Webinar

Learn Heterogeneous C++ with SYCL*

James Reinders
Engineer, Author, Computer Enthusiast

Links to the resources mentioned in this presentation

Same QR code used throughout this presentation

* SYCL is a registered trademark of the Khronos Group, Inc.

Summary

C++ with SYCL* (and oneAPI) allows our application,

in a single version, (yes – in a single build, a single binary!)

to identify and use any number of accelerators

regardless of ***vendor*** or ***architecture*** – and do so

with access to their best performance.

```fortran
! Fortran loop
do i = 1, n
  z(i) = alpha * x(i) + y(i)
end do
```

```cpp
// C++ loop
for (int i=0;i<n;i++) {
  z[i] = alpha * x[i] + y[i];
}
```

```cpp
// SYCL kernel
myq.parallel_for(range{n},[=](id<1> i) {
  z[i] = alpha * x[i] + y[i];
}).wait();
```

EXAMPLE

IS KNOWN TO HAVE ELEMENTS NOT TAUGHT IN THIS BOOK AT ALL. PLEASE ENJOY THIS FORTRAN, SADLY THERE IS NO MORE IN THIS BOOK.

```fortran
! Fortran loop
do i = 1, n
  z(i) = alpha * x(i) + y(i)
end do
```

```cpp
// C++ loop
for (int i=0;i<n;i++) {
  z[i] = alpha * x[i] + y[i];
}
```

```cpp
// SYCL kernel
myq.parallel_for(range{n},[=](id<1> i) {
  z[i] = alpha * x[i] + y[i];
}).wait();
```

"myq"
is how SYCL
supports gives us the ability to direct
work to our device of choice at runtime
(single source: any vendor, any architecture)

EXAMPLE

IS KNOWN TO HAVE ELEMENTS NOT TAUGHT IN THIS BOOK AT ALL. PLEASE ENJOY THIS FORTRAN, SADLY THERE IS NO MORE IN THIS BOOK.

Q: What is SYCL?

A: C++ solution for heterogeneous programming

An open, multivendor, multiarchitecture alternative to proprietary NVIDIA CUDA
(NVIDIA GPUs only)

Provides:

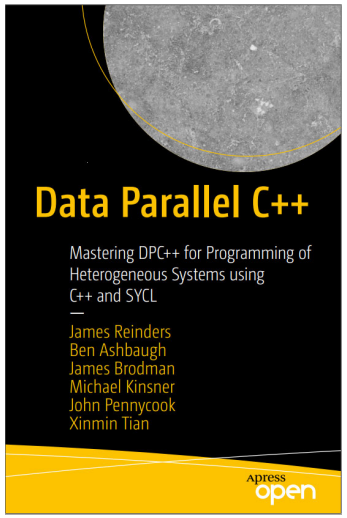
1. find a device,
2. manage memory, and
3. manage offloads.

Q: What is DPC++?

A: LLVM compiler
(includes clang)
implementation of SYCL

Same QR code

The SYCL book



Q: What is oneAPI?

A: Complementary open standards initiative to provide multivendor multiarchitecture support through libraries, profilers, debuggers, etc.

PDF >670K accesses direct from publisher
and
Consistently a top C++ Book on Amazon

Why?

A New Golden Age for Computer Architecture

"The next decade will see a **Cambrian explosion of novel computer architectures**, meaning exciting times for computer architects in academia and industry."

ACM Turing Award laureates
John Hennessy and David Patterson (CACM, Feb 2019, Vol 62, No 2, pp 48-60)

Products

Novel On-Die Accelerators

GPU / Data Parallel

FPGA / Spatial / Dataflow

Deep Learning Optimized

Blockchain

UCIe Universal Chiplet Interconnect Express

Mix & Match Nearly Endless Combinations

Research

Neuromorphic

Graph Analytics and More...

Above are Intel designs… there are many more designs coming from outside Intel, too! We hope to fab those, too!

Without SYCL : recompile for each vendor target

With SYCL : the compiled program is able to find and use accelerators regardless of vendor or architecture

```
1. // SYCL kernel
2. myq.parallel_for(range{n},[=](id<1> i) {
3.   z[i] = alpha * x[i] + y[i];
4. }).wait();
```

* Note: OpenMP and OpenCL share SYCL's open, multivendor, multiarchitecure goals – so they can also do much of what SYCL can in terms of supporting from a single source in a single build.

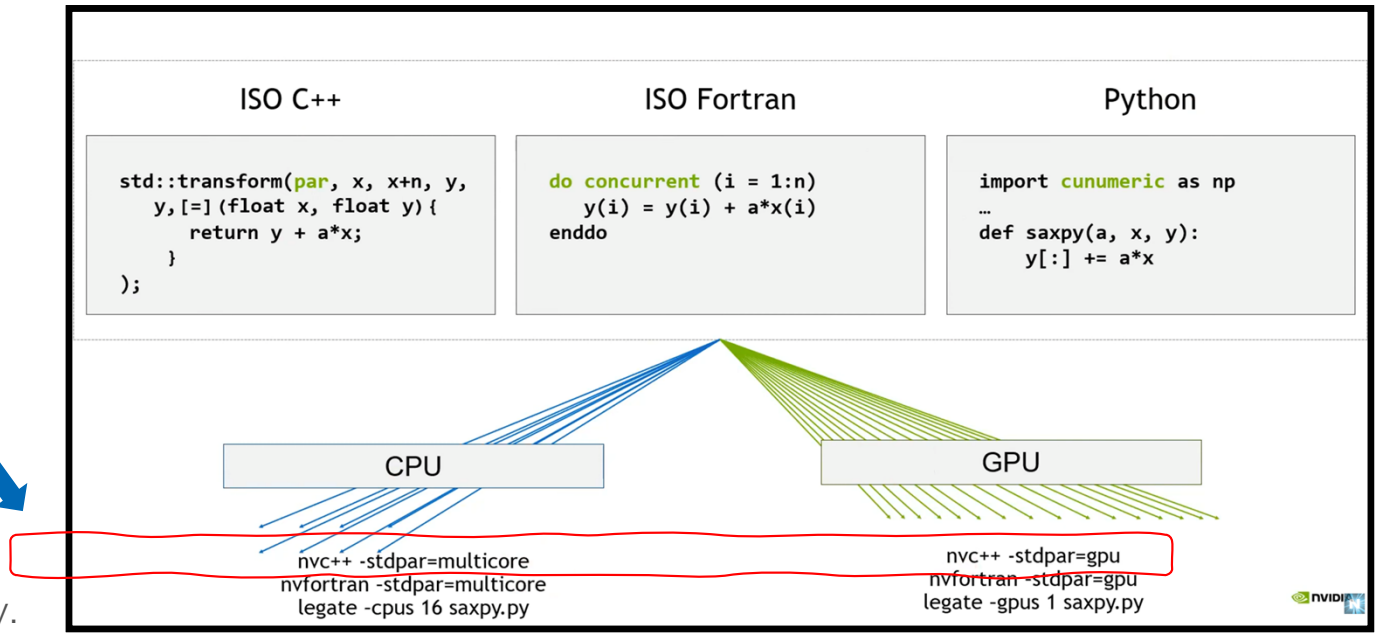
Standard Parallelism is Great – but Compile Often is Not

Slide from YouTube training by Jeff Larkin (NVIDIA*)...
showing clever trick to compile for each target (see https://youtu.be/XqlquGS6o1A)...

vs. SYCL for single source and compile once.

Here
the C++ program is
compiled twice
resulting in two
distinct binaries
One for generic CPU,
and NVIDIA GPU,
and nothing more.
Then users must
choose which
to run.

SYCL has a better way.



Source: https://youtu.be/XqlquGS6o1A

Let's Code : C++ with SYCL

A Complete SYCL Program

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
  queue q;
  int *data = malloc_shared<int>(N, q);
  q.parallel_for(N, [=](auto i) {
      data[i] = i;
  }).wait();
  for (int i=0; i<N; i++) std::cout << data[i] << "\n";
  free(data, q);
  return 0;
}
```

Host code

Accelerator device code

Host code

We can let the runtime pick for us, or we can exercise as much control as we want in device selection.

Choosing a Specific Device for Kernel Execution

SYCL device compute is queue-based

- Each sycl::queue we instantiate is bound a specific device (target for offload)

device_selectors

- Get a device (any device): `sycl::queue queue(); // default_selector{}`
- Get device from class:

```
sycl::queue queue(gpu_selector{});
sycl::queue queue(accelerator_selector{});
sycl::queue queue(cpu_selector{});
sycl::queue queue(host_selector{});
```

- Custom selector:

```
class custom_selector : public device_selector {
  int operator()(……
…
cl::queue queue(custom_selector{});
```

Buffers offer an easy to understand memory model we can control tightly, USM offers a simple C++ pointer based model. Pick one, or use them both, the choice is ours!

Buffer Memory Model

Buffers encapsulate data shared between host and device.

Accessors provide access to data stored in buffers and create data dependences in the graph.

Unified Shared Memory (USM) provides an alternative pointer-based mechanism for managing memory.

```
queue q;
std::vector<int> v(N, 10);
{
  buffer buf(v);
  q.submit([&](handler& h) {
    accessor a(buf, h , write_only);
    h.parallel_for(N, [=](auto i) { a[i] = i; });
  });
}
for (int i = 0; i < N; i++) std::cout << v[i] << " ";
```

Accessor Modes

Access Mode	Description
read_only	Read only Access
write_only	Write-only access. Previous contents not discarded
read_write	Read and Write access

SYCL lets us find and use any device, without restrictions on vendor or architecture.

Submitting to a Device

- A `device` represents a specific accelerator in the system.
- Work is not submitted to devices directly, but to a `queue` associated with the device.
- Creating a queue for a specific device requires a `device_selector`.

```
default_selector selector;
// host_selector selector;
// cpu_selector selector;
// gpu_selector selector;
queue q(selector);
std::cout << "Device: " << q.get_device().get_info<info::device::name>() << std::endl;
```

SYCL does not try to add keywords or change C++ – it uses C++ classes. Very standard. Very portable.

Important Classes in SYCL

Class	Functionality
`sycl::device`	Represents a specific CPU, GPU, FPGA, or other, device that can execute SYCL kernels.
`sycl::queue`	Represents a queue to which kernels can be submitted (enqueued). Multiple queues may map to the same `sycl::device.`
`sycl::buffer`	Encapsulates an allocation that the runtime can transfer between host and device.
`sycl::handler`	Used to define a command-group scope that connects buffers to kernels.
`sycl::accessor`	Used to define the access requirements of specific kernels (e.g. read, write, read-write).

`sycl::range, sycl::nd_range`
`sycl::id, sycl::item,`
`sycl::nd_item`
	Representations of execution ranges and individual execution agents in the range.

Parallel Kernels

- Parallel Kernel allows multiple instances of an operation to execute in parallel.
- Useful to offload parallel execution of a basic for-loop in which each iteration is completely independent and in any order.
- Parallel kernels are expressed using the parallel_for function

for-loop in CPU application

```
for(int i=0; i < 1024; i++){
    a[i] = b[i] + c[i];
});
```

CPU only

Offload to accelerator using parallel_for

```
h.parallel_for(range<1>(1024), [=](id<1> i){
    A[i] =  B[i] + C[i];
});
```

Any vendor, any device - controlled at runtime - including a CPU, GPU, FPGA, etc.

Basic Parallel Kernels

The functionality of basic parallel kernels is exposed via range, id and item classes

- range class is used to describe the iteration space of parallel execution
- id class is used to index an individual instance of a kernel in a parallel execution
- item class represents an individual instance of a kernel function, exposes additional functions to query properties of the execution range

```
h.parallel_for(range<1>(1024), [=](id<1> idx){
        // CODE THAT RUNS ON DEVICE
});
```

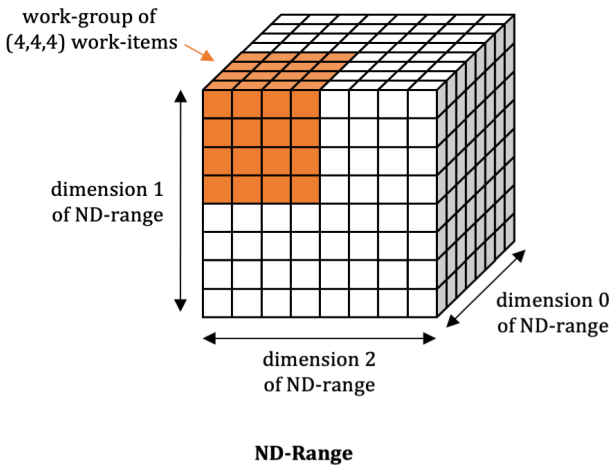
```
h.parallel_for(range<1>(1024), [=](item<1> item){
        auto idx = item.get_id();
        auto R = item.get_range();
        // CODE THAT RUNS ON DEVICE
});
```

ND-Range Kernels

Basic Parallel Kernels are easy way to parallelize a for-loop but does not provide for performance optimization at the hardware level.

ND-Range kernel is another way to expresses parallelism which enables low level performance tuning by providing access to local memory and mapping executions to compute units on hardware.

- The entire iteration space is divided into smaller groups called work-groups, work-items within a work-group are scheduled on a single compute unit on hardware.
- The grouping of kernel executions into work-groups will allow control of resource usage and load balance work distribution.



Asynchronous Execution

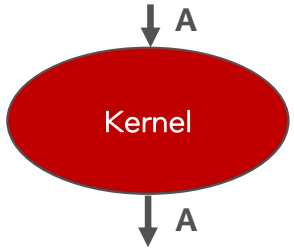
Host

Enqueues kernel to graph, and keeps going

```
#include <CL/sycl.hpp>
constexpr int N=16;
using namespace sycl;
int main() {
  std::vector<int> data(N);
  {
    buffer A(data);
    queue q;
    q.submit([&](handler& h) {
      accessor out(A, h, write_only);
      h.parallel_for(N, [=](auto i) {
        out[i] = i;
      });
    });
  }
  for (int i=0; i<N; ++i) std::cout << data[i];
}
```

Graph

Graph executes asynchronously to host program



Asynchronous Execution

```
int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue q;

  q.submit([&](handler& h) {
    accessor out(A, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor out(A, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor out(B, h, write_only);
    h.parallel_for(R, [=](id<1> i) {
      out[i] = i; }); });

  q.submit([&](handler& h) {
    accessor in(A, h, read_only);
    accessor inout(B, h);
    h.parallel_for(R, [=](id<1> i) {
      inout[i] *= in[i]; }); });
}
```

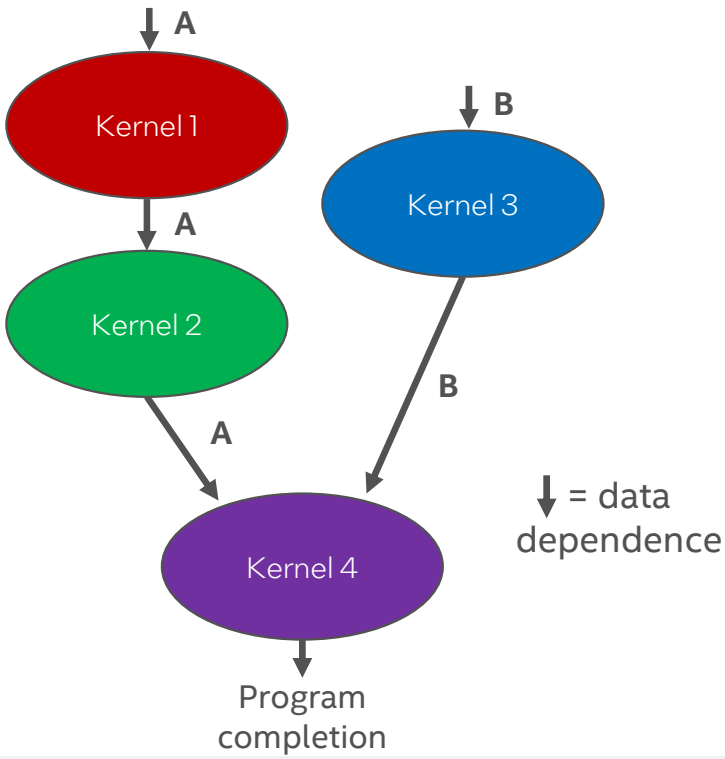
Kernel 1

Kernel 2

Kernel 3

Kernel 4

Data and control dependences are resolved by the runtime



Implication of Asynchronous : Buffer made dependencies obvious, USM does not… so we need to manage that!

Task Scheduling with USM

Explicit Scheduling

- Submitting a kernel returns an Event
- Wait on Events to order tasks

```
auto E = q.submit([&] (handler& h) {
  auto R = range<1>{N};
  h.parallel_for(R, [=] (id<1> ID) {
    auto i = ID[0];
    C[i] = A[i] + B[i];
  });
});
E.wait();
```

Graph Scheduling

- Build Task Graphs from Events

```
auto R = range<1>{N};

auto E = q.submit([&] (handler& h) {
  h.parallel_for(R, [=] (id<1> ID) {…});
});

q.submit([&] (handler& h) {
  h.depends_on(E);
  h.parallel_for(R, [=] (id<1> ID) {…});
});
```

Implication of Async : C++ only supports synchronous errors; SYCL lets us manage asynchronous errors.

Error Handling: C++ Exception-based

e.g. https://developer.codeplay.com/computecppce/latest/sycl-guide-error-handling

```
auto exception_handler = [] (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:\n" <<
        e.what() << std::endl; } } };

sycl::queue queue(sycl::default_selector{}, exception_handler);

queue.submit([&] (sycl::handler& cgh) {
  auto range = sycl::nd_range<1>(sycl::range<1>(1), sycl::range<1>(10));
  cgh.parallel_for<class invalid_kernel>(range, [=] (sycl::nd_item<1>) {});
});

try { queue.wait_and_throw(); }
catch (sycl::exception const& e) {
  std::cout << "Caught synchronous SYCL exception:\n”
    << e.what() << std::endl;
}
```

SYCL is based on C++

- Errors in C++ are handled through exceptions
- SYCL uses exceptions, not return codes!

1. Synchronous exceptions

- Thrown immediately when an API call fails (e.g., can’t create buffer)
- Act as normal C++ exceptions. Things you’d expect OpenCL to error on

2. Asynchronous exceptions

- Caused by a future failure (e.g., during kernel execution – recall that is asynchronous to code)
  - Host program has already moved on to new things!
- Programmer provides processing function, and controls when to process

C++ with SYCL

The Open Solution for Hardware Innovation

- Any Vendor
- Any Architecture
- Increased Portability
- No Vendor Lock-In

Upgrade to C++ with SYCL

NVIDIA CUDA

Migrate from CUDA* to C++ with SYCL*

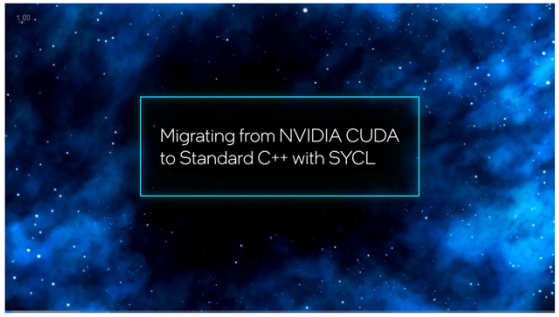
Unlock your code from the constraints of vendor-specific tools and accelerators.

Introduction | Examples | Learn | Port | Community

Introduction

C++ and SYCL* delivers a unified programming model, performance portability, and C++ alignment for applications using accelerators. Explore the following links to:

- Learn how to migrate your code to SYCL
- Download the software to get started
- See examples from other users
- Join the community to showcase your own migration projects



Examples of CUDA Migration to SYCL

Ginkgo Sparse Linear Algebraic Library

For extremely performance-critical code (such as low-level linear algebraic libraries), engineering work-arounds might become necessary to achieve the desired performance and portability goals. Ginkgo library developers demonstrate how such work-arounds might be applicable to other libraries.

Watch the Video

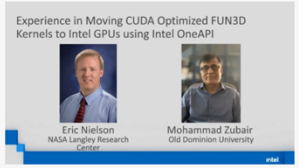
Learn More (GitHub*)

Rodinia Benchmark Suite

Zheming Jin, a researcher at Oak Ridge National Laboratory (ORNL), evaluates DPC++ by porting several benchmarking applications.

Watch the Video

Learn More (GitHub)



Kernels Optimized for CUDA

Eric Nielson (a research scientist at NASA*) and Dr. Mohammad Zubair (a computer science professor at Old Dominion University) share their experiences with moving kernels optimized for CUDA* to oneAPI.

Watch the Video

Pre-exascale Supercomputing

Gordon Brown (a principal software engineer at Codeplay*) talks

Empowering GROMACS Cross-Architecture Development

CUDA "Cheat Sheet"

CUDA	SYCL
thread	work-item
block	work-group
grid	nd-range

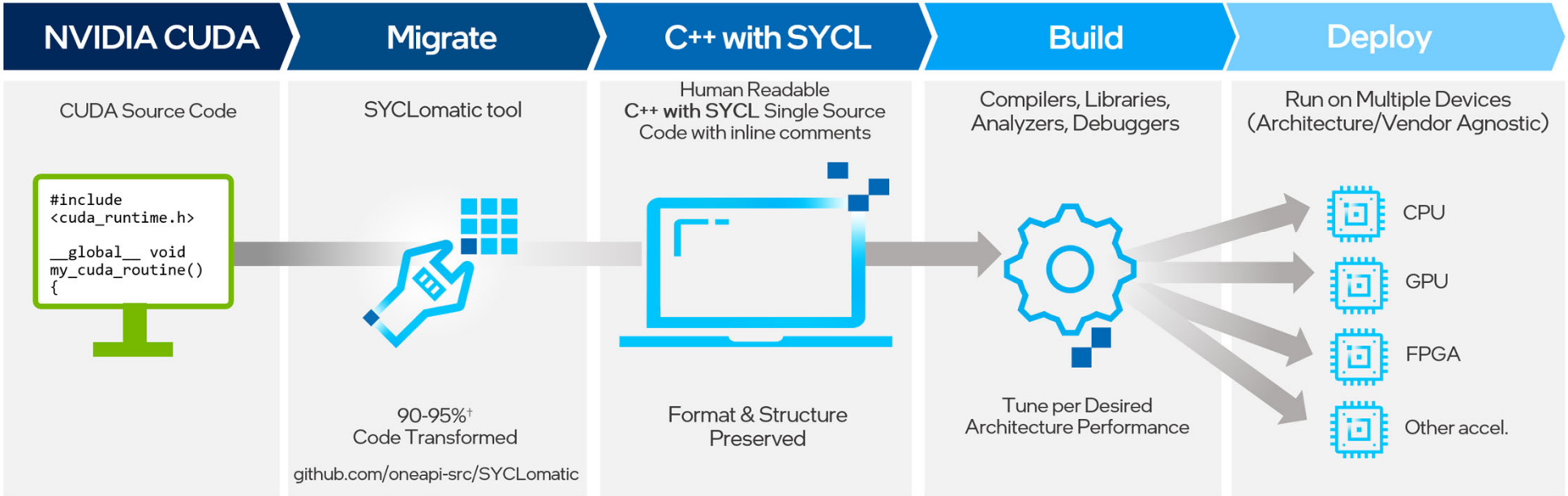
CUDA		SYCL	
Memory Type	Scope	Memory Type	Scope
Register memory	Thread	Private memory	Work-item
Shared memory	Block	Local memory	Work-group
Global memory	Grid (all threads)	Global memory	All work Items

CUDA "Cheat Sheet" (for cuBLAS)

	NVIDIA	AMD	Intel
BLAS	cuBLAS	rocBLAS	oneMKL
Linear Solvers	cuSOLVER	rocSOLVER	oneMKL
Random Numbers	cuRAND	rocRAND	oneMKL
FFT	cuFFT	rocFFT	oneMKL

oneMKL aims to handle them all:
open-source implementation of the oneMKL Data Parallel C++ (DPC++) interface works with multiple devices (backends) uses vendor device-specific libraries underneath

CUDA‡ to SYCL‡ Code Migration & Development Workflow



† Intel estimates as of September 2021. Based on measurements on a set of 70 HPC benchmarks and samples, with examples like Rodinia, SHOC, PENNANT. Results may vary.
‡ Other names and brands may be claimed as the property of others. SYCL is a trademark of the Khronos Group Inc.

Migration example

```cuda
#include <cuda.h>

#define VECTOR_SIZE 4
__global__ void VectorAddKernel (float *A, float *B, float *C)
{
  A[threadIdx.x] = threadIdx.x + 1.0f;
  B[threadIdx.x] = threadIdx.x + 1.0f;
  C[threadIdx.x] = A[threadIdx.x] + B[threadIdx.x];
}

int main()
{



  float *d_A, *d_B, *d_C;
  cudaMalloc(&d_A, VECTOR_SIZE*sizeof(float));
  cudaMalloc(&d_B, VECTOR_SIZE*sizeof(float));
  cudaMalloc(&d_C, VECTOR_SIZE*sizeof(float));



  VectorAddKernel<<<1, VECTOR_SIZE>>>(d_A, d_B, d_C);



  float Result[VECTOR_SIZE] = { };
  cudaMemcpy(Result, d_C,VECTOR_SIZE*sizeof(float),cudaMemcpyDeviceToHost);

  cudaFree(d_A);
  cudaFree(d_B);
  cudaFree(d_C);
}
```

Header files

Kernel

Mem alloc

Kernel call

Mem copy

Mem free

```sycl
#include <CL/sycl.hpp>
#include <dpct/dpct.hpp>
#define VECTOR_SIZE 4
void VectorAddKernel (float *A, float *B, float *C, sycl::nd_item<3> item_ct1)
{
  A[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
  B[item_ct1.get_local_id(2)] = item_ct1.get_local_id(2) + 1.0f;
  C[item_ct1.get_local_id(2)] = A[item_ct1.get_local_id(2)] + B[item_ct1.get_local_id(2)];
}

int main()
{
  dpct::device_ext &dev_ct1 = dpct::get_current_device();
  sycl::queue &q_ct1 = dev_ct1.default_queue();

  float *d_A, *d_B, *d_C;
  d_A = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
  d_B = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);
  d_C = sycl::malloc_device<float>(VECTOR_SIZE, q_ct1);

  q_ct1.submit([&](sycl::handler &cgh){
    cgh.parallel_for(sycl::nd_range(sycl::range(1, 1, VECTOR_SIZE),
      sycl::range(1, 1, VECTOR_SIZE)), [=](sycl::nd_item<3> item_ct1) {
      VectorAddKernel(d_A, d_B, d_C, item_ct1);
    });
  });

  float Result[VECTOR_SIZE] = { };
  q_ct1.memcpy(Result, d_C, VECTOR_SIZE * sizeof(float)).wait();

  sycl::free(d_A, q_ct1);
  sycl::free(d_B, q_ct1);
  sycl::free(d_C, q_ct1);
}
```

From CUDA to SYCL

Michel Migdal – Codeplay / ENSIIE / Paris-Saclay

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

One user's experience:
Results show improved performance after CUDA→SYCL on NVIDIA.
Obviously – your results can and will vary.

CUDA and SYCL isn't about different performance (unless compiler implementations vary in how well they optimize).

The key is that SYCL is multivendor (single source) and CUDA is definitely not.

Results show comparable performance after CUDA→SYCL on NVIDIA and after HIP → SYCL on AMD.

SYCL is delivering on its promise of multivendor support (performance included!)

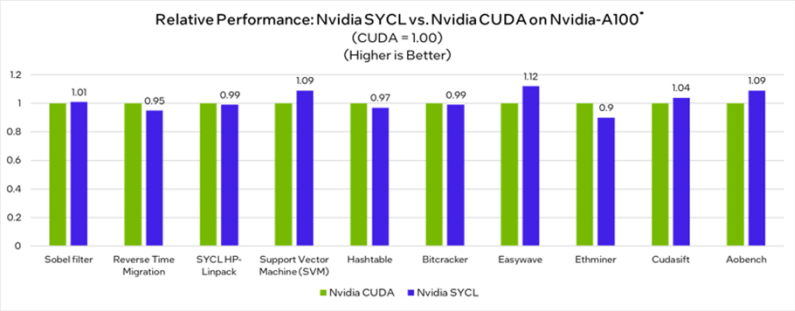
SYCL* Performs Across GPUs

SYCL is an object-oriented, parallel programming language running on multiple computes/accelerators (CPU, GPU, FPGA) from Intel and the industry. It includes a robust programming environment plus libraries/SDKs/tools. **Hypothesis is that SYCL is portable across compute architectures** (write once, run everywhere) and **provides performance comparable to native and established programming environments** on those compute units.

oneAPI

Objective

Analyze performance of SYCL benchmarks spanning many industry domains **on AMD and Nvidia GPUs**, compare benchmark performance with that of same benchmarks implemented in AMD HIP* and Nvidia CUDA* on those machines.



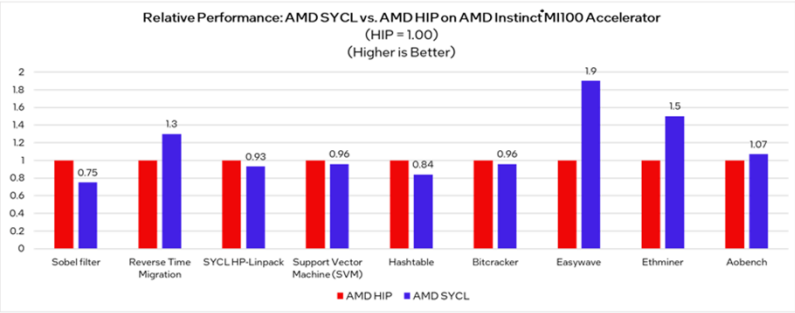
Results

On an NVIDIA A100* system for 10 workloads, **SYCL performance is comparable to CUDA.** On an AMD Instinct* MI100 system, for 9 workloads, **SYCL performance is comparable to HIP.**

Variances are due to maturity and capabilities of different compiler and runtime toolchains.

Methodology

Develop benchmarks in SYCL, CUDA, and HIP, ensuring **algorithms/datasets are the same** across languages. **Tune each version for the compute architecture.** Characterize, analyze, and compare benchmark performance on each system for SYCL and language native to that system.



Analysis

For workloads that measure time (defined as time spent on device and time used in data transfer between host and device). **Workloads selected include domains such as HPC, Heavy Compute, ML/DL, Ray Tracing, and Crypto/Bit-Mining.** Each measurement is run 10 times; first measurement is discarded and then average is taken. **In most cases, better performance was attributed mainly to efficient code generation by compiler** and the time differences in setup/initialization.

Conclusion

We see from the results that **SYCL is a highly performant language on Nvidia and AMD** devices and **performs comparably to native CUDA or HIP code** for diverse workloads. Development environment and supporting components (compilers, tools, libraries) are highly efficient and competitive. Available tools simplify porting code from CUDA to SYCL.

Performance varies by use, configuration and other factors. Learn more on the Performance Index site. Your costs and results may vary.

* SYCL is a registered trademark of the Khronos Group, Inc.

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

intel. innovation

Summary

C++ with SYCL (and oneAPI) allows our application, in a single version, to identify and use any number of accelerators regardless of vendor or architecture – and do so with access to their best performance.

C++ with SYCL (and oneAPI) are foundational efforts to ensure accelerated computing is open, multivendor, and multiarchitecture.

Higher layers of the software stack benefit without being forced to change.

Offering an essential alternative to proprietary foundations.

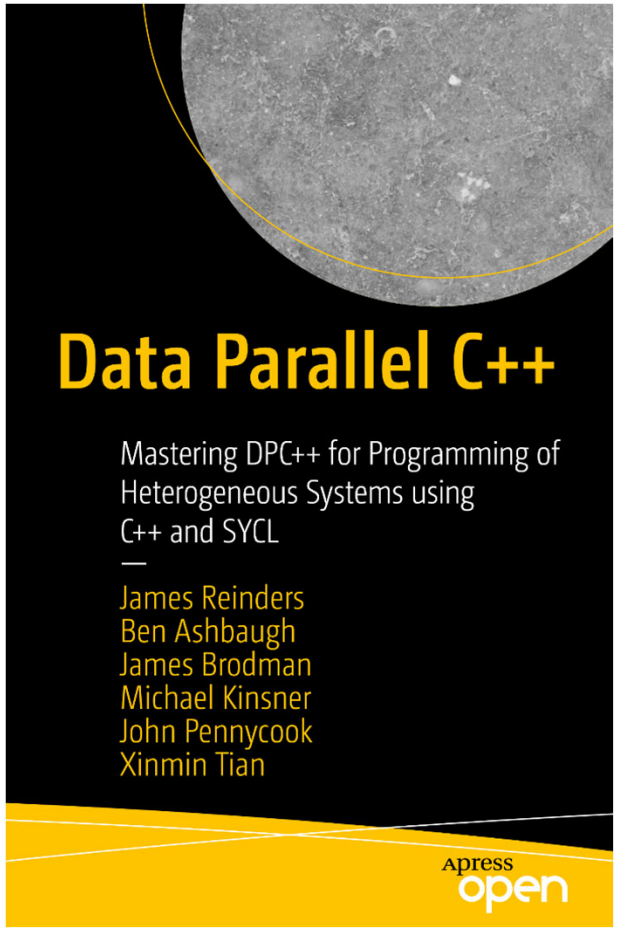
Q&A

https://sycl.tech

SYCL Academy

Same QR code –
Has links to the resources
mentioned in this presentation

Online training, Book download, Sample codes, Articles, and more.

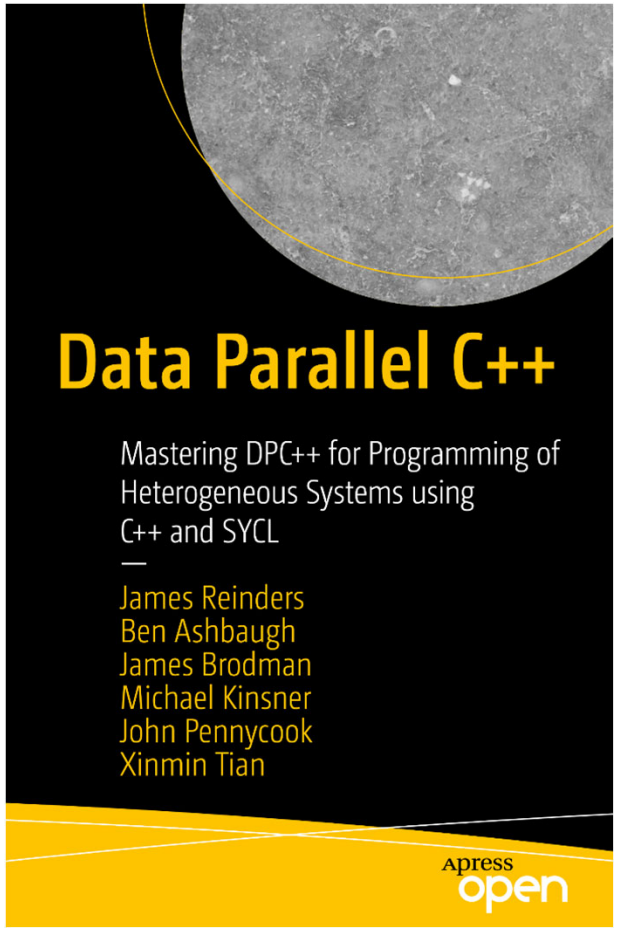


Learn more…

https:// sycl . tech

SYCL Academy

Same QR code



Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details.
No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

© Intel Corporation. Intel, the Intel logo, Xeon, Core, VTune, OpenVINO, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

SYCL is a registered trademark of the Khronos Group, Inc.

intel®

Standard Parallelism is Great – but Compile Often is Not

Standard C++ (including parallelism) is augmented by SYCL

for single source and compile once regardless of vendor or architecture – and with access to full performance.

CHAPTER 13 PRACTICAL TIPS

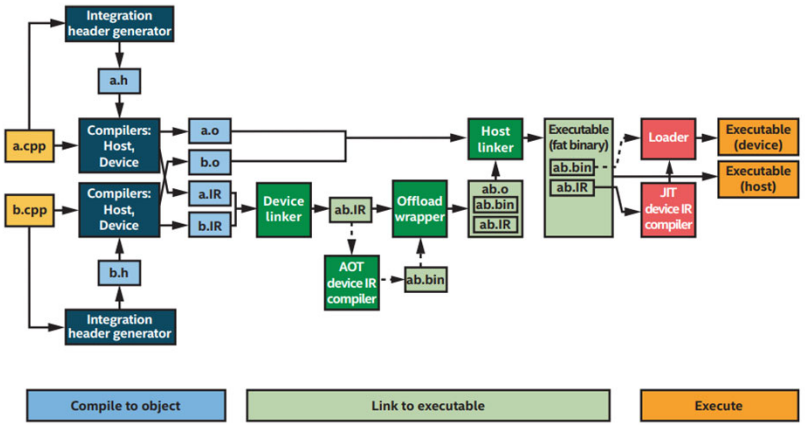


Figure 13-2. *Compilation process: Ahead-of-time and just-in-time options*