oneAPI DevSummit at ISC 21 | June 22, 2021

DESIGN, DEVELOPMENT, AND VALIDATION OF A DPC++ BACKEND FOR OCCA

ANOOP MADHUSOODHANAN PRABHA
CEDRIC ANDREOLLI
PHILLIPE THIERRY
Intel Corporation

SAUMIL SUDHIR PATEL
KRIS ROWE

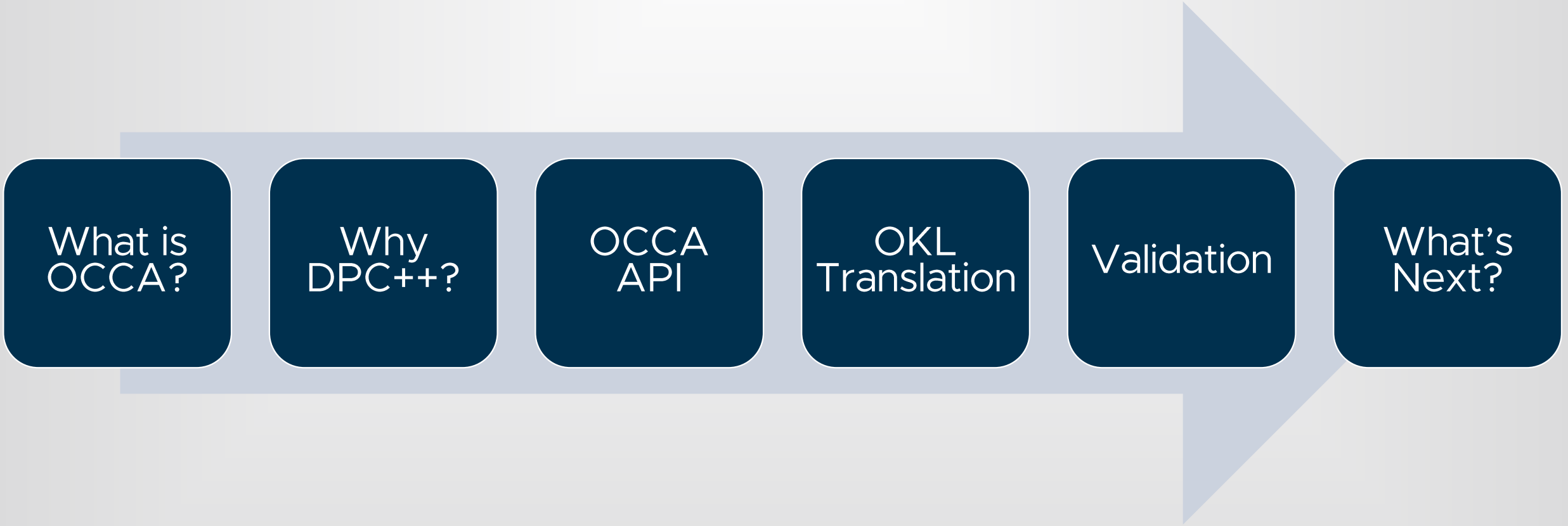
Argonne National Laboratory

SHOUT-OUTS

- David Medina (OCCA lead developer/maintainer)
- Tim Warburton (Virginia Tech, CEED)
- Michael D'mello (Intel Center of Excellence at ALCF)

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

THE BIG PICTURE



THE BIG PICTURE

WHAT IS OCCA?

Open
Concurrent
Compute
Architecture

API

- Unified models for device, memory, etc.
- Backend selection at runtime:
  - Serial, OpenMP
  - CUDA, HIP, OpenCL, Metal
- Lightweight wrappers around backend APIs
- C/C++, Fortran

Kernel Language

- Directive based extension to C/C++
- Transparent translation to backend code
- JIT compilation + caching
- Alternatively, write kernels directly with backend specific code

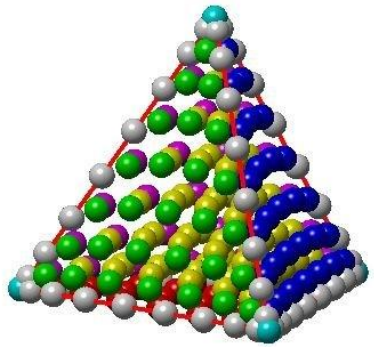
https://github.com/libocca/occa

Command Line Tool

- Hardware info
- Available backends
- Environment variables
- Translate/compile kernels

WHO USES OCCA?

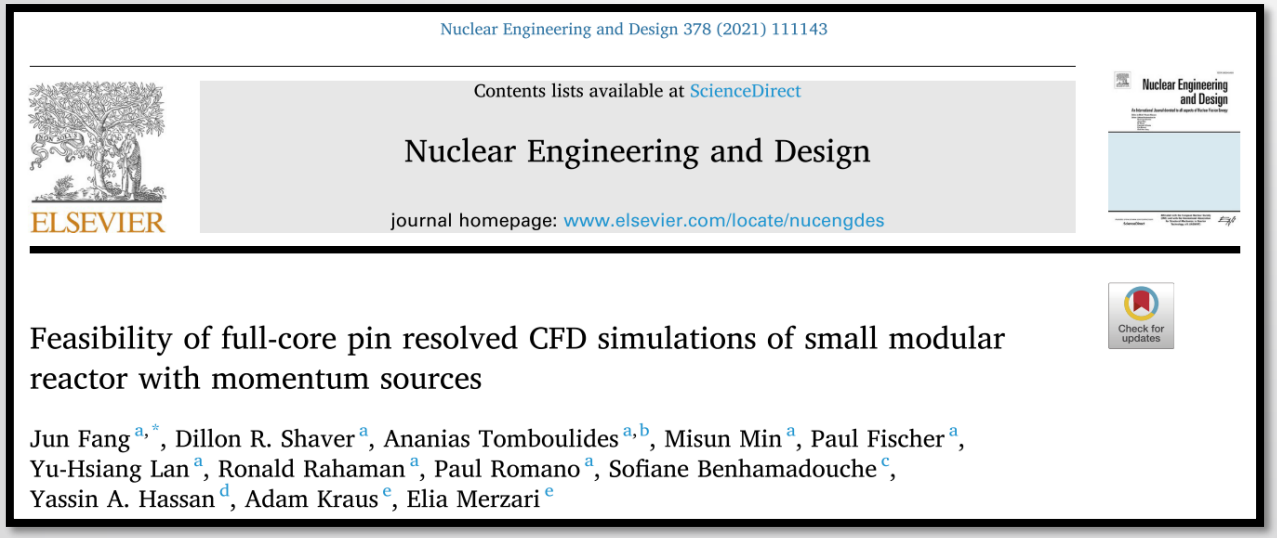
Major applications in the public and private sectors

	CEED EXASCALE DISCRETIZATIONS		NUMA
Parallel Numerical Algorithms Group • libParanumal	**Center for Exascale Discretizations** • NekRS • MFEM • Laghos	**Shell** • Full wave propagation library	**Naval Postgraduate School** • NUMA

WHERE IS OCCA USED?

Laptops to TOP500 supercomputers

First full-core, pin-resolved CFD simulation of an SMR using **NekRS** on **Summit at OLCF** (#2 on TOP500)



Nuclear Engineering and Design 378 (2021) 111143

Contents lists available at ScienceDirect

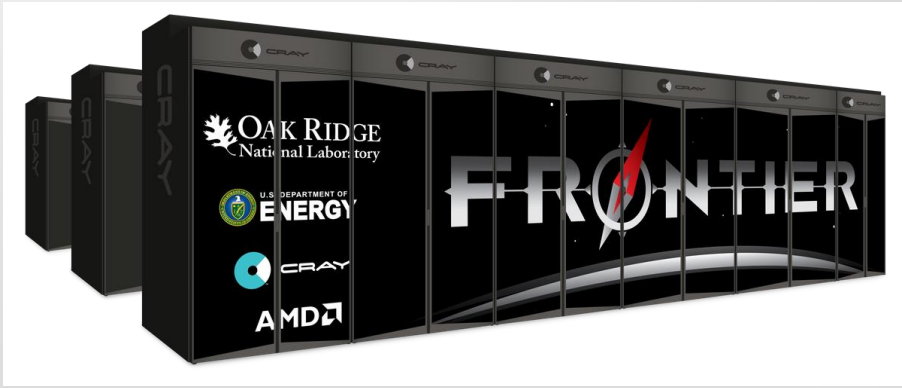
Nuclear Engineering and Design

journal homepage: www.elsevier.com/locate/nucengdes

Feasibility of full-core pin resolved CFD simulations of small modular reactor with momentum sources

Jun Fang [a,*], Dillon R. Shaver [a], Ananias Tomboulides [a,b], Misun Min [a], Paul Fischer [a], Yu-Hsiang Lan [a], Ronald Rahaman [a], Paul Romano [a], Sofiane Benhamadouche [c], Yassin A. Hassan [d], Adam Kraus [e], Elia Merzari [e]

Future Exascale Systems



USING OCCA

Example application

Application Code (Host Only)

```
1  #include <occa.hpp>
2  int main()
3  {
4    int N = 256;
5    std::vector<float> a_host(N, 1.0);
6    std::vector<float> b_host(N, 1.0);
7    std::vector<float> c_host(N);
8
9    occa::device device("{mode: 'dpcpp', device_id: 0, platform_id: 0}");
10
11   occa::memory a_device = device.malloc<float>(N);
12   occa::memory b_device = device.malloc<float>(N);
13   occa::memory c_device = device.malloc(N * sizeof(float));
14
15   a_device.copyFrom(a_host.data());
16   b_device.copyFrom(b_host.data());
17
18   occa::kernel addVectors = device.buildKernel("addVectors.okl","addVectors");
19
20   addVectors(N, a_device, b_device, c_device);
21
22   c_device.copyTo(c_host.data());
23   device.finish();
24
25   return 0;
26 }
```

Device selection

Allocate device memory

Host-to-device transfer

Create kernel

Launch kernel

Device-to-host transfer

USING OCCA

Example application

Application Code (Host Only)

```
#include <occa.hpp>
int main()
{
  int N = 256;
  std::vector<float> a_host(N, 1.0);
  std::vector<float> b_host(N, 1.0);
  std::vector<float> c_host(N);
```

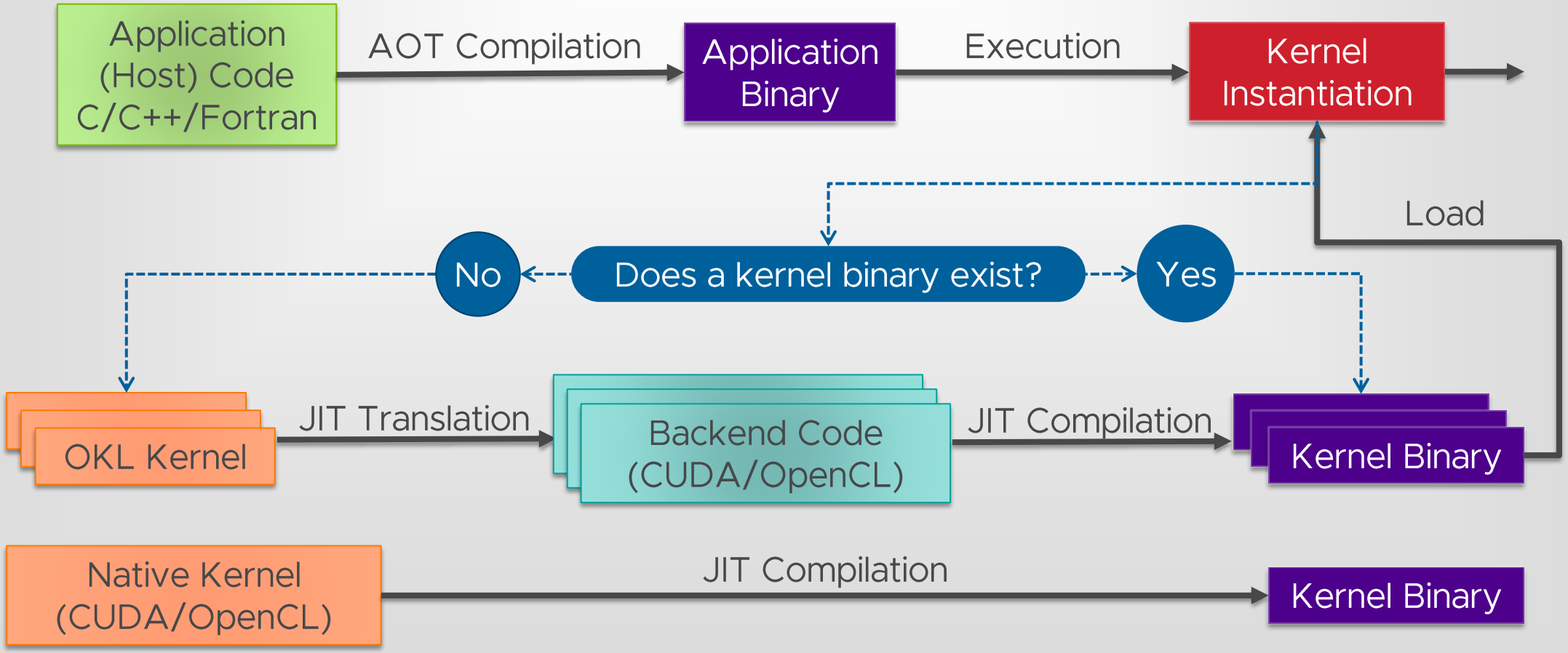
Kernel Code

```
@kernel void addVectors(const int N,
                        const float *a,
                        const float *b,
                        float *c)
{
  int constexpr group_size{16};
  const int number_of_groups = N / group_size;

  for (int group_id = 0; group_id < number_of_groups; ++group_id; @outer)
  {
    for (int local_id = 0; local_id < group_size; ++local_id; @inner)
    {
      const int i = local_id + group_size * group_id;
      c[i] = a[i] + b[i];
    }
  }
}
```

USING OCCA

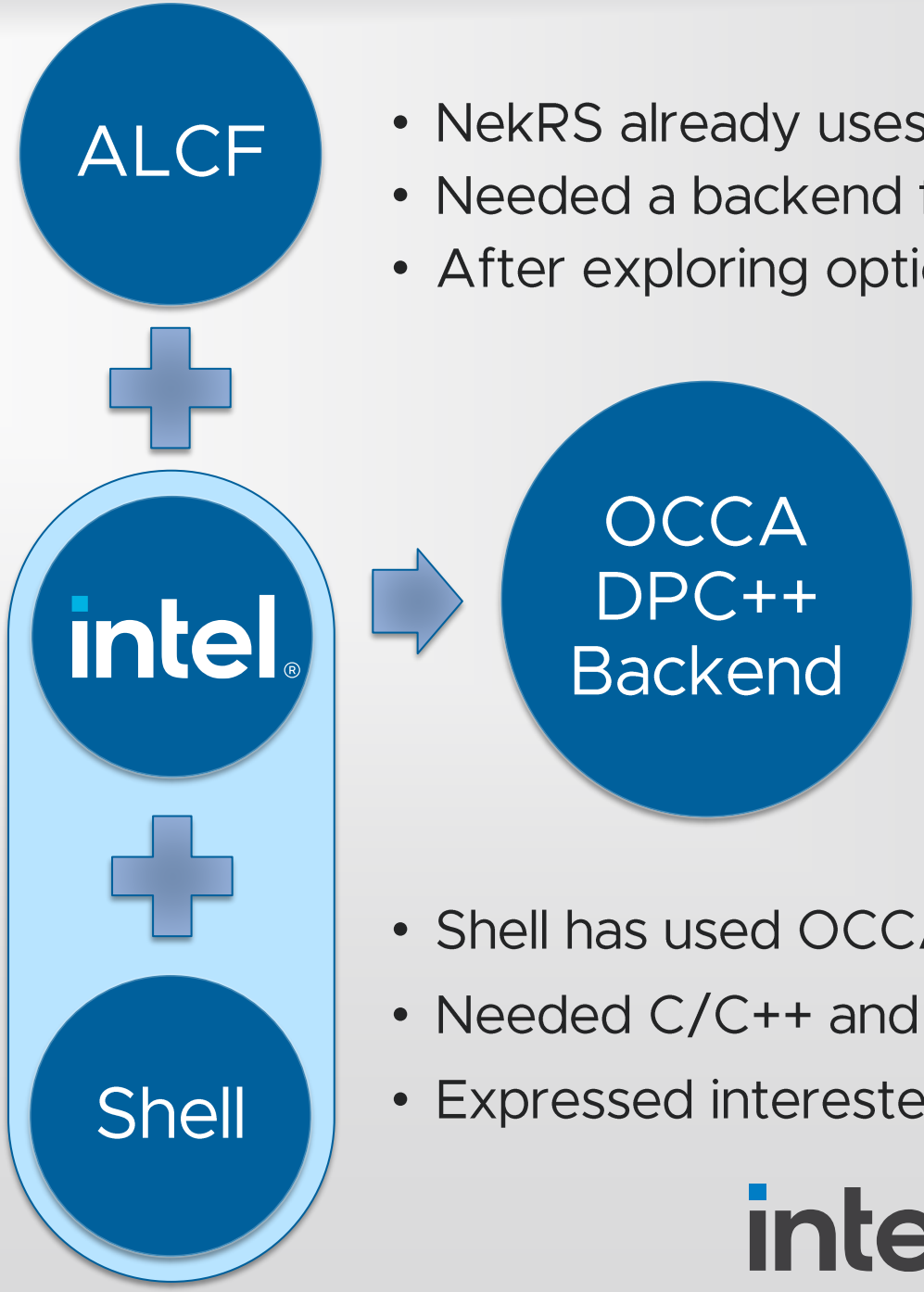
Running an application



THE BIG PICTURE

BETTER TOGETHER

A serendipitous collaboration



- NekRS already uses OCCA
- Needed a backend for Aurora
- After exploring options decided on DPC++

- Shell has used OCCA from its beginning
- Needed C/C++ and Fortran support
- Expressed interested in a DPC++ backend

Why not use DPC++ directly?

Applications already use OCCA extensively	We support but do not "own" these applications

Implementing a DPC++ backend for OCCA was the optimal path given the constraints

DEVELOPMENT APPROACH

OCCA API

- Map OCCA platform, execution, and memory models into DPC++
- Implement core OCCA runtime classes
- Verify using native DPC++ kernels

OKL

- Map OCCA programming model into DPC++
- Implement OKL to DPC++ source translator
- Verify translation using OCCA command line tool

Validate

- Microbenchmark kernels
- Mini-apps
- Full CSE applications

THE BIG PICTURE

OCCA API

Architecture overview

- Frontend provides unified models for
  - Device
  - Memory
  - Kernel
  - Stream
  - StreamTag
- OCCA uses a PIMPL design pattern
- Backends (called modes) extend base classes, implement virtual functions

OCCA API

Backend model

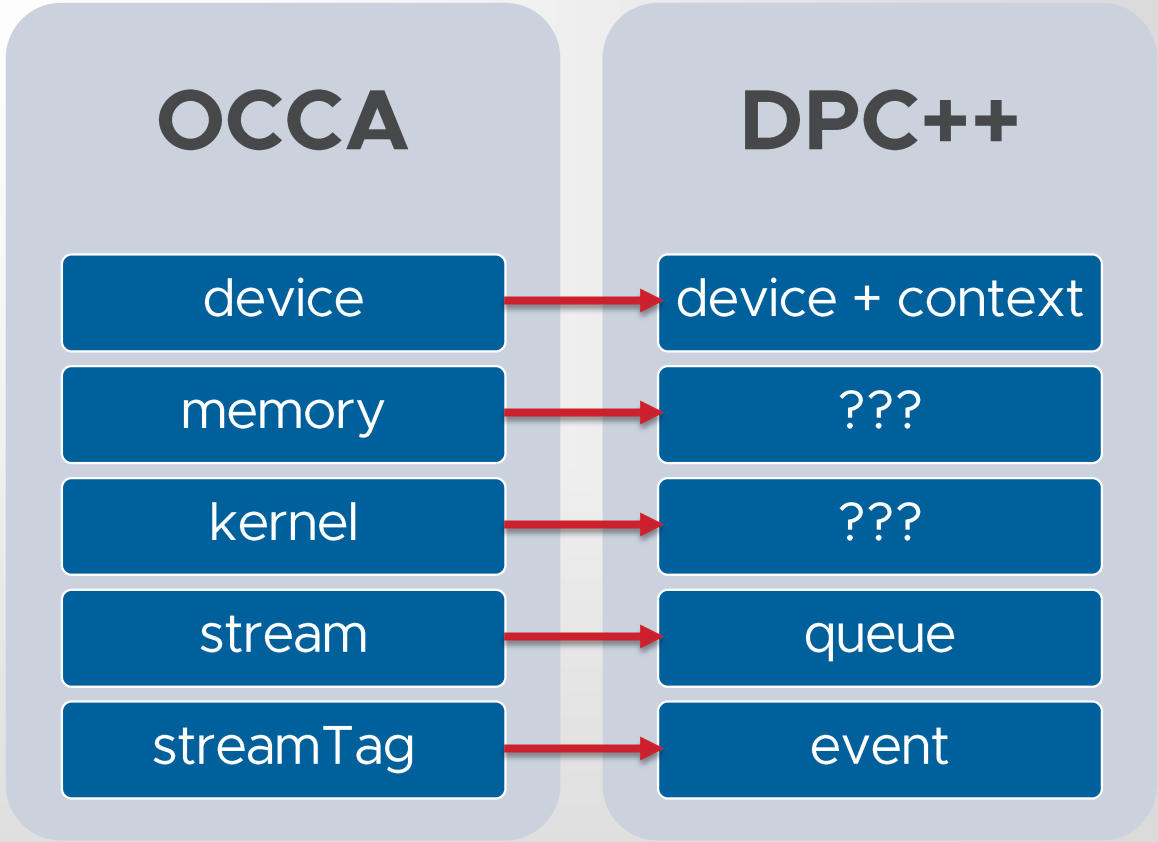
*Class names used here differ slightly from actual names in the OCCA source code

device	memory	kernel	stream	streamTag
–malloc –buildKernel –createStream –tagStream –waitFor –finish	–copyTo –copyFrom –addOffset	–setRunDims –pushArg –run –operator()		

OCCA API

Mapping OCCA models to DPC++

- There is no explicit abstraction for a
  - **Platform**
    - Equivalent to an occa mode
    - Defined via device selection
  - **Context**
    - Assume one device per context



OCCA API

DPC++ backend: memory

OBSERVATIONS

1. Device memory management is ***explicit*** via `malloc`/`free` functions
2. Host/device memory transfers are ***explicit*** via `copyTo`/`copyFrom`
3. Other backends (CUDA, OpenCL) pass memory to kernels via pointers
4. Extensive changes to OCCA's internal API would be required to use DPC++ buffers/accessors

DESIGN CHOICES

- Use DPC++ USM functions for device memory management
- Use **explicit USM** (`malloc_device`) by default for performance reasons
- More flavours of memory via extra argument to `occa::malloc`
  - *Host*: use `malloc_host`
  - *Shared*: use `malloc_shared`

OCCA API

DPC++ backend: kernels

OBSERVATIONS

1. Virtual functions are not supported in DPC++ kernels
2. We would like code in a **single translation unit**
3. OCCA supports OKL and native (backend specific code) kernels
4. Kernel launch bounds
   - Are extracted from OKL kernels
   - Must be set for native kernels

DESIGN CHOICES

- Require native kernels to be defined as `extern "C"` functions with
  - A `queue` as the first argument
  - An `nd_range` as the second
- Translate OKL kernels in the same way, adding a `queue` and `nd_range` to the argument list
- Run kernels via `sys::runFunction`

OCCA API

DPC++ backend

device	memory	kernel	stream	streamTag
–malloc –buildKernel –createStream –tagStream –waitFor –finish **–sycl::device** **–sycl::context**	–copyTo –copyFrom –addOffset **–void***	–setRunDims –pushArg –run –operator() **–void (*f)(…)**	**–sycl::queue**	**–sycl::event**

THE BIG PICTURE

OKL

Programming Model

- **OCCA Kernel Language (OKL)** can be used to write portable kernels
- OKL **attributes** are keywords prefixed by the "@" symbol
- The OKL spec defines attributes for
  - Loop parallelism
  - Memory spaces
  - Synchronization
  - Atomics
- Defined constants can be passed at runtime before JIT compilation

```
1  @kernel void matrixBlockMultiply(
2      @ restrict const double *A @dim(M, K),
3      @ restrict const double *B @dim(K, N),
4      @ restrict double *C @dim(M, N),
5      const int M,
6      const int N,
7      const int K)
8  {
9    for (int nb = 0; nb < N; nb += BLOCK_N; @outer)
10   {
11     for (int mb = 0; mb < M; mb += BLOCK_M; @outer)
12     {
13       for (int nt = 0; nt < BLOCK_N; ++nt; @inner)
14       {
15         for (int mt = 0; mt < BLOCK_M; ++mt; @inner)
16         {
17           double C_ij = 0.0;
18           for (int k = 0; k < K; ++k)
19           {
20             C_mn += A(mb + mt, k) * B(k, nb + nt);
21           }
22           C(mb + nt, nb + nt) = C_mn;
23         }
24       }
25     }
26   }
27 }
```

OCCA Kernel (OKL)

```
@kernel void matrixBlockMultiply(
    @ restrict const double *A @dim(M, K),
    @ restrict const double *B @dim(K, N),
    @ restrict double *C @dim(M, N),
    const int M,
    const int N,
    const int K)
{
  for (int nb = 0; nb < N; nb += BLOCK_N; @outer)
  {
    for (int mb = 0; mb < M; mb += BLOCK_M; @outer)
    {
      for (int nt = 0; nt < BLOCK_N; ++nt; @inner)
      {
        for (int mt = 0; mt < BLOCK_M; ++mt; @inner)
        {
          double C_ij = 0.0;
          for (int k = 0; k < K; ++k)
          {
            C_mn += A(mb + mt, k) * B(k, nb + nt);
          }
          C(mb + nt, nb + nt) = C_mn;
        }
      }
    }
  }
}
```

JIT Translation

CUDA

```
extern "C" __global__ void _occa_matrixBlockMultiply_0(const double *__restrict__ A,
                                                       const double *__restrict__ B,
                                                       double *__restrict__ C,
                                                       const int M,
                                                       const int N,
                                                       const int K)
{
  {
    int nb = 0 + (8 * blockIdx.y);
    {
    }
  }
}
```

OpenCL

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable

__kernel void _occa_matrixBlockMultiply_0(__global const double * restrict A,
                                          __global const double * restrict B,
                                          __global double * restrict C,
                                          const int M,
                                          const int N,
                                          const int K) {
  {
    int nb = 0 + (8 * get_group_id(1));
    {
      int mb = 0 + (32 * get_group_id(0));
      {
        int nt = 0 + get_local_id(1);
        {
          int mt = 0 + get_local_id(0);
          double C_ij = 0.0;
          for (int k = 0; k < K; ++k) {
            C_mn += A[mb + mt + (M * k)] * B[k + (K * (nb + nt))];
          }
          C[mb + nt + (M * (nb + nt))] = C_mn;
        }
      }
    }
  }
}
```

OKL

DPC++ backend

- OCCA has extensive infrastructure for parsing/transforming source code
- An abstract class interface is provided for translating OKL to backend code
- Developer prescribes how to
  - Translate kernel argument and return types
  - Map attributes

- Recall: we required OCCA kernel functions to take a `queue` and `nd_range` as arguments
- Using lambdas for the DPC++ command group and kernel functions is straightforward
- Define the `nd_range`:
  - Local range = `@inner` dimensions
  - Global range = `@inner` x `@outer`

```
@kernel void matrixBlockMultiply(
    @ restrict const double *A @dim(M, K),
    @ restrict const double *B @dim(K, N),
    @ restrict double *C @dim(M, N),
    const int M,
    const int N,
    const int K)
{
  for (int nb = 0; nb < N; nb += BLOCK_N; @outer)
  {
    for (int mb = 0; mb < M; m
    {
      for (int nt = 0; nt < BL
      {
        for (int mt = 0; mt < BLOCK_M; ++mt; @inner)
        {
          double C_ij = 0.0;
          for (int k = 0; k < K; ++k)
          {
            C_mn += A(mb + mt, k) * B(k, nb + nt);
          }
          C(mb + nt, nb + nt) = C_mn;
        }
      }
    }
  }
}
```

OKL → DPC++

```
#include <CL/sycl.hpp>

extern "C" void _occa_matrixBlockMultiply_0(sycl::queue *queue_,
                                            sycl::nd_range<3> *range_,
                                            const double *__restrict__ A,
                                            const double *__restrict__ B,
                                            double *__restrict__ C,
                                            const int &M,
                                            const int &N,
                                            const int &K)
{
  queue_->submit([&](sycl::handler &handler_)
  {
    handler_.parallel_for(*range_,[=](sycl::nd_item<3> item_)

                                  roup(1));

                                  get_group(2));
      {
        int nt = 0 + i_.get_local_id(1);
        {
          int mt = 0 + item_.get_local_id(2);
          double C_ij = 0.0;
          for (int k = 0; k < K; ++k)
          {
            C_mn += A[mb + mt + (M * k)] * B[k + (K * (nb + nt))];
          }
          C[mb + mt + (M * (nb + nt))] = C_mn;
        }
      }
    }
  });
});
}
```

```
@kernel void matrixBlockMultiply(
    @ restrict const double *A @dim(M, K),
    @ restrict const double *B @dim(K, N),
    @ restrict double *C @dim(M, N),
    const int M,
    const int N,
    const int K)
{
  for (int nb = 0; nb < N; nb += BLOCK_N; @outer)
  {
    for (int mb = 0; mb < M; mb += BLOCK_M; @outer)
    {
      for (int nt = 0; nt < BLOCK_N; ++nt; @inner)
      {
        for (int mt = 0; mt < BLOCK_M; ++mt; @inner)
        {
          double C_ij = 0.0;
          for (int k = 0; k < K; ++k)
          {
            C_mn += A(mb + mt, k) * B(k, nb + nt);
          }
          C(mb + nt, nb + nt) = C_mn;
        }
      }
    }
  }
}
```

```
#include <CL/sycl.hpp>

extern "C" void _occa_matrixBlockMultiply_0(sycl::queue *queue_,
                                            sycl::nd_range<3> *range_,
                                            const double *__restrict__ A,
                                            const double *__restrict__ B,
                                            double *__restrict__ C,
                                            const int &M,
                                            const int &N,
                                            const int &K)
{
  queue_->submit([&](sycl::handler &handler_)
  {
    handler_.parallel_for(*range_,[=](sycl::nd_item<3> item_)
    {
      int nb = 0 + (8 * item_.get_group(1));
      {
        int mb = 0 + (32 * item_.get_group(2));
        {
          int nt = 0 + item_.get_local_id(1);
          {
            int mt = 0 + item_.get_local_id(2);
            double C_ij = 0.0;
            for (int k = 0; k < K; ++k)
            {
              C_mn += A[mb + mt + (M * k)] * B[k + (K * (nb + nt))];
            }
            C[mb + mt + (M * (nb + nt))] = C_mn;
          }
        }
      }
    });
  });
}
```

OKL

DPC++ backend: local memory

- Currently, no free functions exists to allocate local memory in a Kernel
  - We are forced to use local accessors
- OCCA device functions frequently use pointer arguments
  - Passing local memory directly via accessor is not compatible
  - *Workaround*: get the raw pointer to memory from the accessor

OKL

```
@shared volatile double s_vec[block];
```

DPC++

```
auto s_vec_acc = sycl::accessor<double,1,
                 sycl::access::mode::read_write,
                 sycl::access::target::local>({256},handler_);

volatile double * s_vec;
s_vec = s_vec_acc.get_pointer().get();
```

THE BIG PICTURE

VALIDATION

OCCA API + OKL

- OCCA's test harness
- Benchmark kernels
  - stream
  - reduction
  - matrix transpose
  - matrix multiply
- Mini-apps: NekBench, ISO3DFD
- NekRS: example cases (e.g., "turbPipe")

- Initial performance is comparable to
  - Using DPC++ directly
  - The OCCA OpenCL backend

All tests were performed on Intel GPUs using the oneAPI Base Toolkit

THE BIG PICTURE

NEXT STEPS

Current Efforts

- OCCA DPC++ backend available on GitHub in near future
- Technical report on development of DPC++ backend
- OCCA performance benchmarks
- **Performance study**—across vendor hardware, all backends
  - *What is the price paid for using a portability framework?*

Future Work

- Propose API for performance features
  - subdevices
  - thread-block/group collectives
  - asynch mem copies (i.e., global ↔ shared/local)

To learn more about OCCA visit https://libocca.org

For source code and ways to get involved checkout https://github.com/libocca/occa

TAKEAWAYS

OCCA is used by mission critical applications in the public and private sector

A new OCCA DPC++ backend was the shortest path to running on Intel GPUs

The DPC++ USM model was central to in designing the new OCCA backend

Local memory use in kernels required workarounds: no free function is available currently

OCCA's test suite, benchmark kernels, and NekRS were used for validation

Understanding and optimizing performance is the next focus

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.