codeplay

Enable AI & HPC to be Open, Safe and Accessible to All

Bringing SYCL ™ to pre-exascale supercomputing with DPC++ for CUDA

Gordon Brown, Codeplay Software

Partners

BERKELEY LAB

Argonne NATIONAL LABORATORY

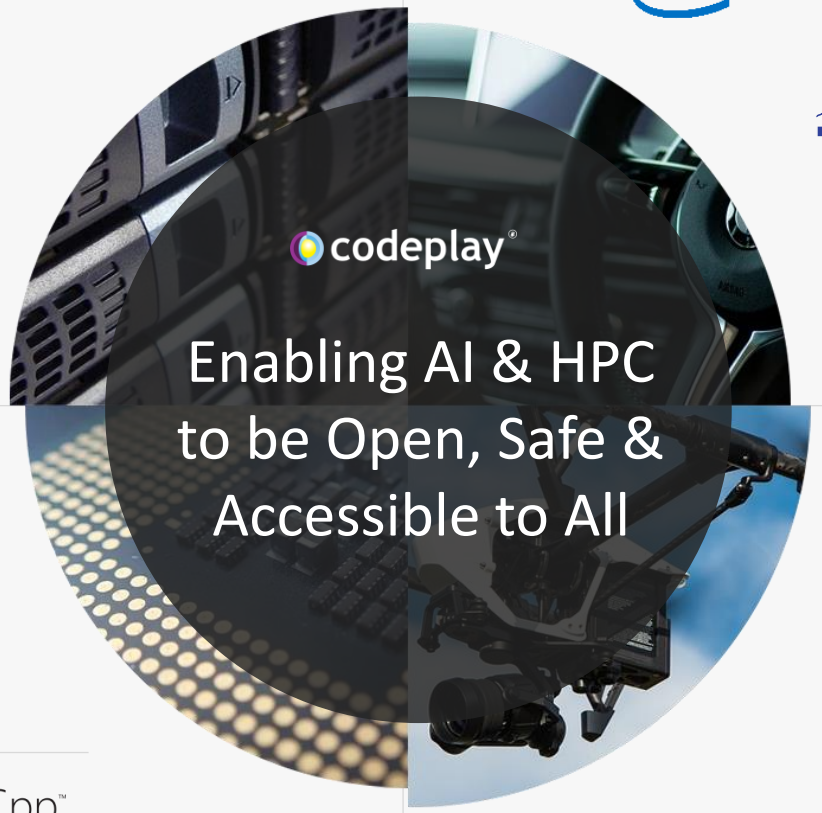
Intel Developer Summit June 2021

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

intel, BROADCOM, CEVA, Imagination, RENESAS, KMC Kyoto Microcomputer Co., Ltd., NSI-TEXE, Berkeley Lab, SYNOPSYS, Argonne National Laboratory

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™

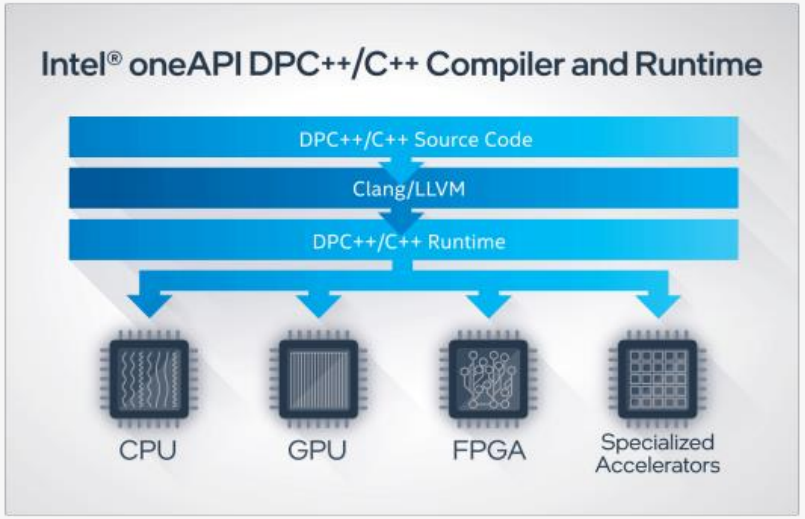
Markets

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

Technologies: Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

SYCL, DPC++, oneAPI

- 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 programming model that includes definitions of standard library interfaces, for common operations such as math.



Codeplay and SYCL

- Part of the SYCL community from the beginning.
- Our team has helped to shape the SYCL standard.
- Implemented the first conformant SYCL product.

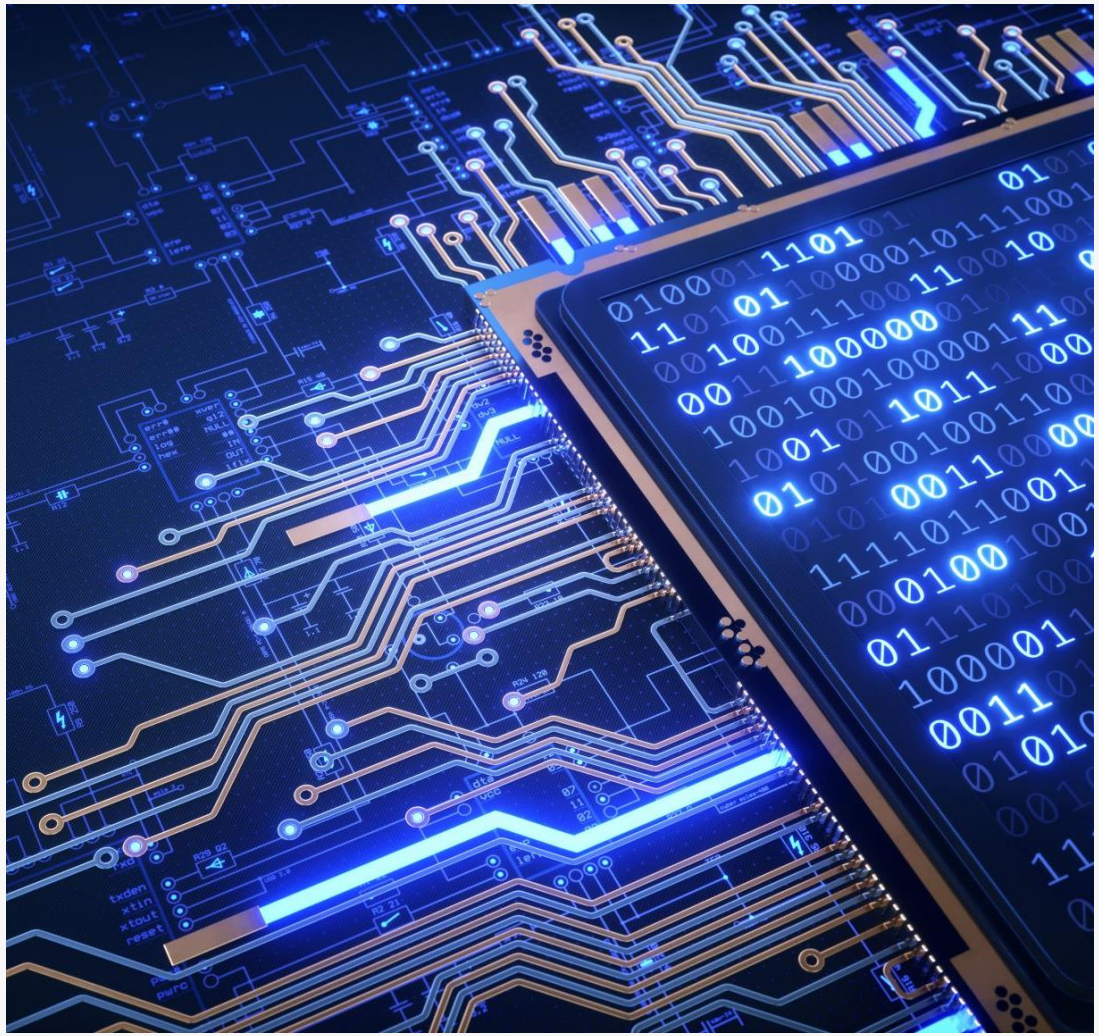
Open Source Contributions

SYCL-DNN SYCL-BLAS SYCL-ML

SYCL 1.2.1 Conformant Implementation

Pre-Exascale and Exascale Supercomputers

- What is a “pre-exascale” supercomputer.
  - A system capable of calculating close to the power of an exascale supercomputer.
  - Exceeding 10^15 floating point operations per second > 1 petaFLOPS.
- What is an “Exascale” supercomputer.
  - A system capable of calculating at least 10^18 floating point operations per second = 1 exaFLOPS.



Bringing SYCL to Exascale

- Argonne National Laboratory (ANL) employs NVIDIA A100 GPU nodes in their ThetaGPU system.
- ANL's upcoming Aurora exascale supercomputer will support SYCL™ on Intel® GPUs.
- Researchers need to run software across both systems.

The Perlmutter Supercomputer

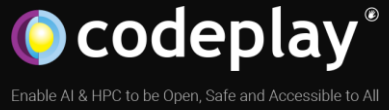
- The new pre-exascale supercomputer at Lawrence Berkeley National Laboratory.
- Named after Saul Perlmutter, astrophysicist at Berkeley Lab and 2011 Nobel Laureate.
- HPE Cray system with CPU-only and GPU-accelerated nodes.
- 6000+ NVIDIA® A100 GPUs.

Performance Portability

- SYCL on Perlmutter and ThetaGPU brings portability and synergy with the Aurora Exascale supercomputer.
- Code can be run without modifications on both supercomputers.
- Researchers collaborate on a single codebase knowing they can target both machines.

DPC++ for CUDA®

- Partnership between Codeplay, Berkeley Lab, and ANL.
- Goals:
  - Expand DPC++ for CUDA support for SYCL 2020 features.
  - Expose NVIDIA Ampere features in DPC++ for CUDA.
  - Optimize DPC++ for CUDA for NVIDIA Ampere hardware.



How To Use DPC++ for CUDA

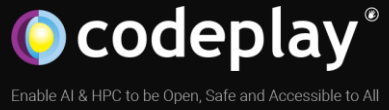
Getting Started with DPC++ for CUDA

- 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`
```

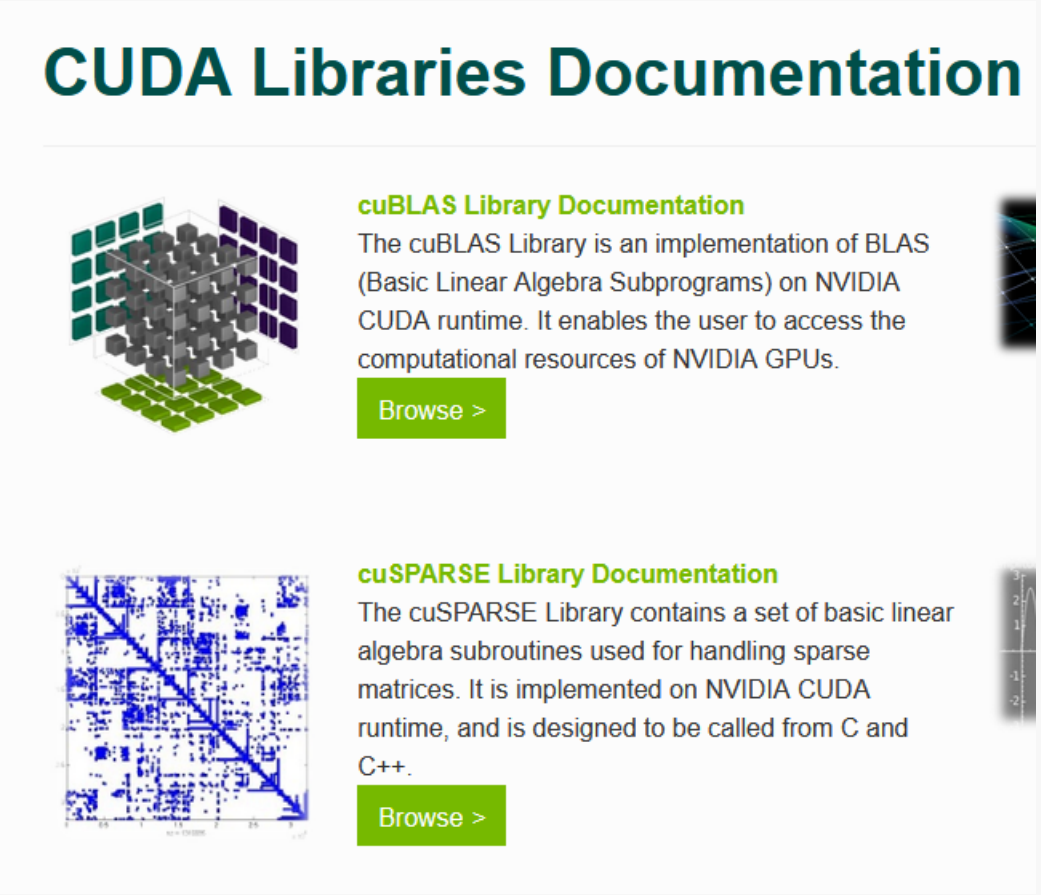
- Step 2 : Call clang++ with your source code

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



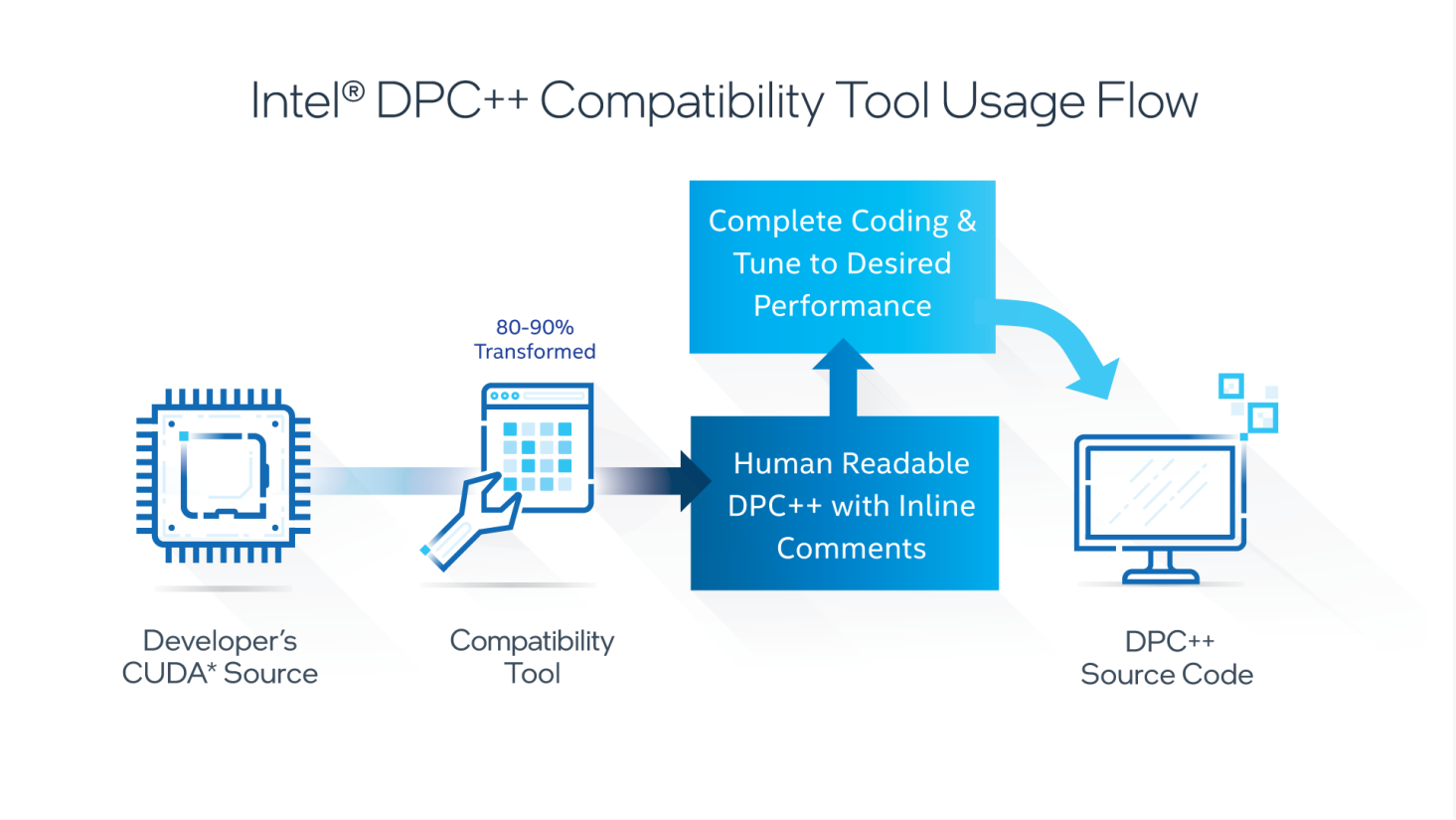
Migrating From CUDA to SYCL

Interoperability with CUDA libraries



- Most CUDA developers use some of the libraries provided by Nvidia and others.
- Examples include cuBLAS and cuDNN.
- Migrating away from these libraries would require a direct replacement.
- DPC++ for CUDA includes interoperability with these libraries.

Compatibility Tool



- Semi-automated code porting with this tool
- Free to use

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);
```

Porting CUDA kernels to SYCL

- CUDA kernels' execution range defined by passing number of blocks per grid, and threads per block
- e.g: `CUDAKernel<<<BlocksPerGrid, ThreadsPerBlock>>>( … );`
- Using SYCL's `nd_range` to define execution range, we specify the global range, and local range
- SYCL global range must be divisible by the local range

Porting CUDA kernels to SYCL

- A SYCL local range = CUDA threads per block
- A SYCL global range = CUDA blocks per grid * threads per block
- Converting the CUDA kernel invocation to SYCL, we need to calculate the global range

Specifying parallel execution range

- 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;

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
 });
}
```

SYCL interop with CUDA libraries

- A useful feature of the CUDA backend is the ability to call native CUDA library routines from SYCL code
- Achieved by invoking an `interop_task` on a command group
- The `interop_task` requires an `interop_handler` parameter

Using interoperability features

- This code snippet demonstrates how to invoke an `interop_task` with an `interop_handler`

The command group

```
myQueue.submit([&](sycl::handler &cgh) {

  // Create accessors here

  cgh.interop_task([=](sycl::interop_handler ih) {

    // Retrieve native object handles from the SYCL backend here

    // Use the native handles as parameters to CUDA library routines

  });
});
```

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.interop_task([=](sycl::interop_handler ih) {

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

                auto cuA = reinterpret_cast<float *>(ih.get_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)

Using Existing CUDA Kernels

- To ease the process of porting CUDA code it is possible to call CUDA kernels from SYCL code in DPC++.
- The implementation tries to match the CUDA syntax.

Invoking CUDA kernels from SYCL

```
int N = 10000; // Size of buffers

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

  auto X_accessor = X_buffer.get_access<access::mode::read_write>(h);

  h.interop_task([=](interop_handler ih) {

    float* dX = reinterpret_cast<float *>(ih.get_mem<backend::cuda>(X_accessor));

    int blockSize = 1024;
    int gridSize = static_cast<int>(ceil(static_cast<float>(N) / static_cast<float>(blockSize))

    myCUDAKernel<<<gridSize, blockSize>>>(dX, N);
  });
});
```

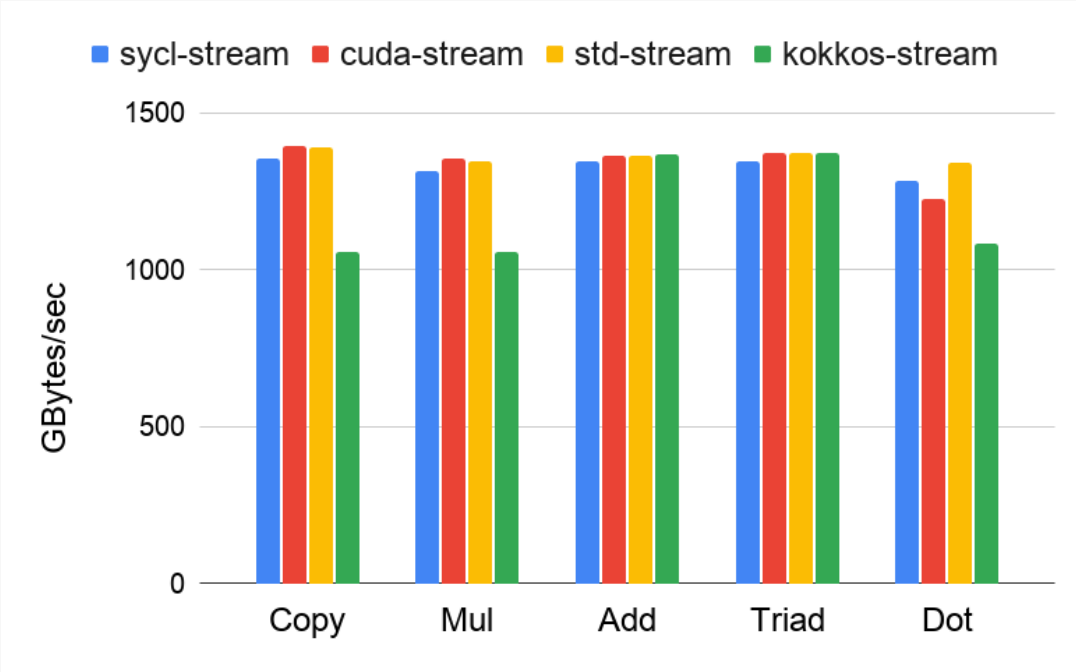
DPC++ performance on Nvidia GPUs

- 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

"BabelStream is a benchmark used to measure the memory transfer rates to/from capacity memory"

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



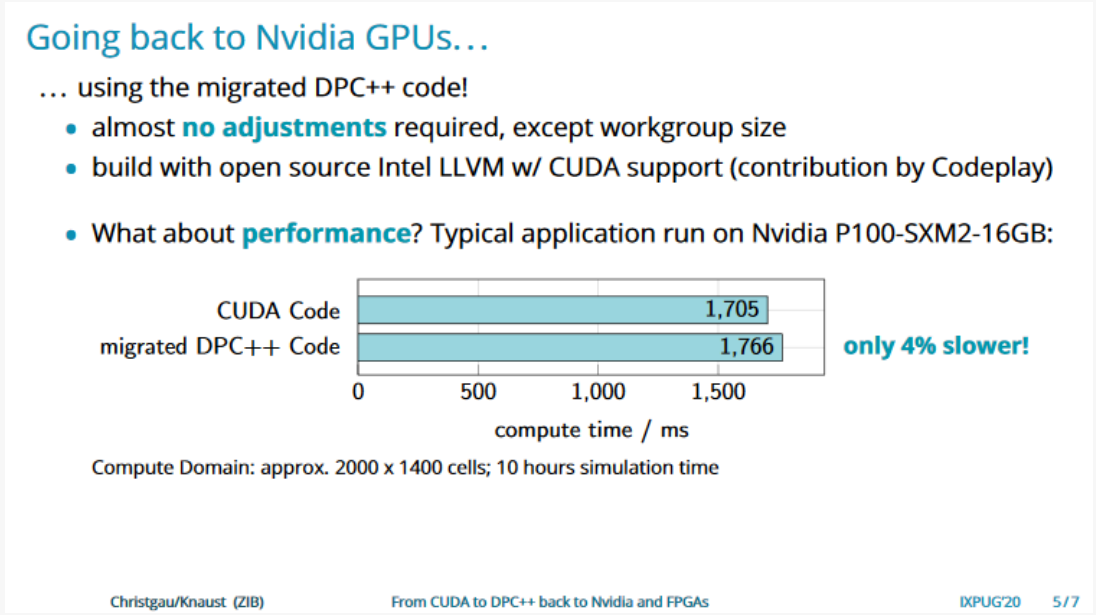
https://github.com/UoB-HPC/BabelStream
@3c637cd04dbbfe3fb49947a0252d33d91abf54fb with default options on A100-SXM4-40GB

clang version 13.0.0 https://github.com/intel/llvm
f126512966d25537c73fd15074a495f825b1f105

CUDA version compiled with nvcc from HPC SDK 20.11 with MEM=DEFAULT NVARCH=sm_80 make -f CUDA.make

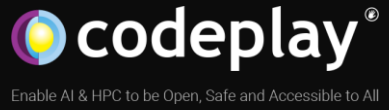
Kokkos-stream compiled with Kokkos 3.3.0

DPC++ Performance on Nvidia GPUs



- Zuse Institute Berlin (ZIB) ported Tsunami simulation code from CUDA to SYCL.
- Direct comparison of CUDA code and DPC++ for Nvidia GPUs.
- Almost no adjustments for optimization.

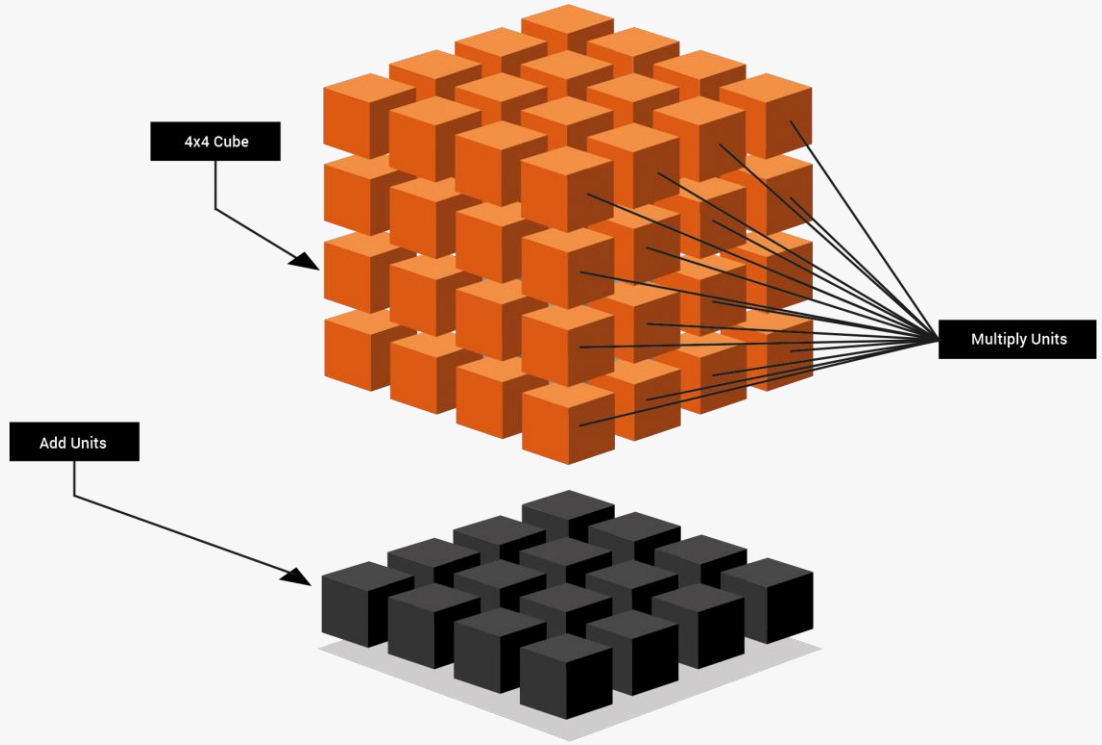
https://www.ixpug.org/resources/download/from-cuda-to-dpc-back-to-nvidia-gpus-and-fpgas-an-oneapi-case-study-with-the-tsunami-simulation-easywave



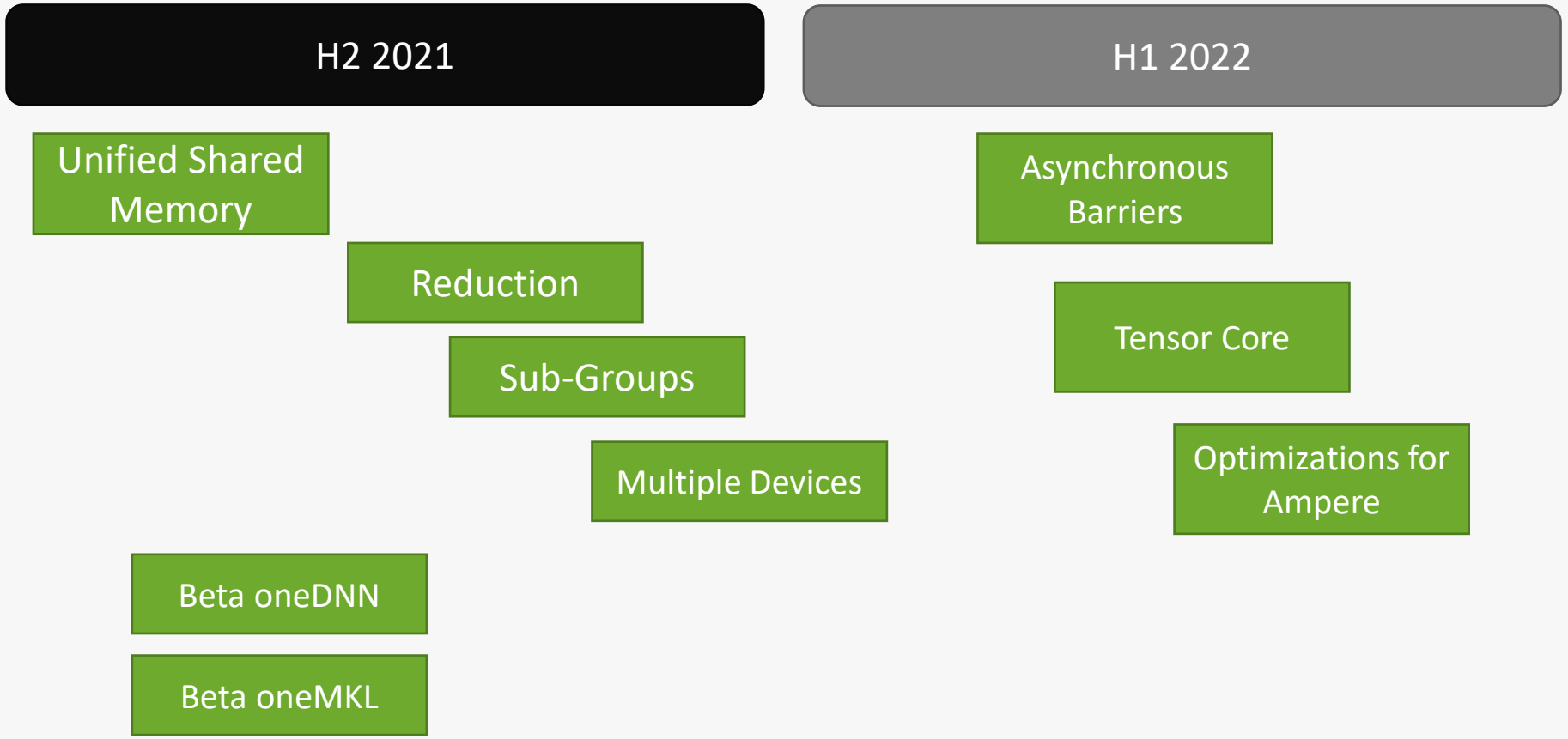
2021 and Beyond

SYCL 2020 Features

- SYCL 2020 feature support in DPC++ for CUDA, including:
  - Unified Shared Memory (USM.)
  - Reductions.
  - Group and sub-group operations.
- Planned SYCL extensions:
  - Asynchronous barriers.
  - Tensor Core support.
- Better DPC++ for CUDA multi-device support.



oneAPI for CUDA: The Future



Where to find out more?

Visit our website for set up instructions and learning materials

www.codeplay.com/oneapiforcuda/

We're Hiring!

codeplay.com/careers/

codeplay

Enable AI & HPC to be Open, Safe and Accessible to All

@codeplaysoft

info@codeplay.com

codeplay.com