codeplay®

Enabling AI to be Open, Safe & Accessible to All

DPC++ on Nvidia GPUs

codeplay

Enabling AI to be Open, Safe & Accessible to All

DPC++ on Nvidia GPUs

Ruyman Reyes Castro
CTO

Stuart Adams,
Staff Software Engineer

IXPUG/TACC

Products

Acoran

Integrates all the industry standard technologies needed to support a very wide range of AI and HPC

ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR-V™, HSA™ and Vulkan™

Markets

High Performance Compute (HPC)
Automotive ADAS, IoT, Cloud Compute
Smartphones & Tablets
Medical & Industrial

Technologies: Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

Company

Leaders in enabling high-performance software solutions for new AI processing systems

Enabling the toughest processors with tools and middleware based on open standards

Established 2002 in Scotland with ~80 employees

Customers

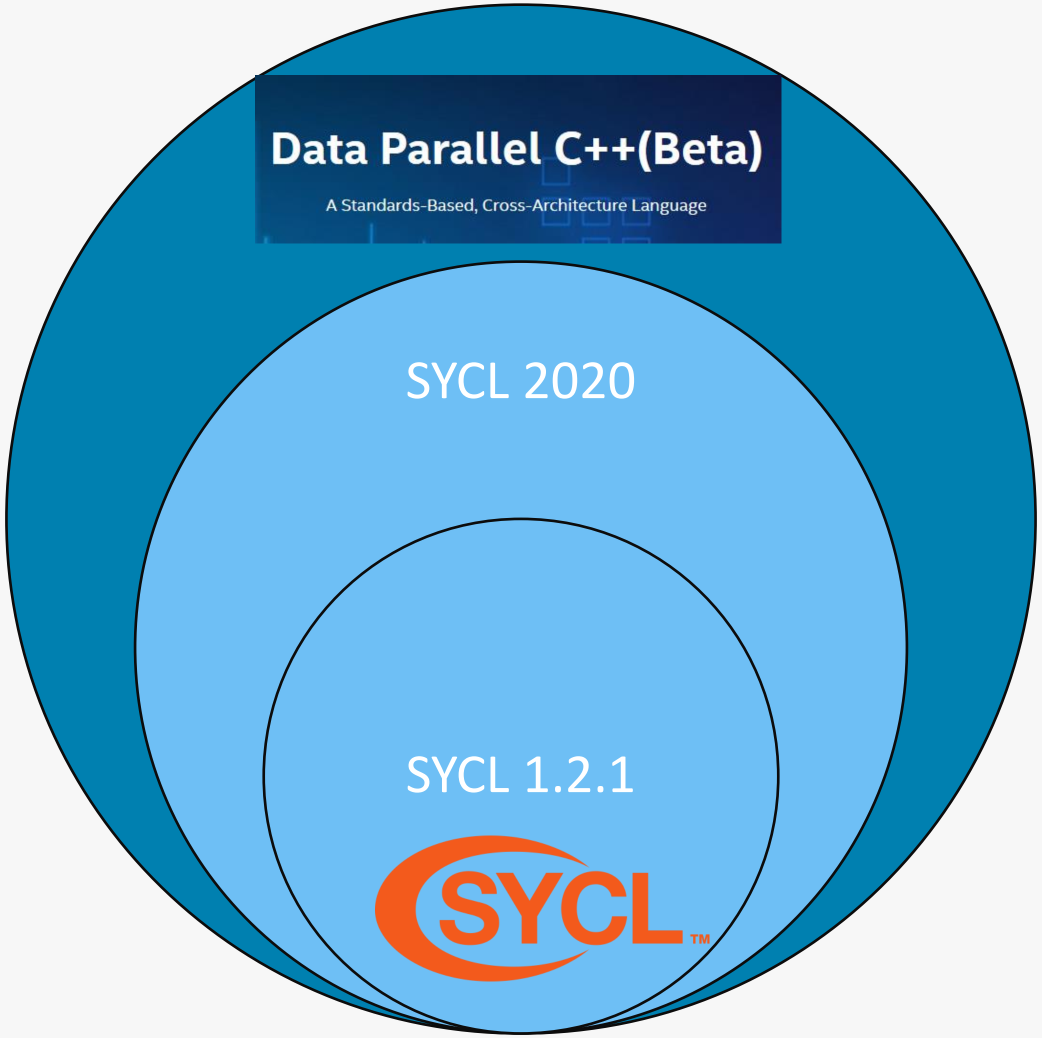
arm, Broadcom, CEVA, Imagination, intel, Renesas, Synopsys

And

Summary

- What is DPC++ and SYCL
- Using *SYCL for CUDA*
- Design of *SYCL for CUDA*
- Implementation of *SYCL for CUDA*
- Interoperability with existing libraries
- Using oneMKL on CUDA
- Conclusions and future work

What is DPC++?



- Data Parallel C++ (DPC++) is an open, standards-based alternative to single-architecture proprietary languages, part of oneAPI spec.
- It is based on C++ and SYCL, allowing developers to reuse code across hardware targets (CPUs and accelerators such as GPUs and FPGAs) and also perform custom tuning for a specific accelerator.

Codeplay and SYCL

- Codeplay has been part of the SYCL community from the beginning
- Our team has helped to shape the SYCL open standard
- We implemented the first conformant SYCL product

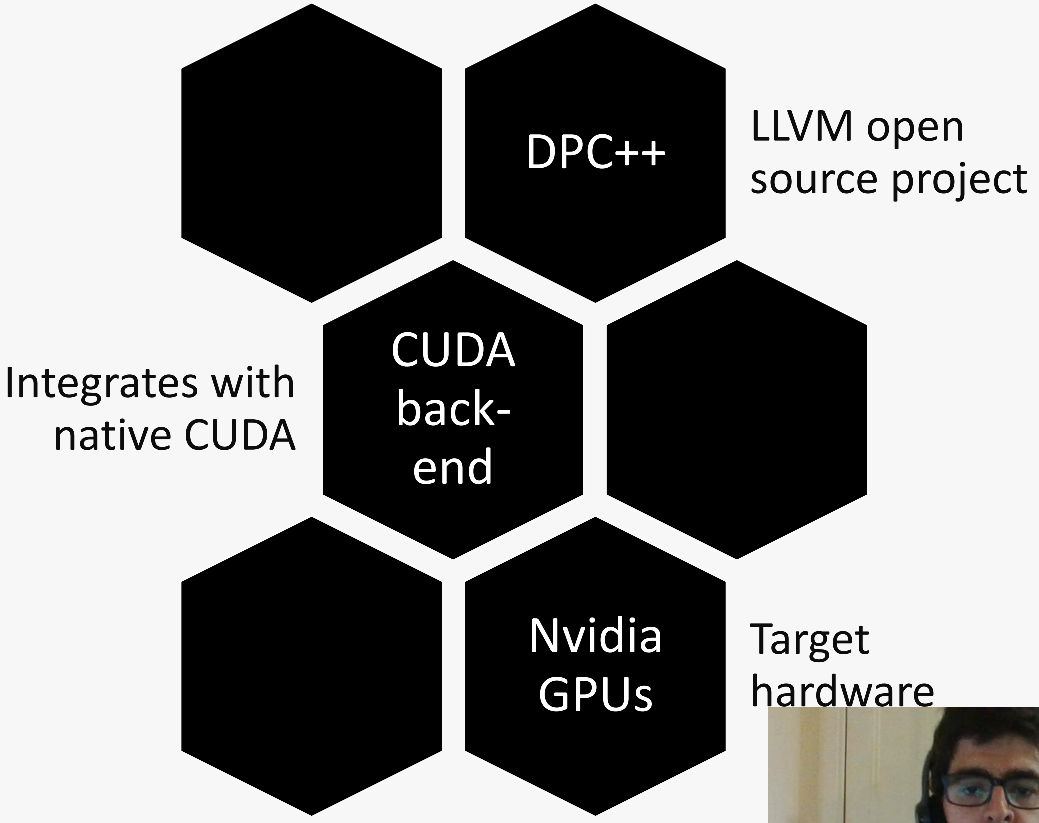
Acoran
The Open Acceleration Platform

ComputeCpp™

SYCL

Codeplay and DPC++

- Our contribution to the open source LLVM project adds support for Nvidia GPUs
- Uses directly CUDA through a plugin mechanism
- Codeplay will help the upstreaming effort so SYCL support is available on clang



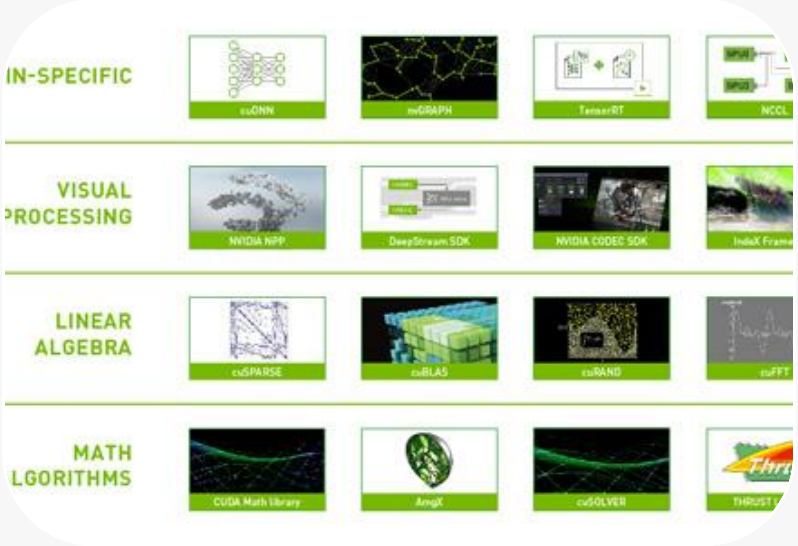
Using DPC++ with CUDA and Nvidia

Incremental porting

- Port CUDA applications to SYCL one kernel at a time

oneAPI Apps on Nvidia

- Existing oneAPI applications can run unmodified on NVIDIA hardware



Access CUDA libraries

- oneAPI / SYCL applications can call native CUDA libraries directly from DAG

Incremental porting

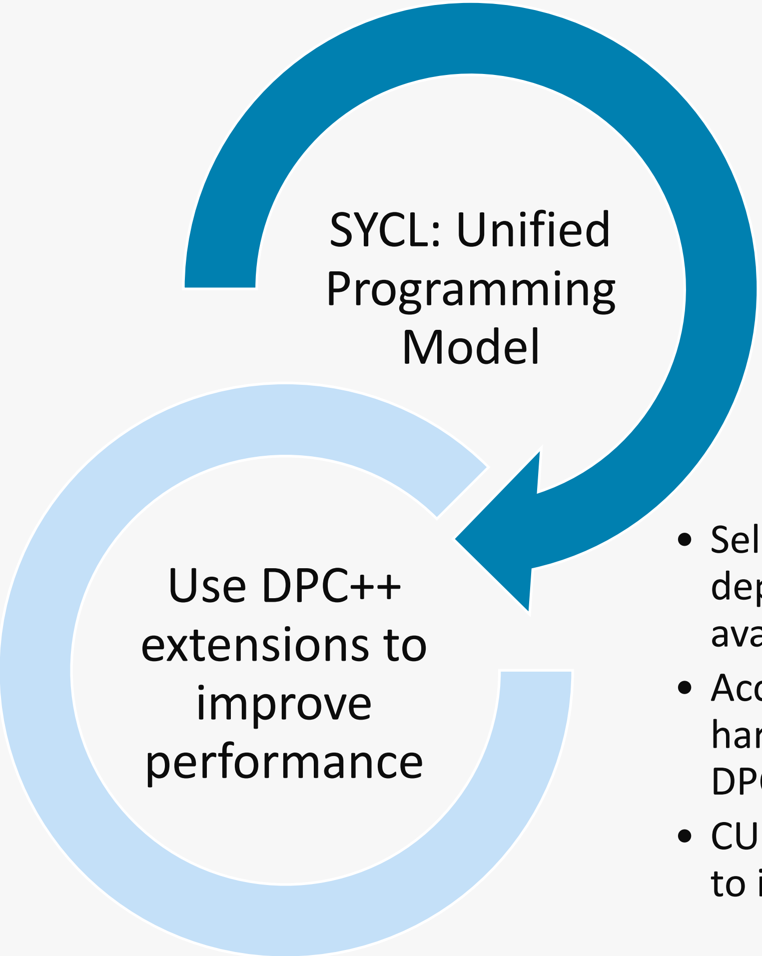
Migrate host code to SYCL and keep your CUDA kernels

Porting to SYCL/DPC++

- Measure performance at any stage using existing CUDA tools
- Can compile your SYCL application with LLVM CUDA
- Replace one CUDA kernel with a SYCL kernel, test and run another

```
// Dispatch a command group with all the dependencies
myQueue.submit([&](handler& h) {
  auto accA = bA.get_access<access::mode::read>(h);
  auto accB = bB.get_access<access::mode::read>(h);
  auto accC = bC.get_access<access::mode::write>(h);

  h.interop_task([=](interop_handler ih) {
    auto d_a = reinterpret_cast<double*>(ih.get_mem<backend::cuda>(accA
    auto d_b = reinterpret_cast<double*>(ih.get_mem<backend::cuda>(accB
    auto d_c = reinterpret_cast<double*>(ih.get_mem<backend::cuda>(accC

    int blockSize, gridSize;
    // Number of threads in each thread block
    blockSize = 1024;
    // Number of thread blocks in grid
    gridSize = (int)ceil((float)n / blockSize);
    // Call the CUDA kernel directly from SYCL
    vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
  });
});
```

Run oneAPI applications on CUDA platforms

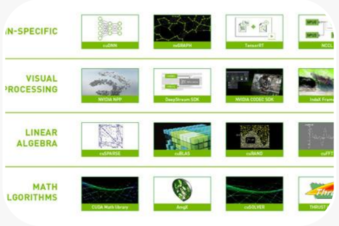


- Run on any platform unmodified
- Identify platform-specific gaps
- Target multiple devices from the same application

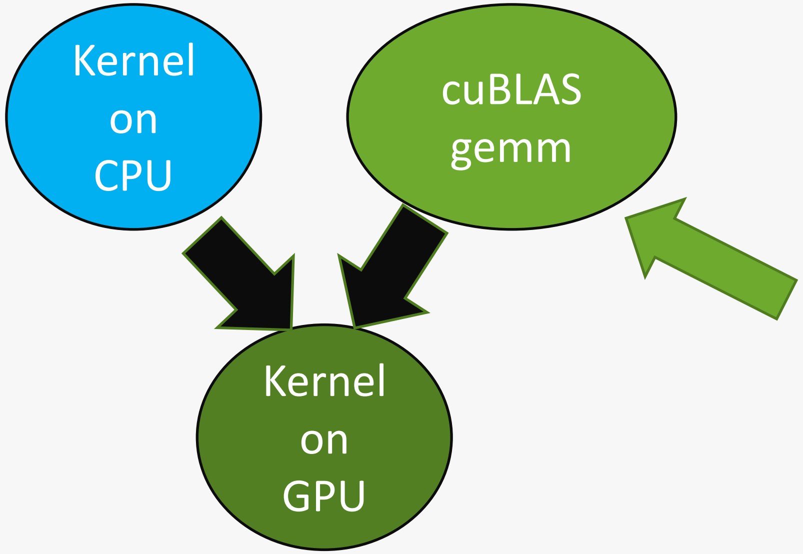
- Select different kernels depending on the available platform
- Access advanced hardware features via DPC++ extensions
- CUDA specific extensions to improve performance

The only code required is a CUDA selector to tell DPC++ to use CUDA devices

```
sycl::queue myQueue{CUDASelector()};

// Command Group creation
auto cg = [&](sycl::handler &h) {
  const auto read_t = sycl::access::mode::read;
  const auto write_t = sycl::access::mode::write;

  auto a = bufA.get_access<read_t>(h);
  auto b = bufB.get_access<read_t>(h);
  auto c = bufC.get_access<write_t>(h);

  h.parallel_for<vec_add>(VecSize,
                    [=](sycl::id<1>
};

myQueue.submit(cg);
```



Use CUDA libraries on SYCL dependency graphs




```
q.submit([&](handler &h) {
  auto d_A = b_A.get_access<sycl::access::mode::read>(h);
  auto d_B = b_B.get_access<sycl::access::mode::read>(h);
  auto d_C = b_C.get_access<sycl::access::mode::write>(h);

  h.interop_task([=](sycl::interop_handler ih) {
    cublasSetStream(handle, ih.get_queue<backend::cuda>());

    auto cuA = reinterpret_cast<float *>(ih.get_mem<backend::cuda>(d_A));
    auto cuB = reinterpret_cast<float *>(ih.get_mem<backend::cuda>(d_B));
    auto cuC = reinterpret_cast<float *>(ih.get_mem<backend::cuda>(d_C));

    CHECK_ERROR(cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, WIDTH, HEIGHT,
                            WIDTH, &ALPHA, cuA, WIDTH, cuB, WIDTH, &BETA,
                            cuC, WIDTH));
  });
});
```

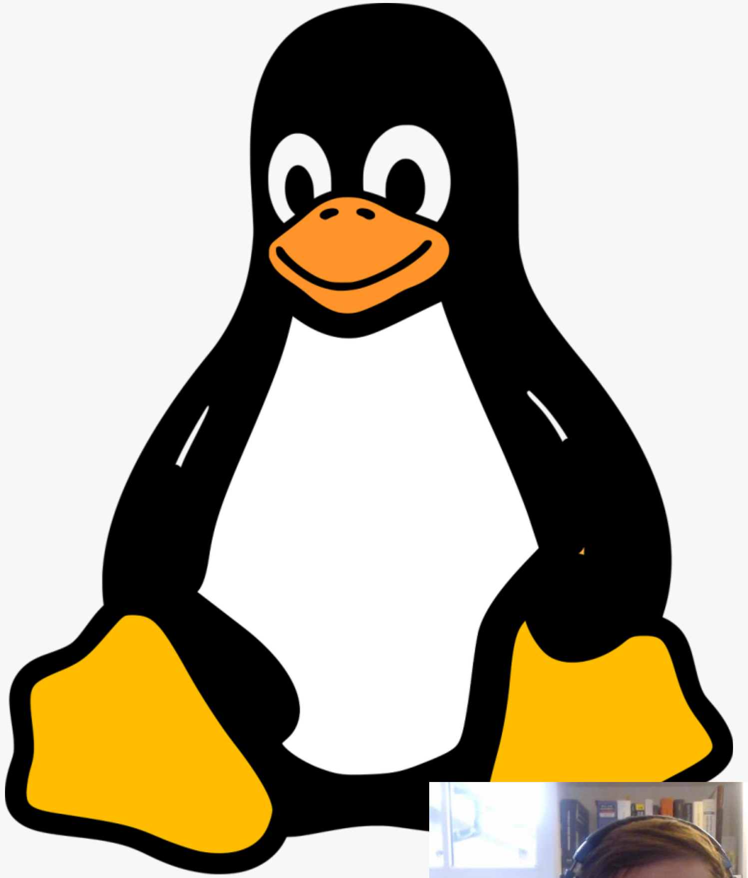
Call to cublasSgemm scheduled alongside the other kernels

DPC++ implements Codeplay's SYCL extensions to ca
libraries from SYCL Directed Acyclic Graphs

Using *SYCL for CUDA*

Requirements

- Linux (Ubuntu 18.04 preferred)
- CUDA 10.1 or newer
- Hardware sm_50 or above



How do you get it?

- Currently in ongoing development, see https://github.com/intel/llvm for up-to-date instructions
- DPC++ releases don't currently include CUDA support.
- The project must be built from source to include CUDA support.
- Build instructions are in the "Getting Started Guide"

Build DPC++ toolchain with support for NVIDIA CUDA

There is experimental support for DPC++ for CUDA devices.

To enable support for CUDA devices, follow the instructions for the Linux DPC++ toolchain, but add the `--cuda` flag to `configure.py`

Enabling this flag requires an installation of CUDA 10.1 on the system, refer to NVIDIA CUDA Installation Guide for Linux.

Currently, the only combination tested is Ubuntu 18.04 with CUDA 10.2 using a Titan RTX GPU (SM 71), but it should work on any GPU compatible with SM 50 or above.

Using *SYCL for CUDA*

- Compile your code using the CUDA target triple

No changes required to your SYCL code

```
clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda-sycldevice \
  simple-sycl-app.cpp -o simple-sycl-app-cuda.exe
```

- Run your application with the CUDA backend enabled

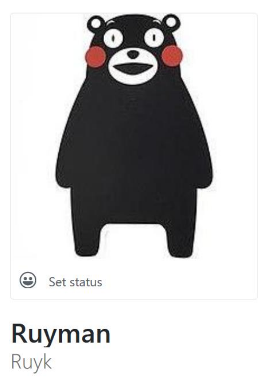
```
SYCL_BE=PI_CUDA ./simple-sycl-app-cuda.exe
```

Environment variable used by default device selection

Looking for Example Code?

- We’ve created examples of how to use the CUDA back-end!
  - See how to write a SYCL device selector for CUDA devices.
  - See how to interop with native CUDA libraries.
  - As a bonus, compare SYCL code with the CUDA equivalent.
  - We are adding more examples soon.

https://github.com/codeplaysoftware/SYCL-For-CUDA-Examples



Design of *SYCL for CUDA*

SYCL for CUDA

SYCL 1.2.1 was intended for OpenCL 1.2

- If a SYCL 2.2 ever existed, it was based on OpenCL 2.2
- What could be a good alternative target to demonstrate SYCL as a High Level Model?
- **Let’s have an open discussion about SYCL for non-OpenCL!**

Sure let’s do Vulkan!

- Not that simple, SYCL was designed for compute rather than graphics
- There is already a potential route via clspv + clvk

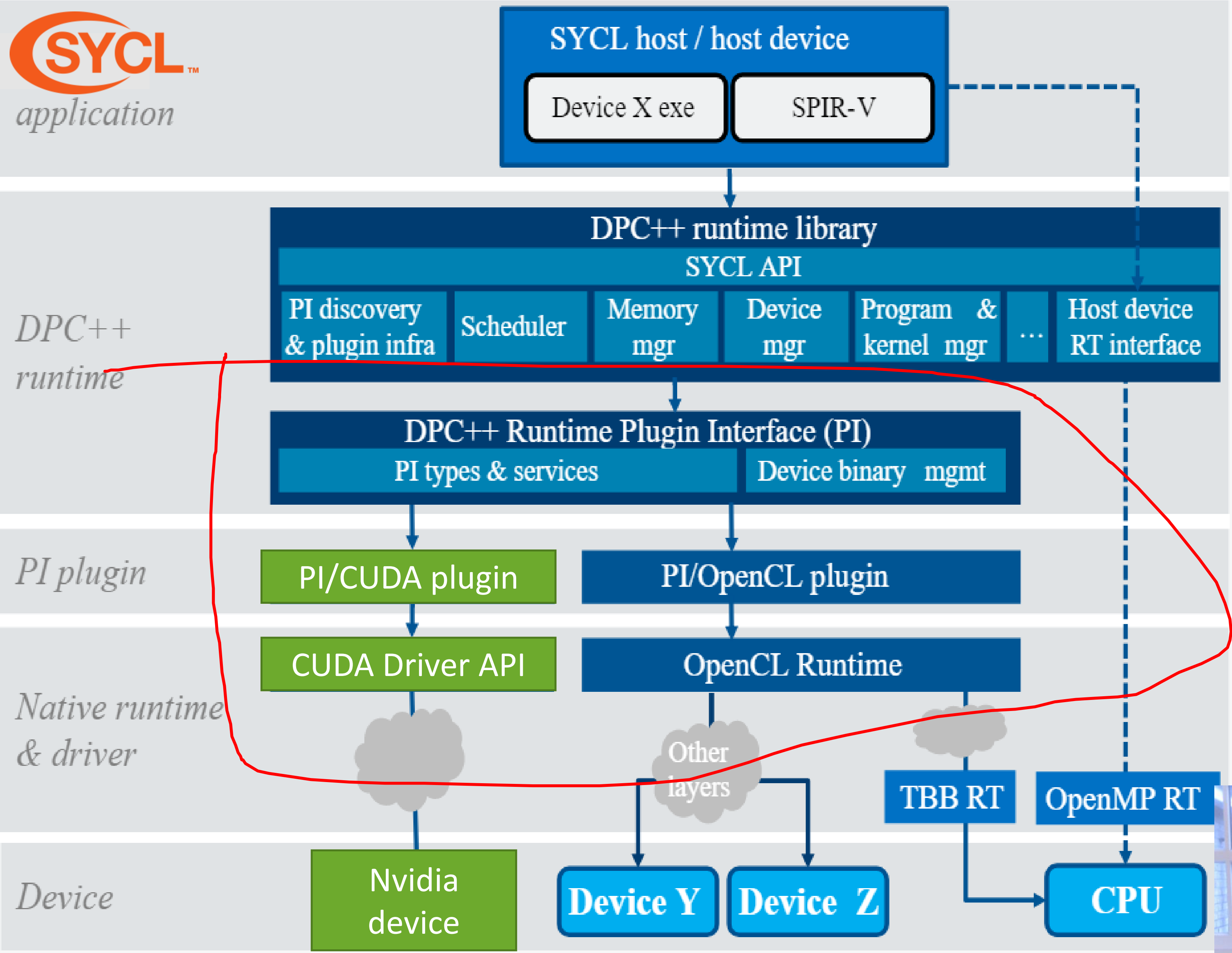
Have you heard about CUDA?

- Existing OpenCL + PTX path (available on ComputeCpp) works but performa
better
- Native CUDA support is the best solution to expand the ecosystem

SYCL 1.2.1 on CUDA

- What works?
  - Platform model (Platform/Device/Context)
  - Buffers, copy
  - NDRange kernels
- What is broken?
  - Interoperability
    - Needs to be revised to allow both OpenCL and CUDA handles to be accessed.
    - Necessary, so implemented via **`get_native`** extension.
  - Images and samplers
    - CUDA images are sampled on construction.
    - SYCL/OpenCL Images are sampled in the kernel.
  - SYCL program class
    - OpenCL compilation model does not match CUDA.

Implementation of *SYCL for CUDA*


SYCL
application
SYCL host / host device
Device X exe
SPIR-V
DPC++ runtime
DPC++ runtime library
SYCL API
PI discovery & plugin infra
Scheduler
Memory mgr
Device mgr
Program & kernel mgr
...
Host device RT interface
DPC++ Runtime Plugin Interface (PI)
PI types & services
Device binary mgmt
PI plugin
PI/CUDA plugin
PI/OpenCL plugin
Native runtime & driver
CUDA Driver API
OpenCL Runtime
Other layers
TBB RT
OpenMP RT
Device
Nvidia device
Device Y
Device Z
CPU

The PlugIn (PI) API

- Decoupling the SYCL runtime from OpenCL
  - Introduce a new C API, mapping closely to OpenCL API design.
  - Acts as a level of indirection between SYCL runtime and the target platform.

- Plugins
  - Each back-end is implemented as a shared library.
  - SYCL runtime can load and use multiple plugins at once.
  - The programmer can control which plugins SYCL runtime will use.

The PI API

PI

sycl/include/CL/sycl/detail/pi.h

```
__SYCL_EXPORT pi_result piMemBufferCreate(pi_context context,
                                          pi_mem_flags flags, size_t size,
                                          void *host_ptr, pi_mem *ret_mem);
```

OpenCL

sycl/plugins/opencl/pi_opencl.cpp

```
pi_result piMemBufferCreate(pi_context context, pi_mem_flags flags, size_t size,
                            void *host_ptr, pi_mem *ret_mem) {
  pi_result ret_err = PI_INVALID_OPERATION;
  *ret_mem = cast<pi_mem>(clCreateBuffer(cast<cl_context>(context),
                                         cast<cl_mem_flags>(flags), size,
                                         host_ptr, cast<cl_int *>(&ret_err)));

  return ret_err;
}
```

CUDA

sycl/plugins/cuda/pi_cuda.cpp

```
pi_result cuda_piMemBufferCreate(pi_context context, pi_mem_flags flags,
                                 size_t size, void *host_ptr,
                                 pi_mem *ret_mem) {
  // Need input memory object
  assert(ret_mem != nullptr);
  // Currently, USE_HOST_PTR is not implemented using host register
  // since this triggers a weird segfault after program ends.
  // Setting this constant to true enables testing that behavior.
  const bool enableUseHostPtr = false;
  const bool performInitialCopy = (flags & P
    || ((flags & PI_MEM_FLAGS_HOST_PTR_USE)
  pi_result retErr = PI_SUCCESS;
  pi_mem retMemObj = nullptr;
```

PI CUDA Limitations

- Single device per context.
- No images.
    - CUDA images do not map to OpenCL or SYCL 1.2.1.
    - No separation of image and sampler.
- No online compilation.

Environment Variables

- **`SYCL_BE`**
  - **`SYCL_BE=PI_CUDA`**: Use PI Cuda backend plugin.
  - **`SYCL_BE=PI_OPENCL`**: Use PI OpenCL backend plugin.
- **`SYCL_PI_TRACE`**
  - **`SYCL_PI_TRACE=1`**: Enable tracing of PI plugins / device discovery.
  - **`SYCL_PI_TRACE=2`**: Enable tracing of PI calls.
  - **`SYCL_PI_TRACE=-1`**: Enable all levels of tracing.

```
---> piPlatformsGet(
        <unknown> : 0
        <unknown> : 0
        <unknown> : 0x7ffe460a6db4
) --->          pi_result : PI_SUCCESS
---> piPlatformsGet(
        <unknown> : 0
        <unknown> : 0
        <unknown> : 0x7ffe460a6db4
) --->          pi_result : PI_SUCCESS
---> piPlatformsGet(
        <unknown> : 0
        <unknown> : 0
        <unknown> : 0x7ffe460a6fec
) --->          pi_result : PI_SUCCESS
---> piPlatformsGet(
        <unknown> : 1
        <unknown> : 0x5620dc502b98
        <unknown> : 0
) --->          pi_result : PI_SUCCESS
---> piDevicesGet(
        pi_platform : 0x7fd76297b040
        <unknown> : 4
        <unknown> : 1
        <unknown> : 0x5620dc502ba0
        <unknown> : 0
) --->          pi_result : PI_SUCCESS
---> piContextCreate(
        <unknown> : 0
        <unknown> : 1
        <unknown> : 0x5620dc502ba0
        <unknown> : 0
        <unknown> : 0
        <unknown> : 0x5620dc502ba8
) --->          pi_result : PI_SUCCESS
---> piQueueCreate(
        <unknown> : 0x5620df9c4310
        <unknown> : 0x5620dc633c70
        <unknown> : 0
        <unknown> : 0x7ffe460a6fe8
) --->          pi_result : PI_SUCCESS
---> piQueueFinish(
        <unknown> : 0x5620dfb29190
```

Interoperability with CUDA

Interop in SYCL 1.2.1

- Many of the SYCL runtime classes encapsulate an associated OpenCL type.
- **`.get()`** member function retains the OpenCL object and returns it.
- *Uh-oh* – we're not using OpenCL anymore!
- How do we expose backend-specific native handles in 1.2.1?

SYCL Generalization Proposal

- Proposal seeks to decouple SYCL from OpenCL.
  - Query the backend at runtime.
  - New **`get_native`** function returns correct native type for a given `backend` enumerator.
  - New **`make`** function creates SYCL objects from native objects.
    - Create a **`sycl::context`** from a **`CUcontext`** or **`cl_context`**.

https://github.com/KhronosGroup/SYCL-Shared/blob/master/proposals/sycl_generaliza

Using `get_native`

```
using namespace cl::sycl;

CUcontext context = get_native<backend::cuda>(syclContext);
CUstream  stream  = get_native<backend::cuda>(syclQueue);
CUdevice  device  = get_native<backend::cuda>(syclDevice);
CUevent   event   = get_native<backend::cuda>(syclEvent);
```

```
enum class backend { opencl, cuda, host };
```

SYCL RT Interop

- Only some features of the proposal are implemented.
  - **`get_native`** is implemented for most CUDA types. It is only implemented for a few OpenCL types.
  - No **`make`** implementation, so interop is strictly from SYCL to CUDA. You cannot create SYCL resources from CUDA resources.

Using Native Libraries in SYCL

- A wide ecosystem of CUDA libraries already exists.
- We want to tap into this ecosystem with SYCL.
- This is not possible in SYCL 1.2.1.
- We needed to find a solution to enable interop between SYCL RT and native CUDA libraries.

Codeplay Interop Task Proposal

- We can borrow a Codeplay proposal.
- New features in **`sycl::handler`** that allow third-party APIs to be called.
- Interop task commands are executed using the same SYCL 1.2.1. dependency tracking mechanisms.
- Native API calls are scheduled for you!

https://github.com/codeplaysoftware/standards-proposals/blob/master/interop_task/ir

Using `interop_task`

```
q.submit([&](handler &h) {
  auto d_A = b_A.get_access<sycl::access::mode::read>(h);
  auto d_B = b_B.get_access<sycl::access::mode::read>(h);
  auto d_C = b_C.get_access<sycl::access::mode::write>(h);

  h.interop_task([=](sycl::interop_handler ih) {
    cublasSetStream(handle, ih.get_queue<backend::cuda>());

    auto cuA = reinterpret_cast<float *>(ih.get_mem<backend::cuda>(d_A));
    auto cuB = reinterpret_cast<float *>(ih.get_mem<backend::cuda>(d_B));
    auto cuC = reinterpret_cast<float *>(ih.get_mem<backend::cuda>(d_C));

    CHECK_ERROR(cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, WIDTH, HEIGHT,
                            WIDTH, &ALPHA, cuA, WIDTH, cuB, WIDTH, &BETA,
                            cuC, WIDTH));
  });
});
```

Get **`CUstream`** from **`interop_handler`**.

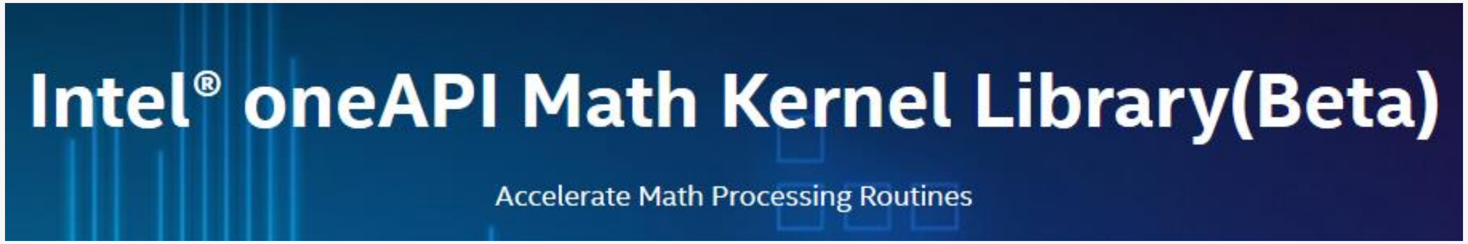
Get **`CUdeviceptr`** from **`interop_handler`**.

Call native CUDA API!

https://github.com/codeplaysoftware/SYCL-For-CUDA-Examples/blob/master/example-02/

Using oneMKL with SYCL on Nvidia GPUs

OneMKL Interface

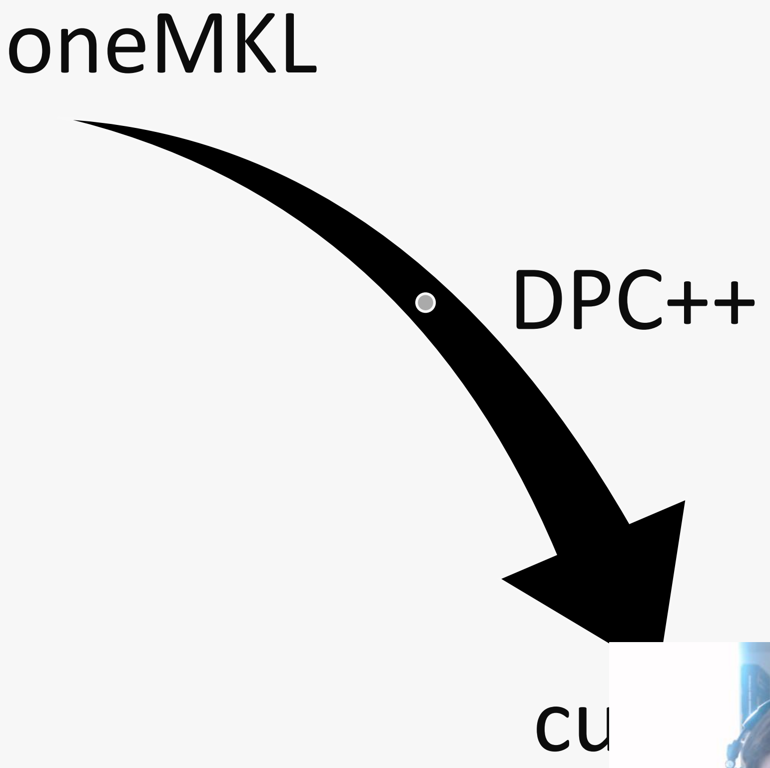


- OneMKL Interface is an open source Math Kernel Library
- Developers can use it to target Intel CPUs and GPUs; and now Nvidia GPUs
- To achieve the best performance for Nvidia GPUs, this library calls native cuBLAS functions

OneMKL Interface on NVIDIA

Achieving Performance

- oneMKL uses the cuBLAS interface directly
- CUDA memory and contexts can be accessed directly from SYCL.
- cuBLAS handle can be associated with the specified SYCL context and underlying CUDA context, directly calling the cuBLAS routine.
- DPC++ runtime manages the kernel scheduling when there are data dependencies among multiple cuBLAS routines.



OneMKL `get_native`

```
CublasScopedContextHandler::CublasScopedContextHandler(cl::sycl::queue queue) {
    placedContext_  = queue.get_context();
    auto device     = queue.get_device();
    auto desired    = cl::sycl::get_native<cl::sycl::backend::cuda>(placedContext_);
    auto cudaDevice = cl::sycl::get_native<cl::sycl::backend::cuda>(device);
    CUresult err;
    CUDA_ERROR_FUNC(cuCtxGetCurrent, err, &original_);
    CUcontext primary;
    cuDevicePrimaryCtxRetain(&primary, cudaDevice);
    bool isPrimary = primary == desired;
    cuDevicePrimaryCtxRelease(cudaDevice);
    if (original_ != desired) {
        // Sets the desired context as the active one for the thread
        CUDA_ERROR_FUNC(cuCtxSetCurrent, err, desired);
        // No context is installed and the suggested context is primary
        // This is the most common case. We can activate the context in the
        // thread and leave it there until all the PI context referring to the
        // same underlying CUDA primary context are destroyed. This emulates
        // the behaviour of the CUDA runtime api, and avoids costly context
        // switches. No action is required on this side of the if.
        needToRecover_ = !(original_ == nullptr && isPrimary);
    }
}
```

https://github.com/oneapi-src/oneMKL/blob/master/src/blas/backends/cublas/cublas_

OneMKL `interop_task`

```
queue.submit([&](cl::sycl::handler &cgh) {
    auto a_acc = a.template get_access<cl::sycl::access::mode::read_write>(cgh);
    auto x_acc = x.template get_access<cl::sycl::access::mode::read>(cgh);
    auto y_acc = y.template get_access<cl::sycl::access::mode::read>(cgh);
    cgh.interop_task([=](cl::sycl::interop_handler ih) {
        auto sc     = CublasScopedContextHandler(queue);
        auto handle = sc.get_handle(queue);
        auto a_     = sc.get_mem<cuDataType *>(ih, a_acc);
        auto x_     = sc.get_mem<cuDataType *>(ih, x_acc);
        auto y_     = sc.get_mem<cuDataType *>(ih, y_acc);
        cublasStatus_t err;
        CUBLAS_ERROR_FUNC(func, err, handle, m, n, (cuDataType *)&alpha, x_, incx, y_, incy, a_,
                          lda);
    });
});
```

https://github.com/codeplaysoftware/standards-proposals/blob/master/interop_task/i

Performance Results

We have run the BabelStream Benchmarks and compared results from:

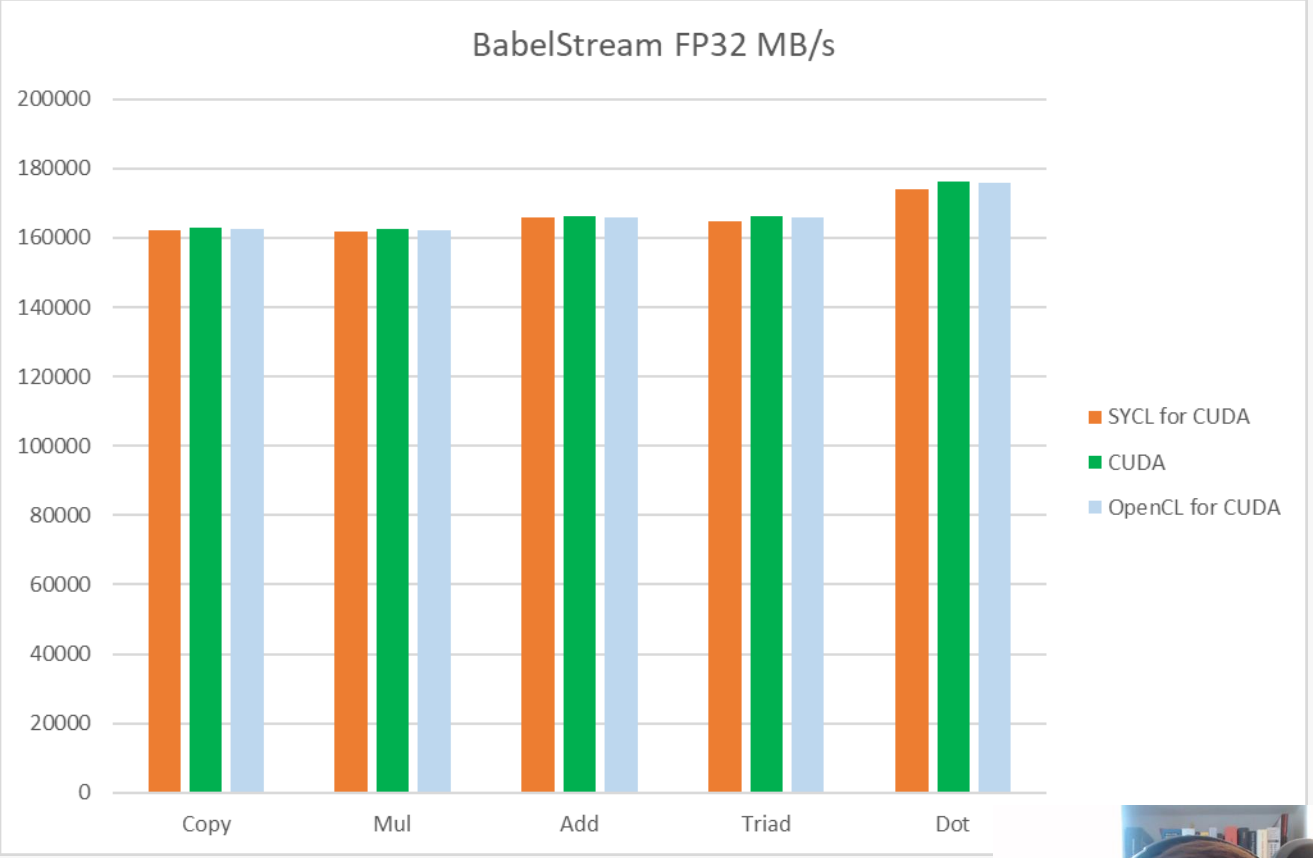
- Native CUDA code
- OpenCL code
- SYCL code using the Nvidia implementation

About BabelStream

"Measure memory transfer rates to/from global device memory on GPUs"

All run on: CUDA 10.1 on GeForce GTX 980

Benchmarks website: http://uob-hpc.github.io/BabelStream



Conclusions & Future Work

Conclusion

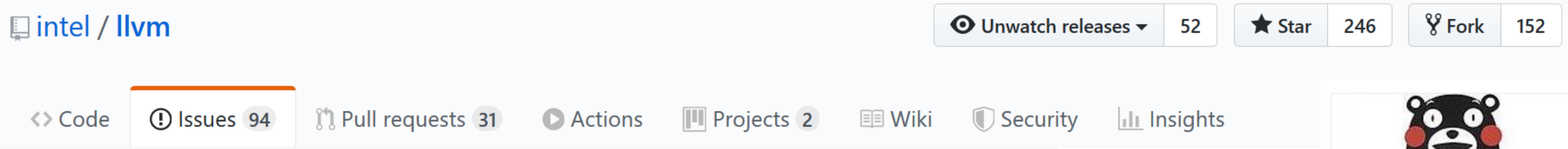
- DPC++ can build SYCL applications that are also CUDA applications
- Using DPC++
  - It's possible for developers to write standard C++ SYCL code and run on Nvidia GPUs
- It's also possible to use the cuBLAS native library via oneMKL
  - Performance is achieved by integrating with native CUDA interfaces
  - It's possible to try it out today using the open source DPC++ LLVM project
  - The only code change required is to change your device selector

Future plans

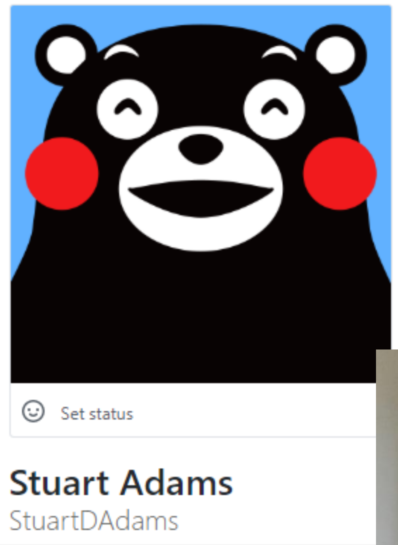
- Our current focus is on conformance with the SYCL compatibility test suite
- We are working on further performance enhancements
- Additional SYCL extensions will be implemented to expand the features available

Participate!

- Join us in the intel/llvm repository



- Report issues and feature requests
- Review or contribute Pull requests



Enabling AI to be Open, Safe & Accessible to All

Thank You!

@codeplaysoft

info@codeplay.com

codeplay.com

Trademark disclaimer

Nvidia and CUDA are trademarks of Nvidia Corporation.

Intel is a trademark of Intel Corporation.

SYCL is a trademark of the Khronos Group Inc.

OpenCL and the OpenCL logo are trademarks of Apple Inc. used by permission by Khronos.

Codeplay and ComputeCpp are trademarks of Codeplay Software Limited.

Other names and brands may be claimed as the property of others.

SYCL for CUDA Hands On

https://github.com/codeplaysoftware/SYCL-For-CUDA-Examples/

Please head over to the *SYCL For CUDA Examples* repo!

Examples

- SYCL application running on CUDA.
- SYCL interop with CUDA Driver API.
- SYCL interop with CUDA Runtime API.

Exercise

- Write SYCL interop with cuBLAS.