Performance portability with the SYCL programming model

Brookhaven National Laboratory
Computational Science Initiative
HPC Seminar Series

Vincent R. Pascuzzi

January 13, 2022

DOE HPC Systems

2013
~10 PFLOPS

2016
~10 PFLOPS

2018
~100 PFLOPS

2021
~100 PFLOPS

2022
~1000 PFLOPS

Homogeneous

Heterogenous

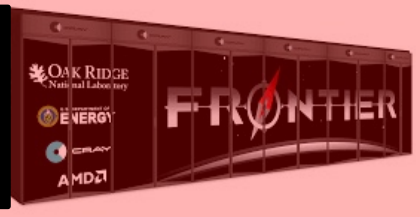
DOE HPC Systems

2013	2016	2018	2021	2022
~10 PFLOPS	~10 PFLOPS	~100 PFLOPS	~100 PFLOPS	~1000 PFLOPS

"EASY"

Homogeneous

"HARD"



Heterogenous

“Easy” vs. “Hard”

CPU

- ~10-100 “complex” cores per socket
- Wide lanes (AVX-512, SVE 128-2048)
- Deep cache hierarchy; NUMA
- MCDRM, HBM, Grace
- Mainly x86 ISA, but also Power, ARM, ... but *primarily C/C++/fortran*

GPU

- ~100-1000 “simple” cores
- Core becoming more complex, specialized; *e.g.*, in-core accelerators for AI/ML
- Interconnects
- HBM, GDDR
- Deepening memory hierarchy; *e.g.*, cache, shared scratchpad, ...
- *Numerous vendors, architectures and programming models*

Programming models (specification/standard/API/...) for heterogeneous platforms

Goals

Excellent support for:

- CPU latency calculations through concurrency/asynchrony
- CPUs throughput via parallelism
- Heterogeneous computational throughput

GUPM (Grand Unified Programming Model): close, but no cigar

- Heterogeneity and/or general massive parallelism
- Cross platform: HPC, clusters, consumer desktop/laptop, embedded devices, ...
- High-bandwidth memory
- Distributed cloud (converged/federated) architecture: beyond classical computing

Directive-based programming models

- Directives/pragmas language constructs used to specify to translator/compiler how to process input
- Typically include library routines (e.g., configuration, orchestration, ...) and environment variables to influence run-time behavior
- Portable! Standards-based!

```
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```

Tells run-time to split the following code block among threads; *e.g.*, two worker threads may each take half the range.

Directive-based programming models

- OpenMP is likely the most performant portable among all, massive support, seasoned
- Developer has full control over implementation of parallel constructs, but:
  - Not required to handle dependencies, conflicts, race conditions, deadlocks, ...
  - Some people find this (naturally) cumbersome

Can we do better? *Use std C++ to express all intra-node parallelism!*

- Lambdas (function object)
- Execution policy (where to run)
- Dimension/shape constructs for data
- Kokkos, RAJA, Alpaka, SYCL, ... currently exploratory research in programming models but aim to be made part of future C++ std

Eventually, it will not be {Kokkos, SYCL,...} anymore---it will just be C++.

C++-based programming models

- Explicit parallelism
- Parallel patterns
- Queue models
- Implicit or explicit data movement

SYCL

Dot-product: Vectorization

```
std::vector<float> data1 = {...};
std::vector<float> data2 = {...};
inner_product(
   datapar,// parallel and vectorizedexecution
   std::begin(data1), std::end(data1),
   std::begin(data2),
   0.0f,
   [](autot1, autot2) { returnt1 + t2; },// std::plus<>()
   [](autot1, autot2) { returnt1 * t2; } // std::multiplies<>()
);
```

Kokkos

Sparse-Matrix-Vector Multiply

```
parallel_for( TeamPolicy<Space>(nrow,AUTO),
KOKKOS_LAMBDA(
TeamPolicy<Space>::member_type member ) {
  const int i = member.league_rank();
  double result = 0 ;
  parallel_reduce(
    TeamThreadRange(member,irow[i],irow[i+1]),
    [&]( int j , double & tmp) { tmp += A[j] * x[jcol[j]];},
    result );
  if ( member.team_rank() == 0 ) y[i] = result ;
});
```

RAJA

```
double* x ; double* y ;
double a ;
RAJA::SumReduction<reduce_policy, double> tsum(0);
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

C++-based programming models

- Explicit pa
- Parallel pa
- Queue mo
- Implicit or
  movemen

SYCL

Dot-product: Vectorization

```
std::vector<float> data1 = {...};
std::vector<float> data2 = {...};
inner_product(
```

Kokkos

Sparse-Matrix-Vector Multiply

```
parallel_for( TeamPolicy<Space>(nrow,AUTO),
KOKKOS_LAMBDA(
TeamPolicy<Space>::member_type member ) {
league_rank();

ge(member,irow[i],irow[i+1]),
tmp) { tmp += A[j] * x[jcol[j]];},

k() == 0 ) y[i] = result ;
```

At this point, it's difficult to decide which programming model to go with.

- RAJA least mature
- Kokkos is most mature
- SYCL has significant industry backing, lower-level (Kokkos, RAJA, ... will almost necessarily—if not already—support SYCL)

Of course, this is a no-brainer if targeting only one vendor...

```
RAJA::MinReduction<reduce_policy, double> tmin(MYMAX);

RAJA::forall< exec_policy > ( IndexSet , [=] (int i) {
     y[i] += a * x[i] ;
     tsum += y[i];
     tmin.min( y[i] );
});
```

HEP Center for Computational Excellence (CCE)

A Department of Energy High-Energy Physics program investigating:

- Performance portability
- I/O
- Complex workflows
- Event generators*

Portable Parallelization Strategies (PPS) effort focuses on performance and portability solutions for current and future HEP software

- Select among the participating experiments a number of x86-based ‘testbeds’ and rewrite the codes in various programming models

HEP-CCE Experiments

* Event generator software is written and maintained by theorists.

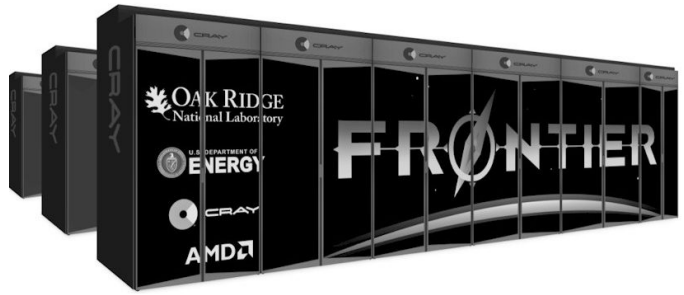
Brookhaven National Laboratory

Limited number of developers (we are physicists) and there are numerous platforms with various architectures

- Large codebases which need to stand the ‘test of time” (~decade)
- We cannot afford to support and maintain multiple codebases
- We need to utilize leadership computing facilities.
- We need portability and achieve a fair level of performance.

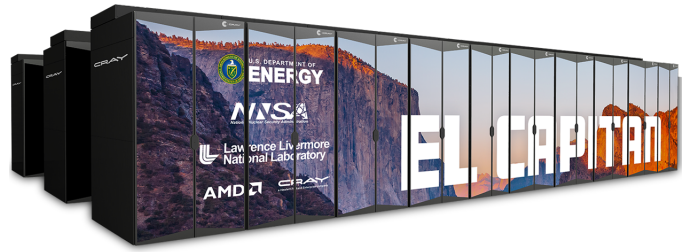
National Energy Research Scientific Computing Center (NERSC), 2021
AMD CPU, Nvidia GPU

Oak Ridge National Laboratory (ORNL), 2021
AMD CPU, AMD GPU



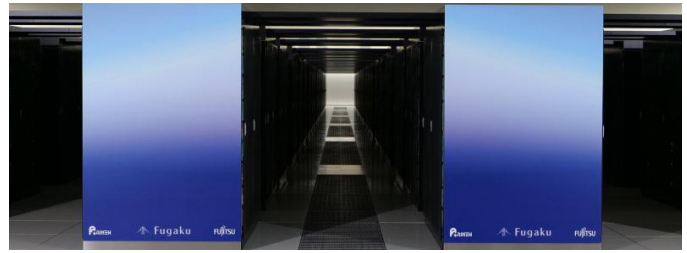
Argonne National Laboratory (ANL), 202?
Intel CPU, Intel GPU

Lawrence Livermore National Laboratory (LLNL), 2023
AMD CPU, AMD GPU



Swiss National Supercomputing Center (CSCS), 2023
Nvidia/ARM CPU, Nvidia GPU

Riken Center for Computational Science, 2021
ARM CPU



Brookhaven National Laboratory

SYCL (pronounced “sickle”)

A C++-based open standard developed by Khronos Group

- Cross-platform abstraction layer

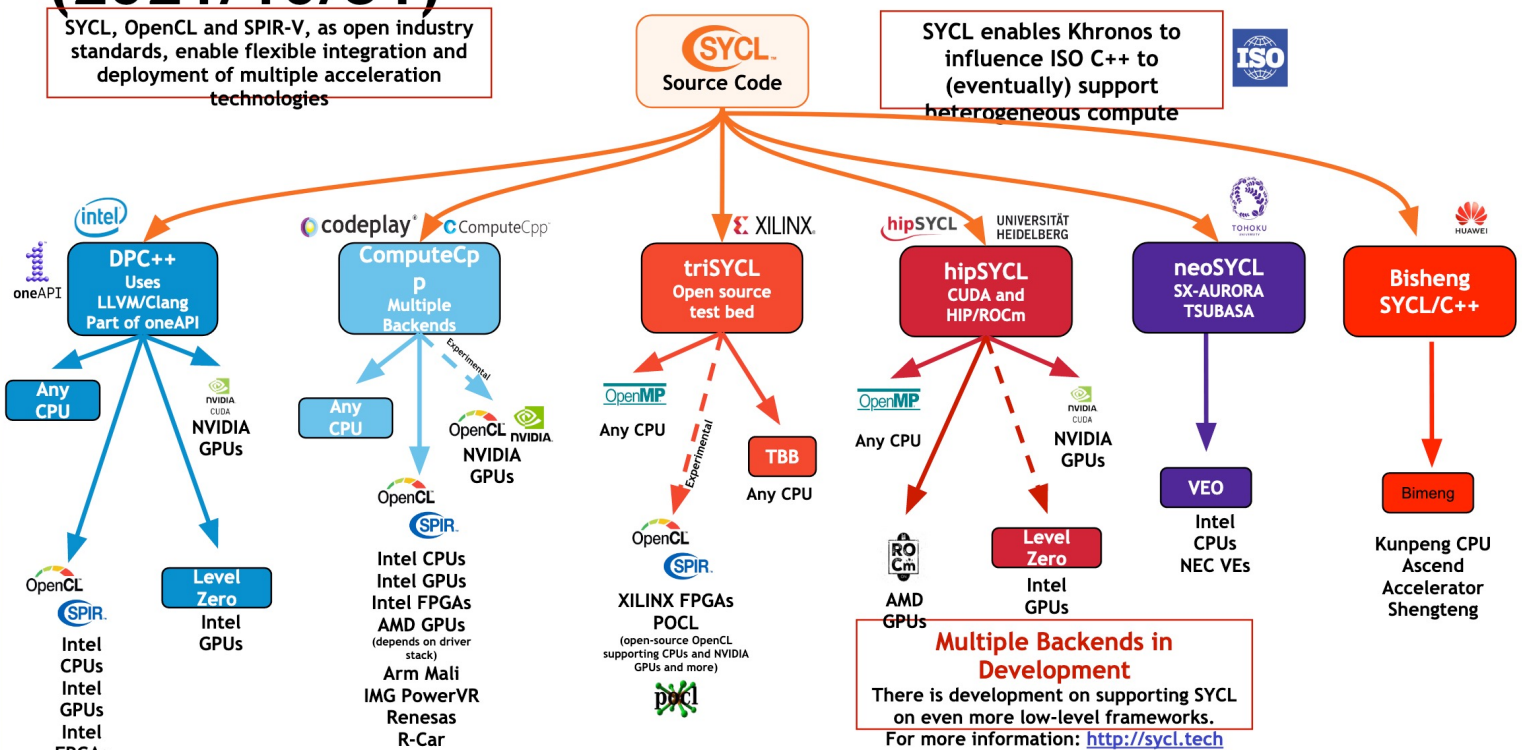
Provides a single-source programming model for development of heterogeneous software systems

- Both low- and high-level codes

Vast ecosystem

- Numerous implementations, targeting different platforms

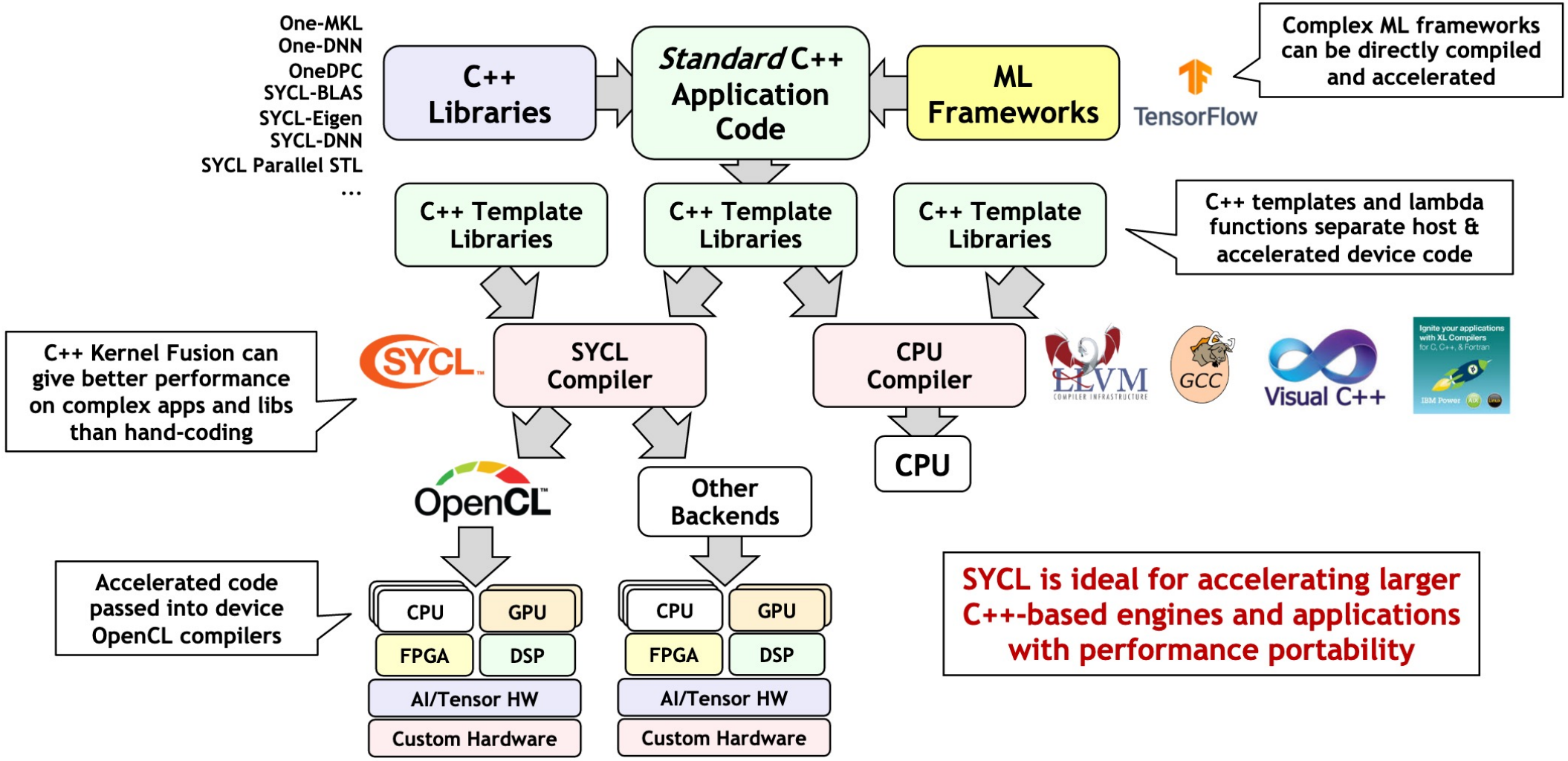
SYCL Implementations in Development (2021/10/31)



Major features of SYCL 2020

- **Unified Shared Memory**: pointer-based approach without buffers and accessors
- **Parallel reductions**: Built-in, optimized reduction operations
- **Work-group and sub-group algorithms**: efficient parallel operations between work-items
- **Sub-devices**: logical partitioning of a physical device (limited support)
- **Atomic operations**: more closely aligned with C++20 standards
- **Improved interoperability**: more efficient acceleration, leverage existing highly-optimized third-party libraries (open or proprietary)

Single-source C++ for heterogeneous applications



Performance portability

"An application is performance portable if it achieves a consistent ratio of the actual time to solution to either the best-known or the theoretical best time to solution on each platform with minimal platform specific code required." [1]

Useful metric should:

- Be measured specific to a set of platforms of interest H
- Be independent of the absolute performance across H
- Be zero if a platform in H is unsupported, and approach zero as the performance of platforms in H approach zero
- Increase if performance increases on any platform in H
- Be directly proportional to the sum of scores across H

Performance

- It runs: {Yes, No}
- It runs efficiently with respect to some baseline

Portability

- Can execute on multiple systems
- Adaptable to varying architectures and platforms

$$\mathcal{P}(a,p;H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall_i \in H \\ 0 & \text{otherwise} \end{cases}$$

Productivity

- SLoC, maintainability, sustainability
- Port/migration/translation

Reproducibility

- For another day...

[1] (definition of) Performance Portability, 2016 Department of Energy Center of Excellence Meeting.

oneMKL open-source interfaces library (OSI)

oneAPI Open-Source Math Library Interface

1st Mariia Kraiiuk
Intel Corporation
United States of America
maria.kraynyuk@intel.com

2nd Mehdi Goli
Codeplay Software Ltd
United Kingdom
mehdi.goli@codeplay.com

3rd Vincent R. Pascuzzi
Lawrence Berkeley National Laboratory
United States of America
vrpascuzzi@lbl.gov

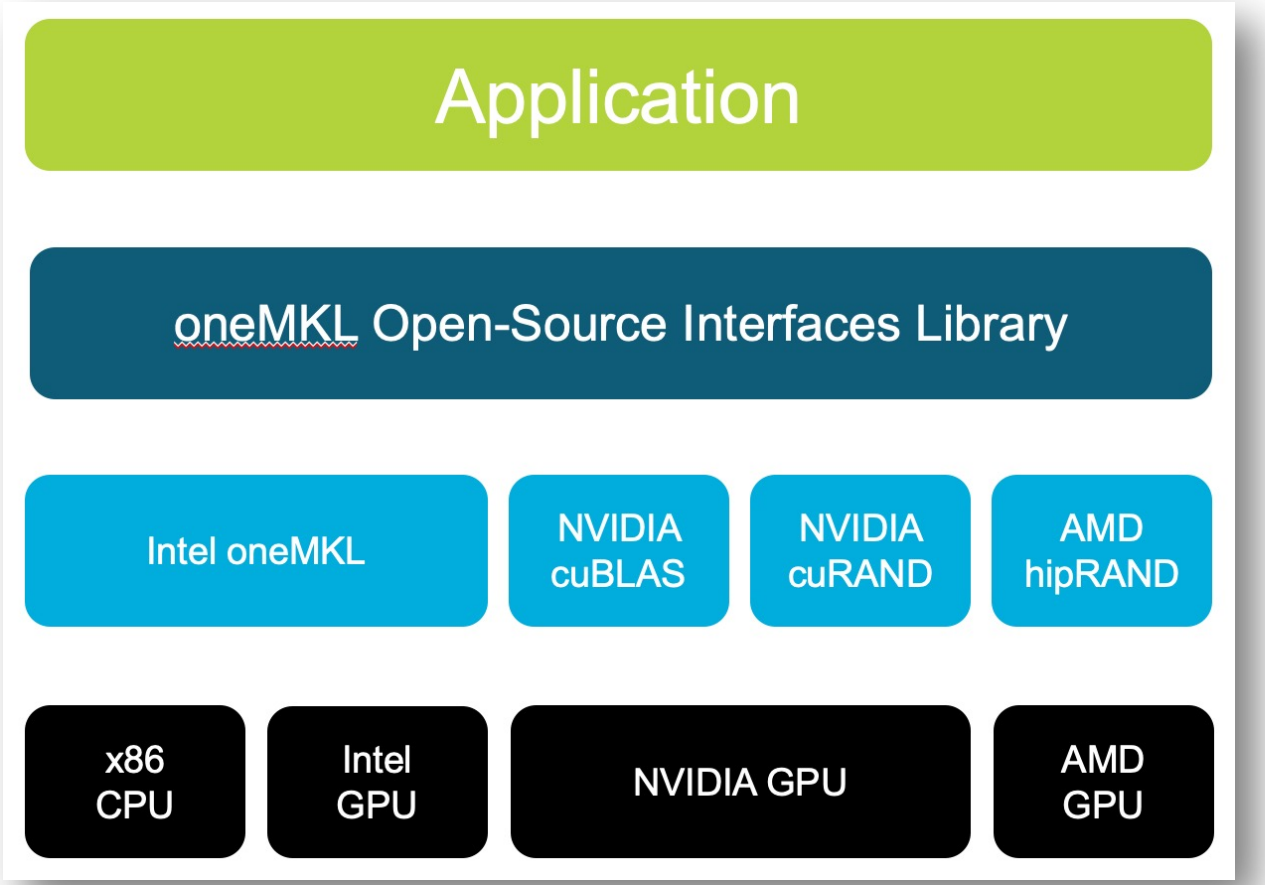
oneAPI

Developed using SYCL programming model
- Part of the oneAPI initiative

Linear algebra and random number generation (RNG) functionality
- NETLIB LAPACK
- Intel oneMKL*

Community-driven
- Technical Advisory Board members provide feedback to the overall oneAPI specification

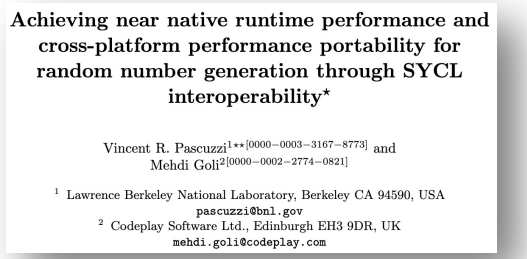
Largely relies on interoperability
- "Plug-in" algorithms optimized for a given architecture



oneapi-src/oneMKL (Github).
* Note the difference between oneMKL OSI and Intel oneMKL.

Brookhaven National Laboratory

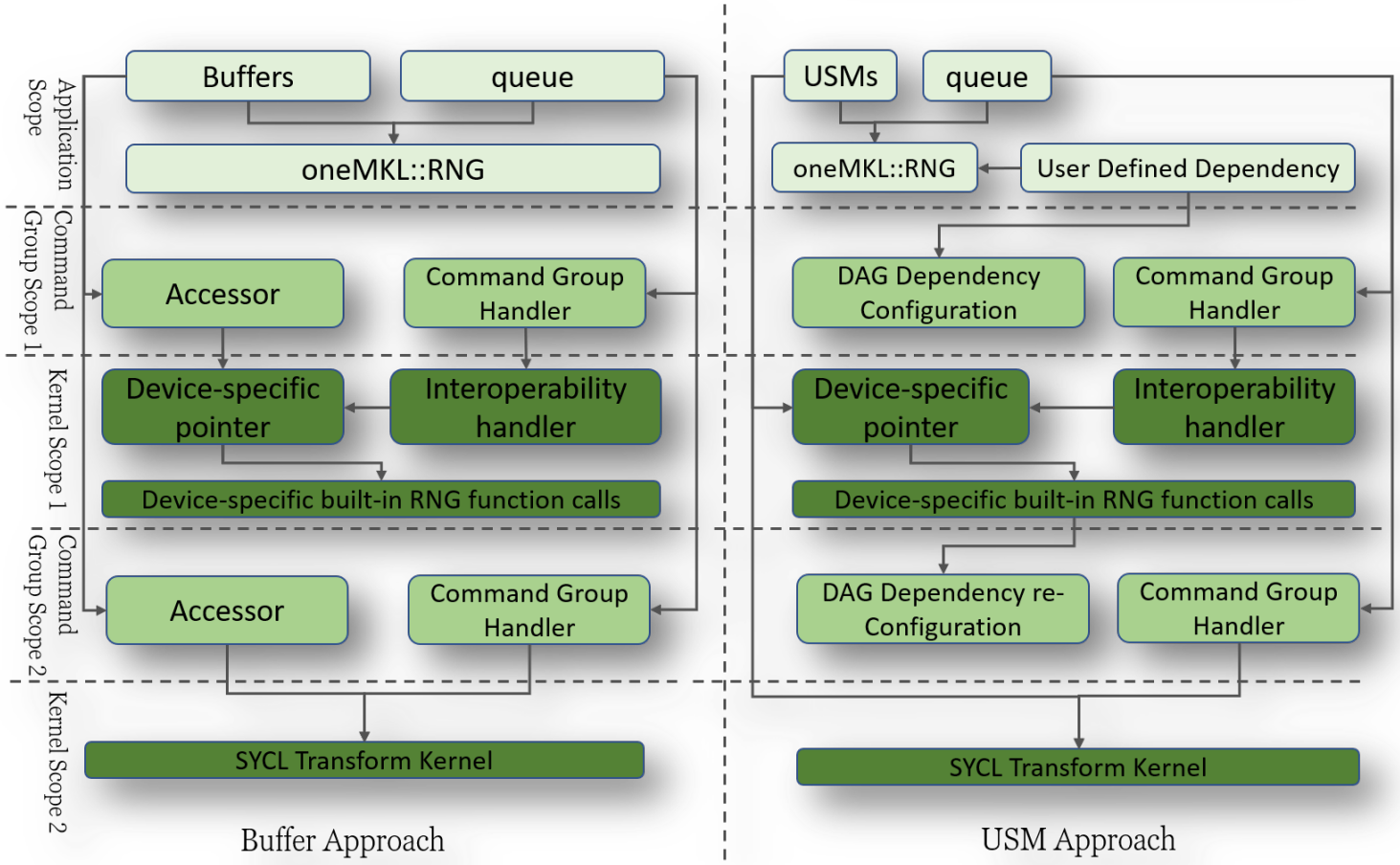
Integrating CUDA and HIP


Achieving near native runtime performance and cross-platform performance portability for random number generation through SYCL interoperability*

Vincent R. Pascuzzi[1**][0000−0003−3167−8773] and Mehdi Goli[2][0000−0002−2774−0821]

[1] Lawrence Berkeley National Laboratory, Berkeley CA 94590, USA
pascuzzi@bnl.gov
[2] Codeplay Software Ltd., Edinburgh EH3 9DR, UK
mehdi.goli@codeplay.com

- Did not have resources to develop a new library
  - Instead, utilize existing highly-optimized libraries
- Required SYCL 2020 features, *e.g.*, `std::atomic_ref`, interoperability, ...
  - intel/llvm
  - illuhad/hipSYCL
- oneMKL OSI does not provide handle to support resource allocation or kernel ordering
  - Explicit synchronization between streams/queues to ensure order
  - Global vs. per-`queue` contexts
- Host and device APIs
  - oneMKL support for host (`curand.h`)

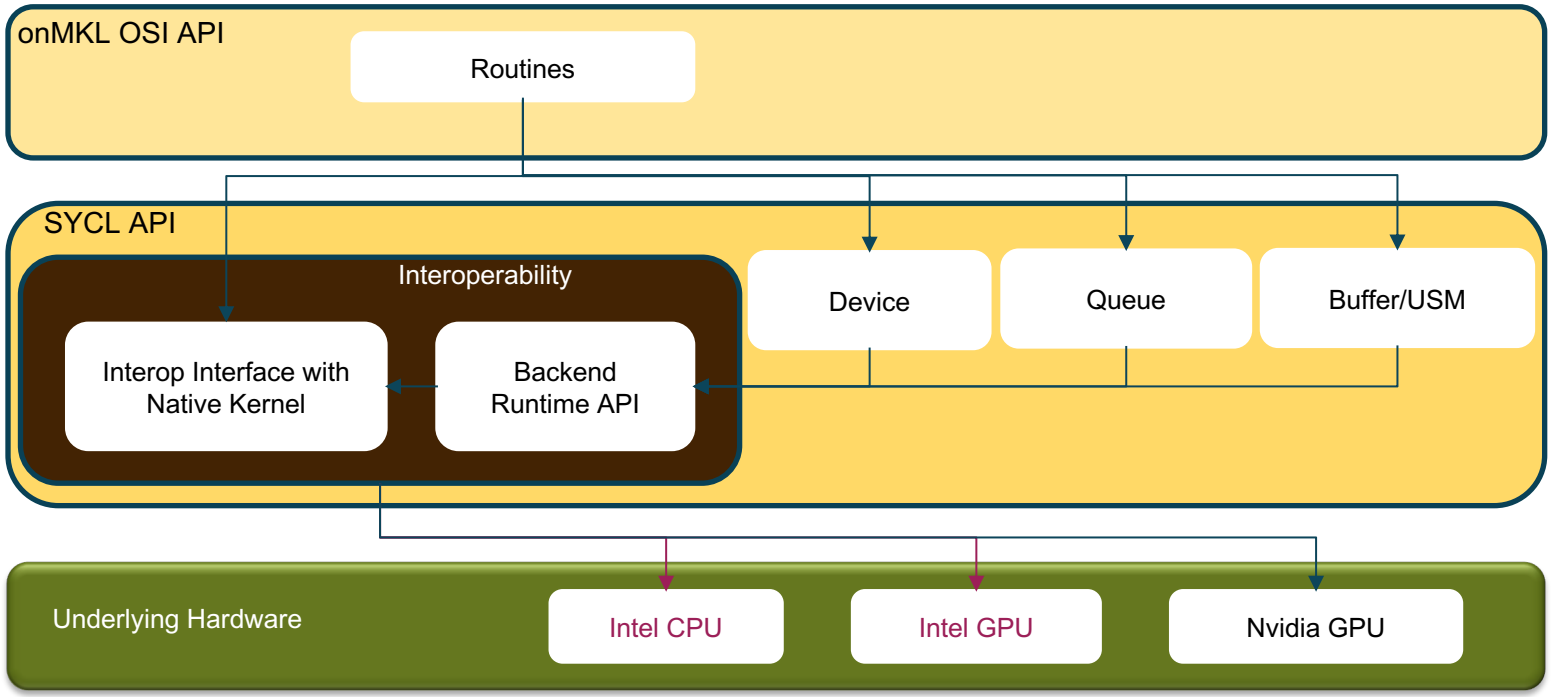


CUDA and HIP routines for oneMKL OSI

User Controls
- Device selection
- Context
- Queue
- Buffer/USM

oneMKL API
- Blas routines
- RNG Routines
- cuBLAS handle*

SYCL API Provides
- Data flow dependency graph
- Kernel dispatch
- Mappings
  - device selection -> CUDA/HIP device
  - Context -> CUDA/HIP Context
  - Queue -> CUDA/HIP Stream
  - Buffer/USM -> CUDA/HIP device pointer
  - Kernel dispatch -> CUDA/HIP launch kernel



Enabling CUDA and HIP backends

Scope Handler
- Map cuBLAS handle to the queue context
- Is bound to per thread SYCL-queue
- Automatic release of the handle upon the queue destruction

Application Scope
- User call to oneMKL
- Command group Scope
- Context/Memory extraction

Kernel Scope
- Call CuBLAS function

```
// Scope Handler
CublasScopedContextHandler(sycl::queue queue,
                           sycl::interop_handler &ih)
    placedContext_ = queue.get_context();
    auto device = queue.get_device();
    auto desired =
        sycl::get_native<sycl::backend::cuda>(placedContext_);
    CUresult err;
    CUDA_ERROR_FUNC(cuCtxGetCurrent, err, &original_);
    if (original_ != desired) {
        CUDA_ERROR_FUNC(cuCtxSetCurrent, err, desired);
        needToRecover_ = !(original_ == nullptr);
    }
}
```

```
//User Application Scope
void main(){
....
auto dev = sycl::device{sycl::gpu_selector()};
queue main_queue(dev, exception_handler);
buffer<float, 1> A_buffer(A.data(), range<1>(A.size()));
buffer<float, 1> B_buffer(B.data(), range<1>(B.size()));
buffer<float, 1> C_buffer(C.data(), range<1>(C.size()));
oneapi::mkl::blas::column_major::gemm(main_queue, transa, transb,
                                      m, n, k, alpha, A_buffer, lda,
                                      B_buffer, ldb, beta, C_buffer, ldc);
....
}
```

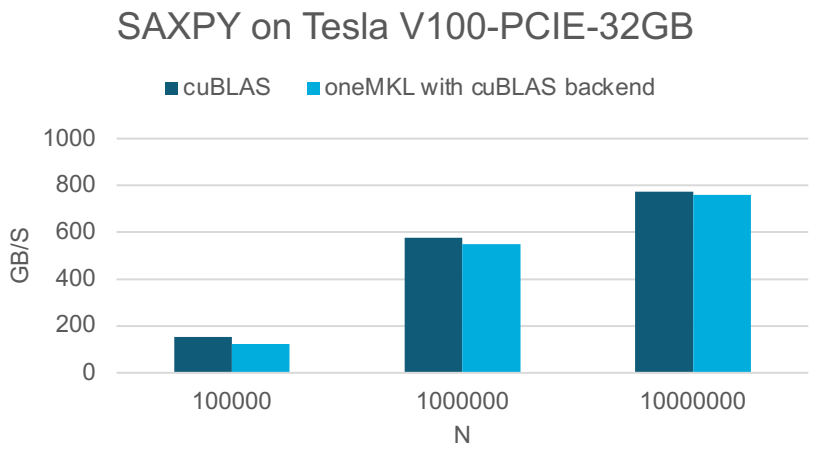
```
template <typename Func, typename T>
inline void gemm(Func func, cl::sycl::queue &queue, ...) {
    // command group scope
    queue.submit([&](cl::sycl::handler &cgh) {
        auto a_acc = a.template get_access<cl::sycl::access::mode::read>(cgh);
        auto b_acc = b.template get_access<cl::sycl::access::mode::read>(cgh);
        auto c_acc = c.template get_access<cl::sycl::access::mode::read_write>(cgh);
        // Kernel scope
        onemkl_cublas_host_task(cgh, queue, [=](CublasScopedContextHandler &sc) {
            auto handle = sc.get_handle(queue);
            auto a_ = sc.get_mem<T*>(a_acc);
            auto b_ = sc.get_mem<T*>(b_acc);
            auto c_ = sc.get_mem<T*>(c_acc);
            cublasStatus_t err;
            CUBLAS_ERROR_FUNC(func, err, handle, get_cublas_operation(transa),
                              get_cublas_operation(transb), m, n, k, (T*)&alpha, a_, lda,
                              b_, ldb, (T*)&beta, c_, ldc);
        });
    });
}
```

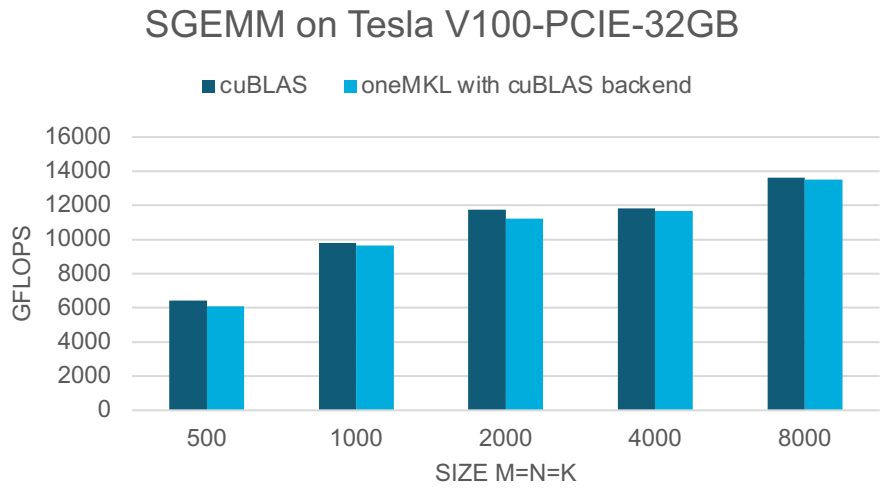
cuBLAS performance evaluation

Compute-bound
BLAS GEMM

$$\cong \leq \%5$$

$$PP = 9.77\ e^{-1}$$

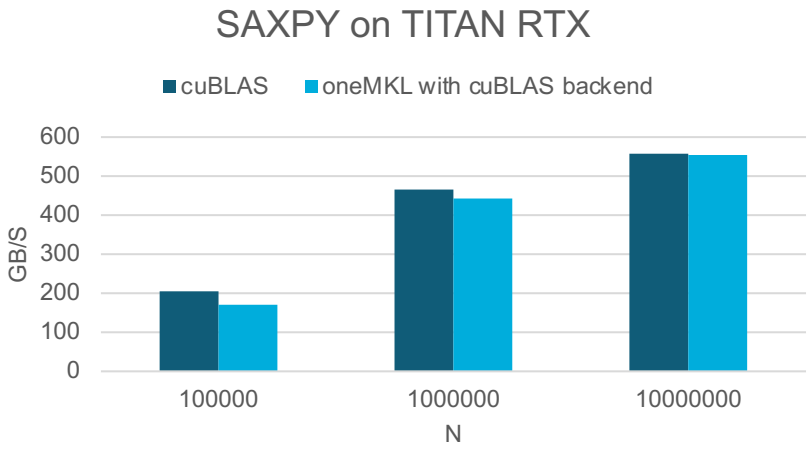


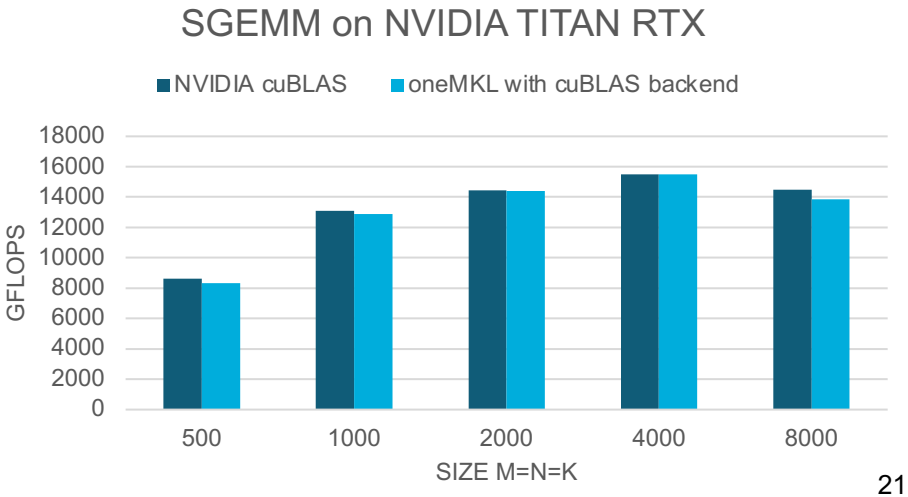


Memory-bound
BLAS AXPY

$$\cong \leq \%5$$

$$PP = 9.77\ e^{-1}$$





RNG algorithms and kernels

oneMKL OSI implements Philox- and MRG-based algorithms
- 36 common high-level generate function templates (PImpl), 18 buffer and 18 USM
- Specify distribution and properties, and output types

`{cu,hip}RAND` have no concept of range, and distributions are coded into specific functions
- SYCL kernels written to address range transformations
- Distribution template parameter used to call correction native generate function

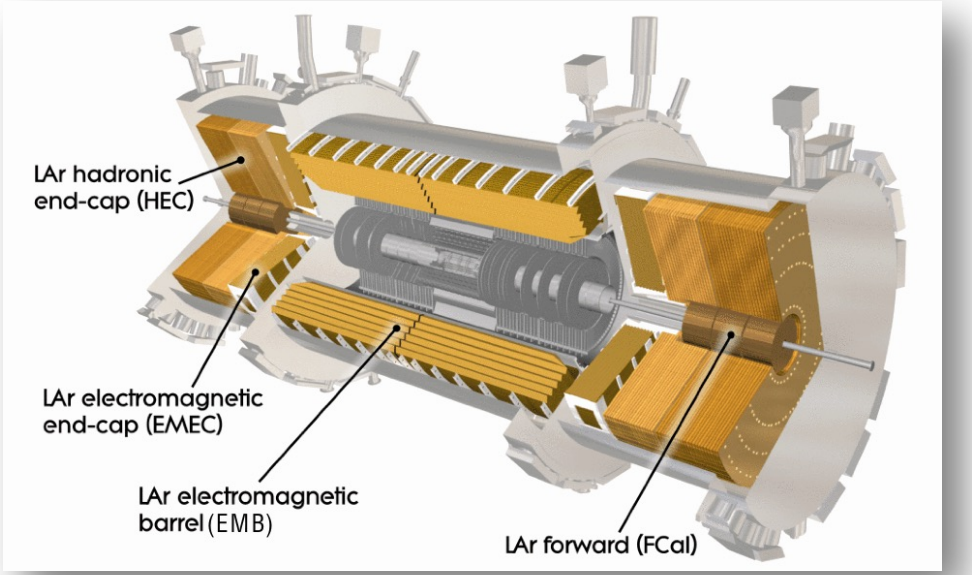
ICDF not supported by `{cu,hip}RAND` pseudorandom generators
- 20/36 `generate` functions supported in our work

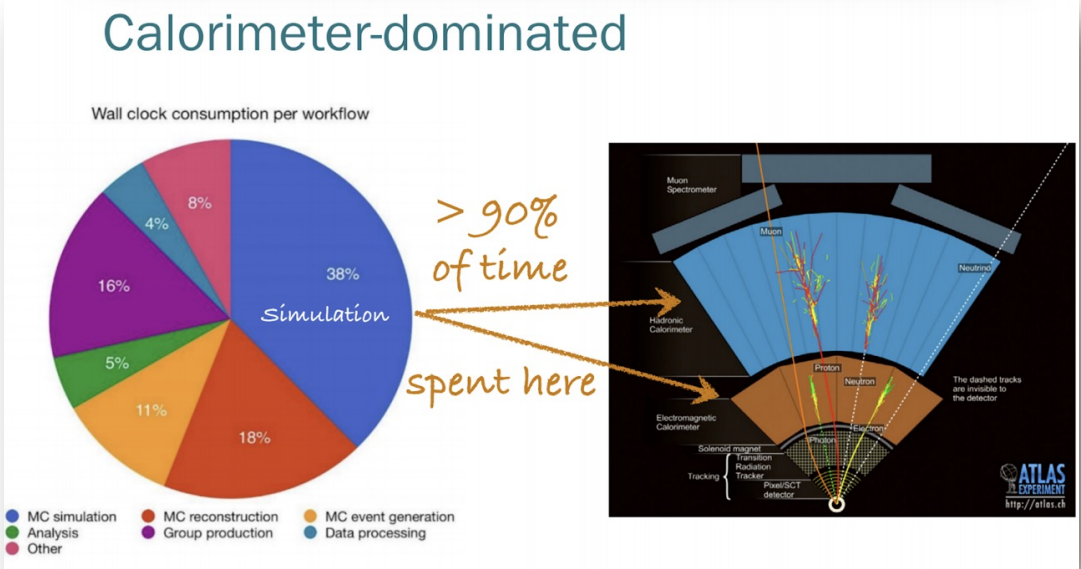
```
virtual inline void generate(
  const oneapi::mkl::rng::uniform<float, uniform_method::standard>& distr,
  std::int64_t n, cl::sycl::buffer<float, 1>& r) override {
    queue_.submit([&](cl::sycl::handler& cgh) {
      auto acc = r.get_access<cl::sycl::access::mode::read_write>(cgh);
      cgh.codeplay_host_task([=](cl::sycl::interop_handle ih) {
      auto r_ptr = reinterpret_cast<float*>(
        ih.get_native_mem<cl::sycl::backend::cuda>(acc));
      curandStatus_t status;
      CURAND_CALL(curandGenerateUniform, status, engine_, r_ptr, n);
      cudaError_t err;
      CUDA_CALL(cudaDeviceSynchronize, err);
    });
  });
  range_transform_fp<float>(queue_, distr.a(), distr.b(), n, r);
}
```

```
template <typename T>
static inline void range_transform_fp(cl::sycl::queue& queue, T a, T b,
                                      std::int64_t n,
                                      cl::sycl::buffer<T, 1>& r) {
  queue.submit([&](cl::sycl::handler& cgh) {
    auto acc =
      r.template get_access<cl::sycl::access::mode::read_write>(cgh);
    cgh.parallel_for(cl::sycl::range<1>(n), [=](cl::sycl::id<1> id) {
      acc[id] = acc[id] * (b - a) + a;
    });
  });
}
```

Benchmark applications

1. Mini-app used to stress hardware for different backends
   - Generates 1-10^8 random numbers
   - Common code to ensure consistent runtime behavior among backends
2. ATLAS parameterized calorimeter simulation software (FastCaloSim)
   - 190k ‘sensors’, ~10 MB geometry
   - Inputs total ~GB, loaded at runtime
   - Single-particle simulations require 10^2-10^7 random numbers per event





Performance evaluation

Numerous definitions of *performance portability*

- Adopt from Pennycook *et. al.*[1]

Introduce *application efficiency* metric, *VAVS*

- Ratio between the time-to-solution (*TTS*) of portable implementation to the native
- Useful for identifying runtime overheads introduced by portability layers

Execute codes on a variety of machines with various software stacks

- GNU compiler for ISO C++
- hipSYCL targeting AMD GPU
- intel/llvm (DPC++) targeting SYCL on x86 and CUDA

$$\mathcal{P}(a,p;H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall_i \in H \\ 0 & \text{otherwise} \end{cases}$$

$$\text{VAVS} \equiv \frac{TTS_{\text{portable}}}{TTS_{\text{native}}}$$

Platform	Driver Version	OS and Kernel	Compiler	RNG Library
AMD Rome 7742	-	OpenSUSE 15.0 4.12	GNU 8.2.0 DPC++	CLHEP 2.3.4.6 oneMKL
Intel Core i7-1080H	-	Ubuntu 20.04 5.8.18	GNU 8.4.0 DPC++	CLHEP 2.3.4.6 oneMKL
Intel UHD Graphics	21.11.19310	Ubuntu 20.04 5.8.18	DPC++	oneMKL
Radeon RX Vega 56	20.50	CentOS 7 3.10.0	HIP 4.0.0 hipSYCL 0.9.0	hipRAND 4.0.0 oneMKL
NVIDIA A100	450.102.04	OpenSUSE 15.0 4.12	CUDA 10.2.89 DPC++	cuRAND 10.2.89 oneMKL

[1] Pennycook *et. al.* (2019) doi:10.1016/j.Future.2017.08.007.

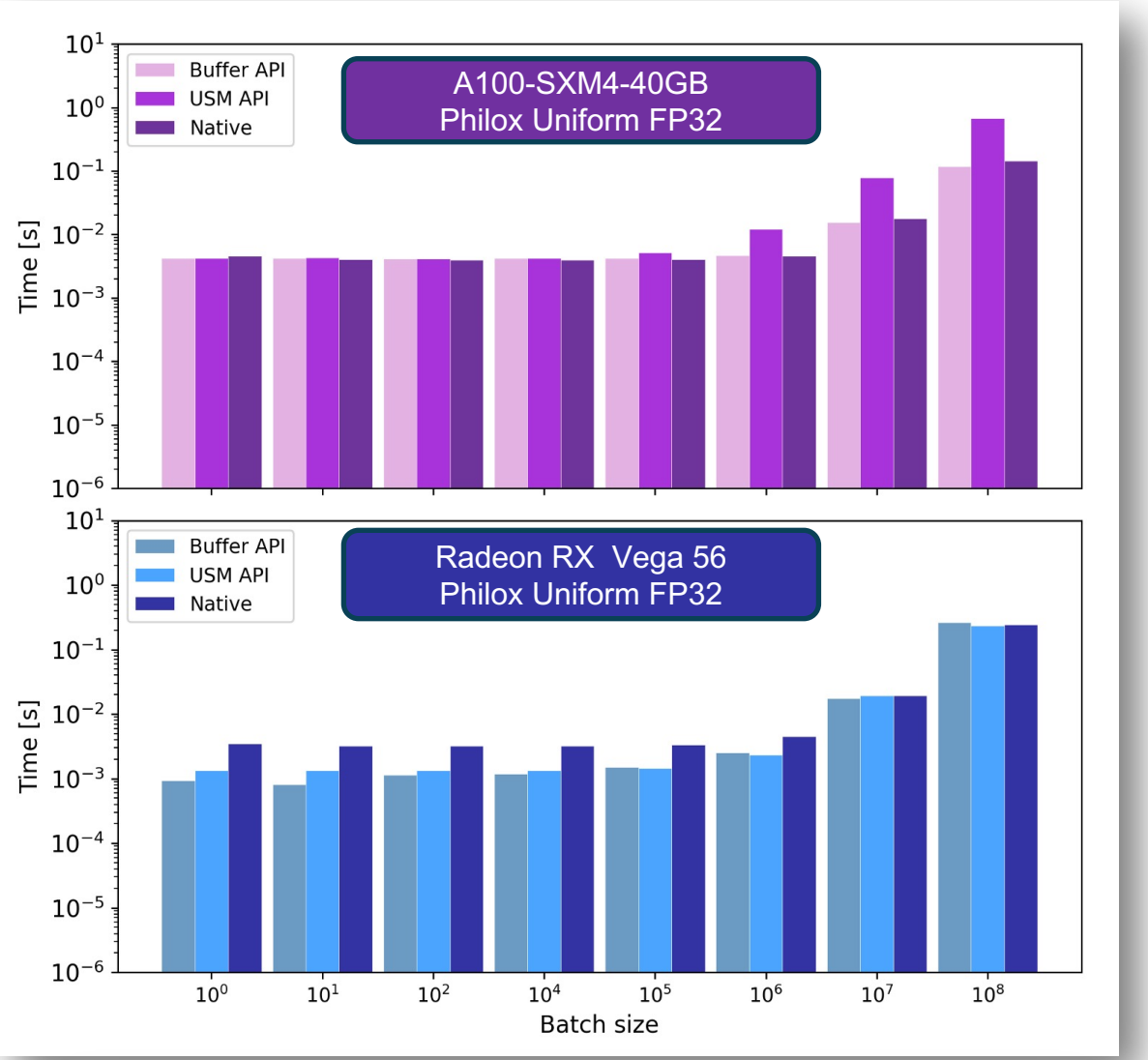
RNG burner

Time-to-solution (TTS) using clocks shown for three kernels: `seed`, `generate` and `transform`
- SYCL oneMKL OSI buffer and USM APIs
- Native `{cu,hip}RAND`

Benchmark ran 100 iterations (none discarded) for each batch size

Increased TTS of USM on A100 due to explicit synchronization

Reduced TTS of SYCL on AMD platform
- Optimizations within hipRAND runtime system for ROCm backend
- Callbacks introduce notable latencies in small kernels
- Nearly callback-free hipSYCL runtime visible for batch sizes < 10^7



H	$\mathcal{P}$ buffer	$\mathcal{P}$ USM	$\mathcal{P}$ Mean (buffer+USM)
{Vega 56, A100}	1.070	0.393	0.575
{Vega 56}	0.974	1.076	1.022
{A100}	1.186	0.240	0.400

Brookhaven National Laboratory

Kernel profiling

Per-kernel TTS and relative occupancy for A100

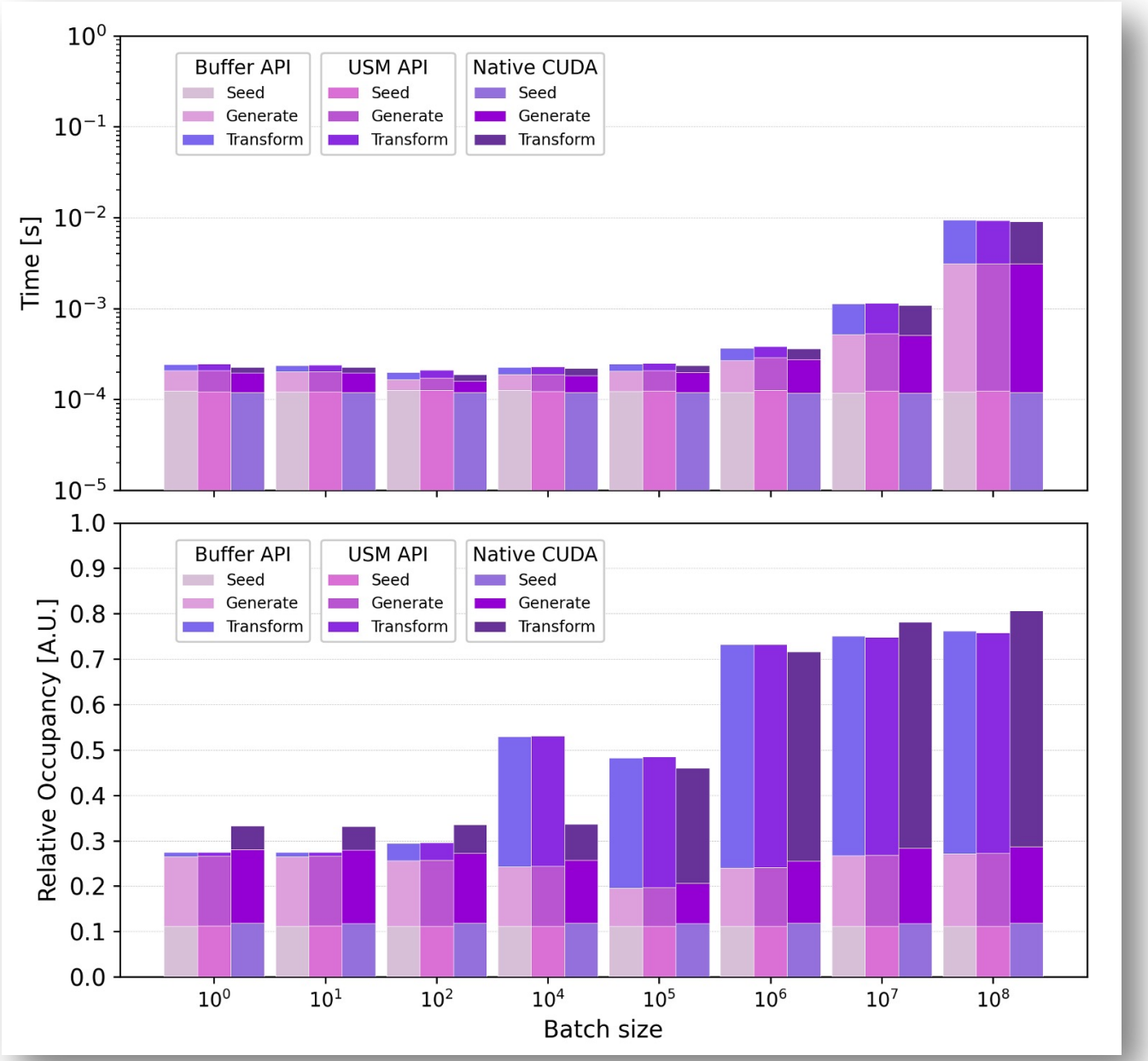
- Data collected using Nvidia Nsight Compute 2020.2.1

Ten iterations for each backend/API

Both cuRAND kernels (seed and generate) are identical between oneMKL OSI and native

Large increase in relative occupancy between 10^2 and 10^4 for cuRAND kernels

- SYCL runtime system optimizes required block size and threads-per-block when not specified
- Native application fixed block size at 256 and SYCL runtime chose 1024 (no performance gains)



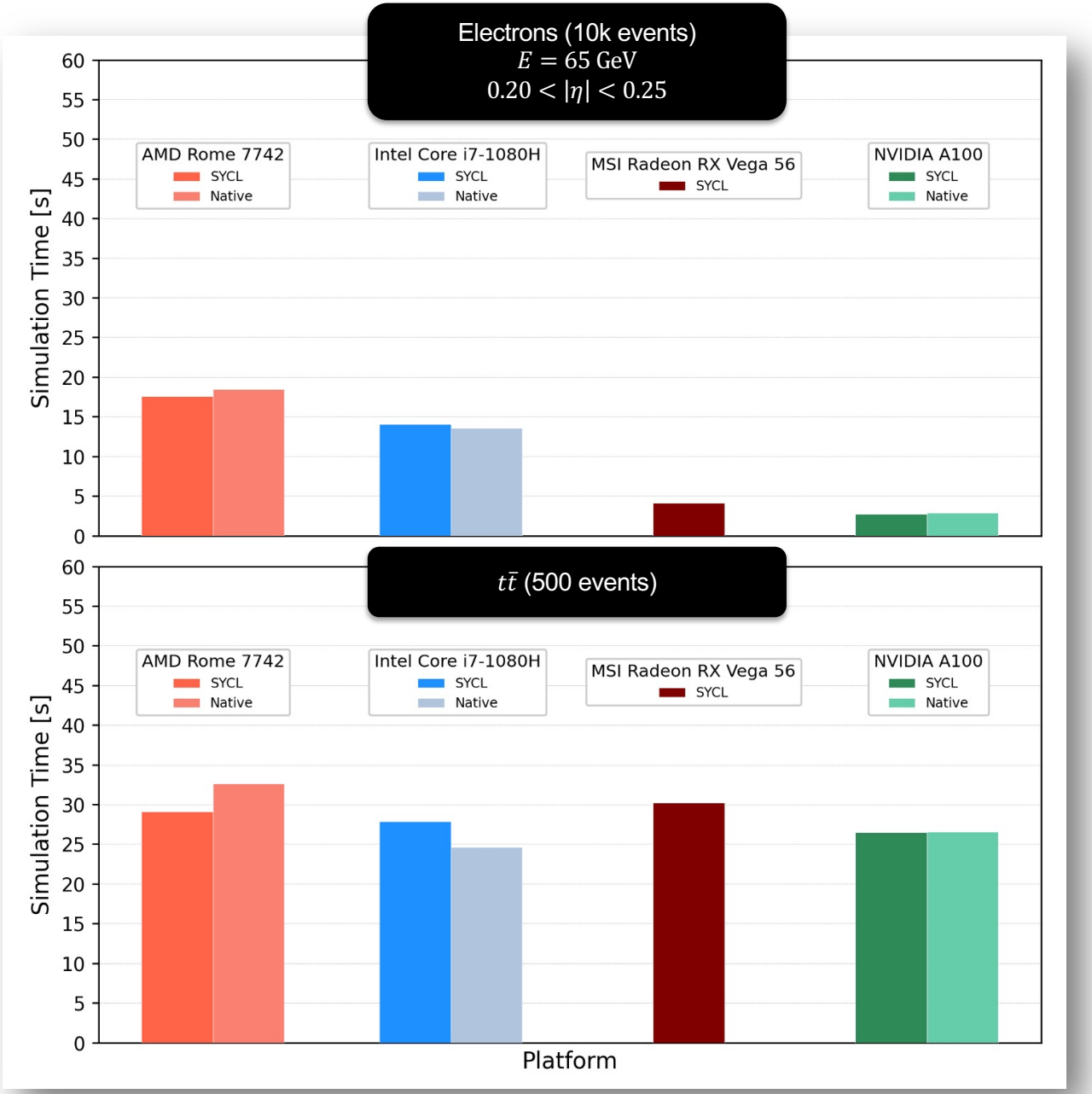
FastCaloSim

Demonstrate performance portability using real-world application (re-)written using SYCL programming model

Ten ‘runs’ of single-electron and top quark pair production simulations

- AMD CPU targeted using `host_device` (TBB, no OpenCL backend)
- Intel CPU targeted using `cpu_device` (OpenCL backend)
- ~80% TTS reduction for single electrons when executed using GPU offload
- Top quark simulations achieve no gains on GPUs due to lack of inter-event parallelism and runtime data movement host → device loading parameterizations

The same source runs across four different platforms with excellent performance



Summary

- High-performance computing is driving science and discovery
  - More computational resource to tackle bigger problems
  - Does not come “for free”
- Growing number of platforms
  - Not as “easy” as writing parallel C++/fortan/... code anymore
  - Some use-cases permit targeting and highly optimizing for one
  - Some use-cases require utilizing as many as possible
- Growing number of programming models
  - OpenMP seems to be best for intra-node parallelization but still need to write kernels
  - Kokkos’ maturity is an advantage; SYCL is lower level (Kokkos now has a SYCL “backend”)
- Interoperability is a wonderful idea
  - Leverage existing highly-optimized, vendor-provided libraries within a SYCL application
  - Main target is to improve (*e.g.*, coordination asynchrony with other libraries, command-group dependencies, callback mechanism, *etc.*)
  - Today, only discussed one type (“host task”); others include kernel ingestion, OpenCL translation
- Is performance portability a pipe dream?
  - Not necessarily!
  - While one may not achieve 100% across all platforms, more science can potentially be done using many v. single HPC

Where from here?

- Heuristic methods for choosing optimal backend
  - Given the problem size, GPUs are not always the best solution
- Purely SYCL-based math libraries (JLSE-approved project)
  - Every application will require significant boiler-plate code when using interoperability
  - Backwards compatibility (e.g., CUDA APIs change often ⟹ maintaining interoperable implementations)
  - New hardware architectures ⟹ source polluted with macros – e.g., `#ifdef` – to deal with many backends
  - What if no vendor-specific libraries are available, e.g., ARM math?
  - Can provide template metaprogramming for users to optimize for their target hardware (we don't care about what they want to run on, only that it does run)
  - Auto-tuning mechanisms: query available devices and configure/tune implementation for automatically choosing [near-]optimal parameters for a given device
- QPU backend
  - Integrate Qiskit, Cirq, DM-Sim ... simulators as additional PIs
    ⟹ concurrent and asynchronous classical-quantum computing

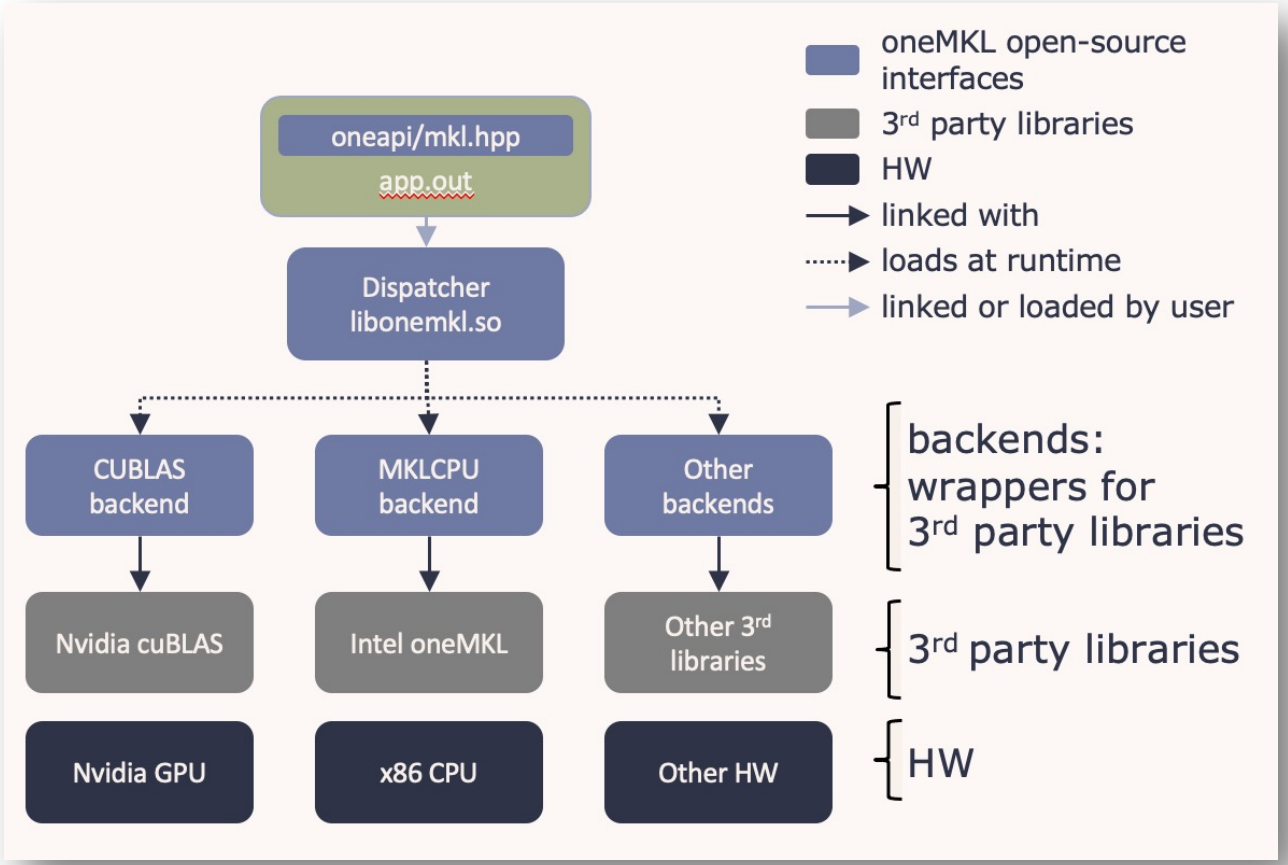
Backup

oneMKL OSI architecture

```
#include "oneapi/mkl.hpp"

int main() {
...
    cpu_device = sycl::device(sycl::cpu_selector());
    gpu_device = sycl::device(sycl::gpu_selector());
    sycl::queue cpu_queue(cpu_device);
    sycl::queue gpu_queue(gpu_device);
...
    oneapi::mkl::blas::column_major::gemm(cpu_queue, ...);
    oneapi::mkl::blas::column_major::gemm(gpu_queue, ...);
...
}
```

```
$ clang++ -fsycl -c -I$ONEMKL/include app.cpp
$ clang+ +-fsycl  app.o -L$ONEMKL/lib -lonemkl -o app.out
$ ./app.out
```



DPC++ runtime Plugin Interface (PI)