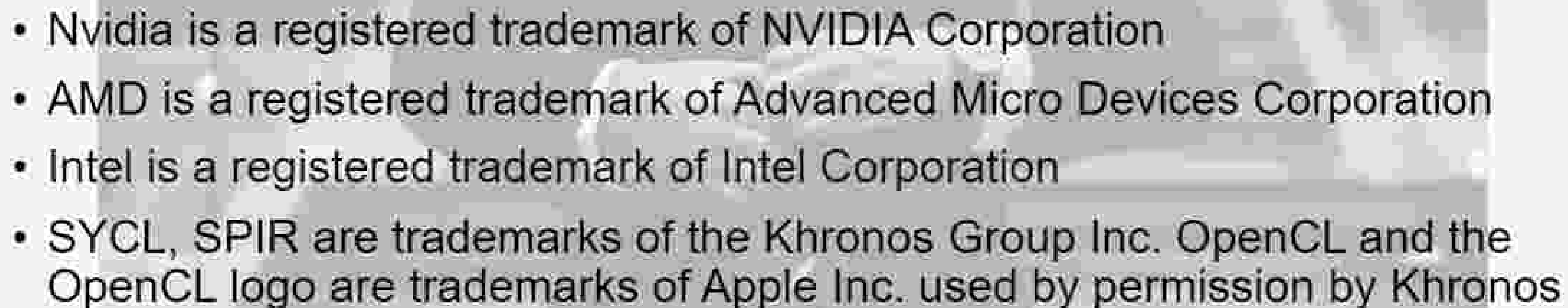
Developing for Nvidia GPUs using SYCL with oneAPI

Joe Todd – Senior Software Engineer

oneAPI Developer Summit at SC – 14/11/2021

Copyright Codeplay Software Ltd 2021



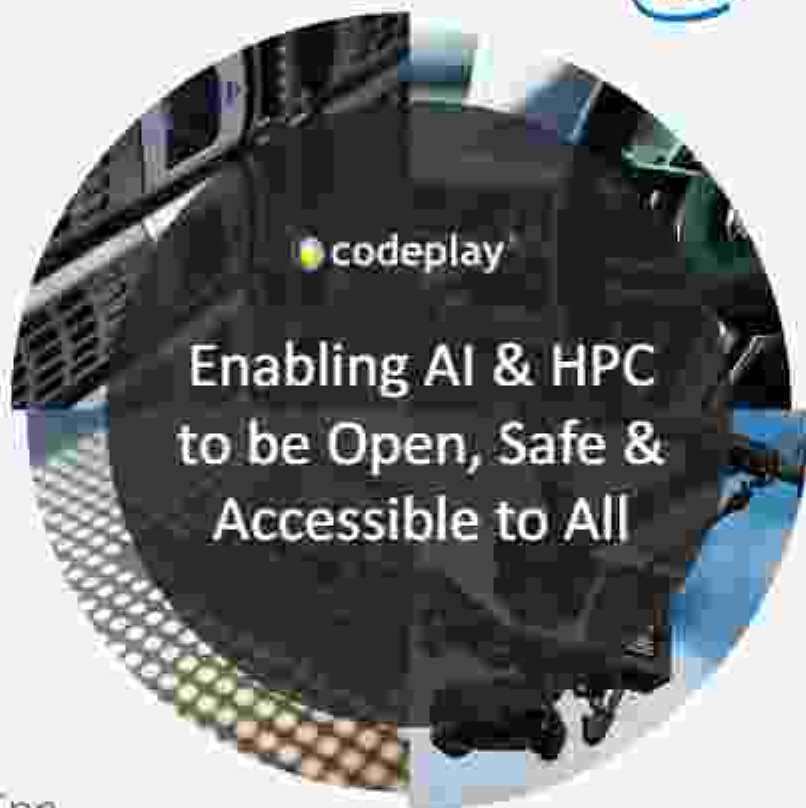
Copyright Codeplay Software Ltd 2021

Company

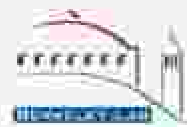
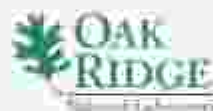
Leaders in enabling high-performance software solutions for new AI processing systems

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

Established 2002 in Scotland with ~80 employees



Partners

And many more!

Products

Acoran

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

ComputeAorta

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

ComputeCpp

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

Markets

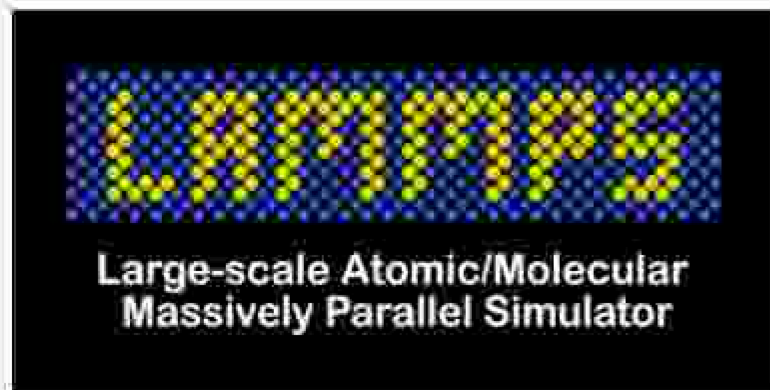
High Performance Compute (HPC)
Automotive ADAS, IoT, Cloud Compute
Smartphones & Tablets
Medical & Industrial

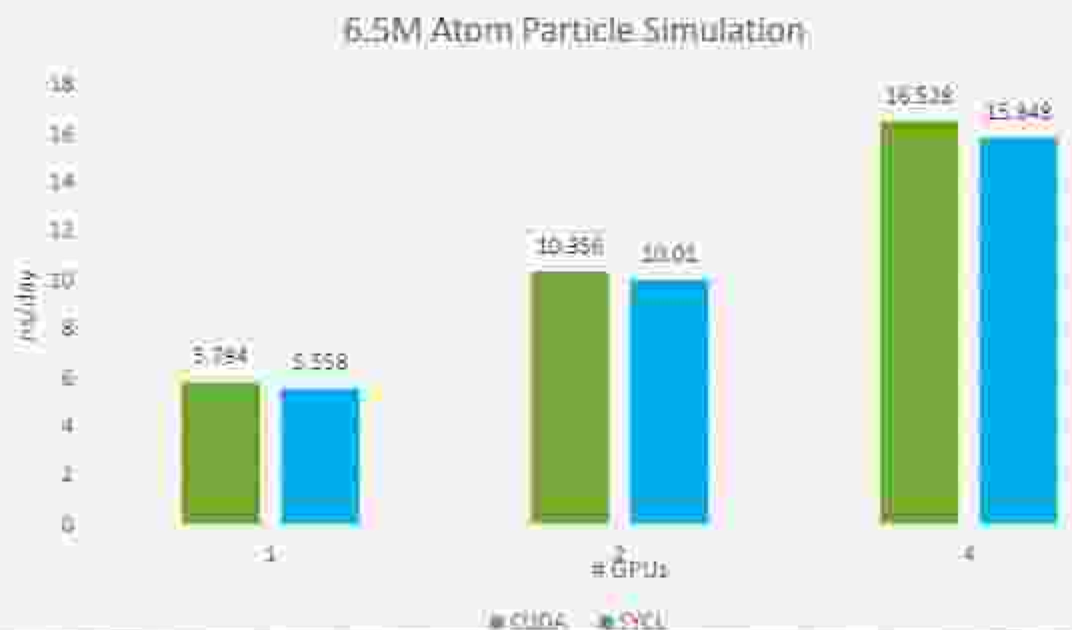
Technologies: Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

Copyright Codeplay Software Ltd 2021

My Recent Work: LAMMPS

- Molecular/Particle Simulator
- Distributed heterogenous computing
- Goal: Match CUDA performance





Copyright Codeplay Software Ltd 2021

Overview

- Benefits of DPC++ and SYCL
- Porting your brain
- Porting your code
- "Hands on" example

Copyright Codeplay Software Ltd 2021

Learning Objectives

1

Understand why you should use SYCL

2

Learn how to port *existing* CUDA code to SYCL

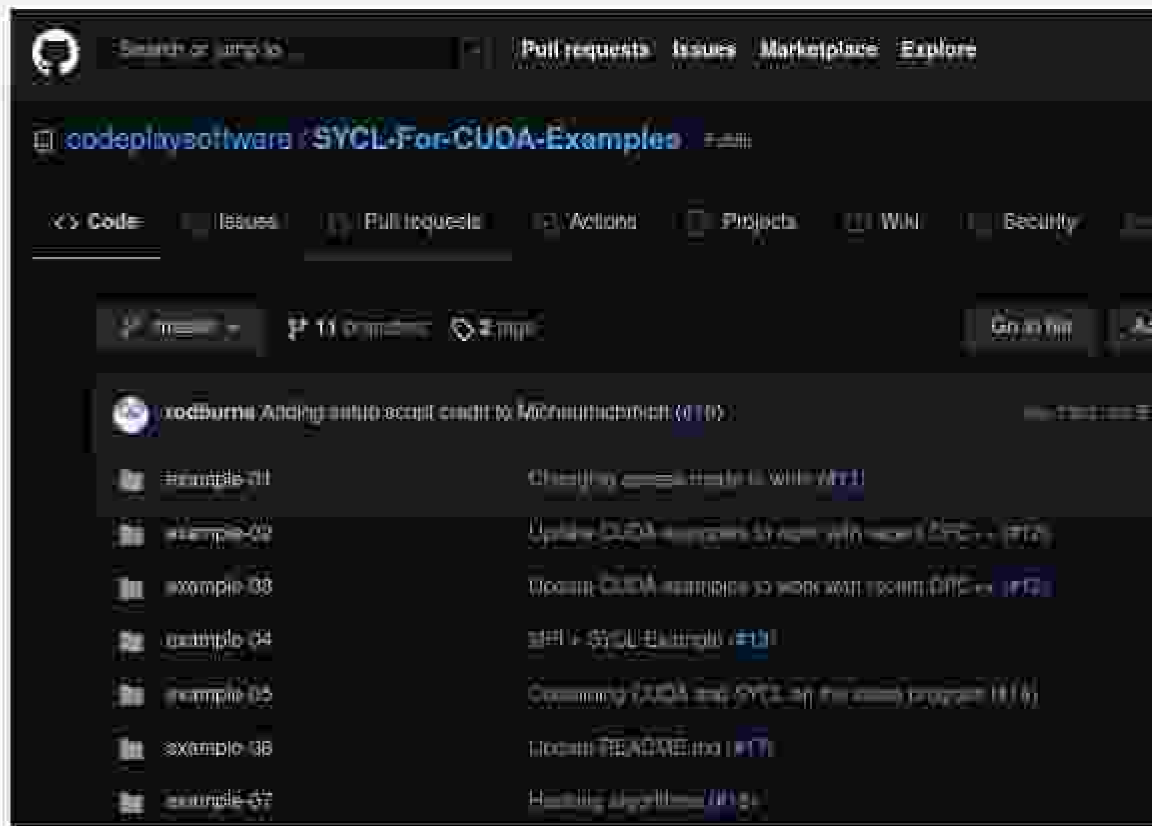
3

Learn how to write new SYCL code, targeting Nvidia hardware

Copyright Codeplay Software Ltd 2021

Practical Material

- Codeplay has a repo of examples with a docker image
- Some include both .cu & .cpp
- All target the SYCL CUDA backend
- I'll signpost these in the slides, and encourage you to play around with them

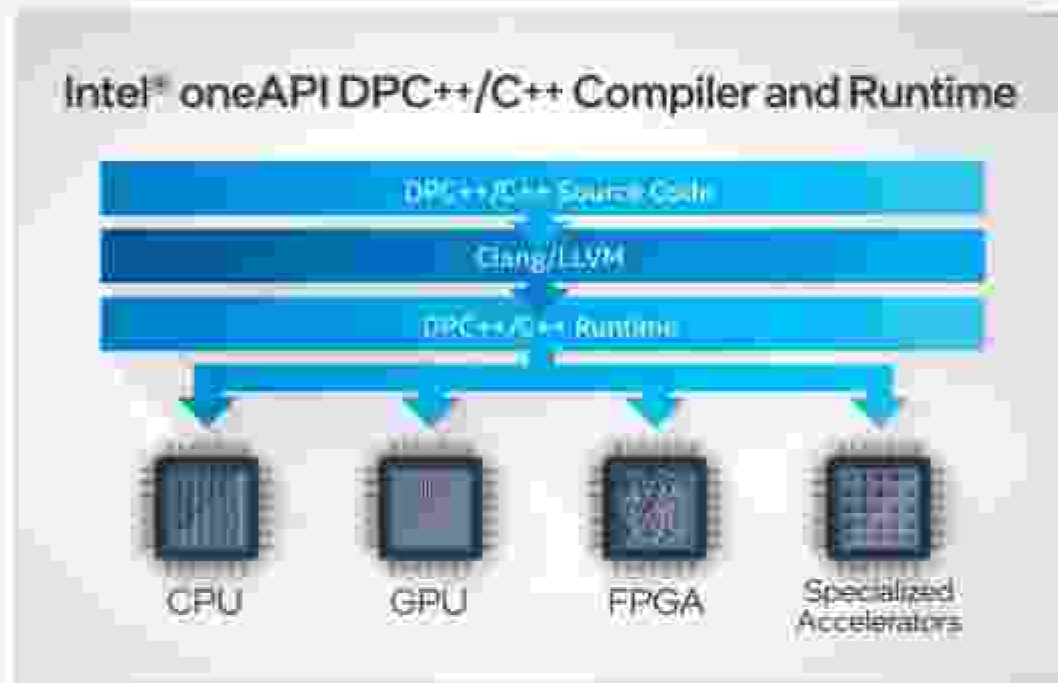


https://github.com/codeplaysoftware/SYCL-For-CUDA-Examples

Copyright Codeplay Software Ltd 2021

What is DPC++

- Data Parallel C++ is an open alternative to single-architecture proprietary languages
- DPC++ is an open source implementation of SYCL with some extensions
- It is part of the oneAPI framework that includes definitions of standard library interfaces, for common operations such as math



Copyright Codeplay Software Ltd 2021

SYCL and DPC++ Enables Supercomputers

"this work supports the productivity of scientific application developers and users through performance portability of applications between Aurora and Perlmutter."

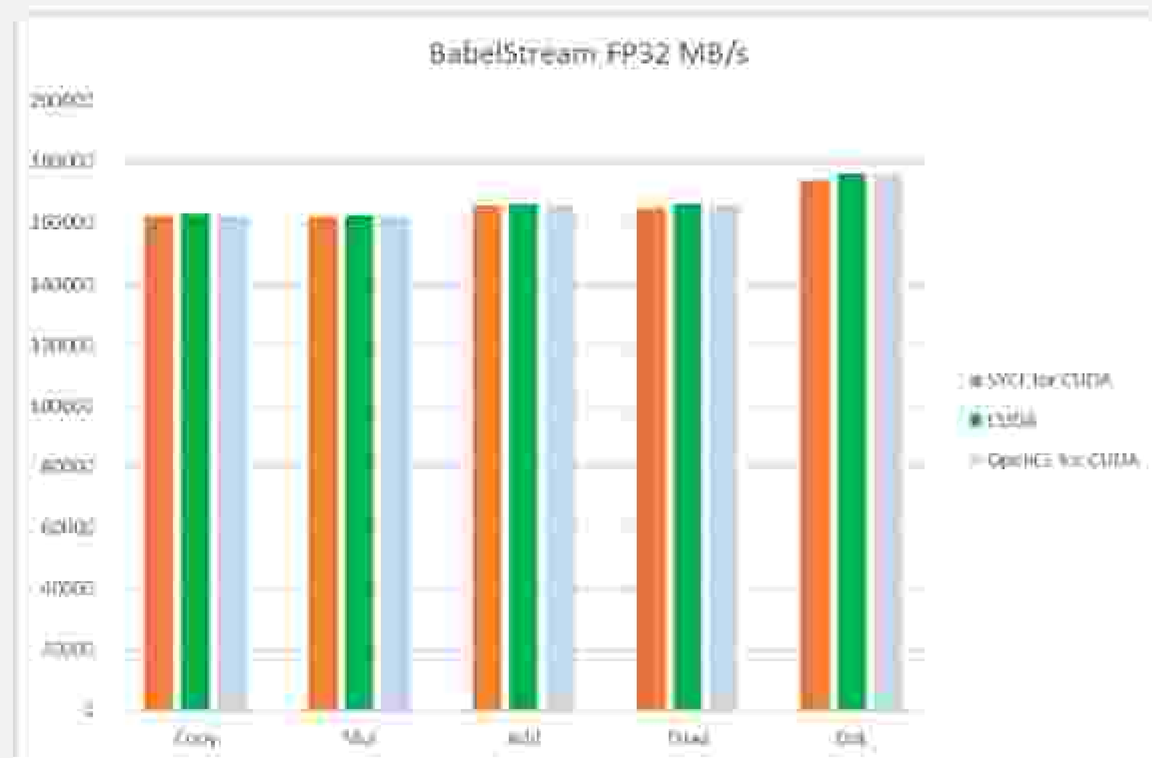
Codeplay works in partnership with US National Laboratories to enable SYCL on exascale supercomputers

Enables a broad range of software frameworks and applications

© 2021 Codeplay Software Ltd.

DPC++ performance on Nvidia GPUs

- This graph compares the BabelStream benchmarks results for:
  - Native CUDA code
  - OpenCL code
  - SYCL code using the CUDA backend
- Run on GeForce GTX 980 with CUDA 10.1
- Minimal SYCL overhead



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

© 2021 Codeplay Software Ltd.

Why use SYCL?

- Code Portability
  - Target GPUs from Nvidia, Intel, AMD
  - Run on CPU, FPGAs, exotic hardware
- Performance Portability
- "But I already know how to use CUDA!"

Copyright Codeplay Software Ltd 2021

Moving to SYCL is easy!

- Most CUDA concepts map 1:1 with SYCL concepts
- SYCL 2020 spec describes this in detail
- The same performance optimizations apply
- The same profiling tools (nvprof, nsys, ncu) still work

Copyright Codeplay Software Ltd 2021

Why use SYCL?

- Code Portability
  - Target GPUs from Nvidia, Intel, AMD
  - Run on CPU, FPGAs, exotic hardware
- Performance Portability
- ~~"But I already know how to use CUDA!"~~
- You basically already know how to use SYCL!

Copyright Codeplay Software Ltd 2021

Porting your brain

Copyright Codeplay Software Ltd 2021

Porting your brain to SYCL

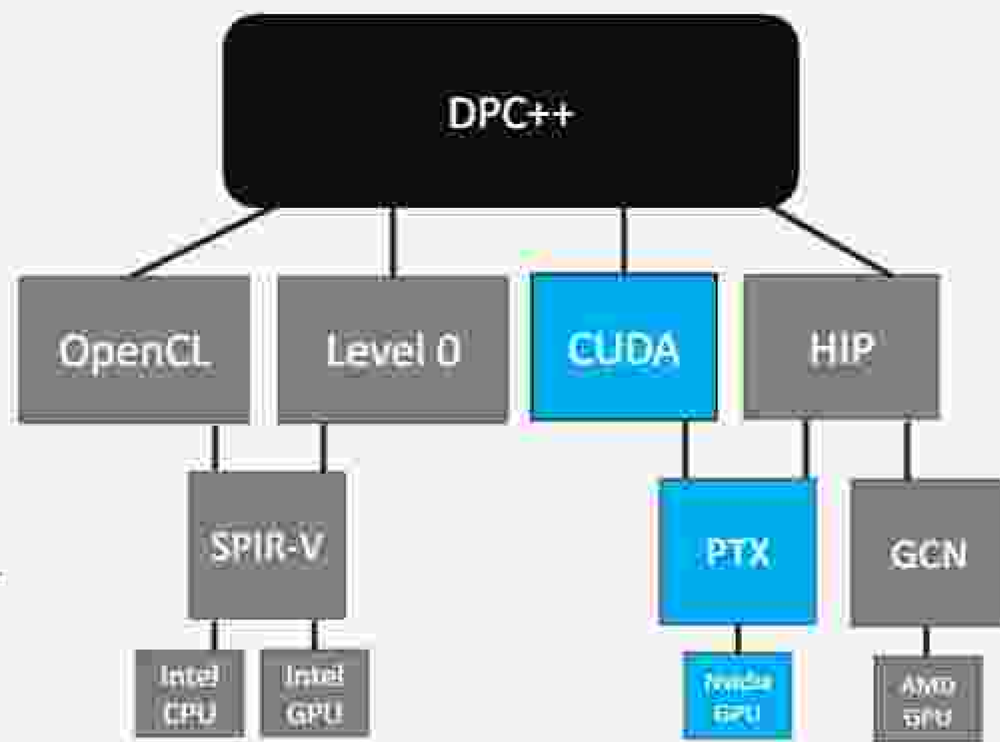
- SYCL has strong GPU heritage (OpenCL)
- Most CUDA concepts map to a SYCL equivalent
- SYCL is an abstraction layer on top of CUDA & others

© 2021 Codeplay Software Ltd.

example-02

CUDA as a backend

- The DPC++ runtime currently implements 4 SYCL backends:
  - OpenCL
  - Level Zero
  - CUDA
  - HIP
- CUDA backend uses CUDA runtime API
- Moving up a level of abstraction, you untether your codebase from specific hardware
- Devs have flexibility to prioritize performance or portability



© 2021 Codeplay Software Ltd.

CUDA as a backend

Check out the draft CUDA backend spec in the *SYCL 2020 spec*

Appendix D: CUDA backend specification

This chapter describes how the SYCL general programming model is mapped on top of CUDA, and how the SYCL generic interoperability interface must be implemented by vendors providing SYCL for CUDA implementations to ensure SYCL applications written for the CUDA backend are interoperable.

Currently in PR #197:

https://github.com/KhronosGroup/SYCL-Docs/pull/197

Viewable as a PDF via Github Actions artefact

© 2021 Codeplay Software Ltd.

Concepts: How SYCL maps to CUDA

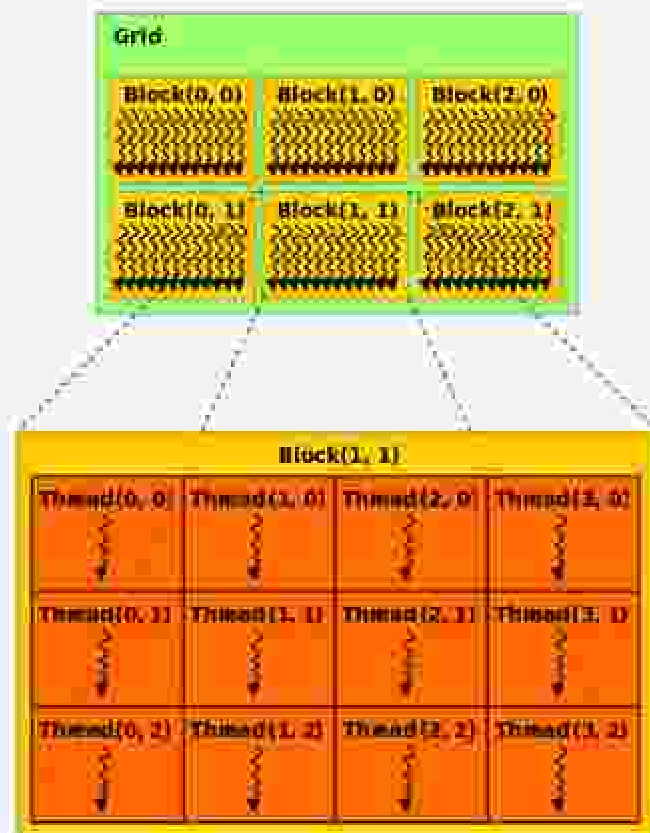
Copyright Codeplay Software Ltd 2021

CUDA	SYCL
CUstream	sycl::queue
CUcontext	sycl::context
CUdevice	sycl::device
CUevent	sycl::event

Copyright Codeplay Software Ltd 2021

It isn't all the same

- SYCL offers out-of-order queues
- Buffers!
- Strong focus on task-graphs
- Specify *total* problem size, not grid & block
  - Optionally specify work-group (block) size



© 2021 Codeplay Software Ltd.

Specifying Problem Size

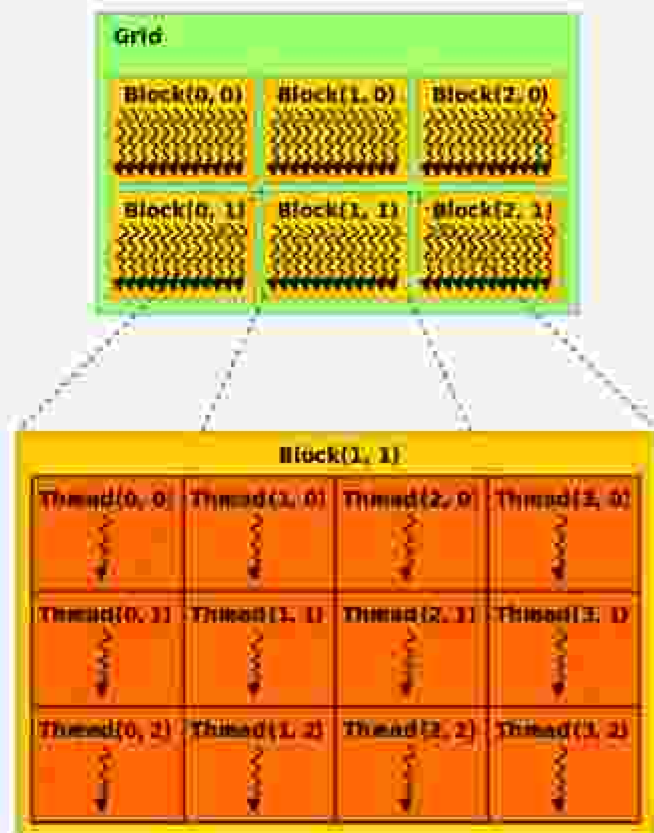
- CUDA kernels' execution range defined by passing number of blocks per grid, and threads per block:

```
CUDAKernel<<<BlocksPerGrid, ThreadsPerBlock>>>( … );
```

- Using sycl::nd_range to define execution range, we specify the global range, and local range

```
sycl::nd_range<1>({global}, {local})
```

- Using sycl::range, just specify the global range!



© 2021 Codeplay Software Ltd.

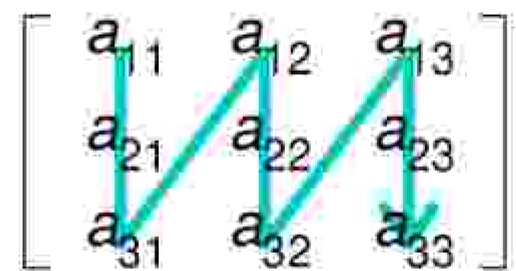
Specifying Problem Size

- A SYCL local range = CUDA threads per block
- A SYCL global range = CUDA blocks per grid * threads per block
- You can leave it to the SYCL runtime to choose your local range
- But probably get better performance if you pick it

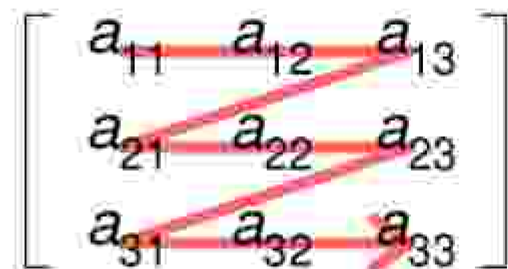
© 2021 Codeplay Software Ltd.

It isn't all the same

- SYCL offers out-of-order queues
- Buffers
- Strong focus on task-graphs
- Specify *total* problem size, not grid & block
  - Optionally specify work-group (block) size
- Index ordering is flipped...



CUDA, OpenCL, FORTRAN



SYCL, C++

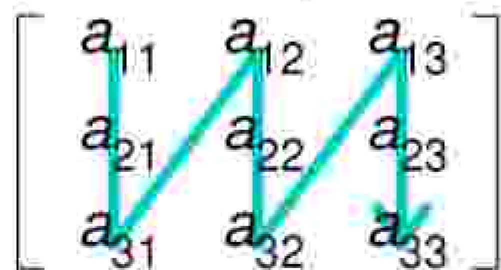
© 2021 Codeplay Software Ltd.

Index Ordering

CUDA:

Threads {**0**, 9, 10} and {**1**, 9, 10} are warp-neighbours.

Max range in **X-dimension** is much larger than Y-, Z-.

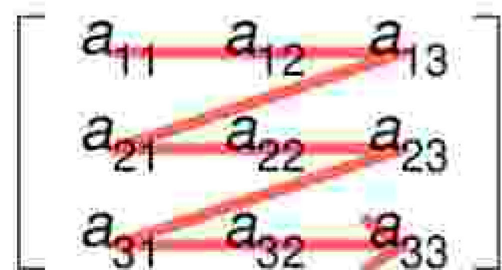


CUDA, OpenCL, FORTRAN

SYCL:

Threads {10, 9, **0**} and {10, 9, **1**} are warp-neighbours.

Max range in **Z-dimension** is much larger than X-, Y-.



SYCL, C++

© 2021 Codeplay Software Ltd.

CUDA	SYCL
[illegible]	[illegible]
CUcontext	sycl::context
CUdevice	sycl::device
CUevent	sycl::event
threadIdx.{x,y,z}	nd_item.get_local_id({0,1,2})

```
__device__
int getGlobalIdx_3D_3D(){
      int blockId = blockIdx.x + blockIdx.y * gridDim.x
                  + gridDim.x * gridDim.y * blockIdx.z;
      int threadId = blockId * (blockDim.x * blockDim.y * blockDim.z)
                   + (threadIdx.z * (blockDim.x * blockDim.y))
                   + (threadIdx.y * blockDim.x) + threadIdx.x;
      return threadId;
}
```


Copyright Codeplay Software Ltd 2021

Porting your code

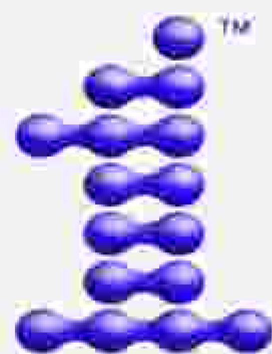
- DPC++ Compatibility Tool
  - Can even handle large projects: 'intercept-build make'
  - Doesn't handle everything
- Use Unified Shared Memory (USM)
- Incremental porting via interop
  - Launch existing CUDA kernels
  - Call existing CUDA libs (cuBLAS, cuDNN)

© 2021 Codeplay Software Ltd.

Using DPC++ with CUDA and Nvidia

Incremental porting

- Port CUDA applications to SYCL one kernel at a time



oneAPI

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

© 2021 Codeplay Software Ltd.

Targeting Nvidia hardware

- Compile DPC++ with CUDA support:

```
git clone https://github.com/intel/llvm.git -b sycl
cd llvm
python ./buildbot/configure.py --cuda -t release --cmake-gen "Unix Makefiles"
cd build
make install -j `nproc`
```

- Specify NVPTX target:

```
clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda vec_add.cpp -o vec_add
```

- Select a 'CUDA' device...

Copyright Codeplay Software Ltd 2021

example-01

sycl::device_selector

- Define a child class of sycl::device_selector
- Check for sycl::backend::cuda

```
class CUDASelector : public sycl::device_selector {
public:
  int operator()(const sycl::device &device) const override {

    if (device.get_platform().get_backend() == sycl::backend::cuda)
      return 1;
    else
      return -1;
  }
};
```

- Or just use sycl::gpu_selector() if you're feeling lazy...

Copyright Codeplay Software Ltd 2021

Kernel Launch Mechanism

- Example of launching a 1D kernel in CUDA, and an equivalent invocation in SYCL

```
__global__ void CUDAKernel( ... ) {
  // Kernel implementation
}
```

```
int N = 10000;

dim3 ThreadsPerBlock(256);
dim3 BlocksPerGrid;

BlocksPerGrid.x =
 ceil(double(N) / double(ThreadsPerBlock.x));

CUDAKernel<<<BlockPerGrid, ThreadsPerBlock>>>(...);
```

```
using namespace sycl;
int N = 10000;
queue myQueue{CUDASelector()};

myQueue.submit([&](handler &h) {

 auto localRange = range<1>(256);
 auto globalRange =
   localRange * range<1>(ceil(double(N) / double(localRange.get(0)));

 auto NDRange = nd_range<1>(globalRange, localRange);

 h.parallel_for<class Kernel>(NDRange, [=](nd_item<1> ndItem) {
    // Kernel implementation
 });
}
```


Copyright Codeplay Software Ltd 2021

SYCL Unified Shared Memory (USM)

- USM memory allocations return pointers which are consistent between the host application and kernel function on a device
- Pointer-based API makes porting from CUDA code easier
- SYCL was historically based on buffers & accessors, these still exist & have some useful features

© 2021 Codeplay Software Ltd.

Porting CUDA host code to SYCL – USM

- In order to use USM in SYCL, we need to explicitly allocate and free memory
- `cudaMallocManaged` calls are replaced by `sycl::malloc_shared`
- `cudaFree` calls are replaced by `sycl::free`
- We have to make sure to explicitly wait for kernel completion before accessing the shared memory region, e.g., by calling `queue.wait()`

© 2021 Codeplay Software Ltd.

Handling memory - USM

```
int N = 10000;

float *x_vec = nullptr;
cudaMallocManaged((void **)&x_vec, N * sizeof(float));

for (int i = 0; i < N; ++i)
   x_vec[i] = N;

// Call CUDA kernel here (modifying x_vec)

cudaDeviceSynchronize();

// Printing results of kernel
for (int i = 0; i < N; ++i)
   std::cout << x_vec[i] << "\n";

cudaFree(x_vec);
```

```
int N = 10000;
sycl::queue myQueue{USMDeviceSelector};

float *x_vec = sycl::malloc_shared(N * sizeof(float), myQueue));

for (int i = 0; i < N; ++i)
   x_vec[i] = N;

// Submit SYCL kernel here (modifying x_vec)

myQueue.wait();

// Printing results of kernel
for (int i = 0; i < N; ++i)
   std::cout << x_vec[i] << "\n";

sycl::free(x_vec, myQueue);
```


© 2021 Codeplay Software Ltd.

Handling memory - USM

```
int N = 10000;

float *x_vec = nullptr;
cudaMallocManaged((void **)&x_vec, N * sizeof(float));

for (int i = 0; i < N; ++i)
  x_vec[i] = N;

// Call CUDA kernel here (modifying x_vec)

cudaDeviceSynchronize();

// Printing results of kernel
for (int i = 0; i < N; ++i)
  std::cout << x_vec[i] << "\n";

cudaFree(x_vec);
```

```
int N = 10000;
sycl::queue myQueue{USMDeviceSelector};

float *x_vec = sycl::malloc_shared(N * sizeof(float), myQueue));

for (int i = 0; i < N; ++i)
  x_vec[i] = N;

// Submit SYCL kernel here (modifying x_vec)

myQueue.wait();

// Printing results of kernel
for (int i = 0; i < N; ++i)
  std::cout << x_vec[i] << "\n";

sycl::free(x_vec, myQueue);
```


© 2021 Codeplay Software Ltd.

Handling memory - USM

```
int N = 10000;

float *x_vec = nullptr;
cudaMallocManaged((void **)&x_vec, N * sizeof(float));

for (int i = 0; i < N; ++i)
   x_vec[i] = N;

// Call CUDA kernel here (modifying x_vec)

cudaDeviceSynchronize();

// Printing results of kernel
for (int i = 0; i < N; ++i)
   std::cout << x_vec[i] << "\n";

cudaFree(x_vec);
```

```
int N = 10000;
sycl::queue myQueue{USMDeviceSelector};

float *x_vec = sycl::malloc_shared(N * sizeof(float), myQueue));

for (int i = 0; i < N; ++i)
   x_vec[i] = N;

// Submit SYCL kernel here (modifying x_vec)

myQueue.wait();

// Printing results of kernel
for (int i = 0; i < N; ++i)
   std::cout << x_vec[i] << "\n";

sycl::free(x_vec, myQueue);
```


© 2021 Codeplay Software Ltd.

Handling memory - USM

```
int N = 10000;

float *x_vec = nullptr;
cudaMallocManaged((void **)&x_vec, N * sizeof(float));

for (int i = 0; i < N; ++i)
  x_vec[i] = N;

// Call CUDA kernel here (modifying x_vec)

cudaDeviceSynchronize();

// Printing results of kernel
for (int i = 0; i < N; ++i)
  std::cout << x_vec[i] << "\n";

cudaFree(x_vec);
```

```
int N = 10000;
sycl::queue myQueue{USMDeviceSelector};

float *x_vec = sycl::malloc_shared(N * sizeof(float), myQueue));

for (int i = 0; i < N; ++i)
  x_vec[i] = N;

// Submit SYCL kernel here (modifying x_vec)

myQueue.wait();

// Printing results of kernel
for (int i = 0; i < N; ++i)
  std::cout << x_vec[i] << "\n";

sycl::free(x_vec, myQueue);
```

codeplay

© 2021 Codeplay Software Ltd.

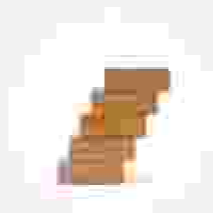
CUDA	SYCL
CUstream	sycl::queue
CUcontext	sycl::context
CUdevice	sycl::device
CUevent	sycl::event
threadIdx.{x,y,z}	nd_item.get_local_id({0,1,2})
blockDim.{x,y,z}	nd_item.get_local_range({0,1,2})
blockIdx.{x,y,z}	nd_item.get_group({0,1,2})
threadIdx.{x,y,z} + blockIdx.{x,y,z} * blockDim.{x,y,z}	nd_item.get_global_id({0,1,2})
(A very long one)	nd_item.get_global_linear_id()
cudaMallocManaged	sycl::malloc_shared (USM shared)
cudaMallocHost	sycl::malloc_host (USM host pinned)
cudaMalloc	sycl::malloc_device (USM device)
cudaFree()	sycl::free()
cudaDeviceSynchronize()	MyQueue.wait()

Copyright Codeplay Software Ltd 2021

USM vs Buffers

- USM maps nicely to CUDA's memory model
- SYCL buffers offer:
  - Scoped memory management
  - Blocks on destruction
  - Automatic task graph generation
- Greenfield project developers – consider buffers

© 2021 Codeplay Software Ltd.



Incremental porting

Migrate host code to SYCL and keep your CUDA kernels

Porting to SYCL/DPC++

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

```
// 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.host_task( = interop_handle ih
    auto dA = reinterpret_cast double* ih.get_native_mem<backend
    auto dB = reinterpret_cast double* ih.get_native_mem backend
    auto dC = reinterpret_cast double* ih.get_native_mem backend

    int blockSize, gridSize;
    // Number of threads in each thread block
    blockSize = 1024;
    // Number of thread blocks in grid
    gridSize = static_cast int ceil static_cast float n / bloc
    // Call the CUDA kernel directly from SYCL
    vecAdd<< gridSize, blockSize >> dA, dB, dC, n ;
  };
};
```


Copyright Codeplay Software Ltd 2021

example-03

SYCL Interop

- Host tasks enable calling CUDA kernels/libraries directly
- Facilitates incremental porting

`host_task`

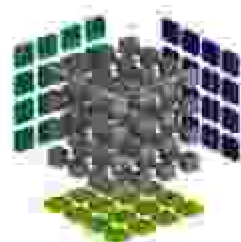
`interop_handle`

```
// 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.host_task( = interop_handle ih
    auto dA = reinterpret_cast double* ih.get_native_mem<backend
    auto dB = reinterpret_cast double* ih.get_native_mem backend
    auto dC = reinterpret_cast double* ih.get_native_mem backend

    int blockSize, gridSize;
    // Number of threads in each thread block
    blockSize = 1024;
    // Number of thread blocks in grid
    gridSize = static_cast int ceil static_cast float n / bloc
    // Call the CUDA kernel directly from SYCL
    vecAdd<< gridSize, blockSize >> dA, dB, dC, n ;
  );
);
```


Copyright Codeplay Software Ltd 2021

CUDA Libraries

CUDA Libraries Documentation



cuBLAS Library Documentation

The cuBLAS Library is an implementation of BLAS (Basic Linear Algebra Subprograms) on NVIDIA CUDA runtime. It enables the user to access the computational resources of NVIDIA GPUs.

Browse >

cuSPARSE Library Documentation

The cuSPARSE Library contains a set of basic linear algebra subroutines used for handling sparse matrices. It is implemented on NVIDIA CUDA runtime, and is designed to be called from C and C++.

Browse >

- Most CUDA developers use some of the libraries provided by Nvidia and others
- Examples include cuBLAS and cuDNN
- DPC++ support for CUDA includes interoperability with these libraries
- oneAPI equivalents exist (oneMKL-BLAS, oneDNN) for when you're ready to migrate

© 2021 Codeplay Software Ltd.

example-02

Library Interop

```
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.host_task([=](sycl::interop_handle ih) {
    cuCtxSetCurrent(ih.get_native_context<backend::cuda>());
    cublasSetStream(handle, ih.get_native_queue<backend::cuda>());
    auto cuA = reinterpret_cast<float *>(ih.get_native_mem<backend::cuda>(d_A));
    auto cuB = reinterpret_cast<float *>(ih.get_native_mem<backend::cuda>(d_B));
    auto cuC = reinterpret_cast<float *>(ih.get_native_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));
  });
});
```


Copyright Codeplay Software Ltd 2021

example-02

SYCL + cuBLAS SCAL example

```
int main(int argc, char *argv[]) {
    sycl::queue myQueue{CUDADeviceSelector()};

    constexpr int N = 1024;
    std::vector<float> h_A(N, 2.f);

    cublasHandle_t cublasHandle;
    cublasCreate(&cublasHandle);

    {
        sycl::buffer<float> b_A(h_A.data(), sycl::range<1>{N});

        myQueue.submit([&](sycl::handler &cgh) {
            auto d_A = b_A.get_access<sycl::access::mode::read_write>(cgh);

            cgh.host_task([=](sycl::interop_handle ih) {

                auto cudaStreamHandle = sycl::get_native<sycl::backend::cuda>(myQueue);
                cublasSetStream(cublasHandle, cudaStreamHandle);

                auto cuA = reinterpret_cast<float
*>(ih.get_native_mem<sycl::backend::cuda>(d_A));

                constexpr float ALPHA = 2.f;
                constexpr int INCX = 1;
                cublasSscal(cublasHandle, N, &ALPHA, cuA, INCX);
            });
        });
    }
    cublasDestroy(cublasHandle);
    return 0;
}
```

1. Create SYCL queue using CUDA backend
2. Create and fill host vector
3. Declare and create a cuBLAS handle
4. Create new scope for buffer creation and queue submission
5. Create SYCL buffer from host data
6. Create a command group submission to queue
7. Inside command group, create a SYCL accessor
8. Invoke the interop_task, passing an interop_handler
9. Inside the interop_task, retrieve the native CUDA stream handle from our queue's CUDA backend
10. Use the handle to set the cuBLAS stream
11. Using the interop_handler, retrieve the native CUDA memory handle from the SYCL accessor and cast it to the appropriate pointer type
12. Call cuBLAS SCAL with desired parameters
13. After you ensure the command group has completed, remember to destroy the cuBLAS handle (here this is guranteed as the buffer destructor is called upon exiting its scope)

Copyright Codeplay Software ltd 2021

SYCL offers flexibility

- Free to keep as much/little CUDA code as you like
- Danger: you port from CUDA to "SYCL for CUDA only"
- Compromise: specialize your implementation using:
  - Preprocessor macros: "#ifdef __NVPTX__"
  - Templates

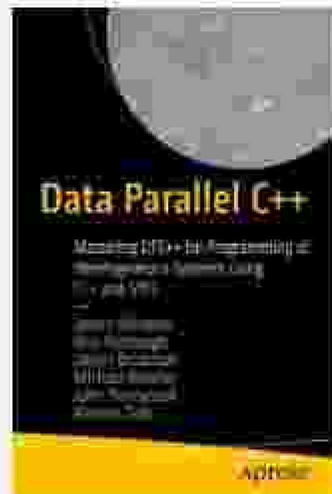
© 2021 Codeplay Software Ltd.

Porting Summary

- Migrating CUDA to SYCL is:
  - Easy
  - Incremental
  - Flexible
- Use existing perf tools
- USM makes porting easy
  - But buffers are really useful

Copyright Codeplay Software Ltd 2021

Learning Resources



- Data Parallel C++ (the SYCL book)
- Zheming Jin's HeCBench repo
  - https://github.com/zjin-lcf/HeCBench
  - 100s of ported codes, SYCL-CUDA-OpenMP Rosetta stone!
- SYCL Academy https://github.com/codeplaysoftware/syclacademy
- SYCL Spec
  - Surprisingly readable
  - PR #197 for CUDA backend spec

Appendix D: CUDA backend specification

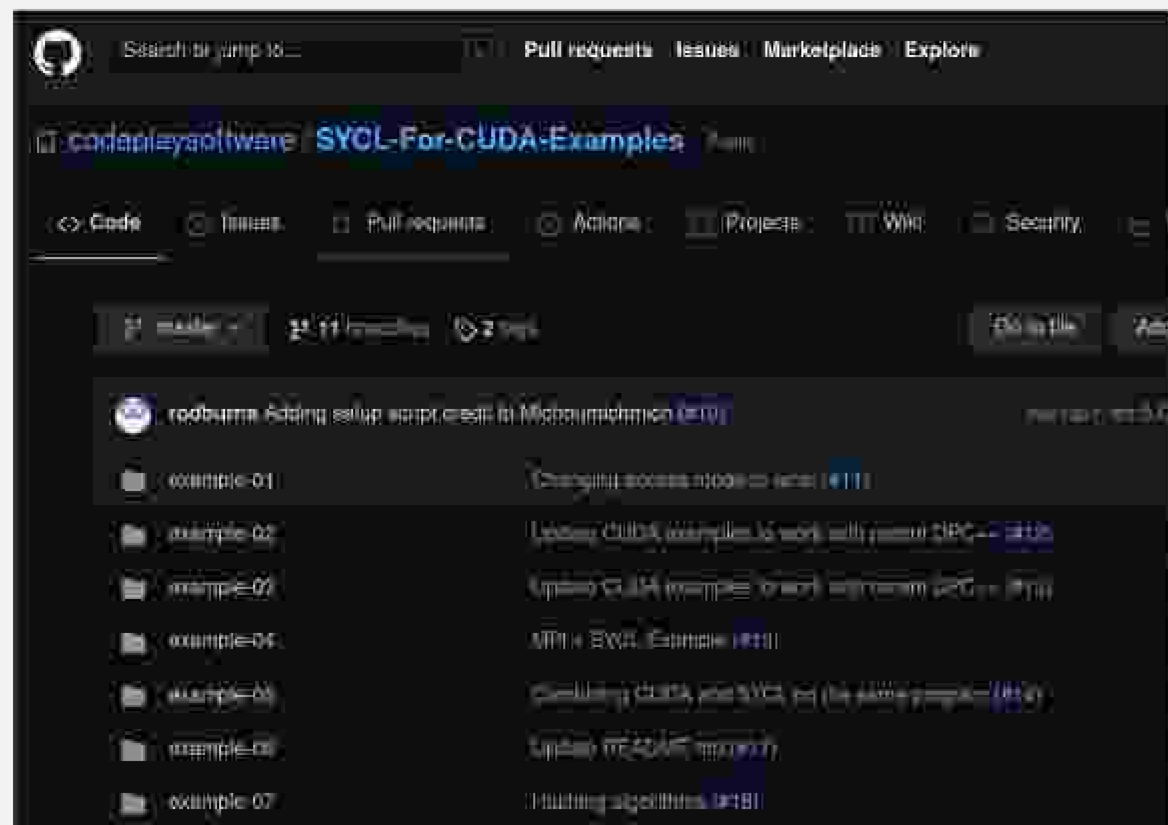
[illegible]

Copyright Codeplay Software Ltd 2021

Hands On Exercises

- Range of topics:
  - CUDA -> SYCL
  - Interop
  - CUDA libraries
  - MPI
  - FORTRAN!
- Easy to get going (docker)
- Get stuck or have a question: sycl@codeplay.com

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

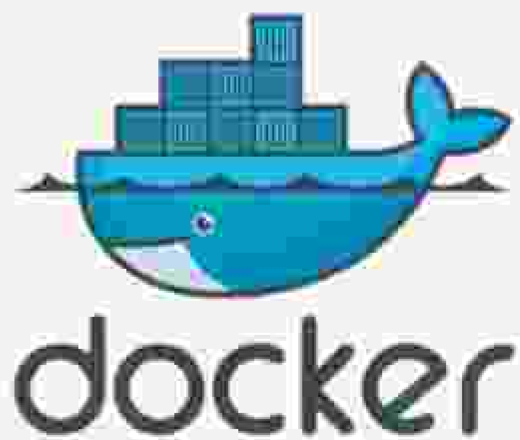


Copyright Codeplay Software Ltd 2021

Dev environment setup

- Use docker image: ruyman/dpcpp_cuda_examples
- Prerequisites:
  - Nvidia hardware
  - Sudo access
  - Docker
  - nvidia-docker2 package

```
# This will pull the image (takes a while)
docker run --gpus all -it ruyman/dpcpp_cuda_examples

# Now we're in the container, find the examples:
cd /home/examples/SYCL-For-CUDA-Examples/
```



Copyright Codeplay Software Ltd 2021

Getting Started with DPC++ for CUDA

- Step 1 : Build DPC++ with CUDA enabled

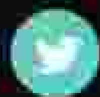
```
git clone https://github.com/intel/llvm.git -b sycl
cd llvm
python ./buildbot/configure.py --cuda -t release --cmake-gen "Unix Makefiles"
cd build
make install -j `nproc`
```

- Step 2 : Call clang++ with your source code

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

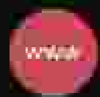

Copyright Codeplay Software Ltd 2021

We're Hiring!



@codeplaysoft

info@codeplay.com



codeplay.com