очередь, включает несколько потоковых процессоров, также называемых ядрами или потоками. Например, GPU Nvidia H100 содержит 132 SM с 64 ядрами на каждый SM, что в сумме даёт 8448 ядер.
- Каждый SM имеет ограниченное количество встроенной памяти, часто называемой общей памятью, которая используется всеми ядрами совместно.
- Ресурсы блока управления на SM разделяются между всеми ядрами.
- Каждый SM оснащен аппаратными планировщиками потоков для выполнения потоков (тредов).
- Помимо этого, каждый SM содержит несколько функциональных блоков или других ускоренных вычислительных устройств.


Same instruction - 32 data pieces
1 WARP - 32 threads
GPC
SM
SM
SM
WARP
WARP
WARP
RAY TRACING CORE
RAY TRACING CORE
CUDA TENSORS
CUDA TENSORS
CUDA TENSORS
TENSOR CORE
TENSOR CORE
TENSOR CORE
accelerate ray tracing
scalar and vector operations
FP64 (double), FP32 (single), INT32, FP16
specifically for matrix operations
FP16, BF16, INT8, INT4, or FP8


Same instruction - 32 data pieces
1 WARP - 32 threads
GPC
SM
SM
SM
WARP
WARP
WARP
RAY TRACING CORE
RAY TRACING CORE
CUDA TENSORS
CUDA TENSORS
CUDA TENSORS
TENSOR CORE
TENSOR CORE
TENSOR CORE
scalar and vector operations
P64 (double), FP32 (single), INT32, FP16
specifically for matrix operations
FP16, BF16, INT8, INT4, or FP8
accelerate ray tracing
Thread block with shared memory, we need shared memory to accumulate results.
So, wait until thread is completed to accumulate.

SIMD - Single Instruction Multiple Data

SIMT - Single Instruction Multiple Threads!

FLOPS - Число операций с плавающей точкой в секунду. (Почему не MAC ???)

Nvidia Ampere A100 пропускная способность 19,5 TFLOPS при 32-битной точности.

Процессор Intel 0,66 24-ядерна - 0,66 TFLOPS при той же 32-битной точности (данные за 2021 год).

Архитектура GPU

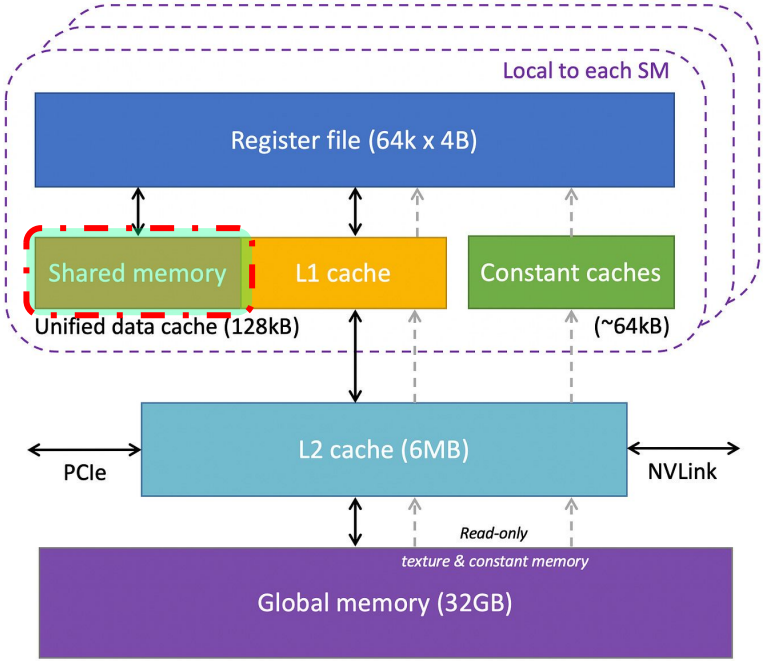
Количество ядер:

SM содержит десятки простых ядер (например, CUDA-ядра в GPU NVIDIA или потоковые процессоры в AMD), но точное количество зависит от архитектуры GPU:

- **Примеры NVIDIA:**
  - Fermi: 32 ядра/SM
  - Ampere GA10x (например, RTX 30-серии): 128 FP32 ядер/SM
  - Ada Lovelace: 128 FP32 ядер/SM
- **Примеры AMD:**
  - RDNA2 Compute Unit (CU): 64 потоковых процессора/CU (аналог SM).

Архитектура GPU

- Общая память (Shared Memory):
  - Программно управляемый кэш, расположенный внутри каждого SM.
  - Быстрая, доступна всем потокам в одном блоке потоков.
  - Используется для обмена данными между потоками внутри блока.
  - Ограниченный размер (например, 64 KB или 96 KB на SM, настраивается в некоторых архитектурах).
  - Работает как scratchpad-память для часто используемых данных.



Регистры

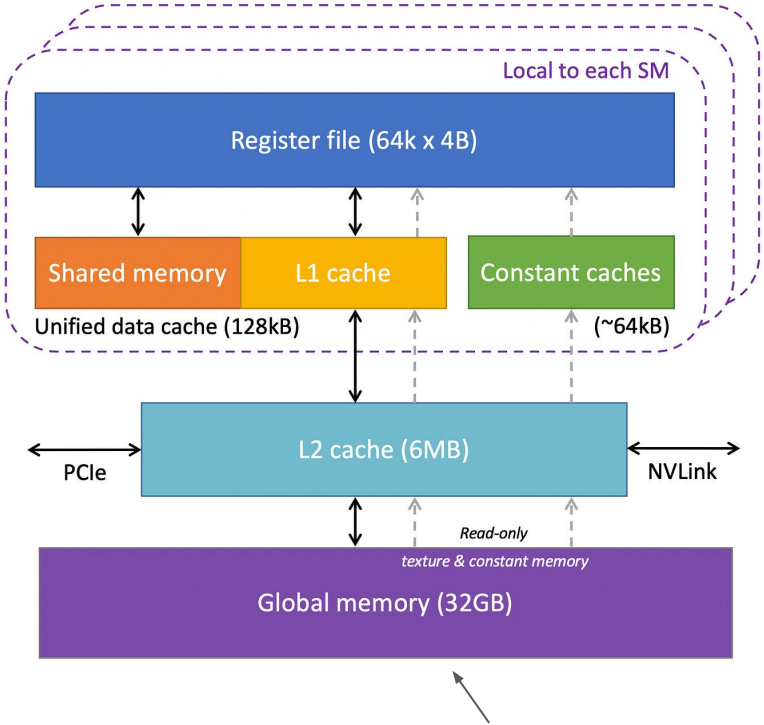
- Самая быстрая память, отдельная для каждого потока.
- Используется для локальных переменных и промежуточных результатов.
- Ограниченное количество (например, 255 регистров/поток в некоторых архитектурах).
- При переполнении данные выгружаются в более медленную память (например, глобальная - DRAM).

Например, модели Nvidia A100 и H100 имеют по 65 536 регистров на один SM.

Глобальная память

- Основная память GPU (DRAM), доступна всем потокам и SM.
- Высокая задержка, но большой объём (например, несколько ГБ).
- Кэшируется кэшем L2 и иногда L1 (зависит от архитектуры).
- Используется для ввода/вывода данных

Например, Nvidia H100 оснащена 80 ГБ памяти с высокой пропускной способностью (HBM) и скоростью передачи данных 3000 ГБ/сек. Из-за значительного расстояния от SM задержка глобальной памяти довольно высока.



Обычно та память про которую мы говорим что ее не хватает.

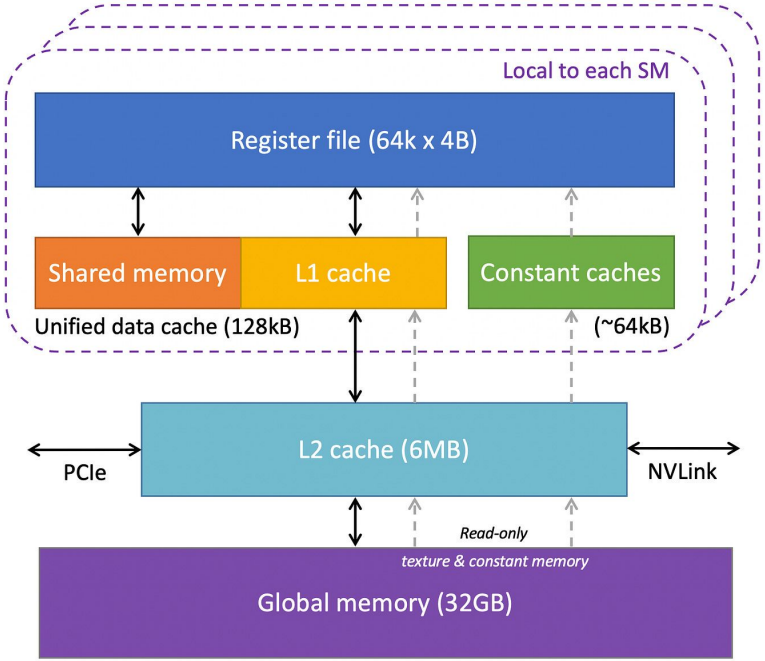
Архитектура GPU

Кэш L1/L2

- Кэш L1: Кэш внутри SM для ускорения доступа к глобальной/локальной памяти.
- Кэш L2: Общий для всех SM, кэширует обращения к глобальной памяти.
- Снижает задержку для часто используемых данных.

Константная память

- Специальная область глобальной памяти только для чтения данных.
- Кэшируется для быстрого доступа при чтении одинаковых значений всеми потоками.
- Используется для констант и параметров ядра.



Выполнение операций на GPU

Shared memory всего 64 kb

Какого размера квадратная матрица туда влезет в FP32, FP16, Int8?

Выполнение операций на GPU

1. For `FP32` (32-bit float, 4 bytes per element)

- **Element size**: 4 bytes.
- **Total elements**:

$$N^2 = \frac{64\ \text{KB}}{4\ \text{bytes}} = \frac{65{,}536}{4} = 16{,}384$$

- **Matrix size**:

$$N = \sqrt{16{,}384} = 128$$

Result: **128×128** matrix.

2. For `FP16` (16-bit float, 2 bytes per element)

- **Element size**: 2 bytes.
- **Total elements**:

$$N^2 = \frac{65{,}536}{2} = 32{,}768$$

- **Matrix size**:

$$N = \sqrt{32{,}768} \approx 181.02 \quad \Rightarrow \quad \text{Floor to } 181$$

Result: **181×181** matrix.

  - Uses $181^2 \times 2 = 65{,}522$ bytes (leaves 14 bytes unused).

3. For `INT8` (8-bit integer, 1 byte per element)

- **Element size**: 1 byte.
- **Total elements**:

$$N^2 = \frac{65{,}536}{1} = 65{,}536$$

- **Matrix size**:

$$N = \sqrt{65{,}536} = 256$$

Result: **256×256** matrix.

Выполнение операций на GPU

Data Type	Matrix Size	Memory Used	Unused Bytes
FP32	128 x 128	65,535	0
FP16	181 x 181	65,522	14
INT8	256 x 256	65,535	0

Выполнение операций на GPU

Наши матрицы сильно больше, как
их перемножать?

Tiling

Shared Memory small but fast Global memory large but slow

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 (2,3)	B (3,3)
B (3,0)	B (3,1)	B (3,2)	B (3,3)

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 (2,3)	A (3,3)
A (3,0)	A (3,1)	A (3,2)	A (3,3)

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 (2,3)	C (3,3)
C (3,0)	C (3,1)	C (3,2)	C (3,3)

Thread [0,0]	Thread [0,1]	Thread [1,0]	Thread [1,1]
A (0,0)	A (0,0)		
A (0,1)	A (0,1)		
A (0,2)	A (0,2)		
A (0,3)	A (0,3)		
B (0,0)	B (0,1)		
B (1,0)	B (1,1)		
B (2,0)	B (2,1)		
B (3,0)	B (3,1)		

Tiling

Thread [0,0] Thread [0,1]

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 (2,3)	B (3,3)
B (3,0)	B (3,1)	B (3,2)	B (3,3)

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 (2,3)	A (3,3)
A (3,0)	A (3,1)	A (3,2)	A (3,3)

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 (2,3)	C (3,3)
C (3,0)	C (3,1)	C (3,2)	C (3,3)

Shared Memory small but fast Global memory large but slow

Thread [0,0]	Thread [0,1]	Thread [1,0]	Thread [1,1]
A (0,0)	A (0,0)		
A (0,1)	A (0,1)		
A (0,2)	A (0,2)		
A (0,3)	A (0,3)		
B (0,0)	B (0,1)		
B (1,0)	B (1,1)		
B (2,0)	B (2,1)		
B (3,0)	B (3,1)		

Tiling

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 (2,3)	B (3,3)
B (3,0)	B (3,1)	B (3,2)	B (3,3)

Tiles fit shared memory

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 (2,3)	A (3,3)
A (3,0)	A (3,1)	A (3,2)	A (3,3)

- Each thread in a block has unique shared memory
- Global memory can be accessed by all blocks

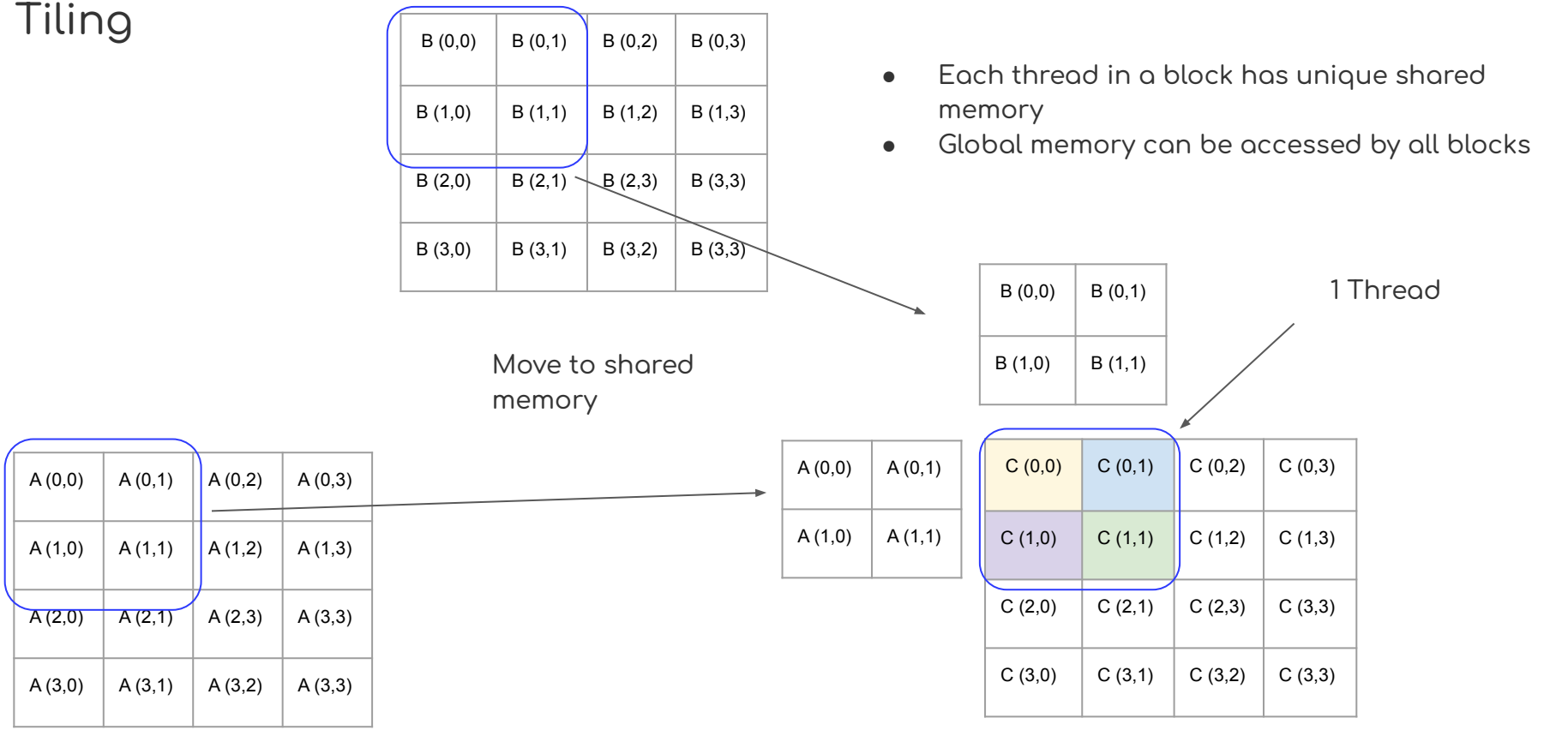
Shared memory

1 Thread

Shared memory

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 (2,3)	C (3,3)
C (3,0)	C (3,1)	C (3,2)	C (3,3)

Tiling



- Each thread in a block has unique shared memory
- Global memory can be accessed by all blocks

Tiling

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 (2,3)	B (3,3)
B (3,0)	B (3,1)	B (3,2)	B (3,3)

- Each thread in a block has unique shared memory
- Global memory can be accessed by all blocks

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 (2,3)	A (3,3)
A (3,0)	A (3,1)	A (3,2)	A (3,3)

Partial dot product

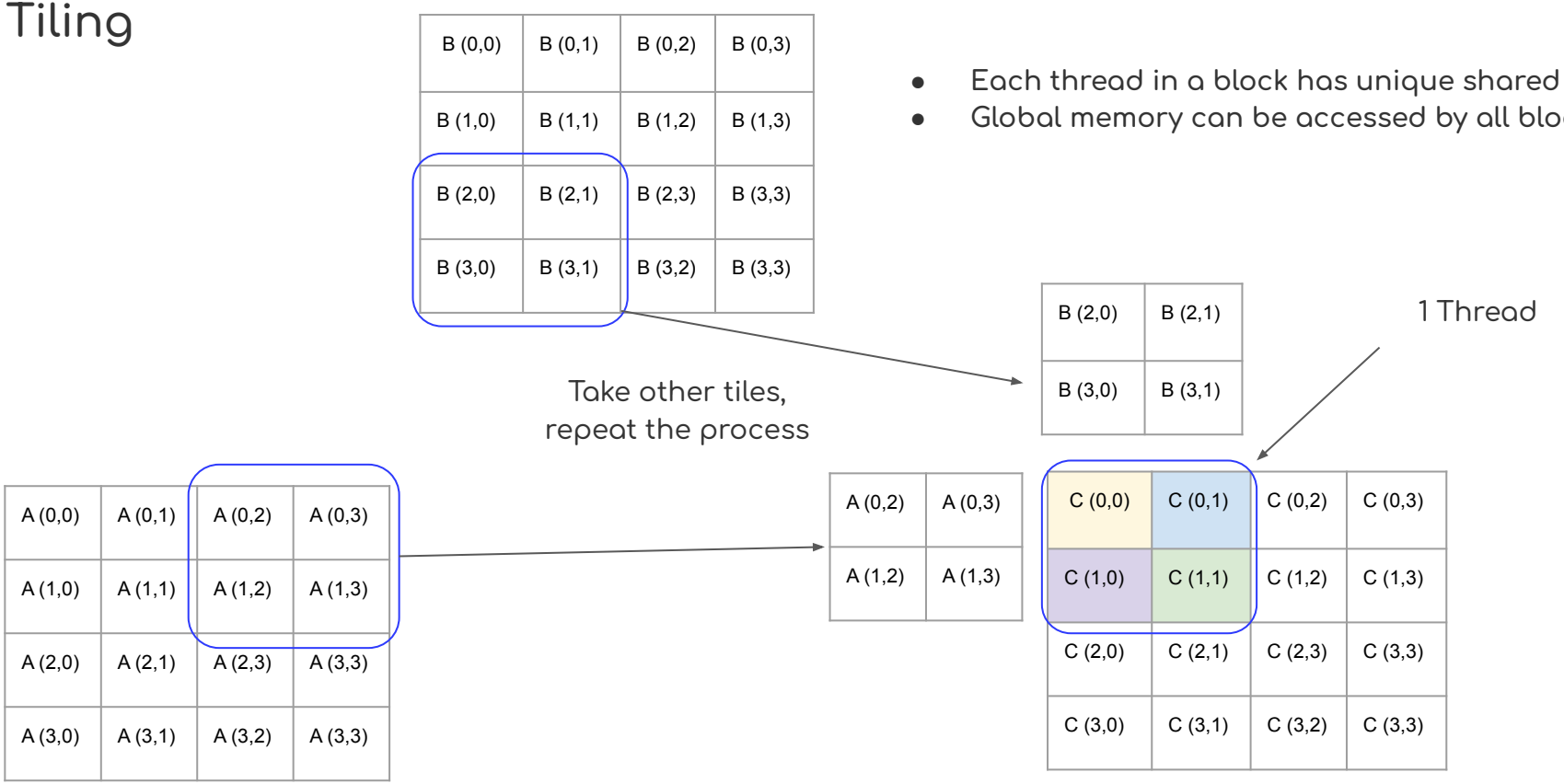
B (0,0)	B (0,1)
B (1,0)	B (1,1)

1 Thread

A (0,0)	A (0,1)
A (1,0)	A (1,1)

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 (2,3)	C (3,3)
C (3,0)	C (3,1)	C (3,2)	C (3,3)

Tiling



- Each thread in a block has unique shared memory
- Global memory can be accessed by all blocks

Tiling

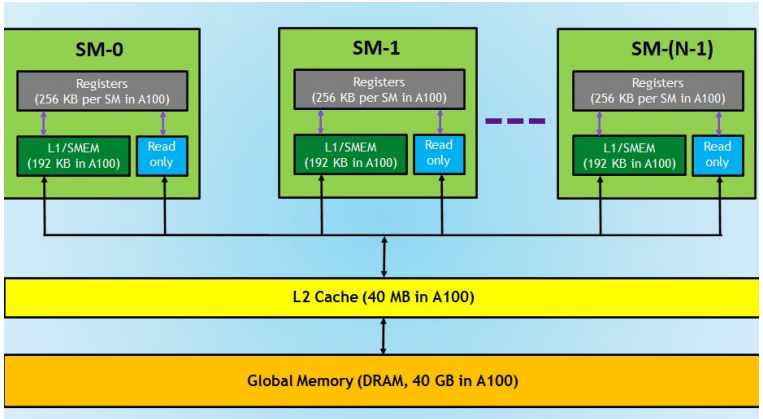
Shared Memory small but fast Global memory large but slow

Thread [0,0]	Thread [0,1]	Thread [1,0]	Thread [1,1]
A (0,0)	A (0,0)		
A (0,1)	A (0,1)		
A (0,2)	A (0,2)		
A (0,3)	A (0,3)		
B (0,0)	B (0,1)		
B (1,0)	B (1,1)		
B (2,0)	B (2,1)		
B (3,0)	B (3,1)		

Thread [0,0]	Thread [0,1]	Thread [1,0]	Thread [1,1]
A (0,0)	A (0,1)		
B (0,0)	B (0,1)		
A (0,2)	A (0,3)		
B (2,0)	B (2,1)		

WARPS, Blocks, etc

Выполнение операций на GPU



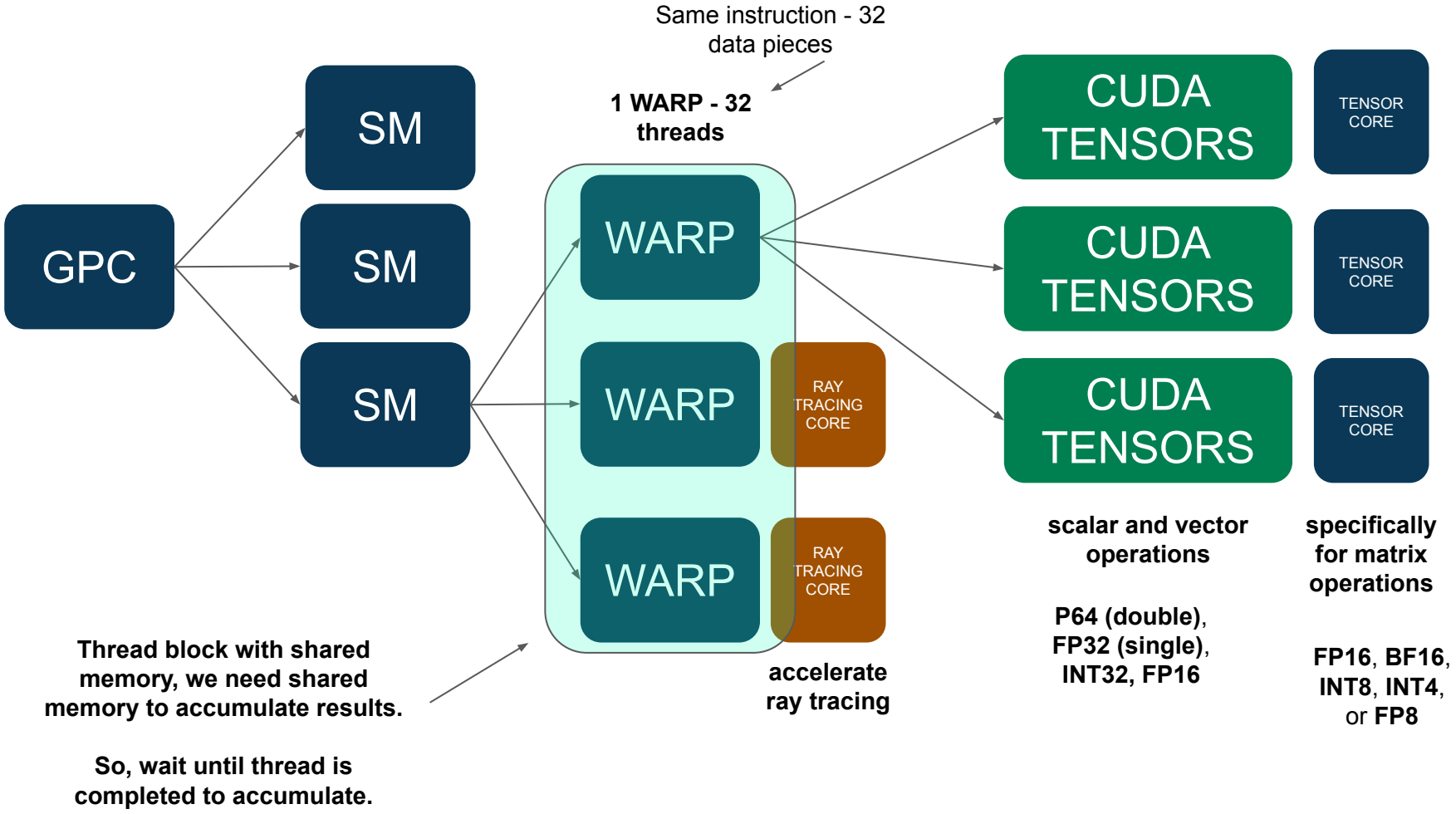
Все потоки одного блока назначаются на один и тот же мультипроцессор (SM).

Однако после этого происходит ещё одно деление потоков.

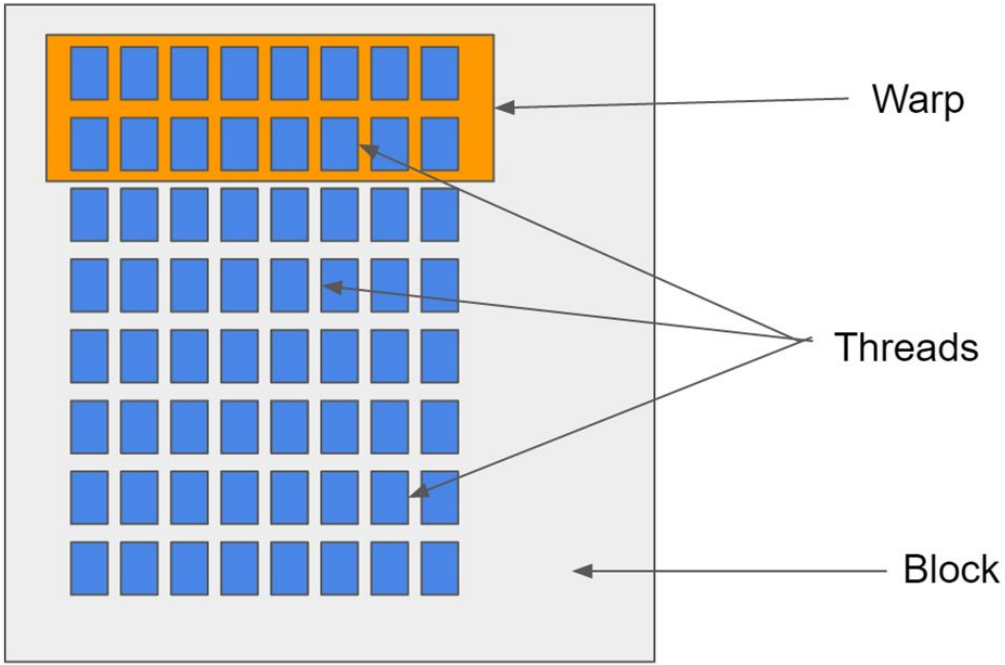
Эти потоки группируются в наборы по 32 потока (warp), которые назначаются для совместного выполнения на наборе ядер, называемом вычислительным блоком.

SM выполняет все потоки в одном warp-е одновременно, извлекая и отправляя одну и ту же инструкцию для всех потоков. Эти потоки затем выполняют эту инструкцию параллельно, но на разных участках данных. В примере со сложением векторов все потоки в warp-е могут выполнять инструкцию сложения, но каждый поток будет работать с разными индексами векторов.

Эта модель выполнения warp-а также называется **SIMT (single instruction multiple threads)** — то есть несколько потоков выполняют одну и ту же инструкцию. Это похоже на SIMD (single instruction multiple data — одиночный поток команд, множественный поток данных) в процессорах.


Same instruction - 32 data pieces
1 WARP - 32 threads
GPC
SM
SM
SM
WARP
WARP
WARP
RAY TRACING CORE
RAY TRACING CORE
CUDA TENSORS
CUDA TENSORS
CUDA TENSORS
TENSOR CORE
TENSOR CORE
TENSOR CORE
scalar and vector operations
P64 (double), FP32 (single), INT32, FP16
specifically for matrix operations
FP16, BF16, INT8, INT4, or FP8
accelerate ray tracing
Thread block with shared memory, we need shared memory to accumulate results.
So, wait until thread is completed to accumulate.

Выполнение операций на GPU



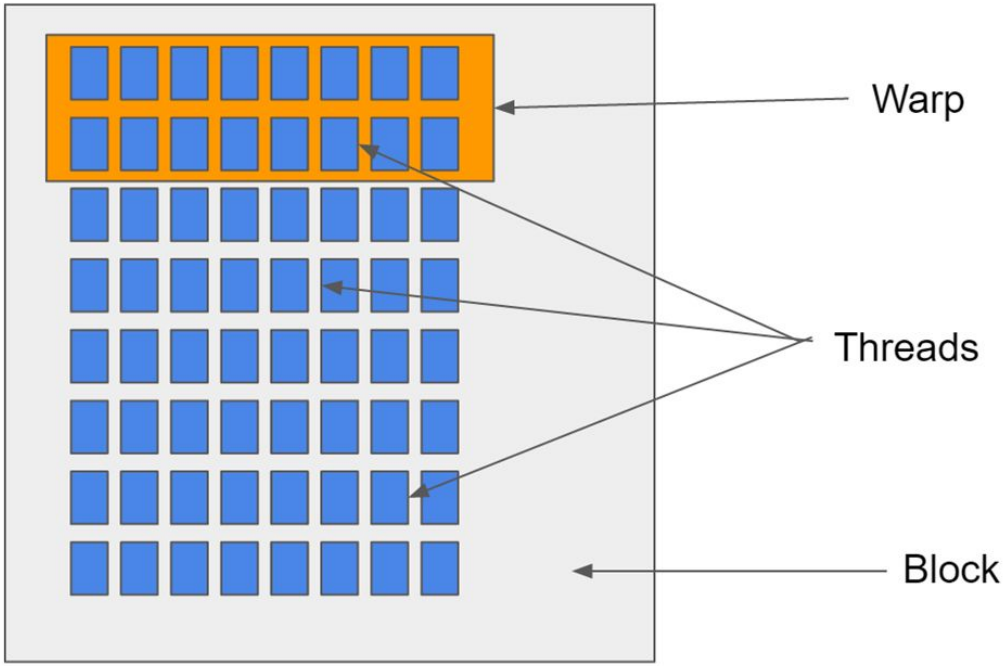
Треды внутри блока физически организованы в варпы (warps).

Треды в одном варпе исполняются вместе и «ждут» друг друга.

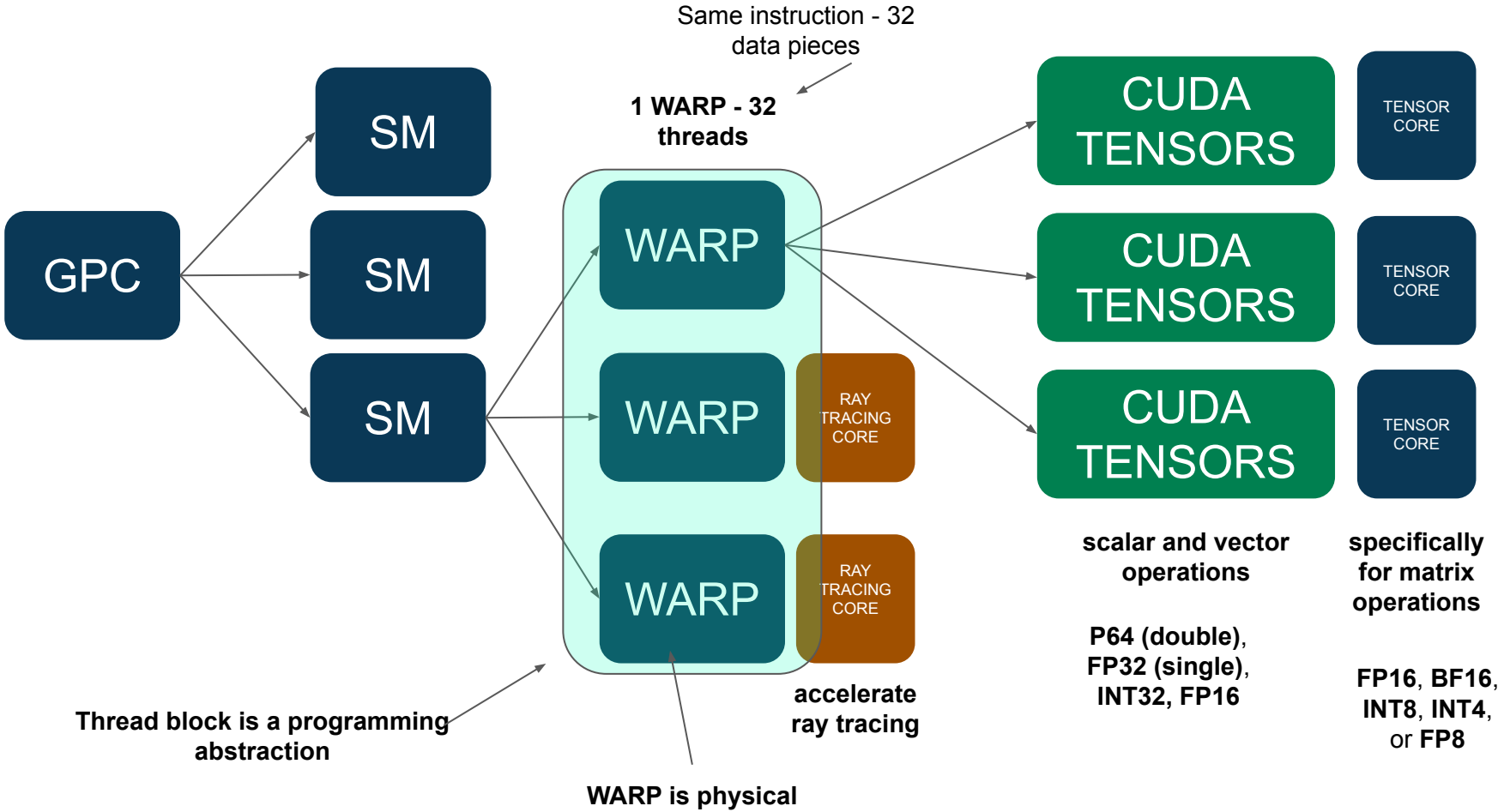
На одном SM может одновременно выполняться несколько варпов (порядка 32 - 64).

Программист может об этом не задумываться без ущерба для корректности (но не для эффективности!)

Выполнение операций на GPU



А зачем варпы?


Same instruction - 32 data pieces
1 WARP - 32 threads
GPC
SM
SM
SM
WARP
WARP
WARP
RAY TRACING CORE
RAY TRACING CORE
CUDA TENSORS
CUDA TENSORS
CUDA TENSORS
TENSOR CORE
TENSOR CORE
TENSOR CORE
scalar and vector operations
P64 (double), FP32 (single), INT32, FP16
specifically for matrix operations
FP16, BF16, INT8, INT4, or FP8
accelerate ray tracing
Thread block is a programming abstraction
WARP is physical

Выполнение операций на GPU

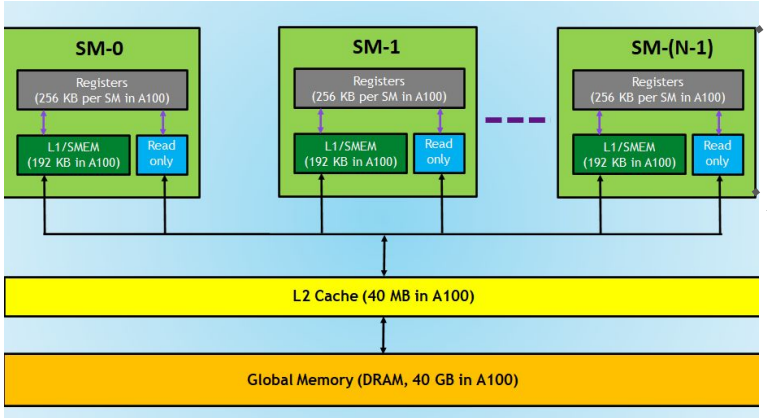
1. Логика управления
   - GPU имеют тысячи потоков, но индивидуальная логика управления для каждого потока была бы слишком сложной и затратной с точки зрения аппаратных ресурсов.
   - Как warp'ы помогают: warp'ы обеспечивают аппаратную простоту: один блок инструкций обслуживает 32 потока, что минимизирует накладные расходы на управление.

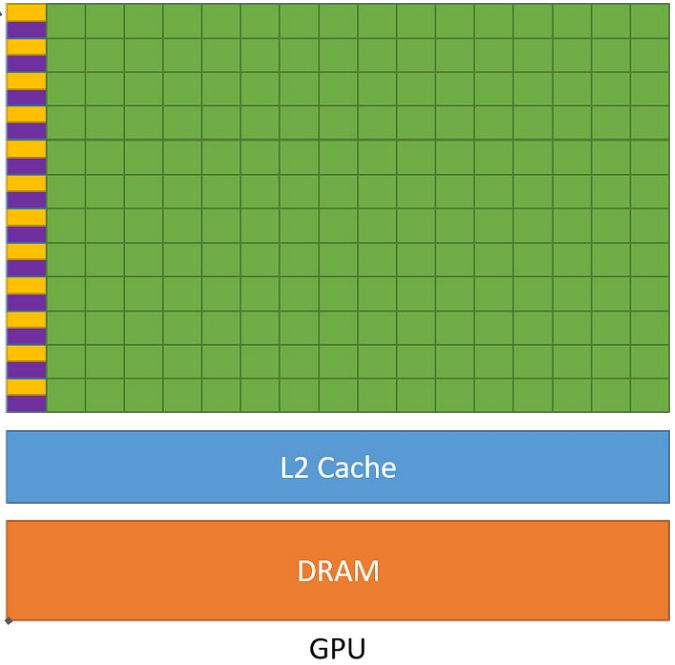
2. Скрытие задержек за счёт переключения
   - Если warp приостанавливается (например, ожидает данные из глобальной памяти), планировщик GPU переключается на другой готовый warp.
   - Почему это важно: GPU не имеют больших кэшей (в отличие от CPU), чтобы скрывать задержки памяти. Переключение на уровне warp'ов позволяет поддерживать занятость исполнительных блоков, максимизируя использование аппаратных ресурсов.

3. Эффективное использование ресурсов
   - Warp'ы совместно используют ресурсы (например, регистры и общую память) внутри одного SM (Streaming Multiprocessor).
   - Почему это важно: Ограниченное количество регистров/SM распределяется между активными warp'ами, а не между отдельными потоками. Без warp'ов поддержка тысяч потоков потребовала бы непрактично большого объёма памяти на чипе.

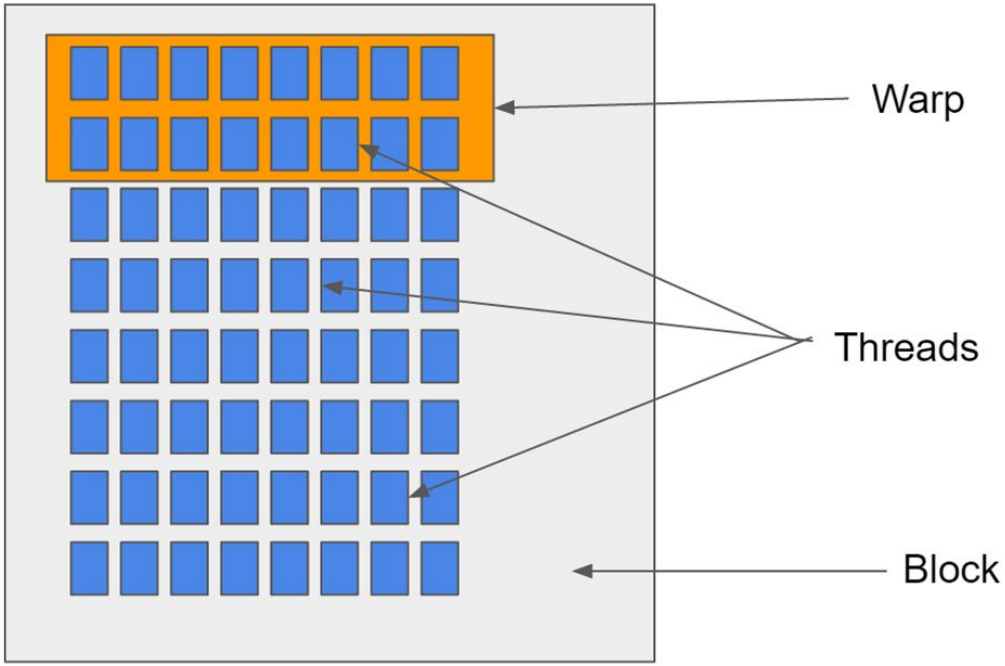
4. Упрощенное планирование
   - Warp'ы обеспечивают пространственную локальность, сокращая потери пропускной способности памяти.
   - Почему это важно: Потоки в warp обращаются к соседним адресам памяти, что позволяет GPU объединять запросы в одну транзакцию. Это снижает нагрузку на шину памяти и повышает эффективность.

Архитектура GPU





Выполнение операций на GPU



А зачем нам Блоки?

Выполнение операций на GPU

	Warp	Block
Инструкция	Одна и таже	
Данные	Warp обрабатывает один tile от матрицы	Разные тайлы
Память	Регистры выделяются на поток внутри варпа	Shared memory распределяется между warps внутри блока

А зачем нам Блоки?

Каждый warp обрабатывает свой тайл матрицы, но инструкции (умножение-накопление) одинаковы для всех.

Выполнение операций на GPU

Примеры.

Неоптимальное использование warps

Проблема: Размер блока не кратен 32 (размеру warp). Например, блок 31x1 потоков.

Почему плохо: GPU всё равно выделяет целый warp (32 потока), и 1 поток остаётся неактивным → **пустая трата ресурсов**.

Выполнение операций на GPU

Примеры.

Плохой выбор размера тайла (tiling)

Проблема:Тайл 8x8 в shared memory при размере warp 32.

Почему плохо: Для загрузки тайла 8x8 требуется 64 потока, но warp содержит 32 потока → потребуется **два warp** на загрузку одного тайла → неэффективно.

Выполнение операций на GPU

Невыровненные размеры матриц

Проблема: Матрица 1000x1000 при тайле 32x32.

Почему плохо: Последние тайлы в строке/столбце будут **неполными** (1000 % 32 = 8 → 8x32 тайл). Многие потоки в warp будут простаивать.

Выполнение операций на GPU

Чрезмерное использование shared memory

Проблема: Тайл 48x48 в shared memory (48x48 * 4 байта = 9 КБ на тайл).

Почему плохо: Если SM поддерживает 96 КБ shared memory, то блоков на SM будет: 96 / 9 ≈ 10. Но каждый блок требует и другие ресурсы → **недоиспользование ядер**.

Выполнение операций на GPU - SUMMARY

1. Запуск ядра и организация потоков

2. Выделение памяти на GPU

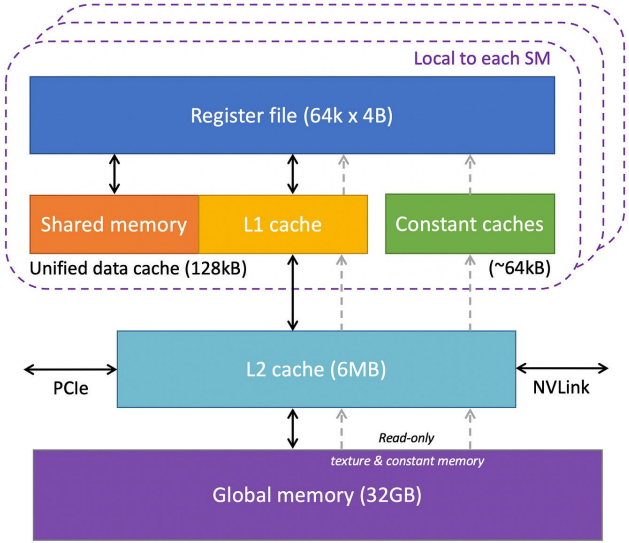
3. Тайлинг: Загрузка тайлов в общую память

4. Синхронизация потоков

5. Вычисление частичных скалярных произведений

6. Повторение тайлинга и накопление

7. Завершение вычислений и запись в глобальную память

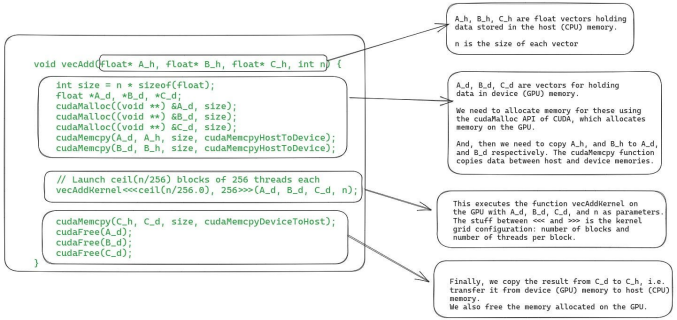


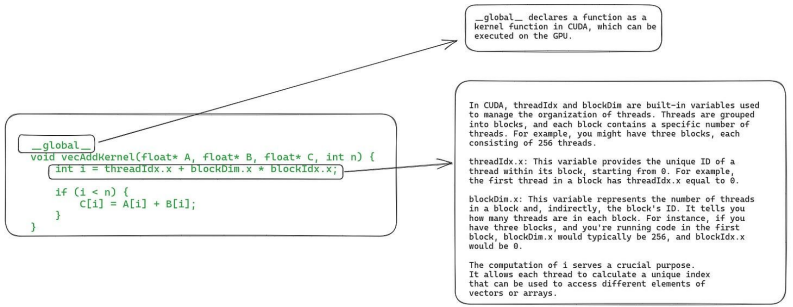
Выполнение операций на GPU - SUMMARY

1. Запуск ядра и организация потоков

Запуск CUDA/GPU ядра с **сеткой блоков потоков**, где каждый блок вычисляет подматрицу (тайл) результата.

- Каждому блоку потоков назначается вычисление **тайла** выходной матрицы (например, 16x16 или 32x32 элементов).
- Каждый поток в блоке вычисляет **один элемент** (или небольшой набор) выходного тайла.
- Для сопоставления потоков с индексами матрицы используется **2D-сетка и размеры блоков**





Выполнение операций на GPU

```
__global__ void sq_mat_mul_kernel(float* d_A, float* d_B, float*
d_C, int N)
{
    // Identifying the thread mapping
    // Working on C[i,j]
    int i = blockDim.y*blockIdx.y + threadIdx.y;
    int j = blockDim.x*blockIdx.x + threadIdx.x;

    // Check the edge cases
    if (i < N && j < N)
    {
        // Value at C[i,j]
        float value = 0;
        // Loop over elements of the two vectors
        for (int k = 0; k < N; k++)
        {
            // Multiply and add
            value += d_A[i*N+k] * d_B[k*N+j];
        }

        // Assigning calculated value
        d_C[i*N+j] = value;
    }
}
```

- **blockDim**: The dimensions (number of threads) of a thread block.
  - blockDim.x is the number of threads in the x-dimension of the block.
  - blockDim.y is the number of threads in the y-dimension of the block.
- **blockIdx**: The index of the thread block.
  - blockIdx.x is the block's index in the x-dimension of the grid.
  - blockIdx.y is the block's index in the y-dimension of the grid.
- **threadIdx**: The thread's index within its block.
  - threadIdx.x is the thread's index in the x-dimension of the block.
  - threadIdx.y is the thread's index in the y-dimension of the block.

```
C[i][j] = A[i][0]*B[0][j] +
          A[i][1]*B[1][j] +
          A[i][2]*B[2][j] +
          ...
          A[i][N-1]*B[N-1][j]
```

Выполнение операций на GPU

```
__global__ void sq_mat_mul_kernel(float* d_A, float* d_B, float* d_C, int N)
{
    // Identifying the thread mapping
    // Working on C[i,j]
    int i = blockDim.y*blockIdx.y + threadIdx.y;
    int j = blockDim.x*blockIdx.x + threadIdx.x;

    // Check the edge cases
    if (i < N && j < N)
    {
        // Value at C[i,j]
        float value = 0;
        // Loop over elements of the two vectors
        for (int k = 0; k < N; k++)
        {
            // Multiply and add
            value += A[i*N+k] * B[k*N+j];
        }

        // Assigning calculated value
        C[i*N+j] = value;
    }
}
```

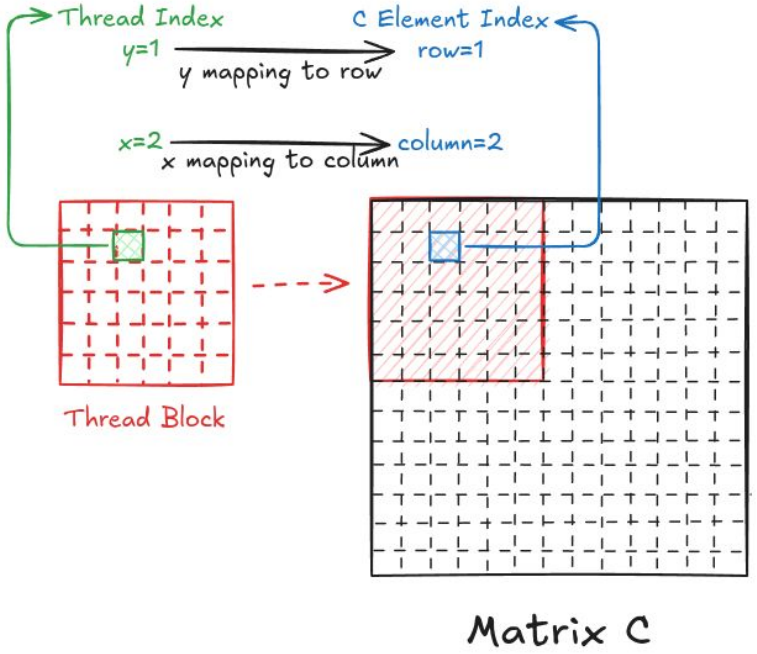
Example:

- N=4, and the grid/block configuration is as follows:
- blockDim.x = 2, blockDim.y = 2 (2x2 threads per block).
- gridDim.x = 2, gridDim.y = 2 (2x2 blocks in the grid).
- This setup creates 4×4=16 threads, one for each element of the 4×4 output matrix d_C.
- Each thread computes one element of d_C based on its global indices (i,j).

source

Выполнение операций на GPU

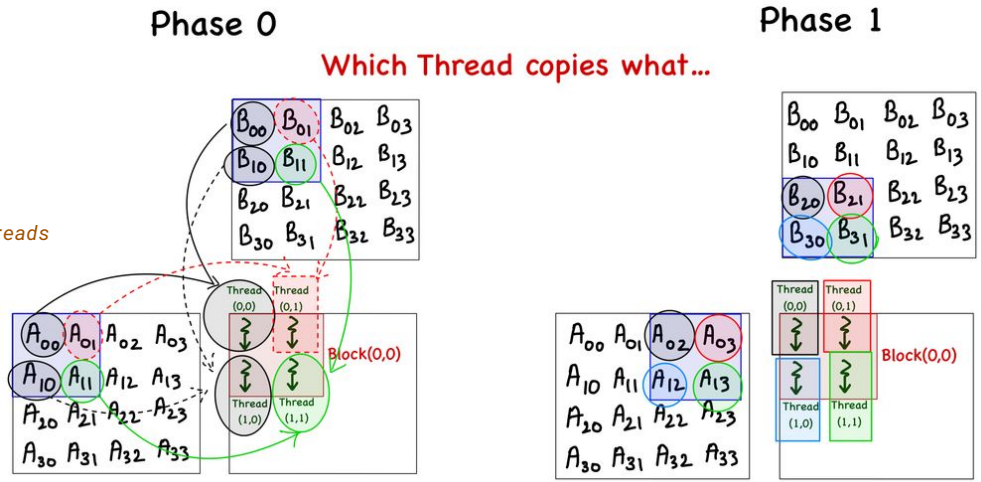
```
__global__ void sq_mat_mul_kernel(float* d_A, float* d_B, float* d_C, int
N)
{
    // Identifying the thread mapping
    // Working on C[i,j]
    int i = blockDim.y*blockIdx.y + threadIdx.y;
    int j = blockDim.x*blockIdx.x + threadIdx.x;

    // Check the edge cases
    if (i < N && j < N)
    {
        // Value at C[i,j]
        float value = 0;
        // Loop over elements of the two vectors
        for (int k = 0; k < N; k++)
        {
            // Multiply and add
            value += A[i*N+k] * B[k*N+j];
        }

        // Assigning calculated value
        C[i*N+j] = value;
    }
}
```



source

Выполнение операций на GPU

```cpp
__global__ void tiled_sq_mat_mul_kernel(float* A, float* B, float* C, int N)
{
    // Ensure that TILE_WIDTH = BLOCK_SIZE
    assert(TILE_WIDTH == blockDim.x);
    assert(TILE_WIDTH == blockDim.y);

    // Ensure N%TILE_WIDTH == 0
    assert(N % TILE_WIDTH == 0);

    // Details regarding this thread
    int by = blockIdx.y;
    int bx = blockIdx.x;

    int ty = threadIdx.y;
    int tx = threadIdx.x;

    // Working on C[i,j]
    int i = TILE_WIDTH*by + ty;
    int j = TILE_WIDTH*bx + tx;

    // Allocating shared memory
    // automatic variables are stored in registers that are private to threads
    __shared__ float sh_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float sh_B[TILE_WIDTH][TILE_WIDTH];

    // Parallel mat mul
    float value = 0;
    for (int phase = 0; phase < N/TILE_WIDTH; phase++)
    {
        // Load Tiles into shared memory
        sh_A[ty][tx] = A[(i)*N + phase*TILE_WIDTH+tx];
        sh_B[ty][tx] = B[(phase*TILE_WIDTH + ty)*N+j];
        __syncthreads(); // Make sure all threads completed its work

        // Dot product
        for (int k = 0; k < TILE_WIDTH; k++)
            value += sh_A[ty][k] * sh_B[k][tx];
        __syncthreads();
    }
    // Assigning calculated value
    C[i*N+j] = value;
}
```



! two smaller loops will perform faster than one big

Parallel compute on Tiles level

source

Выполнение операций на GPU

```cpp
__global__ void tiled_sq_mat_mul_kernel(float* A, float* B, float* C, int N)
{
    // Ensure that TILE_WIDTH = BLOCK_SIZE
    assert(TILE_WIDTH == blockDim.x);
    assert(TILE_WIDTH == blockDim.y);

    // Ensure N%TILE_WIDTH == 0
    assert(N % TILE_WIDTH == 0);

    // Details regarding this thread
    int by = blockIdx.y;
    int bx = blockIdx.x;

    int ty = threadIdx.y;
    int tx = threadIdx.x;

    // Working on C[i,j]
    int i = TILE_WIDTH*by + ty;
    int j = TILE_WIDTH*bx + tx;

    // Allocating shared memory
    // automatic variables are stored in registers that are private to threads
    __shared__ float sh_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float sh_B[TILE_WIDTH][TILE_WIDTH];

    // Parallel mat mul
    float value = 0;
    for (int phase = 0; phase < N/TILE_WIDTH; phase++)
    {
        // Load Tiles into shared memory
        sh_A[ty][tx] = A[(i)*N + phase*TILE_WIDTH+tx];
        sh_B[ty][tx] = B[(phase*TILE_WIDTH + ty)*N+j];
        __syncthreads(); // Make sure all threads completed its work

        // Dot product
        for (int k = 0; k < TILE_WIDTH; k++)
            value += sh_A[ty][k] * sh_B[k][tx];
        __syncthreads();
    }
    // Assigning calculated value
    C[i*N+j] = value;
}
```

А где тут Варпы?

source

Выполнение операций на GPU

```
__global__ void tiled_sq_mat_mul_kernel(float* A, float* B, float* C, int N)
{
    // Ensure that TILE_WIDTH = BLOCK_SIZE
    assert(TILE_WIDTH == blockDim.x);
    assert(TILE_WIDTH == blockDim.y);

    // Ensure N%TILE_WIDTH == 0
    assert(N % TILE_WIDTH == 0);

    // Details regarding this thread
    int by = blockIdx.y;
    int bx = blockIdx.x;

    int ty = threadIdx.y;
    int tx = threadIdx.x;

    // Working on C[i,j]
    int i = TILE_WIDTH*by + ty;
    int j = TILE_WIDTH*bx + tx;

    // Allocating shared memory
    // automatic variables are stored in registers that are private to threads
    __shared__ float sh_A[TILE_WIDTH][TILE_WIDTH];
    __shared__ float sh_B[TILE_WIDTH][TILE_WIDTH];

    // Parallel mat mul
    float value = 0;
    for (int phase = 0; phase < N/TILE_WIDTH; phase++)
    {
        // Load Tiles into shared memory
        sh_A[ty][tx] = A[(i)*N + phase*TILE_WIDTH+tx];
        sh_B[ty][tx] = B[(phase*TILE_WIDTH + ty)*N+j];
        __syncthreads(); // Make sure all threads completed its work

        // Dot product
        for (int k = 0; k < TILE_WIDTH; k++)
            value += sh_A[ty][k] * sh_B[k][tx];
        __syncthreads();
    }
    // Assigning calculated value
    C[i*N+j] = value;
}
```

Используются аппаратно

- В коде размер блока задаётся через blockDim.x и blockDim.y.
- Например, если TILE_WIDTH = 32, то блок имеет размер 32x32 = 1024 потока.
- Эти 1024 потока делятся на 32 warp'а (по 32 потока в каждом).

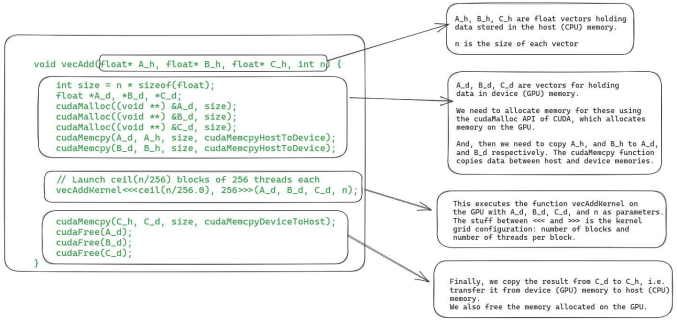
source

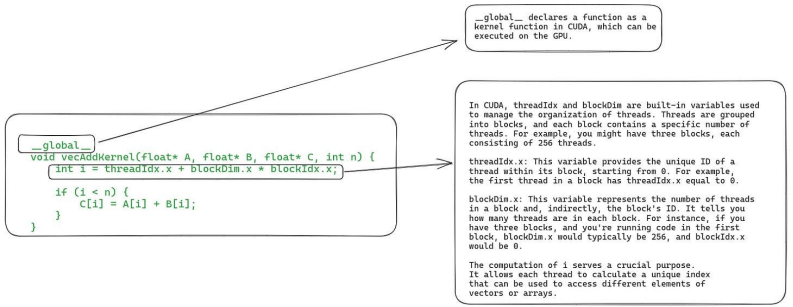
Выполнение операций на GPU - SUMMARY

1. Запуск ядра и организация потоков

Запуск CUDA/GPU ядра с **сеткой блоков потоков**, где каждый блок вычисляет подматрицу (тайл) результата.

- Каждому блоку потоков назначается вычисление **тайла** выходной матрицы (например, 16x16 или 32x32 элементов).
- Каждый поток в блоке вычисляет **один элемент** (или небольшой набор) выходного тайла.
- Для сопоставления потоков с индексами матрицы используется **2D-сетка и размеры блоков**



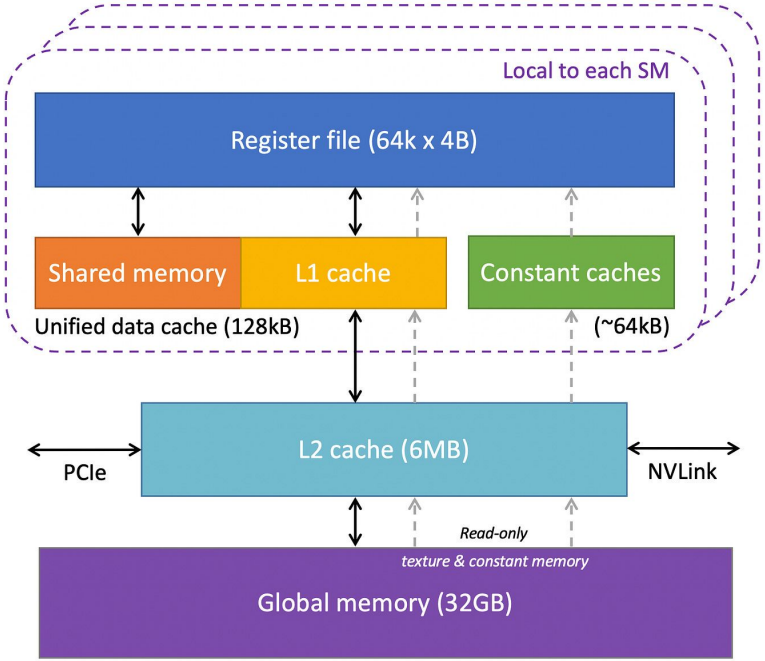


Выполнение операций на GPU - SUMMARY

2. Выделение памяти на GPU

Копирование входных матриц **A** и **B** с CPU (хоста) в глобальную память GPU (устройства).

- **Глобальная память**: Хранит матрицы **A** и **B** для доступа всех потоков/SM.
- **Общая память**: Зарезервирована для каждого блока, чтобы кэшировать тайлы **A** и **B**.



Выполнение операций на GPU - SUMMARY

3. Тайлинг: Загрузка тайлов в общую память

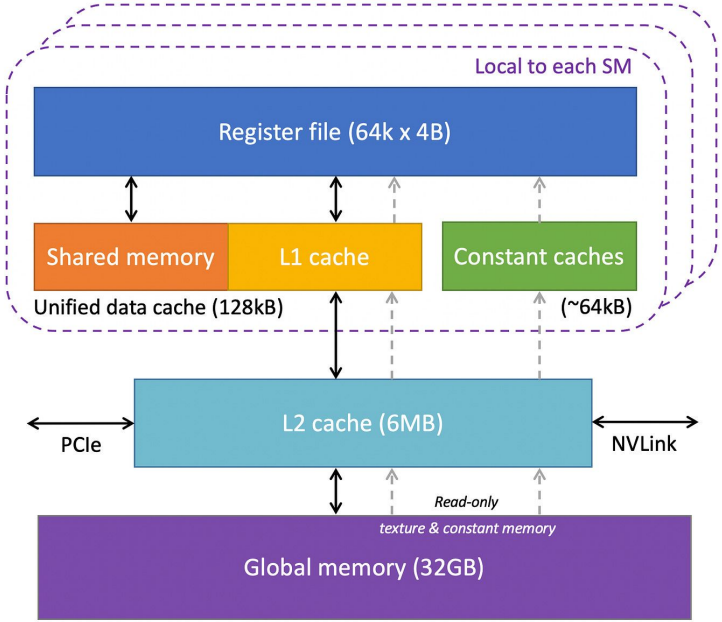
Каждый блок потоков загружает **тайлы** матриц **A** и **B** из глобальной памяти в общую память.

Тайлинг: Разбивает матрицы на меньшие тайлы (например, 16x16), которые помещаются в общую память.

Совместная загрузка: Потоки в блоке работают вместе, чтобы загрузить тайлы:

- Каждый поток загружает **один элемент** из глобальной памяти в общую память.
- Используется **коалесцированный доступ к памяти** (соседние потоки обращаются к соседним адресам глобальной памяти) для повышения эффективности.

Общая память: Тайлы **A** и **B** кэшируются здесь для быстрого повторного доступа.

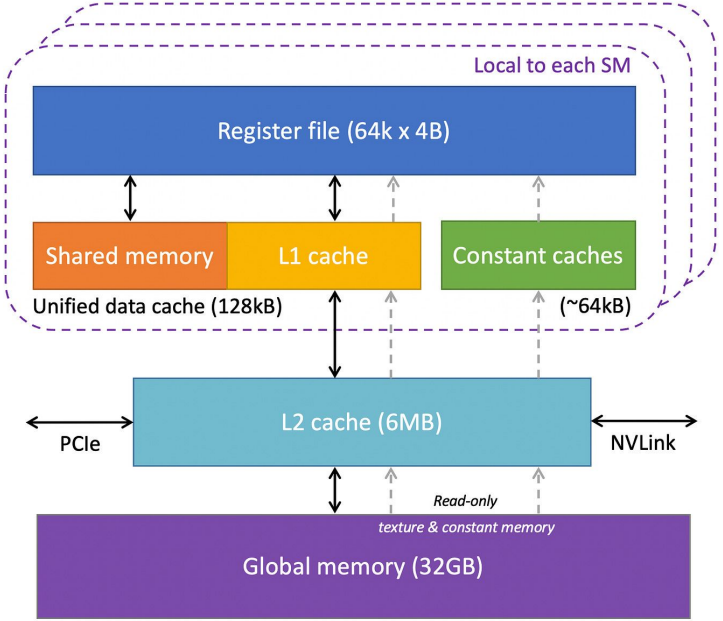


Выполнение операций на GPU - SUMMARY

5. Вычисление частичных скалярных произведений

Каждый поток вычисляет частичное скалярное произведение для своего назначенного элемента выходной матрицы.

- **Цикл по тайлам**: Для каждой пары тайлов (из **A** и **B**) вычислите скалярное произведение.
- **Регистры**: Хранят промежуточные результаты (например, `sum += A_tile[row][k] * B_tile[k][col]`)
- **Эффективность на уровне warps**: Warps (32 потока) выполняют инструкции синхронно (SIMD), скрывая задержки за счёт переключения контекста.

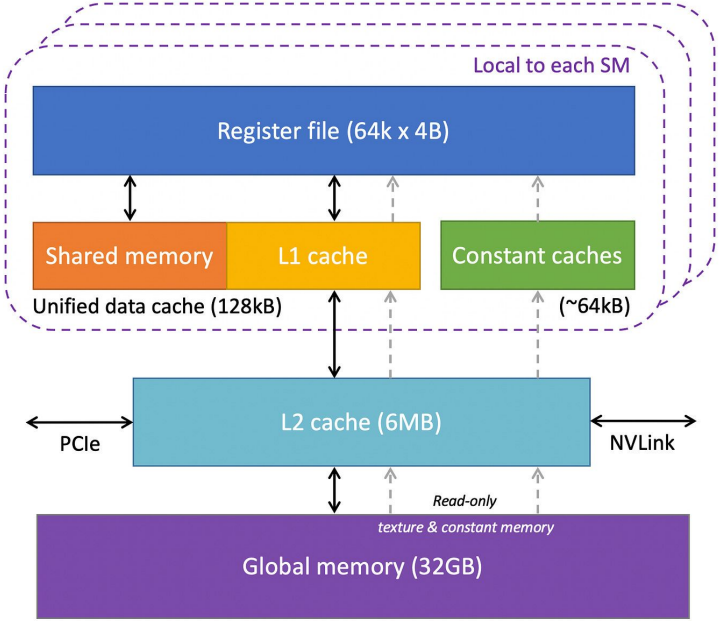


Выполнение операций на GPU - SUMMARY

6. Повторение тайлинга и накопление

Загрузка следующей пары тайлов в общую память, и вычисление их вклада в скалярное произведение, накопление результатов.

- Цикл продолжается, пока все тайлы строки (для **A**) и столбца (для **B**) не будут обработаны.
- Тайлы в общей памяти повторно используются, что сокращает доступ к глобальной памяти.

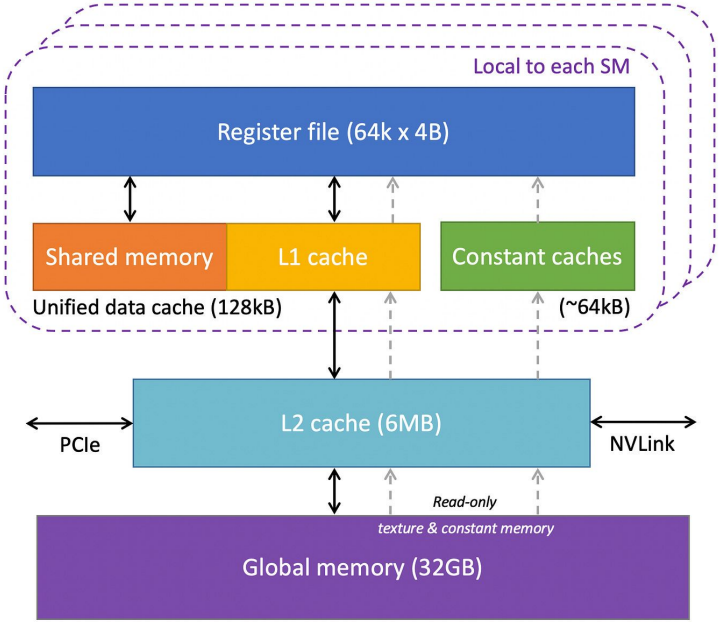


Выполнение операций на GPU - SUMMARY

7. Завершение вычислений и запись в глобальную память

Запись вычисленных значений (хранящееся в регистрах) в выходную матрицу **C** в глобальной памяти.

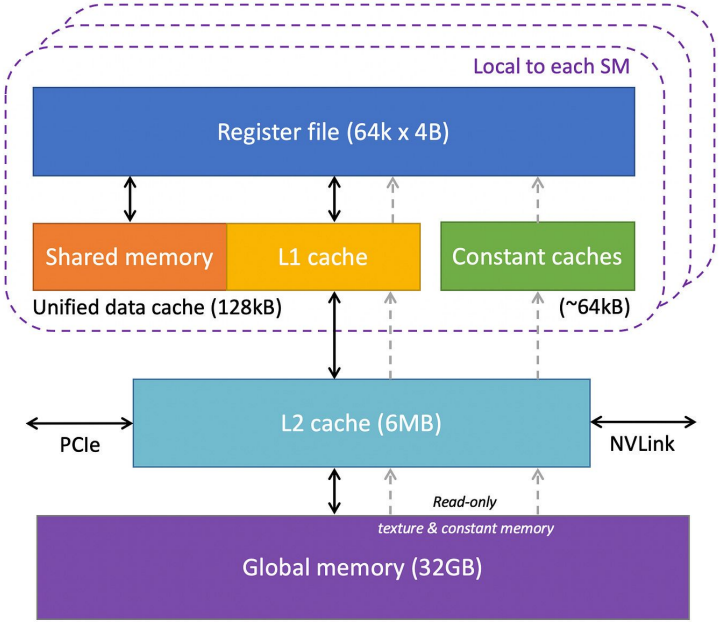
- Каждый поток записывает свой результат в правильную позицию в **C** через глобальную память.
- **Коалесцированные записи**: Обеспечивает что соседние потоки записывают данные в соседние адреса памяти для повышения эффективности.



Выполнение операций на GPU - SUMMARY

Иерархия памяти

1. **Глобальная память**: Исходные данные (A, B) и конечный результат (C).
2. **Общая память**: Кэшированные тайлы A и B для быстрого доступа.
3. **Кэш L1/L2**: Кэширование часто используемых данных из глобальной памяти (например, повторно используемых элементов матриц).
4. **Регистры**: Хранение промежуточных сумм, приватных для потоков.



TRITON vs cuBLAS

Feature	CUDA	Triton
Performance	- Fine-grained control for maximum performance.	- Good performance, but may not match highly optimized CUDA kernels.
Ecosystem	- Mature ecosystem with extensive libraries (cuBLAS, cuDNN, etc.).	- Limited ecosystem; newer and lacks extensive libraries.
Ease of Use	- Steep learning curve; requires knowledge of GPU architecture.	- Python-based, easier for those familiar with Python.
Automatic Optimization	- Manual optimization required for performance tuning.	- Automatic optimization for data locality and parallelism.
Integration	- Widely supported in various frameworks (TensorFlow, PyTorch).	- Well-integrated with PyTorch, ideal for machine learning workloads.
Flexibility	- Low-level access allows for highly customized kernels.	- Provides block-structured iteration spaces for flexibility.
Vendor Lock-In	- Proprietary to Nvidia GPUs, limiting portability.	- Also Nvidia-only, does not solve vendor lock-in.
Development Overhead	- Time-consuming to write and optimize kernels.	- Faster prototyping and deployment of custom kernels.
Community and Support	- Large community and extensive documentation available.	- Smaller community; documentation is still developing.

Вопросы?