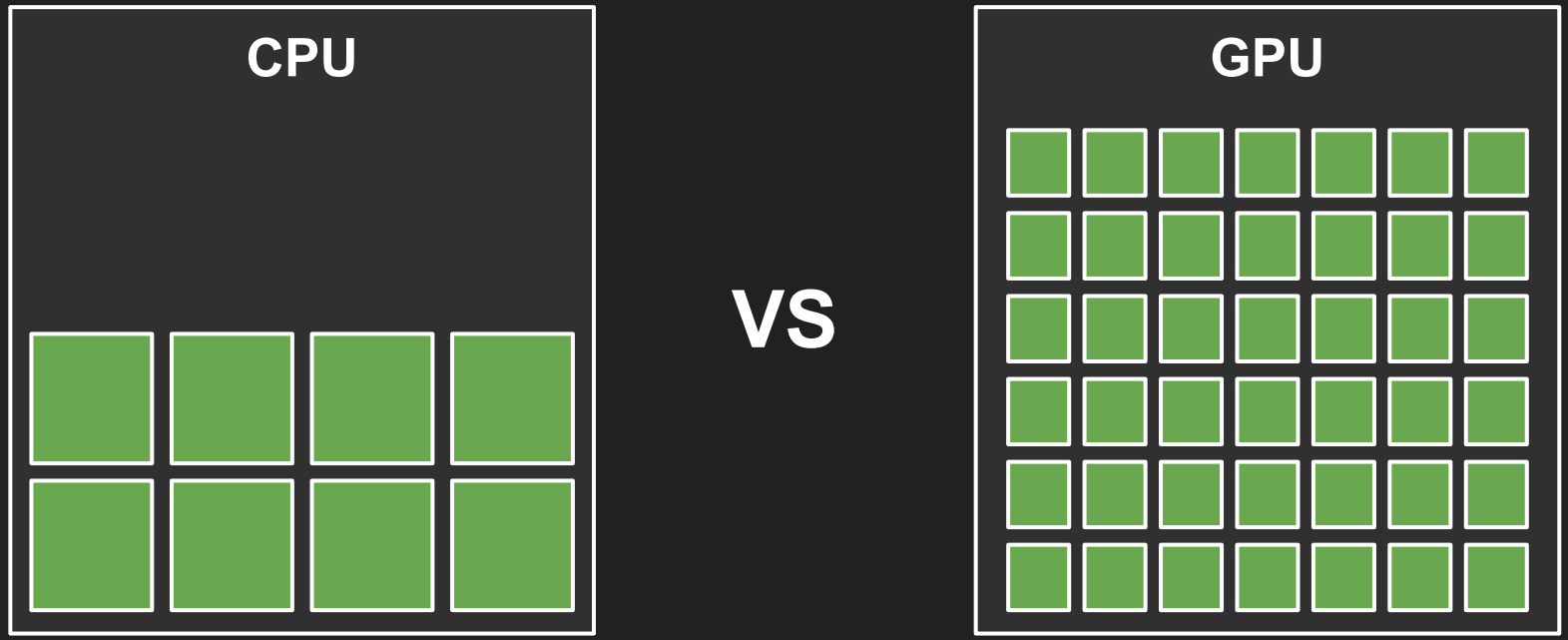
РЕАЛИЗАЦИЯ ЭФФЕКТИВНОЙ РАЗРЕЖЕННОЙ ЛИНЕЙНОЙ АЛГЕБРЫ НА GPU С ИСПОЛЬЗОВАНИЕМ OPENCL

Семинар лаборатории формальных языков

Докладчик: Орачев Егор

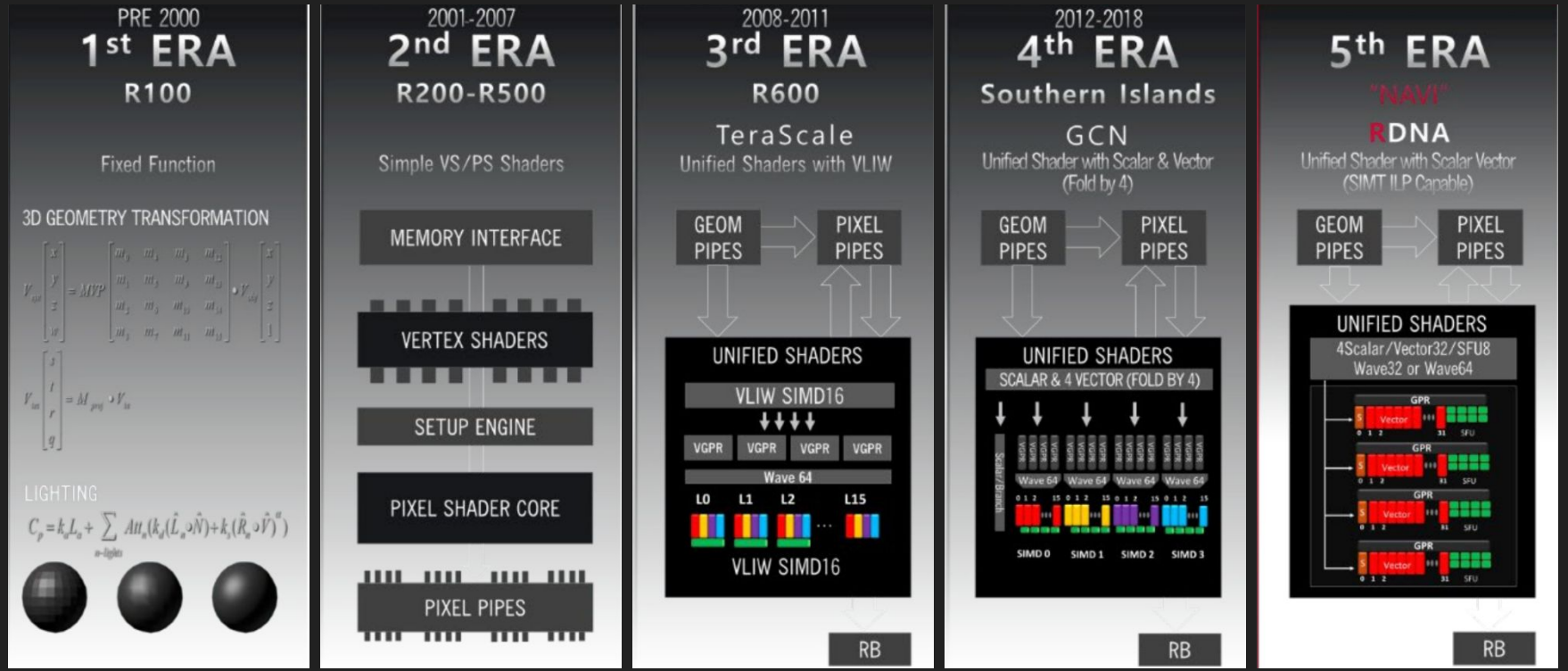
Введение



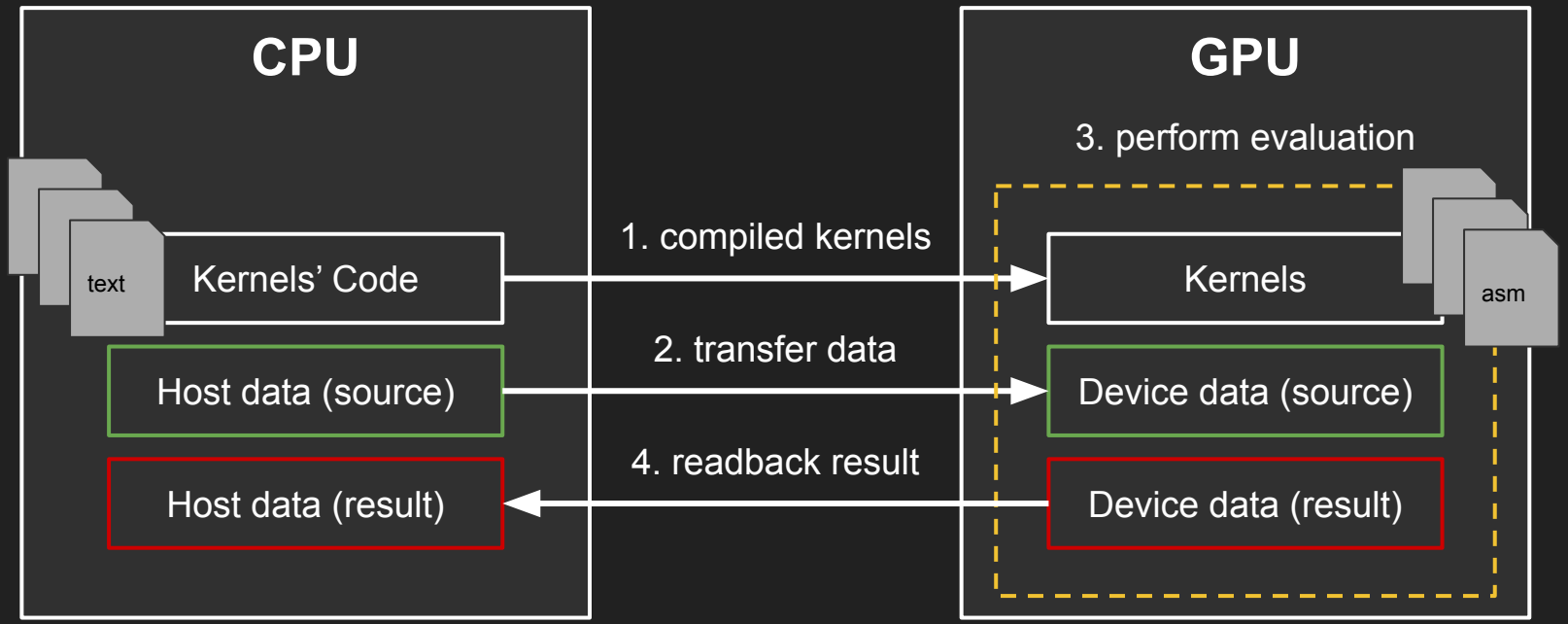
Содержание

- Краткая история развития GPU → Как развивалось?
- Что такое GPGPU-вычисления → Какая идея?
- Введение в OpenCL → Для нас?
- Примеры программ на OpenCL → Как реализовать?
- Современные архитектуры → Как устроено?
- Проблемы производительности в GPU kernels → Как сделать быстро?
- Разреженная линейная алгебра на GPU → Где применять?
- Стандарт GraphBLAS и его развитие → Как развивать?
- Проект spla → Реальный проект?

Краткая история эволюции GPU



GPGPU-вычисления

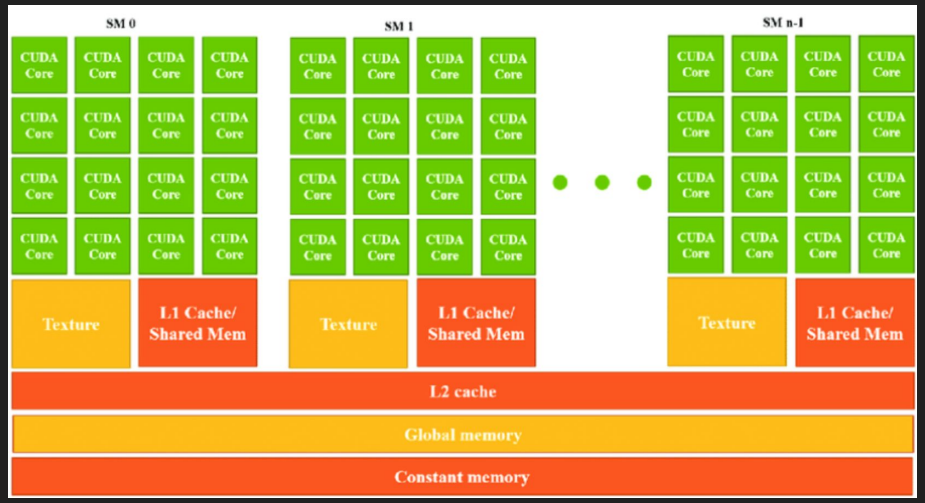


API для работы

- Графические вычисления → Графические приложения, Видеоигры
  - Vulkan
  - Direct3D
  - Metal
  - GNM / GNMX
  - OpenGL
- Неграфические вычисления → Математическое ПО, Анализ данных
  - CUDA
  - OpenCL

CUDA

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



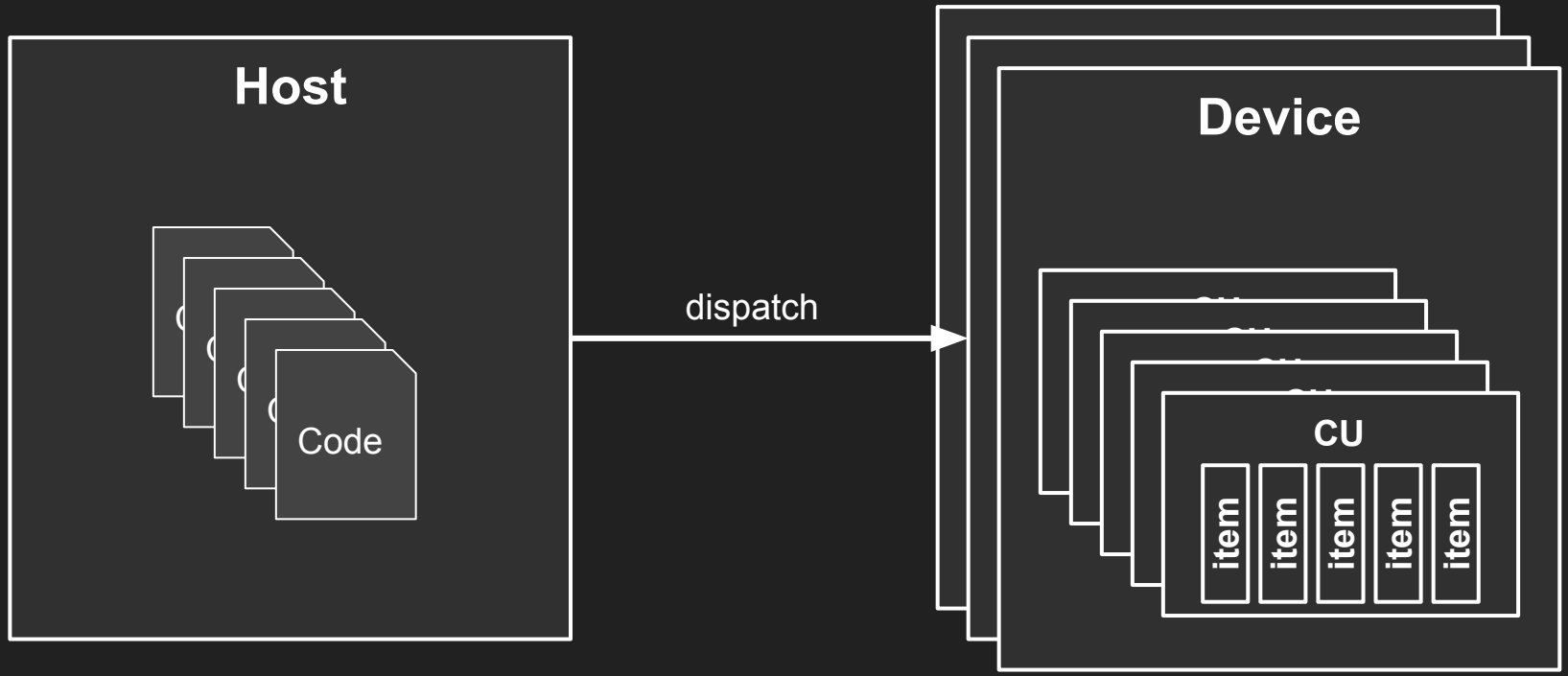
OpenCL

- 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

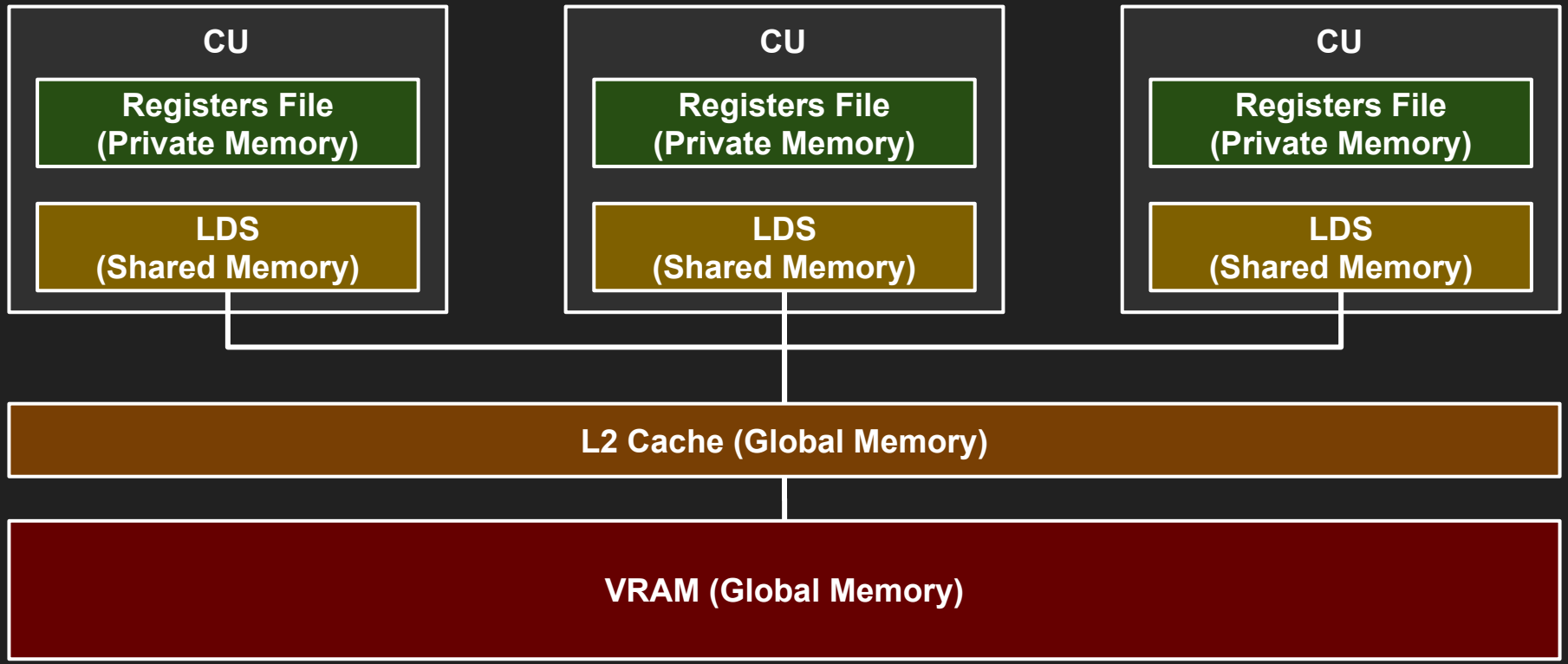
Модель OpenCL

- Platform model
- Memory model
- Execution model
- Programming model

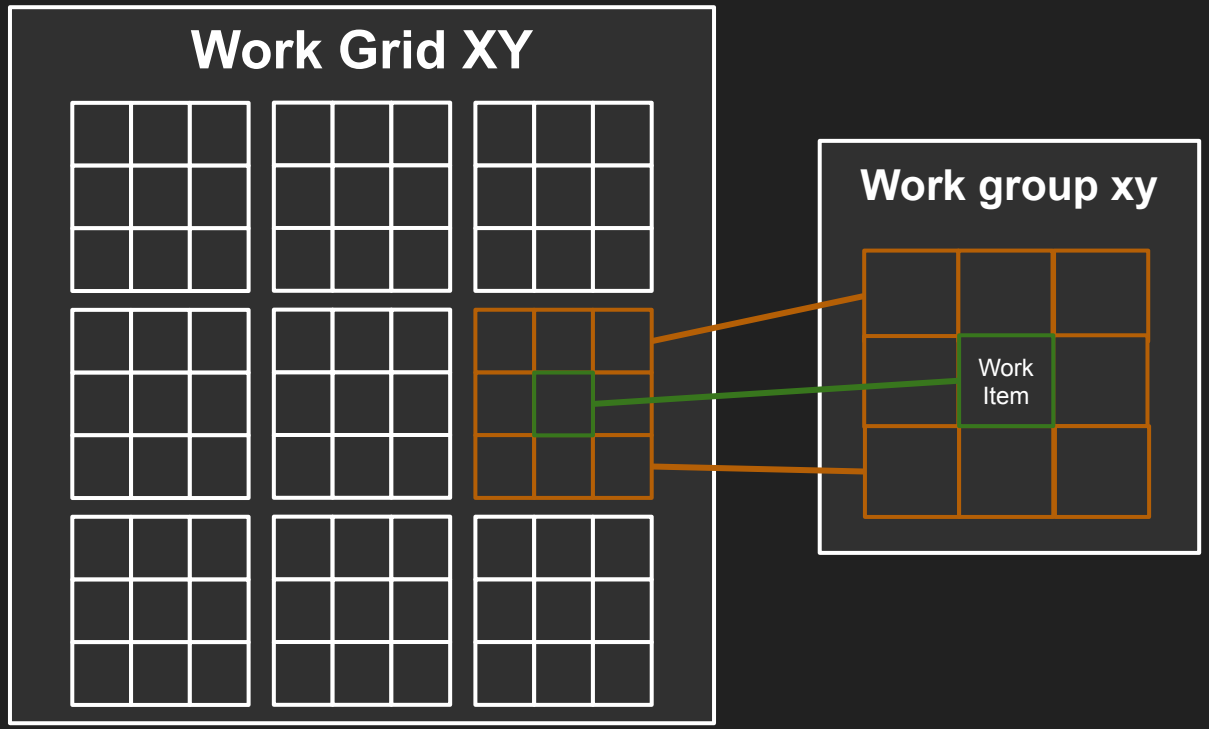
Platform model



Memory model



Execution model



- Total items
  X * Y
- Group size
  x * y
- Group count
  (X / x) * (Y / y)

Programming model

- Task-parallel processing
- Выполнение фиксированной функции над 1 элементом
- Запуск сетки потоков для параллельной обработки всех элементов

Базовые понятия

- 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 занимает N секунд 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

```
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 выполнение
- По умолчанию: идут в строгом порядке
- Выполнение на GPU не синхронизировано с хост-программой
- Требуются явные точки синхронизации

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

Выполнение

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

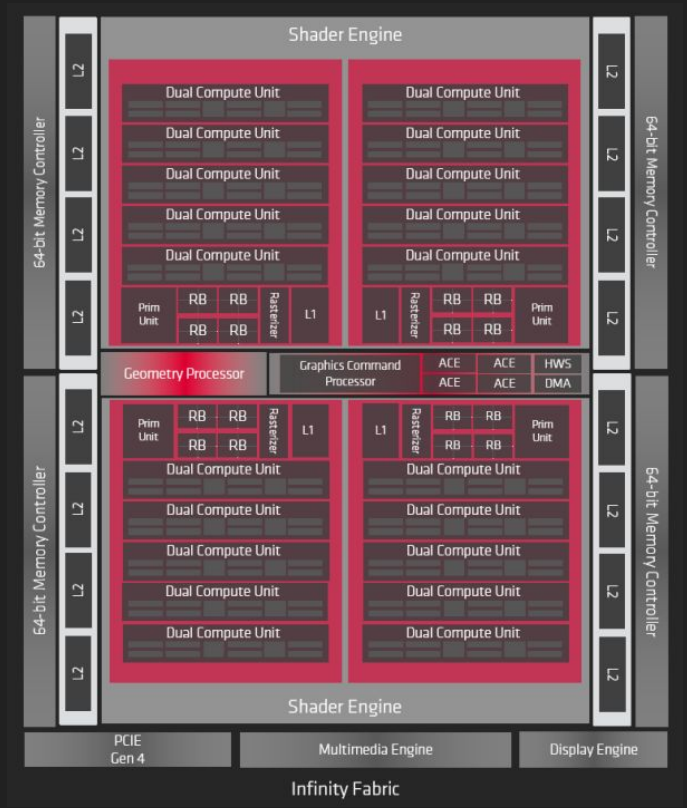
```
cl::NDRange global(N);
cl::NDRange local(32);

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

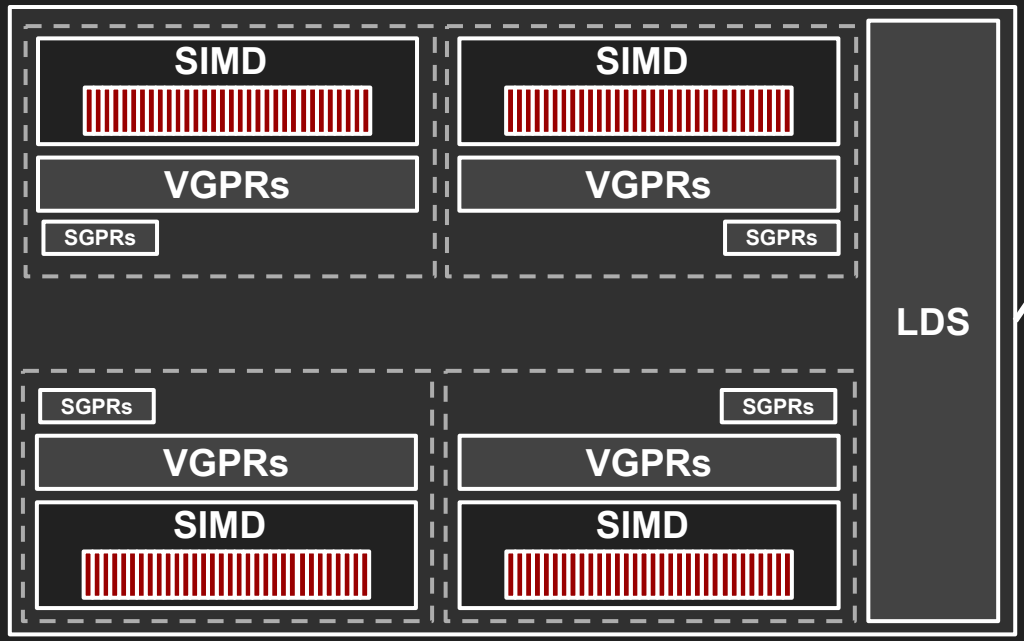
- Nvidia Pascal
- Nvidia Turing
- Nvidia Ampere
- Nvidia Ada Lovelace
- AMD GCN (Graphics Core Next)
- AMD RDNA (Radeon DNA)
- AMD RDNA-2

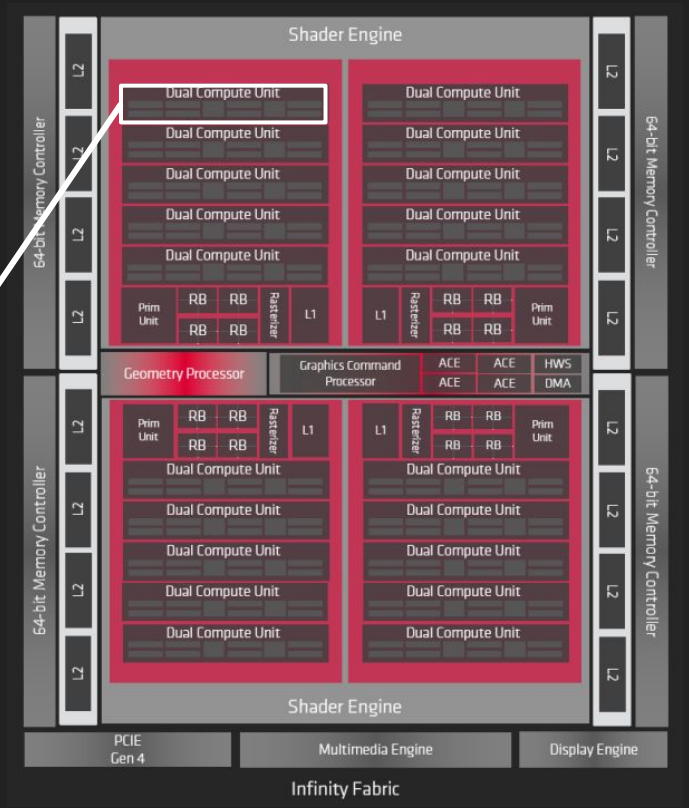
RDNA

- 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



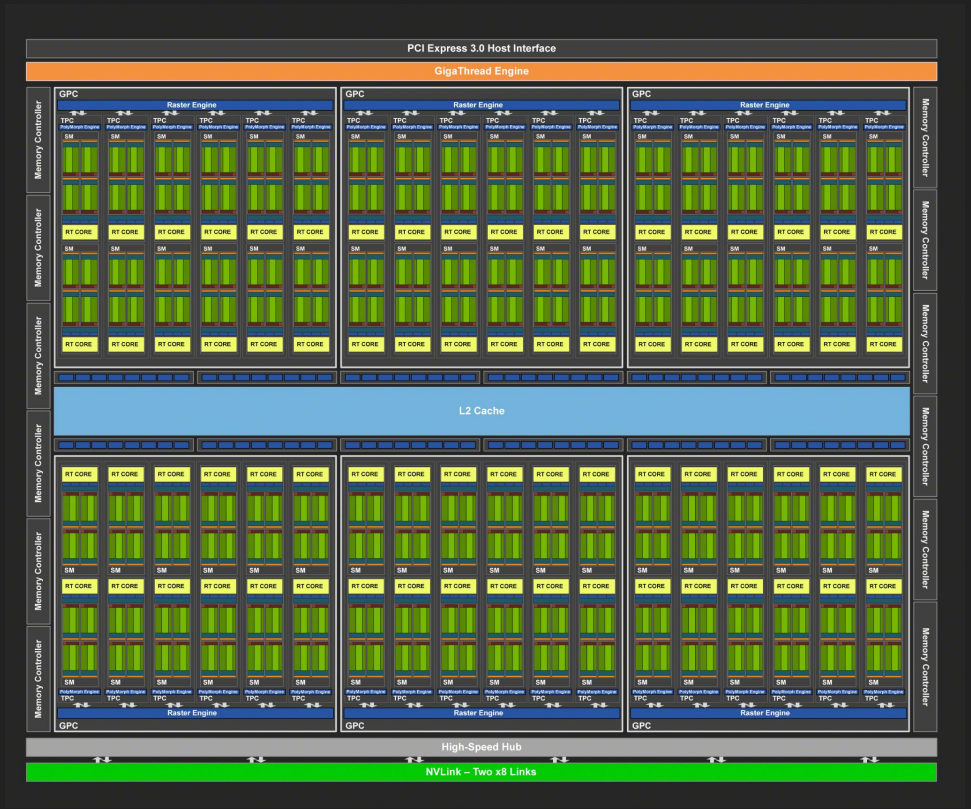
RDNA Compute Unit





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

Распределение работы

- *Quiz: Какой диспатч лучше?*

  `Grid (32, 32) Workgroup (32, 32)`

  `vs`

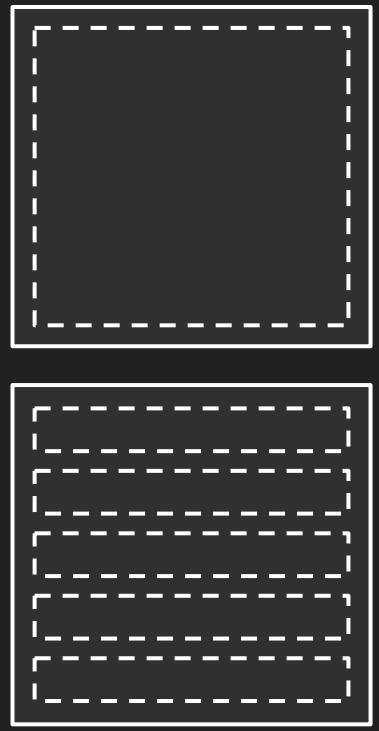
  `Grid (32, 32) Workgroup (32, 1)`

- *Quiz: Какой размер группы лучше?*

  `Workgroup (16, 1)`

  `vs`

  `Workgroup (32, 1)`



подсказка

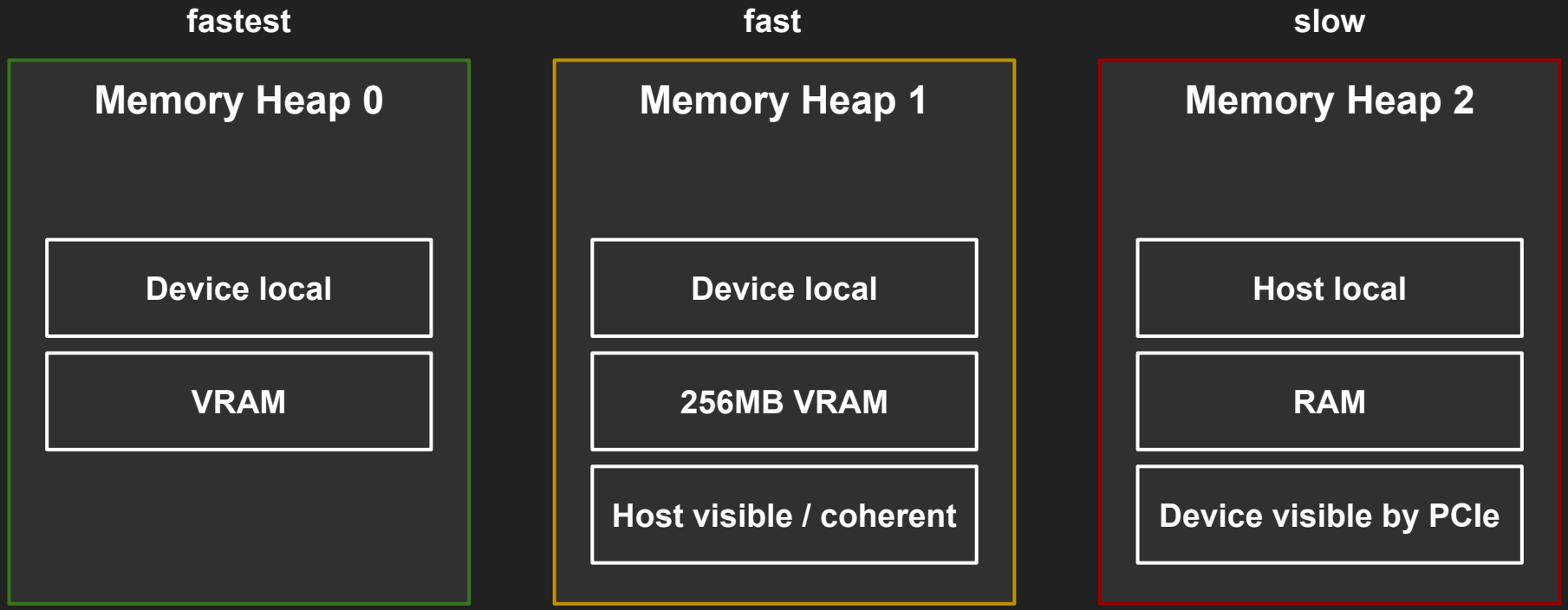
Использование локальной памяти

- 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 памяти

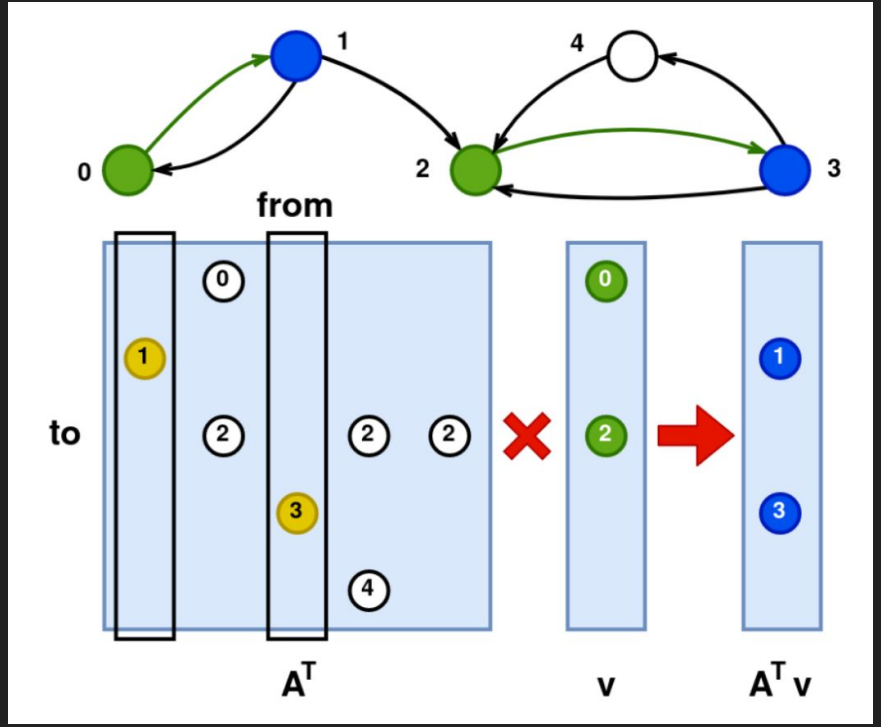

fastest
Memory Heap 0
Device local
VRAM
fast
Memory Heap 1
Device local
256MB VRAM
Host visible / coherent
slow
Memory Heap 2
Host local
RAM
Device visible by PCIe

Флаги памяти в OpenCL

- `CL_MEM_READ_WRITE` → по умолчанию
- `CL_MEM_WRITE_ONLY` → на GPU только запись
- `CL_MEM_READ_ONLY` → на GPU только чтение
- `CL_MEM_USE_HOST_PTR` → использовать RAM
- `CL_MEM_ALLOC_HOST_PTR` → аллоцировать в RAM
- `CL_MEM_COPY_HOST_PTR` → скопировать данные
- `CL_MEM_HOST_WRITE_ONLY` → доступ с CPU на запись только
- `CL_MEM_HOST_READ_ONLY` → доступ с CPU на чтение только
- `CL_MEM_HOST_NO_ACCESS` → нет доступа с CPU

Анализ графов и линейная алгебра

- Анализ графов
- Решение прикладных задач
- Абстракция
- Существующие операции
- Матрицы, вектора, скаляры
- Операции над матрицами
- Параметризация
- GraphBLAS стандарт



GraphBLAS

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

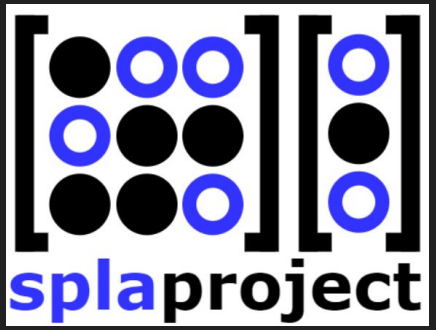
```
GrB_Vector_new(v, GrB_INT32, n);
GrB_Vector q;
GrB_Vector_new(&q, GrB_BOOL, n);
GrB_Vector_setElement(q, true, s);
int32_t level = 0;
GrB_Index nvals;
do {
      ++level;
      GrB_apply(*v,GrB_NULL,GrB_PLUS_INT32,GrB_SECOND_INT32,q,level, GrB_NULL);
      GrB_vxm(q,*v,GrB_NULL,GrB_LOR_LAND_SEMIRING_BOOL,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

Существующие проблемы

- Complex API
- No high-level portable package
- Lack of interoperability
- Blocking API
- Complicated value types management
- Little introspection
- Opaque objects
- No way to specify storage hints
- No GPU support
- Complicated masking patterns
- Missing operations
- Missing serialization

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



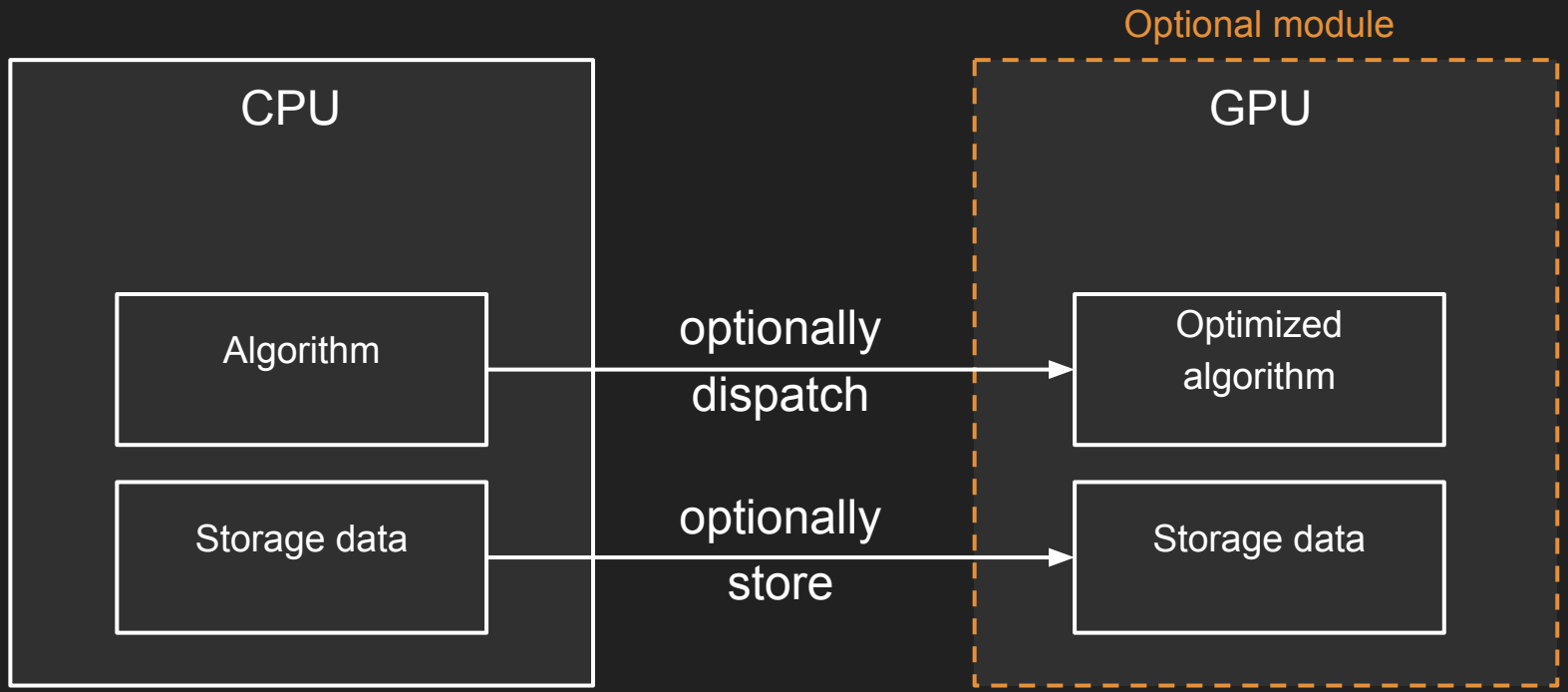
Spla: пример bfs

```
while (!frontier_empty) {
      depth->set_int(current_level);
      exec_v_assign_masked(v, frontier_prev, depth, SECOND_INT, NQZERO_INT);
      exec_mxv_masked(frontier_new, v, A, frontier_prev, BAND_INT, BOR_INT, EQZERO_INT, zero);
      exec_v_reduce(frontier_size, zero, frontier_new, PLUS_INT);

      int observed_vertices;
      frontier_size->get_int(observed_vertices);

      frontier_empty = observed_vertices == 0;
      current_level += 1;

      std::swap(frontier_prev, frontier_new);
}
```

Spla: подход

Complex API	→ Explicit API
No high-level portable package	→ Python package
Lack of interoperability	→ Decorations mechanism
Blocking API	→ Scheduling API
Complicated value types management	→ Storage type
Little introspection	→ Full introspection
Opaque objects	→ State inspection
No way to specify storage hints	→ Multiple formats
No GPU support	→ Optional agnostic GPUs ACC
Complicated masking patterns	→ Explicit select option
Missing operations	→ Easy to extend
Missing serialization	→ Easy to serialize

Spla: концепция



Ресурсы

- https://www.amd.com/system/files/documents/rdna-whitepaper.pdf
- https://man.opencl.org/
- 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/JetBrains-Research/spla
- https://docs.google.com/document/d/1fMmm-Bmew0wpgJRrjyMHy6G-zPq6R6kQlRum560_4S0/edit