From CUDA to SYCL

Michel Migdal – Codeplay / ENSIIE / Paris-Saclay

Day 4: SYCL Summer Sessions 2021

My Experience Using DPC++ for CUDA

- Ported algorithms from CUDA to SYCL
  - CUB primitives (implementing decoupled-lookback algorithm)
  - Hashing algorithms,
  - ZFP for LLNL (Floating Point compression)
- Contribution to Open-Source projects: Microsoft Antares, LLVM,
- Experimentation:
  - Cooperative groups
  - SYCL_FS: Parallel Filesystem IO from a GPU leveraging DMA, Codec offload, …
  - Compile-time memory access optimisation (for stencils)
  - And other non-conventional things such as real-time kernel guarantees/cancelling
- Involvement in Internal SYCL spec working-group
- “Ecosystem” fixes (IDE support for DPC++), tools and documentation

My Experience Using DPC++ for CUDA

Function	Native CUDA	SYCL on CUDA (optimised/original)	SYCL on ComputeCPP CPU (spir64/spirv64)	SYCL on DPC++ CPU (spir64_x86_64)	SYCL on hipSYCL (omp/cuda)
keccak	15.7	23.0	4.14 / 4.08	4.98	4.32 / 12.3
md5	14.6	20.3	6.26 / 8.70	10.5	9.27 / 19.8
blake2b	14.7	21.6 / 18.6	8.10 / 7.85	3.65	6.03 / 12.1
sha1	13.1	19.34 / 14.9	3.61 / 1.53	3.41	4.26 / 14.3
sha256	13.4	19.15 / 13.6	2.23 / 2.00	2.96	2.93 / 13.3
md2	4.18	4.23/ 2.40	0.22 / 0.25	0.112	0.25 / 1.91

cuFiles + nvJPEG + SYCL

The NVIDIA CUDA Toolkit offers cuFiles and nvJPEG to store memory from a GPU to a proprietary file system and to open pictures using hardware acceleration. All the Nvidia API calls have to be issued from the GPU. Programming paradigm shift.

```
fs<T> fs(q, 1, tmp_space); /* Setting up the Driver */
q.submit([&](handler &cgh) { /* Works on a GPU */
    fs_accessor<T> storage_accessor = fs.get_access();
    cgh.single_task([=]() {
        /* Loading a picture using a C++ decoder on the host */
        storage_accessor.load_image(0, "Cat.jpg", tmp_accessor); /* Returns the picture size on success */
        /* Do your thing and then store the data in a simple file */
        if (auto fh = storage_accessor.open<fs_mode::write_only>(0, "Neural_Network_Result.dat")) {
            fh->write(data_buffer, data_length); /* Up to 600k IOPS & 18GB/s */
            fh->close();
        }
    });
}).wait();
```

cuFiles + nvJPEG + SYCL

Features:

- Concurrent filesystem access from a SYCL kernel (but the OS limits on the number of FD per process to a few thousands ☹)
- The API supports DMA (not implemented eveywhere) to avoid memory movements and reduce resource usage
- Large and blocking operations can spawn a CPU thread
- File IO using whole groups to improve performance
- Interoperability with host/CPU codec libraries (to decode h264, jpeg, png, mp3, ...)
- Type-safe API, Host exceptions handled, etc.

Benefits:

- Paradigm shift (see cooperative-groups later): launch of a single kernel and no need to think about the data
- Reduced latencies: the GPU can process a live video stream and output the data to a pipe
- Working on terabyte datasets: no need to partition data. Usually especially painful when there are random accesses.
- Machine learning
- 600 000 IOPS and 18GB/s peak bandwidth.

Setting up your environment

Day 4: SYCL Summer Sessions 2021

Setting up oneAPI for CUDA

- Codeplay's instructions can be found here
- Setup script to build oneAPI, oneDNN, oneMKL and dependencies.
  - Run `CXX=clang++ CC=clang ./build.sh`
  - Result in a folder `deploy` in `$HOME/sycl_workspace`
  - Supports CUDA 10 and 11 (patch required for oneMKL)
- Replace your compiler by clang++ you just built (deploy/bin)
- Add the flags: `-fsycl -fsycl-targets=nvptx64-nvidia-cuda--sm_75`
- Ordering issues, to build for OpenCL &CUDA: `spir64_x86_64,nvptx64-nvidia-cuda`
- Avoid generating integration header with `-fsycl -fsycl-device-only -E-` if you want to run the pre-processor, without having a correct code)
- Allows CLion and other IDE support

Starting to port your code

Day 4: SYCL Summer Sessions 2021

DPC++ Compatibility Tool

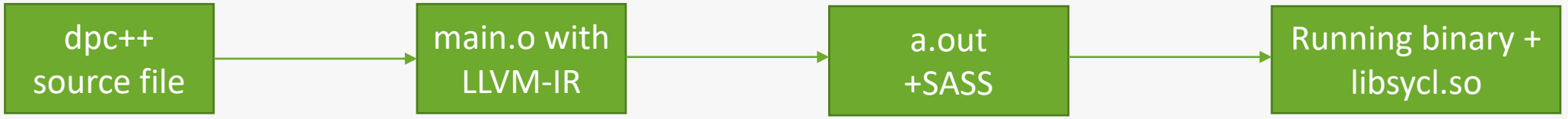
- DPCT is part of the Intel® oneAPI Toolkit: Get Started (does not support CUDA 11.4)
- Check the various flags, especially for default range size and USM type.
- Wrap kernel template launches such as:

```
reduce<float, 1><<<dimBlock, dimGrid>>>(A,B);
```

  In another functions/lambda templates. A single CUDA kernel submission line can be 20 in SYCL. DPCT will partially instantiate the submission so you will endup with unmaintainable code.

- Triple chevron formatting doesn't work `<< <…,…> >>f(…);`

DPC++ Compilation Flow



1. Compiler finds kernel submissions (`__global__`) (called from parallel_for/single_task)
2. Recursive scan to find all device functions (`__device__`)
3. Compiling the kernels to LLVM IR and embeding it in the object files.
4. Compiling host code and embeding integration header.
5. At link-time (.so or executable) LLVM IR compiled to ptx and `ptxas` generates final device code ()
6. At runtime, the **CUDA Driver API** manages the devices, memory, loads the image, etc.
7. See here for more detail

Take-away: All device functions must be declared in the translation unit and no JIT compilation for CUDA (pass the arch flags). Build SHARED objects to increase build speed else all kernels will be recompiled on every executable linking.

Targeting a Nvidia GPU

Day 4: SYCL Summer Sessions 2021

Device selection

```
class cuda_selector : public sycl::device_selector {
public:
        int operator()(const sycl::device &device) const override {
                return (device.get_platform().get_backend() == sycl::backend::cuda
                && device.get_info<sycl::info::device::is_available>()) ? 1 : -1;
        }
};
```

- sycl-ls shows the available devices:

```
[opencl:0] CPU : Intel(R) OpenCL 2.1 [2021.12.6.0.19_160000]
[cuda:0] GPU : NVIDIA CUDA BACKEND 0.0 [CUDA 11.4]
[host:0] HOST: SYCL host platform 1.2 [1.2]
```

- The device selector ranks the devices
- The choosen device is the one with the best score, if the best is negative, an exception is thrown

Error Handling

A lambda expression handler

```
auto exception_handler = [](const sycl::exception_list &exceptions) {
for (std::exception_ptr const &e : exceptions) {
        try {
                std::rethrow_exception(e);
        }
        catch (sycl::exception const &e) {
                std::cout << "Caught asynchronous SYCL exception: " << e.what() << std::endl;
        }
        catch (std::exception const &e) {
                std::cout << "Caught asynchronous STL exception: " << e.what() << std::endl;
        }
}
};
```

Error Handling

Then we construct a queue

```
sycl::queue q;
try {
        sycl::device dev = sycl::device(selector);
        q = sycl::queue(dev, exception_handler);
        q.single_task([](){}).wait_and_throw(); // To check that the device is live
}
catch (...) {
        sycl::device dev = sycl::device(sycl::host_selector());
        q = sycl::queue(dev, exception_handler);
        std::cout << "Warning: Expected device not found! Fall back on: "
                  << dev.get_info<sycl::info::device::name>() << std::endl;
        }
return q;
```

Porting CUDA using SYCL 2020 features

Day 4: SYCL Summer Sessions 2021

What's new in SYCL 2020

- Moving away from OpenCL
- Back-end notion
- Sub-groups (warps)
- Group algorithms
- Unified Shared Memory
- Atomics
- Reduction interface
- CTAD + C++20 template argument deduction from constructor

These features help us to more easily port our code from CUDA to SYCL

Warp-Level Primitives

Warp-level primitives allows inter-thread operations such as broadcast, reductions. It's the SIMT (single instruction, multiple thread) execution model. SYCL takes the execution model further by allowing these operations to be performed "block-wide" (see CUB)

CUDA

- __shfl_xor_sync
- __shfl_down_sync
- __shfl_up_sync
- __shfl_sync
- __all_sync
- __any_sync
- __reduce_sync

SYCL (group algorithms)

- permute_group_by_xor
- shift_group_left
- shift_group_right
- select_from_group
- all_of_group
- any_of_group
- reduce_over_group

Warp primitives - Differences

Missing from CUDA

"warp-size" dependant masks

- __ballot_sync
- __activemask
- __match_any_sync
- __match_all_sync

Bonus in SYCL

- exclusive_scan_over_group
- inclusive_scan_over_group
- joint_inclusive_scan
- joint_exclusive_scan
- joint_reduce

- Group functions use local memory for inter-sub-group communication, but the size cannot be queried, probably a bug in the spec (see issue)
- Poor performance on DPC++ when using groups.
- See group mask extension

Warp primitives – Example

Missing functions can easily be added to SYCL:

```
template<typename T>
T broadcast_leader(const sycl::sub_group &sg, T val) {
    return sycl::select_from_group(sg, val, 0);} //get `val` from warp_id=0


uint32_t ballot(const sycl::sub_group& sg, int predicate) {

    //assert(sg.get_local_range().size() <= 32) !
    size_t id = sg.get_local_linear_id();
    uint32_t local_val = (predicate ? 1u : 0u) << id;
    return sycl::reduce_over_group(sg, local_val, sycl::plus<>());}


q.parallel_for(sycl::nd_range<1>(32, 32),

    [=](sycl::nd_item<1> it) {

         sycl::popcount(ballot(it.get_sub_group(), 1))); // = 32
    }).wait();
```

Warp functions (group algorithms)

SYCL

```
template<typename T>
uint32_t match_any(const sycl::sub_group &sg, T val) {
    uint32_t size= sg.get_local_range().size();
    uint32_t found = 0;
    for (uint32_t i = 0; i < size; ++i) {
        const T other = sycl::select_from_group(sg, val, i);
        found |= (other == val ? 1u : 0u) << i;
    }
    return found;
}
```

PTX produced

```
LBB14_3:
        shfl.sync.idx.b32       %r28, %r2, %r33, 31, %r4;
        setp.eq.s32     %p4, %r28, %r2;
        selp.u32        %r29, 1, 0, %p4;
        shl.b32         %r30, %r29, %r33;
        or.b32          %r32, %r30, %r32;
        add.s32         %r33, %r33, 1;
        setp.ne.s32     %p5, %r3, %r33;
        @%p5 bra        LBB14_3;
        ret;
```

Kernel ranges

- Caveat: Unlike with CUDA, a kernel dimension in a **`sycl::range`** cannot be equal to 0.
- Index flipping: `dim3(x_max, y_max z_max)` ➔ `sycl::range<3>(z_max, y_max, x_max)`
  - Continuous dimension is the last: C-style indexing
- Best kernel parameters can depend on device: CPU v. GPU
  - 1 CPU thread (on openCL) = 1 work-group = 1 streaming multiprocessor
  - On GPU consider maxing out the number of **work-items** then adapt work-groups count
  - On CPU depends on the back-end, openMP v. openCL …
- Queries (can return anything from 0 to 2^{64} -1):

```
q.get_device().get_info<sycl::info::device::max_compute_units>();
q.get_device().get_info<sycl::info::device::max_work_group_size>();
```

- Don’t need to manually compute ids (DPCT does not perform the conversion)
- Avoid passing kernel range informations outside of the **`nd_range`** variable
- Best `block-size` might be different given change in register pressure…

Kernel queries

The kernel descriptor `kernel_device_specific::work_group_size` returns

> *"the maximum number of work-items in a work-group that can be used to execute a kernel on a specific device".*

Size guaranteed to work and smaller than `info::device::max_work_group_size` (because it depends on register count, stack-frame size, etc). Potentially giving up on performance for safety.

If the kernel range is decoupled from work size, we could use:

```
template<typename KernelName>
sycl::nd_range<1> get_max_launch_range(sycl::queue &q) {
    sycl::kernel_id id = sycl::get_kernel_id<KernelName>(); // Throws if not found
    auto kernel = sycl::get_kernel_bundle<sycl::bundle_state::executable>(q.get_context()).get_kernel(id);

    size_t max_items = kernel.get_info<sycl::info::kernel_device_specific::work_group_size>(q.get_device());
    size_t max_groups = q.get_device().get_info<sycl::info::device::max_compute_units>();
    return {sycl::range<1>(max_items * max_groups), sycl::range<1>(max_items)};
}
```

Cooperative Groups & Occupancy

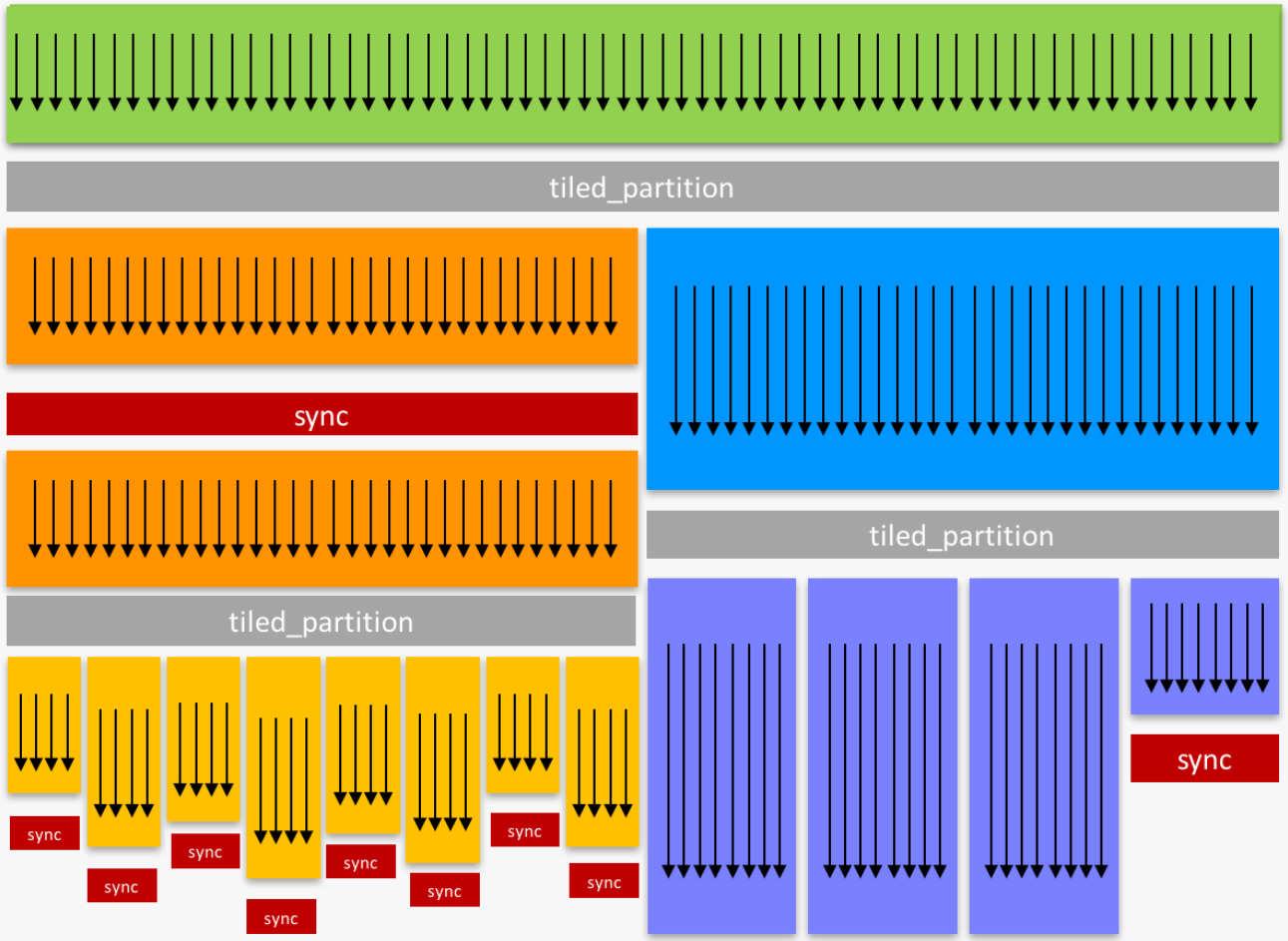


Illustration from Nvidia Developer Blog, Cooperative Groups by Mark Harris and Kyrylo Perelygin

Cooperative Groups & Occupancy

- Flexible groups for grouping threads together, see SYCL sub-groups.
- Whole grid and multi-gpu synchronisation primitives: cannot be ported safely to SYCL
  - No forward-progress guarantees between work-groups (cannot compute occupancy)
  - The device can shared between processes (See Nvidia MPS) so a **kernel submission attribute** is required to block that.
  - Kernels and memory copies can be ran concurrently! (DPC++ use async functions)
  - `acquire_release` memory order not supported but needed for device-wide barriers

Needed:

- Device descriptor for work-group forward progress support.
- Kernel query for “best occupancy” and “maximum nd_range” (local mem size, register, …)
- CUDA exposes all the necessary API calls so we should be able to implement it, but what about other devices? But that won’t solve the lack of **dynamic parallelism** (launching kernels from within kernels)

Atomics

Atomics can be ported from CUDA, but the syntax changes a lot. SYCL's syntax 2020 is based on C++'s atomic_ref<T>.

- System-wide: CUDA suffix: `atomicOp_system => memory_scope::system_group`
- Device-wide `atomicOp_ => memory_scope::device`
- Block-wide: `atomicOp_block => memory_scope::work_group`

```
using atomic_ref_t = ATOMIC_REF_NAMESPACE::atomic_ref<
          Type,
        ATOMIC_REF_NAMESPACE::memory_order::relaxed, // acq_rel not supported
        ATOMIC_REF_NAMESPACE::memory_scope::device,
        sycl::access::address_space::global_space // {local/global/globa_device}_space
>;
```

Debugging

SYCL allows printing to stdout from a kernel using the sycl::stream class

- But it requires passing an extra parameter (tideous)
- `printf` is now supported on the CUDA back-end of DPC++
- …but printf is not part of SYCL
- sycl::stream and printf increase dramatically the size of a kernel and is not recommended (see 4.16.4. Performance note)

If possible, use the host device to debug, but not everything is supported. The openMP back-end of hipSYCL supports most of SYCL 2020 so it can be interesting too.

```
q.submit([&](sycl::handler &h) {
      sycl::stream os(1024, 768, h); // Buffer_bytes,Buffer_per_group
      h.parallel_for(1, [=](sycl::id<1> i) {
          os << i << "\n";
      });
}).wait(); // Don’t forget to .wait() before leaving main()!
```

oneMKL & oneDNN

- Both supported on SYCL with the CUDA back-end
- Leverage cuDNN and cuBLAS
- Undocumented. Setup script here (builds dependencies: oneTBB & lapack-ref)
- Building oneMKL with CUDA 11+ requires a patch: see PR
- oneMKL testing not supported if curand & cublas
- oneMKL != Intel oneMKL (different namespaces). Functions are in onneapi::mkl::blas::column_major::

Memory

Day 4: SYCL Summer Sessions 2021

Memory types

CUDA

- Constant
- Global (grid-wide)
- Shared (block-wide)
- Local (“stack-frame”)
- Private (“registers”)

SYCL

- target::constant_buffer
- target::global_buffer
- target::local
- -
- Private

Unified Addressing & Introspection

SYCL offers Unified Shared Memory with malloc_shared/host/device.

ZFP back-end has an API that does not allow passing information on data pointers. It uses cuda introspection functions such as `cudaPointerGetAttributes` to detect if a memory copy is needed.

SYCL offers introspection on USM pointers too:

- sycl::device devic get_pointer_device(ptr, context);
- sycl::usm::alloc get_pointer_type (ptr, context);

On DPC++, they returns the wrong information if `context.is_host()`. Unsafe workaround here.

USM Smart Pointers

SYCL Unifier Shared Memory pointer's lifetime can be managed with STL's smart pointers. These pointer can also be "decorated" with size and address space information

```
template<typename T>
struct usm_deleter {
        sycl::queue q_;

        explicit usm_deleter(sycl::queue q) : q_(std::move(q)) {}

        void operator()(T *ptr) const noexcept {
                if (ptr) sycl::free(ptr, q_);
        }
};
```

USM Smart Pointers

```
template<typename T, sycl::usm::alloc location>
class usm_unique_ptr : public std::unique_ptr<T, usm_deleter<T>> {
private:
          const size_t count_;
public:
          [[nodiscard]] usm_unique_ptr(size_t count, sycl::queue q)
                    :std::unique_ptr<T,usm_deleter<T>>(sycl::malloc<T>(count, q, location), usm_deleter<T>{q}),
                      count_(count) { }


          [[nodiscard]] inline sycl::span<T> get_span() const noexcept {
                    return {std::unique_ptr<T, usm_deleter<T>>::get(), count_};
          }
};
auto ptr = usm_unique_ptr<int, alloc::device>(10, my_queue);
```

sycl::span<T> provides an implementation of std::span<T> introduced in C++20 and acts like an “array_view” allowing memory safety. You can also get sycl::multi_ptr<> from that implementation, see the rest

Multi-device asynchronous kernels

The hashing library is embarassingly parallel and compute intensive: good candidate to experiment with super-devices: harder than it seems because constant buffers can have blocking destructors:

- You cannot leave a kernel submission untill a computation is finished to launch the kernel on other devices (`std::thread` works but feels wrong)
- Asynchronous ressource management?

A Solution:

- Micro-benchmark to evaluate a queue performance on a given dataset to better distribute the workload.
- Move from “standalone kernels” to class methods + kernels: the class holds the buffers and passes a reference to the kernel.
- Multi-device kernel submission returns a handle that holds (so std::moved in) the ressouces (usm_unique_ptr, buffers, events) and offers the sycl::event api (with deleted copy and asignement operators).

Semantics example:

```
hash::sha3<512> hasher({{cpu_q,  cpu_speed}, {cuda_q, gpu_speed}});
{
        auto e = hasher.hash(data_in, in_block_size, hash_out, count); // Runs in parallel
        // do something else and forgot to wait
} // joins the queues and cleans-up the ressources
```

Local & Constant

- Local Memory (CUDA Shared)
  - 4 bytes per bank
  - 4 bytes alignement & min 4 bytes size to avoid bank conflicts
  - Padding such as:

    GCD(sizeof(struct)/bank_size , bank_count) == 1
  - Adding a single float results in a 3X bandwidth increase (see benchmark)

- Constant memory
  - Cannot be “inlined” unlike on CUDA, see next slide.
  - Avoid accessing different locations from different threads

“Inlining” constant data

CUDA

```
__constant__ static int arr_1[1024] = {...};
__device__ void a_function(int *a){
        #pragma unroll 1024
        for(int i = 0; i < 1024; ++i){
                a[i]=arr_1[i]; // some processing
        }
}
```

The loop will be unrolled and there will be no accesses made to the constant buffer.

SYCL

```
static const int arr_1[1024] = {...}; // reused
void a_function(int *a, constant_accessor acc){
        #pragma unroll 1024
        for(int i = 0; i < 1024; ++i){
                a[i]=acc[i]; // some processing
        }
}
```

The user will create a buffer and an accessor. The loop unrolling can be performed, but memory accesses will not be avoided.

Optimization and performance tips

Day 4: SYCL Summer Sessions 2021

Loop Unrolling

- LLVM has a more aggressive loop unrolling for GPU
- Not as much as CUDA. Check the assembly
- Cannot unroll “constant memory”.
- Microsoft Antares: 40 Gflops → 3700 Gflops just by adding several `#pragma {no}unroll`

“Inlining” constant data

```
static constexpr std::array<int, 1024> arr_1 = {...}; // reused
static constexpr std::array<float 512> arr_2 = {...}; // reused

template <bool first>
void a_function(int *a){
        static constexpr auto array = [](){
                if constexpr (first) return arr_1;
                else return arr_2;
        }();

        #pragma unroll
        for(int i = 0; i < 1024; ++i){
                a[i]= array[i]; // some processing
        }
}
```

“Registerization” of private arrays

```
struct my_struct { // sizeof == 120
    int32_t i = 1, j = 2;
    int array[10] = {0};
    sycl::id<3> some_coordinates{0, 0, 0}, some_more{1, 0, 2};
    sycl::id<3> even_more{0, 3, 1};
};

#pragma unroll
for(int i = 0; i < 10; ++i){
         (void) a_struct.array[i];   // OK
         (void) a_struct.array[i%10]; // Will ALWAYS send ‘a_struct’ to the stack-frame 😟
}
```

“Registerization” of private arrays

- Private arrays (int[…]) stored in stack frame (global memory)…
- All threads must access the same index in the private array (they are stored in a interleaved manner so this will coalesce accesses!)
- If a class contains such an array, it is also stored on the stack.
- …unless “registerization” possible (all accesses must be deduced at compile time!, like #pragma unroll)
- Use shared memory whenever possible (especially things like state-machines)
- Avoid taking the address of a register have no address. LLVM do optimise that away, but can fail to. Data will end-up on the stack.
- sycl::range, id, nd_item, item, nd_range uses array internally on hipSYCL and DPC++! Avoid accessing them with a variable index. Having a template subscript operator or constexpr parameters would solve the issue. That’s how CUDA get’s away with these issues.

```
[i](item<3> it){ *i = it[ *i % 3 ]; }
```

- DPC++ issue and Patch here.

Work-around ?

2x performance on MD2 hash

Hopefully we can force the registerization of our data through a proxy function, `runtime_index_wrapper`, you can find implemented in the support library. Supports:

- **`sycl::id, std::vector, std::array, sycl::vec`**, and C-style arrays: size automatically deduced
- anything that has a subscript operator: user must provide the **`<size>.`**

Two modes:

- Linear: when items from a sub-group will access different indices: O(n) access time
- Logarithmic: O(log(n)) access time, but performs worse (See Andrei Talk)

```
q.parallel_for(sycl::range<3>({32, 32, 32}), [=](sycl::id<3> id) {
          int arr[16];
          #pragma unroll
        for (int i = 0; i < 16; ++i)
              runtime_index_wrapper_log(arr, i % 16) = *ptr; //No stack usage for arr[16]
        *ptr = runtime_index_wrapper(id, *ptr % 3); //O(n=3) look-up time
  }).wait();
```

Before/After (that fits on a slide, so not the previous example)

```
*ptr = data.some_coordinates[ *ptr %3 ];
```

```
    ld.param.u64    %rd1, [_ZTS11kernel_name_param_0];
    add.u64         %rd3, %SPL, 0;
    add.s64         %rd4, %rd3, 48;
    mov.u64         %rd5, 0;
    st.local.u64    [%rd3+48], %rd5;
    st.local.u64    [%rd3+56], %rd5;
    st.local.u64    [%rd3+64], %rd5;
    mov.u64         %rd6, 1;
    st.local.u64    [%rd3+72], %rd6;
    st.local.u64    [%rd3+80], %rd5;
    mov.u64         %rd7, 2;
    st.local.u64    [%rd3+88], %rd7;
    st.local.u64    [%rd3+96], %rd5;
    mov.u64         %rd8, 3;
    st.local.u64    [%rd3+104], %rd8;
    st.local.u64    [%rd3+112], %rd6; // 120 bytes stack
    ld.global.u32   %r1, [%rd1];
    mul.hi.u32      %r2, %r1, -1431655765;
    shr.u32         %r3, %r2, 1;
    mul.lo.s32      %r4, %r3, 3;
    sub.s32         %r5, %r1, %r4;
    mul.wide.u32    %rd9, %r5, 8;
    add.s64         %rd10, %rd4, %rd9;
    ld.local.u64    %rd11, [%rd10];
    st.global.u32   [%rd1], %rd11;
    ret;
```

```
*ptr = magic_wrapper(data.some_coordinates, *ptr % 3);
```

```
    ld.param.u64    %rd1, [_ZTS11kernel_name_param_0];
    mov.u32         %r1, 0;
    st.global.u32   [%rd1], %r1;
    ret;
```

```
struct my_struct { // sizeof == 120
    int32_t i=1, j=2;
    int array[10]={0};
    sycl::id<3> some_coordinates{0,0,0}, more{1,0,2}, even_more{0,3,1};
};

auto ptr = malloc_shared<uint>(1, q);
q.single_task([=]() {
    my_struct data{};
    *ptr = data.some_coordinates[*ptr % 3];
}).wait();

q.single_task([=]() {
    my_struct data{};
    *ptr = magic_wrapper(data.some_coordinates, *ptr % 3);
}).wait();
```

Template & Reflexion

- Move constant data/control flow to template parameters. This allows to tailor kernel for specific computation, removes useless function calls that are preventing optimisations (imagine you have somewhere a function that uses a runtime indice, having it in the control flow can throw your whole dataset to the stack-frame).
  - Kernels more specialized: reduction of the hot path, happier instruction cache and branch predictor on CPU
  - More optimisations like constant folding or unrolling
  - +10% on ZFP and SHA-3
- Use C++20 lambdas for reflexion:

```
template <typename T>
class MyClass {
    using storage_type = decltype([]() {
            if constexpr(sizeof(T) <= 4) {
                    return uint32_t{};
            } else {
                    return uint64_t{};
            }
    }());
    // Rest of the class
};
```

Strength reduction

Replacing instructions by ”cheaper” ones

- No division instructions in SASS
- Emulated instructions: IEEE754 (very)slow
- `-ffast-math` => `div.approx.f32` but no `div.approx.f64`!
- Use constants
- sycl::half (but very very bad performance on CPU/OpenCL! so see previous slide)
- SYCL uses `size_t` for indexing, and device type width must match host’s width :

```
mov.u32         %r4, %ctaid.x;
cvt.u64.u32     %rd27, %r4; //
```

Solution: cast index identifiers to `uint`. In Microsoft Antares it results in a 4% performance increase! See `fsycl-id-queries-fit-in-int`

CUDA flags equivalents

CUDA

- --ftz=true (flush denorms to 0)
- --prec-div=false --prec-sqrt=false
- --fmad=true (fused mad)
- --maxrregcount 64

SYCL

- -fcuda-flush-denormals-to-zero
- -fcuda-approx-transcendental
- -ffp-contract=fast
- -Xcuda-ptxas --maxrregcount=64
- -fsycl-id-queries-fit-in-int
- Re-enabling precise math with :
  `#pragma float_control(precise, on)`

Missing SYCL intrinsics ?

- Prefetching from a SYCL device?
- Check namespaces sycl::ext::intel and sycl::ext::oneapi
- Check DPC++ SYCL Extensions here
- Finally: Inline some PTX 😬

```
template<typename T>
void static inline prefetch_constant(const T *ptr) {
#if defined (__NVPTX__) && defined(__SYCL_DEVICE_ONLY__)
  asm("prefetchu.L1 [%0];" :: "l"(ptr)); // From PTX ISA manual
#else
  (void)ptr;
#endif
}
```

- And now you can prefetch to the uniform cache!

Counterintuitive-ness

In the reduction kernel example (99% of CUB performance!), computing small reductions over data living in device memory is more expensive than copying that memory to the host and performing a serial reduction on the CPU.

- Cheaper to enqueue a several memcpy that launch a kernel
- If kernel operates at memcpy speeds, probably not worth offloading
- **Adding synchronisation** can improve coalesced accesses by using the whole cache lines
- Performing extra computation to avoid branch divergence!
- See the amazing Andrei Alexandrescu's talk "Speed Is Found In The Minds of People": **more computation/instructions can lead to better performances:**

Profiling & Benchmarking

- Profiling with `nvprof` works except if `malloc_shared` used. With named kernels looks better.
- NVIDIA Tools supported: memcheck, racecheck, synccheck and initcheck
- First kernel submission VERY slow: queue setup overhead
- DPC++ will use std::chrono, but that will include SYCL runtime overhead
- Use SYCL profiling to gather precise timings (closer to CUDA events too):

```
sycl::queue q{sycl::gpu_selector{}, sycl::property::queue::enable_profiling{}};
sycl::event e = q.submit(…); // calls to profiling info will wait implicitly
size_t nanoseconds =  e.get_profiling_info<sycl::info::event_profiling::command_end>()
                     -e.get_profiling_info<sycl::info::event_profiling::command_start>();
```

Key Takeaways

- True “generic” programming is hard, SYCL needs to address that. Proposals such as `if devconstexpr` could help. I would prefer function attributes so the device compiler chooses the right specialisation depending on the device target
- When porting CUDA or C++ to SYCL be very careful on registerization and unrolling: it can lead to surprisingly bad performance.
- Don’t fall for the 64 bits trap (yes, size_t is, but not your GPU, threads ids, …)
- Templates are nice.
- Have unit tests you run on various platforms: undefined behaviour is different on CPU and GPU.
- If you publish benchmarks, use the profiling API and not std::chrono.
- Compiler are chaotic systems: adding a function call can reduce a code to a `no-op`.

We're Hiring!

codeplay.com/careers/

codeplay

Enabling AI to be Open, Safe & Accessible to All

Questions?

@codeplaysoft

info@codeplay.com

codeplay.com