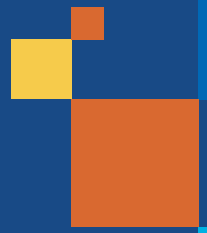
Accelerating IFS Physics Code on Intel GPUs Using oneAPI

Camilo Moreno, Intel HPC



intel®

Programming Challenges

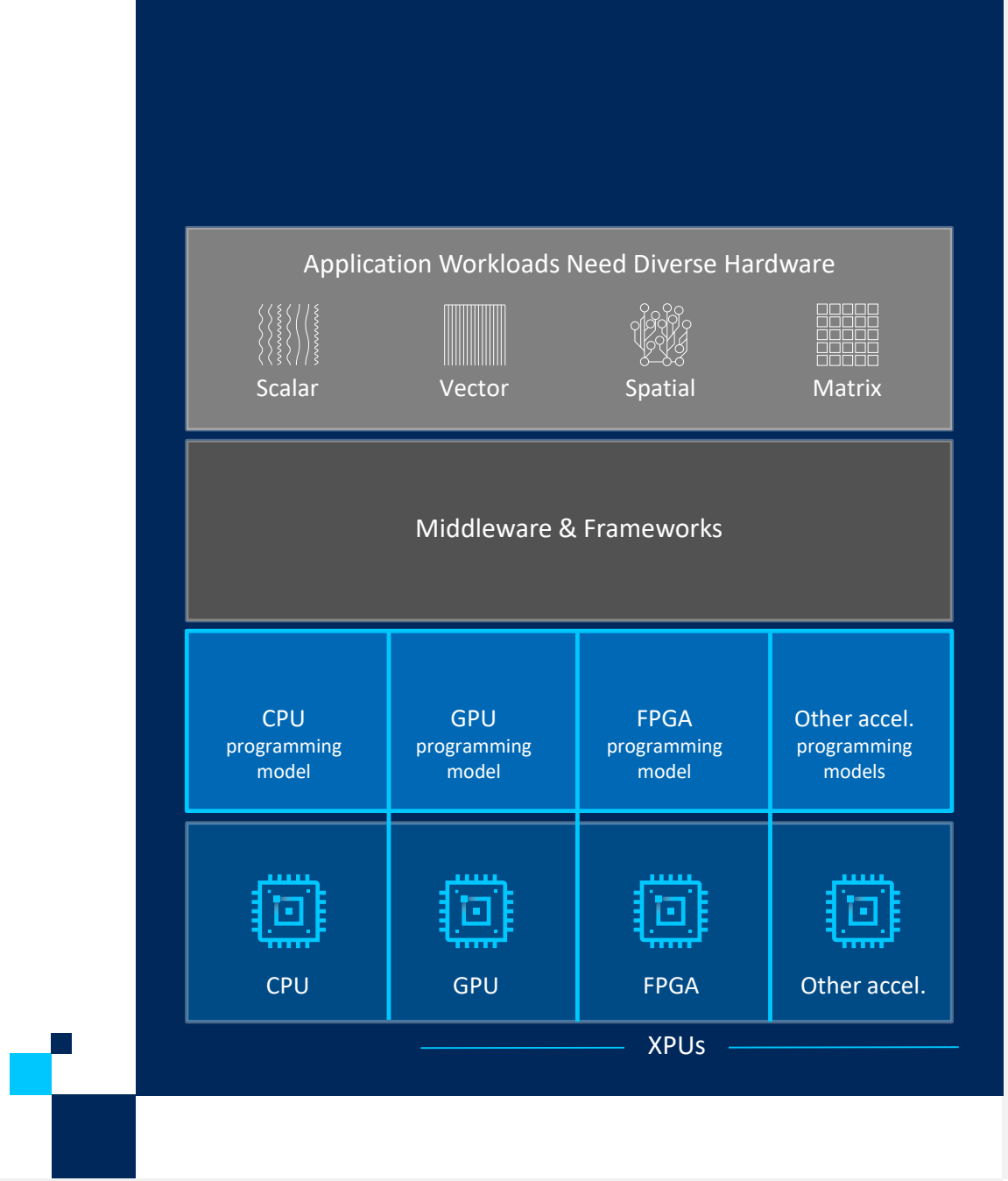
for Multiple Architectures

Growth in specialized workloads

Variety of data-centric hardware required

Separate programming models and toolchains for each architecture are required today

Software development complexity limits freedom of architectural choice



oneAPI

One Programming Model for Multiple Architectures and Vendors

Freedom to Make Your Best Choice

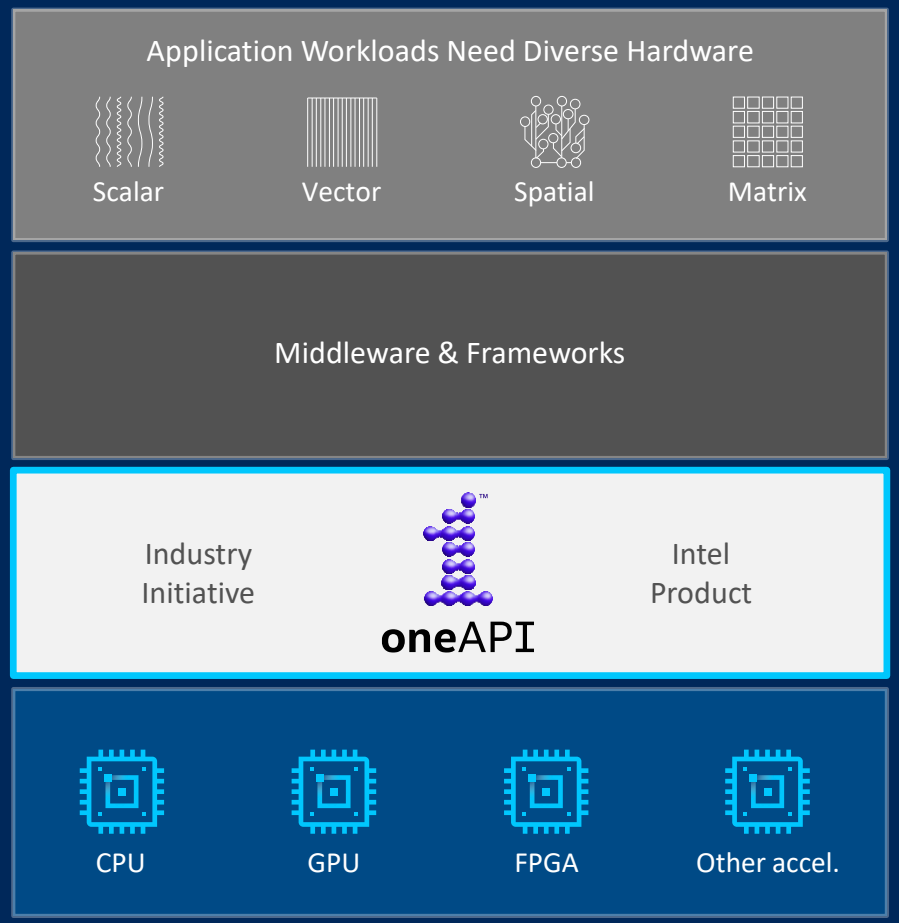
- Choose the best accelerated technology the software doesn't decide for you

Realize all the Hardware Value

- Performance across CPU, GPUs, FPGAs, and other accelerators

Develop & Deploy Software with Peace of Mind

- Open industry standards provide a safe, clear path to the future
- Compatible with existing languages and programming models including C++, Python, SYCL, OpenMP, Fortran, and MPI



Compiler Packages

Intel® oneAPI Base and HPC Toolkit

- Existing IL0 compilers ***icc***, ***icpc, ifort***

ADDED! New Compilers based on LLVM framework

- Drivers: ***dpcpp, icx****, and* ***ifx*** *(beta)*
- DPC++ and offload for OpenMP target supported

Intel® Compilers – Target & Packaging

Intel Compiler	Driver	Target*	OpenMP Support	OpenMP Offload Support	Included in oneAPI Toolkit
Intel® C++ Compiler Classic	***icc***	CPU	Yes	No	HPC, IoT
Intel® oneAPI DPC++/C++ Compiler	***dpcpp***	CPU, GPU, FPGA	Yes	Yes	Base
	icx	CPU GPU	Yes	Yes	Base
Intel® Fortran Compiler Classic	***ifort***	CPU	Yes	No	HPC
Intel® Fortran Compiler (Beta)	***ifx***	CPU, GPU	Yes	Yes	HPC

Cross-Compiler Binary Compatible and Linkable!

software.intel.com/content/www/us/en/develop/articles/oneapi-standalone-components.html

*Intel® Platforms

Data Parallel C++

Standards-based, Cross-architecture Language

DPC++ = ISO C++ and Khronos SYCL and community extensions

Freedom of Choice: Future-Ready Programming Model

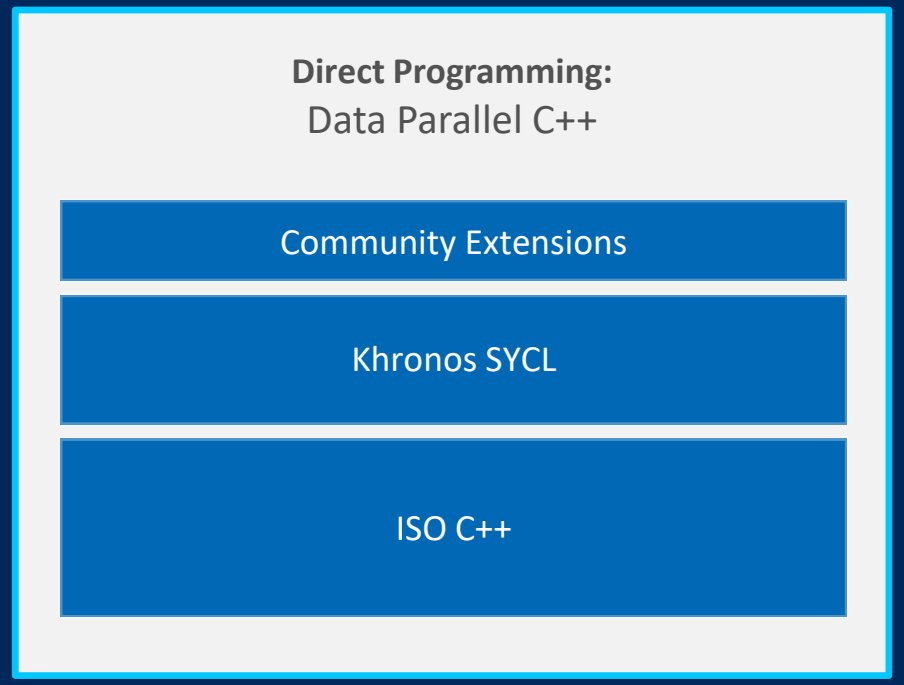
- Allows code reuse across hardware targets
- Permits custom tuning for a specific accelerator
- Open, cross-industry alternative to proprietary language

DPC++ = ISO C++ and Khronos SYCL and community extensions

- Delivers C++ productivity benefits, using common, familiar C and C++ constructs
- Adds SYCL from the Khronos Group for data parallelism and heterogeneous programming

Community Project Drives Language Enhancements

- Provides extensions to simplify data parallel programming
- Continues evolution through open and cooperative development



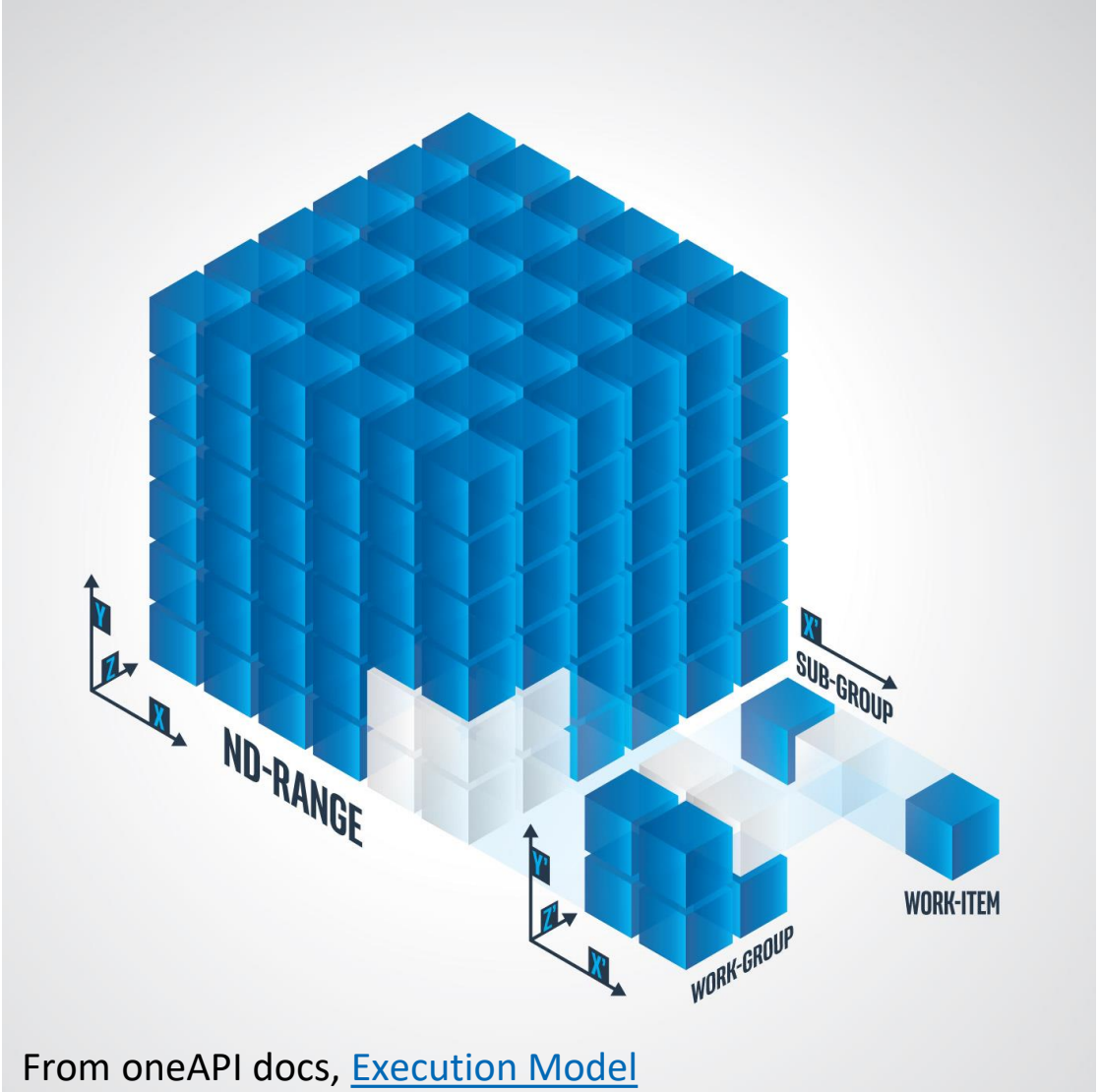
The open source and Intel DPC++/C++ compiler supports Intel CPUs, GPUs, and FPGAs.
Codeplay announced a DPC++ compiler that targets Nvidia GPUs.

oneAPI Development Platforms

- No need to own PVC / **Xe-HPC** to start development
- Can target GPUs in existing Intel laptops and *NUC*s
  - GEN9, GEN11, ***Xe-MAX***
  - Smaller and a bit lacking in HPC features, but good for early development work
- Can target CPU itself as a device for DPC++ “offload”
- Use the FPGA Emulation flow
- Can do all on Intel **DevCloud**
  - A sandbox to develop and run apps on Intel CPUs, GPUs, and FPGAs
  - Free to access, ready with oneAPI software, samples, tutorials.
  - software.intel.com/devcloud/oneapi

Basic SYCL worldview

- **Queue**s submit work to **Device**s
- **Kernel**s run on the device
- **Buffer**s hold data and **Accessor**s encode **kernel** dependencies
- **Range**s define the space of tasks or **work items** to be processed



From oneAPI docs, Execution Model

DPCPP Example

Cloudsc Dwarf App on GPU

Cloudsc Key Facts

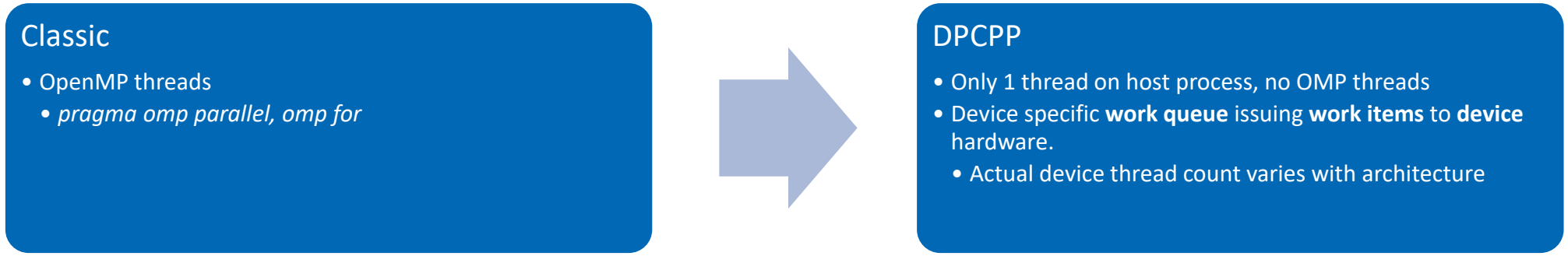
- Represents cloud microphysics code from IFS
- Dataset is a spherical shell of points, each an atmosphere column
- Only dependencies are vertical, among levels in a column
- Columns usually grouped for (vector) efficiency when running on CPU
- Early drafts of the codes used here were provided by Olivier Marsden and Michael Lange from EMCWF.

Adapting to DPC++

```
void cloudsc_driver(...) {

#pragma omp parallel ...
  {
    // ...
    int tid = omp_get_thread_num();
    // ...

#pragma omp for ...
    for (/*blocks*/) {
      cloudsc(...);
    }
  }
}

int cloudsc(...) {
  // casts for input pointers
  // stack allocate temp arrays

  for (/*levels*/) {
    for (/*columns*/) {
      // contents
    }
  }
}
```

```
void cloudsc_driver(...) {
  sycl::default_selector ds;
  sycl::queue q(ds,...) ;

  // USM device alloc inputs/outputs here
  sycl::range<1> global(nblocks*nproma);
  q.submit([&](sycl::handler &h) {
    h.parallel_for(global, [=](sycl::item<1> it) {
      cloudsc(...);
    });
  });
}


int cloudsc(...) {
  for (/*levels*/) {
    // contents
  }
}
```

Adapting to DPC++: Concurrency



- "Threads" can mean different things on different devices
- Instead, a work queue will match pending work to chosen device hardware

Adapting to DPC++

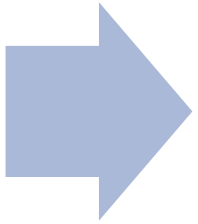
```
void cloudsc_driver(...) {

#pragma omp parallel ...
  {
    // ...
    int tid = omp_get_thread_num();
    // ...

#pragma omp for ...
    for (/*blocks*/) {
      cloudsc(...);
    }
  }
}

int cloudsc(...) {
  // casts for input pointers
  // stack allocate temp arrays

  for (/*levels*/) {
    for (/*columns*/) {
      // contents
    }
  }
}
```

```
void cloudsc_driver(...) {
  sycl::default_selector ds;
  sycl::queue q(ds,...) ;

  // USM device alloc inputs/outputs here
  sycl::range<1> global(nblocks*nproma);
  q.submit([&](sycl::handler &h) {
    h.parallel_for(global, [=](sycl::item<1> it) {
      cloudsc(...);
    });
  });
}


int cloudsc(...) {
  for (/*levels*/) {
    // contents
  }
}
```

Adapting to DPC++: Defining the Workload

Classic

- Kernel call per group of columns; group size optimized (NPROMA) for CPU performance
- Loop ordering in Kernel optimized for vectorization (levels, then columns)



DPCPP

- Use ***Range* to** index, organize full dataset in work queue
  - one column = one ***work item***
- Kernel code does one column only
  - Loop over levels in kernel

- **Ranges** support varying complexity
  - Simple or hierarchical *(*global+local); up to 3 dimensions each
- ***SYCL runtime*** will map work items to the hardware
  - On GPU, a **sub-group** may become a single “thread” on an EU, doing full SIMD ops; a **workgroup** could fully load a HW “subslice” with concurrent tasks

Adapting to DPC++

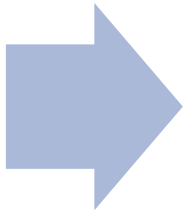
```
void cloudsc_driver(...) {

#pragma omp parallel ...
  {
    // ...
    int tid = omp_get_thread_num();
    // ...

#pragma omp for ...
    for (/*blocks*/) {
      cloudsc(...);
    }
  }
}

int cloudsc(...) {
  // casts for input pointers
  // stack allocate temp arrays

  for (/*levels*/) {
    for (/*columns*/) {
      // contents
    }
  }
}
```

```
void cloudsc_driver(...) {
  sycl::default_selector ds;
  sycl::queue q(ds,...) ;

  // USM device alloc inputs/outputs here
  sycl::range<1> global(nblocks*nproma);
  q.submit([&](sycl::handler &h) {
    h.parallel_for(global, [=](sycl::item<1> it) {
      cloudsc(...);
    });
  });
}


int cloudsc(...) {
  for (/*levels*/) {
    // contents
  }
}
```

Adapting to DPC++: Memory Model

Classic

- Simple data arrays, passed as pointers
  - Rely on CPU's coherent shared cache, etc.
- Dynamic-size, temporary arrays allocated in kernel
  - CPU can allocate memory quickly, e.g., on stack



DPCPP

- **USM** allocations, *device* scope in this case
  - Or we could use SYCL **Buffers** & **Accessors**
- Arrays defined in kernel assumed to be *private* scope for placement
  - We could have *local* arrays for workgroup scope
  - array sizes on device must be compile-time known

- SYCL 1.2.1 uses **Buffers** for data and **Accessors** for kernel dependencies
- SYCL 2020 adds Universal Shared Memory (**USM**)
  - May give up auto tracking & staging of kernel dependencies; But provide more direct control and match more established usage.
- Expressing the *scope* of allocations enables optimized data placement on hardware

Adapting to DPC++

```
tend_loc_t   = (double*) malloc( sizeof(double) * nblocks*nlev*nproma);

s_tend_loc_t   = sycl::malloc_device<double>(nblocks*nlev*nproma, cld_queue);
```

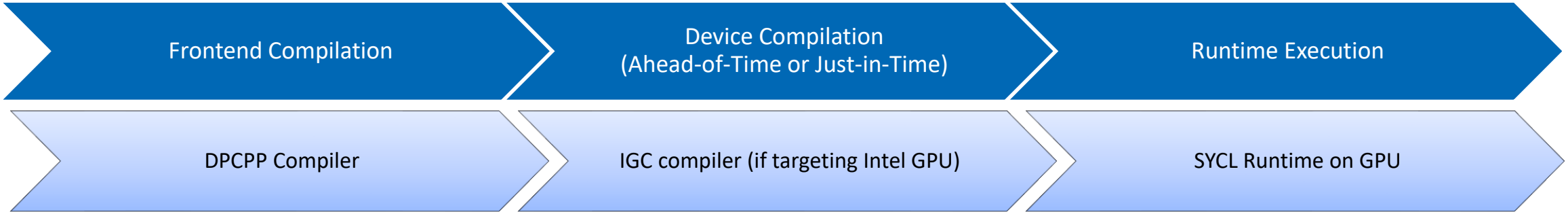
USM “device” scoped allocator

Performance Fine Tuning

- Though DPC++ reduces the need for device-specific code, it has performance tuning possibilities throughout. Here are a few examples.
  - Sycl math (OpenCL-compliant precision) vs **sycl::native** (faster, *implementation-defined* precision)
  - Multi-dimensional and hierarchical **ranges** can help capture locality & dependencies
  - **Sub-group** size can be explicitly requested; SYCL also defines efficient sub-group-level memory operations
  - Offload can be hierarchical, with **workgroup-level** code and data to set up before individual work-items, if necessary
  - API offers access to device details, allowing smart choices at runtime

Build and Run

```
// just use dpcpp instead of, e.g., icpc
dpcpp <source> -O3 -g -o <object> ...
// link into binary as usual
dpcpp <objects> -o <binaryname>
// the above will give a JIT-ready binary

// these extra flags will target GPU with Ahead-of-Time
// device compilation
aot_flags="-fsycl-targets=spir64_gen-unknown-unknown-sycldevice -Xsycl-target-backend '-device skl' "


// at runtime just call it.
// JIT will be done automatically if needed.
cloudsc 1 262144 32
// if you did AoT, the target device must match!
```



- Mostly normal Intel compiler options
- Compiles to device neutral representation linked into otherwise normal host binary

- Source, environment, or compiler options can decide target device
- IGC (Intel Graphics Compiler) is used for Intel GPU target

- Device-specific runtime maps to HW
- Use environment variables to affect runtime behavior

Checking Performance

VTune can analyze GPU offload performance

```
vtune --collect gpu-hotspots -- ./<app> <options>
```

- Report includes:
  - Utilization data on GPU Execution Units, including SIMD lane fill, stall and idle %, etc.
  - GPU cache and memory bandwidth
  - Classic “hotspots” analysis indexed to source lines
- Testing our cloud physics code on GPU
  - Single precision & small test cases to target existing non-HPC hardware
  - VTune verifies good SIMD use and memory bandwidth

Targeting Other Devices

- CPU, several options:
  - Use `cpu_selector` in the source, instead of `gpu_selector` or `default_selector`
  - Ahead-of-Time (“**AoT”**) device compilation – use CPU target options when calling DPCPP
  - Choose CPU device at JIT time with

```
export SYCL_DEVICE_FILTER=“cpu”
```

- FPGA:
  - Use `fpga_selector` in dpc++ code, then compile with:

```
dpcpp -fintelfpga [-Xshardware] …
```

- Using the `SYCL_DEVICE_FILTER` method, we test our DPCPP cloud physics “offloading” to the CPU
  - VTune shows avx512 vectorization in kernel, good utilization of available bandwidth
  - Early performance is within 10% of classic “native” FORTRAN on same system

Conclusions

- Using DPC++ we can write code in a portable manner, describing the computation without hardware-specific optimizations
- API includes many tools for special cases and for fine-tuning of performance on targeted platforms, while retaining portability
- By using open standards, we can target multiple architectures, including from different vendors

Useful Resources

- Download Intel® oneAPI Base Toolkit and Intel® oneAPI HPC Toolkit
- Free Book: Data Parallel C++ - Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL
- Get started with Intel DevCloud for oneAPI
- Industry Initiative: Data Parallel C++ (DPC++) Specification
- Learning Modules for Developing in DPC++

Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details.
No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.


intel