SYCL and oneMKL

Hugh Bird
Rafal Bielski
Duncan McBain
Pablo Lopez Ramos

Argonne – 20th June 2024

codeplay

Enabling AI & HPC To Be Open, Safe & Accessible To All

Established 2002 in **Edinburgh, Scotland**.

Grown successfully to around 100 employees.

In 2022, we became a **wholly owned subsidiary** of Intel.

oneAPI

Committed to expanding the **open ecosystem** for heterogeneous computing.

Through our involvement in oneAPI and SYCL governance, we help to **maintain and develop** open standards.

Developing at the forefront of **cutting-edge research**.

Currently involved in two research projects - **SYCLOPS** and **AERO**, both funded by the Horizon Europe Project.

Today's event

We will show how to achieve **portability** of mathematical computations **across GPU vendors** using **oneMKL**

60 min presentation + 30 min hands-on session

Please ask questions at the end of each section (agenda in the next slide)

Hugh Bird
Staff Software Engineer @ Codeplay
member of Performance Libraries Team and oneMKL contributor

Duncan McBain
Senior Software Engineer @ Codeplay
product owner of the oneAPI Support Team

Pablo Lopez Ramos
Software Engineering Contractor @ Codeplay
member of the oneAPI Support Team

Rafal Bielski
Senior Software Engineer @ Codeplay
supports SYCL users in achieving the best performance

Agenda

- A quick introduction to SYCL
- What is oneMKL
- oneMKL Interface Library
  - What can it do?
  - How do you use it?
  - How does it work?
  - Building it
  - Gotchas
  - Performance
- Workshop
  - Build and run an example with oneMKL

SYCL

A really quick introduction

SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

Open standard provided by the non-profit cross-industry **Khronos Group**

Well-defined **concurrency and memory models** enable more optimisation
and performance opportunities

Single C++ source for all architectures

```cpp
#include <sycl/sycl.hpp>
#include <vector>

int main() {
    constexpr static size_t N{10000};
    std::vector<float> a(N, 1.0f);
    std::vector<float> b(N, 2.0f);
    std::vector<float> c(N, 0.0f);

    sycl::queue q{};
    {
        sycl::buffer buf_a{a};
        sycl::buffer buf_b{b};
        sycl::buffer buf_c{c};
        q.submit([&](sycl::handler& h){
            sycl::accessor acc_a{buf_a, h, sycl::read_write};
            sycl::accessor acc_b{buf_b, h, sycl::read_only};
            sycl::accessor acc_c{buf_c, h, sycl::write_only, sycl::no_init};
            h.parallel_for<class my_kernel>(N, [=](sycl::id<1> id){
                acc_a[id] += acc_b[id];
                acc_c[id] = 2.0f * acc_a[id];
            });
        }).wait();
    }

    for (float x : c) {std::cout << x << " ";}
    std::cout << std::endl;

    return 0;
}
```

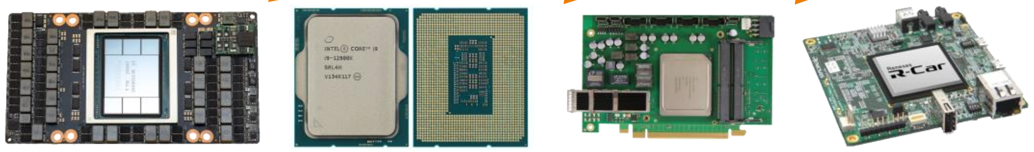
Device management with queues

Memory management with buffers

Submit a work unit to a queue

Execute device code

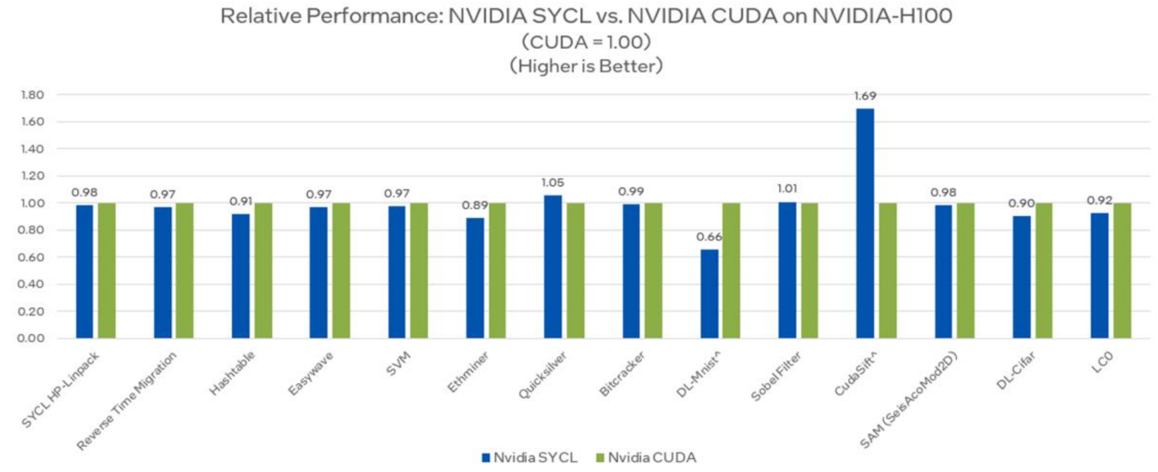
- Standard C++
  - SYCL 2020 based on ISO C++17
- Unlike in other parallel programming APIs, there are:
  - No pragmas or macros
  - No special attributes
  - No language extensions



GPU CPU FPGA Specialised processors

SYCL performance is comparable to native CUDA/HIP

On NVIDIA GPU – SYCL Provides Comparable Performance to CUDA



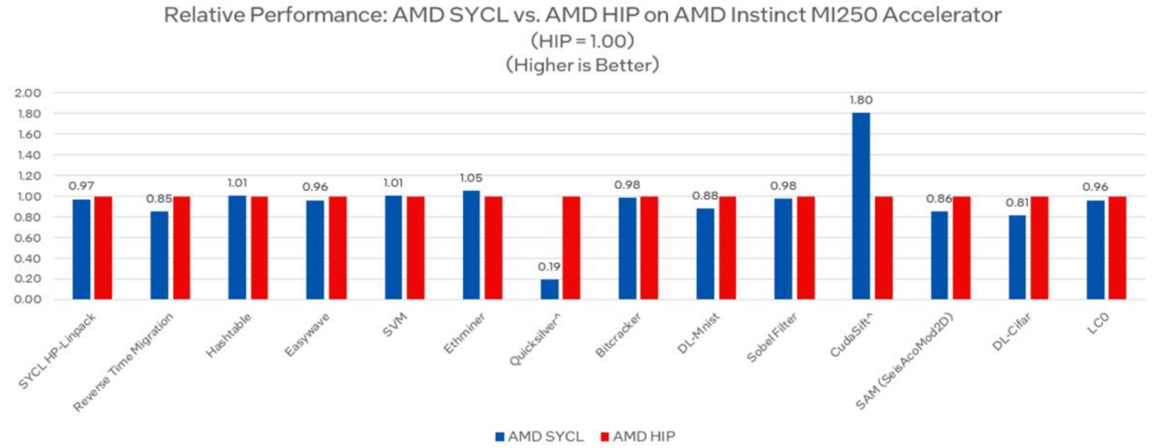
Testing Date: Performance results are based on testing by Intel as of **August 1, 2023** and may not reflect all publicly available updates.

Configuration Details and Workload Setup: Intel® Xeon® Platinum 8360Y CPU @ 2.4GHz, 2 socket, Hyper Thread On, Turbo On, 256GB Hynix DDR4-3200, ucode 0xd000389. GPU: Nvidia H100 PCIe 80GB GPU memory. Software: Velocity Bench benchmark suite branch from 8/1/23. SYCL open source/CLANG 17.0.0, CUDA SDK 12.0 with NVIDIA-NVCC 12.0.76, cuMath 12.0, cuDNN 12.0, Ubuntu 22.04.1. SYCL open source/CLANG compiler switches: -fsycl-targets=nvptx64-nvidia-cuda -Xsycl-target-backend --cuda-gpu-arch=sm_90, NVIDIA NVCC compiler switches: -O3 –gencode arch=compute_90, code=sm_90. Represented workloads with Intel optimizations.

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

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Your costs and results may vary

On AMD GPU – SYCL Provides Comparable Performance to HIP



Testing Date: Performance results are based on testing by Intel as of **August 1, 2023** and may not reflect all publicly available updates.

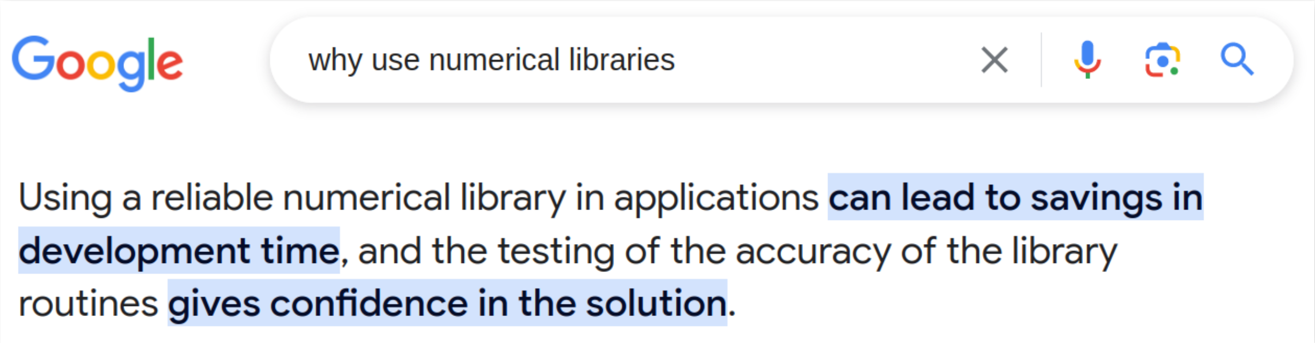
Configuration Details and Workload Setup: AMD EPYC 7313 CPU @ 3.0GHz, 2 socket, AMD Simultaneous Multi-Threading Off, AMD Precision Boost Enabled, 512GB DDR4, ucode 0xa001144. GPU: AMD Instinct MI250 OAM, 128GB GPU memory. Software: Velocity Bench benchmark suite branch from 8/1/23, SYCL open source/CLANG 17.0.0, AMD RoCm 5.6.0 with roc-5.6.31061, hipSolver 5.6.0, rocBLAS 5.6.0, Ubuntu 20.04.4. SYCL open source/CLANG compiler switches: -O3 -fsycl -fsycl-targets=amdgcn-amd-amdhsa -Xsycl-target-backend --offload-arch=gfx90a, AMD-ROCm compiler switches: -O3. Represented workloads with Intel optimizations.

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

Performance varies by use, configuration, and other factors. Learn more at www.Intel.com/PerformanceIndex. Your costs and results may vary

See our blog post for more details on these benchmark results

But should you even write your own kernels?



The open-standard **oneAPI ecosystem** centred around SYCL comes to help!

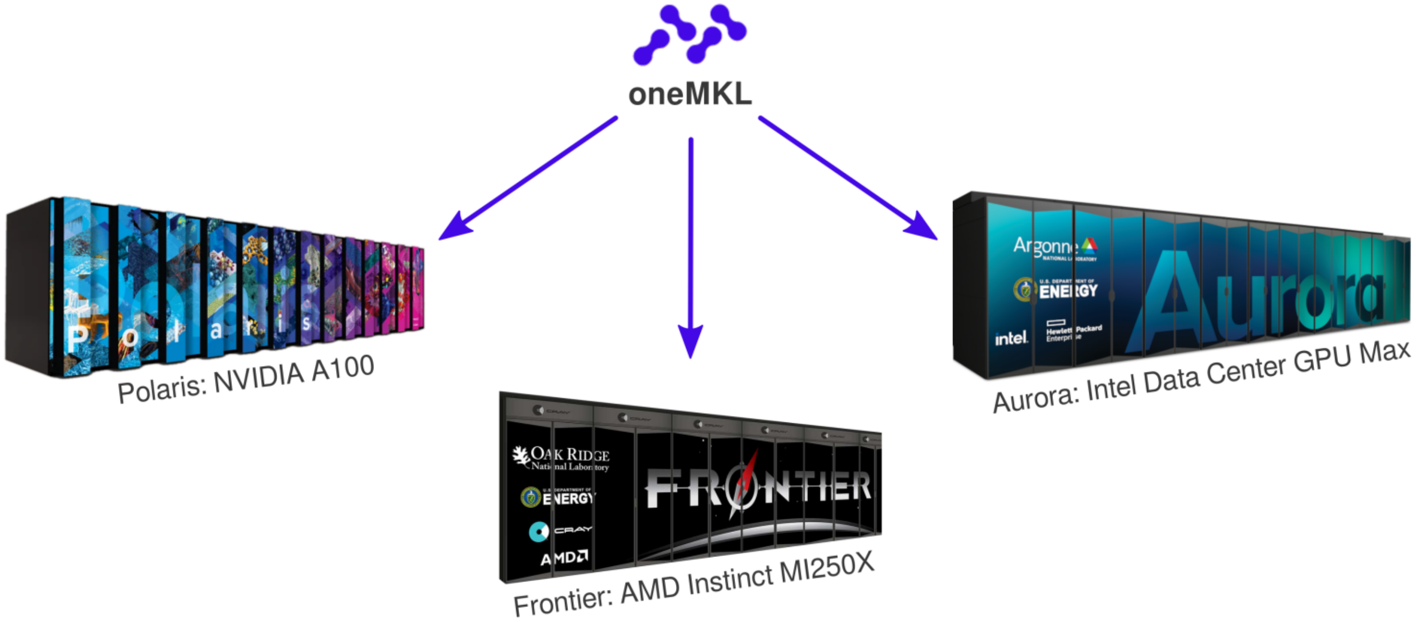
You might be familiar with some of the vendor-specific GPU numerical libraries

- Intel: *Math Kernels Library*
- NVIDIA: *cuBLAS, cuSOLVER, cuRAND, cuFFT*
- AMD: *rocBLAS, rocSOLVER, rocRAND, rocFFT*

Imagine being able to use all of them with single source code → **oneMKL**

oneMKL provides performance and portability

write single source code



run everywhere

The oneMKLs

one API, two implementations, and three things

Pieces of the puzzle

- oneMKL consists of three parts:
- The oneMKL specification - part of the oneAPI specification
- An open-source library implementing the MKL API - oneMKL Interfaces
- The original Intel optimised maths routines - for clarity, *Intel® MKL*

oneAPI and oneMKL

- oneAPI has a *specification* describing how its components should behave
- oneMKL is a *component* of oneAPI covering fundamental mathematical routines for HPC, engineering, science etc.
- The UXL (Unified Acceleration) Foundation develops these specifications
- The specification is open-source, available on GitHub

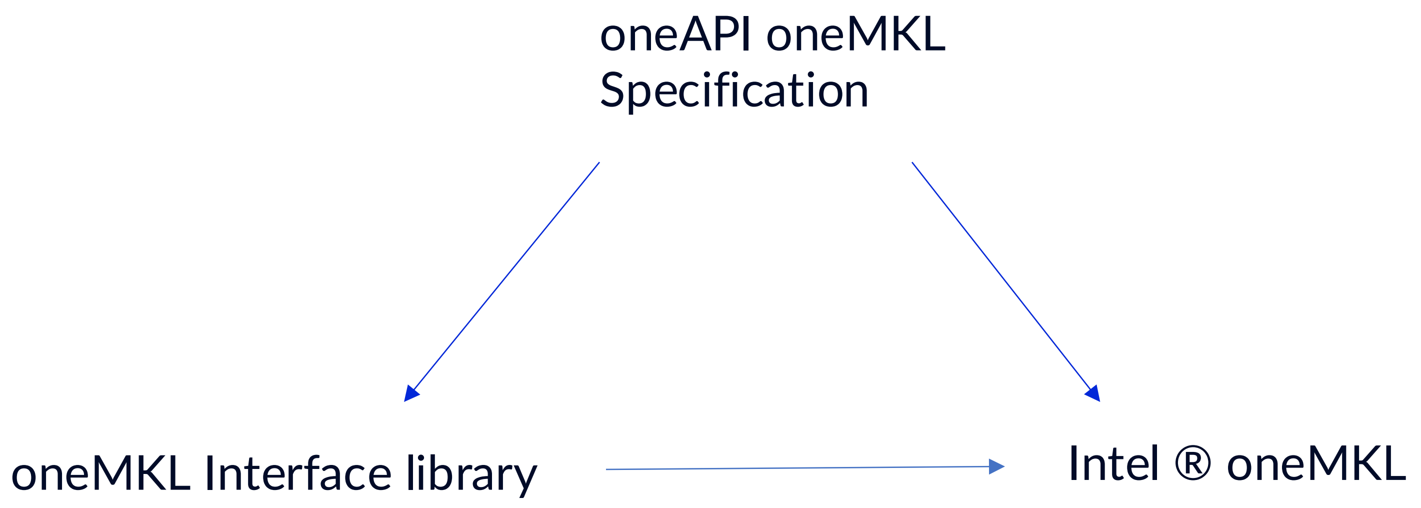
oneMKL Interface Library

- The topic of this presentation!
- Implements the oneMKL specification, dispatching to other libraries under-the-hood
  - Intel (Intel's MKL)
  - Nvidia (cuBLAS, cuRAND, cuFFT etc.)
  - AMD (rocBLAS, rocFFT etc.)
  - And SYCL-supported devices ("generic" SYCL code)
- DPC++ and AdaptiveCpp
- (AdaptiveCpp support varies by backend but is being worked on)

Intel® oneMKL

- We'll refer to this as just "MKL" to reduce the confusion
- Intel CPU and Intel GPU
- Mostly conforms to the oneMKL spec except for some legacy reasons
- Available as part of the Intel oneAPI base toolkit

The oneMKLs

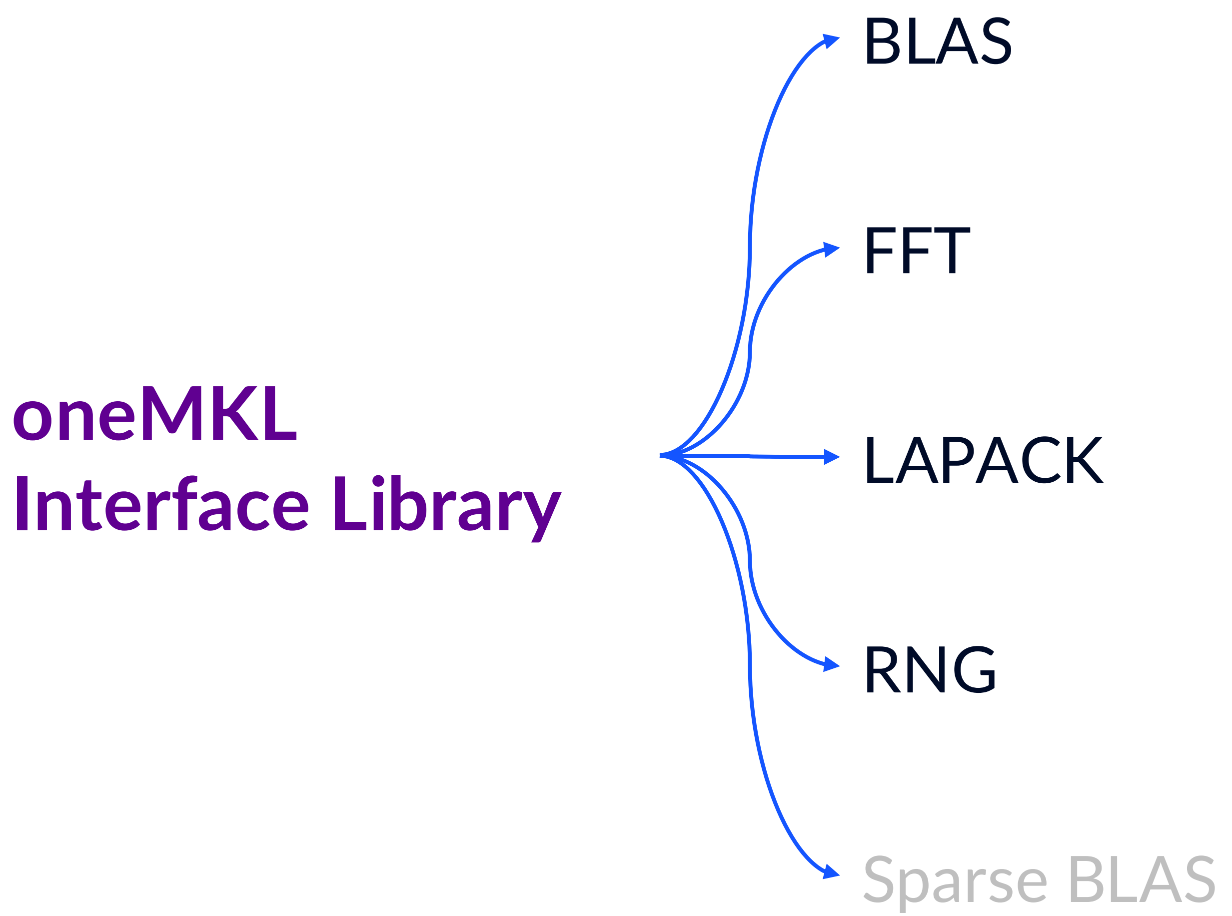


- In short, oneMKL interfaces and Intel® MKL both implement the oneMKL specification
- oneMKL interfaces can dispatch to Intel® MKL as well as other vendor libraries
- Intel® MKL is available via the Intel website as part of the oneAPI base toolkit
- oneMKL interfaces are available on GitHub

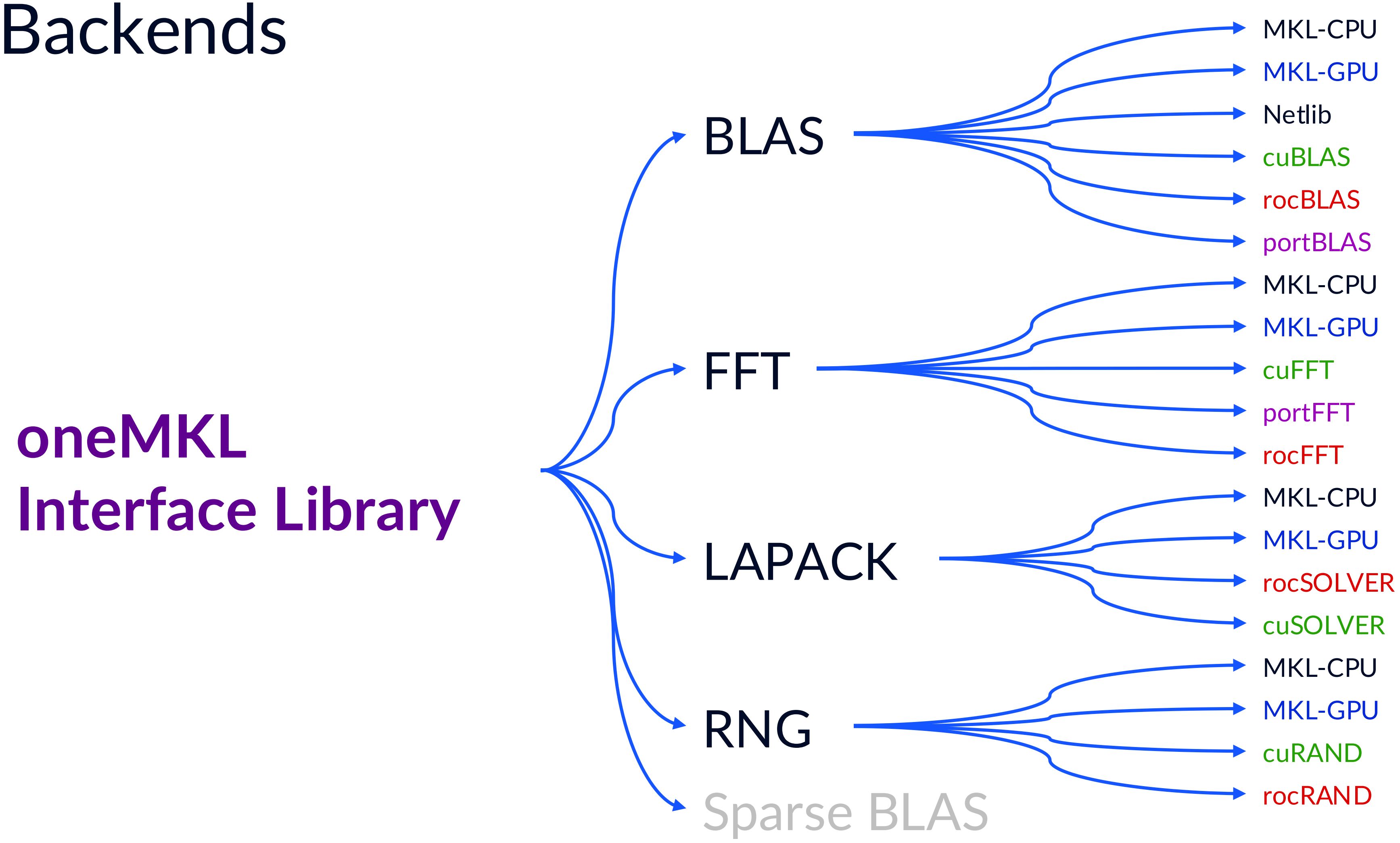
oneMKL: capabilities

Domains

- BLAS
- LAPACK
- DFT
- RNG
- Sparse BLAS



Backends


oneMKL
Interface Library
BLAS
MKL-CPU
MKL-GPU
Netlib
cuBLAS
rocBLAS
portBLAS
FFT
MKL-CPU
MKL-GPU
cuFFT
portFFT
rocFFT
LAPACK
MKL-CPU
MKL-GPU
rocSOLVER
cuSOLVER
RNG
MKL-CPU
MKL-GPU
cuRAND
rocRAND
Sparse BLAS

Runtime dispatch

```
// Get a sycl::queue on any vendor's device.
sycl::queue myQueue;


// oneMKL handles the dispatch to the
// correct backend library.
oneapi::mkl::<fn>(myQueue, …);
```

- oneMKL can build with support for multiple vendors at once.
- oneMKL can automatically dispatch to the correct backend library.
- Backends are lazily `dlopened`

Static dispatch

```
using oneapi::mkl;

// Choose a particular device
sycl::queue intelQueue(myIntelGPUSelector);

// Use a selector that uses a particular /
// oneMKL backend.
backend_selector<backend::mklgpu>
        mklgpuSelector{intelGpuQueue};

// Call a backend function directly.
oneapi::mkl::<fn>(mklgpuSelector, …);
```

- Avoid overhead of dispatch tables by linking directly against backend libraries.

oneMKL demo

oneMKL: using it

BLAS

```
sycl::queue syclQueue;

// Your data needs to be accessible on the GPU.
auto dev_A = sycl::malloc_device<float>(sizeA, syclQueue);
// … allocate memory, give it relevant values.

// Its like the BLAS API, but taking a queue argument. The USM API returns an event.
gemm_done = oneapi::mkl::blas::column_major::gemm(syclQueue, transA, transB, m, n, k, alpha,
                                                  dev_A, ldA, dev_B, ldB, beta, dev_C, ldC);

// Wait for the work to finish.
gemm_done.wait_and_throw();
```

https://github.com/oneapi-src/oneMKL/blob/develop/examples/blas/run_time_dispatching/level3/gemm_usm.cpp

Random number generation

```
using oneapi::mkl;

// A random number generator is linked to a sycl::queue
rng::default_engine engine(syclQueue, seed);
rng::uniform<float> distribution(low, high);

// Use the state we generated earlier.
auto eventOut = rng::generate(distribution, engine, n, deviceMem);

// Wait for the work to finish.
eventOut.wait_and_throw()
```

https://github.com/oneapi-src/oneMKL/blob/develop/examples/rng/run_time_dispatching/uniform_usm.cpp

DFT

```
using oneapi::mkl;

// A descriptor describes the DFT you want…
dft::descriptor<dft::precision::SINGLE, dft::domain::REAL> desc(N);
desc.set_value(dft::config_param::PLACEMENT, dft::config_value::INPLACE);

// Once set, it is committed on for the chosen queue.
desc.commit(syclQueue);

// Compute the DFTs…
auto computeEvent = dft::compute_forward(desc, x_usm);

// Wait for the result.
computeEvent.wait_and_throw();
```

https://github.com/oneapi-src/oneMKL/blob/develop/examples/dft/run_time_dispatching/real_fwd_usm.cpp

LAPACK

```
using oneapi::mkl;
// Some APIs need scratch memory to be pre-allocated.
std::int64_t getrf_scratchpad_size = lapack::getrf_scratchpad_size<float>(syclQueue, m, n, lda);
float* getrf_scratchpad = sycl::malloc_shared<float>(getrf_scratchpad_size, syclQueue);

// … More allocs, etc.
// LU factorization on device
auto getrfDone = lapack::getrf(syclQueue, m, n, devA, lda, dev_ipiv, getrf_scratchpad, getrf_scratchpad_size);
// Use LU factorization to solve system on device. Needs LU factorization to be complete.
auto getrsDone = lapack::getrs(syclQueue, trans, n, nrhs, devA, lda, dev_ipiv,
                               devB, ldb, getrs_scratchpad, getrs_scratchpad_size, {getrfDone});

// Wait until calculations are done
syclQueue.wait_and_throw();
```

https://github.com/oneapi-src/oneMKL/blob/develop/examples/lapack/run_time_dispatching/getrs_usm.cpp

CMake

oneMKL is installed

```
find_package(oneMKL REQUIRED)

// Link everything, runtime dispatch
target_link_library(mytarget PRIVATE MKL::onemkl)

// Link against specific backend
target_link_library(mytarget PRIVATE
            MKL::onemkl_<domain>_<backend>)
```

Using FetchContent

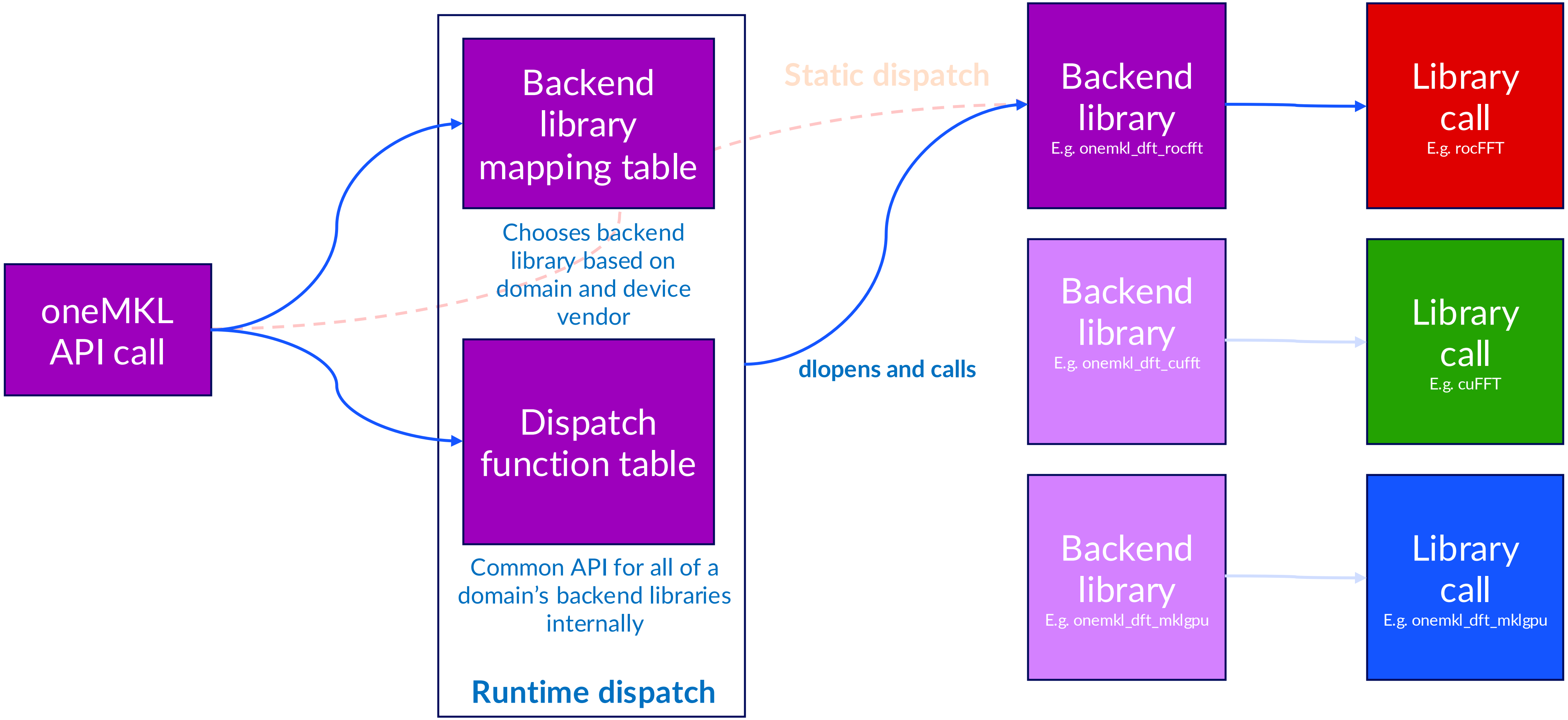
```
include(FetchContent)
set(BUILD_FUNCTIONAL_TESTS OFF)
set(BUILD_EXAMPLES OFF)
set(ENABLE_<BACKEND_NAME>_BACKEND ON)
FetchContent_Declare(
        onemkl_interface_library
        GIT_REPOSITORY https://github.com/oneapi-src/oneMKL.git
        GIT_TAG develop
)
FetchContent_MakeAvailable(onemkl_interface_library)

target_link_libraries(myTarget PRIVATE onemkl)
// or for a specific backend
target_link_libraries(myTarget PRIVATE onemkl_<domain>_<backend>)
```

… And add <install_dir>/lib to your LD_LIBRARY_PATH if its installed in a non-standard location, otherwise dlopen doesn’t work.

oneMKL: on the inside

The runtime dispatch mechanism



oneMKL: building it

The documentation makes it look harder than it is

With DPC++

```
cmake   $ONEMKL_DIR \
        -GNinja \
        -DCMAKE_CXX_COMPILER=icpx \
        -DCMAKE_C_COMPILER=icx \
        -DENABLE_MKLGPU_BACKEND=ON \
        -DENABLE_MKLCPU_BACKEND=ON \
        -DENABLE_CUFFT_BACKEND=ON \
        -DENABLE_CUBLAS_BACKEND=ON \
        -DENABLE_ROCRAND_BACKEND=ON \
        -DENABLE_FUNCTIONAL_TESTS=OFF \
        -DHIP_TARGETS=gfx90a
```

- Building isn’t that complicated.
  - Enable the backends you want
  - Set HIP_TARGETS on AMD
  - Disable functional tests in most cases
- What does supported vs unsupported mean?
  - Supported is what we actually test with
  - But using icpx + Codeplay plugins does work, and it's probably what you should do.
- On AMD, you can only have a single arch right now.

oneMKL: gotchas

oneMKL: gotchas

- Backend libraries don't all support every feature
  - Eg. The cuFFT backend doesn't support scaling.

- Backend libraries make different guarantees
  - Eg. The rocFFT backend can modify input.

oneMKL: gotchas

- Variadic functions like
  - `desc.set_value(dft::config_param::INPUT_STRIDES, myStrides);`
  - The spec uses `int64_t`
  - Variadic arguments means that the compiler won't tell you you're wrong.

- LD_LIBRARY_PATH

Coming from Intel® MKL

Intel® MKL

```
#include <oneapi/mkl/dfti.hpp>
```

```
DFTI_INPLACE
```

oneMKL Interface Library

```
#include <oneapi/mkl/dft.hpp>
```

```
oneapi::mkl::dft::config_value::INPLACE
```

oneMKL: performance

Performance

oneMKL is a thin wrapper calling native backend libraries

- very little overhead, negligible in typical HPC use cases
  - we are working on improving the overhead for small workloads where it may be more visible
- you get **comparable performance + portability**

Let's test this with a simple GEMM example!

$$C \leftarrow alpha * op(A) * op(B) + beta * C$$

op(X) is one of **op(X) = X** or **op(X) = X^T** or **op(X) = X^H**

alpha and **beta** are scalars

A, **B** and **C** are matrices

op(A) is an **m**-by-**k** matrix

op(B) is a **k**-by-**n** matrix

C is an **m**-by-**n** matrix

Performance

- Code from VelocityBench hplinpack DPC++ example
- We call it with double-precision matrices with {m,n,k} = {16384, 2048, 2048} filled with random values in the range 0.0–1.0
- Three code versions compiled into four executables:
  - CUDA: `cublasDgemm`
  - HIP: `hipblasDgemm`
  - IntelMKL / oneMKL (same API): `oneapi::mkl::blas::column_major::gemm`

Performance

Same code runs on 7 different devices from 3 different vendors (6 GPUs and 1 CPU)

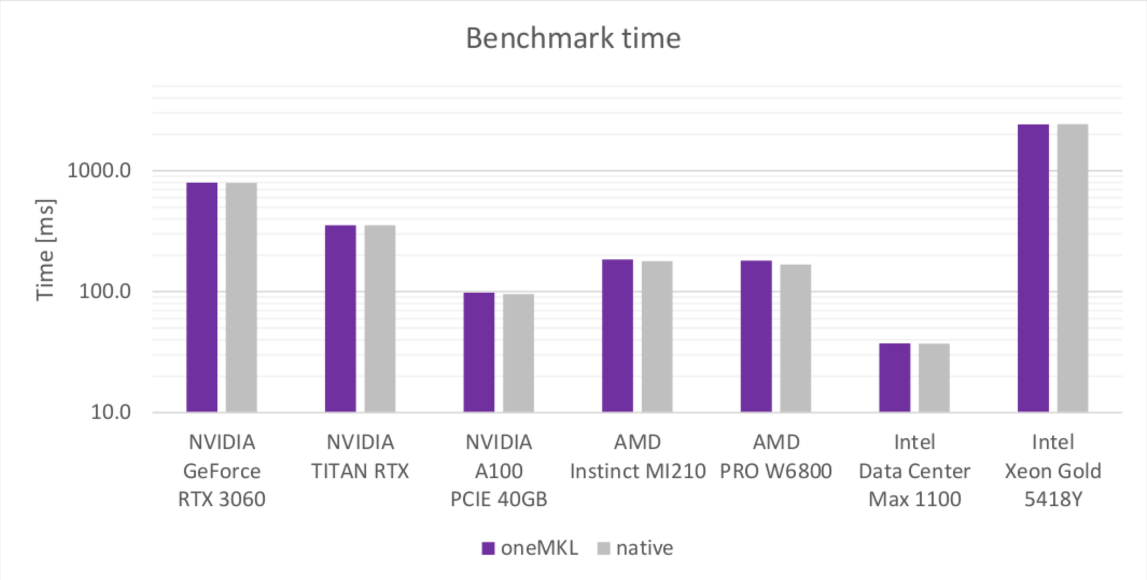
Comparable results to the native library in all cases

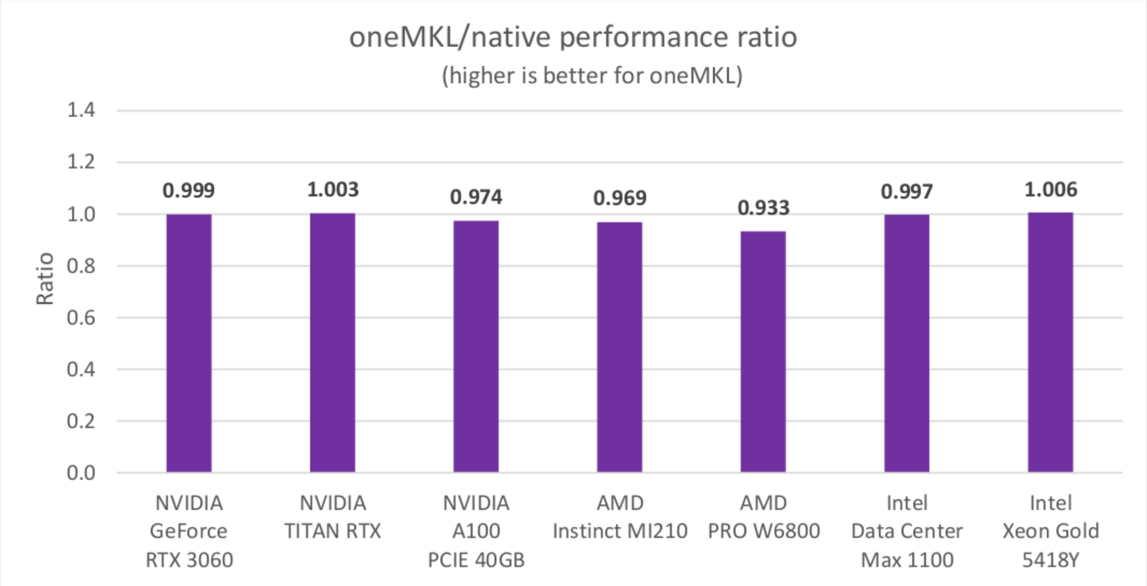
No need to maintain three versions of the code if just one does it!

Performance varies by use, configuration and other factors. 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.

Details of the software and hardware used to produce these results are available in the backup slides.





"native" means cuBLAS on NVIDIA GPU, hipBLAS on AMD GPU and Intel MKL on Intel GPU/CPU

Time to give it a go

… But first!

- Want something that oneMKL doesn’t have / support?
  - Make an issue!
- Find a bug?
  - Make an issue!
- Finding something confusing?
  - Make an issue!

Issues let us justify spending time on improving oneMKL.

Hands-on with oneMKL

- We have a pre-prepared single-page application that needs the oneMKL section added on GitHub:
- https://github.com/codeplaysoftware/syclacademy/tree/main/Code_Exercises/OneMKL_gemm
- Instructions are on the page, but the starting skeleton is in the "source" file, with the answer in the "solution" file, but we'd encourage you to give it a go before checking!

Hints

- The sample is performing a GEMM, so add this function
- If you are using the compiler standalone, i.e. without CMake or similar, flags are required:
- `icpx -fsycl -L$ENV{MKLROOT} -lonemkl solution_onemkl_usm_gemm.cpp`
- If using USM, copies to the device and back will be required

oneAPI Plugins for NVIDIA/AMD

Scan QR code or visit **developer.codeplay.com** codeplay®

Disclaimers

A wee bit of legal

Performance varies by use, configuration and other factors.

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.

Performance benchmark details

Main function: https://gist.github.com/rafbiels/e93b70098d46e947ce825eb1cc95f6b5
VelocityBench dpcpp_dgemm.cpp: https://github.com/oneapi-src/Velocity-Bench/blob/50343b438e838ceae1eb11a10196d3ae90aebb67/hplinpack/dpcpp/hpl-2.3/src/dpcpp/dpcpp_dgemm.cpp

Base compilation command: `icpx -fsycl -fsycl-targets=${SYCL_TARGET} ${OFFLOAD_ARCH_FLAGS} -o onemkl main.cpp dpcpp_dgemm.cpp`

Extra flags:
oneMKL: `-lonemkl`
Intel MKL: `-DMKL_ILP64  -qmkl=parallel -qtbb`
cuBLAS: `-DUSE_CUBLAS -lcublas -lcuda -lcudart -L$(dirname $(which nvcc))/../lib64`
hipBLAS: `-DUSE_HIPBLAS -D__HIP_PLATFORM_AMD__=${HIP_TARGET} -L${ROCM_PATH}/hipblas/lib/ -L${ROCM_PATH}/hip/lib -lhipblas -lamdhip64`

Software stack: Ubuntu 22.04.4 LTS, oneAPI Base Toolkit 2024.1, CUDA 12.4, ROCm 5.4.3, oneMKL interfaces commit 6d6a7b711dbc55c49370b8ddbcc9db6e81a6ac27 + PR #490

Hardware (6 machines):
1. Intel i9-12900K CPU + NVIDIA GeForce RTX 3060 GPU
2. Intel Xeon Platinum 8268 CPU + NVIDIA TITAN RTX GPU
3. Intel Xeon Gold 6326 CPU + NVIDIA A100 PCIE 40GB GPU
4. 2x AMD EPYC 7402 CPU + AMD Instinct MI210 GPU
5. Intel i9-12900K CPU + AMD Radeon PRO W6800 GPU
6. 2x Intel Xeon Gold 5418Y CPU + Intel Data Center Max 1100 GPU

Tested on 31 May 2024