



## Ginkgo – An Open Source Math Library in the oneAPI Ecosystem

Hartwig Anzt, Terry Cojean, Fritz Goebel, Thomas Gruetzmacher, Aditya Kashi, Pratik Nayak, Tobias Ribizel, Yuhsiang M. Tsai  
Steinbuch Centre for Computing (SCC)



Hartwig Anzt



Terry Cojean



Fritz Göbel



Thomas  
Grützmacher



Aditya Kashi



Pratik Nayak



Tobias Ribizel



Mike Tsai

*This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration and the Helmholtz Impuls und Vernetzungsfond VH-NG-1241.*

# A Sustainable Open Source Math Software



- **High performance sparse linear algebra**
  - Linear solvers, eigenvalue solvers;
  - Advanced preconditioning techniques
  - Decoupling of arithmetic precision (hardware-supported) and memory precision;
  - Linear algebra building blocks;
  - Extensible, sustainable, production-ready;

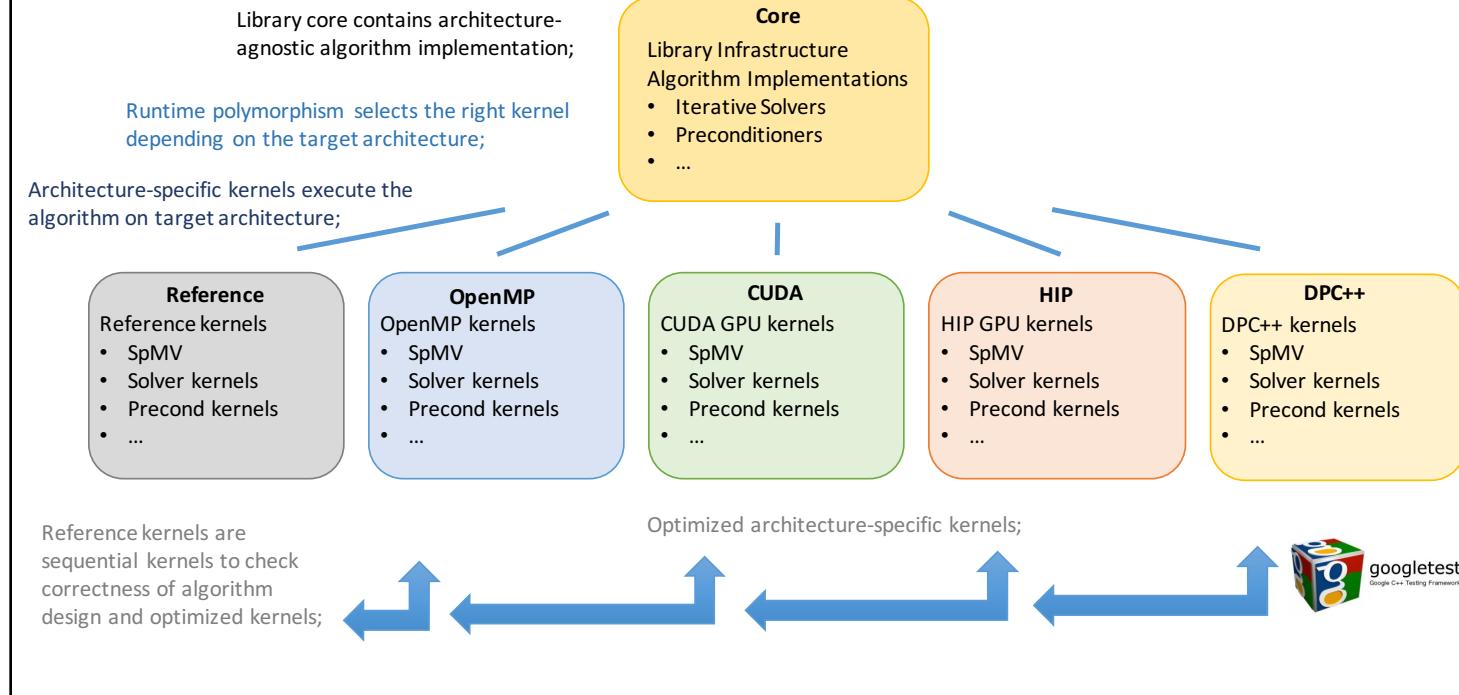
- **Open source, community-driven**
  - Freely available (BSD License), GitHub, and Spack.
  - Part of the **E4S** and **xSDK** software stack.
  - Collaborative Effort:



- Can be used from **MFEM**, and **deal.II**.



**GPU-centric high performance sparse linear algebra ecosystem.** Sustainable and extensible ecosystem with support for AMD GPUs, NVIDIA GPUs, and Intel GPUs.



<https://ginkgo-project.github.io/>

# A Sustainable Open Source Math Software

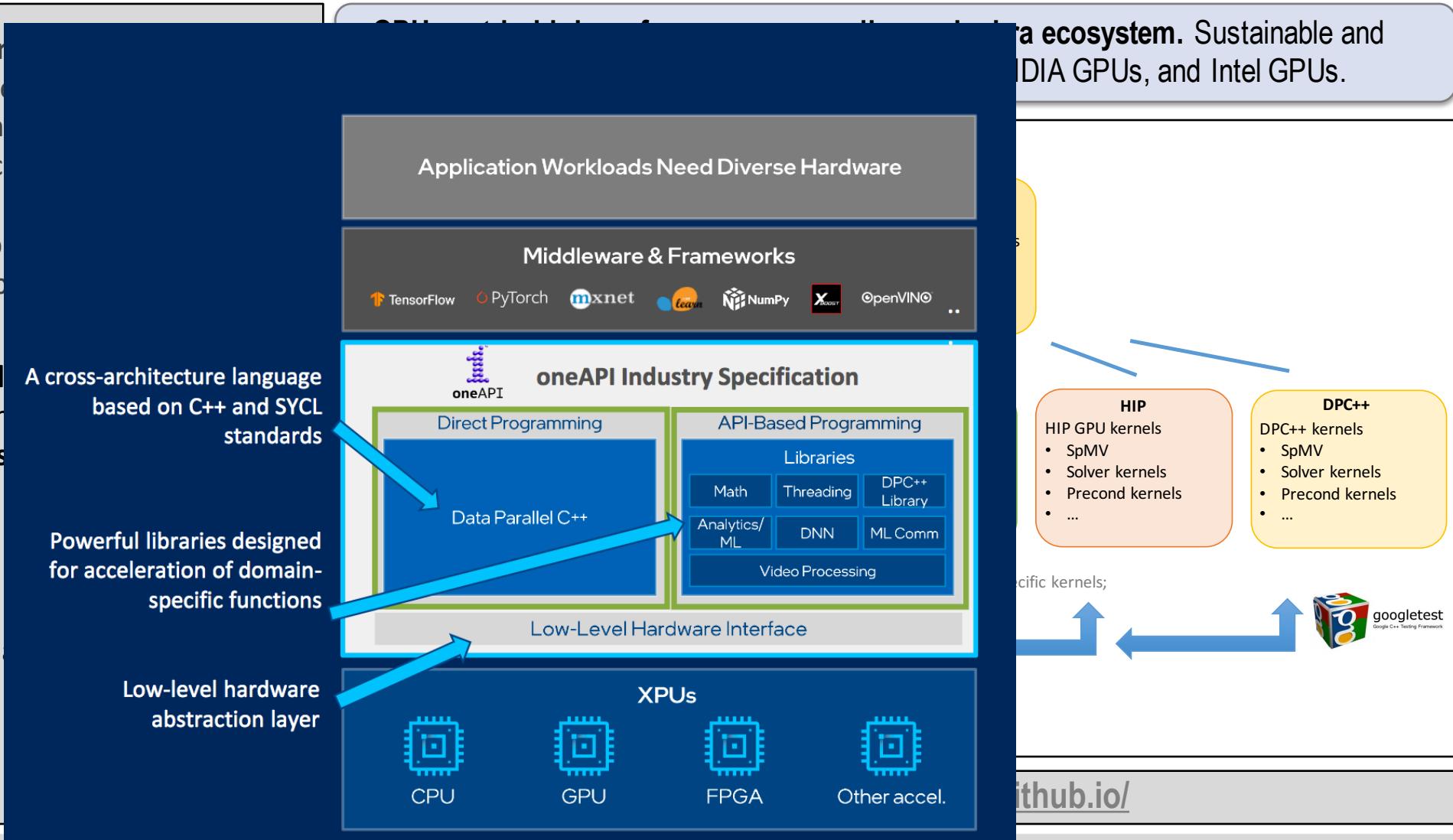


- **High performance sparse linear solvers**
    - Linear solvers, eigenvalue problems
    - Advanced preconditioning techniques
    - Decoupling of arithmetic (hardware supported) and memory access patterns
    - Linear algebra building blocks
    - Extensible, sustainable, parallel

- **Open source, community-driven**
    - Freely available (BSD License)
    - Part of the **E4S and xSDKs**
    - Collaborative Effort:



- Can be used from **MFEM**,



# A Sustainable Open Source Math Software



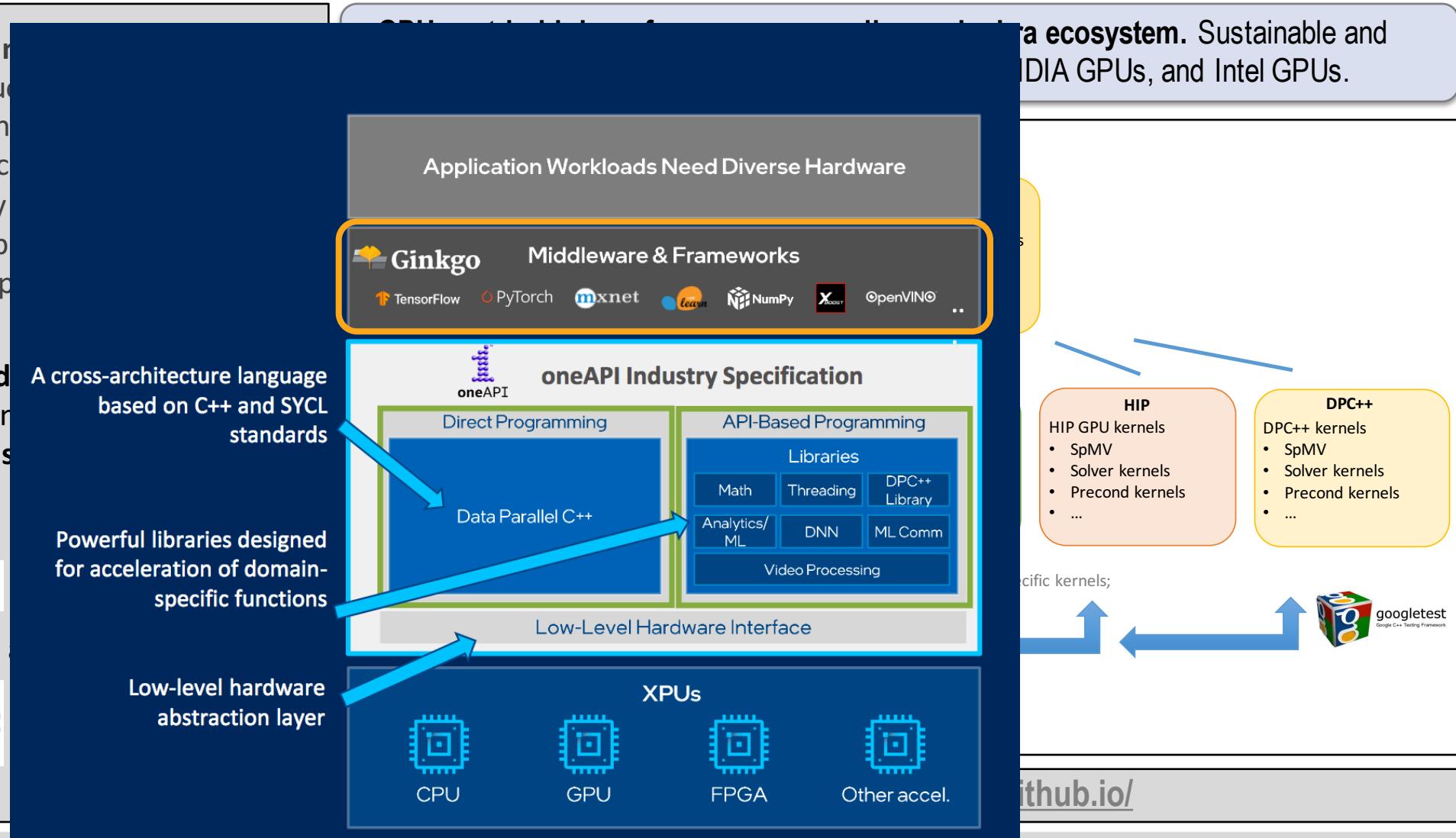
- **High performance sparse linear solvers**
    - Linear solvers, eigenvalue problems
    - Advanced preconditioning techniques
    - Decoupling of arithmetic (hardware supported) and memory access patterns
    - Linear algebra building blocks
    - Extensible, sustainable, parallel

- **Open source, community-driven**
    - Freely available (BSD License)
    - Part of the E4S and xSDK stacks

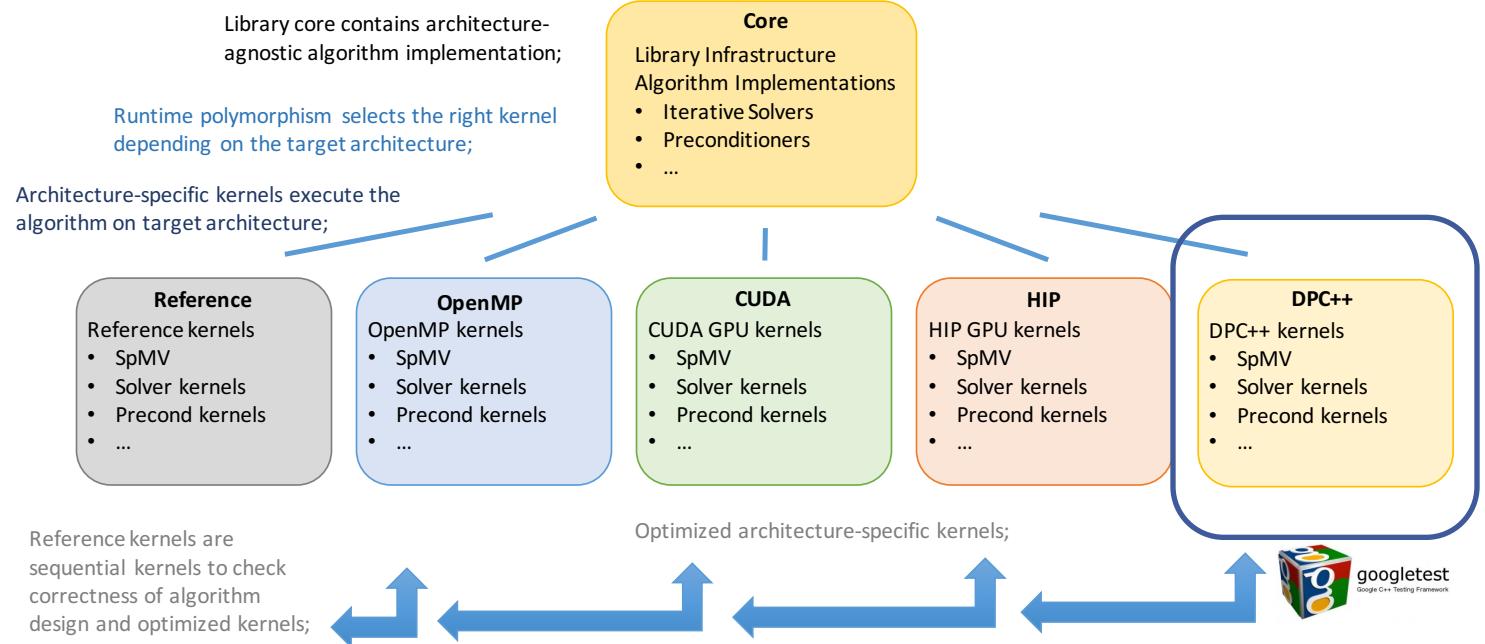
# A cross-architecture language based on C++ and SYCL standards

Powerful libraries designed  
for acceleration of domain-  
specific functions

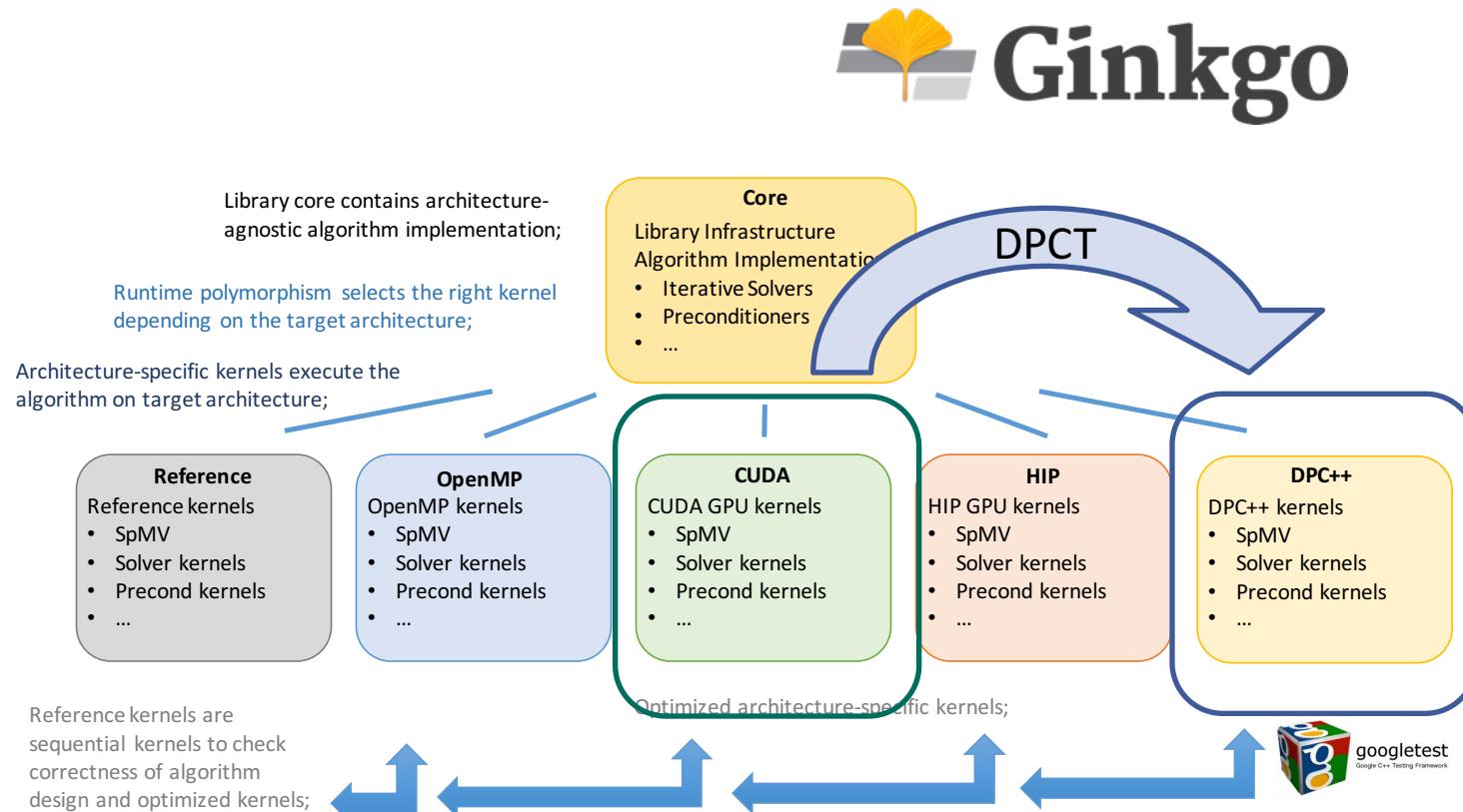
Low-level hardware abstraction layer



# Porting Ginkgo's CUDA Code to the oneAPI Ecosystem

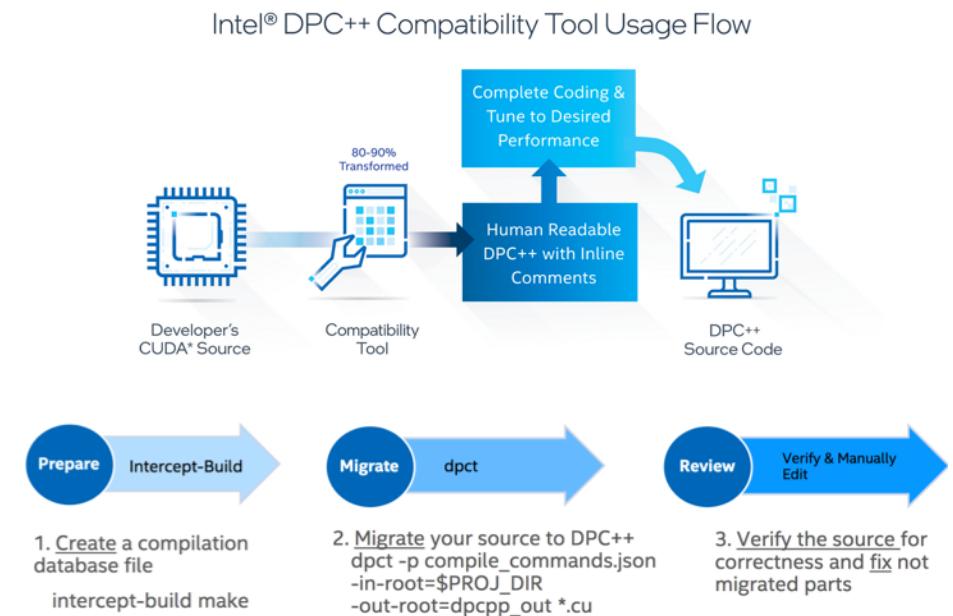


# Porting Ginkgo's CUDA Code to the oneAPI Ecosystem



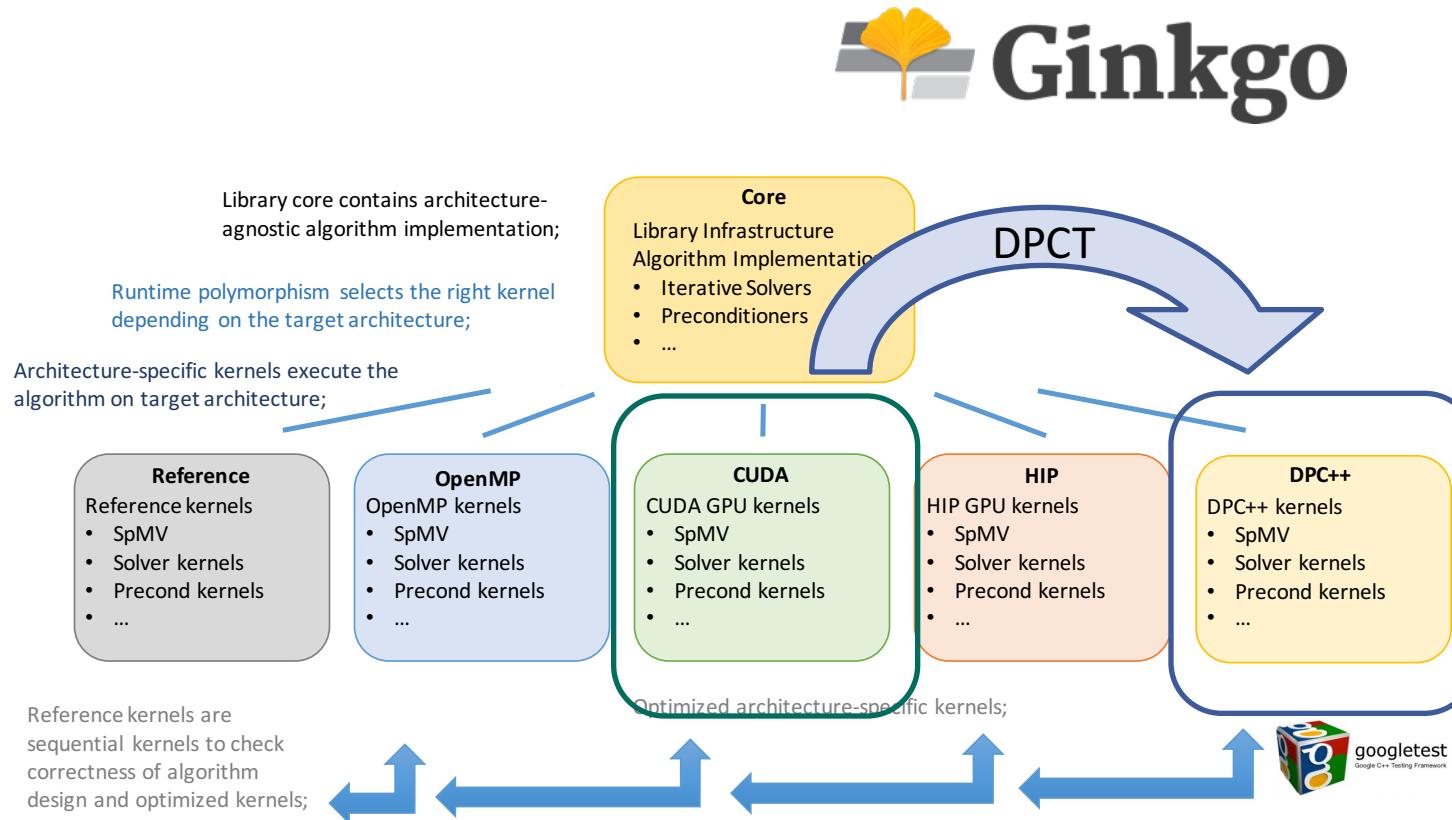
We generate the Ginkgo DPC++ backend from the CUDA backend via DPCT porting tool.

## Cross-Architecture Programming for Accelerated Compute, Freedom of Choice for Hardware Intel® DPC++ Compatibility Tool



<https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-intel-dpcpp-compatibility-tool/top.html>

# Porting Ginkgo's CUDA Code to the oneAPI Ecosystem



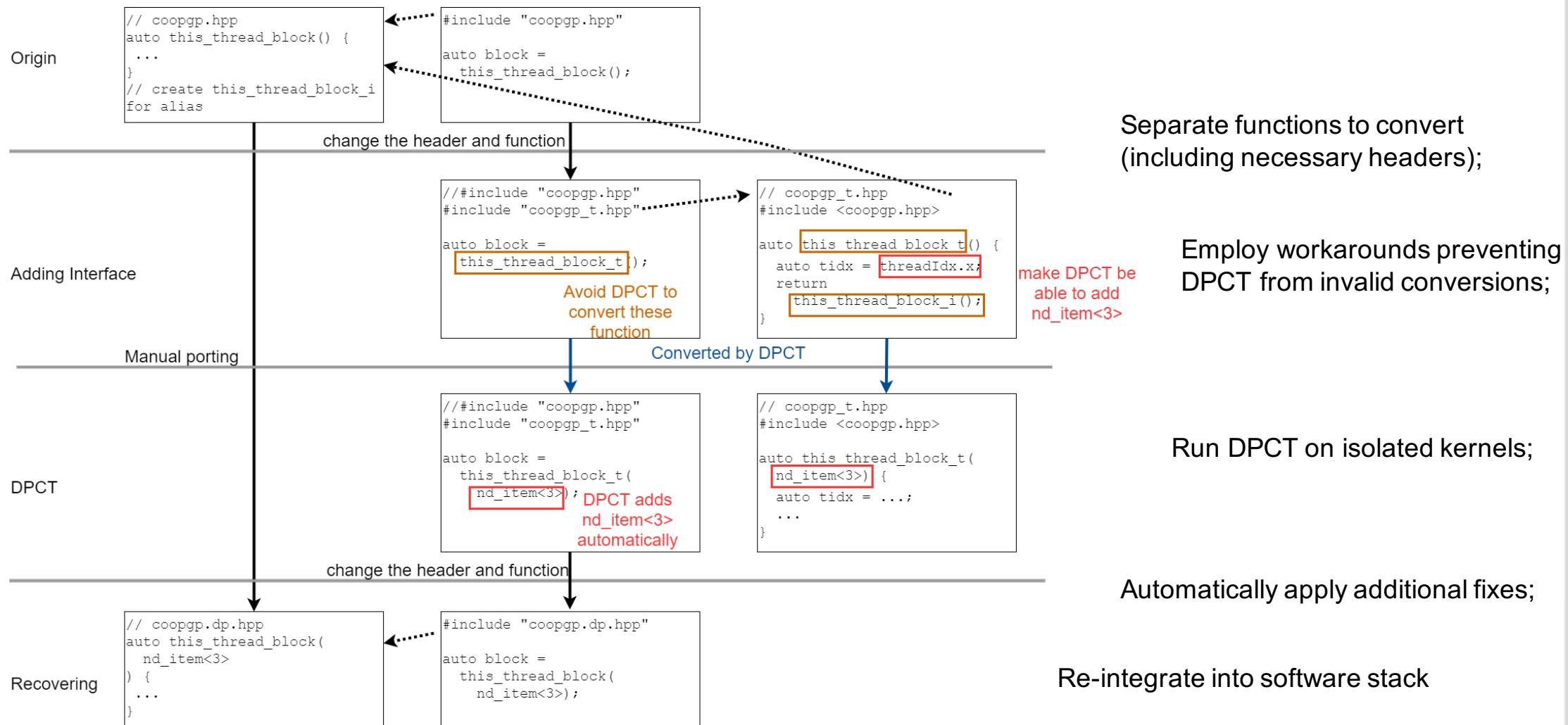
We generate the Ginkgo DPC++ backend from the CUDA backend via DPCT porting tool.

## Encountered Limitations:

- DPCT requires knowledge about all functionality dependencies -> need to mimic dependencies;
- The conversion fails for heavily-templated functionality;
- DPCT fails to convert cooperative group functionality or atomics, both needed for high performance;
- Ginkgo's software architecture requires some customized solutions;
- We want to run on diverse Intel oneAPI architectures;

<https://software.intel.com/content/www/us/en/develop/documentation/get-started-with-intel-dpcpp-compatibility-tool/top.html>

# Porting Ginkgo's CUDA Code to the oneAPI Ecosystem



# Re-engineering cooperative group calls

*DPCT fails to handle cooperative group statements*

```
template <unsigned subwarp_size, typename ValueType>
__global__ __launch_bounds__(32) void reduce(ValueType *a) {
    auto subwarp = cooperative_groups::tiled_partition<16>(
        cooperative_groups::this_thread_block());
    auto local_data = a[threadIdx.x];
    #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[threadIdx.x] = local_data;
```

*Different way to assign the group size*

*Manually insert cooperative group calls*

*Different name for “shuffle”*

```
template<unsigned subgroup_size, typename ValueType>
void reduce(ValueType *a, sycl::nd_item<3> item_ct1) {
    /*
    DPCT1007:0: Migration of this CUDA API is not supported by the Intel(R)
    DPC++ Compatibility Tool.
    */
    sycl::group<3> subwarp = item_ct1.get_sub_group();
    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;
}
```

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

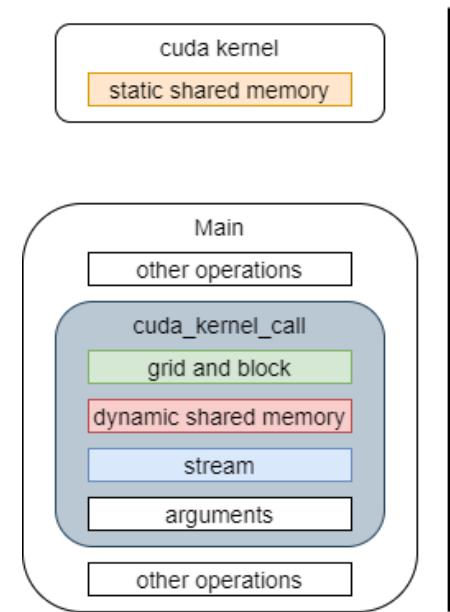
# Aim for similar backend designs

## CUDA

- In-kernel static shared memory allocation
- In-sync execution

## DPC++

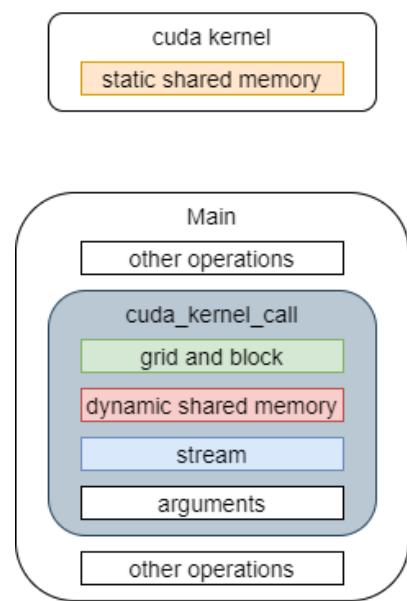
- Static shared memory allocated outside kernel
- SYCL-style asynchronous kernel execution



# Aim for similar backend designs

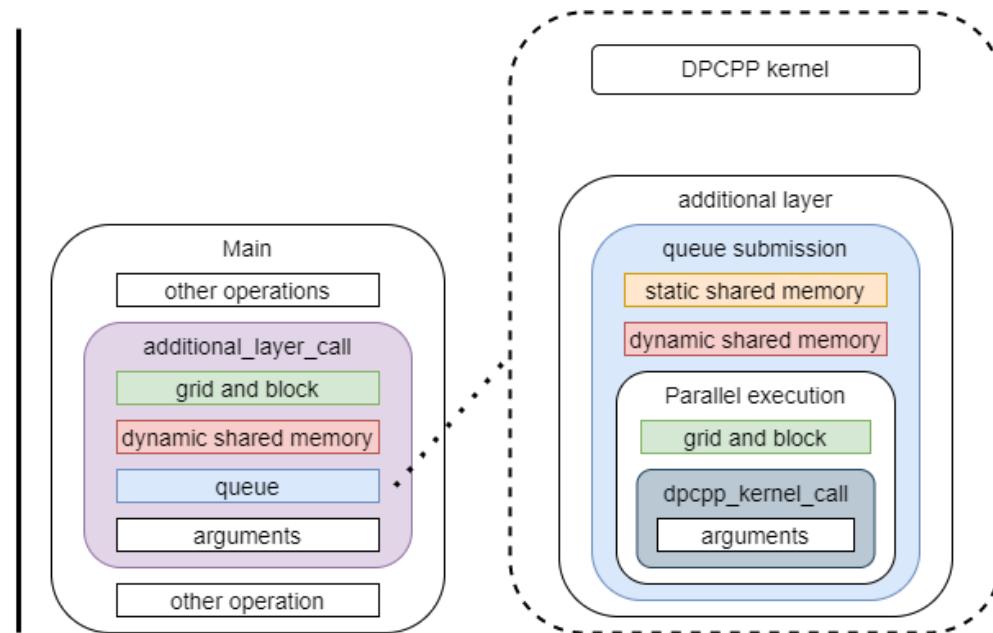
## CUDA

- In-kernel static shared memory allocation
- In-sync execution

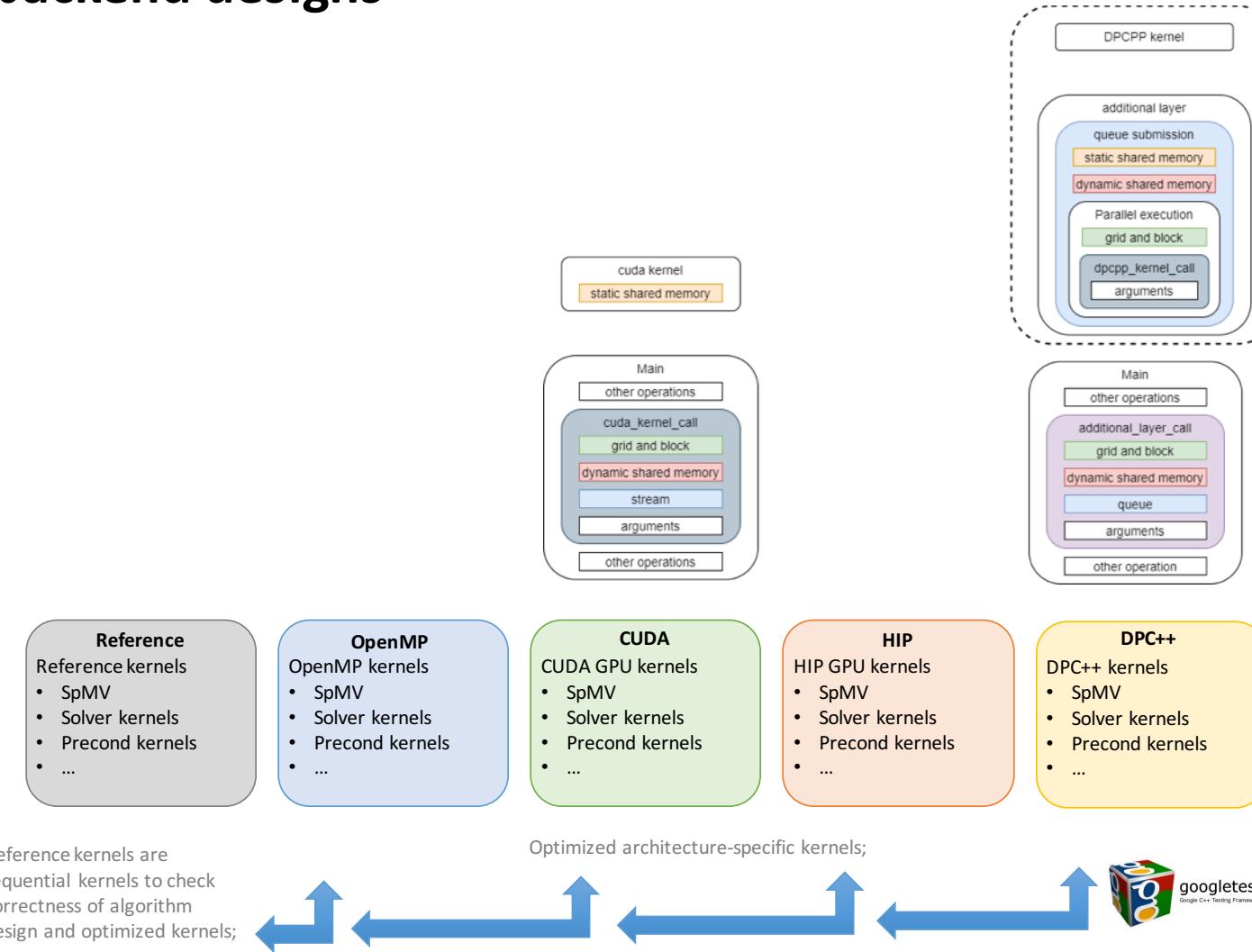


## DPC++

- Static shared memory allocated outside kernel
  - SYCL-style asynchronous kernel execution
- Mimic backend similarity by adding intermediate layer
- Synchronize streams
- Handle static shared memory allocation in intermediate layer



# Aim for similar backend designs



# The oneAPI hardware zoo

---

oneAPI/DPC++ is supported by hardware architectures with different characteristics:

- *Intel GPUs support subgroup sizes 8, 16, 32*
- *Intel CPUs support subgroup sizes 4, 8, (16)*
- *Gen9 GPUs support max workgroup size 256*
- *DG1 GPUs support max workgroup size 512*
- ...

➤ We need a way to choose a valid (and good!) configuration for all hardware architectures.

# The oneAPI hardware zoo

oneAPI/DPC++ is supported by hardware architectures with different characteristics:

- Intel GPUs support subgroup sizes 8, 16, 32
- Intel CPUs support subgroup sizes 4, 8, (16)
- Gen9 GPUs support max workgroup size 256
- DG1 GPUs support max workgroup size 512
- ...

➤ We need a way to choose a valid (and good!) configuration for all hardware architectures.

```
#if workgroup_size = 256  
    #if subgroup_size = 4  
        #if kernel_mode == 1  
            ...  
        #elif kernel_mode == 2  
            ...  
        #elif kernel_mode == 3  
            ...  
    #else  
        ...  
    #elif subgroup_size = 8  
        ...
```

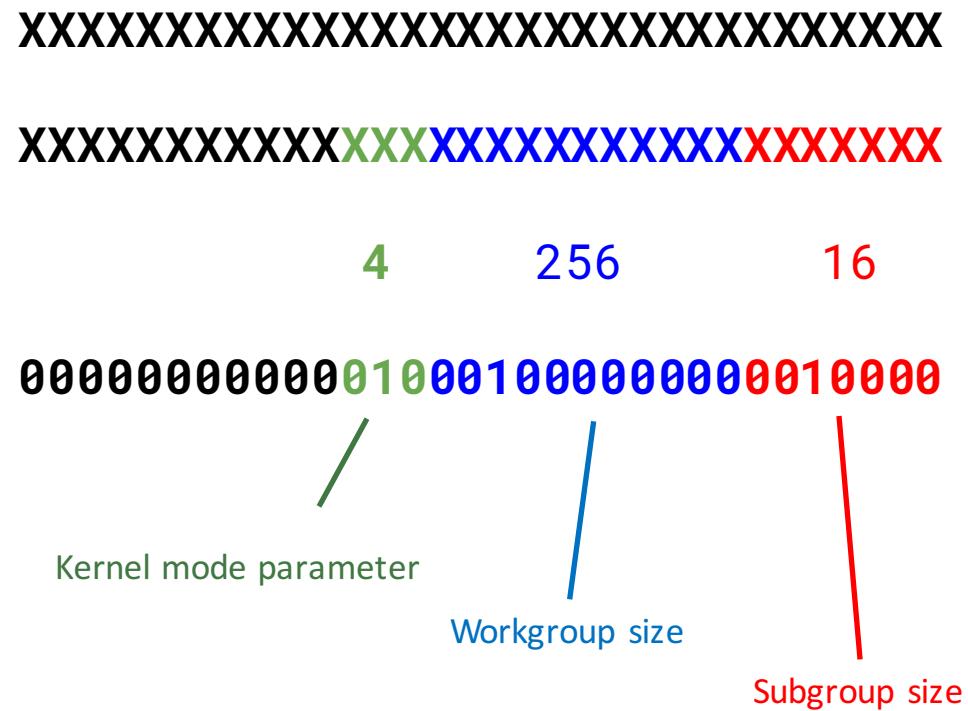
# The oneAPI hardware zoo

oneAPI/DPC++ is supported by hardware architectures with different characteristics:

- Intel GPUs support subgroup sizes 8, 16, 32
- Intel CPUs support subgroup sizes 4, 8, (16)
- Gen9 GPUs support max workgroup size 256
- DG1 GPUs support max workgroup size 512
- ...

➤ We need a way to choose a valid (and good!) configuration for all hardware architectures.

We use the ConfigSet to encode all info into one number and couple this concept with generating the kernels for all valid configs at compile time and selecting the kernel suitable for the given hardware at compile time.



# The oneAPI hardware zoo

use 32 bits

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

encoding for ConfigSet<3, 11, 7>

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

encode the config encode(4, 256, 16)

4            256            16

the result is  $4 * 2^{18} + 256 * 2^7 + 16$

0000000000010001000000000010000

 Kernel mode parameter

 Workgroup size

 Subgroup size

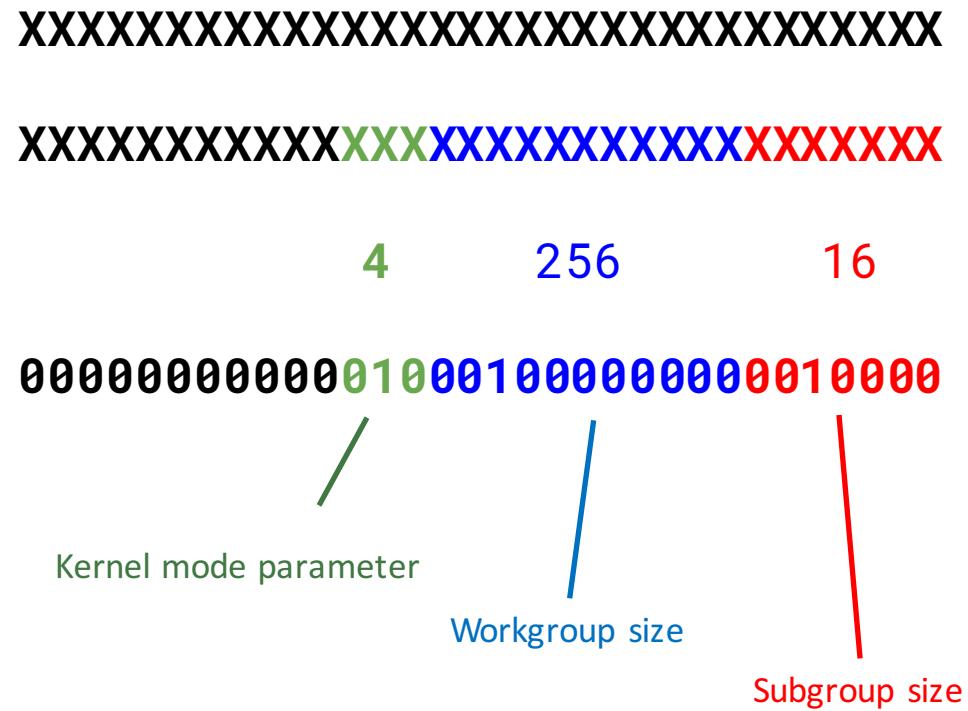
# The oneAPI hardware zoo

oneAPI/DPC++ is supported by hardware architectures with different characteristics:

- Intel GPUs support subgroup sizes 8, 16, 32
- Intel CPUs support subgroup sizes 4, 8, (16)
- Gen9 GPUs support max workgroup size 256
- DG1 GPUs support max workgroup size 512
- ...

➤ We need a way to choose a valid (and good!) configuration for all hardware architectures.

We use the ConfigSet to encode all info into one number and couple this concept with generating the kernels for all valid configs at compile time and selecting the kernel suitable for the given hardware at compile time.



# The oneAPI hardware zoo

oneAPI/DPC++ is supported by hardware architectures with different characteristics:

- Intel GPUs support subgroup sizes 8, 16, 32
- Intel CPUs support subgroup sizes 4, 8, (16)
- Gen9 GPUs support max workgroup size 256
- DG1 GPUs support max workgroup size 512
- ...

- We need a way to choose a valid (and good!) configuration for all hardware architectures.

We use the ConfigSet to encode all info into one number and couple this concept with generating the kernels for all valid configs at compile time and selecting the kernel suitable for the given hardware at compile time.

```
// give coding way
using Cfg = ConfigSet<11, 7>;
// give all possible configuration
constexpr auto cfg_list =
    value_list<std::uint32_t, Cfg::encode(512, 32), Cfg::encode(256, 16)>();

template <std::uint32_t cfg>
[[intel::reqd_work_group_size(1, 1, cfg::decode<0>(cfg))]] void kernel(args...)
{
    constexpr auto wg_size = Cfg::decode<0>(cfg);
    constexpr auto sg_size = Cfg::decode<1>(cfg);
    // it will handle [[intel::reqd_sub_group_size(sg_size)]]
    cooperative_group<sg_size>(this_thread_block(item_ct1));
    // implementation ...
}

void kernel_call(...)
{
    const auto desired_cfg =
        get_cfg(cfg_list, validate); // this is in runtime;
    // kernel_selection loops over cfg_list by template to call
    // kernel<desired_cfg>
    kernel_selection(
        cfg_list,
        [&desired_cfg](std::uint32_t cfg) { return desired_cfg == cfg; }, ...);
}
```

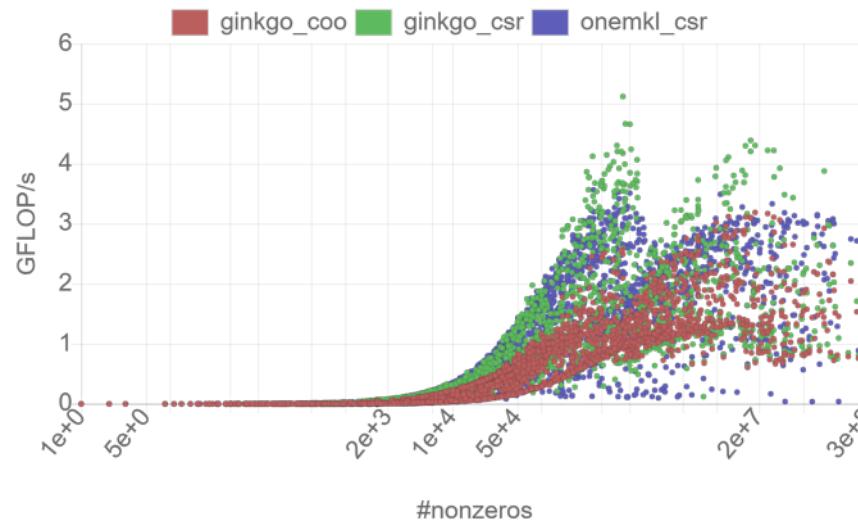
# Ginkgo's Functionality and Performance in the DPC++ ecosystem

Porting completed for:

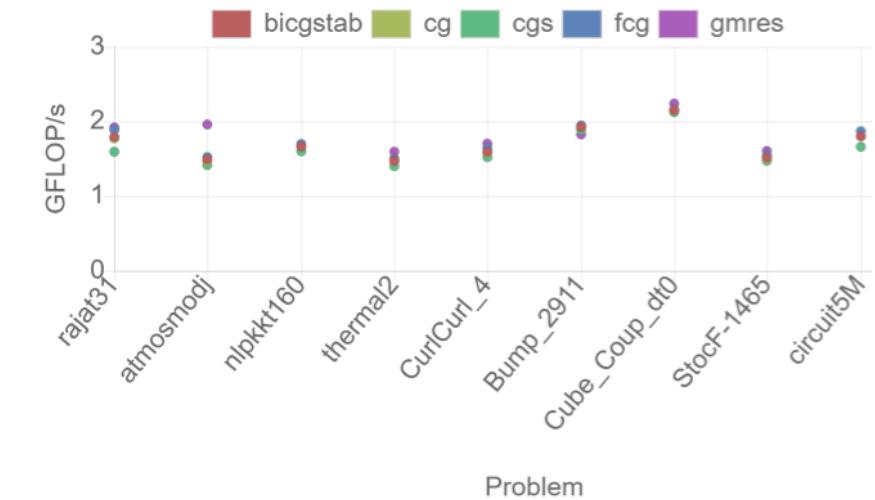
- Basic BLAS functionality
- Sparse matrix vector product
  - CSR, COO, ELL, SellP, hybrid
- Iterative linear solvers
  - CG, BiCGSTAB, CGS, FCG, GMRES



Ginkgo SpMV



Ginkgo Krylov Solvers



Porting ongoing for:

- SpGEMM, SpGEAM
- IDR Krylov solver
- Multigrid methods
- Advanced preconditioners  
(ParILU, ParILUT, block-Jacobi...)



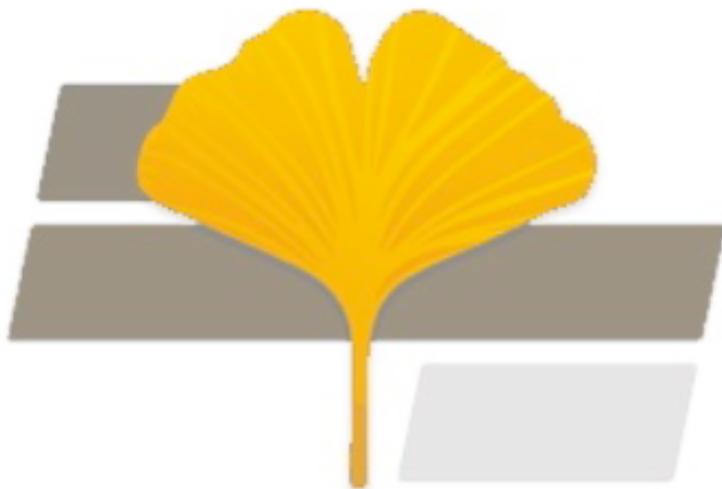
Left: Performance of Ginkgo's SpMV kernels and OneMKL SpMV on a Intel Gen9 GPU.  
Right: Performance of Ginkgo's Krylov solver on an Intel Gen9 GPU.

Test case: (All) Matrices available in the Suite Sparse Matrix Collection\* that fit the GPU memory (<3e8. nonzeros).

\*<https://people.engr.tamu.edu/davis/suitesparse.html>

# Using Ginkgo's DPC++ backend to run simulations in the oneAPI ecosystem

<https://github.com/ginkgo-project/ginkgo/blob/develop/examples/heat-equation/heat-equation.cpp>



Video realizing the heat equation simulation in the Intel DevCloud:  
<http://www.icl.utk.edu/~hanzt/slides/GinkgoHeatEqDevCloud.mp4>

```
32 ****<DESCRIPTION>*****
33 This example solves a 2D heat conduction equation
34
35  $u : [0, d]^2 \rightarrow \mathbb{R}$ 
36  $\partial_t u = \Delta u + f$ 
37
38 with Dirichlet boundary conditions and given initial condition and
39 constant-in-time source function  $f$ .
40
41 The partial differential equation (PDE) is solved with a finite difference
42 spatial discretization on an equidistant grid: For  $n$  grid points,
43 and grid distance  $h = 1/n$  we write
44
45 
$$u_{i,j}' = \alpha (u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4 u_{i,j}) / h^2 + f_{i,j}$$

46
47 We then build an implicit Euler integrator by discretizing with time step  $\tau$ 
48
49 
$$(u_{i,j}^{k+1} - u_{i,j}^k) / \tau = \alpha (u_{i-1,j}^{k+1} - u_{i+1,j}^{k+1} + u_{i,j-1}^{k+1} - u_{i,j+1}^{k+1} - 4 u_{i,j}^{k+1}) / h^2 + f_{i,j}$$

50
51 and solve the resulting linear system for  $u_{\cdot}^{k+1}$  using Ginkgo's CG
52 solver preconditioned with an incomplete Cholesky factorization for each time
53 step, occasionally writing the resulting grid values into a video file using
54 OpenCV and a custom color mapping.
55
56 The intention of this example is to provide a mini-app showing matrix assembly,
57 vector initialization, solver setup and the use of Ginkgo in a more complex
58 setting.
59 ****<DESCRIPTION>*****
```



## Next Steps

Porting completed for:

- Basic BLAS functionality
- Sparse matrix vector product
  - CSR, COO, ELL, SelLP, hybrid
- Iterative linear solvers
  - CG, BiCGSTAB, CGS, FCG, GMRES
- Simulation workflows



Porting ongoing for:

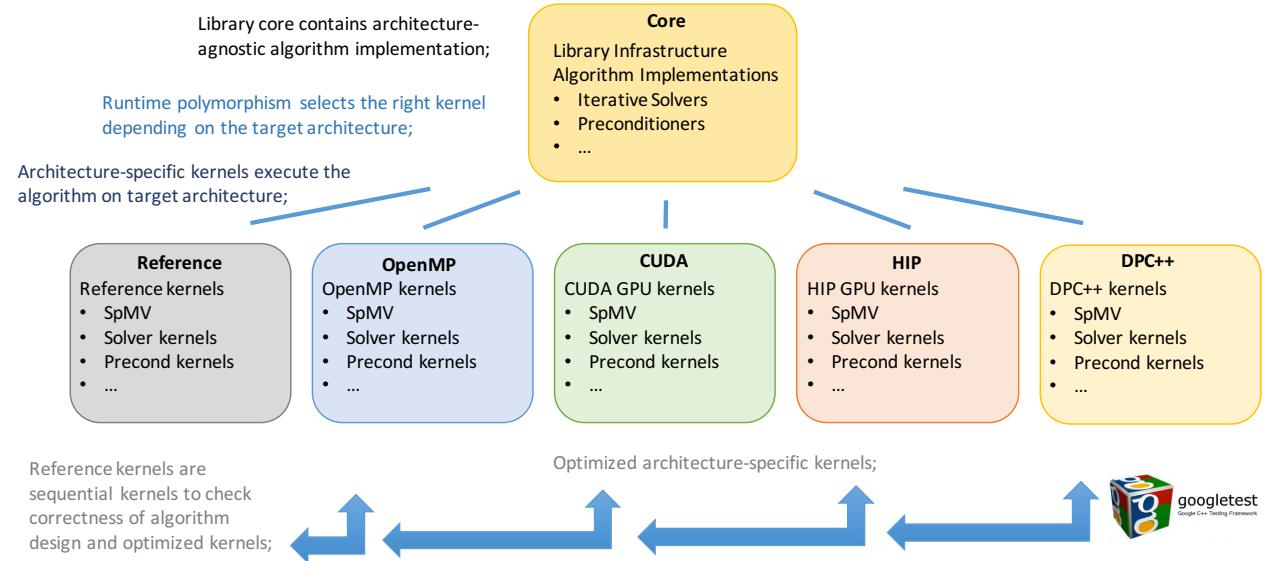
- SpGEMM, SpGEAM
- IDR Krylov solver
- Multigrid methods
- Advanced preconditioners  
(ParILU, ParILUT, block-Jacobi...)



- Evaluate performance of DPC++ backend on AMD and NVIDIA devices;



**GPU-centric high performance sparse linear algebra ecosystem.** Sustainable and extensible ecosystem with support for AMD GPUs, NVIDIA GPUs, and Intel GPUs.



We want to thank the Intel Development team for all the help we receive!

In particular, Alina Shadrina, George Silva, Sujata Tibrewala, Klaus-Dieter Oertel, Edmund Preiss, and Arti Gupta.