1 SYCL memory basics

SYCL is a single-source programming model. This poses a problem. Normaly in heterogeneous systems, there are a diverse variety of memory types (we'll see some of them later). On the other hand, in C++ all memory is the same. So the main problem for single-source system is to find a way to somehow map one upto another.

Central ideas in this mapping are ideas of **buffer** and **accessor**. Roughly saying buffer represents GPU memory and accessor gives us a way to work with GPU memory from C++.

We will illustrate this with very basic example, which adds two vectors.

```
// T const *AVec is host pointer to vector A
// the same for BVec and CVec
// size_t Sz is vector size

cl::sycl::range<1> NumOfItems{Sz};
cl::sycl::buffer<T, 1> BufferA(AVec, NumOfItems, {host_ptr});
cl::sycl::buffer<T, 1> BufferB(BVec, NumOfItems, {host_ptr});
cl::sycl::buffer<T, 1> BufferC(CVec, NumOfItems, {host_ptr});

// making calculation on GPU
DeviceQueue.submit([&](cl::sycl::handler &Cgh) {
  auto A = BufferA.template get_access<sycl_read>(Cgh);
  auto B = BufferB.template get_access<sycl_read>(Cgh);
  auto C = BufferC.template get_access<sycl_write>(Cgh);

  auto Kern = [A, B, C](cl::sycl::id<1> wiID) {
    C[wiID] = A[wiID] + B[wiID];
  };
  Cgh.parallel_for<class vector_add_buf<T>>(NumOfItems, Kern);
});

// checking with host calculations
auto A = BufferA.template get_access<sycl_read>();
auto B = BufferB.template get_access<sycl_read>();
auto C = BufferC.template get_access<sycl_read>();

for (int I = 0; I < Sz; ++I)
  if (C[I] != A[I] + B[I]) {
    std::cerr << "At index: " << I << ". ";
```

```
30      std::cerr << C[I] << " != " << A[I] + B[I] << "\n";
31      std::terminate();
32    }
```

Listing 1: Vector addition with SYCL buffers

Here things are somewhat abbreviated (SYCL often requires really long namespace chains).

```
1  // buffer attribute
2  constexpr auto host_ptr =
3      cl::sycl::property::buffer::use_host_ptr{};
4
5  // accessor types
6  constexpr auto sycl_read = cl::sycl::access::mode::read;
7  constexpr auto sycl_write = cl::sycl::access::mode::write;
```

This is buffer attribute, showing, that buffer have underlying host pointer. Also we have accessor type abbreviations for read-only and write-only things.

Next we create buffer, which is sort of handle to GPU memory. On creation we explicitly tell which part of CPU memory to take and send to GPU.

```
1  cl::sycl::range<1> NumOfItems{Sz};
2  cl::sycl::buffer<T, 1> bufferA(AVec, NumOfItems, {host_ptr});
```

We are specifying host pointer, range, and leaving everything else to SYCL API. Now our buffer have its representative in the SYCL world. It will manage data and send it back and forth if required. If we do not want it to write back after kernel is done, we can block writeback explicitly.

```
1  BufferA.set_final_data(nullptr);
```

Now we are getting an **accessor** A. Accessors are C++ objects, acting as pointers to the corresponding address space. There is no notion of address space in C++, so we need this abstract layer to fill the gap.

```
1  auto Evt = DeviceQueue.submit([&](cl::sycl::handler &Cgh) {
2  // .....
3    auto A = BufferA.template get_access<sycl_read>(Cgh);
```

Which kind of memory are pointed-to by this accessor? Inside `cl::sycl::buffer`, method `get_access` returns accessor to global memory only. That is why now we will limit ourselves only with different flavors of global memory. For other memory kinds, see (1.2) and later. Inside offload working with such accessor, indexed by SYCL id is much like working with host pointer, indexed by integral value.

```
auto Kern = [A, B, C](cl::sycl::id<1> wiID) {
  C[wiID] = A[wiID] + B[wiID];
};
Cgh.parallel_for<class vector_add_buf<T>>(NumOfItems, Kern);
```

At first glance this looks slightly overcomplicated? Yes, it should. For some reasons we need:

- Create buffer from host pointer. Capture this buffer to device queue submit.
- Get accessor from buffer. Capture this accessor to kernel lambda. At this point you have some idea about which type of memory you are addressing.
- Use accessor inside kernel lambda, index it by workitem id.

Even more: we can have this line, getting an accessor not only inside device queue. We might have it in host code, after we have writeback to buffer `C`.

```
auto C = BufferC.template get_access<sycl_read>();
```

Host-side accessor also behaves like pointer with same indexing operator and we can check on host side that everything is correct.

Is it the only way? Can we do things different? Lets start investigating...

1.1 Host, device and shared memory

Figure (1) represents our first, rather naive mental model of host, device and shared memory. You may think that GPU have its own memory onboard, and CPU does as well. So when you are working with buffers as we did in example to (1), you explicitly instruct SYCL runtime when to move memory from CPU to GPU and when to read results back.

SYCL runtime also builds implicit task graph, based on accessors and schedules tasks, honoring access dependencies. This is not really about memory, so lets pass this topic for now.

Other way to write vector add is to write it using **shared memory**.

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

// copy to shared memory
```

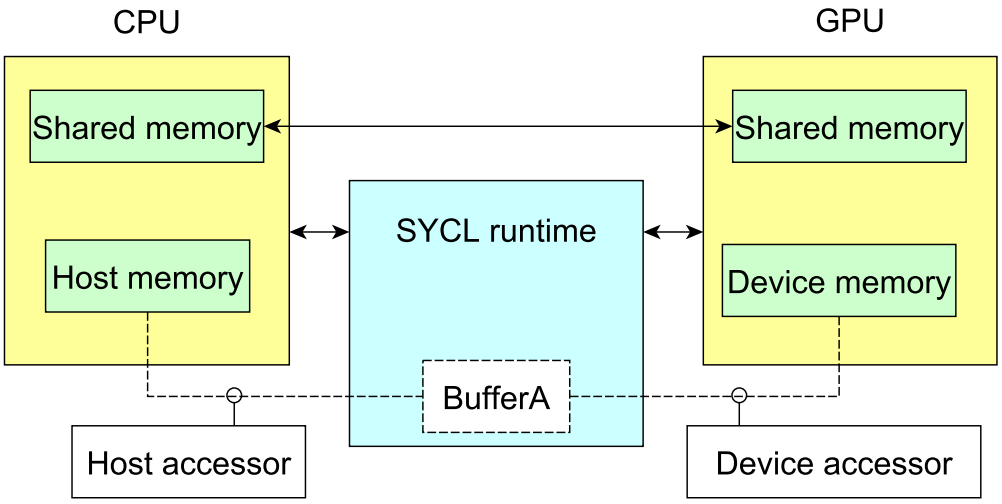

Figure 1: SYCL device and shared memory

```
6  std::copy(AVec, AVec + Sz, A);
7  std::copy(BVec, BVec + Sz, B);
8
9  DeviceQueue.parallel_for(numOfItems,
10     [=](auto n) { C[n] = A[n] + B[n]; });
11
12 DeviceQueue.wait();
13
14 // copy back to normal memory
15 std::copy(C, C + Sz, CVec);
```

Listing 2: Vector addition in shared memory

In this case you are not telling device queue to create buffers or accessors. Instead you just order to put everything in shared memory and let hardware to synchronize this memory in the background.

1.2 Private, local and global memory

Any modern GPU (at least those, produced by NVidia, AMD and Intel) have notion of shared local memory and private memory. In abstract model of SYCL this is simple: we have work-items and each work-item have small amount of private memory. Work-items are combined in work-groups, each item within work-group have shared access to bigger amount of local memory

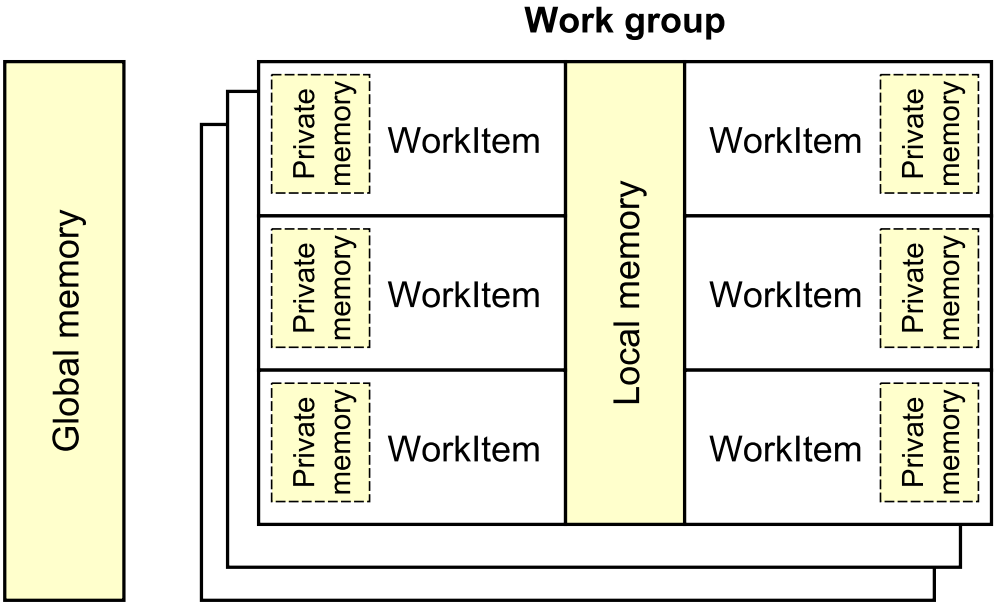

Figure 2: SYCL global, local and private memory

and finally all work-items have shared access to global memory as represented on (fig. 2).

In this sense, shared virtual memory is also (logically) a kind of global memory, since it is accessible to all work-items.

Basic example on how private memory affects performance is GEMM example.

In matrix multiplication iteration space is a cell of the resulting matrix, so each workitem as shown on (fig. 3) basically does dot product of vectors and then accumulate results.

As a first approach we can come up with simple program from (lst. 3).

```
1  auto A = BufferA.template get_access<sycl_read>(Cgh);
2  auto B = BufferB.template get_access<sycl_read>(Cgh);
3  auto C = BufferC.template get_access<sycl_write>(Cgh);
4
5  auto Kernmul = [A, B, C, AY](cl::sycl::id<2> WorkItem) {
6    const int Row = WorkItem.get(0);
7    const int Col = WorkItem.get(1);
8
9  #ifdef NOPRIVATE
10   for (int K = 0; K < AY; K++)
```

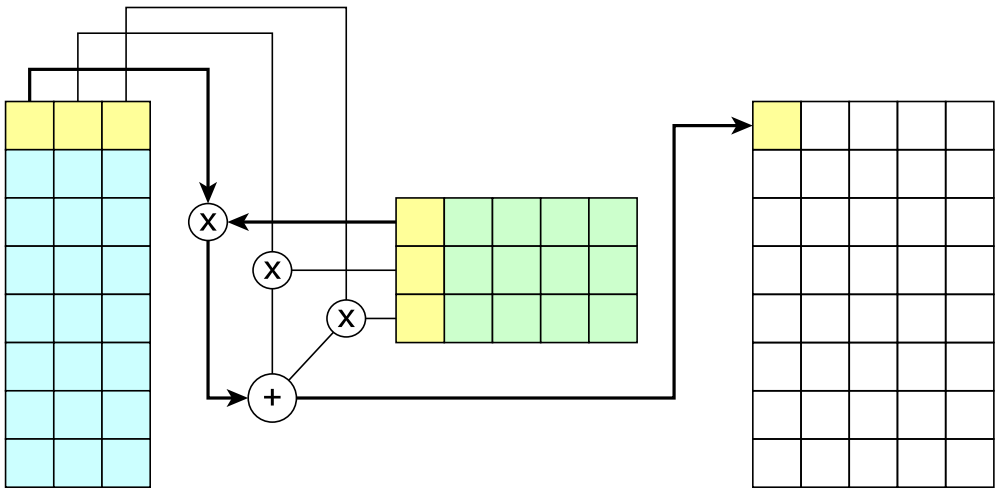

Figure 3: SYCL global, local and private memory

```
11      C[Row][Col] += A[Row][K] * B[K][Col];
12  #else
13    // this variable is workitem-private
14    T Sum = 0;
15    for (int K = 0; K < AY; K++)
16      Sum += A[Row][K] * B[K][Col];
17    C[Row][Col] = Sum;
18  #endif
19  };
20  Cgh.parallel_for<class mmult_naive_buf<T>>(Csz, Kernmul);
```

Listing 3: Naive GEMM with private memory

Here we are doing most naive matrix multiplication possible. Still results on TGLLP with and without private variable are dramatically different, see (fig. 4).

Of course, usage of local memory for matrix multiplication improves things even more. To use it, we need local accessor, which represents something like pointer to local memory.

Main idea for listing (lst. 4) is to have a little chunk of the local memory (like 16 x 16 array). Then we tile matrix into small submatrices. Loop over tiles updates private sum for each tile. After we are done, sum becomes correct and is written to global memory.

Critical new idea here is idea of **barrier**. Barrier acts like semaphore: it stops thread and resume it after all threads will reach the barrier. In this

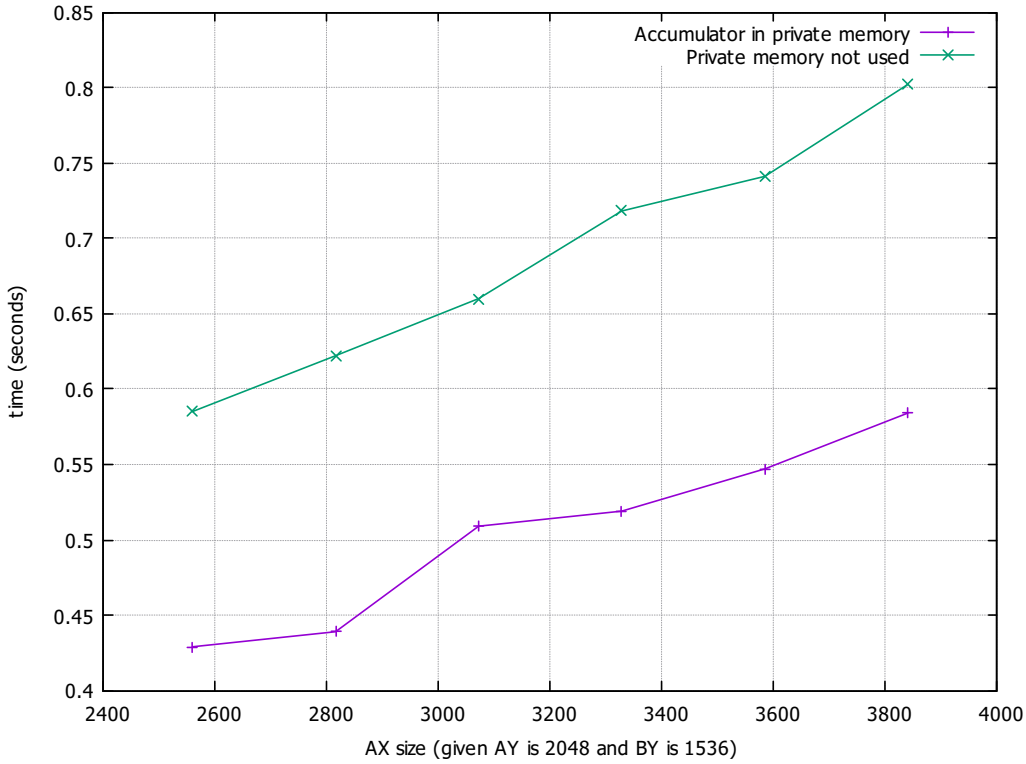

Figure 4: TGLLP (Iris Xe) results for private memory usage for kernel from (lst. 3)

case we are waiting for all threads in the work group.

```
auto A = BufferA.template get_access<sycl_read>(Cgh);
auto B = BufferB.template get_access<sycl_read>(Cgh);
auto C = BufferC.template get_access<sycl_write>(Cgh);

// local memory
using LTy = cl::sycl::accessor<T, 2, sycl_read_write,
    sycl_local>;
LTy Asub{BlockSize, Cgh}, Bsub{BlockSize, Cgh};

auto KernMul = [=](cl::sycl::nd_item<2> It) {
  int Row = It.get_local_id(0);
  int Col = It.get_local_id(1);
  int GlobalRow = LSZ * It.get_group(0) + Row;
  int GlobalCol = LSZ * It.get_group(1) + Col;
  int NumTiles = AY / LSZ;

  T Sum = 0;
  for (int Tile = 0; Tile < NumTiles; Tile++) {
    int TiledRow = LSZ * Tile + Row;
    int TiledCol = LSZ * Tile + Col;
    Asub[Row][Col] = A[GlobalRow][TiledCol];
    Bsub[Row][Col] = B[TiledRow][GlobalCol];
    // waiting for all threads to fill Asub[Row][Col]
    It.barrier(sycl_local_fence);
    for (int K = 0; K < LSZ; K++)
      Sum += Asub[Row][K] * Bsub[K][Col];
    // waiting for all threads to use Asub[Row][Col]
    It.barrier(sycl_local_fence);
  }
  C[GlobalRow][GlobalCol] = Sum;
};
Cgh.parallel_for<class mmult_local_buf<T>>(Range, KernMul);
```

Listing 4: GEMM with local and private memory

We may plot results of local memory usage on the same graph (fig. 4). We may see one more tremendous improvement.

Downside of explicit local memory usage is requirement for user to manually control barriers. Another interesting feature of SYCL programming is explicit hierarchical parallelism.

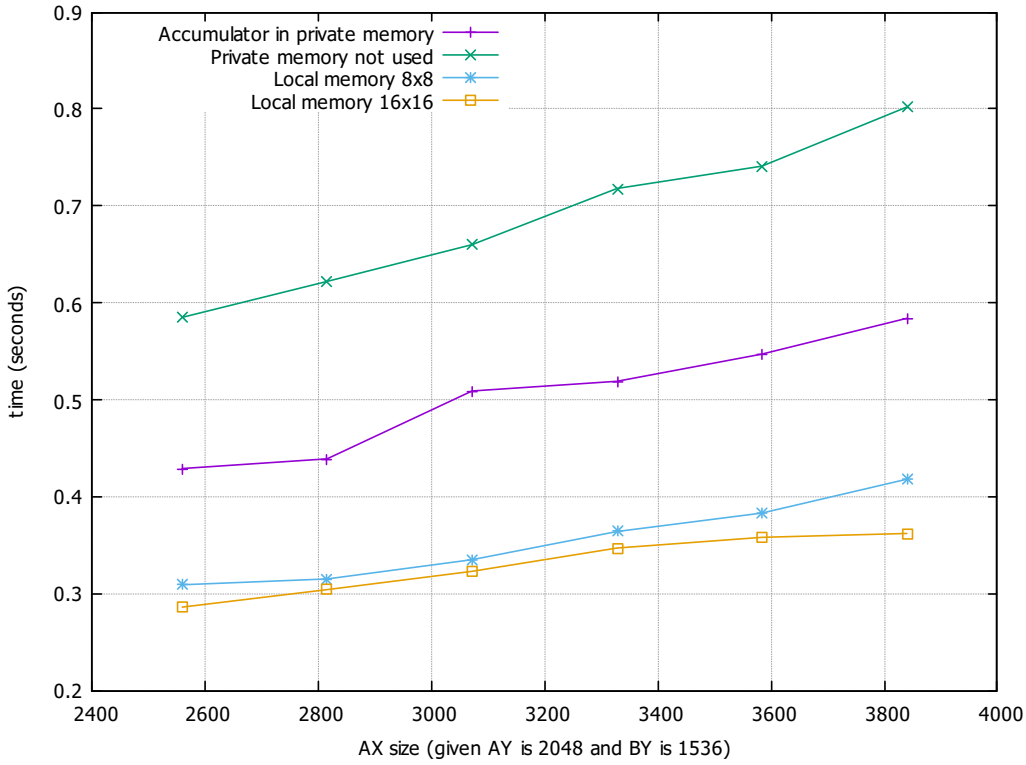

Figure 5: TGLLP (Iris Xe) results for local and private memory usage for GEMM

```
auto KernMul = [=](cl::sycl::group<2> Group) {
  Group.parallel_for_work_item([&](cl::sycl::h_item<2> It) {
    int GlobalRow = It.get_global_id(0);
    int GlobalCol = It.get_global_id(1);
    C[GlobalRow][GlobalCol] = 0;
  });
....
};
Cgh.parallel_for_work_group<class mmult_groups<T>>(
  NumGroups, BlockSize, KernMul);
```

Listing 5: Idea of hierarchial parallelism

Simplest idea represented on (lst. 5). This listing starts kernel as parallel for work group and then inside this submission runs parallel for work item. We are relying on implicit barriers between work item submissions.

We might note, that this kind of parallelism requires private memory accessor. In old way, when we used ND range, we could just have variable inside `parallel_for` and that's all. Now things are more complicated: any private variable may live in the scope of `parallel_for_workitem` submission only. But in order to set value of result in the end, we need some private variable, visible to any of submissions.

```
auto A = BufferA.template get_access<sycl_read>(Cgh);
auto B = BufferB.template get_access<sycl_read>(Cgh);
auto C = BufferC.template get_access<sycl_write>(Cgh);

cl::sycl::range<2> NumGroups{AX / LSZ, BY / LSZ};
cl::sycl::range<2> BlockSize{LSZ, LSZ};
const int NumTiles = AY / LSZ;

// local memory
using LTy = cl::sycl::accessor<T, 2, sycl_read_write,
    sycl_local>;
LTy Asub{BlockSize, Cgh}, Bsub{BlockSize, Cgh};

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

  Group.parallel_for_work_item(
      [&](cl::sycl::h_item<2> It) { Sum(It) = 0; });

  for (int Tile = 0; Tile < NumTiles; Tile++) {
```

```
20      Group.parallel_for_work_item([&](cl::sycl::h_item<2> It) {
21        int GlobalRow = It.get_global_id(0);
22        int GlobalCol = It.get_global_id(1);
23        int Row = It.get_local_id(0);
24        int Col = It.get_local_id(1);
25        int TiledRow = LSZ * Tile + Row;
26        int TiledCol = LSZ * Tile + Col;
27
28        Asub[Row][Col] = A[GlobalRow][TiledCol];
29        Bsub[Row][Col] = B[TiledRow][GlobalCol];
30      });
31
32      // rely on automatic barrier
33
34      Group.parallel_for_work_item([&](cl::sycl::h_item<2> It) {
35        int Row = It.get_local_id(0);
36        int Col = It.get_local_id(1);
37
38        for (int K = 0; K < LSZ; K++)
39          Sum(It) += Asub[Row][K] * Bsub[K][Col];
40      });
41
42      // rely on automatic barrier
43    }
44
45    // now assign sum to its cell
46    Group.parallel_for_work_item([&](cl::sycl::h_item<2> It) {
47      int GlobalRow = It.get_global_id(0);
48      int GlobalCol = It.get_global_id(1);
49      C[GlobalRow][GlobalCol] = Sum(It);
50    });
51  };
52
53  Cgh.parallel_for_work_group<class mmult_groups_priv<T>>(
54      NumGroups, BlockSize, KernMul);
```

Listing 6: Hierarchial GEMM with private memory

To see example of private memory accessor you may want to look at (lst. 6). Here we do have `Sum` variable, declared as:

```
1 cl::sycl::private_memory<int, 2> Sum(Group);
```

We can use it anywhere, accumulate things and finally use it to assign value to the output buffer in the global memory.

So logically we do have three kinds of memory (also we do have global constant one, let's forget it for now). Is it all we need to know?

Lets continue diving deeper...

2 GPU hardware and how to think about memory

2.1 Stateless, stateful and local memory

2.2 Scratch, TPM and registers

3 Full stack and how to understand your compiler

3.1 Stateful to stateless transformation

3.2 Generic address space resolution

3.3 Workgroups are useful

3.4 Spill means no performance

3.5 Memory bank conflicts

4 Vectorization and more

4.1 Gathering accesses and Laplace equation

4.2 Explicit stateful memory

4.3 Samplers for compute