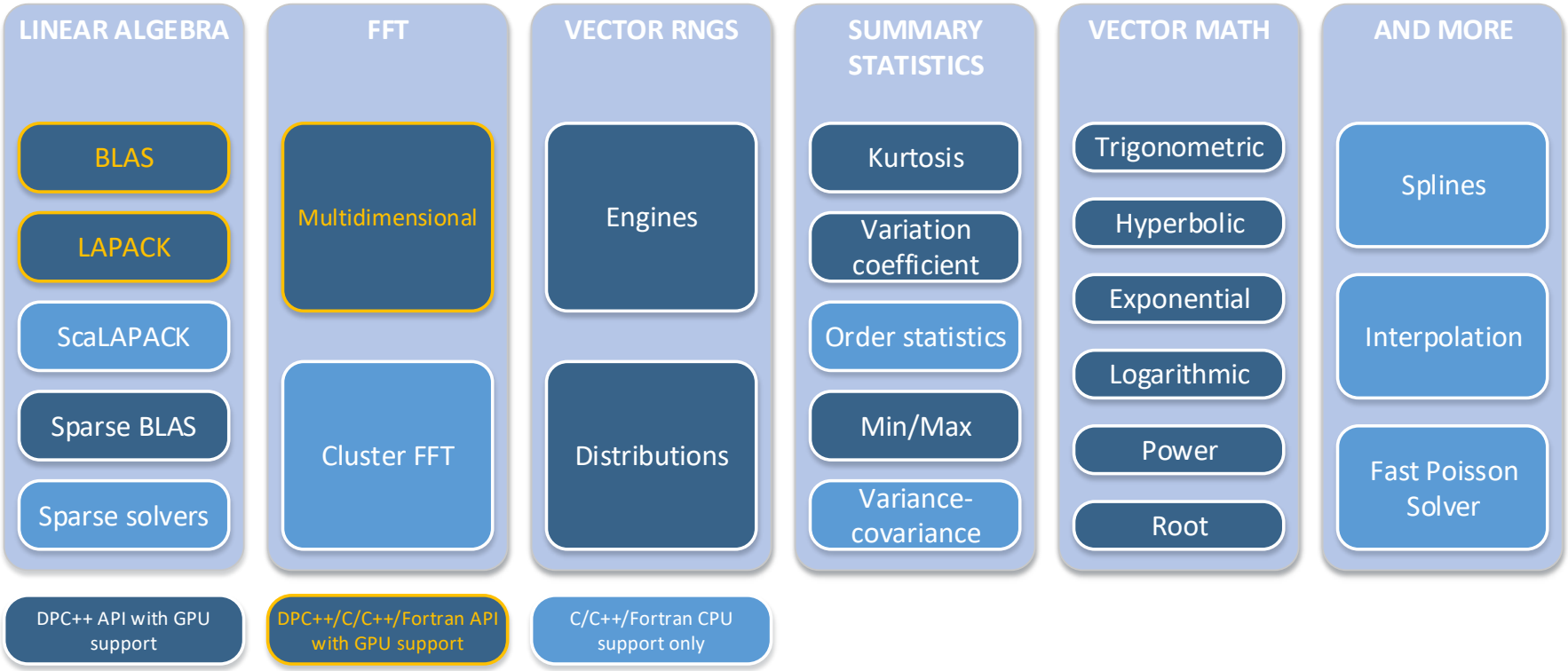
Aurora Early Adopters Series

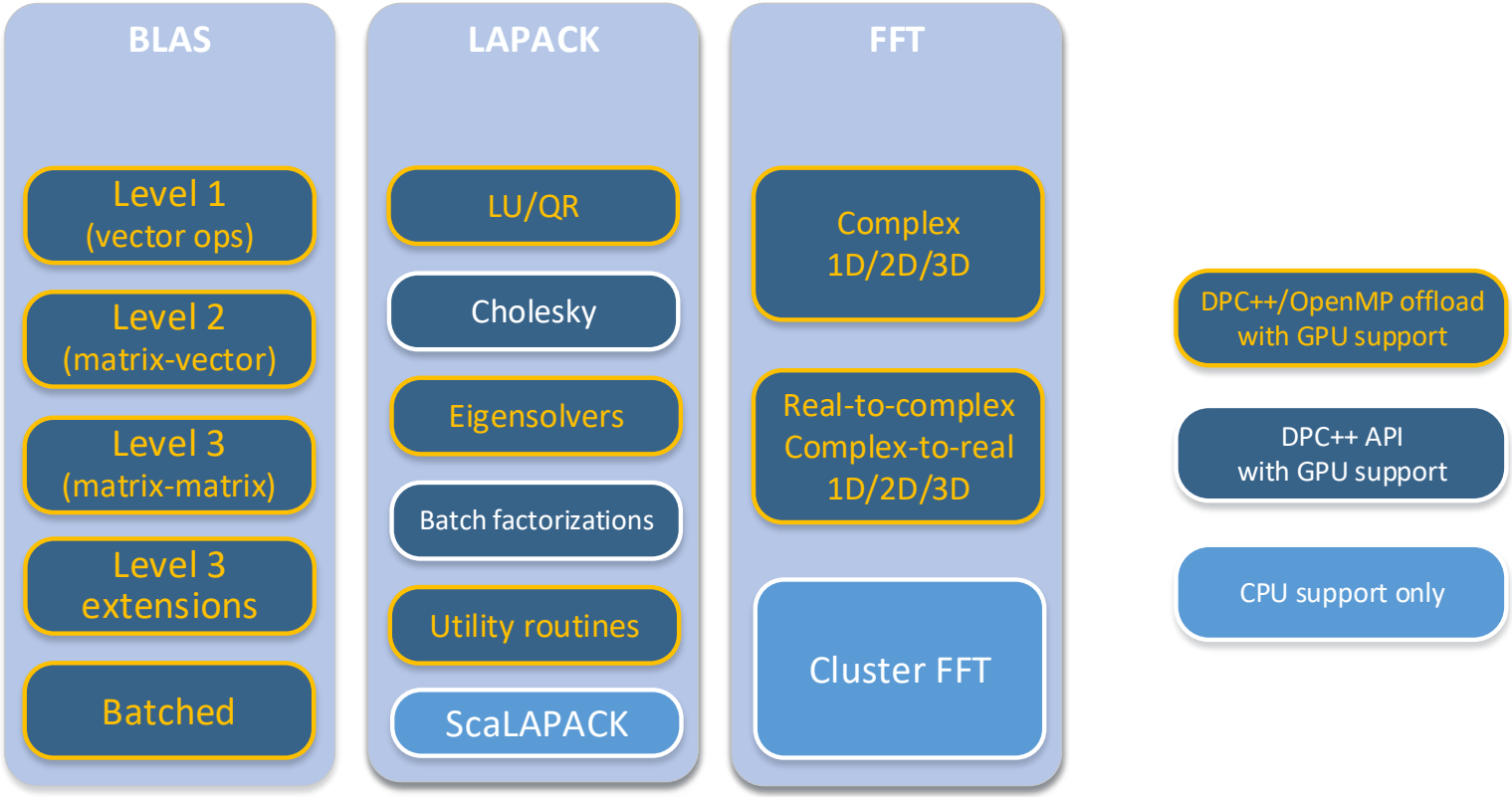
Overview of the Intel® oneAPI Math Kernel Library

Peter Caday

What's in Intel® oneMKL (Beta)?



Zoom in: Dense Linear Algebra + FFT



Potential GPU Usage Models

Example: multiply double-precision matrices C ← AB

- On host (A/B/C in host memory)
- Automatic offload to GPU (A/B/C on host)
- OpenMP offload (A/B/C on host or device)

Lots of CPU ↔ GPU transfer overhead!

- Manual offload (A/B/C on host or device)
- Inside device kernel (A/B/C on device)

```
dgemm(…, A, …, B, …, C, …);
```

```
dgemm(…, A, …, B, …, C, …);
```

```
#pragma omp target variant dispatch …
dgemm(…, A, …, B, …, C, …);
```

```
dgemm(device_queue, …, A, …, B, …, C, …);
```

```
void my_kernel(…) {
     dgemm(…, A, …, B, …, C, …);
}
```

oneMKL GPU Usage Models

	Automatic offload	OpenMP offload	Manual offload	Device API
Invocation side	CPU			GPU
Data location	CPU	GPU / CPU	GPU / CPU / shared	GPU
Interface	C/C++/Fortran	C/C++/Fortran + OpenMP	DPC++	DPC++
EOY support	None	Most oneMKL GPU functionality	All oneMKL GPU functionality	Limited support (selected RNG)

Using oneMKL OpenMP Offload Interfaces

Offload: Key OpenMP Directives (C)

```
#pragma omp target data
```

Map host-side variables to device variables inside this block.

```
#pragma omp target enter data
#pragma omp target exit data
```

Map/unmap host-side variables to device variables: the two halves of #pragma omp target data.

```
#pragma omp target
```

Offload execution of block to the GPU.

```
#pragma omp target variant dispatch
```

Offload oneMKL calls inside this block to the GPU.

GEMM with oneMKL C API

$$C \leftarrow \alpha AB + \beta C$$

```
int main() {
    long m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10;
    long sizea = lda * k, sizeb = ldb * n, sizec = ldc * n;
    double alpha = 1.0, beta = 0.0;

    // Allocate matrices
    double *A = (double *) mkl_malloc(sizeof(double) * sizea, 64);
    double *B = (double *) mkl_malloc(sizeof(double) * sizeb, 64);
    double *C = (double *) mkl_malloc(sizeof(double) * sizec, 64);

    // Initialize matrices […]
    …



        // Compute C = A * B on CPU
        cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k,
                    alpha, A, lda, B, ldb, beta, C, ldc);


    …
}
```

GEMM with oneMKL C OpenMP Offload

```
int main() {
    long m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10;
    long sizea = lda * k, sizeb = ldb * n, sizec = ldc * n;
    double alpha = 1.0, beta = 0.0;

    // Allocate matrices
    double *A = (double *) mkl_malloc(sizeof(double) * sizea, 64);
    double *B = (double *) mkl_malloc(sizeof(double) * sizeb, 64);
    double *C = (double *) mkl_malloc(sizeof(double) * sizec, 64);

    // Initialize matrices […]
    …
#pragma omp target data map(to:A[0:sizea],B[0:sizeb]) map(tofrom:C[0:sizec])
    {
#pragma omp target variant dispatch use_device_ptr(A, B, C) nowait
        {
            // Compute C = A * B on GPU
            cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k,
                        alpha, A, lda, B, ldb, beta, C, ldc);
        }
    }
    …
}
```

$C \leftarrow \alpha AB + \beta C$

Use `target data map` to send matrices to the device

Use `target variant dispatch` to request GPU execution for cblas_dgemm

List mapped device pointers in the `use_device_ptr` clause

Optional `nowait` clause for asynchronous execution
Use `#pragma omp taskwait` for synchronization

GEMM with oneMKL Fortran OpenMP Offload

```
… // module files
include “mkl_omp_offload.f90”

program main
use mkl_blas_omp_offload

integer             :: m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10
integer             :: sizea = lda * k, sizeb = ldb * n, sizec = ldc * n
double              :: alpha = 1.0, beta = 0.0
double, allocatable :: A(:), B(:), C(:)

// Allocate matrices…
allocate(A(sizea))
…

// Initialize matrices…
…
!$omp target data map(to:A(1:sizea), B(1:sizeb)) map(tofrom:C(1:sizec))
!$omp target variant dispatch use_device_ptr(A, B, C) nowait

! Compute C = A * B on GPU
call dgemm(‘N’, ‘N’, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc)

!$omp end target variant dispatch
!$omp end target data
…
end program
```

Module for Fortran OpenMP offload

Use `target data map` to send matrices to the device

Use `target variant dispatch` to request GPU execution for `dgemm`

List mapped device pointers in the `use_device_ptr` clause

Optional `nowait` clause for asynchronous execution
Use `!$omp taskwait` for synchronization

Using oneMKL DPC++ Interfaces

Data Parallel C++ (DPC++) Introduction

- **SYCL** is a C++-based, single-source programming language for heterogeneous computing.
- **DPC++** is SYCL + many new extensions.
  - *e.g.* pointer-based programming (Unified Shared Memory)
- Open, standards-based, multi-vendor.

https://software.intel.com/en-us/oneapi

DPC++: Key SYCL Constructs for oneMKL Usage

```
sycl::queue Q{sycl::cpu_selector{}};
sycl::queue Q{sycl::gpu_selector{}};
sycl::queue Q{device};
```

Create device queue attached to a given device or device type.

All device execution goes through a queue object.

```
void *mem = sycl::malloc_shared(bytes, Q);
void *mem = sycl::malloc_device(bytes, Q);
```

Allocate device-accessible memory. `malloc_shared` memory is also accessible from the host.

```
sycl::buffer<T,1> mem(elements);
sycl::buffer<T,1> mem(elements, hostptr);
```

Smart buffer object. Migrates memory automatically and tracks data dependencies.

Can be attached to host memory (synchronized at creation and destruction).

GEMM with oneMKL C API

$$C \leftarrow \alpha AB + \beta C$$

```
int main() {


    int64_t m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10;
    int64_t sizea = lda * k, sizeb = ldb * n, sizec = ldc * n;
    double alpha = 1.0, beta = 0.0;



    // Allocate matrices
    double *A = (double *) mkl_malloc(sizeof(double) * sizea);
    double *B = (double *) mkl_malloc(sizeof(double) * sizeb);
    double *C = (double *) mkl_malloc(sizeof(double) * sizec);

    // Initialize matrices […]
    …
    cblas_dgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, m, n, k,
                alpha, A, lda, B, ldb, beta, C, ldc);
    …
}
```

GEMM with oneMKL DPC++

```
int main() {
    using namespace oneapi::mkl;

    int64_t m = 10, n = 6, k = 8, lda = 12, ldb = 8, ldc = 10;
    int64_t sizea = lda * k, sizeb = ldb * n, sizec = ldc * n;
    double alpha = 1.0, beta = 0.0;

    sycl::queue Q{sycl::gpu_selector{}};

    // Allocate matrices
    double *A = malloc_shared<double>(sizea, Q);
    double *B = malloc_shared<double>(sizeb, Q);
    double *C = malloc_shared<double>(sizec, Q);

    // Initialize matrices […]
    …
    auto e = blas::gemm(Q, transpose::N, transpose::N, m, n, k,
                        alpha, A, lda, B, ldb, beta, C, ldc);
    …
}
```

$C \leftarrow \alpha AB + \beta C$

Set up GPU queue

Allocate CPU/GPU-accessible shared memory

New oneMKL DPC++ API

Computation is performed on given queue

Output **e** is a sycl::event object representing command completion

Call `e.wait()` to wait for completion

Batch Computations in oneMKL

Batching Overview

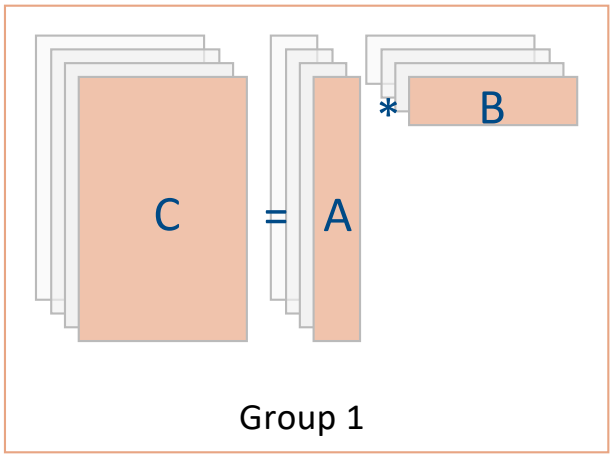
- Execute multiple **independent** operations of the same type in a single call
  - e.g. invert 100 different 8x8 matrices
- Benefits: increased parallelism, reduced overhead
- Available APIs:
  - BLAS: gemm, trsm, axpy
  - LAPACK*: LU (getrf, getri, getrs), Cholesky (potrf, potrs), QR (geqrf, orgqr, ungqr)
  - FFT: all DFTs

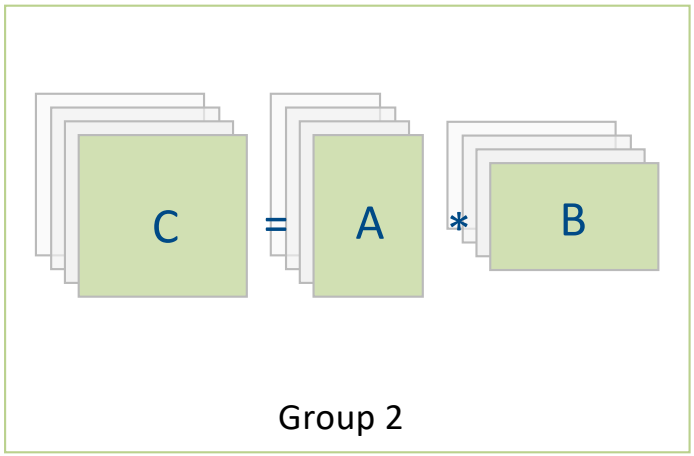
* only available in DPC++

BLAS/LAPACK Group APIs

Group = set of operations with identical parameters (size, transpose…)
but different matrix/vector data

Group batch APIs process one or more groups simultaneously.





BLAS/LAPACK Group APIs

Group = set of operations with identical parameters (size, transpose…)
but different matrix/vector data

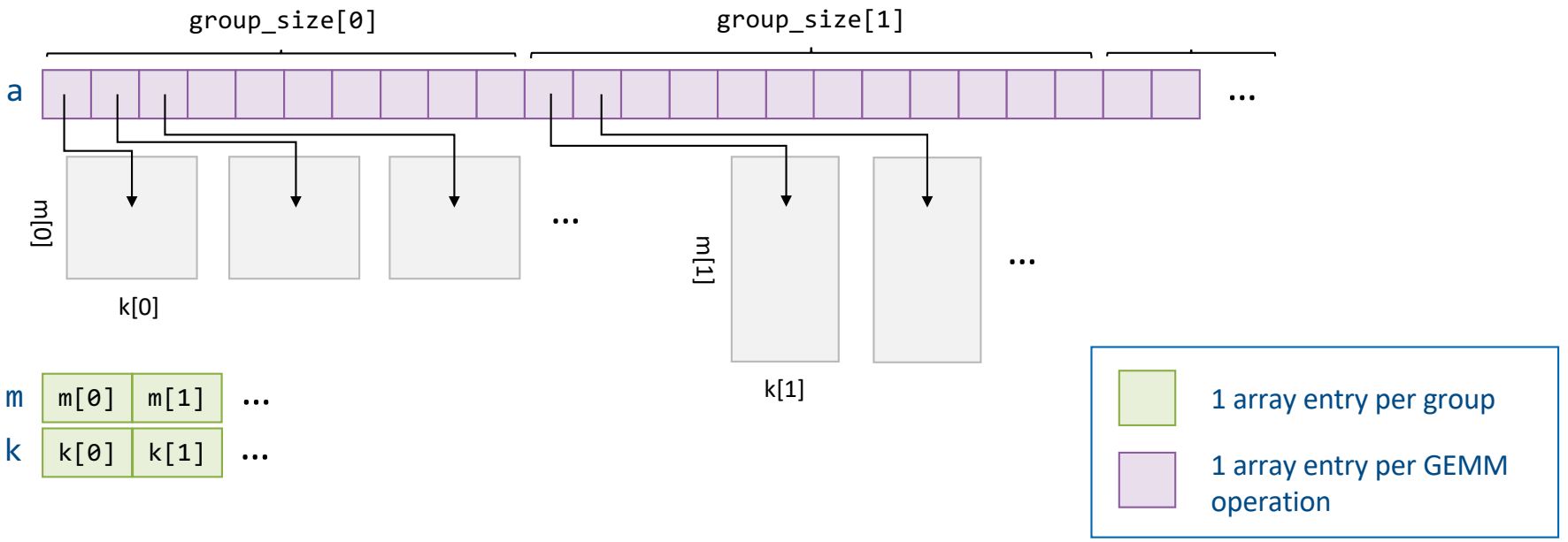
Group batch APIs process one or more groups simultaneously.

Examples:

- *n* operations, 1 group: all parameters identical
- *n* operations, *n* groups: each operation has different parameters

Example: Batch DGEMM

```
cblas_dgemm_batch(CblasColMajor, transA, transB, m, n, k, alpha, a, lda, b, ldb,
                  beta, c, ldc, num_groups, group_sizes);
```



BLAS/LAPACK Strided DPC++ APIs

- DPC++ oneMKL adds strided APIs for simple batch cases
  - **Single group**: all matrix/vector sizes, parameters are homogeneous
  - **Fixed stride** between successive matrices/vectors in batch
    - Base address + stride replaces array of pointers
    - Strides on inputs may be zero to reuse an input for all operations in the batch

Strided Batch Snippet - LU

```
#include “oneapi/mkl.hpp”
using namespace oneapi::mkl;

int64_t batch = 100;                  // 100 matrices
int64_t N = 10;                       // Matrix size – square in this example
int64_t stride = N * N;               // 10x10 matrices are contiguous in memory
int64_t stride_piv = N;               // Pivot entries also contiguous in memory

sycl::queue Q{sycl::gpu_selector{}};

// Allocate memory for matrices and pivot indices, as well as scratch space.
auto A_array      = sycl::malloc_shared<double>(stride * batch, Q);
auto pivot_array  = sycl::malloc_shared<double>(stride_piv * batch, Q);

auto scratch_size = lapack::getrf_batch_scratchpad_size(Q, n, n, n, stride, stride_piv, batch);
auto scratch      = sycl::malloc_shared<double>(scratch_size, Q);
…
// [Initialize A_array here]

// Batch computation
lapack::getrf_batch(Q, n, n, A_array, n, stride, pivot_array, stride_piv, scratch, scratch_size)
            .wait();
```

oneMKL Resources

Resources

Websites	https://software.intel.com/en-us/intel-mkl https://software.intel.com/en-us/oneapi/onemkl
Forum	https://software.intel.com/en-us/forums/intel-math-kernel-library
Developer Reference	https://software.intel.com/en-us/oneapi-mkl-dpcpp-developer-reference
Link Line Advisor	http://software.intel.com/en-us/articles/intel-mkl-link-line-advisor
Benchmarks	https://software.intel.com/en-us/intel-mkl/benchmarks

Notices & Disclaimers

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.

The benchmark results reported herein may need to be revised as additional testing is conducted. The results depend on the specific platform configurations and workloads utilized in the testing, and may not be applicable to any particular user's components, computer system or workloads. The results are not necessarily representative of other benchmarks and other benchmark results may show greater or lesser impact from mitigations.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

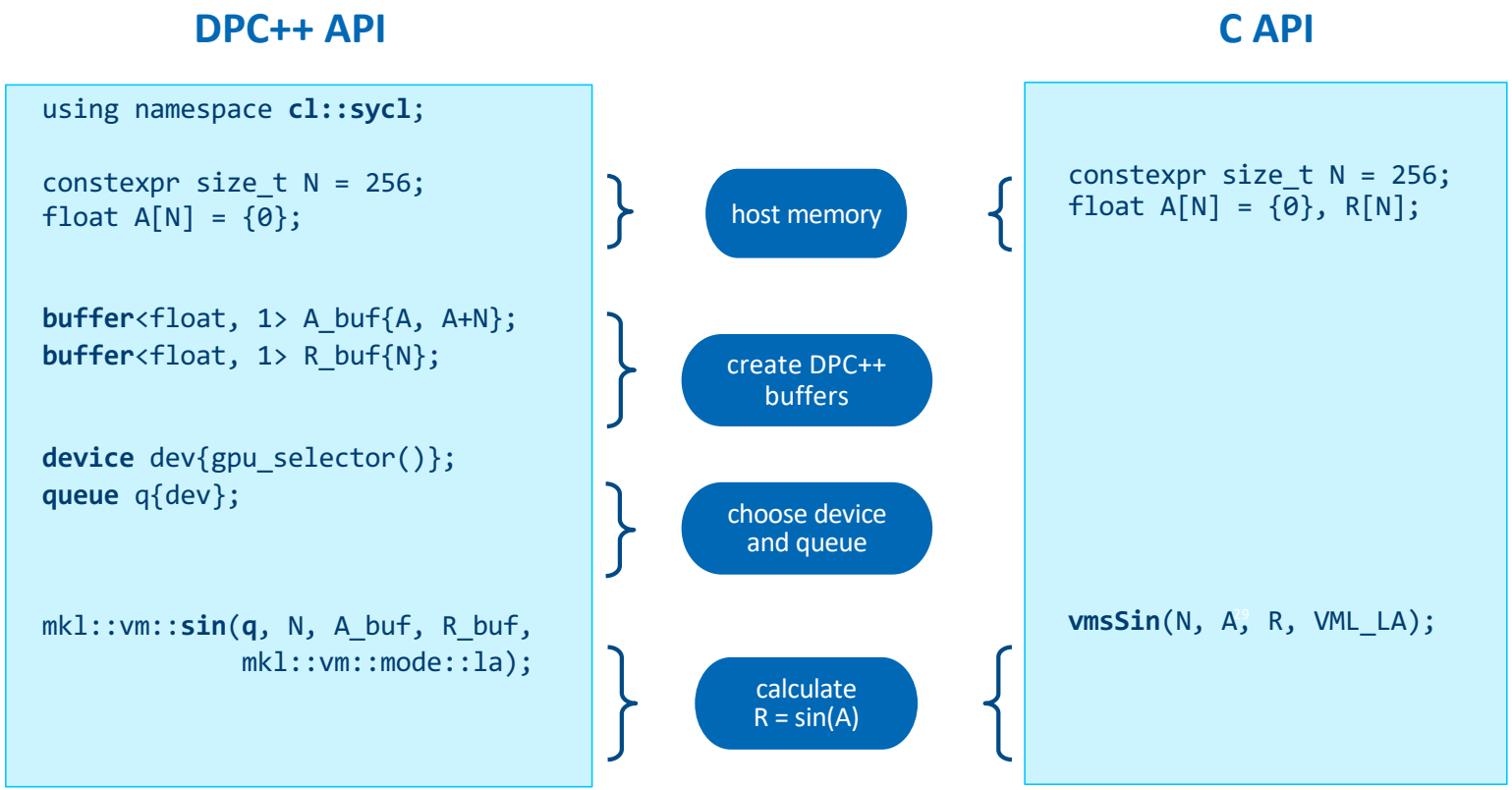
Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Questions & Answers

Backup

Example: Parallel Sine



Example: Multi-Device FFT (1 of 2)

DPC++ API

```
const int64_t len = 16;

mkl::dft::Descriptor<mkl::dft::Precision::SINGLE
,
                     mkl::dft::Domain::COMPLEX>
                         desc;
auto status = desc.init(len);

status = desc.setValue(mkl::dft::placement,
                       mkl::dft::not_inplace);


float *A = ...,
      *B = ...;


buffer<float, 1> A_buf{A, range<1>(2*len)};
buffer<float, 1> B_buf{B, range<1>(2*len)};
```

descriptor creation

host memory

DPC++ buffers

C API

```
M
KL_LONG dim = 1, len = 16;

DFTI_DESCRIPTOR_HANDLE hand = NULL;
MKL_LONG status = DftiCreateDescriptor(
    &hand, DFTI_SINGLE, DFTI_COMPLEX,
    dim, len);

status = DftiSetValue(hand,
    DFTI_PLACEMENT, DFTI_NOT_INPLACE);


float *A = ...,
      *B = ...;
```

Example: Multi-Device FFT (2 of 2)

DPC++ API

```
mkl::dft::Descriptor descCPU = desc;
mkl::dft::Descriptor descGPU = desc;

device devCPU{cpu_selector()},
       devGPU{gpu_selector()};
queue qCPU{devCPU},
      qGPU{devGPU};


status = descCPU.commit(qCPU);
status = descGPU.commit(qGPU);


status = descCPU.computeForward (A_buf, B_buf);
status = descGPU.computeBackward(B_buf, A_buf);


// Resources automatically freed when descriptor
//  goes out of scope.
```

devices and queues

precompu-tation

FFTs

cleanup

C API

```
status = DftiCommitDescriptor(hand);

status = DftiComputeForward(hand, A, B);

DftiFreeDescriptor(&hand);
```

Example: Matrix Multiplication (1 of 2)

Buffer API

```
using mkl::blas::gemm;
int64_t n = 32;

device dev({host, cpu, gpu}_selector());
queue Q(dev);

double *A = ..., *B = ..., *C = ...;

buffer<double, 1> A_buf{A, range<1>(n * n)},
                  B_buf{B, range<1>(n * n)},
                  C_buf{C, range<1>(n * n)};


gemm(Q, mkl::transpose::N, mkl::transpose::N,
        n, n, n, 1.0, A_buf, n, B_buf, n,
        0.0, C_buf, n);
```

device setup

prepare matrices

C = A * B

USM API

```
using mkl::blas::gemm;
int64_t n = 32;

device dev({host, cpu, gpu}_selector());
queue Q(dev);

size_t bytes = n * n * sizeof(double);
double *A = malloc_shared(bytes, dev,
                          Q.get_context());
double *B = malloc_shared(…);
double *C = malloc_shared(…);


gemm(Q, mkl::transpose::N, mkl::transpose::N,
     n, n, n, 1.0, A, n, B, n,
     0.0, C, n);

Q.wait_and_throw();
```

Example: Matrix Multiplication (2 of 2)

Buffer API

```
using mkl::blas::gemm;
int64_t n = 32;

device dev({host, cpu, gpu}_selector());
queue Q(dev);

double *A = ..., *B = ..., *C = ...;

buffer<double, 1> A_buf{A, range<1>(n * n)},
                  B_buf{B, range<1>(n * n)},
                  C_buf{C, range<1>(n * n)},
                  D_buf{D, range<1>(n * n)};

gemm(Q, mkl::transpose::N, mkl::transpose::N,
        n, n, n, 1.0, A_buf, n, B_buf, n,
        0.0, C_buf, n);

gemm(Q, mkl::transpose::N, mkl::transpose::N,
        n, n, n, 1.0, C_buf, n, A_buf, n,
        0.0, D_buf, n);
```

device setup

prepare matrices

C = A * B

D = C * A

USM API

```
using mkl::blas::gemm;
int64_t n = 32;

device dev({host, cpu, gpu}_selector());
queue Q(dev);

size_t bytes = n * n * sizeof(double);
double *A = malloc_shared(bytes, dev,
                          Q.get_context());
double *B = malloc_shared(…);
double *C = malloc_shared(…);
double *D = malloc_shared(…);

event e = gemm(Q, mkl::transpose::N, mkl::transpose::N,
               n, n, n, 1.0, A, n, B, n,
               0.0, C, n);

gemm(Q, mkl::transpose::N, mkl::transpose::N,
        n, n, n, 1.0, C, n, A, n,
        0.0, D, n, {e});
```

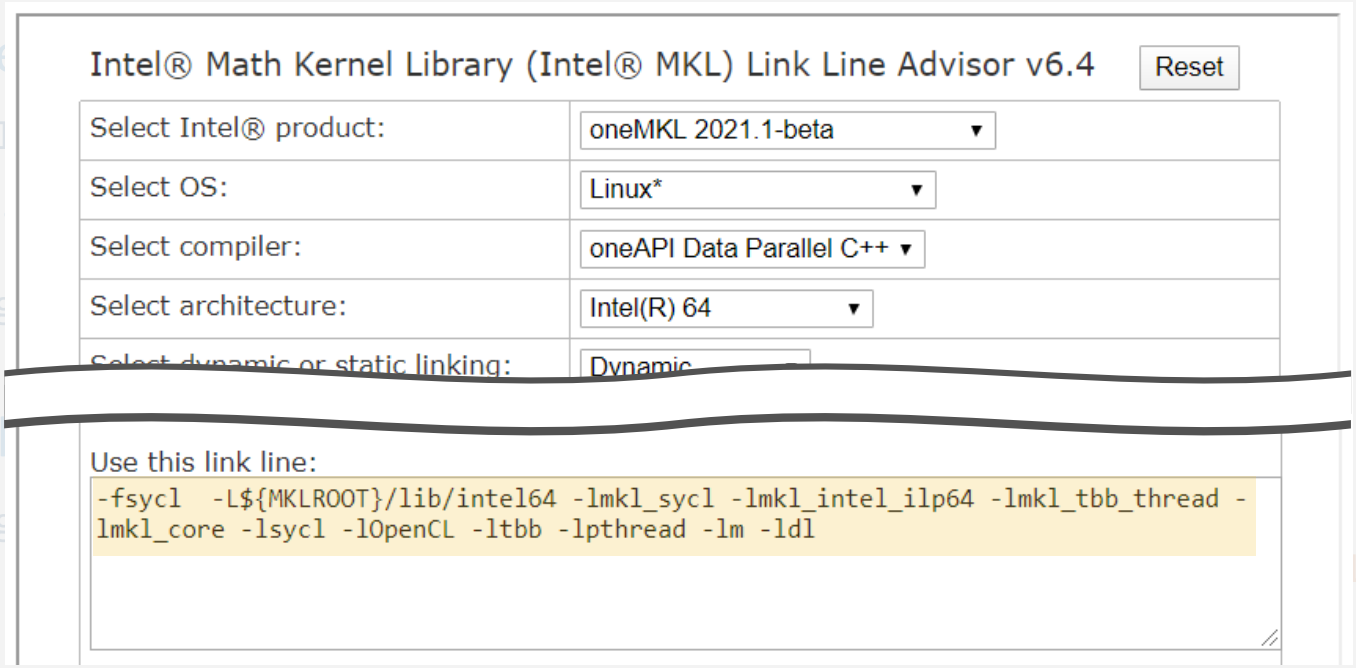
Error Handling

```
try {
    mkl::blas::gemm(Q, …);
}
catch (cl::sycl::exception &e) {
    // SYCL errors
    //    e.g. incorrect USM pointer provided
}
```


34

Building and Linking with oneMKL (Linux)

- Heade
- #incl
- Build
- clang
- Link fi
- clang
- Reference: https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor



https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor

Building and Linking with oneMKL (Windows)

- Headers

```
#include “mkl_sycl.hpp”
```

- Build flags (new flags **highlighted**)

```
clang++ -fsycl –I“%MKLROOT”\include -DMKL_ILP64
        -c source.cpp -o source.o
```

- Link flags (example: static link; **sequential** threading)

```
clang++ -fsycl -foffload-static-lib=“%MKLROOT%”\lib\intel64\mkl_sycl.lib
        -static -lmkl_intel_ilp64.lib -lmkl_sequential.lib -lmkl_core.lib
        -lsycl -lOpenCL
```

- Reference: https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor

Batch API example – C API

- Two groups of GEMM calls:

```
#include “mkl.h”
int group_count = 2;
// Create arrays of size group_count to store GEMM arguments
CBLAS_TRANSPOSE transA[] = {CblasNoTrans, CblasNoTrans};
CBLAS_TRANSPOSE transB[] = {CblasTrans, CblasNoTrans};
MKL_INT   m[] = {4, 3};
MKL_INT   k[] = {4, 6};
MKL_INT   n[] = {8, 3};
MKL_INT   lda[] = {4, 6};
MKL_INT   ldb[] = {4, 6};
MKL_INT   ldc[] = {8, 3};
double    alpha[] = {1.0, 1.0};
double    beta[] = {0.0, 2.0};
MKL_INT   size_per_grp[] = {20, 30};

// Call cblas_dgemm_batch to perform 50 GEMM operations
cblas_dgemm_batch(CblasRowMajor, transA, transB, m, n, k,
                  alpha, a_array, lda, b_array, ldb,
                  beta, c_array, ldc, group_count, size_per_group);
```