oneMath

このドキュメントは、UXL Foundation の GitHub で公開されている「oneMath 0.1 Documentation」（2025 年 11 月現在）の日本語参考訳です。原文は更新される可能性があります。原文と翻訳文の内容が異なる場合は原文を優先してください。

本ドキュメントはレイアウト調整および校閲を行っておりません。誤字脱字、製品名や用語の表記、レイアウト等の不具合が含まれる可能性があることを予めご了承ください。

内容

概要

oneMath は、oneMath 仕様のオープンソース実装であり、複数のライブラリー（バックエンド）を使用して複数のデバイスで動作できます。

はじめに

コンパイラーの選択

必要なバックエンドとアプリケーションのオペレーティング・システムに応じてコンパイラーを選択する必要があります。

- アプリケーションがインテル® GPU を必要とする場合、Linux ではインテル® oneAPI DPC++ コンパイラー icpx、Windows では icx を使用してください。
- Linux アプリケーションで NVIDIA GPU が必要な場合、NVIDIA CUDA をサポートする oneAPI DPC++ コンパイラーの最新ソースから clang++ をビルドするか、AdaptiveCpp リポジトリーにある AdaptiveCpp を使用します（LAPACK または DFT ドメインを除く）。
- Linux アプリケーションで AMD GPU が必要な場合、HIP AMD をサポートする oneAPI DPC++ コンパイラーの最新ソースから clang++ をビルドするか、AdaptiveCpp を使用します。
- インテル® GPU、NVIDIA GPU、または AMD GPU を必要としない場合、Linux ではインテル® oneAPI DPC++ コンパイラー icpx、oneAPI DPC++ コンパイラー clang++、または AdaptiveCpp を使用できます。Windows ではインテル® oneAPI DPC++ コンパイラー icx または oneAPI DPC++ コンパイラー clang-cl を使用できます。

DPC++ でプロジェクトをビルド

このページでは、インテル® oneAPI DPC++ コンパイラー、またはオープンソースの oneAPI DPC++ コンパイラーを使用して oneMath をビルドする方法について説明します。AdaptiveCpp を使用してプロジェクトをビルドする方法は、AdaptiveCpp でプロジェクトをビルドを参照してください。

環境設定

1. 必要な DPC++ コンパイラー（インテル® DPC++ または Open DPC++ - コンパイラーの選択を参照）をインストールします。
2. このプロジェクトのクローンを作成します。クローンされたリポジトリーのルート・ディレクトリーは、<path to onemath> と呼ばれます。
3. 必要な依存関係をすべてビルドしてインストールします。

ビルドコマンド

各種コンパイラーとバックエンドのビルドコマンドでは、主にコンパイラーとバックエンドの CMake オプション値の設定が異なります。このセクションでは、一般的なビルドコマンドについて説明します。バックエンド固有の詳細については、バックエンドのセクションで説明し、CMake 呼び出しの例を示します。

Linux では、ビルドコマンドの一般的な形式は次のようになります（Windows でのビルドについては、Windows 向けのビルドを参照してください）:

```
# Inside <path to onemath>
mkdir build && cd build
cmake .. -DCMAKE_CXX_COMPILER=$CXX_COMPILER ¥ # icpx または clang++ である必要があります
        -DCMAKE_C_COMPILER=$C_COMPILER        ¥ # icx または clang である必要があります
        -DENABLE_MKLCPU_BACKEND=False         ¥ # オプション: この MKLCPU バックエンドは、デフォルトで
True に設定されています
        -DENABLE_MKLGPU_BACKEND=False         ¥ # オプション: この MKLGPU バックエンドは、デフォルトで
True に設定されています
        -DENABLE_<BACKEND_NAME>_BACKEND=True   ¥ # その他のバックエンドを有効にします（オプション）
        -DENABLE_<BACKEND_NAME_2>_BACKEND=True ¥ # 複数のバックエンドを一度に有効にできます。
        -DBUILD_FUNCTIONAL_TESTS=False        ¥ # テストのビルドの詳細については、「テストの構築と実行」の
ページを参照してください。デフォルトは True
        -DBUILD_EXAMPLES=False                 # オプション: デフォルトは True
cmake --build .
cmake --install . --prefix <path_to_install_dir>  # 完全なパッケージ構造が必要
```

上記でインテル® oneAPI DPC++ コンパイラーを使用する場合、`$CXX_COMPILER` と `$C_COMPILER` をそれぞれ `icpx` と `icx` に設定し、Open DPC++ コンパイラーを使用する場合は `clang++` と `clang` に設定する必要があります。

バックエンドを有効にするには、バックエンドごとに `-DENABLE_<BACKEND_NAME>_BACKEND=True` を設定する必要があります。デフォルトでは、`MKLGPU` および `MKLCPU` バックエンドのみが有効になっています。複数のデバイスベンダーによる複数のバックエンドを一度に有効にできます（ただし、oneMath の汎用 SYCL BLAS および portFFT を使用する場合は制限があります）。コンパイラーでサポートされているバックエンドは、oneMath でサポートされている構成の表に記載されており、CMake オプション名は以下の表に記載されています。一部のバックエンドでは、追加のパラメーターが必要となる場合があります。追加のガイドについては、以下の関連セクションを参照してください。

バックエンド・ライブラリーが複数のドメイン（BLAS、LAPACK、DFT、RNG、スパース BLAS など）をサポートしている場合、選択したドメインのみを有効にすることが望ましい場合があります。それには、`TARGET_DOMAINS` 変数を設定します。TARGET DOMAINS セクションを参照してください。

デフォルトでは、ライブラリーはサンプルとテストもビルドします。これらは、パラメーター `BUILD_FUNCTIONAL_TESTS` と `BUILD_EXAMPLES` を `False` に設定することで無効にできます。機能テストをビルドするには、BLAS および LAPACK ドメイン用の追加の外部ライブラリーが必要です。詳細については、テストのビルドと実行セクションを参照してください。

サポートされている重要なビルドオプションは次のとおりです:

CMake オプション	利用可能な値	デフォルト値
ENABLE_MKLCPU_BACKEND	True、False	True

CMake オプション	利用可能な値	デフォルト値
ENABLE_MKLGPU_BACKEND	True、False	True
ENABLE_CUBLAS_BACKEND	True、False	False
ENABLE_CUSOLVER_BACKEND	True、False	False
ENABLE_CUFFT_BACKEND	True、False	False
ENABLE_CURAND_BACKEND	True、False	False
ENABLE_CUSPARSE_BACKEND	True、False	False
ENABLE_NETLIB_BACKEND	True、False	False
ENABLE_ARMPL_BACKEND	True、False	False
ENABLE_ARMPL_OMP	True、False	True
ENABLE_ARMPL_OPENRNG	True、False	False
ENABLE_ROCBLAS_BACKEND	True、False	False
ENABLE_ROCFFT_BACKEND	True、False	False
ENABLE_ROCSOLVER_BACKEND	True、False	False
ENABLE_ROCRAND_BACKEND	True、False	False
ENABLE_ROCSPARSE_BACKEND	True、False	False
ENABLE_MKLCPU_THREAD_TBB	True、False	True
ENABLE_GENERIC_BLAS_BACKEND	True、False	False
ENABLE_PORTFFT_BACKEND	True、False	False
BUILD_FUNCTIONAL_TESTS	True、False	True
BUILD_EXAMPLES	True、False	True
TARGET_DOMAINS (list)	blas 、 lapack 、 rng 、 dft 、 sparse_blas	すべてのドメイン

追加のビルドオプションについては、追加のビルドオプションのセクションを参照してください。

TARGET_DOMAINS

oneMath は複数のドメインをサポートしています: BLAS、DFT、LAPACK、RNG、スパース BLAS。oneMath によってビルドされたドメインは、`TARGET_DOMAINS` パラメーターを使用して選択できます。ほとんどの場合、`TARGET_DOMAINS` は、有効になっているバックエンド・ライブラリーでサポートされるドメインに応じて自動的に設定されます。ほとんどのバックエンド・ライブラリーは、これらのドメインの 1 つだけをサポートしますが、複数のドメインをサポートするライブラリーもあります。たとえば、`MKLCPU` バックエンドはすべてのドメインをサポートします。`MKLCPU` でコンパイルし oneMath の BLAS ドメインのみのサポートを有効にするには、`TARGET_DOMAINS` を `blas` に設定します。BLAS と DFT を有効にするには、`-DTARGET_DOMAINS="blas dft"` を使用します。

バックエンド

インテル® oneMKL 向けのビルド

インテル® oneMKL バックエンドは、x86 CPU とインテル® GPU の両方で複数のドメインをサポートします。x86 CPU 用のインテル® oneMKL を使用する MKLCPU バックエンドはデフォルトで有効になっており、パラメーター `ENABLE_MKLCPU_BACKEND` で制御されます。インテル® GPU 用のインテル® oneMKL を使用する MKLGPU バックエンドはデフォルトで有効になっており、パラメーター `ENABLE_MKLGPU_BACKEND` で制御されます。

インテル® oneAPI DPC++ コンパイラーを使用する場合、インテル® oneMKL が自動的に検出される可能性があります。検出されない場合、パラメーター `MKL_ROOT` を oneMKL のインストール・プレフィクスを指すように設定できます。あるいは、`MKLROOT` 環境変数を手動で設定するか、パッケージで提供される環境スクリプトを使用して設定することもできます。

CUDA 向けにビルド

CUDA バックエンドは、`ENABLE_CUBLAS_BACKEND`、`ENABLE_CUFFT_BACKEND`、`ENABLE_CURAND_BACKEND`、`ENABLE_CUSOLVER_BACKEND`、および `ENABLE_CUSPARSE_BACKEND` で有効にできます。

CUDA ライブラリーを使用するには、追加のパラメーターは必要ありません。ほとんどの場合、CUDA ライブラリーは CMake で自動的に検出されます。

ROCm 向けにビルド

ROCm バックエンドは、`ENABLE_ROCBLAS_BACKEND`、`ENABLE_ROCFFT_BACKEND`、`ENABLE_ROCSOLVER_BACKEND`、`ENABLE_ROCRAND_BACKEND`、および `ENABLE_ROCSPARSE_BACKEND` で有効にできます。

RocBLAS、*RocSOLVER*、*RocRAND*、および *RocSPARSE* の場合、ターゲット・デバイス・アーキテクチャーを設定する必要があります。これは、`HIP_TARGETS` パラメーターで設定できます。たとえば、MI200 シリーズ GPU のビルドを有効にするには、`-DHIP_TARGETS=gfx90a` を設定します。現在、DPC++ は一度に 1 つの HIP ターゲットに対してのみビルドできます。これは、将来のバージョンで変更される可能性があります。

よく使用されるアーキテクチャーを以下に示します:

アーキテクチャー	AMD GPU 名
gfx90a	AMD Instinct™ MI210/250/250X アクセラレーター
gfx908	AMD Instinct™ MI 100 アクセラレーター
gfx906	AMD Radeon Instinct™ MI50/60 アクセラレーター AMD Radeon™ (Pro) VII グラフィックス・カード
gfx900	Radeon Instinct™ MI 25 アクセラレーター Radeon™ RX Vega 64/56 グラフィックス

ROCm がインストールされているホストでは、`rocminfo` ツールを使用してデバイス・アーキテクチャーを取得できます。アーキテクチャーは `Name:` 行に表示されます。

他の SYCL デバイス向けのビルド

SYCL は、幅広いアクセラレーター上でポータブルな異種コンピューティングを可能にします。その結果、プロジェクトで想定されていないアクセラレーターで oneMath を使用することが可能となります。

汎用 SYCL デバイスの場合、汎用 BLAS と portFFT バックエンドのみが有効になります。ユーザーは、デバイスに適切な `-fsycl-targets` を設定し、パフォーマンスに必要なその他のオプションも設定します。oneMath 汎用 SYCL BLAS のビルドと portFFT のビルドを参照してください。サポートされていない構成については、広範なテストを行うことを強く推奨します。

純粋な SYCL バックエンド: 汎用 BLAS と portFFT

汎用 SYCL BLAS および portFFT は、DPC++ コンパイラーでサポートされているすべての SYCL ターゲットで動作する実験的な純粋な SYCL バックエンドです。複数のターゲットをサポートしているため、同じドメイン内の他のバックエンドや、`MKLCPU` または `MKLGPU` バックエンドでは有効にできません。どちらのライブラリーも実験段階であり、現在は操作と機能のサブセットのみをサポートしています。

最高のパフォーマンスを得るには、両方のライブラリーをチューニングする必要があります。詳細については個別のセクションを参照してください。

汎用 SYCL BLAS と portFFT はどちらもヘッダーのみのライブラリーとして使用され、見つからない場合は自動的にダウンロードされます。

oneMath 汎用 SYCL BLAS 用のビルド

onemath 汎用 SYCL BLAS は、`-DENABLE_GENERIC_BLAS_BACKEND=True` を設定することで有効になります。

デフォルトでは、汎用 BLAS バックエンドは特定のデバイス用にチューニングされていません。最高のパフォーマンスを実現するには、チューニングが必要です。汎用 SYCL BLAS バックエンドは、次の 2 つの方法でコンパイラー定義を追加することで、特定のハードウェア・ターゲットに合わせてチューニングできます:

1. `-DGENERIC_BLAS_TUNING_TARGET=<target>` を使用して、チューニング・ターゲットを手動で指定します。oneMath SYCL BLAS ターゲットのリストは、こちらでご覧いただけます。これにより、必要に応じて `-fsycl-targets` が自動的に設定されます。
2. `-fsycl-targets` で 1 つのターゲットが設定されている場合、構成手順では oneMath SYCL BLAS チューニング・ターゲットを自動的に検出しようとします。`CMAKE_CXX_FLAGS` を介して `-fsycl-targets` を手動で指定できます。`-fsycl-targets` の詳細については、DPC++ ユーザーマニュアルを参照してください。

OneMath SYCL BLAS は JIT コンパイルに大きく依存しています。これにより、一部のシステムではタイムアウトが発生する可能性があります。この問題を回避するには、チューニング・ターゲットまたは `sycl-targets` を通じて事前コンパイル（AOT）を使用します。

portFFT のビルド

portFFT は、`-DENABLE_PORTFFT_BACKEND=True` を設定することで有効になります。

デフォルトでは、portFFT バックエンドは特定のデバイス用にチューニングされていません。チューニング・フラグの

詳細は portFFT リポジトリーに記載されており、構成時に設定できます。一部のチューニング構成は、一部のターゲットと互換性がない可能性があることに注意してください。

portFFT ライブラリーは、`CMAKE_CXX_FLAGS` で指定されたものと同じ `-fsycl-targets` を使用してコンパイルされます。何も見つからない場合、`-fsycl-targets=spir64` 用にコンパイルされ、コンパイラーがサポートしている場合は `nvptx64-nvidia-cuda` 用にコンパイルされます。HIP ターゲットを有効にするには、`HIP_TARGETS` を指定する必要があります。`-fsycl-targets` の詳細については、DPC++ ユーザーマニュアルを参照してください。

Arm パフォーマンス・ライブラリーのビルド

Arm パフォーマンス・ライブラリー・バックエンドは、`-DENABLE_ARMPL_BACKEND=True` を設定することで aarch64 プラットフォームで有効になります。

デフォルトでは、`ARMPLROOT` 環境変数を検索します。別の ArmPL を使用する場合、`-DARMPL_ROOT=<armpl_install_prefix>` を使用できます。

デフォルトの動作では、ArmPL ライブラリーの OpenMP フレーバーが使用されますが、これは `-DENABLE_ARMPL_OMP=True/False` フラグを使用して変更できます。

ArmPL は、乱数ジェネレーター・インターフェイスの実装として OpenRNG プロジェクトをバンドルしています。RNG ドメイン用の oneMath ArmPL バックエンドは、ArmPL の代わりに OpenRNG のオープンソース・バージョンを使用してビルドできます。このビルドは、aarch64 と x86_64 の両方の CPU アーキテクチャーをサポートします。ArmPL バイナリーが利用できない場合、*-DTARGET_DOMAINS=rng -DENABLE_ARMPL_BACKEND=True* で oneMath をビルドすると、ビルドはデフォルトで OpenRNG を使用するように切り替わります。オプション *-DENABLE_ARMPL_OPENRNG=True* を使用して、OpenRNG の使用を強制することもできます。

追加のビルドオプション

oneMath をビルドするときに、`ONEMATH_SYCL_IMPLEMENTATION` オプションを設定することで SYCL 実装を指定できます。設定可能な値は以下です:

- インテル® oneAPI DPC++ コンパイラーおよび oneAPI DPC++ コンパイラーの場合、`dpc++`（デフォルト）です。
- AdaptiveCpp SYCL 実装用の `AdaptiveCpp`。

このオプションを使用する場合、AdaptiveCpp でプロジェクトをビルドを参照してください。

次の表は、CMake オプションとそのデフォルト値の詳細を示しています:

CMake オプション	利用可能な値	デフォルト値
BUILD_SHARED_LIBS	True、False	True
BUILD_DOC	True、False	False

注

AMD バックエンド向けに `clang++` を使用してビルドする場合、さらに `ONEAPI_DEVICE_SELECTOR` を

`hip:gpu` に設定し、対象のハードウェアに応じて `-DHIP_TARGETS` を指定する必要があります。このバックエンドは、このドキュメント執筆時点では `gfx90a` アーキテクチャー（MI210）でのみテストされています。

注

`BUILD_FUNCTIONAL_TESTS=True`（デフォルトのオプション）でビルドする場合、単一の CUDA バックエンドのみをビルドできます（#270）。

CMake の呼び出し例

Nvidia GPU のサポートとテストを無効にして、Ninja ビルドシステムを使用して oneMath をビルドします:

```
cmake $ONEMATH_DIR ¥
    -GNinja ¥
    -DCMAKE_CXX_COMPILER=clang++ ¥
    -DCMAKE_C_COMPILER=clang ¥
    -DENABLE_MKLGPU_BACKEND=False ¥
    -DENABLE_MKLCPU_BACKEND=False ¥
    -DENABLE_CUFFT_BACKEND=True ¥
    -DENABLE_CUBLAS_BACKEND=True ¥
    -DENABLE_CUSOLVER_BACKEND=True ¥
    -DENABLE_CURAND_BACKEND=True ¥
    -DENABLE_CUSPARSE_BACKEND=True ¥
    -DBUILD_FUNCTIONAL_TESTS=False
```

`$ONEMATH_DIR` は oneMath ソースを直接示します。x86 CPU（`MKLCPU`）およびインテル® GPU（`MKLGPU`）バックエンドはデフォルトで有効になっていますが、ここでは無効にしています。Nvidia GPU のバックエンドはすべて明示的に有効にする必要があります。テストは無効になっていますが、サンプルは引き続きビルドされます。

テストを無効にして AMD GPU をサポートする oneMath をビルドします:

```
cmake $ONEMATH_DIR ¥
    -DCMAKE_CXX_COMPILER=clang++ ¥
    -DCMAKE_C_COMPILER=clang ¥
    -DENABLE_MKLCPU_BACKEND=False ¥
    -DENABLE_MKLGPU_BACKEND=False ¥
    -DENABLE_ROCFFT_BACKEND=True  ¥
    -DENABLE_ROCBLAS_BACKEND=True ¥
    -DENABLE_ROCSOLVER_BACKEND=True ¥
    -DENABLE_ROCSPARSE_BACKEND=True ¥
    -DHIP_TARGETS=gfx90a ¥
    -DBUILD_FUNCTIONAL_TESTS=False
```

`$ONEMATH_DIR` は oneMath ソースを直接示します。x86 CPU（`MKLCPU`）およびインテル® GPU（`MKLGPU`）バックエンドはデフォルトで有効になっていますが、ここでは無効にしています。AMD GPU のバックエンドはすべて

明示的に有効にする必要があります。テストは無効になっていますが、サンプルは引き続きビルドされます。

テストを有効にして、x86 CPU、インテル® GPU、AMD GPU、Nvidia GPU をサポートする DFT ドメイン用の oneMath をビルドします:

```
cmake $ONEMATH_DIR ¥
    -DCMAKE_CXX_COMPILER=icpx ¥
    -DCMAKE_C_COMPILER=icx ¥
    -DENABLE_ROCFFT_BACKEND=True ¥
    -DENABLE_CUFFT_BACKEND=True ¥
    -DTARGET_DOMAINS=dft ¥
    -DBUILD_EXAMPLES=False
```

これはサポートされている構成ではなく、AMD および Nvidia GPU プラグイン用の Codeplay の oneAPI が必要であることに注意してください。MKLCPU および MKLGPU バックエンドはデフォルトで有効になっており、Nvidia GPU および AMD GPU のバックエンドは明示的に有効にしています。`-DTARGET_DOMAINS=dft` を指定すると、DFT バックエンドのみがビルドされます。これが設定されていない場合、MKLGPU および MKLCPU で BLAS、LAPACK、および RNG を使用できるバックエンド・ライブラリーも有効になります。サンプルのビルドは無効になっています。機能テストが無効化されていないため、テストがビルドされます。

汎用 SYCL デバイス上で BLAS ドメイン用の oneMath をビルドします:

```
cmake $ONEMATH_DIR ¥
    -DCMAKE_CXX_COMPILER=clang++ ¥
    -DCMAKE_C_COMPILER=clang ¥
    -DENABLE_MKLCPU_BACKEND=False ¥
    -DENABLE_MKLGPU_BACKEND=False ¥
    -DENABLE_GENERIC_BLAS_BACKEND=True
```

これはテスト済みの構成ではないことに注意してください。これにより、汎用 SYCL デバイス用に、汎用 SYCL BLAS バックエンドのみを使用して oneMath がビルドされます。

汎用 SYCL デバイス上で DFT ドメイン用の oneMath をビルドします:

```
cmake $ONEMATH_DIR ¥
    -DCMAKE_CXX_COMPILER=clang++ ¥
    -DCMAKE_C_COMPILER=clang ¥
    -DENABLE_MKLCPU_BACKEND=False ¥
    -DENABLE_MKLGPU_BACKEND=False ¥
    -DENABLE_PORTFFT_BACKEND=True
```

これはテスト済みの構成ではないことに注意してください。これは、Open DPC++ プロジェクトでサポートされている汎用 SYCL デバイス用に、portFFT バックエンドのみを使用して oneMath をビルドします。

プロジェクトのクリーンアップ

ほとんどのユースケースでは、ビルド・ディレクトリーをクリーンアップする必要なくプロジェクトをビルドします。ま

た、ビルド・ディレクトリーをクリーンアップして、`build` フォルダーを削除して新しいフォルダーを作成することもできます。ビルドファイルをクリーンアップしながらビルド構成を保持する場合、次のコマンドが役立ちます。

```
# ビルドに "GNU/Unix Makefiles" を使用する場合、
make clean

# ビルドに "Ninja" を使用する場合、
ninja -t clean
```

Windows 向けのビルド

Windows 向けのビルドは Linux ビルドに似ていますが、サポートされるバックエンドは少なくなっています。さらに、Ninja ビルドシステムを使用する必要があります。以下に例を示します:

```
# Inside <path to onemath>
md build && cd build
cmake .. -G Ninja [-DCMAKE_CXX_COMPILER=<path_to_icx_compiler>¥bin¥icx] # 環境変数 PATH に icx が見つからない場合にのみ必要
                  [-DCMAKE_C_COMPILER=<path_to_icx_compiler>¥bin¥icx]   # 環境変数 PATH に icx が見つからない場合にのみ必要
                  [-DMKL_ROOT=<mkl_install_prefix>]                     # 環境変数 MKLROOT が設定されていない場合にのみ必要
                  [-DREF_BLAS_ROOT=<reference_blas_install_prefix>]     # テストにのみ必要
                  [-DREF_LAPACK_ROOT=<reference_lapack_install_prefix>] # テストにのみ必要
ninja
ctest
cmake --install . --prefix <path_to_install_dir> # 完全なパッケージ構造が必要
```

ビルド の FAQ

clangrt ビルトイン・ライブラリーが見つかりません

いくつかの ROCm ライブラリーを使用して oneMath をビルドしようとしたときに発生しました。解決策はいくつか考えられます:

- Open DPC++ をソースからビルドする場合、外部プロジェクトのコンパイルオプションに `compiler-rt` を追加します: `--llvm-external-projects compiler-rt`
- 変数 `HIP_CXX_COMPILER` を HIP ツールキット `clang++` パスに手動で設定します（例: `-DHIP_CXX_COMPILER=/opt/rocm/6.1.0/llvm/bin/clang++`）。`icpx` と `rocm` の clang バージョンに互換性がない場合、oneMath はリンクに失敗する可能性があります。

CBLAS が見つかりません（不足: CBLAS ファイル）

BLAS ドメインとともにテストが有効になっている場合に発生します。テストにはリファレンス BLAS 実装が必要ですが、見つかりません。BLAS ライブラリーをインストールまたはビルドし、テストのビルドと実行の説明に従って`-DREF_BLAS_ROOT` を設定します。あるいは、`-DBUILD_FUNCTIONAL_TESTS=False` を設定してテストを無効にすることもできます。

エラー: 無効なターゲット ID ''; フォーマットはプロセッサー名、オプションのコロンで区切られた機能リスト、有効/無効記号の順です（例: 'gfx908:sramecc+:xnack-'）

HIP_TARGET が設定されていません。ROCm 向けにビルドを参照してください。

AdaptiveCpp でプロジェクトをビルド

環境設定

1. AdaptiveCpp をビルドしてインストールします。利用可能な AdaptiveCpp バックエンド、その依存関係、およびインストールの詳細については、AdaptiveCpp インストールの readme を参照してください。
2. このプロジェクトのクローンを作成します。クローンされたリポジトリーのルート・ディレクトリーは、`<path to onemath>` と呼ばれます。
3. 必要な依存関係を手動でダウンロードしてインストールします。

ビルドコマンド

ほとんどの場合、oneMath のビルドは、コンパイラーを設定し、ビルドに使用するバックエンドを選択するだけです。

Linux の場合（他の OS は AdaptiveCpp コンパイラーでサポートされていません）:

```
# Inside <path to onemath>
mkdir build && cd build
cmake .. -DONEMATH_SYCL_IMPLEMENTATION=adaptivecpp ¥ # AdaptiveCpp が使用されていることを示します
        -DENABLE_MKLGPU_BACKEND=False             ¥ # MKLGPU バックエンドは AdaptiveCpp ではサポートされていません
        -DENABLE_<BACKEND_NAME>_BACKEND=True      ¥ # バックエンドを有効にします（オプション）
        -DENABLE_<BACKEND_NAME_2>_BACKEND=True    ¥ # 複数のバックエンドを一度に有効にできます。
        -DACPP_TARGETS=omp¥;hip:gfx90a,gfx906     ¥ # サポートされているデバイスに応じてターゲット・アーキテクチャーを設定します
        -DBUILD_FUNCTIONAL_TESTS=False            ¥ # テストのビルドの詳細については、*テストのビルドと実行* のセクションを参照してください。デフォルトは True。
        -DBUILD_EXAMPLES=False                   # オプション: デフォルトは True。
cmake --build .
cmake --install . --prefix <path_to_install_dir> # 完全なパッケージ構造が必要
```

バックエンドを有効にするには、バックエンドごとに `-DENABLE_<BACKEND_NAME>_BACKEND=True` を設定する必要があります。デフォルトでは、`MKLGPU` および `MKLCPU` バックエンドは有効になっていますが、AdaptiveCpp では `MKLGPU` を無効にする必要があります。コンパイラーでサポートされているバックエンドは、oneMath でサポートされている構成の表に記載されており、CMake オプション名は以下の表に記載されています。一部のバックエンドでは、追加のパラメーターが必要となる場合があります。追加のガイドについては、以下の関連セクションを参照してください。ターゲット・アーキテクチャーは `ACPP_TARGETS` で指定できます。指定しないと `generic`（汎用）ターゲットが使用されます。AdaptiveCpp のドキュメントを参照してください。

バックエンド・ライブラリーが複数のドメイン（BLAS、DFT、RNG など）をサポートしている場合、選択したドメイン

のみを有効にすることが望ましい場合があります。それには、`TARGET_DOMAINS` 変数を設定します。詳細については、_build_target_domains を参照してください。

デフォルトでは、ライブラリーはサンプルとテストもビルドします。これらは、パラメーター `BUILD_FUNCTIONAL_TESTS` と `BUILD_EXAMPLES` を False に設定することで無効にできます。機能テストをビルドするには、追加の外部ライブラリーが必要になる場合があります。詳細については、テストのビルドと実行セクションを参照してください。

サポートされている最も重要なビルドオプションは次のとおりです:

CMake オプション	利用可能な値	デフォルト値
ENABLE_MKLCPU_BACKEND	True、False	True
ENABLE_CUBLAS_BACKEND	True、False	False
ENABLE_CUFFT_BACKEND	True、False	False
ENABLE_CURAND_BACKEND	True、False	False
ENABLE_NETLIB_BACKEND	True、False	False
ENABLE_ARMPL_BACKEND	True、False	False
ENABLE_ROCBLAS_BACKEND	True、False	False
ENABLE_ROCFFT_BACKEND	True、False	False
ENABLE_ROCRAND_BACKEND	True、False	False
ENABLE_MKLCPU_THREAD_TBB	True、False	True
BUILD_FUNCTIONAL_TESTS	True、False	True
BUILD_EXAMPLES	True、False	True
TARGET_DOMAINS (list)	blas、dft、rng	サポートされているすべてのドメイン

追加のビルドオプションについては、追加のビルドオプションを参照してください。

バックエンド

CUDA 向けにビルド

CUDA バックエンドは、`ENABLE_CUBLAS_BACKEND`、`ENABLE_CUFFT_BACKEND`、`ENABLE_CURAND_BACKEND` で有効にできます。

ターゲット・アーキテクチャーは、`ACPP_TARGETS` パラメーターで指定できます。たとえば、Nvidia A100（Ampere アーキテクチャー）をターゲットにするには、`-DACPP_TARGETS=cuda:sm_80` を設定します。ここで、`80` は CUDA compute capability 8.0 に対応します。Compute capability と Nvidia GPU 製品の対応については、Nvidia のウェブサイトに記載されています。カンマで区切ったリストを使用して、複数のアーキテクチャーを有効にできます。AdaptiveCpp のドキュメントを参照してください。

CUDA ライブラリーを使用するには、追加のパラメーターは必要ありません。ほとんどの場合、CUDA ライブラリーは CMake で自動的に検出されます。

ROCm 向けにビルド

ROCm バックエンドは、`ENABLE_ROCBLAS_BACKEND`、`ENABLE_ROCFFT_BACKEND`、`ENABLE_ROCRAND_BACKEND` で有効にできます。

ターゲット・アーキテクチャーは、`ACPP_TARGETS` パラメーターで指定できます。AdaptiveCpp のドキュメントを参照してください。たとえば、MI200 シリーズをターゲットにするには、`-DACPP_TARGETS=hip:gfx90a` を設定します。カンマで区切ったリストを使用して、複数のアーキテクチャーを有効にできます。たとえば、`-DACPP_TARGETS=hip:gfx906,gfx90a` とセミコロンを使用した複数の API（`-DACPP_TARGETS=omp¥;hip:gfx906,gfx90a`）などです。

一般的な AMD GPU アーキテクチャーについては、DPC++ ビルドガイドの ROCm 向けにビルドを参照してください。

プロジェクトのクリーンアップ

ほとんどのユースケースでは、ビルド・ディレクトリーをクリーンアップする必要なくプロジェクトをビルドします。また、ビルド・ディレクトリーをクリーンアップして、`build` フォルダーを削除して新しいフォルダーを作成することもできます。ビルドファイルをクリーンアップしながらビルド構成を保持する場合、次のコマンドが役立ちます。

```
# ビルドに "GNU/Unix Makefiles" を使用する場合、
make clean


# ビルドに "Ninja" を使用する場合、
ninja -t clean
```

テストのビルドと実行

機能テストはデフォルトで有効になっており、CMake ビルド・パラメーター `-DBUILD_FUNCTIONAL_TESTS=True/False` で有効/無効にできます。有効なバックエンドとターゲットドメインに関連するテストのみがビルドされます。

BLAS および LAPACK ドメインのテストをビルドするには、リファレンスの追加ライブラリーが必要です。

- BLAS: リファレンス BLAS ライブラリーが必要です。
- LAPACK: リファレンス LAPACK ライブラリーが必要です。

BLAS と LAPACK の両方の場合、32 ビットと 64 ビットの両方のインデックスをサポートする共有ライブラリーが必要です。

リファレンス LAPACK 実装（BLAS を含む）は次のようにビルドできます:

```
git clone https://github.com/Reference-LAPACK/lapack.git
cd lapack; mkdir -p build; cd build
cmake -DCMAKE_INSTALL_PREFIX=~/lapack -DCBLAS=True -DLAPACK=True -DLAPACKE=True -
DBUILD_INDEX64=True -DBUILD_SHARED_LIBS=True ..
cmake --build .-j --target install
```

```
cmake -DCMAKE_INSTALL_PREFIX=~/lapack -DCBLAS=True -DLAPACK=True -DLAPACKE=True -
DBUILD_INDEX64=False -DBUILD_SHARED_LIBS=True ..
cmake --build .-j --target install
```

そして、`-DREF_BLAS_ROOT=/path/to/lapack/install` と `-DREF_LAPACK_ROOT=/path/to/lapack/install` を設定して oneMath で使用します。

プロジェクト全体を再ビルドすることなくテストを再実行できます。

テストを実行するには、テストバイナリーを個別に実行するか、`ctest` CMake テスト・ドライバー・プログラムを使用します。

```
# すべてのテストを実行
ctest
# GPU 固有のテストのみを実行
ctest -R Gpu
# CPU テストを除外
ctest -E Cpu
```

`ctest` のその他のオプションについては、ctest のマニュアルページを参照してください。

CMake のプロジェクトで oneMath を使用

CMake ビルドツールを使用すると、独自のプロジェクトで oneMath を使用できます。ディレクトリーを手動でリンクしてインクルードする代わりに、oneMath プロジェクトでエクスポートされた CMake ターゲットを使用できます。oneMath は 2 つの形式のいずれかで使用できます。ターゲット名は、採用するアプローチによって異なります:

- バイナリー配布またはソースからビルドされた、以前にインストールされたコピーを使用できます。これは、CMake の `find_package` コマンドを使用してインポートできます。using_from_installed_binary セクションを参照してください。
- または、CMake の FetchContent 機能を使用して、ビルドプロセスの一部として CMake に oneMath を自動的にダウンロードしてビルドすることもできます。using_with_fetchcontent セクションを参照してください。

インストールされた oneMath を使用

すでにソースからビルドするか、配布バイナリーとして oneMath がインストールされている場合、CMake の `find_package(oneMath REQUIRED)` でそれらを使用できます。ターゲット・ライブラリーまたはアプリケーションに使用されるコンパイラーは、oneMath のビルドに使用されるコンパイラーと同一である必要があります。

以下に例を示します:

```
find_package(oneMath REQUIRED)
target_link_libraries(myTarget PRIVATE ONEMATH::onemath)
```

oneMath の要件に応じて、異なるターゲットを使用できます。実行時ディスパッチを使用してライブラリー全体をリンクするには、`ONEMATH::onemath` ターゲットを使用する必要があります。コンパイル時ディスパッチを備えた特定のバックエンドでは、`ONEMATH::onemath_<domain>_<backend>` を使用する必要があります。

バイナリーを使用する場合、ビルド中に有効になったバックエンドを知っておくと便利な場合があります。バックエンドの存在を確認するには、CMake の `if(TARGET <target>)` 構造を使用できます。例えば、`cufft` バックエンドを使用する場合:

```
if(TARGET ONEMATH::onemath_dft_cufft)
    target_link_libraries(myTarget PRIVATE ONEMATH::onemath_dft_cufft)
else()
    message(FATAL_ERROR "oneMath was not built with CuFFT backend")
endif()
```

CMake の FetchContent を使用

CMake の FetchContent 機能を使用すると、ビルドの一部として oneMath をダウンロード、ビルド、インストールできます。

以下に例を示します:

```
include(FetchContent)
set(BUILD_FUNCTIONAL_TESTS False)
set(BUILD_EXAMPLES False)
set(ENABLE_<BACKEND_NAME>_BACKEND True)
FetchContent_Declare(
        onemath_library
        GIT_REPOSITORY https://github.com/uxlfoundation/oneMath.git
        GIT_TAG develop
)
FetchContent_MakeAvailable(onemath_library)

target_link_libraries(myTarget PRIVATE onemath)
```

`FetchContent_Declare` の前にビルド・パラメーターを適切に設定する必要があります。DPC++ でプロジェクトをビルドまたは AdaptiveCpp でプロジェクトをビルドを参照してください。

実行時ディスパッチを使用してメイン・ライブラリーにリンクするには、ターゲット `onemath` を使用します。コンパイル時ディスパッチを使用して特定のバックエンドにリンクするには、ターゲット `onemath_<domain>_<backend>` を使用します。例えば、`onemath_dft_cufft` です。

デベロッパー・リファレンス

スパース線形代数

スパースドメインの最新の仕様については、こちらをご覧ください。

このページでは、スパースドメインの実装固有またはバックエンド固有の詳細について説明します。

OneMKL インテル® CPU および GPU バックエンド

既知の制限事項:

- `no_optimize_alg` を除くすべての操作のアルゴリズムは、デフォルトのアルゴリズムにマップされます。
- 必要な外部ワークスペースのサイズは常に 0 バイトです。
- `oneapi::math::sparse::set_csr_data` 関数と `oneapi::math::sparse::set_coo_data` 関数は、操作またはその最適化関数ですでに使用されているハンドルでは使用できません。使用すると、`oneapi::math::unimplemented` 例外がスローされます。
- `oneapi::math::sparse::spsv_alg::no_optimize_alg` と、`oneapi::math::sparse::matrix_property::sorted` プロパティーを持たないスパース行列を指定した `spsv` を使用すると、`oneapi::math::unimplemented` 例外がスローされます。
- `oneapi::math::transpose::conjtrans` であり、`oneapi::math::sparse::matrix_property::symmetric` プロパティーを持つスパース行列でインテル® GPU 上で `spmm` を使用すると、`oneapi::math::unimplemented` 例外がスローされます。
- `type_view` `matrix_descr::symmetric` または `matrix_descr::hermitian` を持つ `oneapi::math::transpose::conjtrans` である疎行列で `spmv` を使用すると、`oneapi::math::unimplemented` 例外がスローされます。
- インテル® GPU で、`oneapi::math::transpose::conjtrans` であるスパース行列を使用して `spsv` を使用すると、`oneapi::math::unimplemented` 例外がスローされます。
- スカラー・パラメーター `alpha` と `beta` は、同期やホストへのコピーを防ぐためホストポインターである必要があります。

cuSPARSE バックエンド

既知の制限事項:

- COO 形式では、インデックスを行ごとに並べ替える必要があります。cuSPARSE のドキュメントを参照してください。プロパティー `matrix_property::sorted_by_rows` または `matrix_property::sorted` を指定せずに COO 形式の行列を使用するスパース演算では、`oneapi::math::unimplemented` 例外がスローされます。
- アルゴリズム `spmm_alg::csr_alg3` と `transpose::nontrans` 以外の `opA`、または `transpose::conjtrans` の `opB` で `spmm` を使用すると、`oneapi::math::unimplemented` 例外がスローされます。
- アルゴリズム `spmm_alg::csr_alg3`、`opB=transpose::trans`、および実数 fp64 精度で `spmm` を使用すると、`oneapi::math::unimplemented` 例外がスローされます。この構成は CUDA 12.6.2 以降では失敗する可能性があります。関連する問題については、`here<https://forums.developer.nvidia.com/t/cusparse-spmm-sample-failing-with-misaligned-address/311022>` を参照してください。
- `spmv` を `matrix_descr::general` 以外の `type_view` で使用すると、`oneapi::math::unimplemented` 例外がスローされます。
- アルゴリズム `spsv_alg::no_optimize_alg` と共に `spsv` を使用すると、必須の前処理が実行される場合があります。

- oneMath では、cusparseSpMM_preprocess や cusparseSpMV_preprocess などの前処理関数を呼び出さずに、デフォルト以外のアルゴリズムを使用する方法は提供されていません。必要であれば、気軽に要望を作成してください。

rocSPARSE バックエンド

既知の制限事項:

- spmv を matrix_descr::general 以外の type_view で使用すると、oneapi::math::unimplemented 例外がスローされます。
- COO 形式では、インデックスを行順に並べ替えてから列順に並べ替える必要があります。rocSPARSE COO ドキュメントを参照してください。プロパティー matrix_property::sorted なしで COO 形式の行列を使用するスパース演算では、oneapi::math::unimplemented 例外がスローされます。
- CSR 形式では、各行内で列インデックスをソートする必要があります。rocSPARSE CSR ドキュメントを参照してください。プロパティー matrix_property::sorted なしで CSR 形式の行列を使用するスパース演算では、oneapi::math::unimplemented 例外がスローされます。
- 同じスパース行列ハンドルを複数の操作 spmm、spmv、または spsv で再利用することはできません。使用すると、oneapi::math::unimplemented 例外がスローされます。#332 を参照してください。

操作アルゴリズムのマッピング

次の表は、oneMath アルゴリズムがバックエンドのアルゴリズムにどのようにマッピングされるか示しています。アルゴリズムの詳細な説明については、バックエンドのドキュメントを参照してください。

同等のアルゴリズムを持たないバックエンドでは、バックエンドのデフォルトの動作にフォールバックします。

spmm

spmv

spsv

サードパーティー・ライブラリーを oneAPI Math ライブラリー（oneMath）に統合

このステップバイステップのチュートリアルでは、oneMath で新しいサードパーティー・ライブラリーを有効にする例を示します。

oneMath には、インターフェイス・レイヤーのヘッダーベースの実装（include ディレクトリー）と、サードパーティー・ライブラリーごとのバックエンド・レイヤーのソースベースの実装（src ディレクトリー）があります。サードパーティー・ライブラリーを有効にするには、oneMath の両方の部分を更新し、新しいサードパーティー・ライブラリーを oneMath ビルドおよびテストシステムに統合する必要があります。

新しいバックエンド・ライブラリーとヘッダーの命名には、次のテンプレートを使用します:

```
onemath_<domain>_<3rd-party library short name>[<wrapper for specific target>]
```

ここで、<wrapper for specific target> は、同じサードパーティー・ライブラリーから複数のラッパーが提供さ

れている場合にのみ必要です。たとえば、CPU ターゲット `onemath_blas_mklcpu.so` 用のインテル® oneMKL C API を使用したラッパーと、GPU ターゲット `onemath_blas_mklgpu.so` 用のインテル® oneMKL DPC++ API を使用したラッパーなどです。

複数のラッパーが必要ない場合は、`<domain>` と `<3rd-party library short name>` のみが必要です（例: `onemath_rng_curand.so`）。

1. ヘッダーファイルを作成

新しいバックエンド・ライブラリーごとに、次の 2 つのヘッダーファイルを作成する必要があります:

- 新しいサードパーティー・ライブラリー・ラッパーへのエントリーポイントの宣言を含むヘッダーファイル
- 新しいサードパーティー・ライブラリー用のコンパイル時ディスパッチ・インターフェイス（oneMath 使用モデルを参照）

サンプル・ヘッダー・ファイル `include/oneapi/math/blas/detail/newlib/onemath_blas_newlib.hpp`

```
namespace oneapi {
namespace math {
namespace newlib {

void asum(sycl::queue &queue, std::int64_t n, sycl::buffer<float, 1> &x, std::int64_t incx,
          sycl::buffer<float, 1> &result);
```

コンパイル時ディスパッチ・インターフェイスの例:

`include/oneapi/math/blas/detail/newlib/blas_ct.hpp` の `newlib` およびサポートされているデバイス `newdevice` のコンパイル時ディスパッチ・インターフェイス・テンプレートのインスタンス化の例。

```
namespace oneapi {
namespace math {
namespace blas {

template <>
void asum<library::newlib, backend::newdevice>(sycl::queue &queue, std::int64_t n,
                                               sycl::buffer<float, 1> &x, std::int64_t incx,
                                               sycl::buffer<float, 1> &result) {
    asum_precondition(queue, n, x, incx, result);
```

```
    oneapi::math::newlib::asum(queue, n, x, incx, result);
    asum_postcondition(queue, n, x, incx, result);
}
```

2. ヘッダーファイルを統合

以下に、oneMath の上位レベルのインクルード・ディレクトリー構造を示します:

```
include/
   oneapi/
      math/
         math.hpp -> oneMath spec APIs
         types.hpp  -> oneMath spec types
         blas.hpp   -> oneMath BLAS APIs w/ pre-check/dispatching/post-check
         detail/    -> implementation specific header files
            exceptions.hpp      -> oneMath exception classes
            backends.hpp        -> list of oneMath backends
            backends_table.hpp   -> table of backend libraries for each domain and device
            get_device_id.hpp    -> function to query device information from queue for Run-time
dispatching
         blas/
            predicates.hpp -> oneMath BLAS pre-check post-check
            detail/      -> BLAS domain specific implementation details
               blas_loader.hpp     -> oneMath Run-time BLAS API
               blas_ct_templates.hpp -> oneMath Compile-time BLAS API general templates
               cublas/
                  blas_ct.hpp         -> oneMath Compile-time BLAS API template instantiations for
<cublas>
                  onemath_blas_cublas.hpp -> backend wrappers library API
               mklcpu/
                  blas_ct.hpp         -> oneMath Compile-time BLAS API template instantiations for
<mklcpu>
                  onemath_blas_mklcpu.hpp -> backend wrappers library API
               <other backends>/
         <other domains>/
```

新しいサードパーティー・ライブラリーを oneMath ヘッダーベースに統合するには、次のファイルを更新する必要があります:

- include/oneapi/math/detail/backends.hpp: 新しいバックエンドを追加

  例: newbackend バックエンドを追加する

```
   enum class backend { mklcpu,
```

```
+                     newbackend,
   static backendmap backend_map = { { backend::mklcpu, "mklcpu" },
+                                     { backend::newbackend, "newbackend" },
```

- include/oneapi/math/detail/backends_table.hpp: サポートされているドメインとデバイス用の新しいバックエンド・ライブラリーを追加

  例: blas ドメインと newdevice デバイスに対して newlib を有効にする

```
   enum class device : uint16_t { x86cpu,
                                  ...
+                                 newdevice
                                };

   static std::map<domain, std::map<device, std::vector<const char*>>> libraries = {
       { domain::blas,
         { { device::x86cpu,
             {
   #ifdef ONEMATH_ENABLE_MKLCPU_BACKEND
                 LIB_NAME("blas_mklcpu")
   #endif
              } },
+          { device::newdevice,
+            {
+   #ifdef ONEMATH_ENABLE_NEWLIB_BACKEND
+                 LIB_NAME("blas_newlib")
+   #endif
+              } },
```

- include/oneapi/math/detail/get_device_id.hpp: 実行時ディスパッチ用の新しいデバイス検出メカニズムを追加

  例: キューがホストをターゲットとしている場合は、newdevice を有効にします

```
   inline oneapi::math::device get_device_id(sycl::queue &queue) {
       oneapi::math::device device_id;
+       if (queue.is_host())
+           device_id=device::newdevice;
```

- include/oneapi/math/blas.hpp: コンパイル時ディスパッチ・インターフェイス用に作成されたヘッダーファイルをインクルードします（oneMath 使用モデルを参照）。

例: 1. ヘッダーファイルを作成ステップで作成された include/oneapi/math/blas/detail/newlib/blas_ct.hpp を追加します。

```
   #include "oneapi/math/blas/detail/mklcpu/blas_ct.hpp"
   #include "oneapi/math/blas/detail/mklgpu/blas_ct.hpp"
+  #include "oneapi/math/blas/detail/newlib/blas_ct.hpp"
```

1. ヘッダーファイルを作成ステップで作成された新しいファイルにより、BLAS ドメイン・ヘッダー・ファイルの構造が次のように更新されます。

```
include/
    oneapi/dal.hpp
        math/
            blas.hpp -> oneMath BLAS APIs w/ pre-check/dispatching/post-check
            blas/
                predicates.hpp -> oneMath BLAS pre-check post-check
                detail/        -> BLAS domain specific implementation details
                    blas_loader.hpp       -> oneMath Run-time BLAS API
                    blas_ct_templates.hpp -> oneMath Compile-time BLAS API general templates
                    cublas/
                        blas_ct.hpp           -> oneMath Compile-time BLAS API template
instantiations for <cublas>
                        onemath_blas_cublas.hpp -> backend wrappers library API
                    mklcpu/
                        blas_ct.hpp           -> oneMath Compile-time BLAS API template
instantiations for <mklcpu>
                        onemath_blas_mklcpu.hpp -> backend wrappers library API
    +         newlib/
    +                blas_ct.hpp           -> oneMath Compile-time BLAS API template
instantiations for <newbackend>
    +                onemath_blas_newlib.hpp -> backend wrappers library API
                    <other backends>/
            <other domains>/
```

3. ラッパーを作成

ラッパーは、データ並列 C++（DPC++）入力データタイプをサードパーティー・ライブラリーのデータタイプに変換し、サードパーティー・ライブラリーの対応する実装を呼び出します。各サードパーティー・ライブラリーのラッパーは、個別の oneMath バックエンド・ライブラリーにビルドされます。libonemath.so ディスパッチャー・ライブラリーは、実行時ディスパッチ用のインターフェイスを使用している場合は実行時にラッパーをロードします。また、コンパイル時ディスパッチ用のインターフェイスを使用している場合は、ラッパーと直接リンクします（詳細については、oneMath 使用モデルを参照してください）。

```
src/
   include/
      function_table_initializer.hpp -> general loader implementation w/ global libraries table
   blas/
      function_table.hpp -> loaded BLAS functions declaration
      blas_loader.cpp -> BLAS wrappers for loader
      backends/
         cublas/ -> cuBLAS wrappers
         mklcpu/ -> Intel oneMKL CPU wrappers
         mklgpu/ -> Intel oneMKL GPU wrappers
         <other backend libraries>/
   <other domains>/
```

各バックエンド・ライブラリーには、選択したドメインのすべての関数のテーブルが含まれている必要があります。

例: 既に作成および統合されたヘッダーファイルに基づいて `newlib` のラッパーを作成し、1 つの `asum` 関数のみを有効にします

2 つの新しいファイルを作成します:

- `src/blas/backends/newlib/newlib_wrappers.cpp` - `include/oneapi/math/blas/detail/newlib/onemath_blas_newlib.hpp` のすべての関数の DPC++ ラッパー
- `src/blas/backends/newlib/newlib_wrappers_table_dyn.cpp` - ランタイム・ディスパッチャーのシンボルの構造（ラッパーと同じ場所）、サフィックス `_dyn` は、このファイルが動的ライブラリーにのみ必要であることを示します。

次に、`src/blas/backends/newlib/newlib_wrappers.cpp` を変更して、サードパーティー・ライブラリー `libnewlib.so` の C 関数 `newlib_sasum` を有効にできます。

この関数を有効にするには、次の操作を行います:

- `newlib_sasum` 関数宣言を含むヘッダーファイル `newlib.h` をインクルードします
- すべての DPC++ パラメーターを適切な C タイプに変換します。入力および出力 DPC++ バッファーの `get_access` メソッドを使用して行ポインターを取得します
- C 関数呼び出しを含む DPC++ カーネルを `single_task` として `newlib` に送信します

次のコード例は、`src/blas/backends/newlib/newlib_wrappers.cpp` に対して更新されます:

```
   #if __has_include(<sycl/sycl.hpp>)
   #include <sycl/sycl.hpp>
   #else
   #include <CL/sycl.hpp>
   #endif


   #include "oneapi/math/types.hpp"
```

```
    #include "oneapi/math/blas/detail/newlib/onemath_blas_newlib.hpp"
+
+    #include "newlib.h"

    namespace oneapi {
    namespace math {
    namespace newlib {

    void asum(sycl::queue &queue, std::int64_t n, sycl::buffer<float, 1> &x, std::int64_t incx,
              sycl::buffer<float, 1> &result) {
-       throw std::runtime_error("Not implemented for newlib");
+       queue.submit([&](sycl::handler &cgh) {
+           auto accessor_x      = x.get_access<sycl::access::mode::read>(cgh);
+           auto accessor_result = result.get_access<sycl::access::mode::write>(cgh);
+           cgh.single_task<class newlib_sasum>([=]() {
+               accessor_result[0] = ::newlib_sasum((const int)n, accessor_x.get_pointer(), (const
int)incx);
+           });
+       });
    }

    void asum(sycl::queue &queue, std::int64_t n, sycl::buffer<double, 1> &x, std::int64_t incx,
              sycl::buffer<double, 1> &result) {
        throw std::runtime_error("Not implemented for newlib");
    }
Updated structure of the src folder with the newlib wrappers:
src/
    blas/
        loader.hpp -> general loader implementation w/ global libraries table
        function_table.hpp -> loaded BLAS functions declaration
        blas_loader.cpp -> BLAS wrappers for loader
        backends/
            cublas/ -> cuBLAS wrappers
            mklcpu/ -> Intel oneMKL CPU wrappers
            mklgpu/ -> Intel oneMKL GPU wrappers
 +          newlib/
 +              newlib.h
 +              newlib_wrappers.cpp
 +              newlib_wrappers_table_dyn.cpp
            <other backend libraries>/
```

```
    <other domains>/
```

4. ラッパーをビルドシステムに統合

サードパーティー・ライブラリーの新しいラッパーを oneMath ビルドシステムに統合するのに作成/更新する必要があるファイルのリストは次のとおりです:

- 新しいサードパーティー・ライブラリーの新しいオプション `ENABLE_XXX_BACKEND` を `CMakeList.txt` ファイルの先頭に追加します。

  例: `CMakeList.txt` ファイルの先頭にある `newlib` の変更

```
    option(ENABLE_MKLCPU_BACKEND "" ON)
    option(ENABLE_MKLGPU_BACKEND "" ON)
+   option(ENABLE_NEWLIB_BACKEND "" ON)
```

- `ENABLE_XXX_BACKEND` 条件の下にある新しいサードパーティー・ライブラリーのラッパーを含む新しいディレクトリー（`src/<domain>/backends/<new_directory>`）を `src/<domain>/backends/CMakeList.txt` ファイルに追加します。

  例: `src/blas/backends/CMakeLists.txt` の `newlib` の変更

```
    if(ENABLE_MKLCPU_BACKEND)
        add_subdirectory(mklcpu)
    endif()
+
+   if(ENABLE_NEWLIB_BACKEND)
+       add_subdirectory(newlib)
+   endif()
```

新しいサードパーティー・ライブラリーとその依存関係を見つけるには、`cmake/FindXXX.cmake` cmake 構成ファイルを作成します。

例: `newlib` の新しい設定ファイル `cmake/FindNEWLIB.cmake`

```
include_guard()
# NEWLIB_ROOT cmake 変数または環境変数 NEWLIBROOT でライブラリー名を検索
find_library(NEWLIB_LIBRARY NAMES newlib
    HINTS ${NEWLIB_ROOT} $ENV{NEWLIBROOT}
    PATH_SUFFIXES "lib")
# ライブラリーが見つかったことを確認してください
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(NEWLIB REQUIRED_VARS NEWLIB_LIBRARY)
# ライブラリーの cmake ターゲットを設定
add_library(ONEMATH::NEWLIB::NEWLIB UNKNOWN IMPORTED)
```

```
set_target_properties(ONEMATH::NEWLIB::NEWLIB PROPERTIES
    IMPORTED_LOCATION ${NEWLIB_LIBRARY})
```

- 新しいサードパーティー・ライブラリーのバックエンド・レイヤーをビルドする方法を指定するには、`src/<domain>/backends/<new_directory>/CMakeList.txt` cmake 構成ファイルを作成します。

  既存のバックエンドを参照して、`cmake/FindXXX.cmake` ファイルを作成します。

  新しい `cmake/FindXXX.cmake` ファイルに関する情報とサードパーティー・ライブラリーとのリンク方法に関する指示を config ファイルに追加する必要があります。

  例: `src/blas/backends/newlib/CMakeLists.txt` ファイルを更新する

```
    # サードパーティーのライブラリーを追加
-   # find_package(XXX REQUIRED)
+   find_package(NEWLIB REQUIRED)
    target_link_libraries(${LIB_OBJ}
        PUBLIC ONEMATH::SYCL::SYCL
-       # Add third-party library to link with here
+       PUBLIC ONEMATH::NEWLIB::NEWLIB
    )
```

これで、`newlib` のバックエンド・ライブラリーをビルドして、サードパーティー・ライブラリーの統合が正常に完了したことを確認できます（詳細については、cmake でビルドを参照してください）

```
cd build/
cmake ..-DNEWLIB_ROOT=<path/to/newlib> ¥
    -DENABLE_MKLCPU_BACKEND=OFF ¥
    -DENABLE_MKLGPU_BACKEND=OFF ¥
    -DENABLE_NEWLIB_BACKEND=ON ¥          # 新しいサードパーティライブラリバックエンドを有効にする
    -DBUILD_FUNCTIONAL_TESTS=OFF          # このステップでは、ビルドのみを行います
cmake --build .-j4
```

5. テストシステムを更新

機能テスト用の新しいサードパーティー・ライブラリーを有効にするには、次のファイルを更新します:

- `src/config.hpp.in`: 新しいサードパーティー・ライブラリー用の cmake オプションを追加して、このマクロを機能テストに伝えるようにします

  例: `ENABLE_NEWLIB_BACKEND` を追加する

```
   #cmakedefine ONEMATH_ENABLE_MKLCPU_BACKEND
+  #cmakedefine ONEMATH_ENABLE_NEWLIB_BACKEND
```

- tests/unit_tests/CMakeLists.txt: 新しいバックエンド・ライブラリーにテストをリンクする手順を追加

  例: newlib バックエンド・ライブラリーを追加する

```
   if(ENABLE_MKLCPU_BACKEND)
       add_dependencies(test_main_ct onemath_blas_mklcpu)
       if(BUILD_SHARED_LIBS)
           list(APPEND ONEMATH_LIBRARIES onemath_blas_mklcpu)
       else()
           list(APPEND ONEMATH_LIBRARIES -foffload-static-
lib=${CMAKE_LIBRARY_OUTPUT_DIRECTORY}/libonemath_blas_mklcpu.a)
           find_package(MKL REQUIRED)
           list(APPEND ONEMATH_LIBRARIES ${MKL_LINK_C})
       endif()
   endif()
+
+    if(ENABLE_NEWLIB_BACKEND)
+       add_dependencies(test_main_ct onemath_blas_newlib)
+       if(BUILD_SHARED_LIBS)
+           list(APPEND ONEMATH_LIBRARIES onemath_blas_newlib)
+       else()
+           list(APPEND ONEMATH_LIBRARIES -foffload-static-
lib=${CMAKE_LIBRARY_OUTPUT_DIRECTORY}/libonemath_blas_newlib.a)
+           find_package(NEWLIB REQUIRED)
+           list(APPEND ONEMATH_LIBRARIES ONEMATH::NEWLIB::NEWLIB)
+       endif()
+   endif()
```

- tests/unit_tests/include/test_helper.hpp: 新しいバックエンドとのコンパイル時ディスパッチ・インターフェイスのヘルパー関数を追加し、呼び出すデバイスを指定します

  例: ホストの場合、newdevice を使用して newlib コンパイル時ディスパッチ・インターフェイスのヘルパー関数を追加します

```
   #ifdef ONEMATH_ENABLE_MKLGPU_BACKEND
       #define TEST_RUN_INTELGPU(q, func, args) ¥
           func<oneapi::math::backend::mklgpu> args
   #else
       #define TEST_RUN_INTELGPU(q, func, args)
   #endif
+
+  #ifdef ONEMATH_ENABLE_NEWLIB_BACKEND
```

```
+    #define TEST_RUN_NEWDEVICE(q, func, args) ¥
+        func<oneapi::math::backend::newbackend> args
+ #else
+     #define TEST_RUN_NEWDEVICE(q, func, args)
+ #endif
  #define TEST_RUN_CT(q, func, args)              ¥
      do {                                       ¥
+         if (q.is_host())                        ¥
+             TEST_RUN_NEWDEVICE(q, func, args);   ¥
```

- tests/unit_tests/main_test.cpp: ターゲットデバイスをテスト対象デバイスのベクトルに追加

  例: newlib のターゲットデバイス CPU を追加する

```
          }
      }
+
+ #ifdef ONEMATH_ENABLE_NEWLIB_BACKEND
+     devices.push_back(sycl::device(sycl::host_selector()));
+ #endif
```

これで、有効になっているサードパーティー・ライブラリーの機能テストをビルドして実行できるようになりました（詳細については、cmake を使用したビルドを参照してください）。

```
cd build/
cmake .. -DNEWLIB_ROOT=<path/to/newlib> ¥
    -DENABLE_MKLCPU_BACKEND=OFF ¥
    -DENABLE_MKLGPU_BACKEND=OFF ¥
    -DENABLE_NEWLIB_BACKEND=ON  ¥
    -DBUILD_FUNCTIONAL_TESTS=ON
cmake --build .-j4
ctest
```