Experience From Ginkgo Porting to the SYCL Ecosystem

Yu-Hsiang Tsai[1], Terry Cojean[1], Tobias Ribizel[1], Hartwig Anzt[2,1]

[1]Karlsruhe Institute of Technology
[2]Innovative Computing Laboratory, University of Tennessee

Ginkgo: high-performance open-source cross-platform C++ sparse linear algebra library

We support several sparse linear algebra components
like different formats: Coo, Csr, Ell, Sellp, Hybrid…
krylov solvers: CG, BiCG, BiCGStab, GMRES, IDR…
preconditioners: BlockJacobi, ParILU/IC, ISAI
batch functionalities and GPU-resident direct solver

https://ginkgo-project.github.io

Ginkgo

Cross-platform of Ginkgo for easy use

```
#include <ginkgo/ginkgo.hpp>
#include <iostream>

int main()
{
    // Instantiate a GPU executor
    auto gpu =
-       gko::CudaExecutor::create(0, gko::OmpExecutor::create());
+       gko::DpcppExecutor::create(0, gko::ReferenceExecutor::create());
    // Read data
    auto A = gko::read<gko::matrix::Csr<>>(std::cin, gpu);
    auto b = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
    auto x = gko::read<gko::matrix::Dense<>>(std::cin, gpu);
    // Create the solver
    auto solver =
        gko::solver::Cg<>::build()
            .with_preconditioner(gko::preconditioner::Jacobi<>::build().on(gpu))
            .with_criteria(
                gko::stop::Iteration::build().with_max_iters(1000u).on(gpu),
                gko::stop::ResidualNormReduction<>::build()
                    .with_reduction_factor(1e-15)
                    .on(gpu))
            .on(gpu);
    // Solve system
    solver->generate(give(A))->apply(lend(b), lend(x));
    // Write result
    write(std::cout, lend(x));
}       You, 3 minutes ago • Uncommitted changes
```

Ginkgo provides the same interface but with native device language

Library core contains architecture-agnostic algorithm implementation;

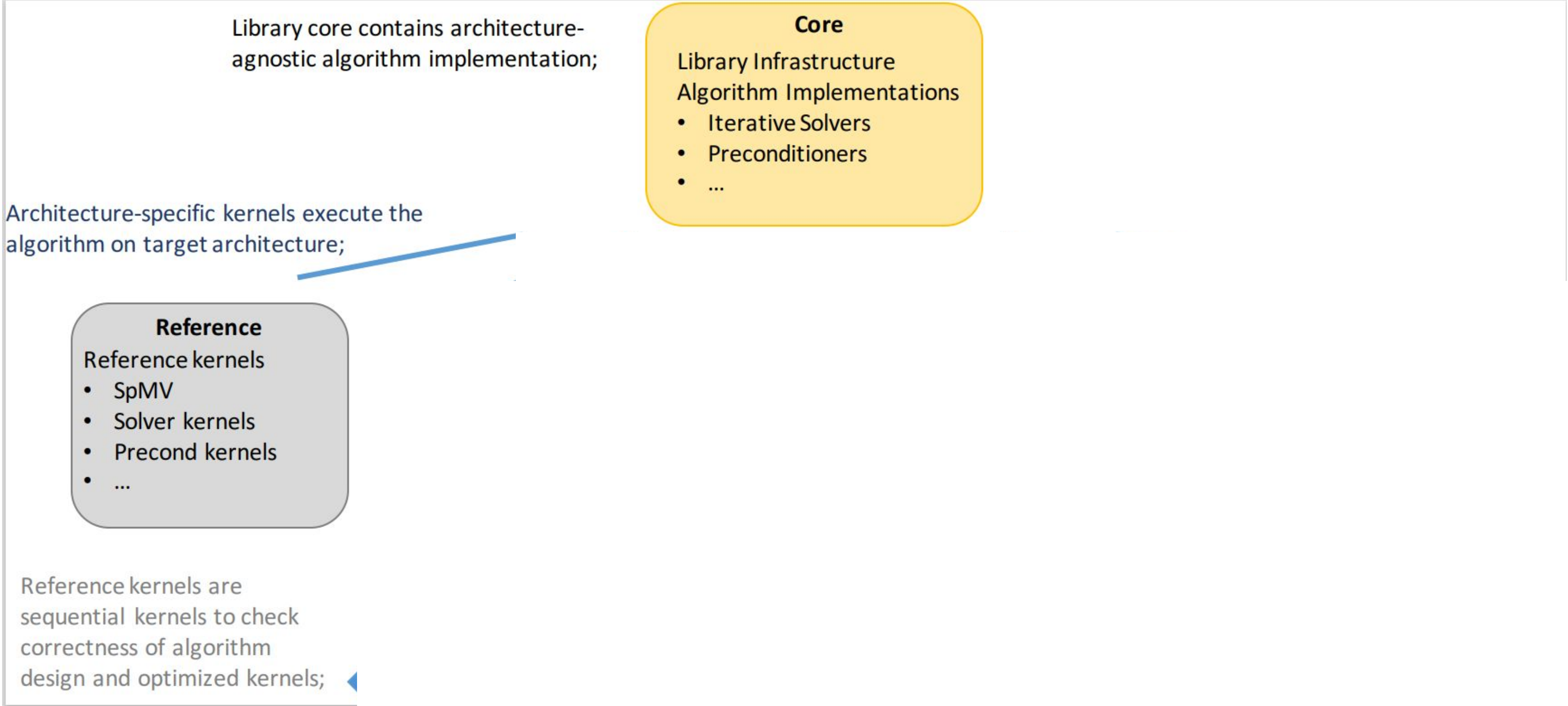
Core

Library Infrastructure

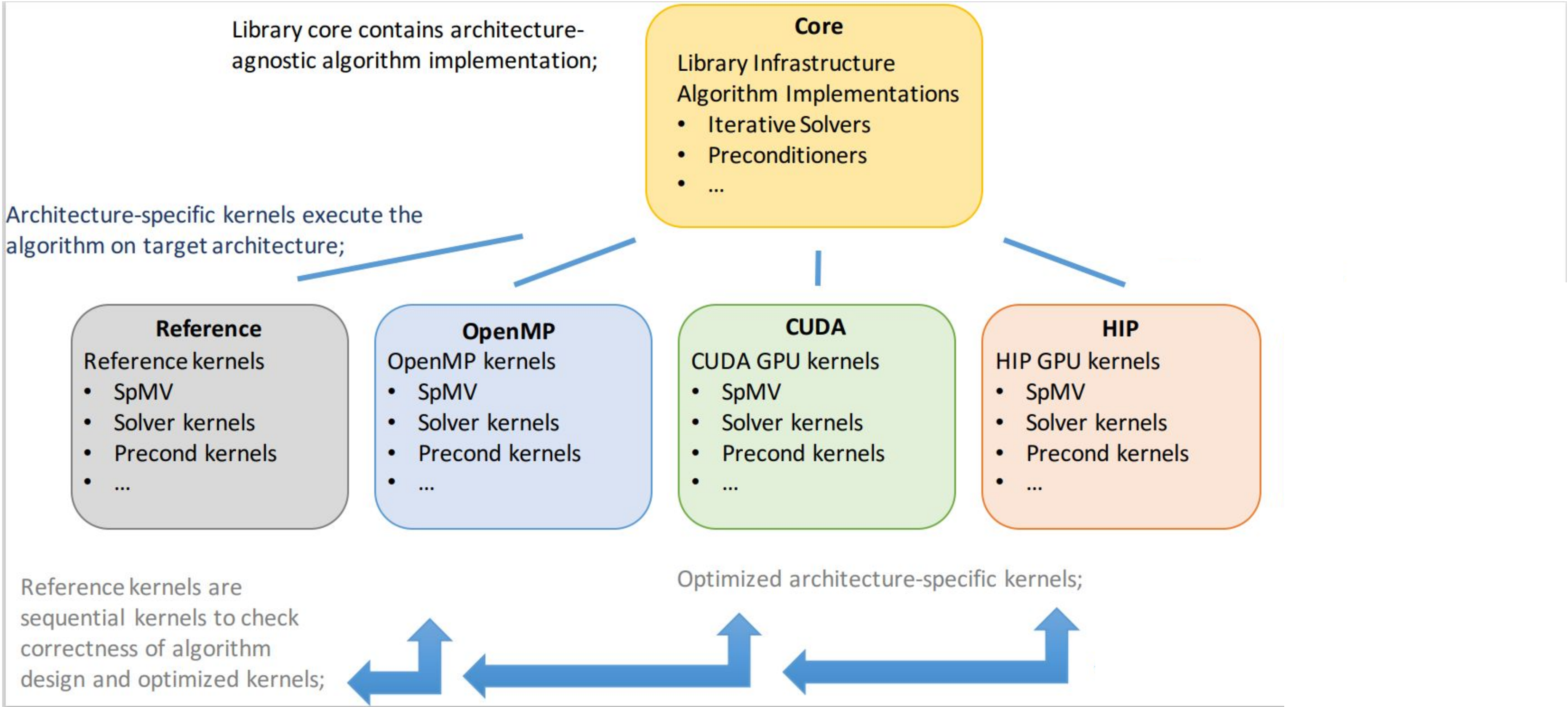
Algorithm Implementations

- Iterative Solvers
- Preconditioners
- ...

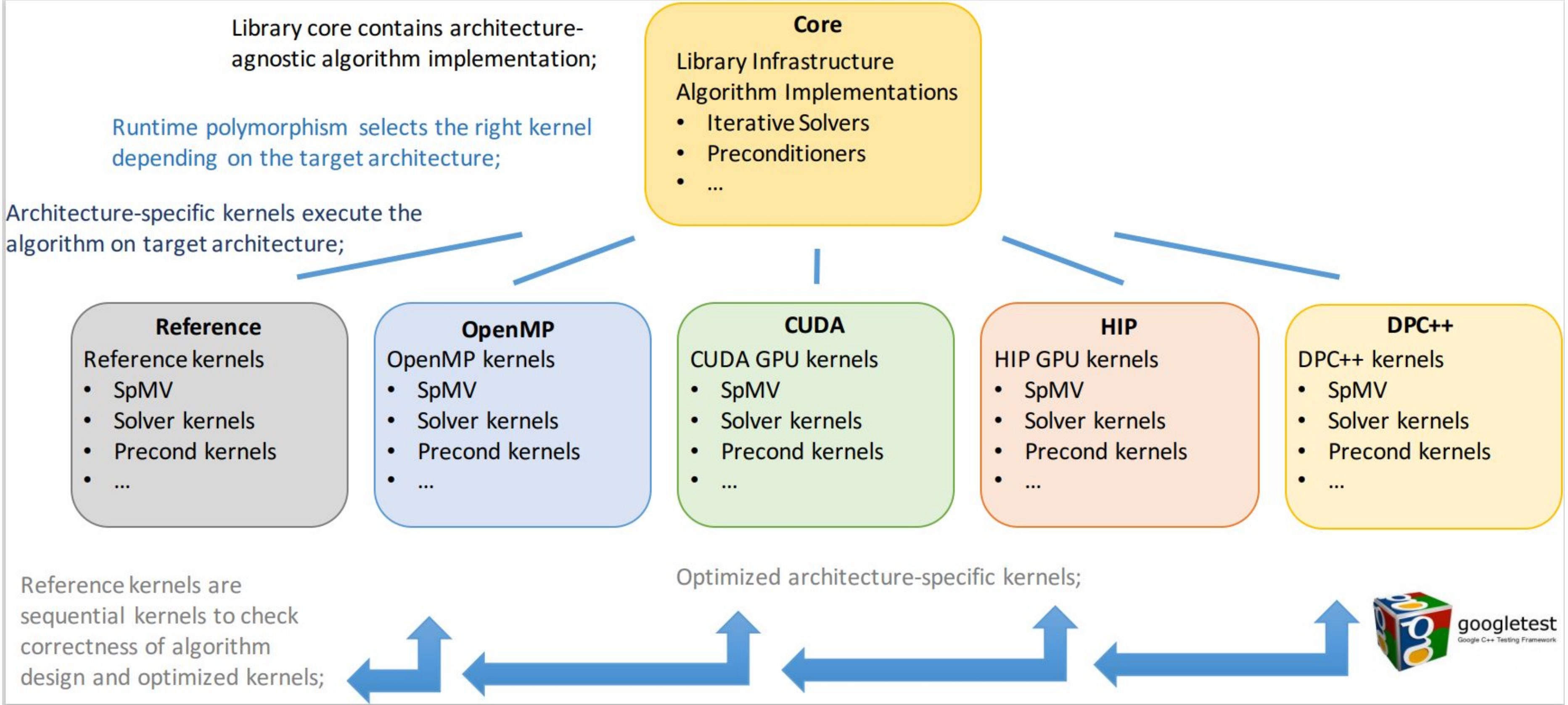
Ginkgo provides the same interface but with native device language



Ginkgo provides the same interface but with native device language



Ginkgo provides the same interface but with native device language



Reduction kernel (CUDA)

```
template <typename ValueType, int subwarp_size = 16>
__global__ void reduction_kernel(const int num, ValueType* val)
{
    auto thread_block = cooperative_groups::this_thread_block();
    auto subwarp =
        cooperative_groups::tiled_partition<subwarp_size>(thread_block);
    // block_size only be 64
    __shared__ ValueType tmp[64];
    auto tid = threadIdx.x;
    auto local_data = val[tid];
#pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    tmp[tid] = local_data;
    __syncthreads();
    if (tid < 32) {
        tmp[tid] += tmp[tid + 32];
    }
    __syncthreads();
    val[tid] = tmp[tid];
}
```

DPCT result

it does not complain but it is not supported

```
template <typename ValueType, int subwarp_size = 16>
void reduction_kernel(const int num, ValueType* val, sycl::nd_item<3> item_ct1,
                      ValueType *tmp)
{
    auto thread_block = item_ct1.get_group();
    auto subwarp =
        cooperative_groups::tiled_partition<subwarp_size>(thread_block);
    // block_size only be 64

    auto tid = item_ct1.get_local_id(2);
    auto local_data = val[tid];
#pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    tmp[tid] = local_data;
    /*
    DPCT1065:0: Consider replacing sycl::nd_item::barrier() with
    sycl::nd_item::barrier(sycl::access::fence_space::local_space) for better
    performance if there is no access to global memory.
    */
    item_ct1.barrier();
    if (tid < 32) {
        tmp[tid] += tmp[tid + 32];
    }
    /*
    DPCT1065:1: Consider replacing sycl::nd_item::barrier() with
    sycl::nd_item::barrier(sycl::access::fence_space::local_space) for better
    performance if there is no access to global memory.
    */
    item_ct1.barrier();
    val[tid] = tmp[tid];
}
```

Difficulties

DPCT will go over all “local” files related to the target files and try converting them

- it may be stuck and lead the failed conversion
- converted code all at once gives headache for review
- DPCT does not handle the failure automatically when we provide the equalivent codes in SYCL
- DPCT uses nd_item<3>, allocate all shared_memory like dynamic allocation

Isolate the code

We extract the neccessary code to outside

treat all other files as the system library such that dpct does not check them

DPCT handles the file in two ways

- “local” file: it automatically traverse all files in the current folder or subfolder
    - it will go through and convert the header file.
- “system” file: the files out of current folder
    - it only checks the interface, but will not check the implementation
    - will not perform any conversion on these file

New Issue

If the function is in local file, dpct adds the item_ct1 for us if function needs.
If the function is in system file, dpct does not add the item_ct1

When we put all other files as “system” file, DPCT will not pass the nd_item for the thread index information.

- If DPCT can convert it without any issue, leave it as local files
- If DPCT can not convert it, we combine the local and system file to provide a fake interface (and then we can deal it with our own ported codes)

Fake Interface - the bridge

local: dpct converts the code and knows how to add the corresponding entries from SYCL but maybe fails.
system: dpct does not converts the code and only check the interface

DPCT uses the text matching. When the function has the same name as cuda, we may need to change the function name to avoid dpct change them directly.

Fake Interface - the bridge

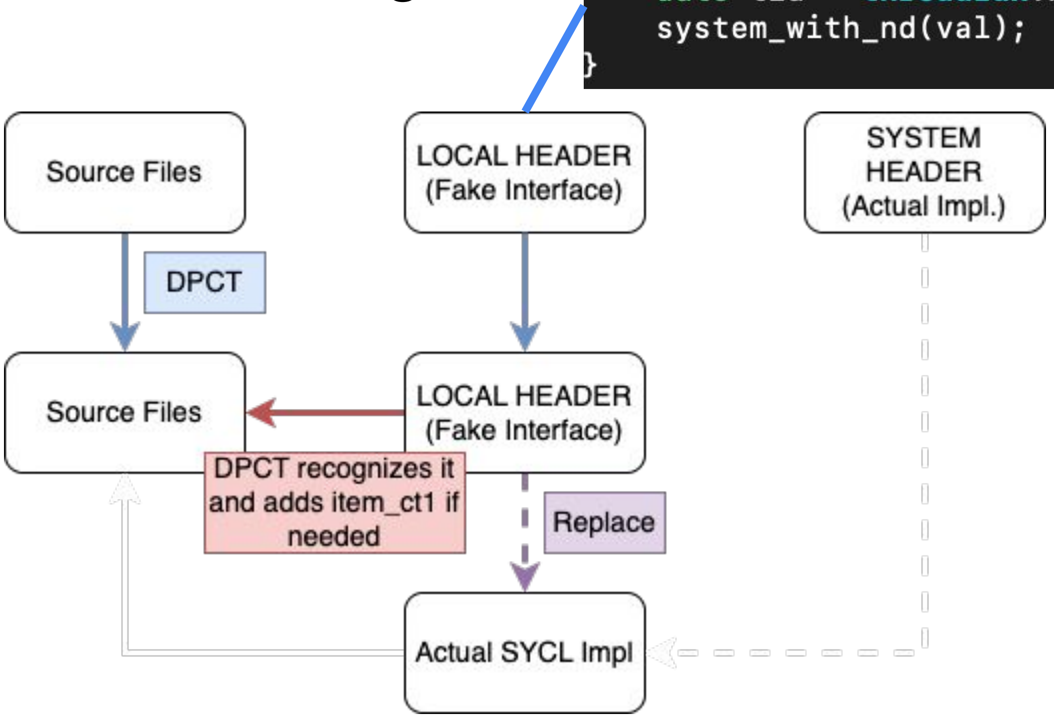
We can provide an additional fake interface which provides the same interface as original implementation.

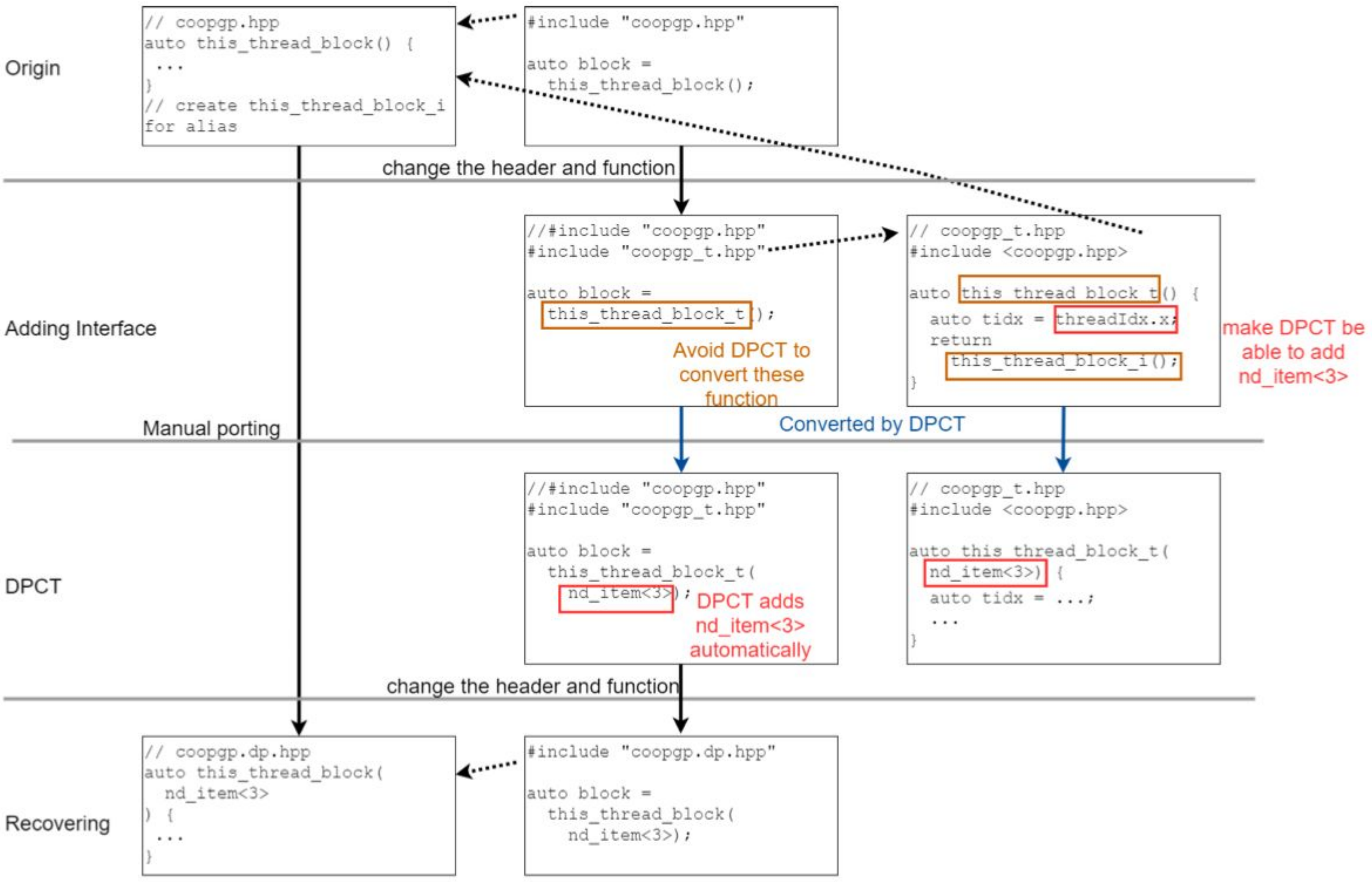
The fake interface only contains a **trick:** `auto tid = threadIdx.x` if it needs nd_item and then pass all arguments to the real implementation.

What dpct see these files?
First, it will see the fake_interface as local file, so it will try to convert the code, which add the nd_item for us due to the trick.
Then, the real implementation is as system file, so dpct will only check the original call is matching without touching the code. In the end, we just need to replace the call with our own sycl code.

Fake Interface - the bridge

```
__device__ void local_with_nd(float* val) {
    auto tid = threadIdx.x;
    system_with_nd(val);
}
```




Origin
Adding Interface
DPCT
Recovering
// coopgp.hpp
auto this_thread_block() {
...
}
// create this_thread_block_i for alias
#include "coopgp.hpp"
auto block = this_thread_block();
change the header and function
//#include "coopgp.hpp"
#include "coopgp_t.hpp"
auto block = this_thread_block_t();
Avoid DPCT to convert these function
// coopgp_t.hpp
#include <coopgp.hpp>
auto this_thread_block_t() {
auto tidx = threadIdx.x;
return this_thread_block_i();
}
make DPCT be able to add nd_item<3>
Manual porting
Converted by DPCT
//#include "coopgp.hpp"
#include "coopgp_t.hpp"
auto block = this_thread_block_t(nd_item<3>);
DPCT adds nd_item<3> automatically
// coopgp_t.hpp
#include <coopgp.hpp>
auto this_thread_block_t(nd_item<3>) {
auto tidx = ...;
...
}
change the header and function
// coopgp.dp.hpp
auto this_thread_block(nd_item<3>) {
...
}
#include "coopgp.dp.hpp"
auto block = this_thread_block(nd_item<3>);

Kernel Submission

```
reduction_kernel<<<1, 64>>>(64, d_A);
```

static shared memory out of function

```
q_ct1.submit([&](sycl::handler& cgh) {
    sycl::accessor<float, 1, sycl::access_mode::read_write,
                   sycl::access::target::local>
        tmp_acc_ct1(sycl::range<1>(64), cgh);

    cgh.parallel_for(sycl::nd_range<3>(sycl::range<3>(1, 1, 64),
                                       sycl::range<3>(1, 1, 64)),
                     [=](sycl::nd_item<3> item_ct1) {
                         reduction_kernel(
                             64, d_A, item_ct1,
                             (float*)tmp_acc_ct1.get_pointer());
                     });
});
```

different favor of the index

SYCL index vs dim3

SYCL handle index in different way than cuda dim3

dim3(x) -> threadId is contiguous along with threadIdx.x (x-axis)
dim3(x, y) -> threadId is contiguous along with threadIdx.x then threadIdx.y
dim3(x, y, z) -> same: threadId always follows fixed(first) axis first

sycl_range(u) -> threadId is contiguous along with get_local_id(0) (u-axis)
sycl_range(u, v) -> threadId is contiguous along with get_local_id(1) (v-axis)
then get_local_id(0) (u-axis)
sycl_range(u, v, w) -> threadId always follows the last axis first.

It’s required to change get_local_id index if adding a new axis

Provide dim3 for SYCL

For example, dim3(x, [y, z]) from cuda is converted to sycl_range<3>(z, y, x)

dim3(x) -> sycl_range<3>(1, 1, x)

…

dim3(x, y, z) -> sycl_range<3>(z, y, x)

we can provide our own dim3 for sycl such that we can still use cuda-like way in SYCL

Fake interface works on not only kernels but also host-cuda functions

additional host interface

dpct currently still port the static shared memory to the dynamic way

(it may be good when the static memory allocation is stable)

Also, sycl submit the range and corresponding lambda code.

Using the additional host interface, we can give the similar interface as cuda.

With the sycl dim3,

`kernel(dim3 grid, dim3 block, dynamic shared, queue, args)`

device_code_split

DPC++ provides different way to split the device code: **`per_kernel`** or **`per_source`**

```
template <typename VT>
void kernel_1(){}

template <typename VT>
void kernel_2(){}
```

device_code

: dependency or kernel essential components

device_code_split

DPC++ provides different way to split the device code: **`per_kernel`** or **`per_source`**

```
template <typename VT>
void kernel_1(){}

template <typename VT>
void kernel_2(){}
```

device_code

```
kernel_1<double>(){}
```

```
kernel_1<float>(){}
```

```
kernel_2<int>(){}
```

○

○

○

```
kernel_2<int64>(){}
```

`per_kernel`

: dependency or kernel essential components

per_kernel

Using **`per_kernel`** will make each kernel instantiation in different units.

Pro:
- Putting invalid kernel is okay.
- JIT compilation time comes with its own kernel only, so its JIT relatively faster than **`per_source`**

Con:
- Give more complexity to the compiler because each instantiation needs to be complete
- Compilation time is long
- Dependency duplication
- It will make the library big especially for debug build.

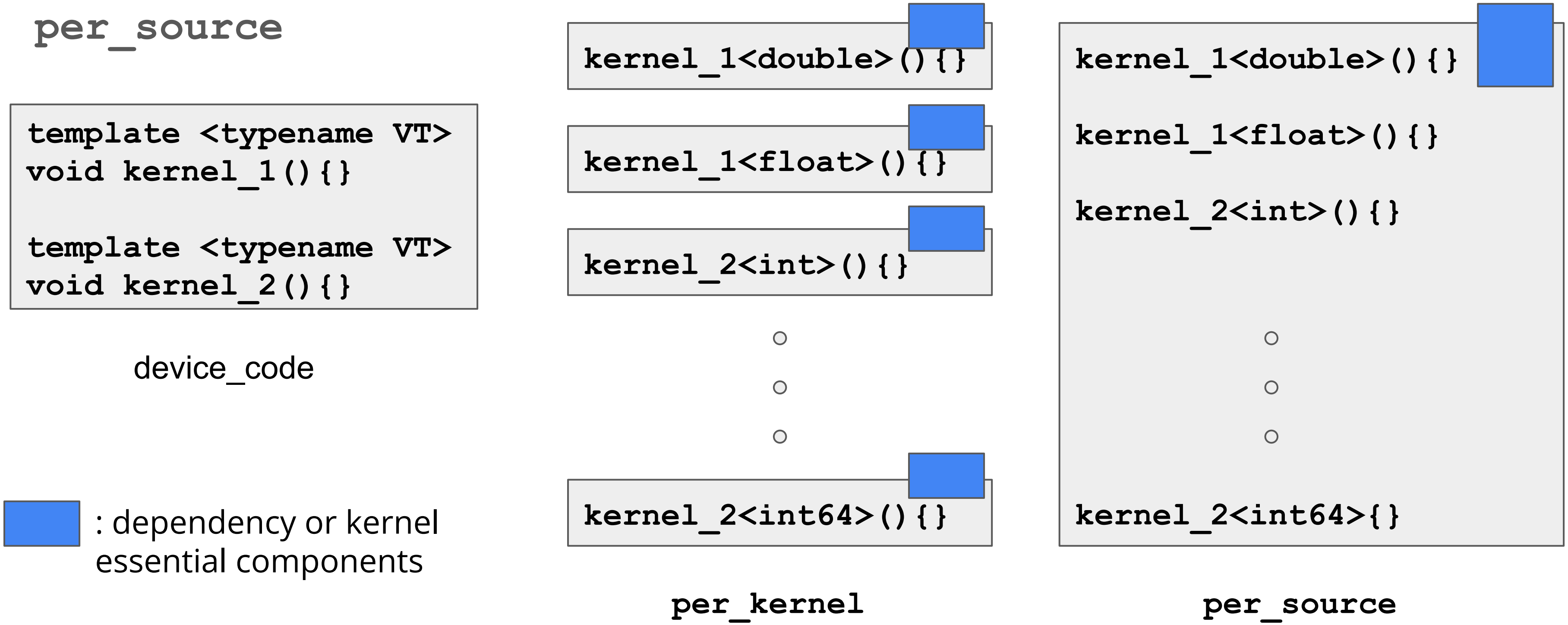
The corresponding error is

It will throw relocation truncated to fit: R_X86_64_GOTPCREL.... and PC-relative offset overflows in PLT entry …

The error makes sense because each instantiation is isolated and complete.

device_code_split

DPC++ provides different way to split the device code: **`per_kernel`** or **`per_source`**



per_source

Pro:
- Reduce the size of debug library.
- Compile faster than **`per_kernel`**.
- Reuse the kernel essential part or dependency

Con:
- All kernel instantiations need to be valid on the device.
- Takes more time on the first kernel of each device source file.

Issue: Too many kernels lead OOM issue on CPU. GPU does not face this issue. With the Intel team, we already submitted this issue and they are working on it

For example, we have a function which needs to select valuetype, workgroup size, subgroup size, (virtual) sub-subgroup size. It gives ~360 kernels in one function and leads this issue.

Devices with varying parameters

CPU can support 4, 8, 16, 32, 64 subgroup size and larger max workgroup size than 1024.
(32, 64 can be used after one of DPC++ 2021 release.)

GPU can support 8, 16, 32 subgroup size. However, different GPUs support different max workgroup size like 256, 512.

Gen9 Integrated GPU uses 256 as max workgroup size.
Gen12 Integrated GPU/Gen12LP Discrete GPU use 512 as max workgroup size.

Changes from per_kernel to per_source

Originally, we went for the **`per_kernel`** way which instantiated all possible kernels into Ginkgo library.

However, we faced the too large debug library issue and we need to support the AOT compilation in the future.

Thus, we need to make subgroup size and workgroup size adjustable such that we can use the valid configuration for using Ahead of Time(AOT) compilation or **`per_source`**.

SYCL does not support early exit

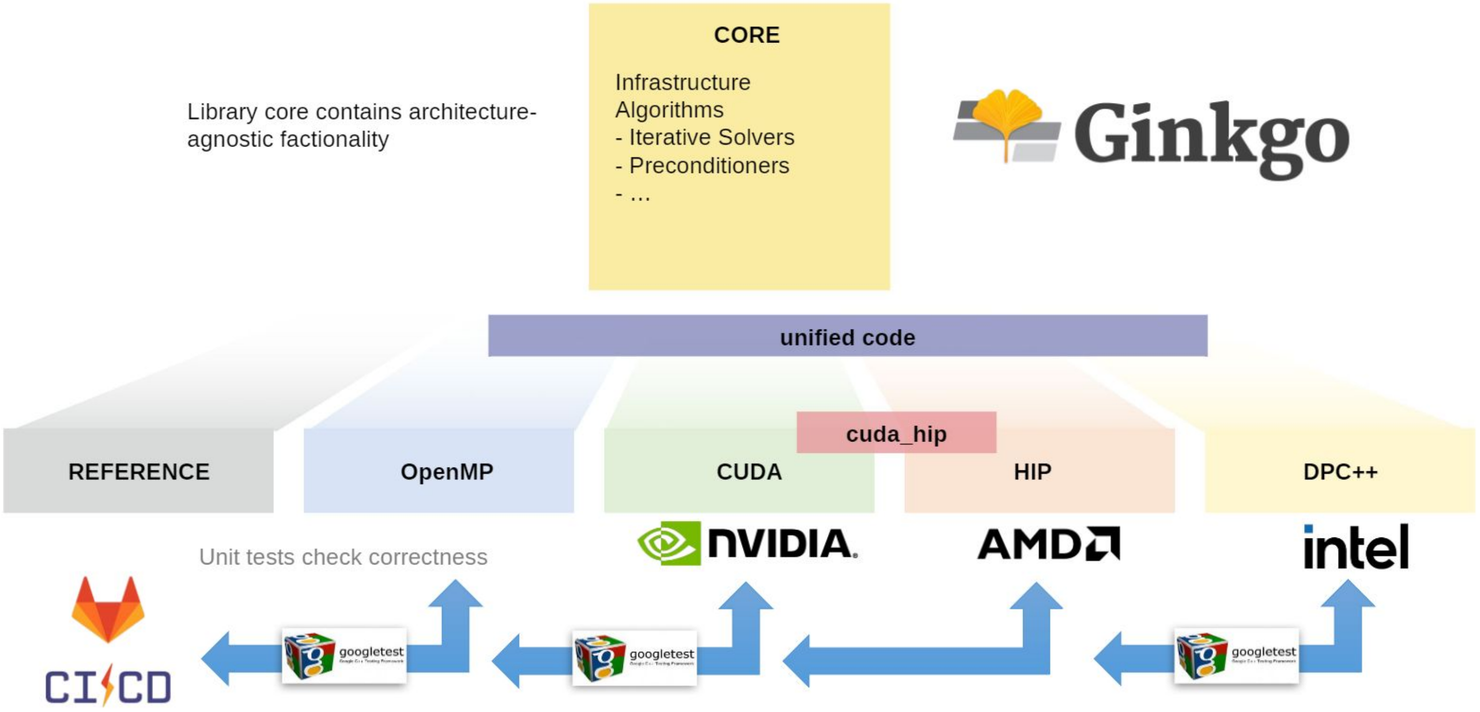
Any early exit thread in kernels requiring synchronization leads undefined behavior in SYCL.

```
{
if (condition) {
  return;
}

// process
__syncthreads();
// process
}
```

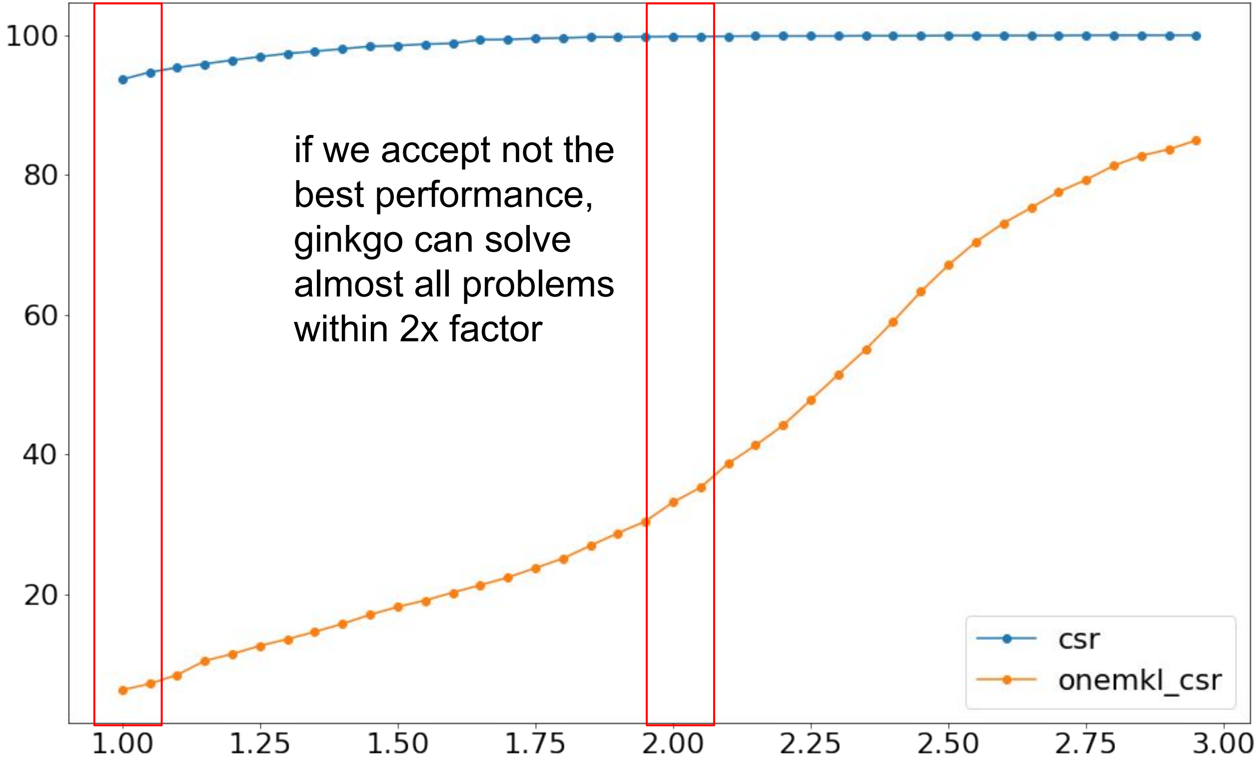
```
{
if (!condition) {
  // process
}
item_ct1.barrier()
if (!condition) {
  // process
}
}
```

Enhance maintaince



Performance

Ginkgo gives better performance among 90% of all suitesparse real matrices.



Conclusion

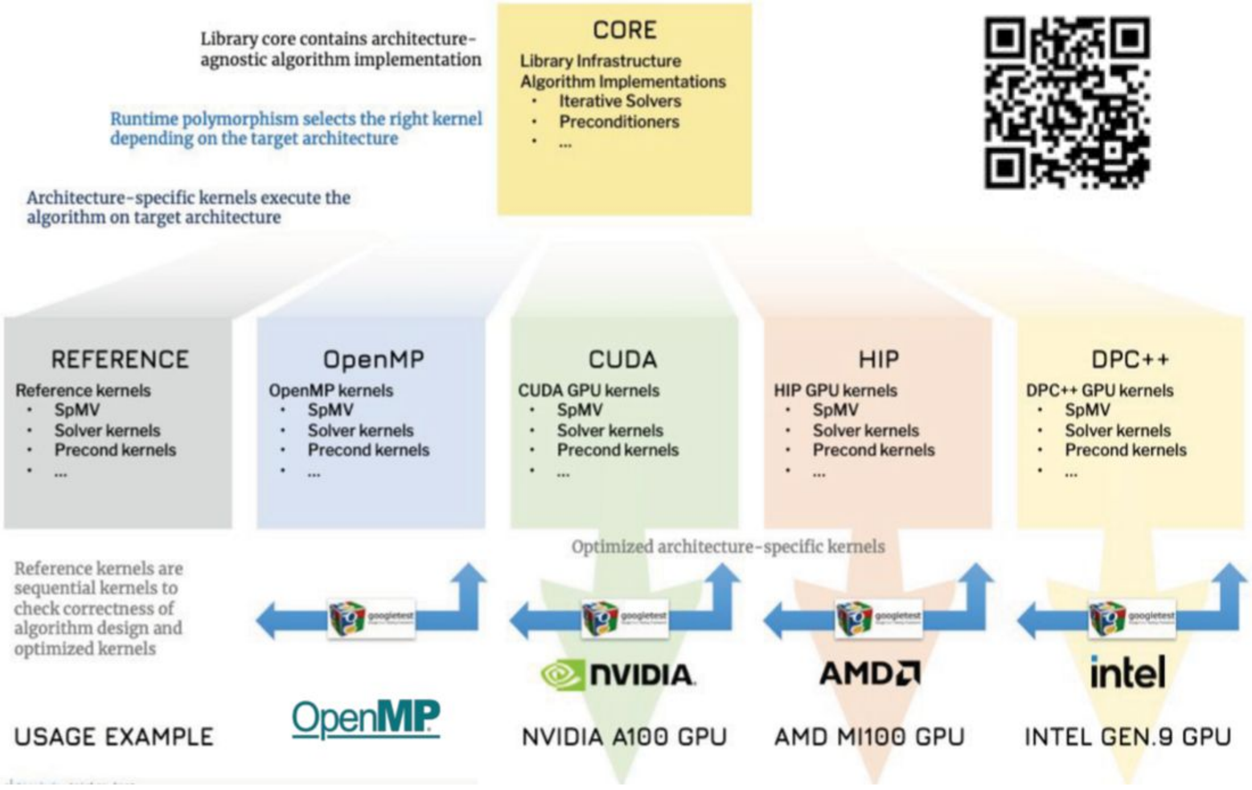
We use the oneAPI ecosystem to prepare Ginkgo for Intel GPUs

Ginkgo provides comprehensive sparse linear algebra support for devices supporting DPC++/SYCL including intel GPUs.

We use oneAPI to make Ginkgo be a SYCL-available library. We demonstrate our workaround for some issues on SYCL.

We are looking forward to the addition of the sub-subgroup feature, more oneDPL funtionality.

Thanks!



	Functionality	OMP	CUDA	HIP	DPC++
Basic	SpMV	✓	✓	✓	✓
	SpMM	✓	✓	✓	✓
	SpGeMM	✓	✓	✓	✓
Krylov solvers	BiCG	✓	✓	✓	✓
	BiCGSTAB	✓	✓	✓	✓
	CG	✓	✓	✓	✓
	CGS	✓	✓	✓	✓
	GMRES	✓	✓	✓	✓
	IDR	✓	✓	✓	✓
Preconditioners	(Block-)Jacobi	✓	✓	✓	✓
	ILU/IC		✓	✓	✓
	Parallel ILU/IC	✓	✓	✓	✓
	Parallel ILUT/ICT	✓	✓	✓	✓
	Sparse Approximate Inverse	✓	✓	✓	✓
Batched	Batched BiCGSTAB	✓	✓	✓	
	Batched CG	✓	✓	✓	
	Batched GMRES	✓	✓	✓	
	Batched ILU	✓	✓	✓	✓
	Batched ISAI	✓	✓	✓	✓
	Batched Jacobi	✓	✓	✓	✓
AMG	AMG preconditioner	✓	✓	✓	✓
	AMG solver	✓	✓	✓	✓
	Parallel Graph Match	✓	✓	✓	✓
Sparse direct	Symbolic Cholesky	✓	✓	✓	✓
	Numeric Cholesky	UNDER DEVELOPMENT			
	Symbolic LU	UNDER DEVELOPMENT			
	Numeric LU	✓	✓	✓	
	Sparse TRSV	✓	✓	✓	
Utilities	On-Device Matrix Assembly	✓	✓	✓	✓
	MC64/RCM reordering	✓			
	Wrapping user data	✓			
	Logging	✓			
	PAPI counters	✓			