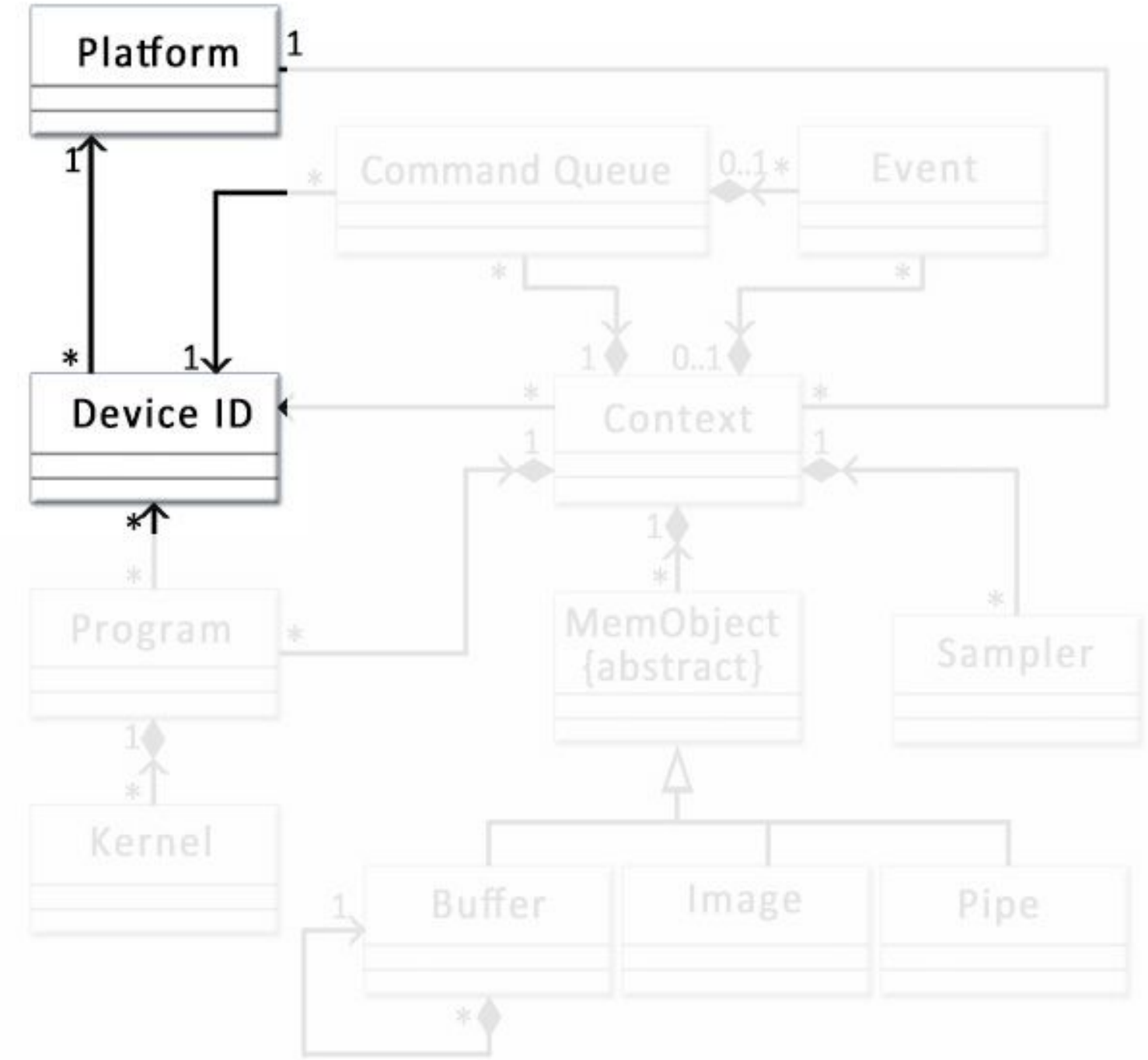
Введение в OpenCL. Архитектура видеокарты.

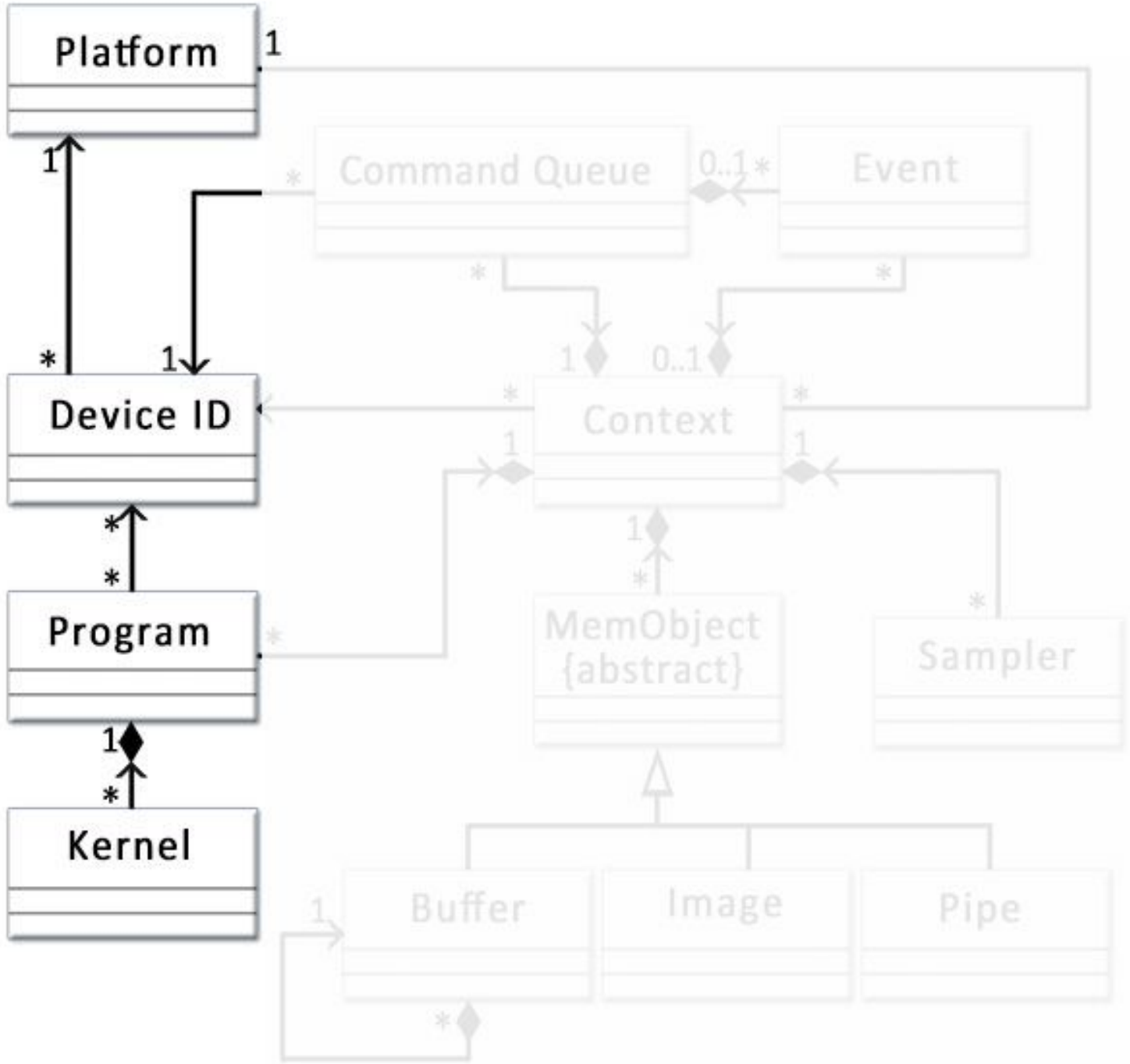
Вычисления на видеокартах. Лекция 2

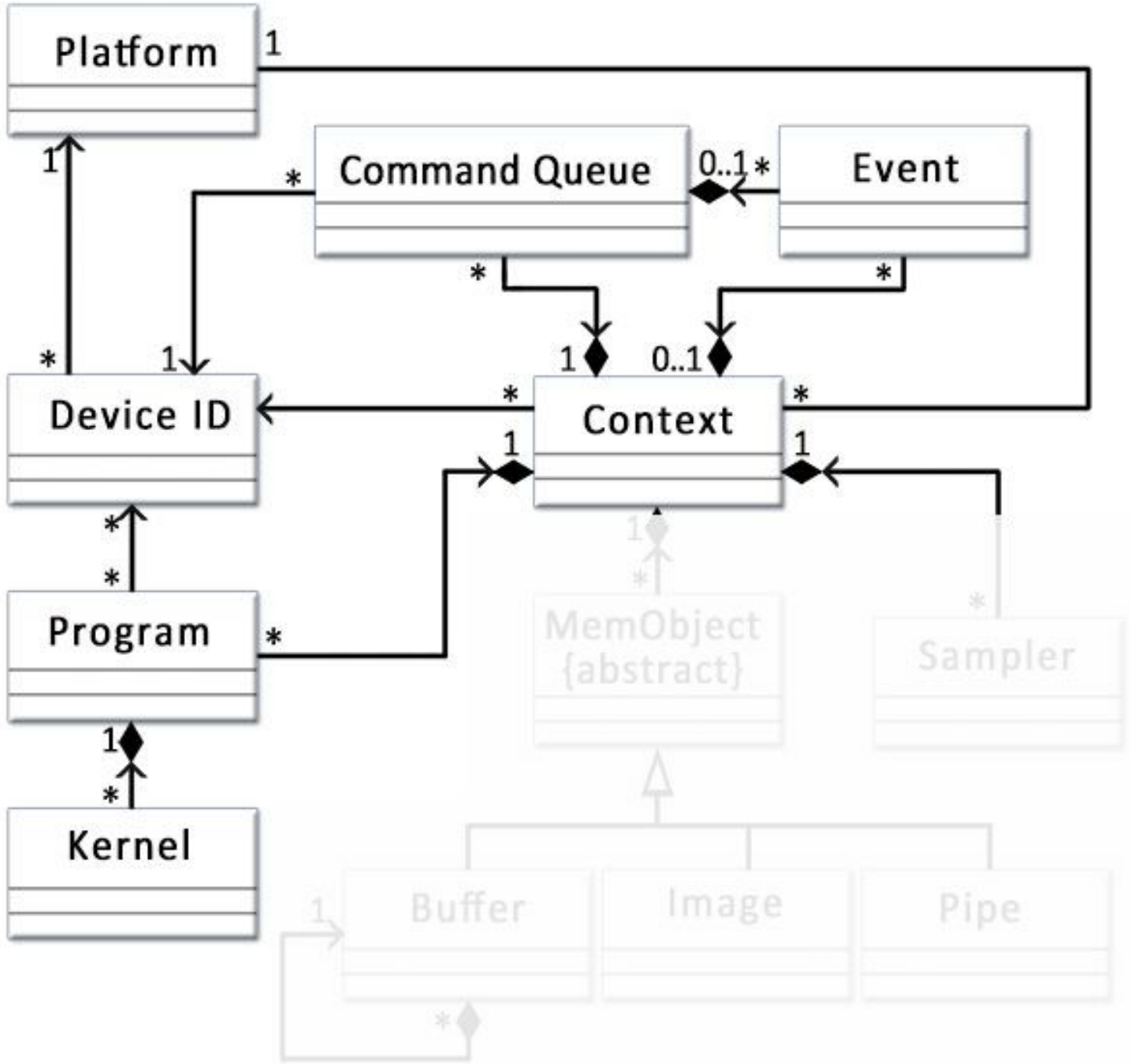
Warps/Wavefronts, code divergence, occupancy, registry spilling, coalesced memory access, banks conflicts.

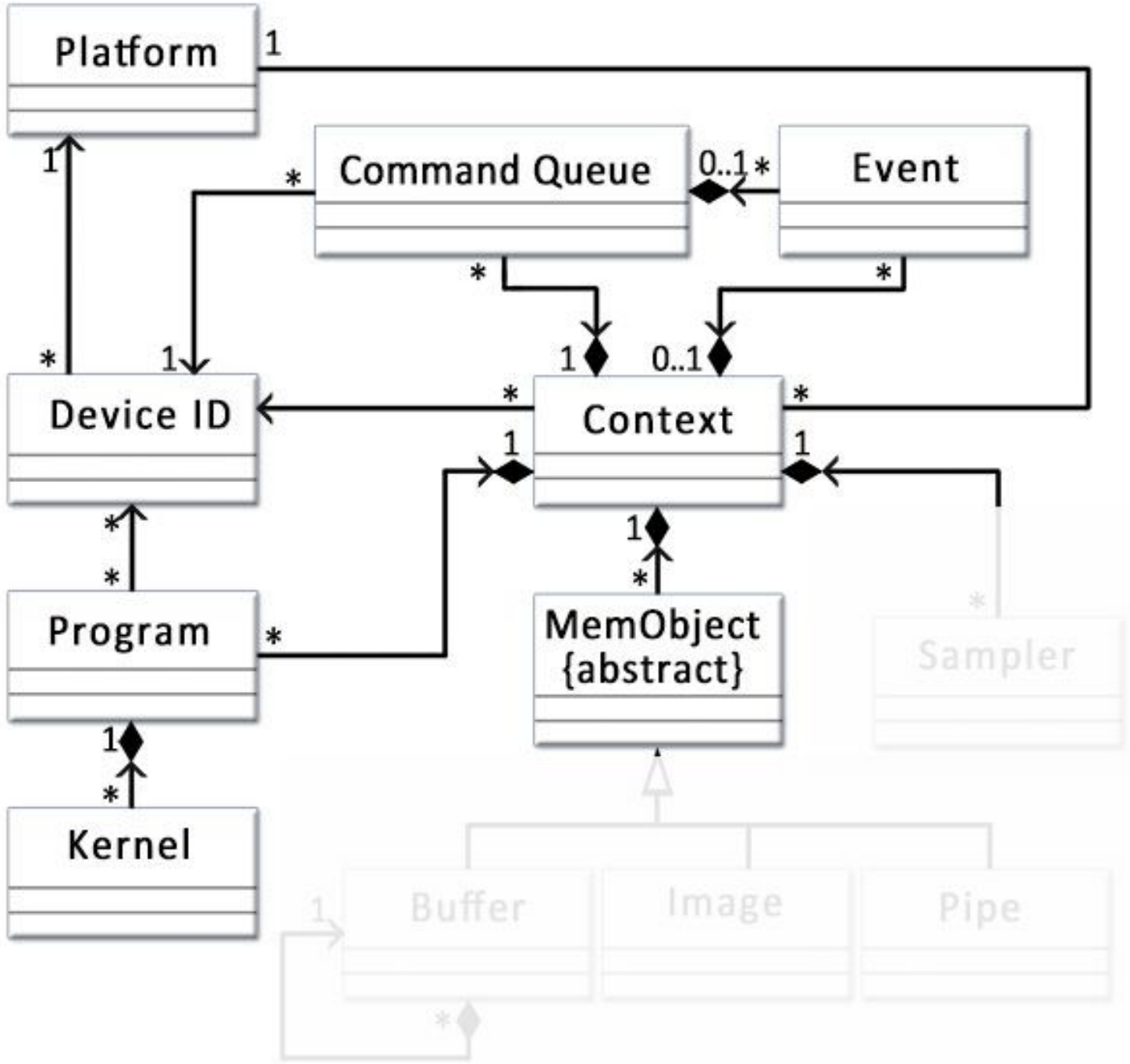
Полярный Николай

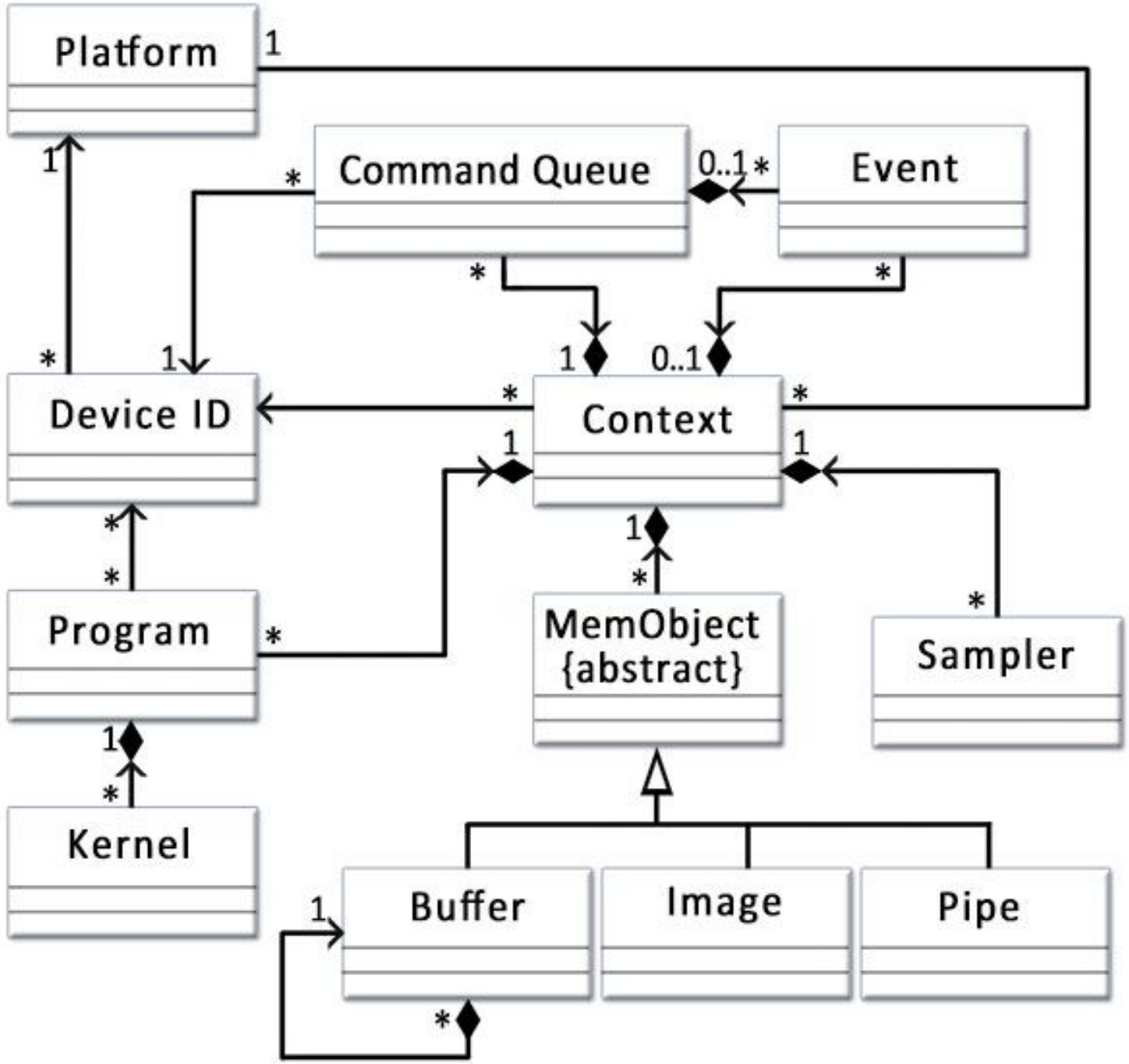
polarnick239@gmail.com


Platform
1
1
*
1
Device ID
*
Command Queue
0..1
*
Event
*
*
1
0..1
*
Context
*
1
1
1
*
*
Program
*
MemObject
{abstract}
Sampler
1
*
Kernel
1
Buffer
Image
Pipe
*


Platform
Command Queue
Event
Device ID
Context
Program
MemObject
{abstract}
Sampler
Kernel
Buffer
Image
Pipe


Platform
1
1
*
Command Queue
0..1
*
Event
*
*
*
1
1
0..1
*
Device ID
*
Context
*
1
1
*
*
Program
*
1
*
Kernel
MemObject
{abstract}
Sampler
Buffer
Image
Pipe
1
*


Platform
Command Queue
Event
Device ID
Context
Program
MemObject
{abstract}
Sampler
Kernel
Buffer
Image
Pipe
1
*
0..1


Platform
Command Queue
Event
Device ID
Context
Program
MemObject
{abstract}
Sampler
Kernel
Buffer
Image
Pipe
1
*
0..1

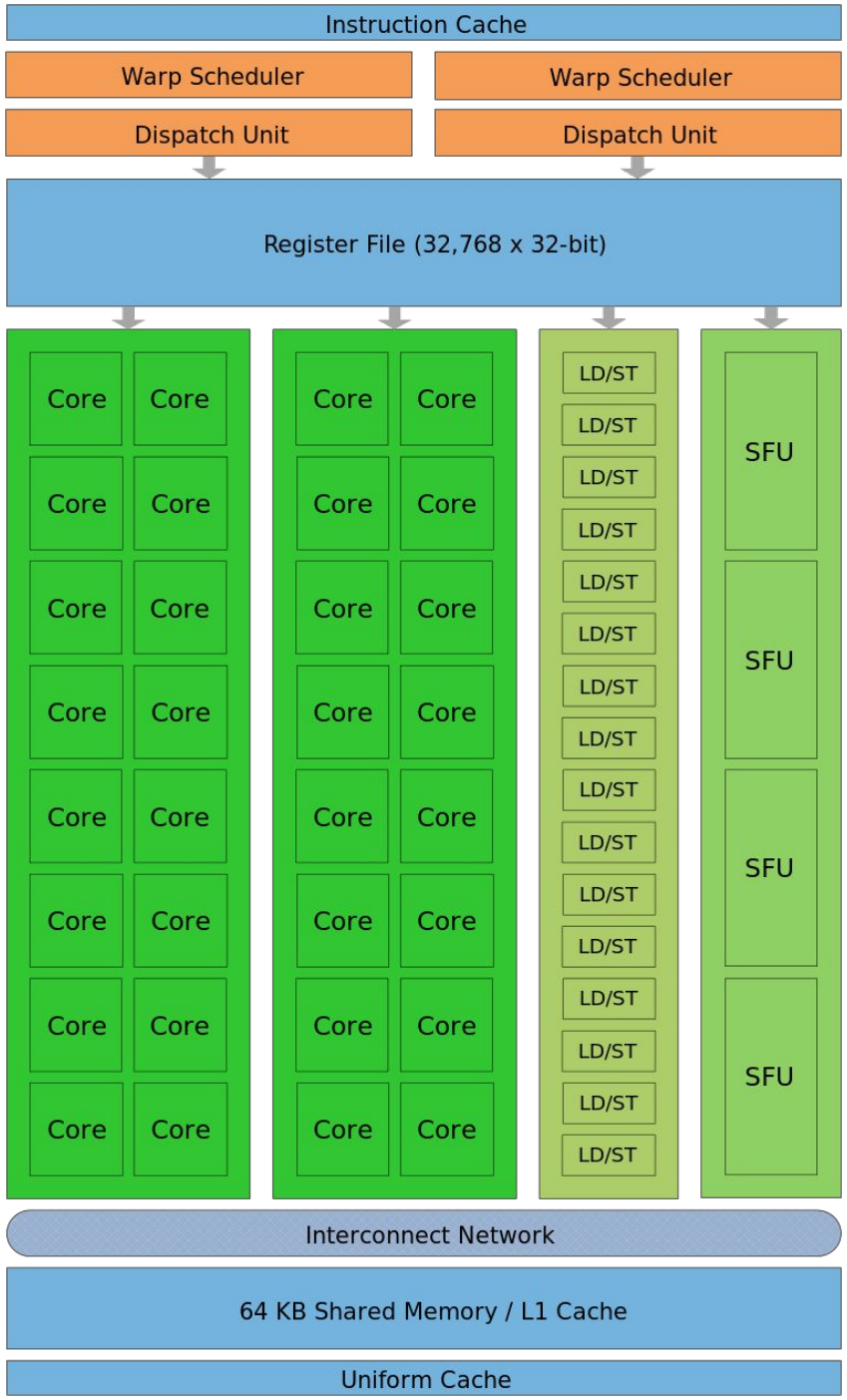
Пример kernel

```
__kernel void aplusb(__global const float* a,
                     __global const float* b,
                     __global       float* c,
                     unsigned int n)
{
    const unsigned int index = get_global_id(0);

    if (index >= n)
        return;

    c[index] = a[index] + b[index];
}
```

Примеры вызовов и локальной памяти

```
size_t  get_global_size     (uint dimindx);
size_t  get_global_id       (uint dimindx);
size_t  get_local_size      (uint dimindx);
size_t  get_local_id        (uint dimindx);
size_t  get_num_groups      (uint dimindx);
size_t  get_group_id        (uint dimindx);
size_t  get_global_offset   (uint dimindx);
uint    get_work_dim        ();
```

```
__local int local_memory_array[128];

barrier(CLK_LOCAL_MEM_FENCE);
```



NVIDIA: **32** threads in warp (**32**x**Core** слева)

AMD: **64** threads in wavefront

L1 Cache это то же самое что и:
= **Local Memory** (терминология **OpenCL**)
= Shared Memory (терминология **CUDA**)

Потоки в одном warp/wavefront имеют общий указатель на исполняемую инструкцию. Поэтому **code divergence** - частая причина низкой производительности:

```
if (predicate) {
    value = x[i];
} else {
    value = y[i];
}
```

Occupancy и registers spilling

Большая пропускная способность достигается сокрытием latency.

Это достигается большим количеством запущенных work groups. При исполнении запроса к памяти случившаяся latency в текущей рабочей группе прячется переключением вычислений на другую work group.

Чем больше на одном вычислителе (streaming multiprocessor/computation unit/SIMD unit) рабочих групп (это и есть **occupancy**, т.е. занятость/заполненность) - тем реже все рабочие группы оказываются в состоянии “ждем запрос памяти” и тем реже вычислитель будет простаивать, т.к. тем чаще у него находится рабочая группа в которой можно что-то посчитать.

Более точно:
Occupancy = ЧислоАктивных / МаксимальноеЧислоАктивных рабочих групп

Occupancy и registers spilling

Количество рабочих групп которые могут быть запущены на одном computation unit это минимум из:

- Количество **регистров** / количество используемых регистров в kernel
- Количество **Local memory** / количество используемой Local memory
- Максимальное допустимое количество рабочих групп (~10)

Так же зависит от кратности размера рабочей группы на размер warp/WaveFront. Поэтому рекомендуется делать размер рабочей группы кратным 64.

Можно сэкономить используемые регистры выгрузив их в глобальную память - это называется **registers spilling**. Может позволить достичь лучшей **occupancy** обменяв часть регистров на медленную память.

Occupancy > 60% - уже достаточно хорошо.
Стоит пытаться ее увеличить только если кернел **memory-bound**.
(**перед оптимизациями надо профилировать**)

Coalesced memory access

Если потоки из одного warp/wavefront делают запрос к памяти, то эти запросы склеются в столько запросов, сколькими кеш-линиями покрываются запрошенные данные.

Иначе говоря:

Если потоки запрашивают одновременно данные которые лежат подряд - то достигнутая пропускная способность будет максимальной, т.к. такие запросы склеются.

Если же потоки запрашивают далекие друг от друга данные (с расстоянием большим чем кеш-линия) - то достингутая пропускная способность будет минимальна.

Размер кеш линии ориентировочно от 32 байт до 128 байт.

Banks conflicts (Local memory/L1)

Если все потоки в warp/wavefront обращаются **к одному и тому же значению** в local memory - значение считается быстро за одну операцию (произойдет broadcast).

Banks conflicts (Local memory/L1)

Local memory состоит из **memory banks**. К каждой memory bamk одновременно может обратиться лишь один поток.

Поэтому если все потоки в warp/wavefront обращаются к **разным memory banks** - обращения произойдут **параллельно**, т.е. быстро.

Если потоки в warp/wavefront обращаются к **одинаковым memory banks** - эти обращения сериализуются и будут исполнены **последовательно**, т.е. медленно.

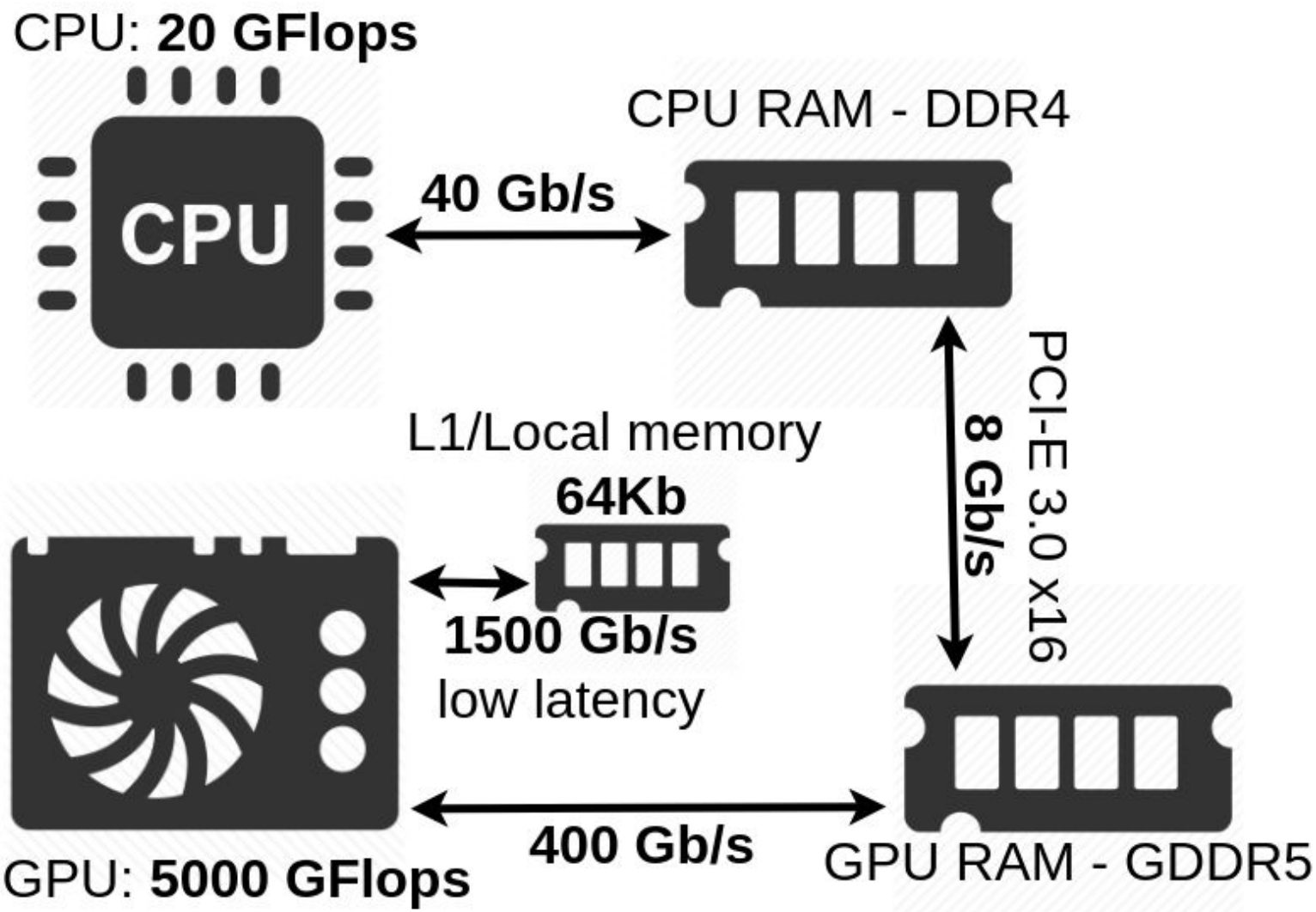
Banks conflicts (Local memory/L1)

В зависимости от устройства **Local memory** разбита ориентировочно на **32 memory banks**.

Адресное пространство **Local memory** распределено по ним с шагом в **4 байта**.

```
Bank    |      1      |      2      |      3      |...
Address |  0  1  2  3 |  4  5  6  7 |  8  9 10 11 |...
Address | 64 65 66 67 | 68 69 70 71 | 72 73 74 75 |...
...
```

Ресурсы


CPU: 20 GFlops
CPU
40 Gb/s
CPU RAM - DDR4
PCI-E 3.0 x16
8 Gb/s
L1/Local memory
64Kb
1500 Gb/s
low latency
400 Gb/s
GPU: 5000 GFlops
GPU RAM - GDDR5

```
int sum_cpu(int xs[n], int n) {
    int sum = 0;
    for (int i = 0; i < n; ++i) {
        sum += xs[i];
    }
    return sum; // Найти сумму чисел
}
```

```
__kernel void sum_gpu_1(__global const int* xs, int n,
                        __global int* res) {
    int id = get_global_id(0);
    res[0] += xs[id];
}
```

```
__kernel void sum_gpu_2(__global const int* xs, int n,
                        __global int* res) {
    int id = get_global_id(0);
    atomic_add(res, xs[id]);
}
```

```
#define VALUES_PER_WORK_ITEM 64
__kernel void sum_gpu_3(__global const int* xs, int n,
                        __global int* res) {
    int id = get_global_id(0);

    int sum = 0;
    for (int i = 0; i < VALUES_PER_WORK_ITEM; ++i) {
        sum += xs[id * VALUES_PER_WORK_ITEM + i];
    }
    atomic_add(res, sum);
}
```

```
#define VALUES_PER_WORK_ITEM 64
__kernel void sum_gpu_4(__global const int* xs, int n,
                        __global int* res) {
    int localId = get_local_id(0);
    int groupId = get_group_id(0);
    int groupSize = get_group_size(0);

    int sum = 0;
    for (int i = 0; i < VALUES_PER_WORK_ITEM; ++i) {
        sum += xs[groupId*groupSize*VALUES_PER_WORK_ITEM
                  + i*groupSize + localId];
    }
    atomic_add(res, sum);
}
```

```
#define WORK_GROUP_SIZE 256
__kernel void sum_gpu_5(__global const int* xs, int n,
                        __global int* res) {
    int localId = get_local_id(0);
    int globalId = get_global_id(0);

    __local int local_xs[WORK_GROUP_SIZE];
    local_xs[localId] = xs[globalId];

    int sum = 0;
    for (int i = 0; i < WORK_GROUP_SIZE; ++i) {
        sum += local_xs[i]; // <- Ошибка 1
    }
    atomic_add(res, sum); // <- Ошибка 2
}
```

```
#define WORK_GROUP_SIZE 256
__kernel void sum_gpu_5(__global const int* xs, int n,
                        __global int* res) {
    int localId = get_local_id(0);
    int globalId = get_global_id(0);

    __local int local_xs[WORK_GROUP_SIZE];
    local_xs[localId] = xs[globalId];

    barrier(CLK_LOCAL_MEM_FENCE); // <- Дождались всех
    if (localId == 0) {
        int sum = 0;
        for (int i = 0; i < WORK_GROUP_SIZE; ++i) {
            sum += local_xs[i];
        }
        atomic_add(res, sum); // <- Только localId==0
    }
}
```

```
#define WORK_GROUP_SIZE 256
__kernel void sum_gpu_6(__global const int* xs, int n,
                        __global int* res) {
    int localId = get_local_id(0);
    int globalId = get_global_id(0);

    __local int local_xs[WORK_GROUP_SIZE];
    local_xs[localId] = xs[globalId];

    barrier(CLK_LOCAL_MEM_FENCE);
    for (int nvalues = WORKGROUP_SIZE; nvalues > 1; nvalues /= 2) {
        if (2 * localId < nvalues) {
            int a = values[localId];
            int b = values[localId + nvalues/2];
            values[localId] = a + b;
        }
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    if (localId == 0) {
        atomic_add(res, local_xs[0]);
    }
}
```