

# ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ ВЫЧИСЛЕНИЙ НА GPU С ИСПОЛЬЗОВАНИЕМ OPENCL

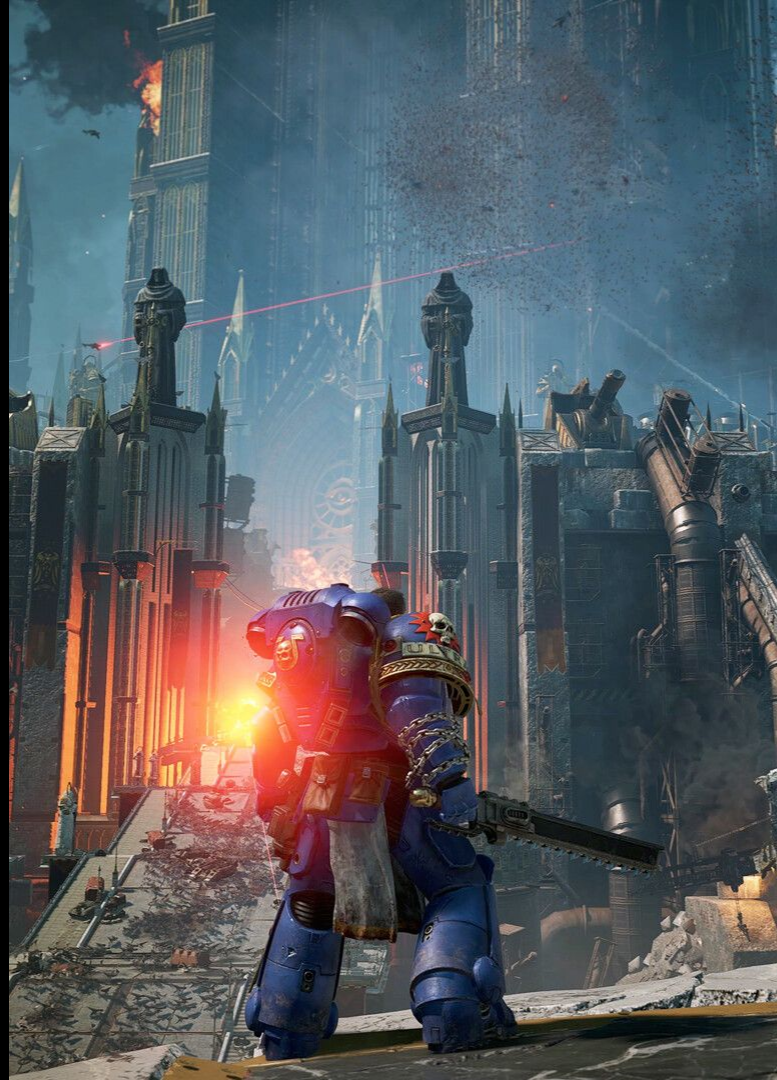
ЕГОР ОРАЧЕВ,  
RENDER PROGRAMMER,  
SABER INTERACTIVE

5 ИЮНЯ, 2024

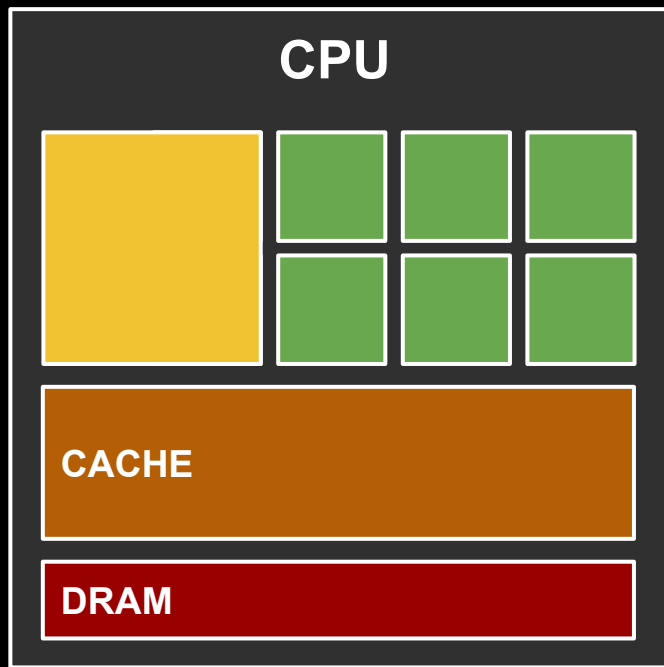


# О себе

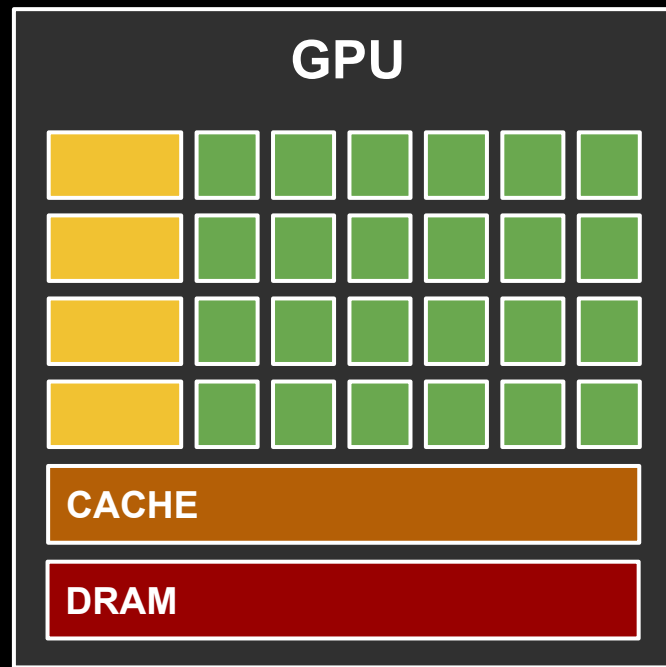
- Более 2-х лет работаю в Saber Interactive на позиции Render Programmer
- Работаю над проектом **Warhammer 40K: Space Marine 2** и не только
- Был интерном в лаборатории языковых инструментов JetBrains Research
- Учился в магистратуре и бакалавриате СПбГУ на специальности “Программная инженерия”



# Мотивация



**VS**



# Содержание

- Краткая история развития GPU
  - Что такое GPGPU-вычисления
  - Введение в OpenCL
  - Примеры программ на OpenCL
  - Современные архитектуры
  - Проблемы производительности в GPU
  - Разреженная линейная алгебра на GPU
  - Стандарт GraphBLAS и проект spla
- Как развивалось
  - Идея
  - Для нас
  - Как реализовать
  - Как устроено
  - Как сделать быстро
  - Где применять
  - Реальный проект

# Краткая история эволюции GPU

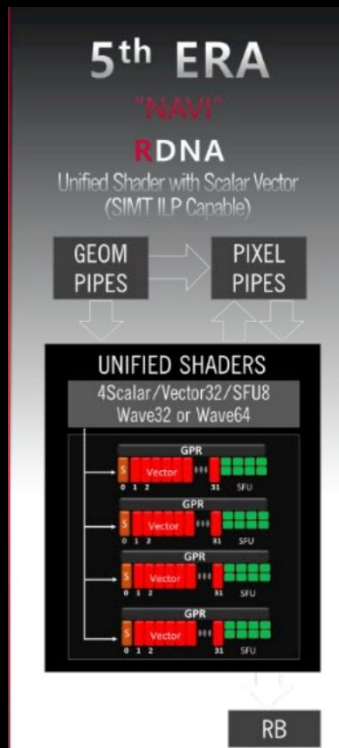
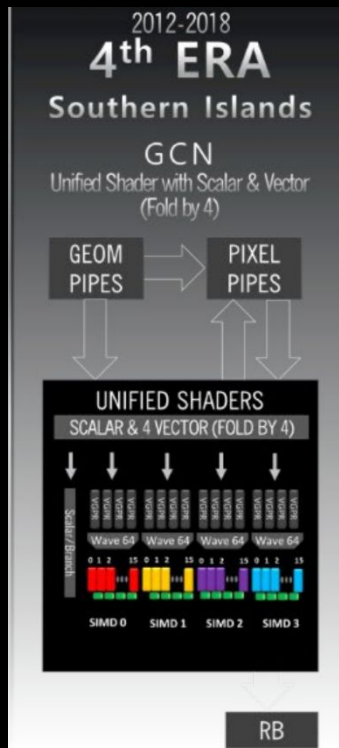
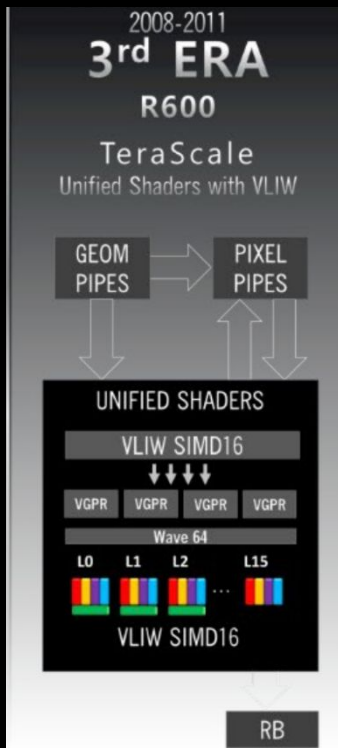
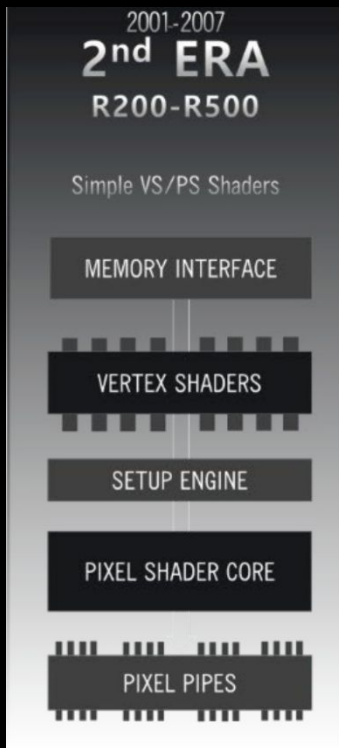

**PRE 2000**  
**1st ERA**  
**R100**

Fixed Function

3D GEOMETRY TRANSFORMATION

$$V_{out} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = MVP \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix} \cdot V_{in} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$
$$V_{out} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = M_{proj} \cdot V_{in}$$

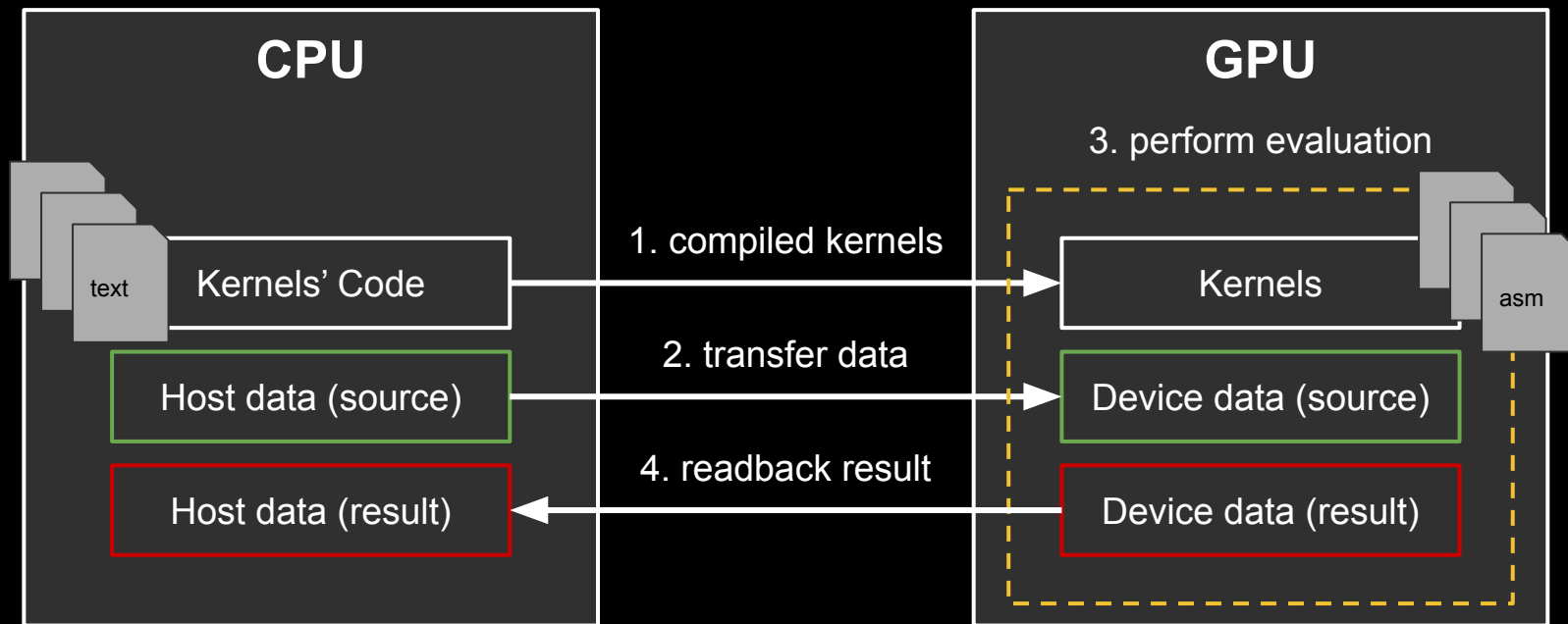
LIGHTING

$$C_p = k_a L_a + \sum_{n=lights} Att_n(k_d(\hat{L}_a \cdot \hat{N}) + k_s(\hat{R}_a \cdot \hat{V})^g)$$


# Типовое GPU приложение



# GP GPU-вычисления



# Timeline вычислений

**CPU**

Компиляция  
GPU ядер

Подготовка  
данных

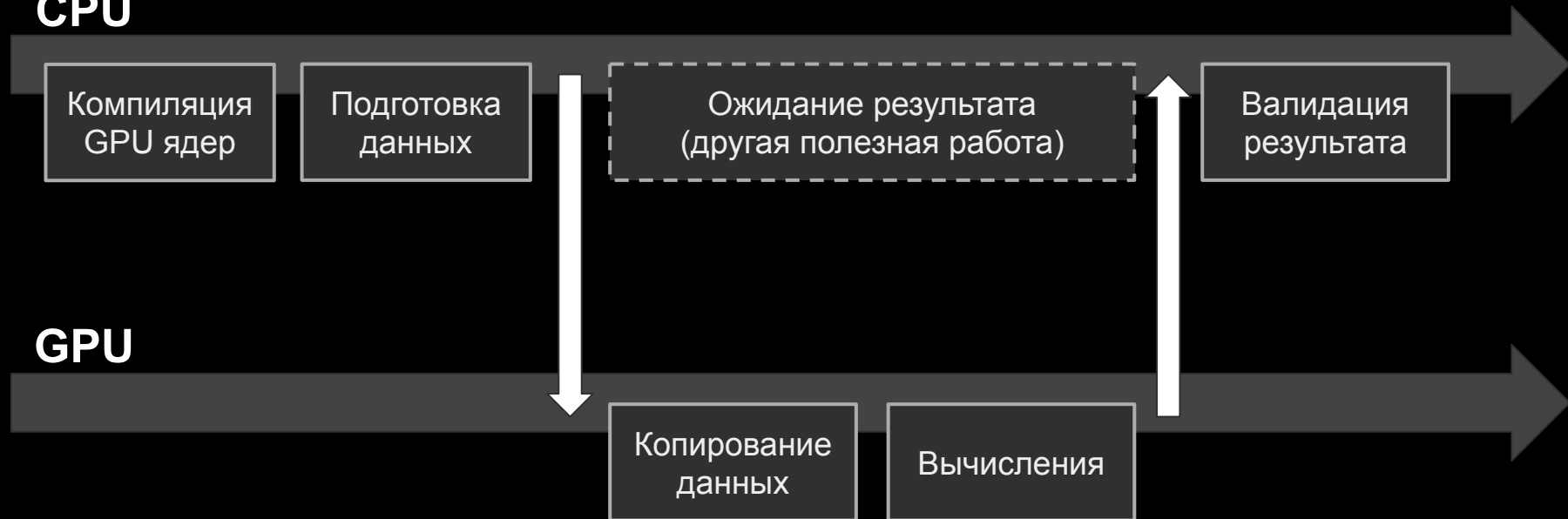
Ожидание результата  
(другая полезная работа)

Валидация  
результата

**GPU**

Копирование  
данных

Вычисления





# API для работы

- Графические вычисления

- Vulkan
- Direct3D
- Metal
- GNM / GNMX
- OpenGL

→ Графические приложения  
Пользовательский интерфейс  
Видеоигры

- Неграфические вычисления

- CUDA
- OpenCL

→ Математическое ПО  
Машинное обучение  
Анализ данных

# CUDA

- Compute unified device architecture
- Платформа параллельных вычислений
- Промышленный API
- Язык, модель, набор инструментов, библиотеки, компиляторы, etc.



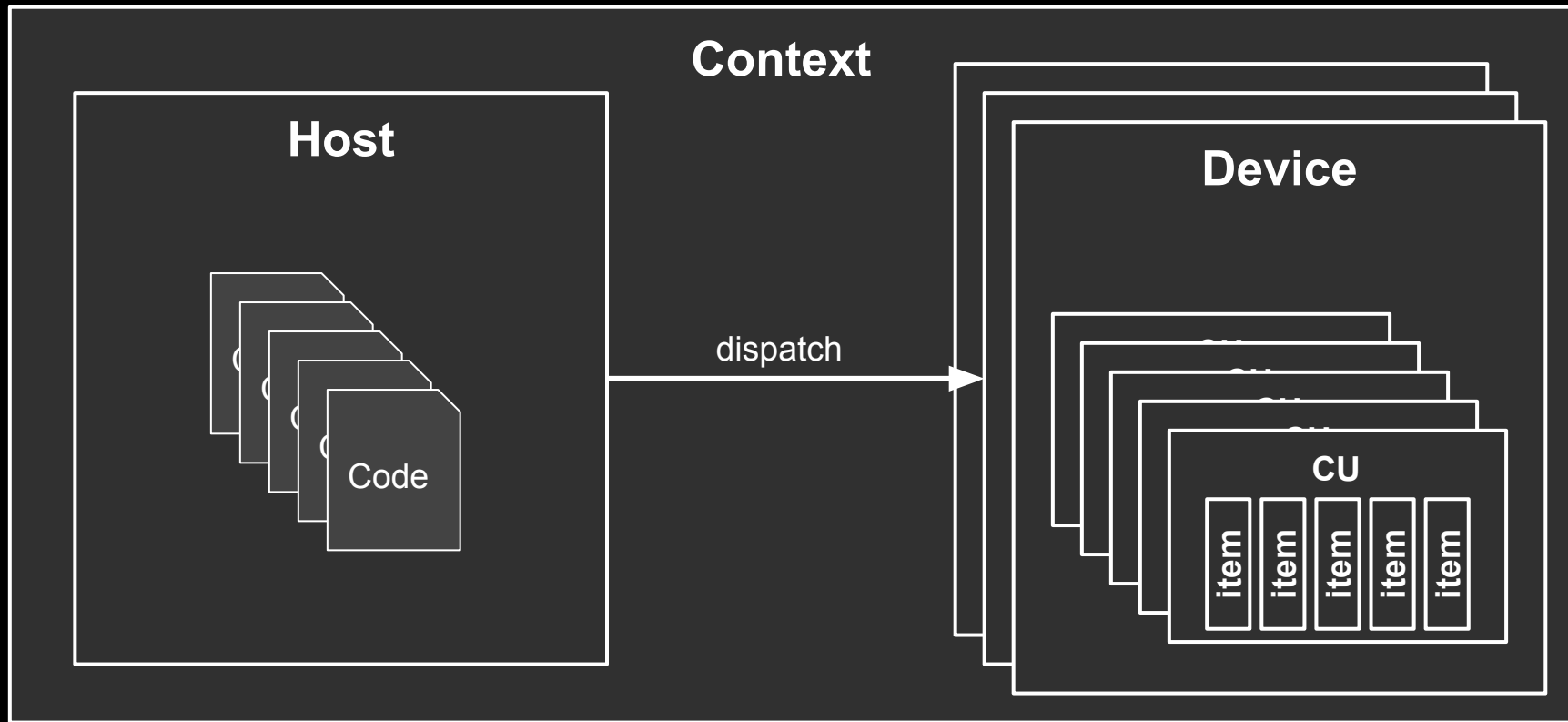
# OpenCL

- Open Computing Language
- Фреймворк для написания компьютерных программ, связанных с параллельными вычислениями на различных графических и центральных процессорах, а также FPGA
- Включает язык программирования, и интерфейс программирования приложений
- Обеспечивает параллелизм на уровне инструкций и на уровне данных и является осуществлением техники GPGPU
- Является полностью открытым стандартом
- Доступен на ускорителях Intel, Nvidia, AMD

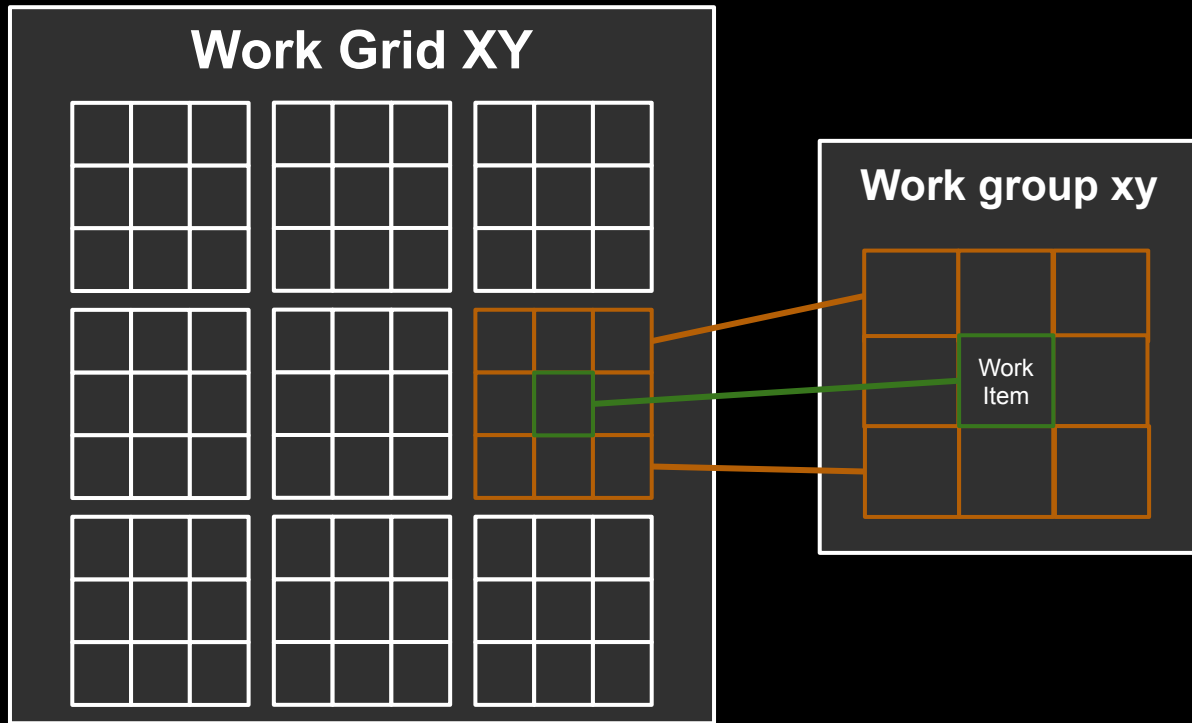
# Модель OpenCL

- Platform  
Устройства и контекст выполнения
- Memory  
Иерархия памяти, видимость операций чтения/записи, кеш
- Execution  
Выполнение программ на GPU и синхронизация
- Programming  
Языковые и программные инструменты для работы

# Platform model

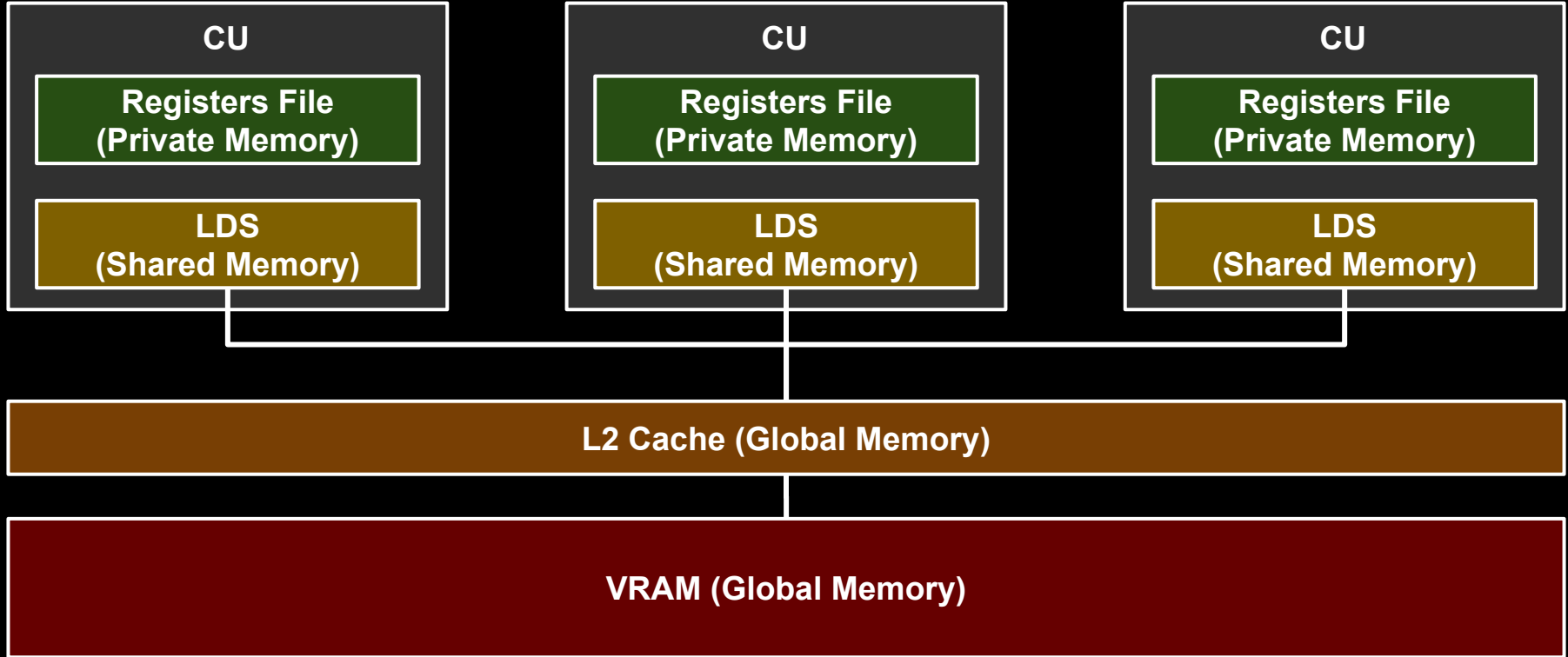


# Execution model



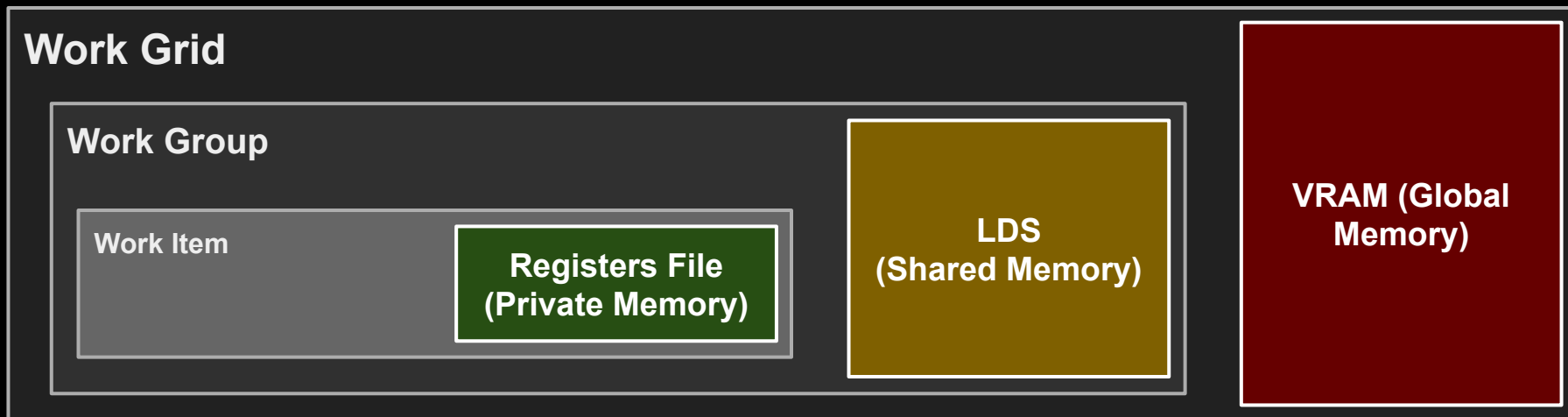
- Total items  
 $X * Y$
- Group size  
 $x * y$
- Group count  
 $(X / x) * (Y / y)$
- В примере  
Сетки 9x9  
Размер группы 3x3

# Memory model



# Memory model

- Одному **Work Item** доступны его приватные регистры
- Все потоки внутри **Work Group** могут иметь общую Shared Memory
- Всем **Work Group**'ам доступна общая глобальная память





# Programming model

- Task-parallel processing
- Выполнение фиксированной функции над 1 элементом
- Запуск сетки потоков для параллельной обработки всех элементов
- Определение программы (ядра) как процедуры на C подобном языке, которая вызывается для каждого work item

```
__kernel void my_kernel(__global const int* a, __global const int* b, __global int* c) {  
    size_t idxInGrid = get_global_id(0); // Item id in grid  
    size_t idxInGroup = get_local_id(0); // Item id in group  
    int x = a[idxInGrid]; // Load from global memory  
    __shared int d[GROUP_SIZE]; // Declare shared array, visible for all items inside our group  
    . . .  
}
```

# Базовые определения

- Platform → Платформа
- Device → Устройство
- Context → Контекст
- Buffer → Буфер (регион памяти)
- Program → Программа
- Kernel → Ядро
- CommandQueue → Очередь команд
- Dispatch → Запрос на исполнение
- NDRange → N-мерный регион

# Пример программы

- Создать C++ проект
- Редактировать файл main.cpp
- Использовать `#include <CL/opencl.hpp>`

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.15 FATAL_ERROR)
project(cl_intro LANGUAGES CXX)

find_package(OpenCL REQUIRED)
add_executable(cl_example main.cpp)
target_link_libraries(cl_example PRIVATE OpenCL)
```

# Платформа

- Конкретная доступная реализация OpenCL на вашем устройстве
- Зависит от вендора
- Intel, Nvidia, AMD, Apple, etc.

```
std::vector<cl::Platform> platforms;  
cl::Platform::get(&platforms);  
cl::Platform platform = platforms.front();
```

# Девайс

- Физически доступное устройство для выполнения вычислений
- Девайс имеет определенного вендора
- Доступен только в рамках одной платформы
- Имеют разный тип: GPU, CPU, ACCELERATOR, CUSTOM

```
std::vector<cl::Device> devices;  
platform.getDevices(CL_DEVICE_TYPE_GPU, &devices);  
cl::Device device = devices.front();
```

# Контекст

- Объединяет один или несколько девайсов
- Среда для выполнения OpenCL кода и команд

```
cl::Context ctx(device);
```

# Буфер

- Непрерывная область памяти
- Доступен для чтения/записи внутри OpenCL ядер
- Можно читать/писать со стороны хост-приложения
- Способ передачи данных между CPU - GPU, GPU - GPU, etc.

```
cl::Buffer a(ctx, CL_MEM_READ|CL_MEM_COPY_HOST_PTR, sizeof(int)*N, p_a);  
cl::Buffer b(ctx, CL_MEM_READ|CL_MEM_COPY_HOST_PTR, sizeof(int)*N, p_b);  
cl::Buffer c(ctx, CL_MEM_WRITE, sizeof(int)*N, nullptr);
```

# Программа

- Программа это текст на C подобном языке с спец. возможностями
- Объект программы создается из исходного кода
- Компиляция может осуществляться в runtime
- Процесс компиляции *медленный* даже для простых программ

```
std::string kernel_code =  
"__kernel void add(__global const int* a, __global const int* b, __global int* c, uint count) { "  
"    size_t idx = get_global_id(0); "  
"    if (idx < count) { c[idx] = a[idx] + b[idx]; } "  
"}";
```

```
cl::Program program(ctx, kernel_code);    // context where to create  
program.build(device, "-cl-std=CL1.2");  // build op-code for a specific device
```



# Ядро

- Специальная именованная функция внутри программы
- Может быть поставлена на исполнение со стороны хост-программы
- Имеет состояние, набор аргументов
- Для создания требуется скомпилированная `cl::Program`

```
cl::Kernel kernel(program, "add");  
kernel.setArg(0, a); // Buffet  
kernel.setArg(1, b); // Buffer  
kernel.setArg(2, c); // Buffer  
kernel.setArg(3, N); // const uint
```

# Очередь

- Последовательность команд для выполнения
- По умолчанию: идут в строгом порядке
- Выполнение на GPU не синхронизировано с хост-программой
- Требуются явные точки синхронизации (GPU  $\rightarrow$  GPU, GPU  $\rightarrow$  CPU)

```
cl::CommandQueue queue(ctx);
```

# Выполнение

- NDRange конфигурирует логическую сетку потоков
- Global – общий размер сетки X,Y,Z
- Local – размер групп x,y,z на которые разбивается global сетка

```
cl::NDRange global(N);  
cl::NDRange local(32);  
  
queue.enqueueNDRangeKernel(kernel, cl::NDRange(), global, local);  
queue.enqueueReadBuffer(c, false, 0, sizeof(int)*N, p_c);  
queue.finish();  
  
// After this point we can observe result in p_c
```

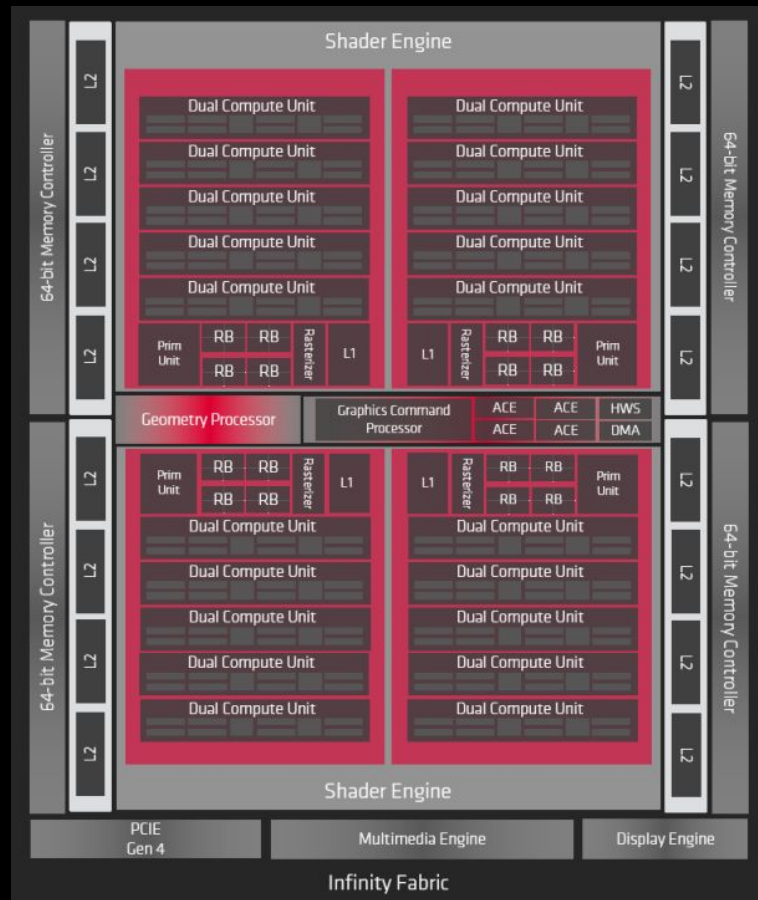
# Архитектуры

- Nvidia Pascal
- Nvidia Turing
- Nvidia Ampere
- Nvidia Ada Lovelace
- AMD GCN (Graphics Core Next)
- AMD RDNA (Radeon DNA)
- AMD RDNA-2

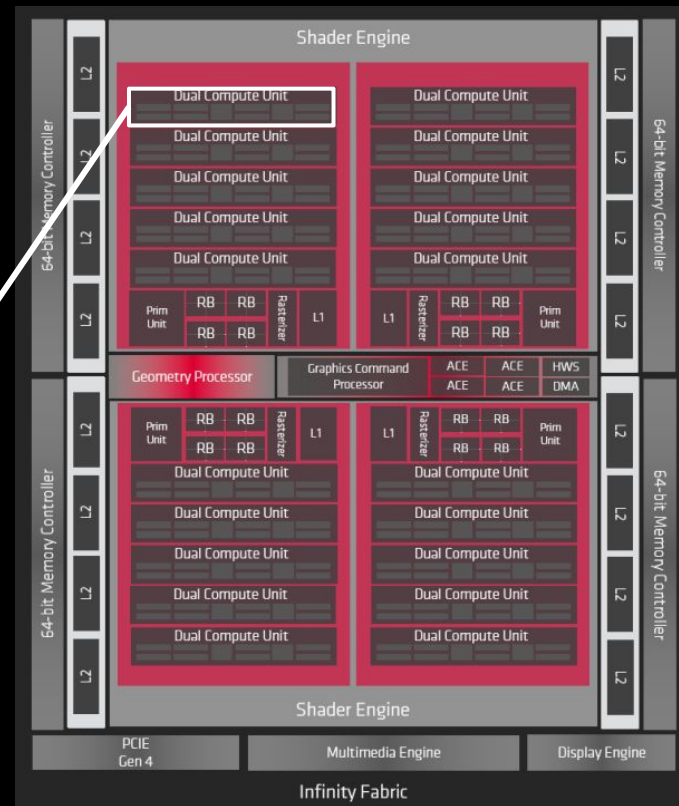
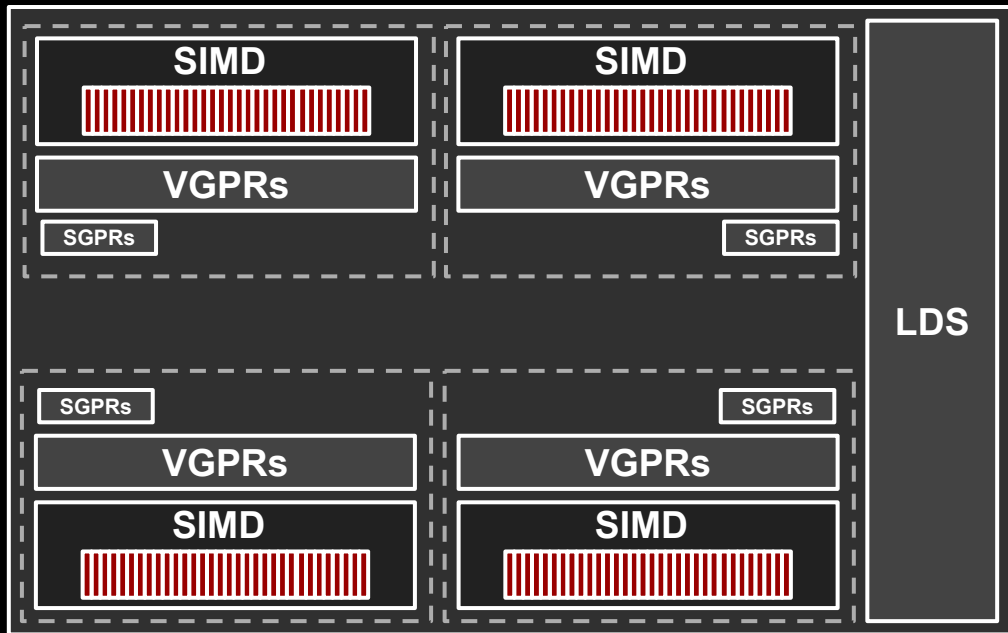


# RDNA

- GPU состоит из набора CU
- CU имеет общий LDS
- Workgroup выполняется на одном CU
- CU (Compute Unit) состоит из набора SIMD (32 ALU) процессоров
- SIMD процессор исполняет 1 *wave32* за 1 такт процессора
- SIMD процессор имеет фиксированный набор VGPR и SGPR регистров

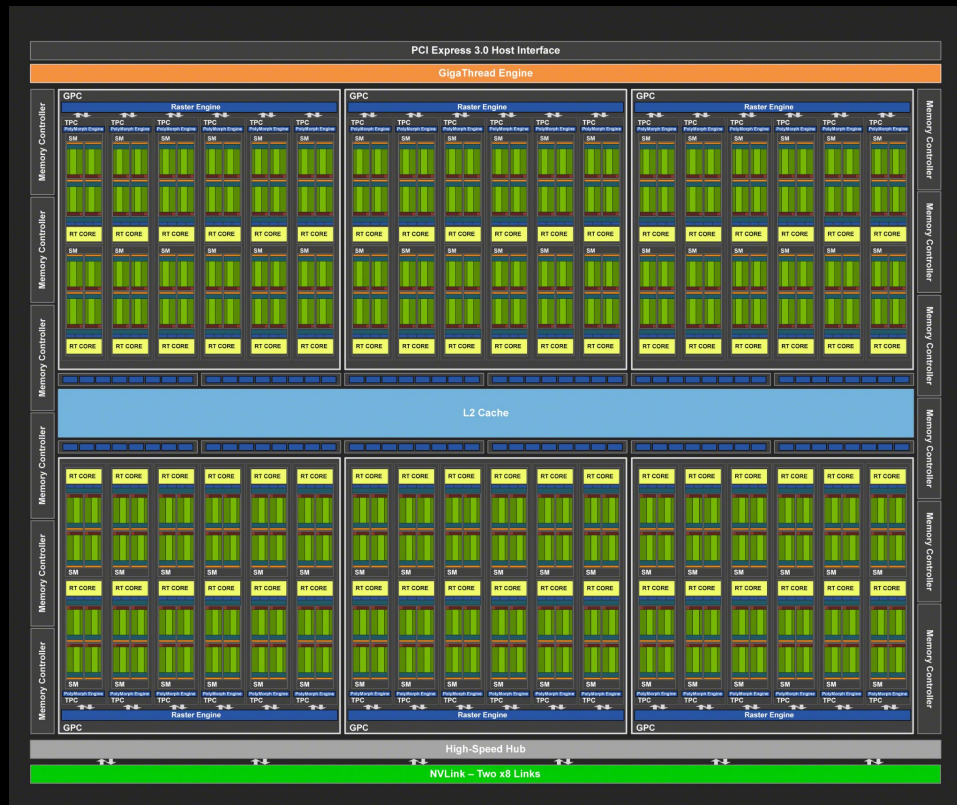


# RDNA Compute Unit



# Nvidia GPUs Architecture

- GPU состоит из набора SM
- SM имеет общий L1 кэш
- SM (streaming multiprocessor) состоит из набора CUDA cores
- CUDA cores объединены в группы *warp* по 32 по принципу SIMD
- 1 SIMD группа выполняется за 1 такт
- Workgroup выполняется на одном SM процессоре



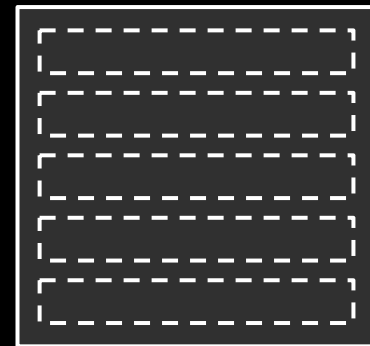
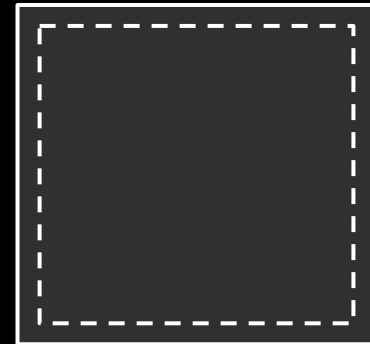
# Факторы производительности

- Occupancy → active / total waves
- Utilization → active / total compute units
- VGPR usage → per wave
- SGPR usage → per wave
- LDS usage → per work group
- Divergence → per wave
- Kernels creation → cpu cost of compilation



# Распределение работы

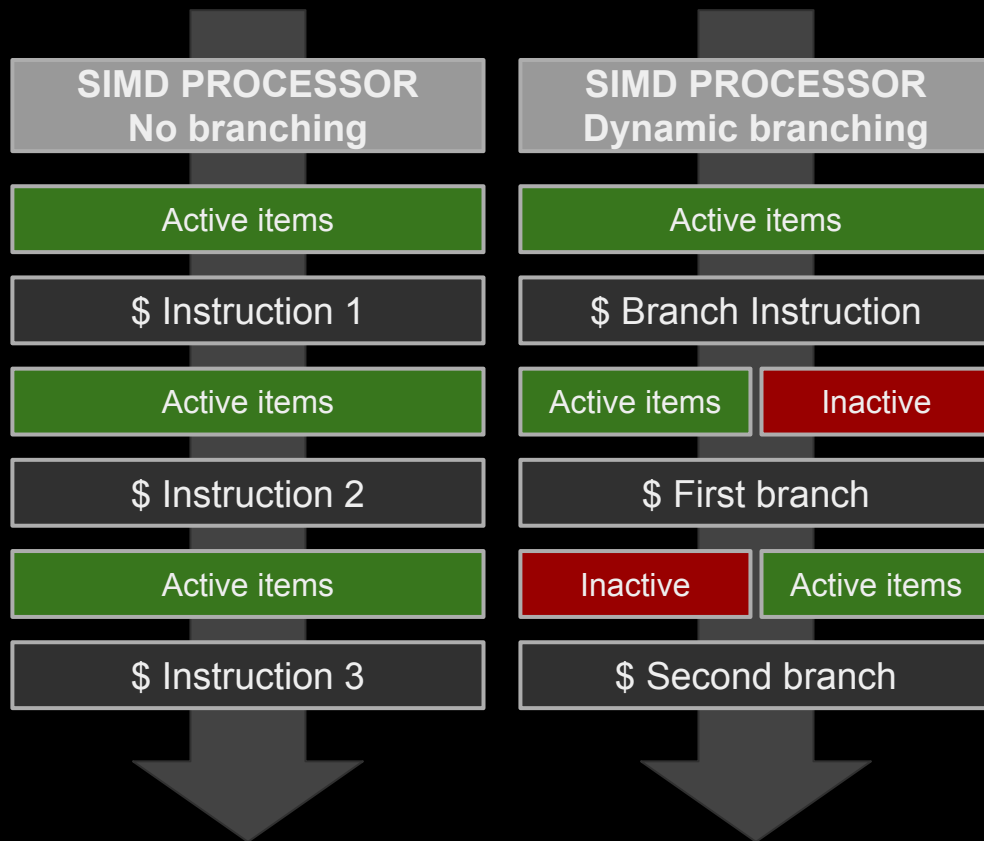
- *Quiz: Какой диспатч лучше?*  
Grid (32, 32) Workgroup (32, 32)  
vs  
Grid (32, 32) Workgroup (32, 1)
- *Quiz: Какой размер группы лучше?*  
Workgroup (16, 1)  
vs  
Workgroup (32, 1)



подсказка

# Дивергенция выполнения

- GPU это массив SIMD процессоров
- 1 SIMD процессор выполняет 1 инструкцию над N элементами за такт
- Что произойдет, если необходимо выполнить разные инструкции над разными элементами?



# Использование локальной памяти

- Локальная (shared) расположена физически на CU
- По скорости уступает только регистрам
- В разы! быстрее чем обращение к глобальной памяти
- Ограничена статически на 1 рабочую группу

## Хорошее правило:

- Записать данные в LDS
- Выполнить вычисление
- Сохранить результат в глобальной памяти

# Типы глобальной памяти

fastest

## Memory Heap 0

Device local

VRAM

fast

## Memory Heap 1

Device local

256MB VRAM

Host visible / coherent

slow

## Memory Heap 2

Host local

RAM

Device visible by PCIe

# Флаги памяти в OpenCL

- CL\_MEM\_READ\_WRITE
- CL\_MEM\_WRITE\_ONLY
- CL\_MEM\_READ\_ONLY
- CL\_MEM\_USE\_HOST\_PTR
- CL\_MEM\_ALLOC\_HOST\_PTR
- CL\_MEM\_COPY\_HOST\_PTR
- CL\_MEM\_HOST\_WRITE\_ONLY
- CL\_MEM\_HOST\_READ\_ONLY
- CL\_MEM\_HOST\_NO\_ACCESS

→ по умолчанию

→ на GPU только запись

→ на GPU только чтение

→ использовать RAM

→ аллоцировать в RAM

→ скопировать данные

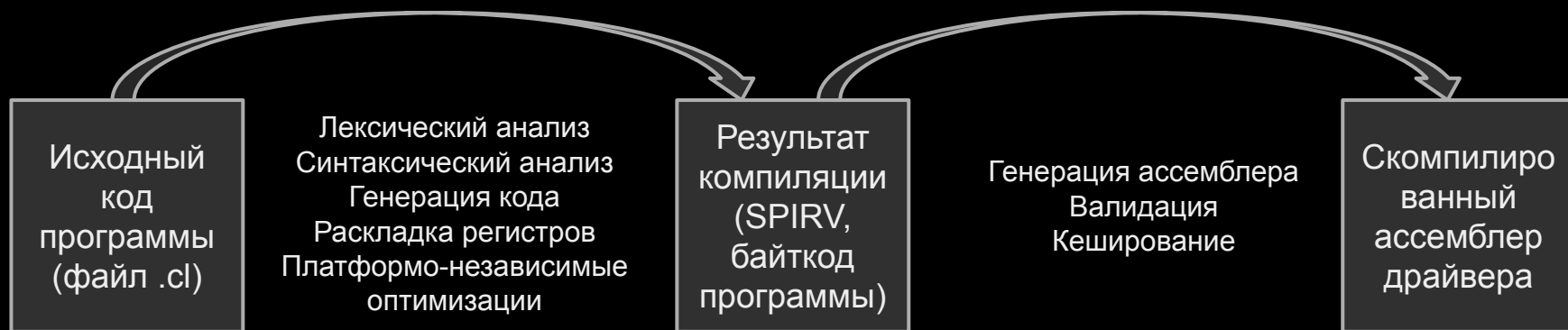
→ доступ с CPU на запись только

→ доступ с CPU на чтение только

→ нет доступа с CPU

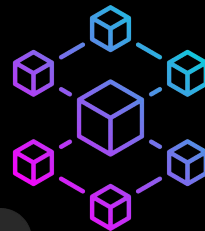
# Компиляция ядер

- Исходная программа с набором ядер написана на языке .cl
- (Видимое для пользователя)  
Компиляция программы в *байткод* при ее создании (**десятки секунд**)
- (Скрытое от пользователя)  
Компиляция отдельных ядер в *ассемблер* специфичный для данной видеокарты и конкретной версии драйвера (**сотни миллисекунд**)



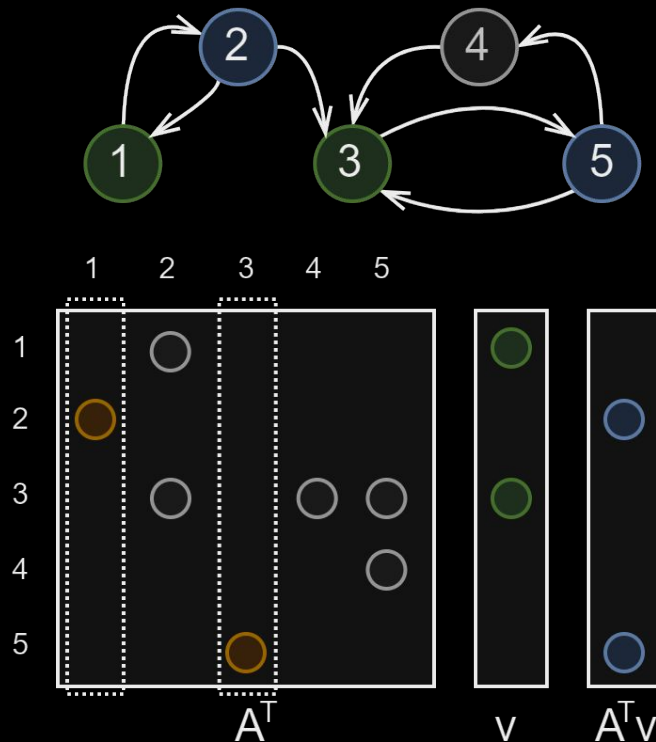
# Практическое применение

- Игровые движки / видеоигры
- Графические редакторы
- Видеомонтаж / фоторедактирование
- UI фреймворки
- Машинное обучение
- Нейронные сети
- Анализ данных
- Блокчейн



# Анализ графов, линейная алгебра и GPU

- Решение прикладных задач
- Хорошая абстракция
- Матрицы, вектора, скаляры
- Операции над матрицами
- Произвольные типы
- GraphBLAS стандарт
- GPU-алгоритмы для вычислений





# GraphBLAS

- Математическая нотация транслированная C API
- Стандарт операций для анализа графов
- SuiteSparse, IBM GraphBLAS, Huawei GraphBLAS, Gunrock GraphBLAST

```
GrB_Vector_new(v, GrB_INT32, n);
GrB_Vector q;
GrB_Vector_new(&q, GrB_BOOL, n);
GrB_Vector_setElement(q, true, s);
int32_t level = 0;
GrB_Index nvals;
do {
    ++level;
    GrB_apply(*v, GrB_NULL, GrB_PLUS_INT32, GrB_SECOND_INT32, q, level, GrB_NULL);
    GrB_vxm(q, *v, GrB_NULL, GrB_LOR_LAND_SEMIRING_BOOL, q, A, GrB_DESC_RC);
    GrB_Vector_nvals(&nvals, q);
} while (nvals);
```

# Spla

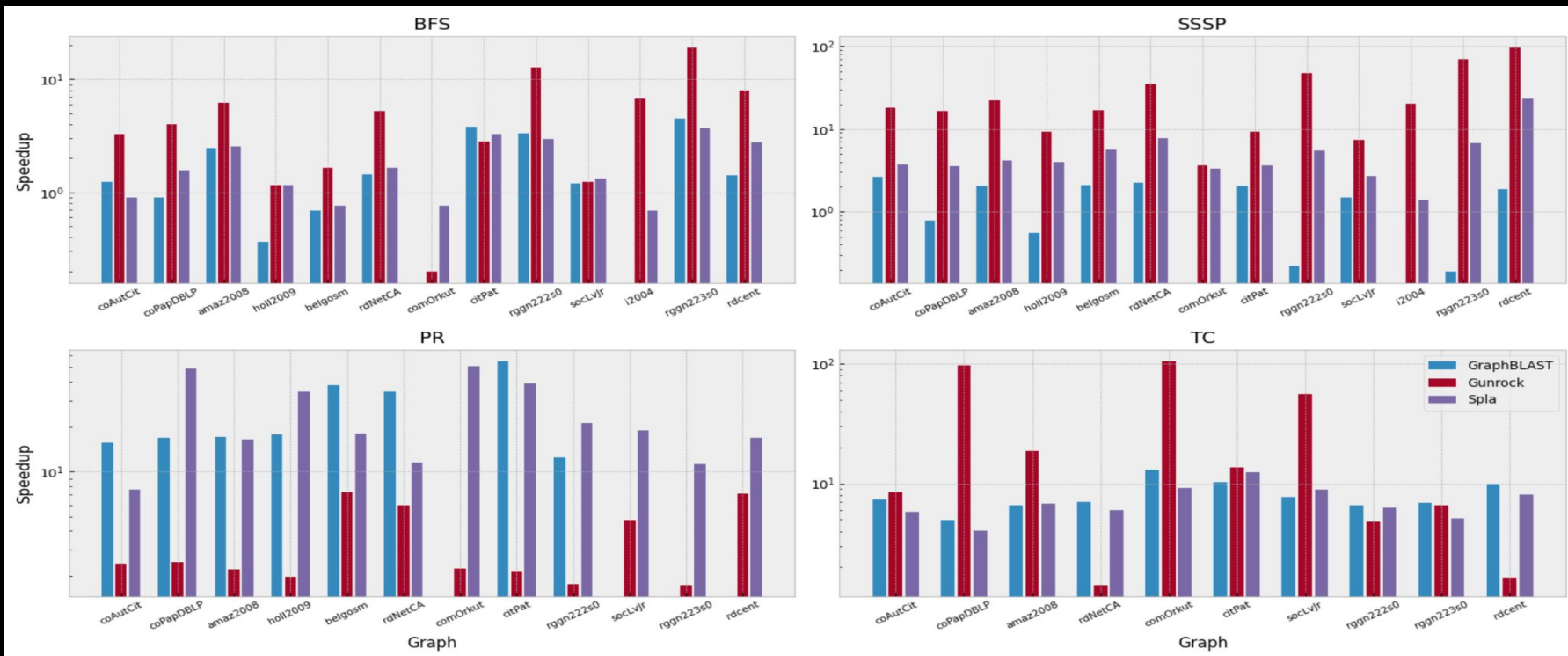
- *“An open-source generalized sparse linear algebra framework with vendor-agnostic GPUs accelerated computations”*
- Библиотека примитивов линейной алгебры
- Опциональное GPU ускорение
- Конфигурация типов элементов
- Выбор операций
- Асинхронное выполнение



# Spla: пример bfs

```
def bfs(s: int, A: Matrix):  
    v = Vector(A.n_rows, INT)  
    front = Vector.from_lists([s], [1], A.n_rows, INT)  
    front_size = 1  
    depth = Scalar(INT, 0)  
    count = 0  
  
    while front_size > 0:  
        depth += 1  
        count += front_size  
        v.assign(front, depth, op_assign=INT.SECOND, op_select=INT.NQZERO)  
        front = front.vxm(v, A, op_mult=INT.LAND, op_add=INT.LOR, op_select=INT.EQZERO)  
        front_size = front.reduce(op_reduce=INT.PLUS).get()  
  
    return v, count, depth.get()
```

# Spla: производительность



Ускорение относительно SuiteSparse::GraphBLAS реализации

# Литература и ссылки

- <https://www.amd.com/system/files/documents/rdna-whitepaper.pdf>
- <https://man.opencl.org/>
- [http://ccfit.nsu.ru/arom/data/CUDA\\_/08%20OpenCL.pdf](http://ccfit.nsu.ru/arom/data/CUDA_/08%20OpenCL.pdf)
- <https://cmp.phys.msu.ru/sites/default/files/OpenCL.pdf>
- <https://medium.com/analytics-vidhya/cuda-compute-unified-device-architecture-part-3-f52476576d6d>
- <https://github.com/SparseLinearAlgebra/spla>