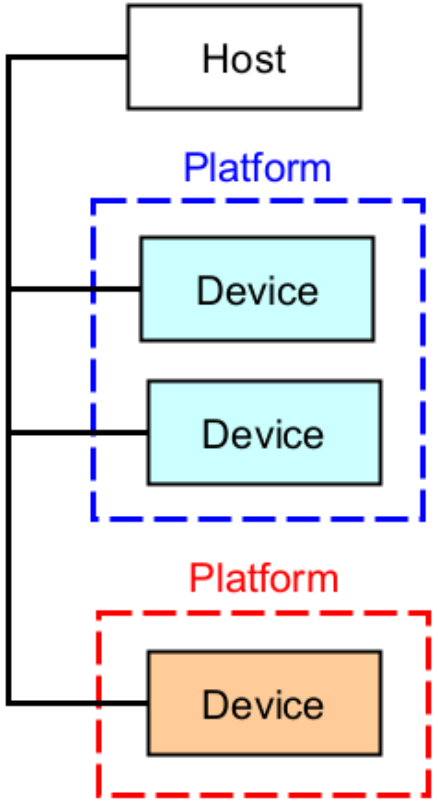
MEMORY FOR OFFLOADS

Память как концепция в гетерогенном программировании

К. Владимиров, МФТИ, 2022
mail-to: konstantin.vladimirov@gmail.com

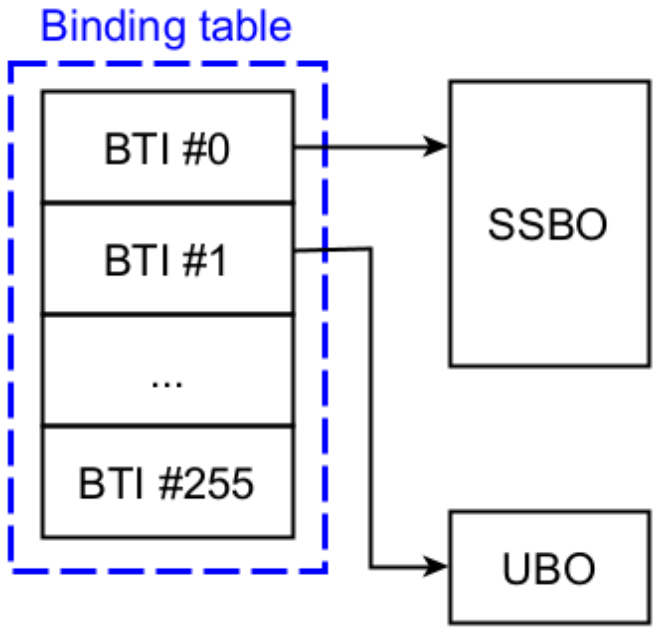
Гетерогенные вычисления

- Две выделенных роли: **host** и **device**.
- Host – это устройство, которое инициирует вычисления.
- Device – это устройство, на котором производятся вычисления.
- Примеры устройств:
  - Видеокарты и графические ускорители.
  - Карты и ускорители для машинного обучения.
  - Другие CPU помельче и даже тот же самый CPU.
- Исторически первым опытом человечества в гетерогенном программировании стали видеокарточки.



Stateful память в GLSL: binding table

```
struct Particle { vec4 pos, vel; };

layout(std140, binding = 0) buffer Pos {
  Particle particles[];
};

layout (binding = 1) uniform UBO {
  float deltaT;
  int count;
} ubo;

for (int i = 0; i < ubo.count; i += DataSz)
  if (i + x < ubo.count)
    someData[x] = particles[i + x].pos;
```



Particle System Compute Shader

Binding table и структура указателя

- "Указатель" particles, таким образом, – это индекс в BTI и смещение в буфере.

```
for (int i = 0; i < ubo.count; i += DataSz)
  if (i + x < ubo.count)
    someData[x] = particles[i + x].pos;
```

- То, что выглядит как доступ к массиву, является доступом к памяти через два уровня.
- Такая память называется **stateful** (пример state – это базовый адрес в памяти).
- В таких языках, как GLSL, просто нет концепции **stateless** (настоящего) указателя.

Настоящего?

- Слово "настоящий" в отношении указателя может показаться казуистикой.
- Но давайте посмотрим на вот такой простой пример.

```
int *uboPtr = &ubo.deltaT; // так сделать нельзя
int *ssboPtr = &particles[i + x].pos; // и так тоже
int *somePtr = (CTUnknownValue > 5) ? ssboPtr : uboPtr;
*somePtr = 8; // store в какой буфер? По какому индексу?
```

- По определению BTI задаётся именем буфера. Stateful pointer – это индекс.

```
int * addrspace(binding(1)) ssboPtr = offset(ubo.deltaT);
```

Typed и untyped память

- Концепция stateful памяти есть и в compute языках при работе со сложно структурированными изображениями.

```
accessor<float4, 2, sycl_read, sycl_image> Img(Src, Cgh);
sampler Sampler(unnormalized, clamp, nearest);
....
Value = Img.read(Coords, Sampler);
```

- Здесь state – это уже не только базовый адрес, но и детали Image: формат, длина и ширина, питч. По определению typed память является stateful.
- Напротив, обычный буфер – это **untyped**, и он может быть и stateful, и stateless.

clCreateImage2D

Separate-source системы

- Все графические API separate-source по уважительным причинам: язык GLSL это не C, там вся память stateful и нет unstructured control flow.
- OpenCL является C-based и тоже separate-source. Ну и вы понимаете….

```
const char *vakernel = "__kernel void vector_add("
" __global int *A, __global int *B, __global int *C) {"
" int i = get_global_id(0);"
" C[i] = A[i] + B[i];"
"}";
```

- Интересный вопрос: чему равен sizeof(int) и сколько вы будете отлаживать ошибку, если он не совпадёт с хостовым?

cl_vectoradd.cc

Design goal: no type unsafety in API calls

- В стиле Шона Парента мы можем объявить design goal.

> No raw loops.
> No raw synchronization primitives.

- Я предлагаю, чтобы наша цель звучала так.

> No type unsafety in API calls.

- Как и в случае с остальными принципами проектирования, этот не так прост…

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), buffer);
```

Design goal: no type unsafety in API calls

- В стиле Шона Парента мы можем объявить design goal.

> No raw loops.
> No raw synchronization primitives.

- Я предлагаю, чтобы наша цель звучала так.

> No type unsafety in API calls.

- Как и в случае с остальными принципами проектирования, этот не так прост…

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), buffer);
```

- Но, кстати, он касается, конечно же, не только OpenCL.

```
memcpy(dest, src, num_elements * sizeof(int));
```

Подходы к single-source API

- Мы можем расширить C++ новыми ключевыми словами и скобками.

```
__global__ void vecAdd(int *A, int *B, int *C, int N);

....

cudaMalloc((void **)&a_d, n * sizeof(int));
cudaMalloc((void **)&b_d, n * sizeof(int));
cudaMalloc((void **)&c_d, n * sizeof(int));
vecAdd<<<block_no, block_size>>>(a_d, b_d, c_d, n);
```

- У нас есть и другие варианты….

Подходы к single-source API

- Мы можем расширить C++ новыми ключевыми словами и скобками.

- Мы можем расширить C++ новыми прагмами.

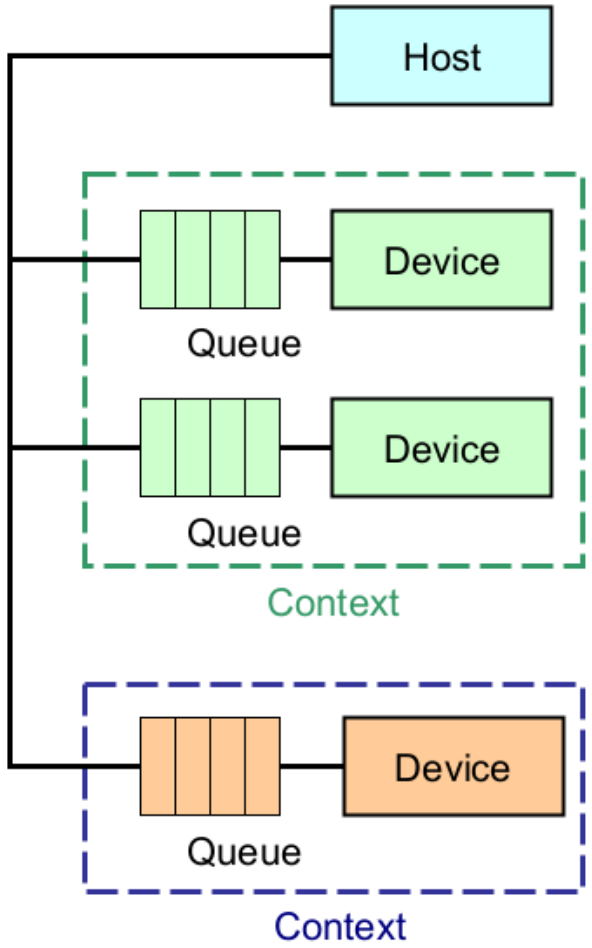
```
#pragma omp target parallel private(ompSum) shared(totalSum)
{
  ompSum = 0;

  #pragma omp for
  for (int i = 0; i < N; i++)
    ompSum += array[i];

  #pragma omp critical
  totalSum += ompSum;
}
```

Подходы к single-source API

- Мы можем расширить C++ новыми ключевыми словами.
- Мы можем расширить C++ новыми прагмами.
- Мы можем попробовать обойтись обычным C++.

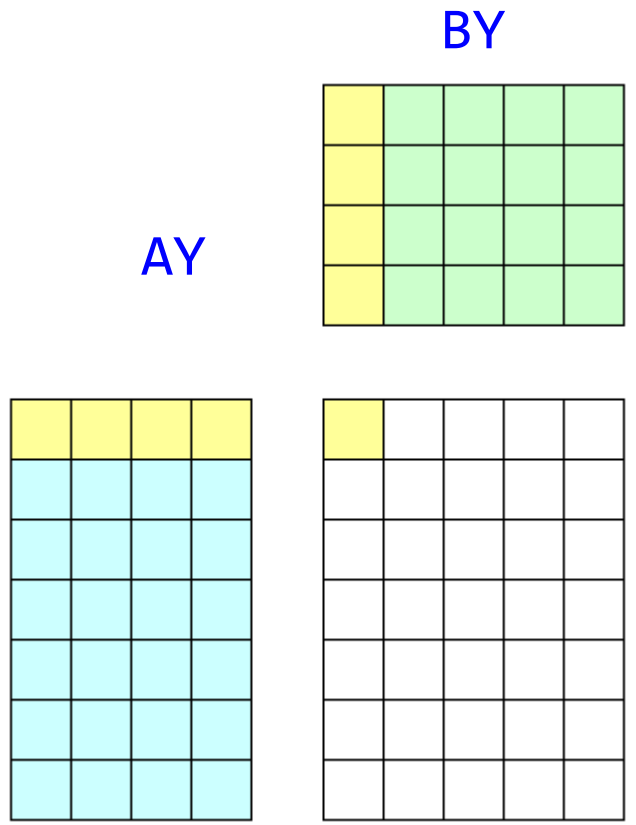
```
auto Event = Queue.submit(
  [&](sycl::handler &Cgh) {
    .....
    Cgh.parallel_for<name>(Range, DeviceFn);
  }
);
```

- Такая программа на **SYCL** может быть скомпилирована как обычная C++ программа.



Glorious GEMM example

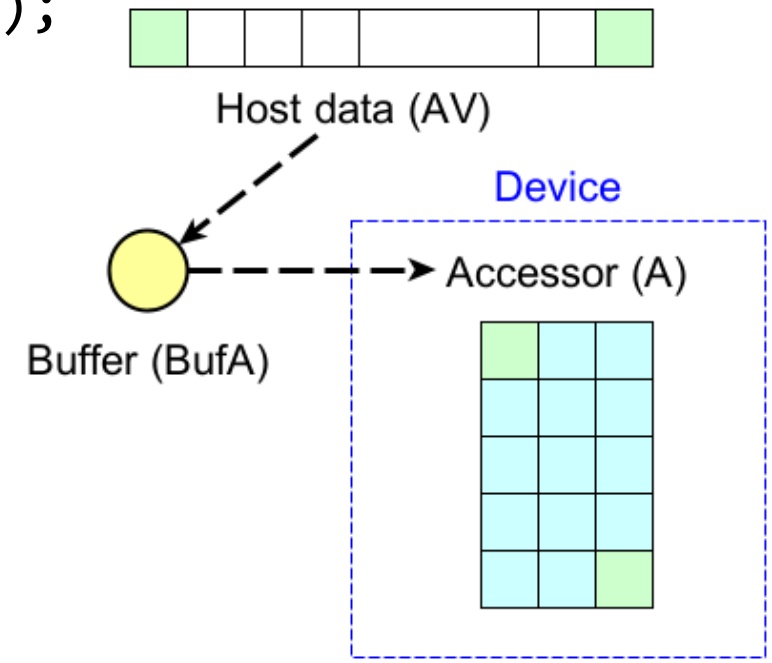
```
Queue.submit([&](sycl::handler &Cgh) {
  ......
  auto K = [=](sycl::id<2> Id) {
    const int Row = Id.get(0);
    const int Col = Id.get(1);
    T Sum = 0;

    for (int K = 0; K < AY; K++)
      Sum += A[Row][K] * B[K][Col];

    C[Row][Col] = Sum;
  };

  Cgh.parallel_for<class m<T>>({AX, BY}, K);
}
```



matmult.cc

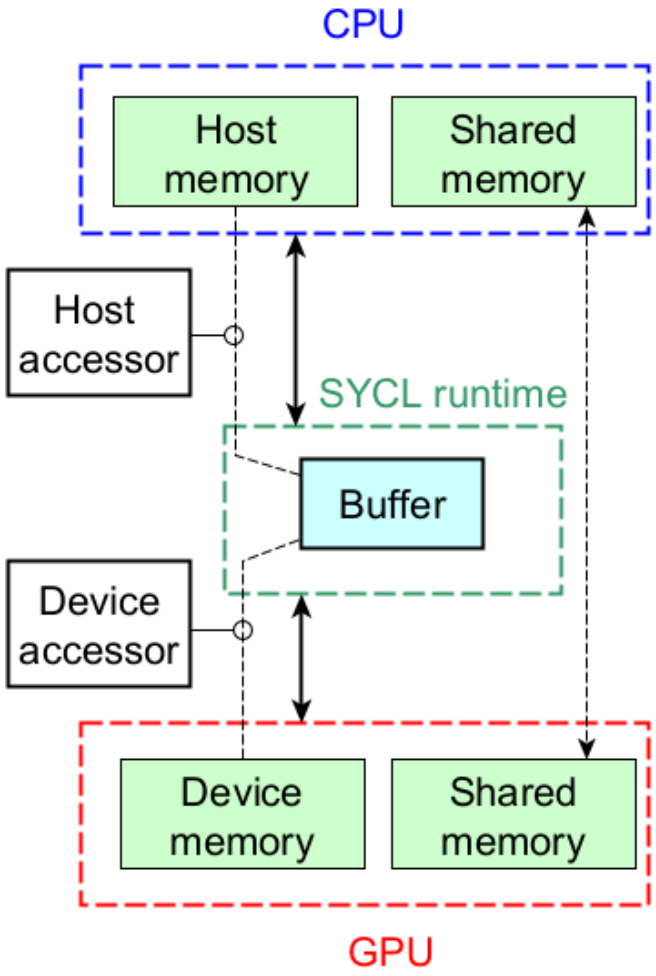
Accessors: пересылка на устройство

```
std::vector<T> AH;
.....
sycl::buffer<T, 2> BufA(AH.data(), AH.size());
.....
Queue.submit([&](sycl::handler &Cgh) {
  auto A = BufA.get_access<sycl_read>(Cgh);
  .....
  auto Kernmul = [=](sycl::id<2> Id) {
    int Row = Id.get(0), Col = Id.get(1);
    T Sum = 0;

    for (int K = 0; K < AY; K++)
      Sum += A[Row][K] * B[K][Col];
```



Shared memory: неявная пересылка

```
std::vector<T> AH;
.....
T *A = malloc_shared<T>(AX * AY, Queue);
std::copy(AH.begin(), AH.end(), A);
.....
Queue.submit([&](sycl::handler &Cgh) {

  auto Kernmul = [=](sycl::id<2> Id) {
    int Row = Id.get(0), Col = Id.get(1);
    T Sum = 0;

    for (int K = 0; K < AY; K++)
      Sum += A[Row * AY + K] * B[K * AY + Col];
```



Device memory: полный контроль.

- Мы можем максимально приблизиться к clEnqueueCopyBuffer.

```
std::vector<T> AH;
.....
T *A = malloc_device<T>(AX * AY, Queue);
auto EvtCpyData = Queue.copy(AH.data(), A, AH.size());
.....
Queue.submit([&](sycl::handler &Cgh) {
   Cgh.depends_on(EvtCpyData);
   auto Kernmul = [=](sycl::id<2> Id) { .....
```

- Обратим внимание: мы в явном виде построили task graph. Аксессоры строят его автоматически.

matmult_device.cc

Всё это – разновидности device memory

- Допустим, у нас есть буфер.

```
sycl::buffer<T, 2> BufA(AH.data(), AH.size());
```

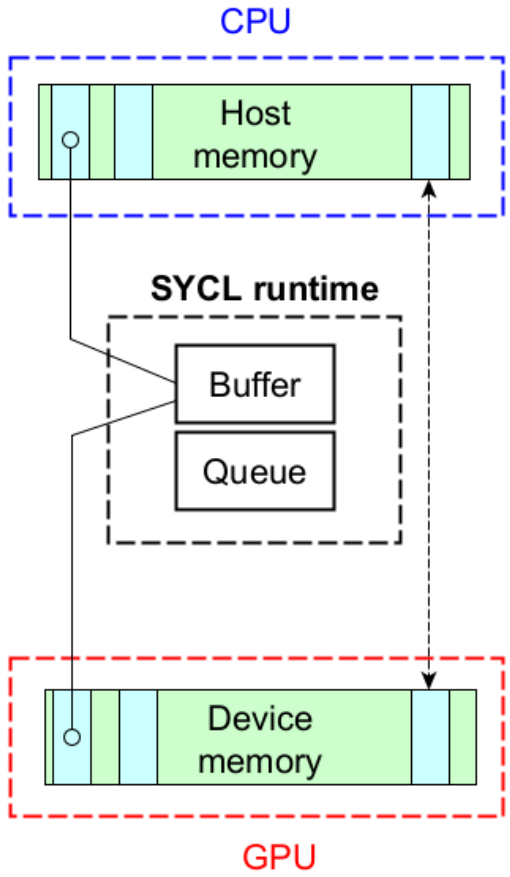
- Первый вариант: используем память по умолчанию.

```
auto A = BufA.template get_access<read>(Cgh);
```

- Второй вариант: явно указываем, что хотим device memory.

```
using AccTy = accessor<T, 2, read, device>;

AccTy A{BufA, Cgh};
```

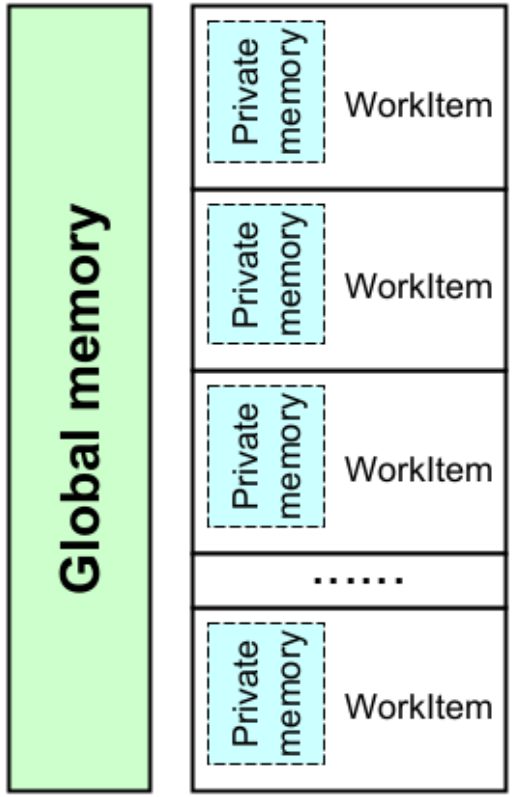
- Чтобы memory sharing был возможен, device memory должна быть **stateless** (т.е. "настоящей").



Идея приватной памяти

```
// K, Row, Col и Sum в приватной памяти
const int Row = Id.get(0);
const int Col = Id.get(1);
T Sum = 0;

// A[x][y], B[y][z] и C[x][z] в глобальной
for (int K = 0; K < AY; K++)
  Sum += A[Row][K] * B[K][Col];

C[Row][Col] = Sum;
```

- Приватная память задумана как быстрое и близкое хранилище небольших данных.

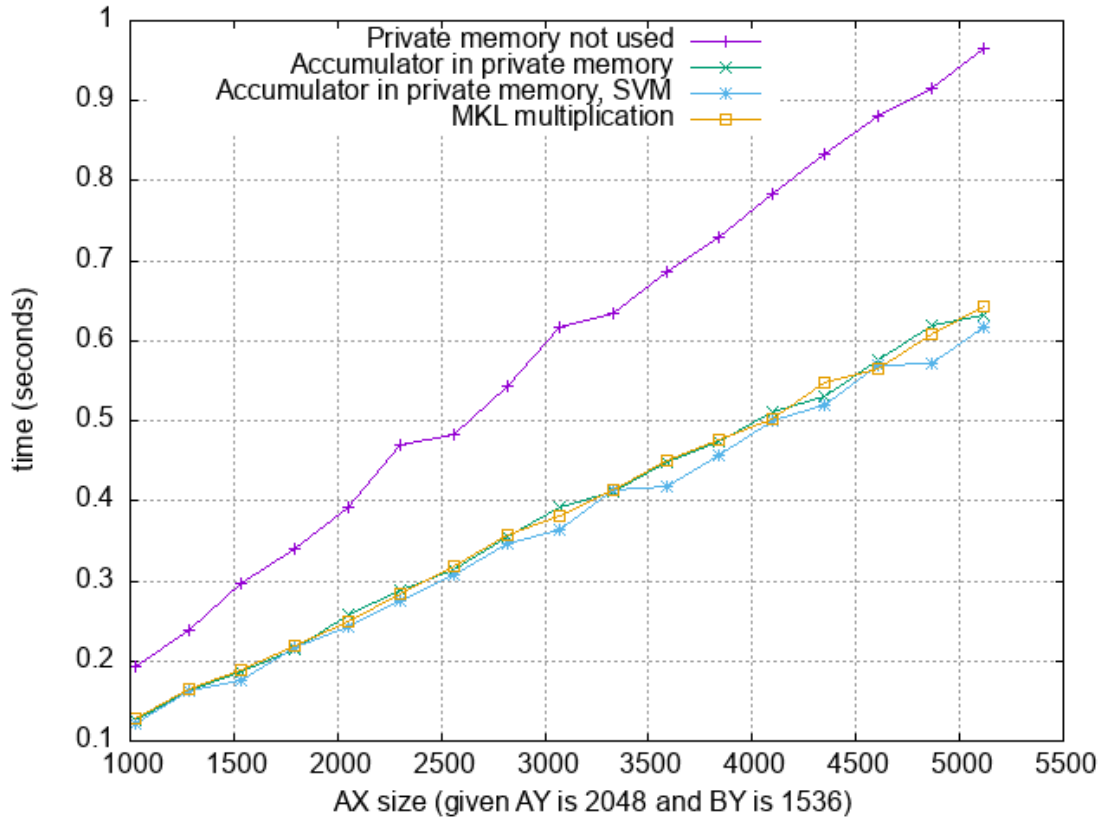


Перемножение средствами oneMKL

```
std::vector<sycl::event> GemmDependencies;
float Alpha = 1.0f, Beta = 0.0f;
mkl::transpose TransA = mkl::transpose::nontrans;
mkl::transpose TransB = mkl::transpose::nontrans;

auto *A = sycl::malloc_shared<T>(AX * AY, DeviceQueue);
auto *B = sycl::malloc_shared<T>(AY * BY, DeviceQueue);
auto *C = sycl::malloc_shared<T>(AX * BY, DeviceQueue);
std::copy(Aptr, Aptr + AX * AY, A);
std::copy(Bptr, Bptr + AY * BY, B);

mkl::blas::gemm(DeviceQueue, TransA, TransB, AX, BY, AY,
  Alpha, A, AX, B, AY, Beta, C, AX, GemmDependencies);
```


matmult_mkl.cc

Замеры с приватной памятью



- Вариант с приватной памятью.

```
T Sum = 0;

for (int K = 0; K < AY; K++)
  Sum += A[Row][K] * B[K][Col];

C[Row][Col] = Sum;
```

- Вариант без неё.

```
for (int K = 0; K < AY; K++)
  C[Row][Col] +=
    A[Row][K] * B[K][Col];
```


Optimization outside atomic

Приватная память – это регистры

```
label5:
  ...
  gather4_scaled.R (M1, 16) T6 0x0:ud V192.0 V194.0
  gather4_scaled.R (M5, 16) T6 0x0:ud V193.0 V195.0
  mad (M1, 16) V116(0,0)<1> V194(0,0)<1;1,0>
               V175(0,0)<1;1,0> V116(0,0)<1;1,0>
  mad (M5, 16) V117(0,0)<1> V195(0,0)<1;1,0>
               V176(0,0)<1;1,0> V117(0,0)<1;1,0>
  ...
  goto (M1, 1) label5
  ...
  scatter4_scaled.R (M1, 16) T6 0x0:ud V227.0 V116.0
  scatter4_scaled.R (M5, 16) T6 0x0:ud V228.0 V117.0
```

Приватная память – это регистры

```
label3:
  ...
  gather4_scaled.R (M1, 16) T6 0x0:ud V229.0 V264.0
  gather4_scaled.R (M5, 16) T6 0x0:ud V230.0 V265.0
  mad (M1, 16) V264(0,0)<1> V262(0,0)<1;1,0>
               V237(0,0)<1;1,0> V264(0,0)<1;1,0>
  mad (M5, 16) V265(0,0)<1> V263(0,0)<1;1,0>
               V238(0,0)<1;1,0> V265(0,0)<1;1,0>
  scatter4_scaled.R (M1, 16) T6 0x0:ud V229.0 V264.0
  scatter4_scaled.R (M5, 16) T6 0x0:ud V230.0 V265.0
  ....
  goto (M1, 1) label3
```

Stateless to stateful?

- На прошлом слайде мы видели, что load – это gather scaled.

```
gather4_scaled.R (M1, 16) T6 0x0:ud V229.0 V264.0
```

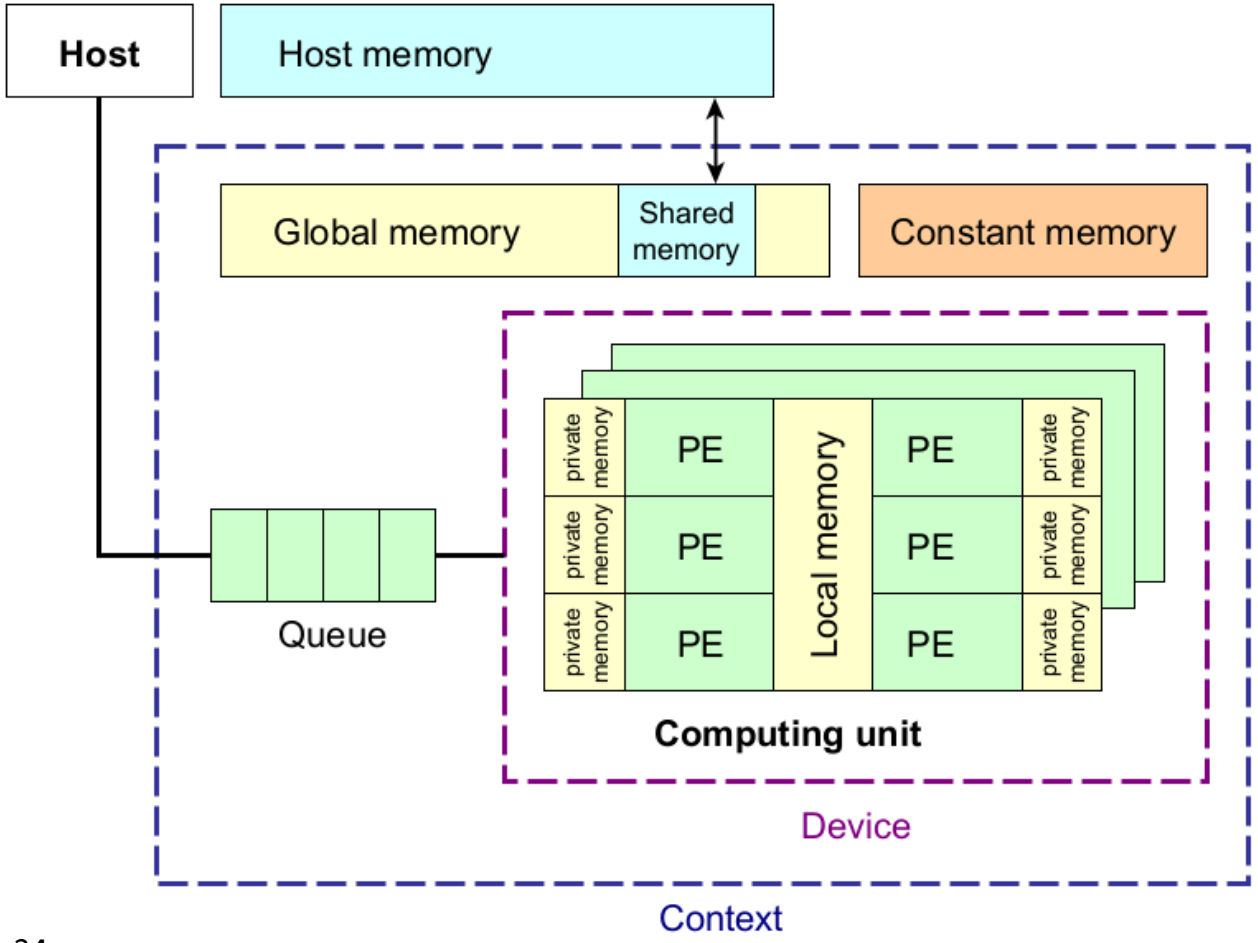
- Здесь T6 – это binding table index. Но мы же просили shared memory….

```
auto *A = sycl::malloc_shared<T>(Sz, DeviceQueue);
auto *B = sycl::malloc_shared<T>(Sz, DeviceQueue);
auto *C = sycl::malloc_shared<T>(Sz, DeviceQueue);

// вычисления ниже будут проведены в stateful памяти
cl::sycl::range<1> numOfItems{Sz};
DeviceQueue.parallel_for(numOfItems,
                         [=](auto n) { C[n] = A[n] + B[n]; });
DeviceQueue.wait();
```


GATHER4_SCALED

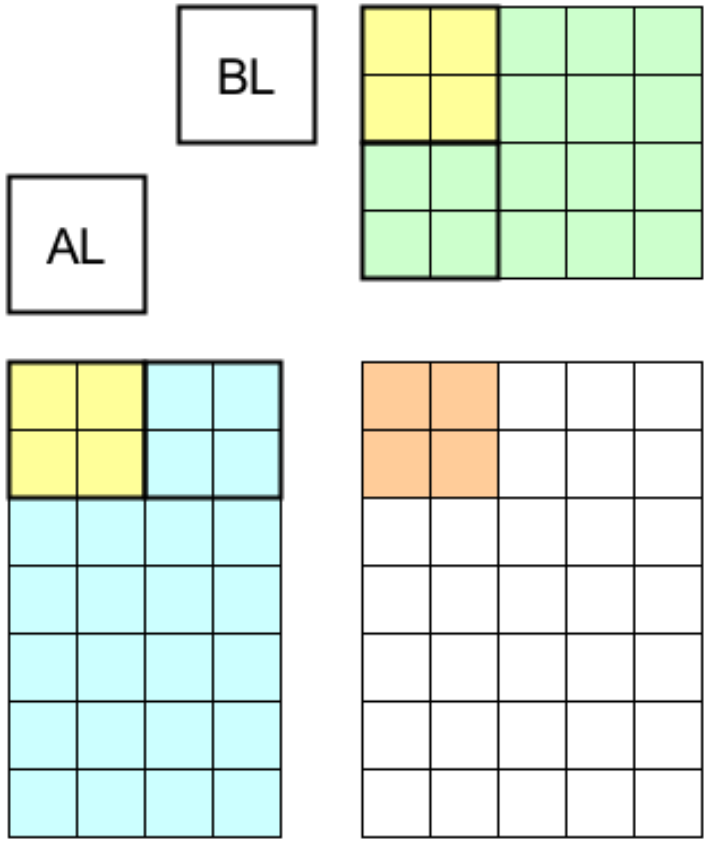
Клинч терминологии

SYCL	CUDA
private	local
local (shared local, SLM)	shared
shared (shared virtual, unified shared, SVM, USM)	unified
global	device



Local memory – это быстро, но мало

```
Sum = 0; N = AY / LSZ;

for (int Tile = 0; Tile < N; Tile++) {
  const int TX = LSZ * Tile + LX;
  const int TY = LSZ * Tile + LY;
  AL[TX][TY] = A[GX * AY + TX];
  BL[TX][TY] = B[TX * BY + GY];
  It.barrier(sycl_local_fence);

  for (int K = 0; K < LSZ; K++)
    Sum += AL[LX][K] * BL[K][LY];
  It.barrier(sycl_local_fence);
}
```

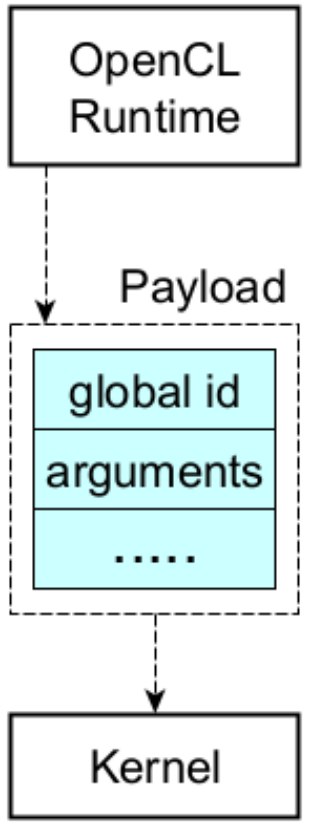


matmult_local.cc

Захваченные переменные

```
const int AY = AY_from_program_arguments;
.....
T *A = malloc_shared<T>(AX * AY, Queue);
std::copy(AH.begin(), AH.end(), A);
.....
Queue.submit([&](sycl::handler &Cgh) {

  auto Kernmul = [=](sycl::id<2> Id) {
    const int LSZ = Id.get_local_size(0);
    const int N = AY / LSZ;
```

- Обратим внимание: константа AY для кернела константой не является. Local size мы тут тоже извлекаем из payload.



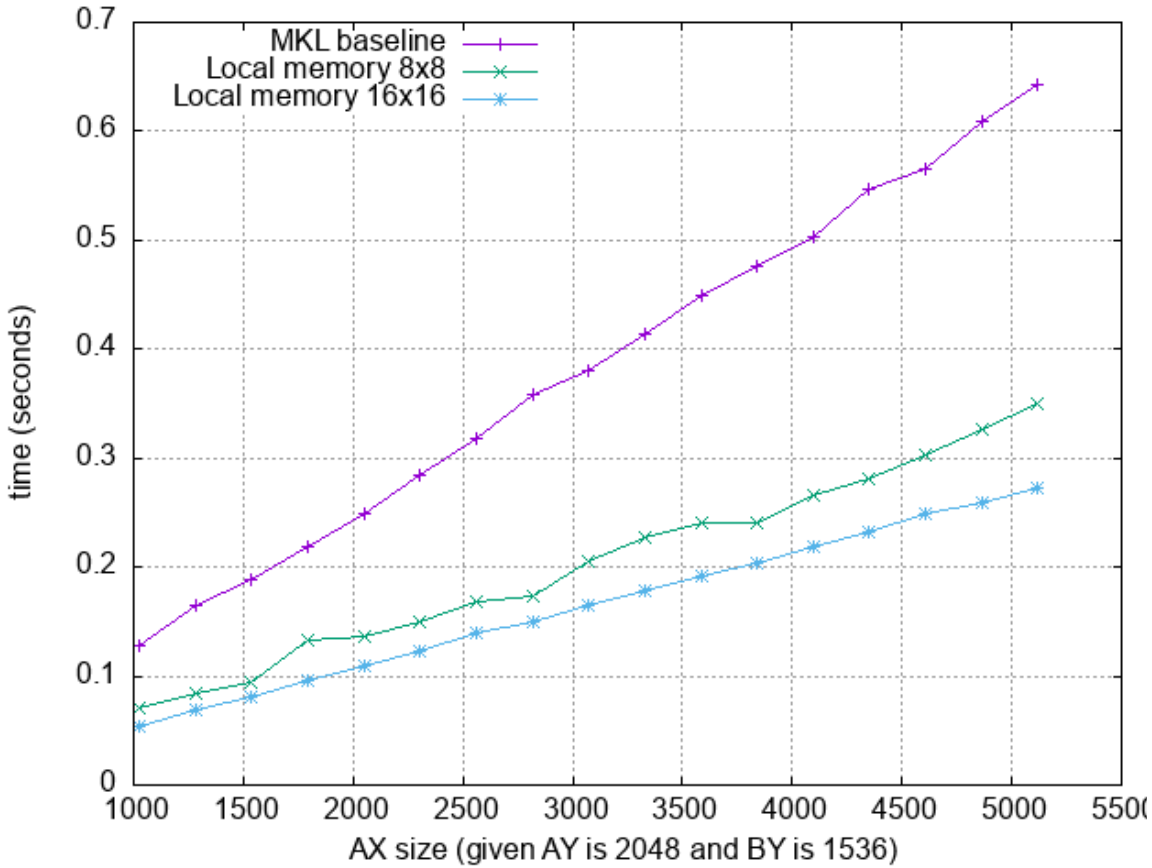
Специализационные константы

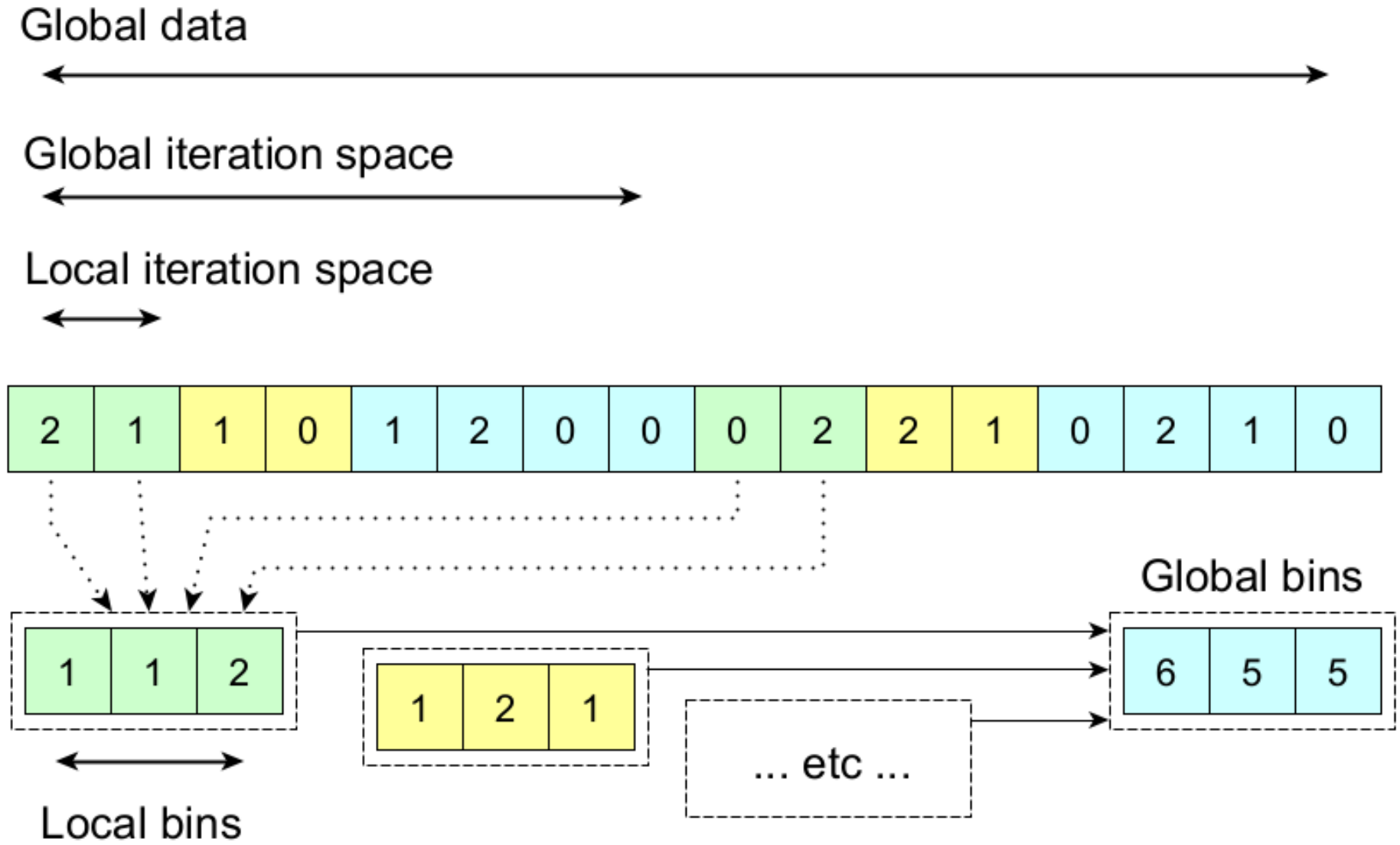
```
int AY = 512; // можно и не const

const static sycl::specialization_id<int> AYC;
.....
DeviceQueue.submit([&](sycl::handler &Cgh) {
  Cgh.template set_specialization_constant<AYC>(AY);

  auto Kernmul = [=](sycl::id<2> Id, sycl::kernel_handler Kh) {
    int Row = Id.get(0), Col = Id.get(1);
    T Sum = 0;
    const int AYK = Kh.get_specialization_constant<AYC>();
```

- Есть некий эффект на время компиляции: специализированный кернел нужно обязательно JIT-компилировать.

matmult_local_shared_spec.cc

Замеры с локальной памятью





https://en.wikipedia.org/wiki/Histogram

Атомики и барьеры

```
for (int I = LocalX; I < NumBins; I += LSZ)
  LocalHist[I] = 0;
Id.barrier();

for (int I = GlobalX; I < NumData; I += GSZ) {
  const T Next = Data[I];
  local_atomic_ref<T>(LocalHist[Next]).fetch_add(1);
}
Id.barrier();

for (int I = LocalX; I < NumBins; I += LSZ) {
  const T Data = LocalHist[I];
  global_atomic_ref<T>(Bins[I]).fetch_add(Data);
}
```


hist_local_acc.cc

Атомики в C++ и в SYCL

[atomics.ref.generic.general], namespace std

```
template <class T> struct atomic_ref {
  T load(memory_order = memory_order::seq_cst) const noexcept;
  // ....
```

[SYCL spec, 4.15.3], namespace sycl

```
template <typename T, memory_order DefaultOrder,
          memory_scope DefaultScope,
          address_space Space = address_space::generic_space>
class atomic_ref {
  T load(memory_order order = default_read_order,
          memory_scope scope = default_scope) const noexcept;
```


Atomic references

Примеры, использованные выше

```
template <typename T> using global_atomic_ref =
  sycl::atomic_ref<T, sycl::memory_order::relaxed,
                   sycl::memory_scope::system,
                   sycl::access::address_space::global_space>;

template <typename T> using local_atomic_ref =
  sycl::atomic_ref<T, sycl::memory_order::relaxed,
                   sycl::memory_scope::work_group,
                   sycl::access::address_space::local_space>;
```

- Обратите внимание, что scope повторяется дважды: как параметр шаблона и как параметр функции load.

Проблемы consistency model

- Формальная модель в SYCL пока не определена.
- Она должна быть трёхмерной. У нас есть scope, address space и ordering.
- Например, если мы делаем release store в workgroup scope, local address space и делаем acquiring load в global scope для generic address space, должно ли между ними возникать отношение synchronizes with?
- К слову, стандарт OpenCL 2.0 определяет два разных отношения synchronizes-with, глобальное и локальное, и делает это не случайно.

OpenCL-2.0 (pdf spec)

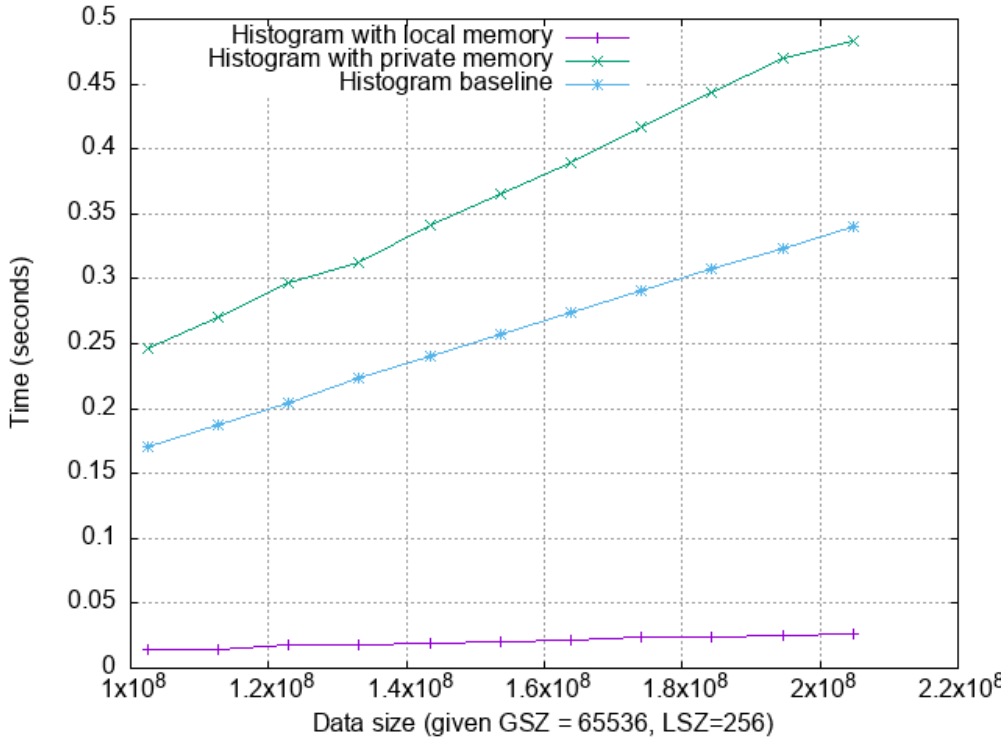
Странная идея: приватная гистограмма

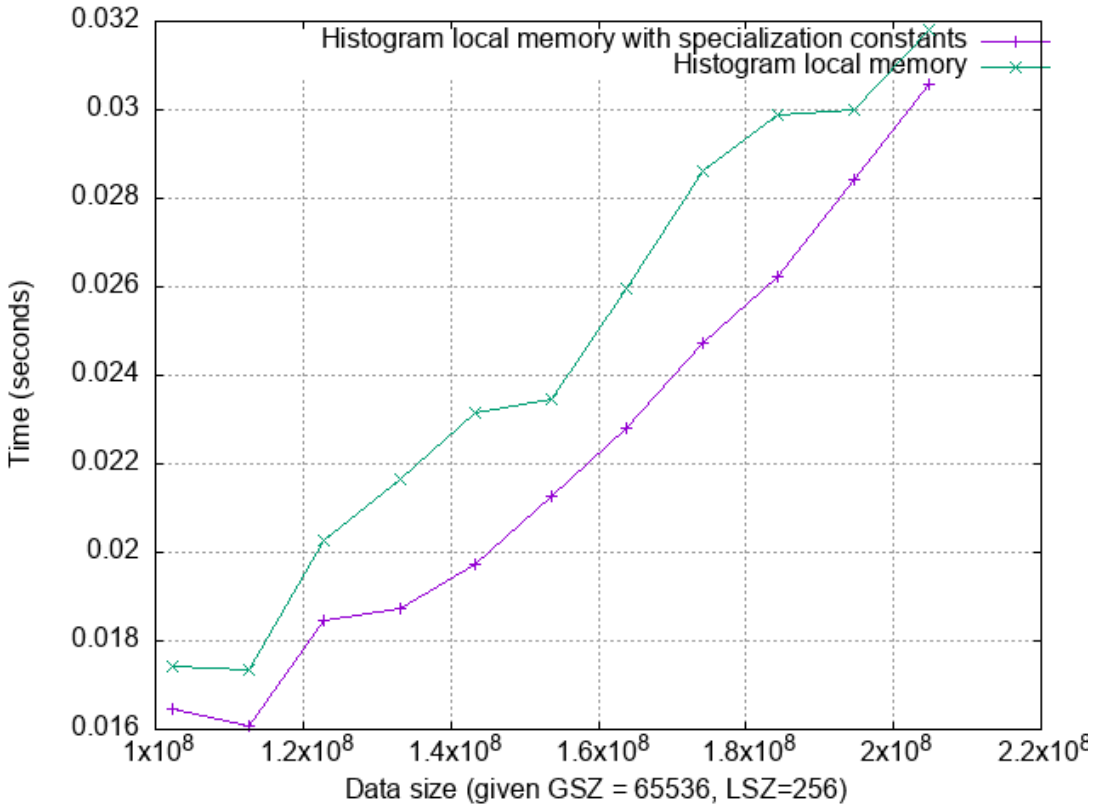
```
T PrivateHist[MAX_HSZ] = {0};
NumTiles = NumData / NumBins;

for (int I = GlobalX; I < NumTiles; I += GSZ / NumBins)
  for (int J = 0; J < NumBins; J += 1)
    PrivateHist[Data[I * NumBins + J]] += 1

for (int I = 0; I < NumBins; I += 1) {
  const T Data = PrivateHist[I];
  global_atomic_ref<T>(Bins[I]).fetch_add(Data);
}
```

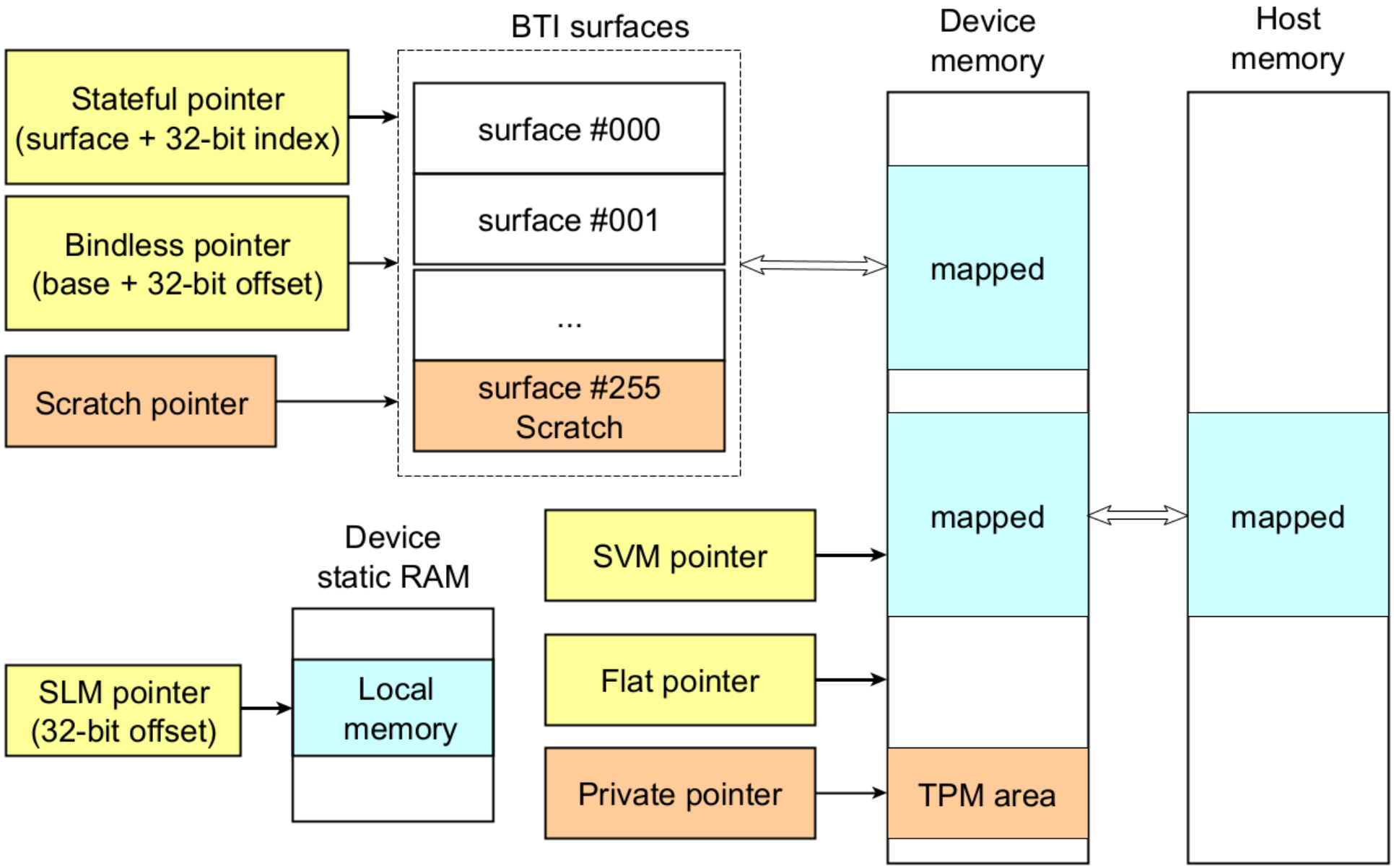
- Кажется, мы избавились от атомарной работы с локальной памятью….

hist_private.cc

Private memory – это точно регистры?






BTI surfaces
Device memory
Host memory
Stateful pointer (surface + 32-bit index)
Bindless pointer (base + 32-bit offset)
Scratch pointer
surface #000
surface #001
...
surface #255 Scratch
mapped
mapped
mapped
Device static RAM
SVM pointer
Flat pointer
Private pointer
SLM pointer (32-bit offset)
Local memory
TPM area

Что указатель означает в C++?

- Аналог stateless: указатели на данные, указатели на функции.
- Аналог stateful: указатели на члены и функции-члены классов (pointer-to-member).
- Работа регламентируется [conv.ptr], [conv.mem], [conv.fctptr] и [expr.type], где определены правила pointer conversions и построения composite pointer types.
- Пример типичного утверждения из [expr.rel] для сравнения указателей:

If both operands are pointers, pointer conversions (7.3.12) and qualification conversions (7.3.6) are performed to bring them to their **composite pointer type** (7.2.2). After conversions, the operands shall have the same type.

Composite pointer types: правила

- Если T_1 nullptr, то T_2 (3.1, 3.2).
- Если T_1 – это cv_1 void pointer, то cv_{12} void pointer. Здесь cv_{12} – это union (3.3).
- Если T_1 указывает на noexcept функцию, а T_2 на такую же не-noexcept функцию, то T_2 (3.4). То же для указателей на функции-члены (3.6)
- Если T_1 – это указатель на cv_1 C_1 и T_2 – это указатель на cv_2 C_2, где C_1 reference related [dcl.init.ref] с типом C_2, то их cv-combined [conv.qual] тип (3.5). То же самое для указателей на поля (3.7).
- Если T_1 и T_2 – это similar types [conv.qual], то результат – их cv-combined тип.
- И, как обычно, [...] otherwise, a program that necessitates the determination of a composite pointer type is ill-formed.

C++: смешивать нельзя, но хочется.

```
template <typename R, typename... Args>
class fire_once<R(Args...)> {
  std::unique_ptr<void, void (*)(void *)> ptr;
  R (*invoke)(void *, Args...) = nullptr;

// constructor from anything callable
template <typename F> fire_once(F &&f) {
  auto pf = std::make_unique<F>(std::move(f));

  invoke = +[](void *pf, Args... args) -> R {
    F *f = reinterpret_cast<F *>(pf);
    return (*f)(std::forward<Args>(args)...); // сомнительно
```

- Выкрутиться можно, но проблема в том, что надо выкручиваться.

SYCL: смешивать почти можно.

- Указатели на функцию в SYCL-20 запрещены.
- Зато все остальные типы указателей можно смешивать и получать generic pointers.

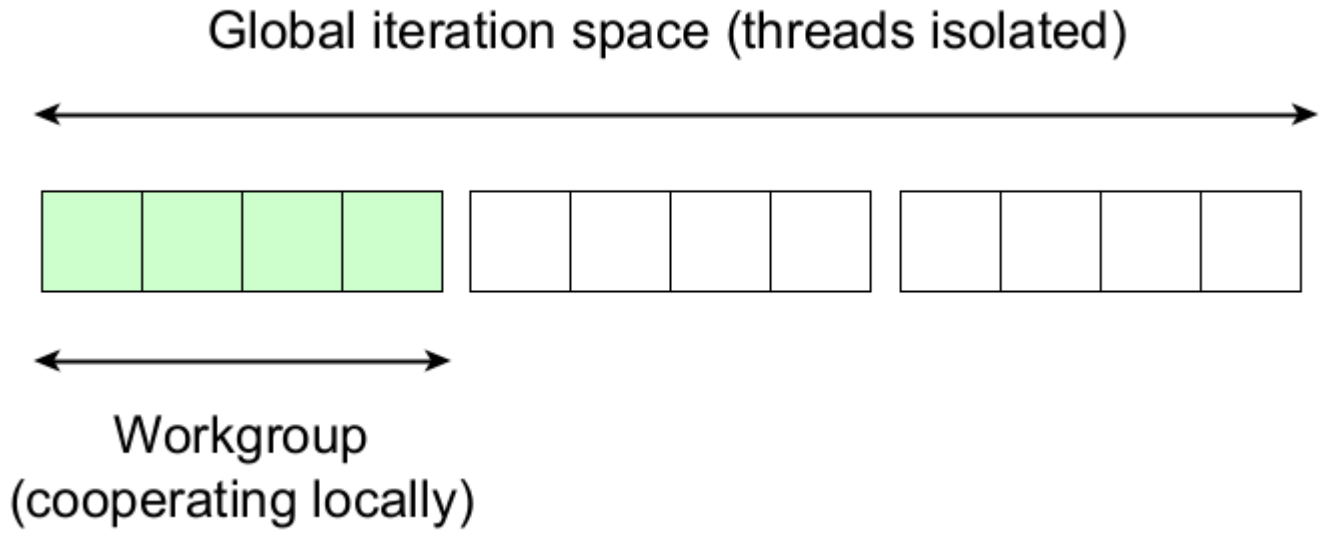
```
T *Ptr = Cond ? &GlobalData[I] : &LocalData[I];

*Ptr = 5; // что тут сгенерировать компилятору?
```

- Generic address space – это крайне интересная концепция.
- Существует статическое и динамическое разрешение.
- Хотели бы мы в C++ концепцию generic pointer, например между указателем и указателем на функцию?

Intel Graphics ISA Tutorial

Концепция рабочей группы

- Рабочая группа – это элемент кооперативной многозадачности.
- Очень часто в реализациях на CPU потоки, составляющие рабочую группу, – это корутины (или что-то вроде них).



Иерархический параллелизм в SGEMM

```
using LTy = sycl::accessor<T, 2, read_write, local>;
LTy Asub{LSZ, Cgh}, Bsub{LSZ, Cgh};

auto KernG = [=](sycl::group<2> Group) {
  sycl::private_memory<int, 2> Sum(Group);

  for (int Tile = 0; Tile < NumTiles; Tile++)
    Group.parallel_for_work_item([&](sycl::h_item<2> It) {
      ....
    });
    // implicit barrier
};

Cgh.parallel_for_work_group<class mmgr>(NumGr, LSZ, KernG);
```

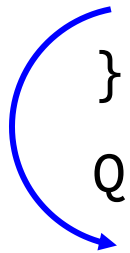

matmult_groups_priv.cc

Два мира многопоточности: exec-style

```
namespace ex = std::execution;

// мы можем получить scheduler откуда угодно
auto Sch = ex::thread_pool.scheduler();

// начинаем работу
auto Begin = ex::schedule(sch);

// асинхронная композиция задач
auto Hi = ex::then(begin, []{ return 13;});
auto Add_42 = ex::then(hi, [](int arg) { return arg + 42; });

// получаем значение
auto [Val] = std::this_thread::sync_wait(Add_42).value();
```

Два мира многопоточности: SYCL-style

```
// мы можем получить queue откуда угодно
auto Q = sycl::queue{gpu_selector}; int V;

// асинхронная композиция задач: event-based
auto EvtHi = Q.submit([&](handler &Cgh){
  auto Val = accessor(V); Cgh.single_task([=] { Val = 13; });
});

Q.submit([&](handler &Cgh){
  auto Val = accessor(V); // или Cgh.depends(EvtHi)
  Cgh.single_task [=]{ Val += 42; });
});

auto Val = host_accessor(V); // получаем значение V
```



В чём большая разница? В барьерах.

- Концепция рабочей группы: потоки исполнения внутри одного устройства, делящие одну локальную память.

```
for (auto i : N_i)
  for (auto j : N_j)
    parallel for (auto k : N_k)
      do(i, j, k); // порядок по k не важен, и N_k не велико
```

- Если у нас нет барьеров, мы не можем тут переставить циклы.

```
do(0, 0, 1), do(0, 0, 0), do(0, 0, 2),
do(0, 1, 2), do(0, 1, 1), do(0, 1, 0), ....
```

- Нас не устроит: `do(0, 0, 1), do(0, 0, 0), do(0, 1, 0), ....`

Пример: битоническая сортировка

```
for (int Step = 0; Step < N; Step++) {
  for (int Stage = Step; Stage >= 0; Stage--) {
    auto Evt = DeviceQueue.submit([=](sycl::handler &Cgh) {
      auto Kern = [=](sycl::id<1> Id) {
        // parallel swap next stage for next step
      };
      Cgh.parallel_for<class bsort>(sycl::range<1>{Sz}, Kern);
    });
    Evt.wait(); // безальтернативно
  }
}
```

- Но, имея в руках барьеры и локальную память, мы можем лучше.

bitonic_device.cc

Пример: битоническая сортировка

```
auto Evt = DeviceQueue.submit([=](sycl::handler &Cgh) {
  LTy Cache{LocalMemorySize, Cgh};
  auto Kern = [=](sycl::nd_item<1> Id) {
    int G = Id.get_global_id(0), L = Id.get_local_id(0);
    Cache[L] = Array[G];
    Id.barrier();

    for (int Step = 0; Step < StepEnd; Step++) {
      for (int Stage = Step; Stage >= 0; Stage--) {
        // parallel swap next stage for next step in Cache
        Id.barrier();
      }
    }
```


bitonic_device_local.cc

В общем случае

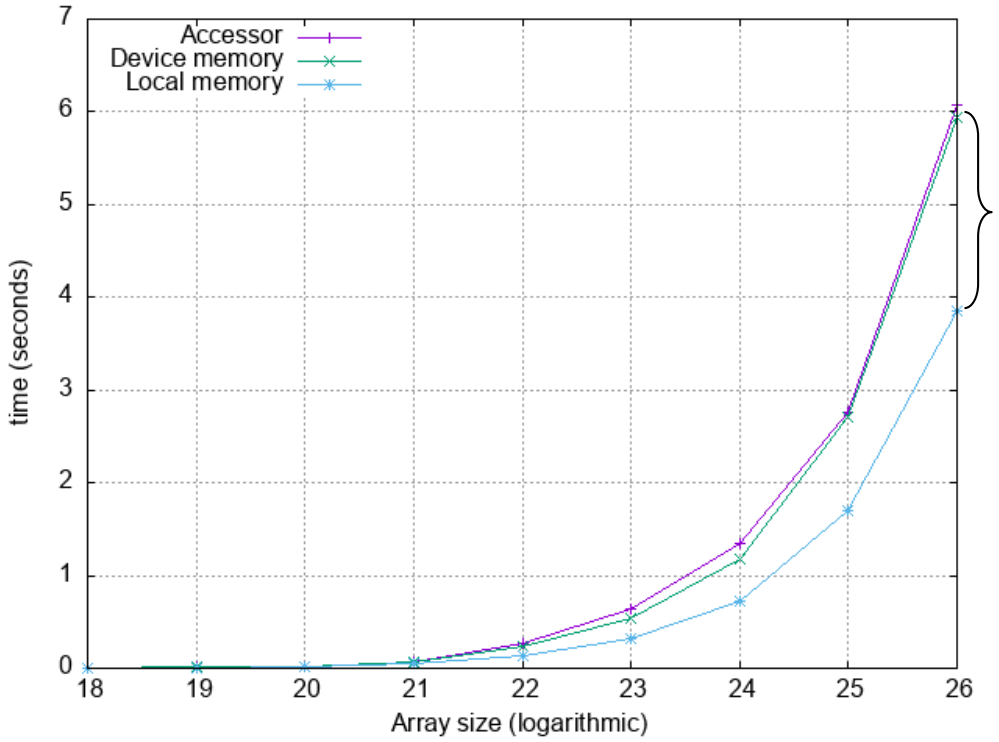
```
for (auto i : N_i)
  for (auto j : N_j)
    parallel for (auto k : N_k)
      do(i, j, k); // порядок по k не важен, и N_k не велико
```

- Трансформируется в следующий более выгодный вид:

```
parallel for (auto k : N_k)
  for (auto i : N_i)
    for (auto j : N_j) {
      do(i, j, k); // порядок по k не важен, и N_k не велико
      barrier(k);  // тут все work items ждут перед итерацией
    }
```

Слон, которого никто не видит

- Авторы предложения в стандарт по std::execution привыкли мыслить терминами thread pools и CPU.
- Они плохо понимают или вообще не понимают специфику гетерогенного параллелизма.
- Ничего похожего на разные пространства имён, барьеры или хотя бы иерархический параллелизм не предлагается.



std::execution draft

Рекомендованные вопросы

- Расскажите про свёртки, эффективность typed памяти и про ограниченность объёма локальной памяти.
- Расскажите грустную историю про то почему компилятор не двигает store вниз.
- Расскажите больше про странные правила для атомиков в OpenCL 2.0, OpenCL 3.0 и SYCL.
- Расскажите про внезапные оптимизации приватной гистограммы.
- Расскажите про объекты редукции.
- Хочется больше знать про VISA ASM и про реальный ассемблер для видеокарт Intel Xe.

Литература

- Khronos group – SYCL specification, 2020 https://www.khronos.org/registry/SYCL/specs/sycl-2020/html/sycl-2020.html
- Konstantin Bobrovskii, "Intel Open Source SYCL Compiler Project", EVS, 2019
- Intel OneAPI GPU optimization guide https://www.intel.com/content/www/us/en/develop/documentation/oneapi-gpu-optimization-guide
- Intel OneAPI Compiler documentation https://intel.github.io/llvm-docs/UsersManual.html
- DPC++ User API documentation https://intel.github.io/llvm-docs/doxygen/modules.html
- Programming parallel computers https://ppc.cs.aalto.fi/
- Gordon Brown, "A Modern C++ Programming Model for GPUs using Khronos SYCL", CppCon, 2018
- Specification for ASM: https://01.org/linuxgraphics/documentation/hardware-specification-prms
- GenISA introduction: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-gen-assembly.html