Challenges Faced When Porting Ginkgo to the SYCL Ecosystem

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

[1]Karlsruhe Institute of Technology, [2]University of Tennessee

Overview

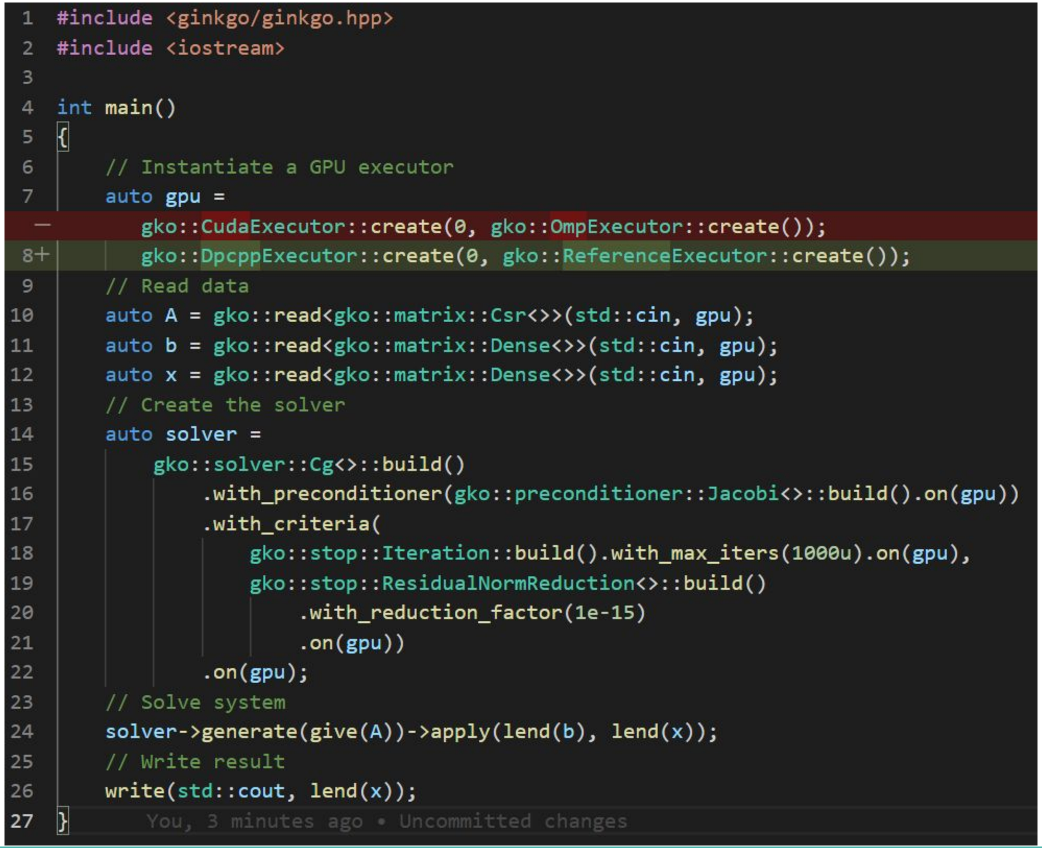
- Ginkgo
- Challenges and Workarounds
  - kernel attributes propagation rules changes
  - device_code_split - per_kernel and per_source
  - devices with varying parameters
- Ginkgo Performance on Intel GPUs

Cross-Platform Sparse Linear Algebra Library

Ginkgo

Ginkgo for cross-platform

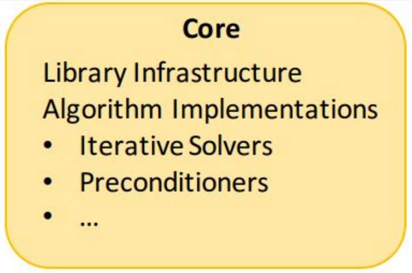
Ginkgo is an open source C++ math library with focus on sparse linear algebra functionality on GPUs. It provides the same interface across different devices to users.



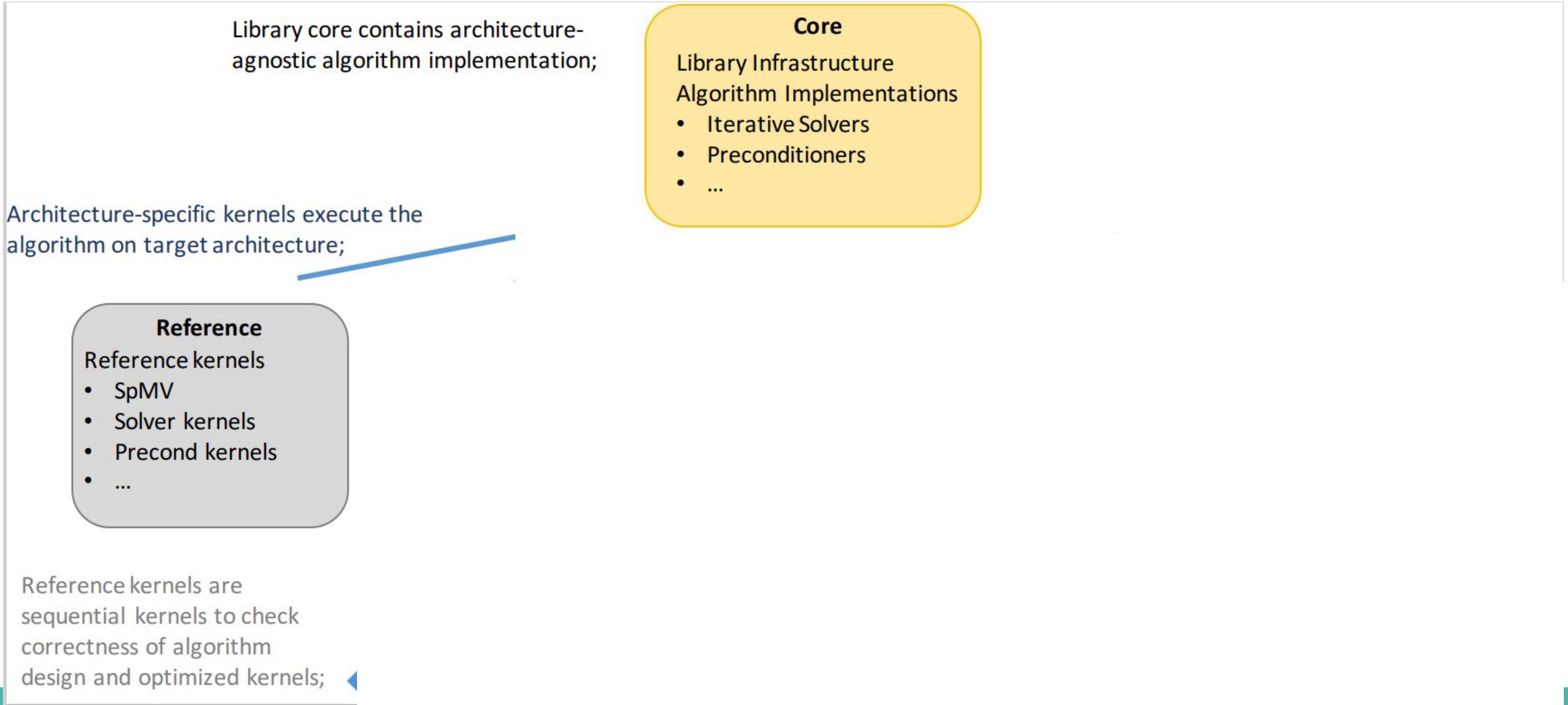
```
#include <ginkgo/ginkgo.hpp>
#include <iostream>

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;



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
- …

Architecture-specific kernels execute the algorithm on target architecture;

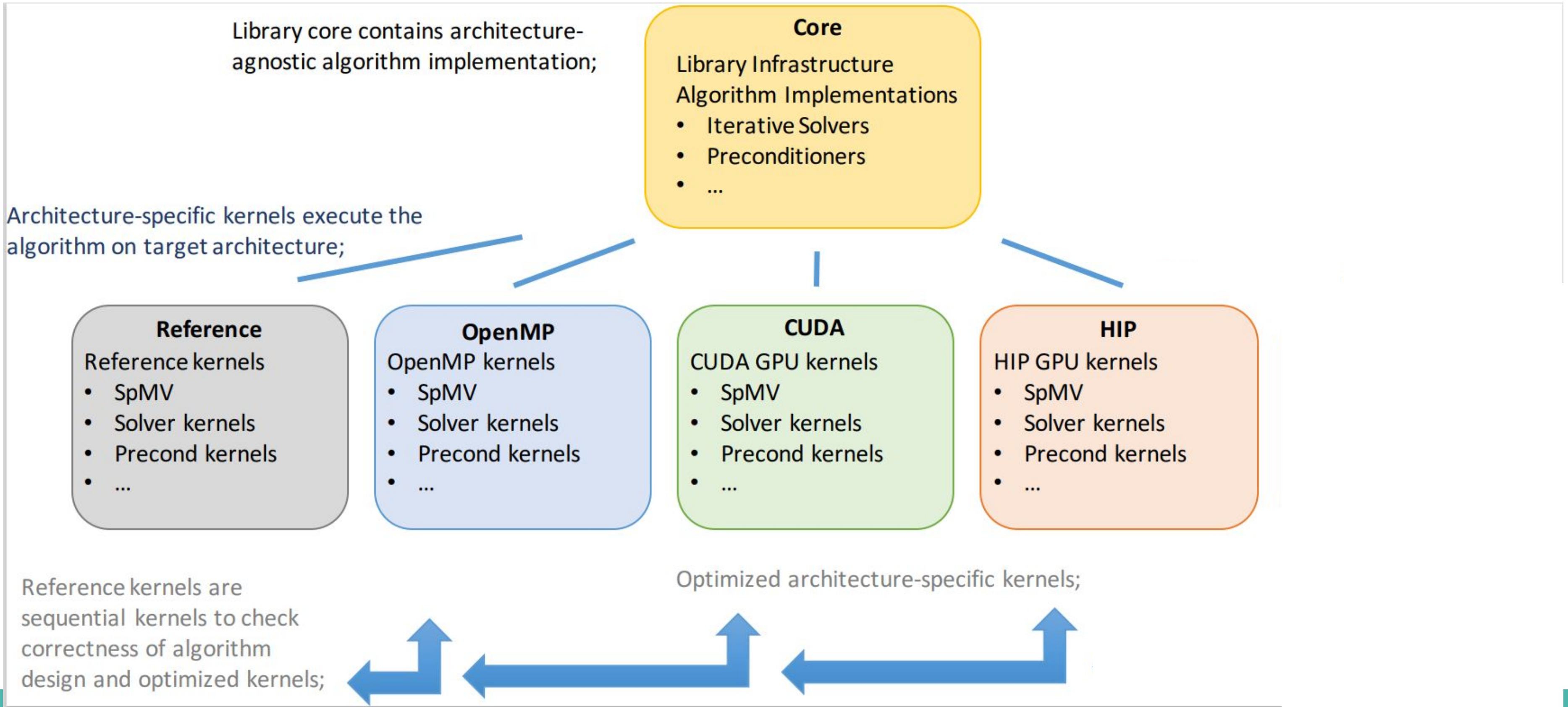
Reference

Reference kernels

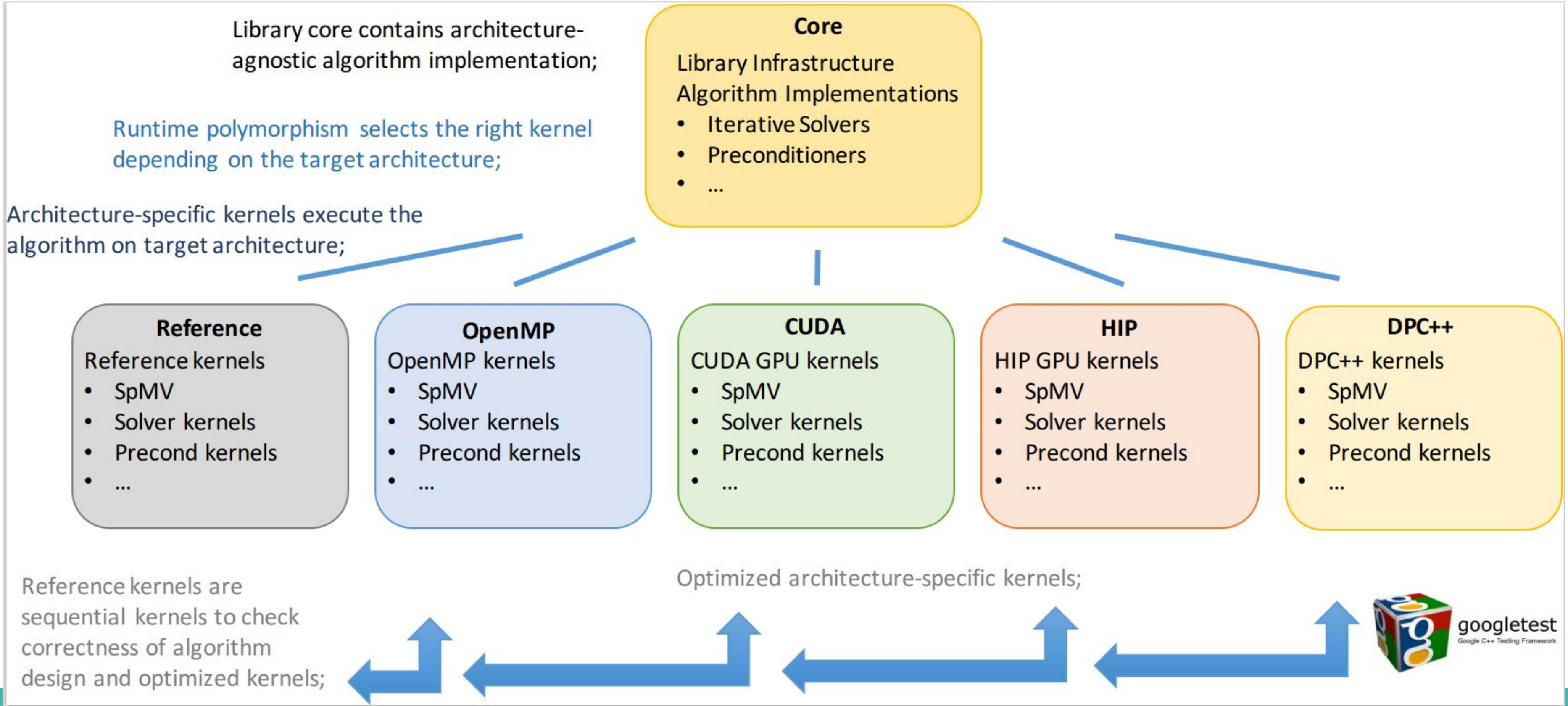
- SpMV
- Solver kernels
- Precond kernels
- …

Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels;

Ginkgo provides the same interface but with native device language



Ginkgo provides the same interface but with native device language


Library core contains architecture-agnostic algorithm implementation;
Runtime polymorphism selects the right kernel depending on the target architecture;
Architecture-specific kernels execute the algorithm on target architecture;
Core
Library Infrastructure
Algorithm Implementations
• Iterative Solvers
• Preconditioners
• …
Reference
Reference kernels
• SpMV
• Solver kernels
• Precond kernels
• …
OpenMP
OpenMP kernels
• SpMV
• Solver kernels
• Precond kernels
• …
CUDA
CUDA GPU kernels
• SpMV
• Solver kernels
• Precond kernels
• …
HIP
HIP GPU kernels
• SpMV
• Solver kernels
• Precond kernels
• …
DPC++
DPC++ kernels
• SpMV
• Solver kernels
• Precond kernels
• …
Reference kernels are sequential kernels to check correctness of algorithm design and optimized kernels;
Optimized architecture-specific kernels;
googletest
Google C++ Testing Framework

Ginkgo supports oneAPI

oneMKL has comprehensive support for dense linear algebra but features few basic routines for sparse linear algebra.

Ginkgo provides sparse linear algebra functionality and numerical algorithms beyond basic building blocks like SpMV.
With Ginkgo, complete numerical simulation workflows can be realized: iterative solvers, preconditioners, algebraic multigrid, etc.

oneAPI allows to execute on GPUs and CPUs from Intel and oneDPL gives similar functions like Thrust.

Ginkgo brings more functionality to oneAPI

Matrix: Dense, Csr, Coo, Ell, Sellp, Hybrid, fixed-blockCsr, SparseCsr

Solver: CG, BiCG, BiCGStab, CGS, FCG, (CB-)GMRES, IDR, Multigrid

Preconditioner: Jacobi, ISAI, (Par)ILU, (Par)IC

Mixed Precision support:

Mixed precision preconditioners
Mixed precision GMRES (CB-GMERS)
Mixed precision Multigrid

Challenges and Workarounds

Kernel attributes propagation rule changes

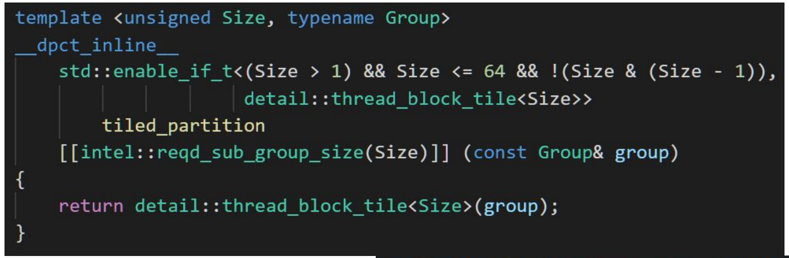
When SYCL 1.2.1 with Intel extension (before DPC++ 2022), the kernel attributes from inner components can be propagated to the top level.

Rely on it, Ginkgo introduces the cooperative group for DPC++.
It allows that we set the subgroup_size inside the kernel, and we can use it like CUDA cooperative group.

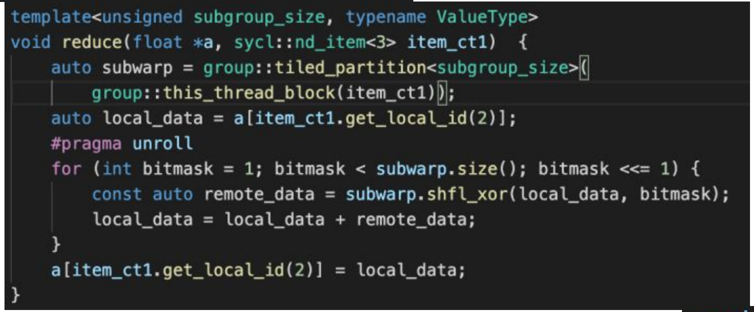
However, SYCL 2020 and DPC++ 2022 disabled this rule.
It can only set the kernel attributes from the top call.

We also show our high interest about the sub-subgroup size

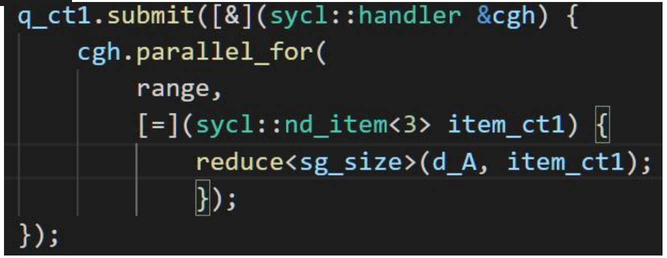
SYCL 1.2.1: use subgroup like cooperative groups



```
template <unsigned Size, typename Group>
__dpct_inline__
    std::enable_if_t<(Size > 1) && Size <= 64 && !(Size & (Size - 1)),
                     detail::thread_block_tile<Size>>
        tiled_partition
    [[intel::reqd_sub_group_size(Size)]] (const Group& group)
{
    return detail::thread_block_tile<Size>(group);
}
```



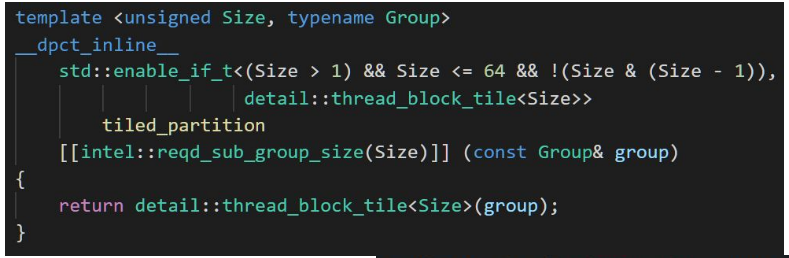
```
template<unsigned subgroup_size, typename ValueType>
void reduce(float *a, sycl::nd_item<3> item_ct1)  {
    auto subwarp = group::tiled_partition<subgroup_size>(
        group::this_thread_block(item_ct1));
    auto local_data = a[item_ct1.get_local_id(2)];
    #pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[item_ct1.get_local_id(2)] = local_data;
}
```



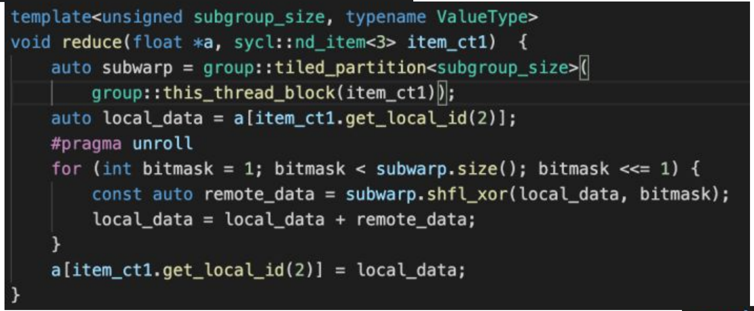
```
q_ct1.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(
        range,
        [=](sycl::nd_item<3> item_ct1) {
            reduce<sg_size>(d_A, item_ct1);
        });
});
```

create the subgroup with template
the kernel attributes are propagated to outside

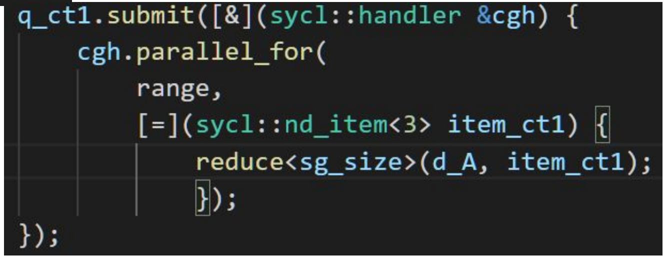
SYCL 1.2.1: use subgroup like cooperative groups



```
template <unsigned Size, typename Group>
__dpct_inline__
    std::enable_if_t<(Size > 1) && Size <= 64 && !(Size & (Size - 1)),
                     detail::thread_block_tile<Size>>
        tiled_partition
    [[intel::reqd_sub_group_size(Size)]] (const Group& group)
{
    return detail::thread_block_tile<Size>(group);
}
```



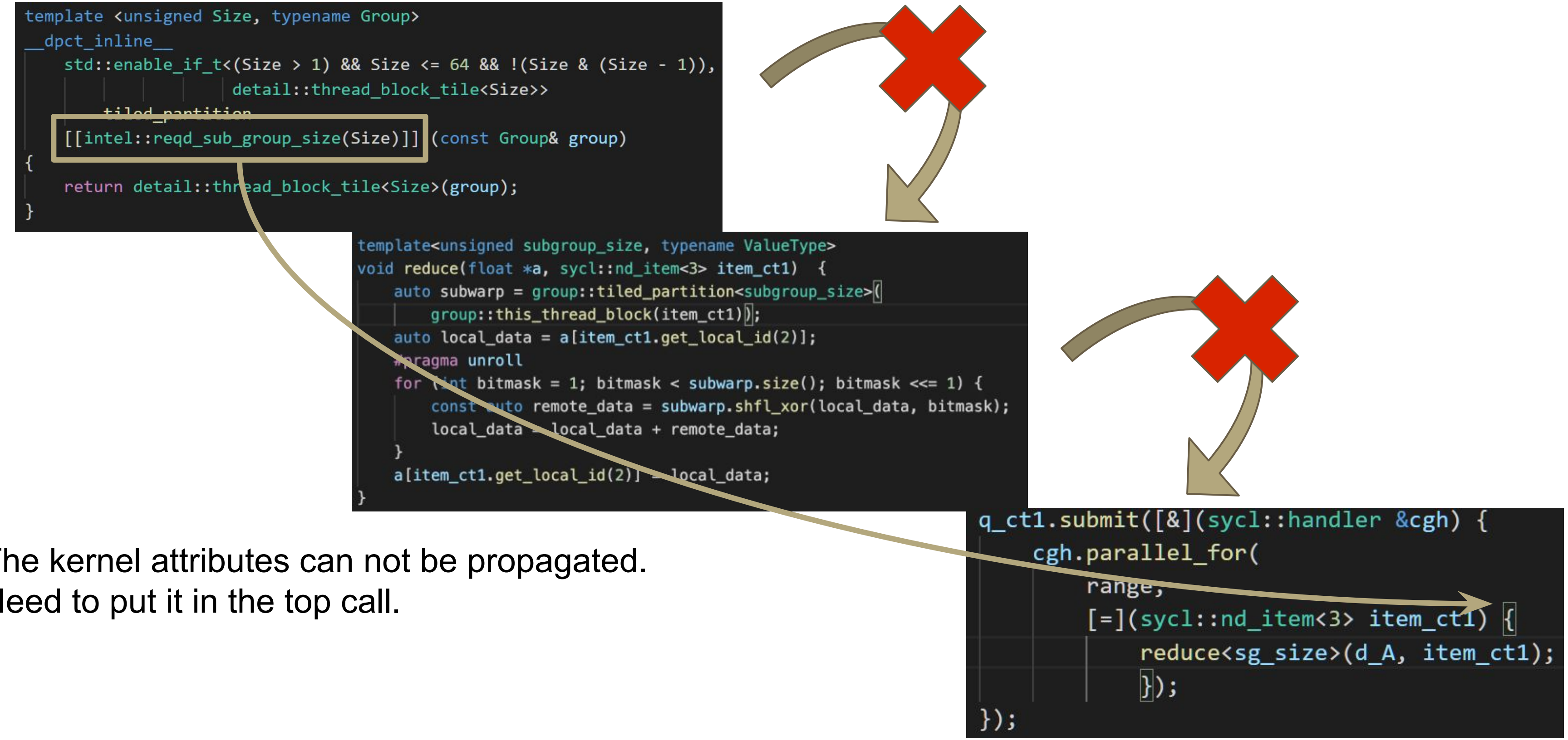
```
template<unsigned subgroup_size, typename ValueType>
void reduce(float *a, sycl::nd_item<3> item_ct1)  {
    auto subwarp = group::tiled_partition<subgroup_size>(
        group::this_thread_block(item_ct1));
    auto local_data = a[item_ct1.get_local_id(2)];
    #pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[item_ct1.get_local_id(2)] = local_data;
}
```



```
q_ct1.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(
        range,
        [=](sycl::nd_item<3> item_ct1) {
            reduce<sg_size>(d_A, item_ct1);
        });
});
```

create the subgroup with template
the kernel attributes are propagated to outside

SYCL 2020: decide the subgroup size before using it



```
template <unsigned Size, typename Group>
__dpct_inline__
    std::enable_if_t<(Size > 1) && Size <= 64 && !(Size & (Size - 1)),
                     detail::thread_block_tile<Size>>
        tiled_partition
    [[intel::reqd_sub_group_size(Size)]] (const Group& group)
{
    return detail::thread_block_tile<Size>(group);
}
```

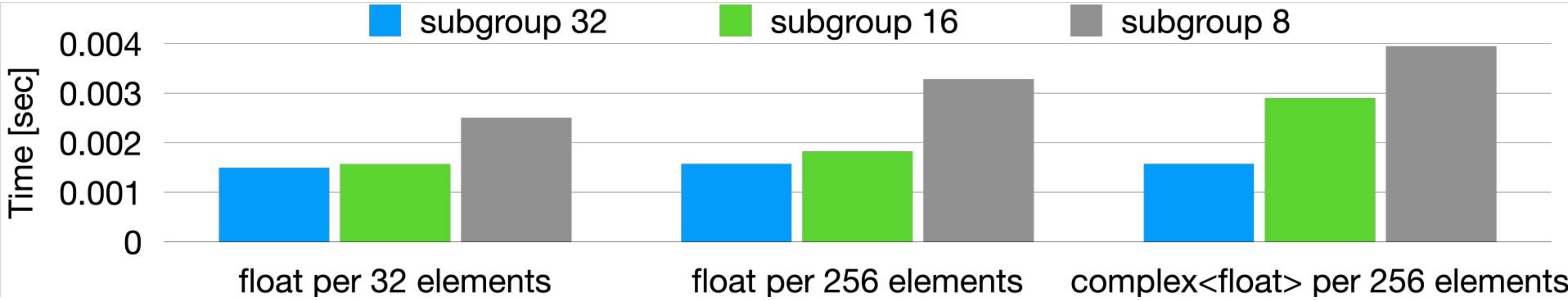
```
template<unsigned subgroup_size, typename ValueType>
void reduce(float *a, sycl::nd_item<3> item_ct1)  {
    auto subwarp = group::tiled_partition<subgroup_size>(
        group::this_thread_block(item_ct1));
    auto local_data = a[item_ct1.get_local_id(2)];
    #pragma unroll
    for (int bitmask = 1; bitmask < subwarp.size(); bitmask <<= 1) {
        const auto remote_data = subwarp.shfl_xor(local_data, bitmask);
        local_data = local_data + remote_data;
    }
    a[item_ct1.get_local_id(2)] = local_data;
}
```

```
q_ct1.submit([&](sycl::handler &cgh) {
    cgh.parallel_for(
        range,
        [=](sycl::nd_item<3> item_ct1) {
            reduce<sg_size>(d_A, item_ct1);
        });
});
```

The kernel attributes can not be propagated.
Need to put it in the top call.

Why we need to adjust the subgroup size

The performance of subset reduction on 10,000,000 elements on Gen12 GPU (Intel(R) Iris(R) Xe MAX Graphics - Gen12LP)
"float per 32 elements" means using float as the element type and performing reduction 32 by 32 elements



Auto generate selection

We need to generate the kernels for different situation in compile time and then use the desired one to fit the current situation in runtime.

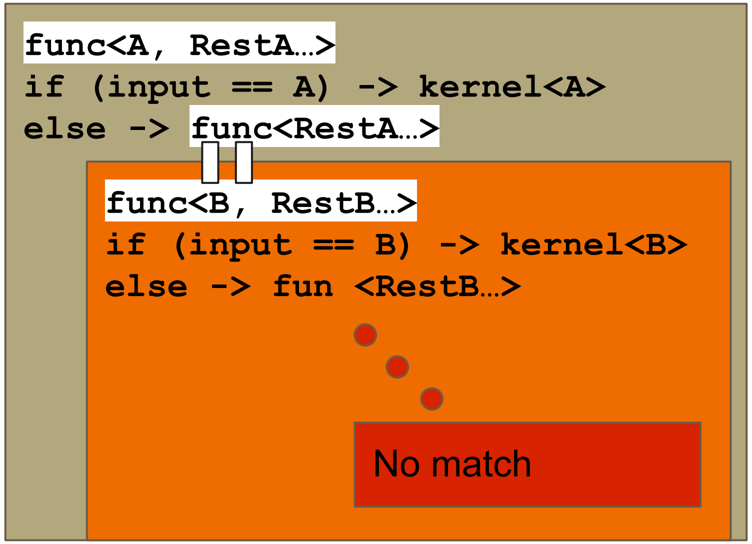
possible configs list = <A, B, C, ...>

Auto generate selection

We need to generate the kernels for different situation in compile time and then use the desired one to fit the current situation in runtime.

possible configs list = <A, B, C, ...>

Compile time:
template expanding is in compile time, so the kernel<A>, kernel <B> are already compiled

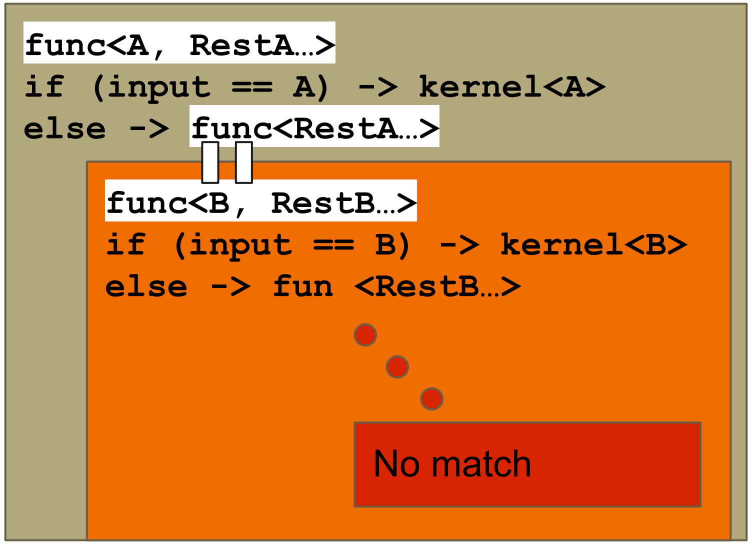

```
func<A, RestA…>
if (input == A) -> kernel<A>
else -> func<RestA…>
    func<B, RestB…>
    if (input == B) -> kernel<B>
    else -> fun <RestB…>
```

Auto generate selection

We need to generate the kernels for different situation in compile time and then use the desired one to fit the current situation in runtime.

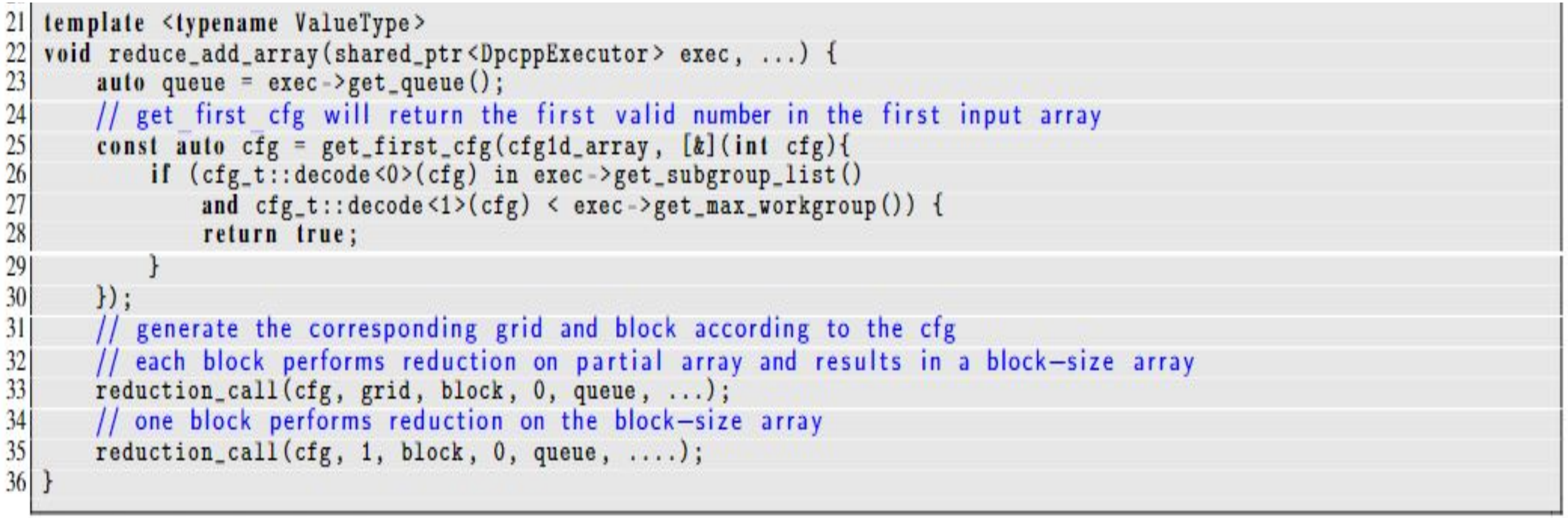
possible configs list = <A, B, C, ...>

Compile time:
template expanding is in compile time, so the kernel<A>, kernel <B> are already compiled



Run time:
we generate the nested if-else function. We pass the desired config into the function in runtine, and the nested conditions run the matched kernel.

Selection example - usage



```
template <typename ValueType>
void reduce_add_array(shared_ptr<DpcppExecutor> exec, ...) {
    auto queue = exec->get_queue();
    // get_first_cfg will return the first valid number in the first input array
    const auto cfg = get_first_cfg(cfg1d_array, [&](int cfg){
        if (cfg_t::decode<0>(cfg) in exec->get_subgroup_list()
            and cfg_t::decode<1>(cfg) < exec->get_max_workgroup()) {
            return true;
        }
    });
    // generate the corresponding grid and block according to the cfg
    // each block performs reduction on partial array and results in a block-size array
    reduction_call(cfg, grid, block, 0, queue, ...);
    // one block performs reduction on the block-size array
    reduction_call(cfg, 1, block, 0, queue, ....);
}
```

Choose the valid configiration and pass it to function.

device_code_split

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

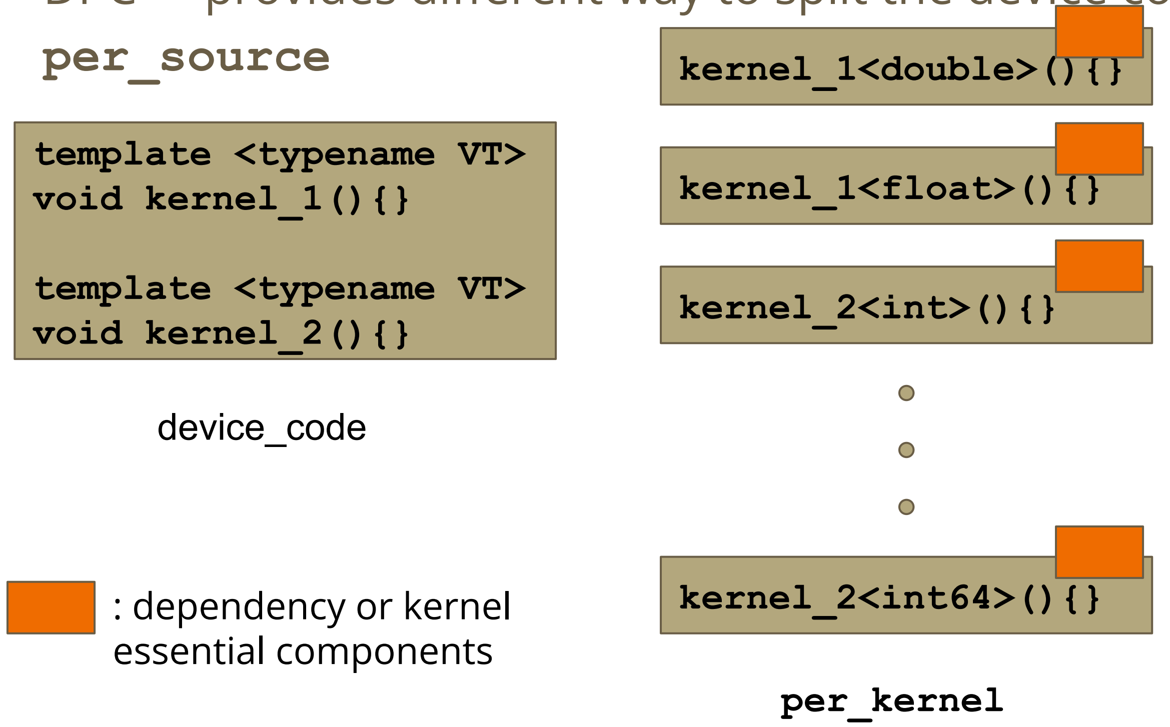
```
template <typename VT>
void kernel_1(){}

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.

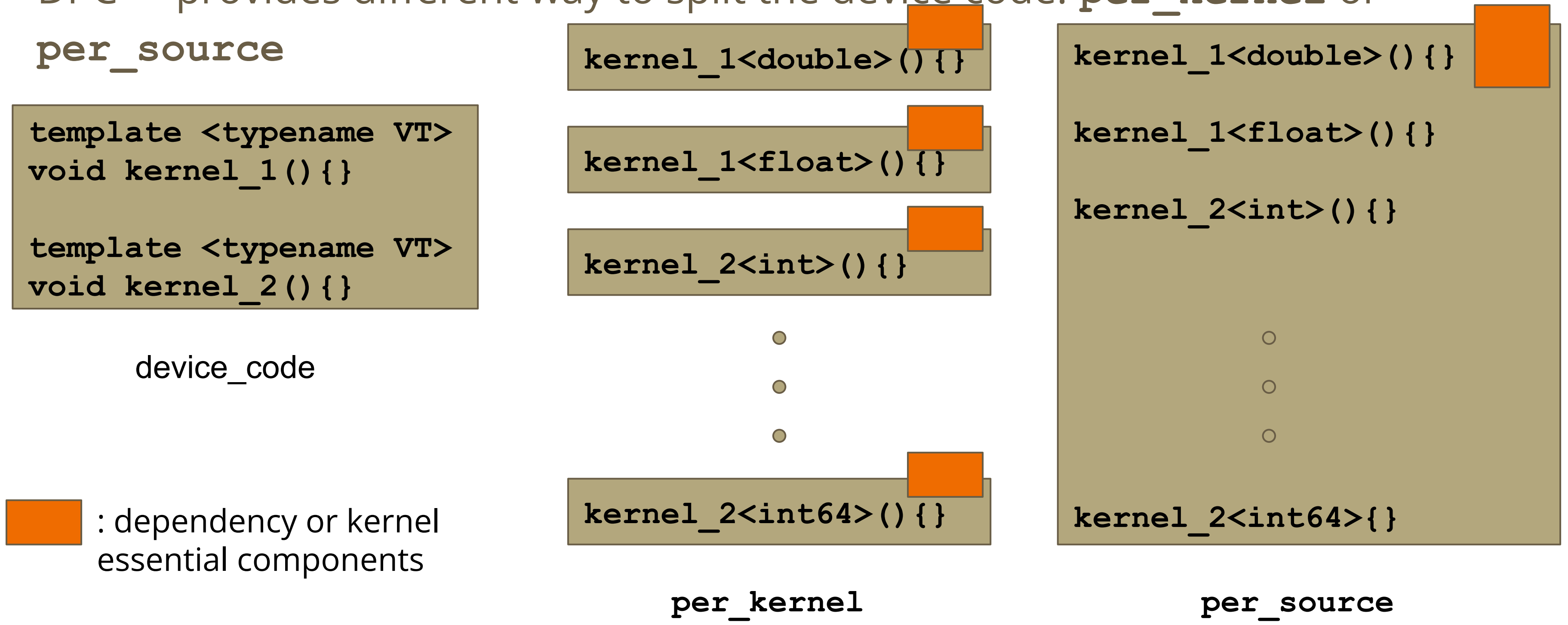
The corresponding error is

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`**



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

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

device_code

🟧 : dependency or kernel essential components

```
kernel_1<double>(){}
kernel_1<float>(){}
kernel_2<int>(){}
kernel_2<int64>(){}
```

`per_kernel`

```
kernel_1<double>(){}
kernel_1<float>(){}
kernel_2<int>(){}
kernel_2<int64>{}
```

`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`**.

CMake adjusts the subgroup/workgroup size list

Ginkgo introduces the cmake options for subgroup/workgroup size.

```
-DGINKGO_DPCPP_WORKGROUPS=256,512
```

```
-DGINKGO_DPCPP_SUBGROUPS=8,16,32
```

Ginkgo will produce the pair from these two lists.

<8, 256>, <16, 256>, <32, 256>, <8, 512>, <16, 512>, <32, 512>

Two level selection - split device and kernel selection

We use two levels selection to generate the corresponding kernels.

The first level only chooses which device attributes to use.

Then, the second level chooses which kernel arguments to run.

Note. The kernel is invalid only when the kernel is compiled with invalid subgroup or workgroup size. The local memory larger than the allowed size of device is not considered invalid in JIT.

Readability - type for device attributes

Originally, we use ConfigSet to embed the device attributes and the kernel arguments together. However, it leads some challenges when reading the codes.

```
wg_size = Config::decode<0>(cfg);
sg_size = Config::decode<1>(cfg);
```

Because we use two level selection, those attributes are splitted now.
We can also transfer the device attributes to type information.

```
Cfg::workgroup_size;
Cfg::subgroup_size;
```

We support both in the selection phase, but we will transfer it to the corresponding type in the device kernels.

Ginkgo Performance on Intel GPUs

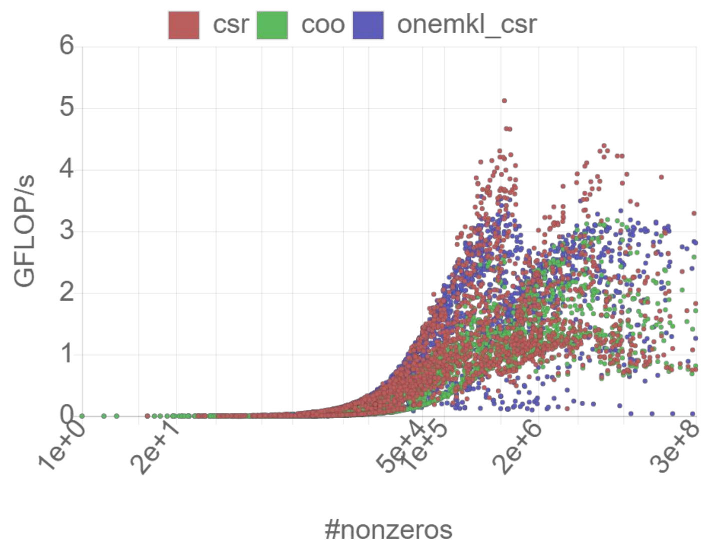
Benchmark detail

We use two Intel GPUs from devcloud in the following benchmark.

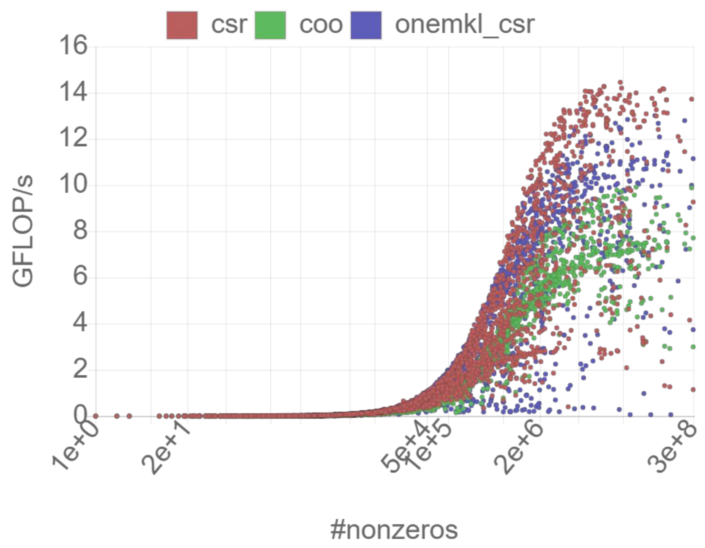
- Intel Integrated GPU: Intel(R) UHD Graphics P630, which is a integrated gpu with Intel(R) Xeon(R) E-2146G CPU (Gen9)
  It supports float and double precision.
- Intel Discrete GPU: Intel(R) Iris(R) Xe MAX Graphics, which is a Gen12 discrete GPU.
  It only supports float precision. (Gen12LP)

We write the GPU kernels. We do not use OpenMP offloading for GPU.

Ginkgo SpMV performance on Intel Gen9/Gen12 GPU



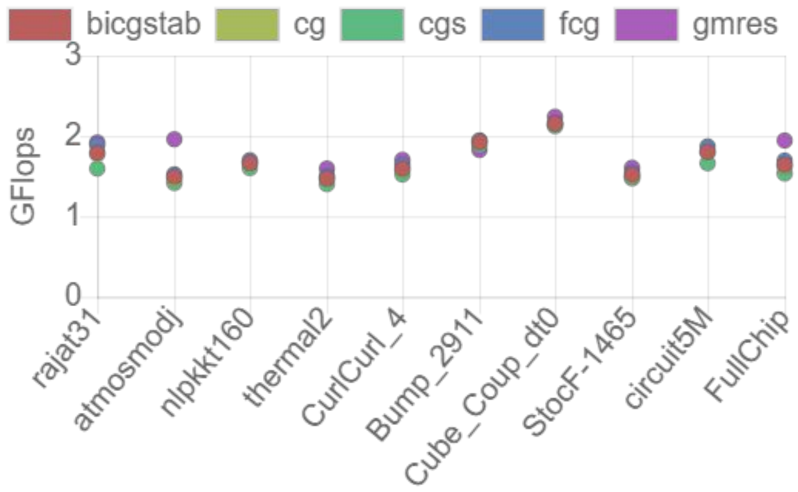
Intel Gen9 GPU
[double precision results]



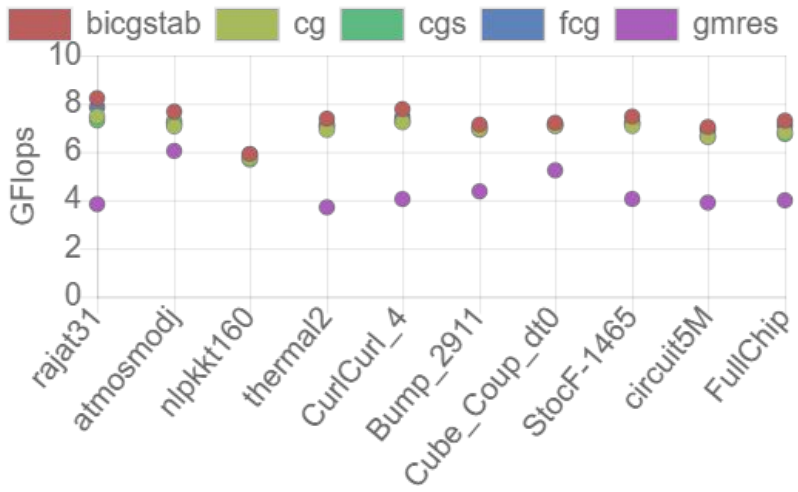
Intel Gen12LP GPU
[single precision results]

Sparse Matrix vector Multiplication (SpMV) kernel, performance comparison of Ginkgo functionality and oneMKL kernels.

Ginkgo solvers performance on Intel Gen9/Gen12 GPU

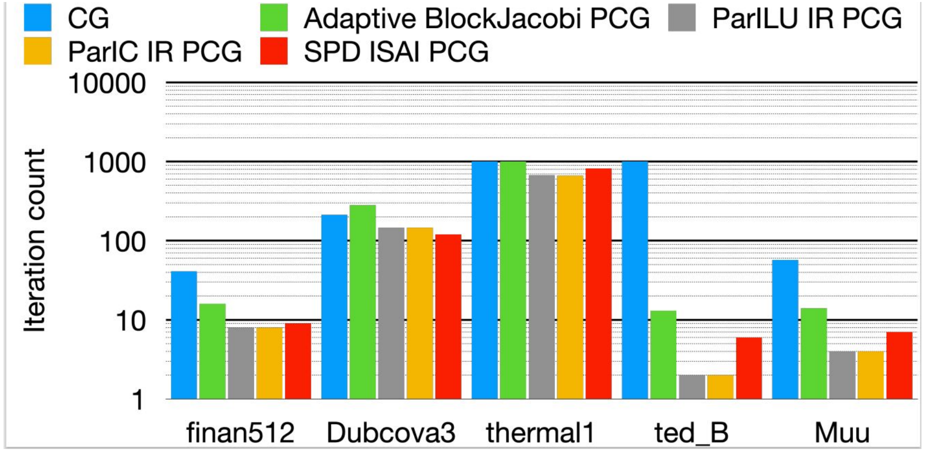


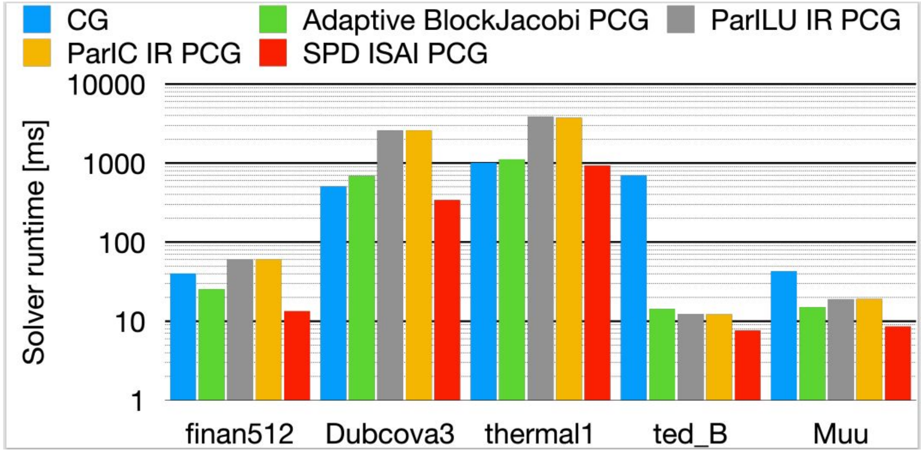
Intel Gen9 GPU
[double precision results]



Intel Gen12LP GPU
[single precision results]

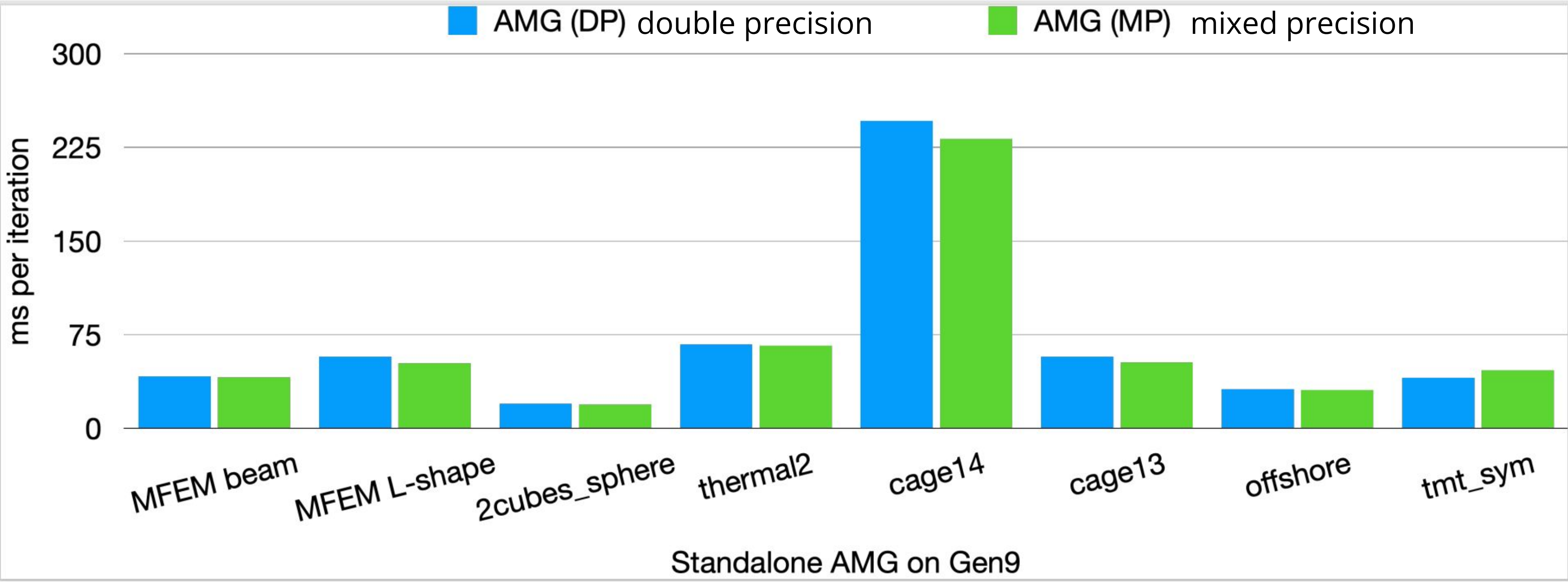
Ginkgo preconditioner effectiveness on Gen12LP GPU





Effectiveness of Ginkgo's preconditioners on Intel GPUs in terms of convergence improvement [iteration count] and time-to-solution acceleration. [solver runtime]

Ginkgo Algebraic Multigrid (AMG)



Ginkgo's Algebraic Multigrid (AMG) method in double precision and mixed precision mode running on Intel GPUs.

Conclusion

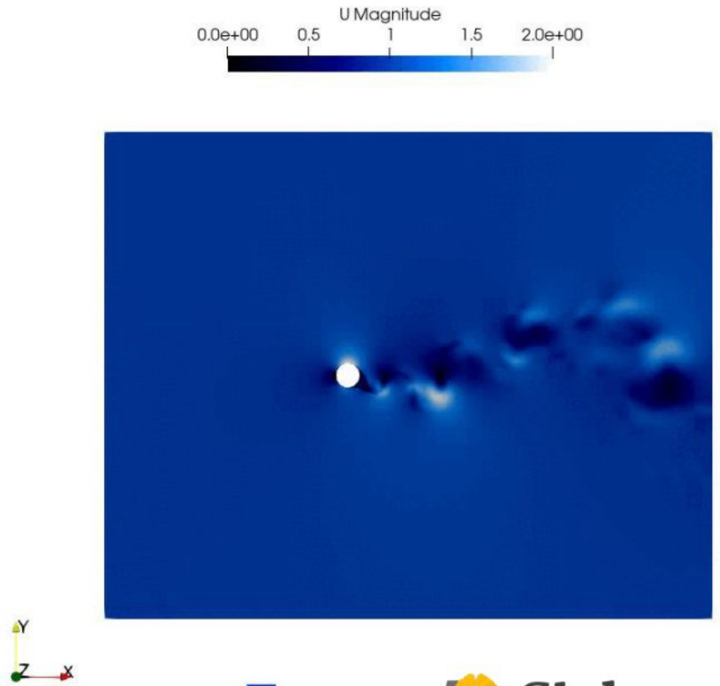
We use the oneAPI ecosystem to prepare Ginkgo for Intel GPUs

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, and adoption of oneDPL by other architectures.

Thanks!



OpenFOAM® Ginkgo

	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	✓	✓	✓	✓
	Algebraic Multigrid	✓	✓	✓	✓
Batched	Batched BiCGSTAB	✓	✓	✓	
	Batched CG	✓	✓	✓	
	Batched GMRES		✓	✓	