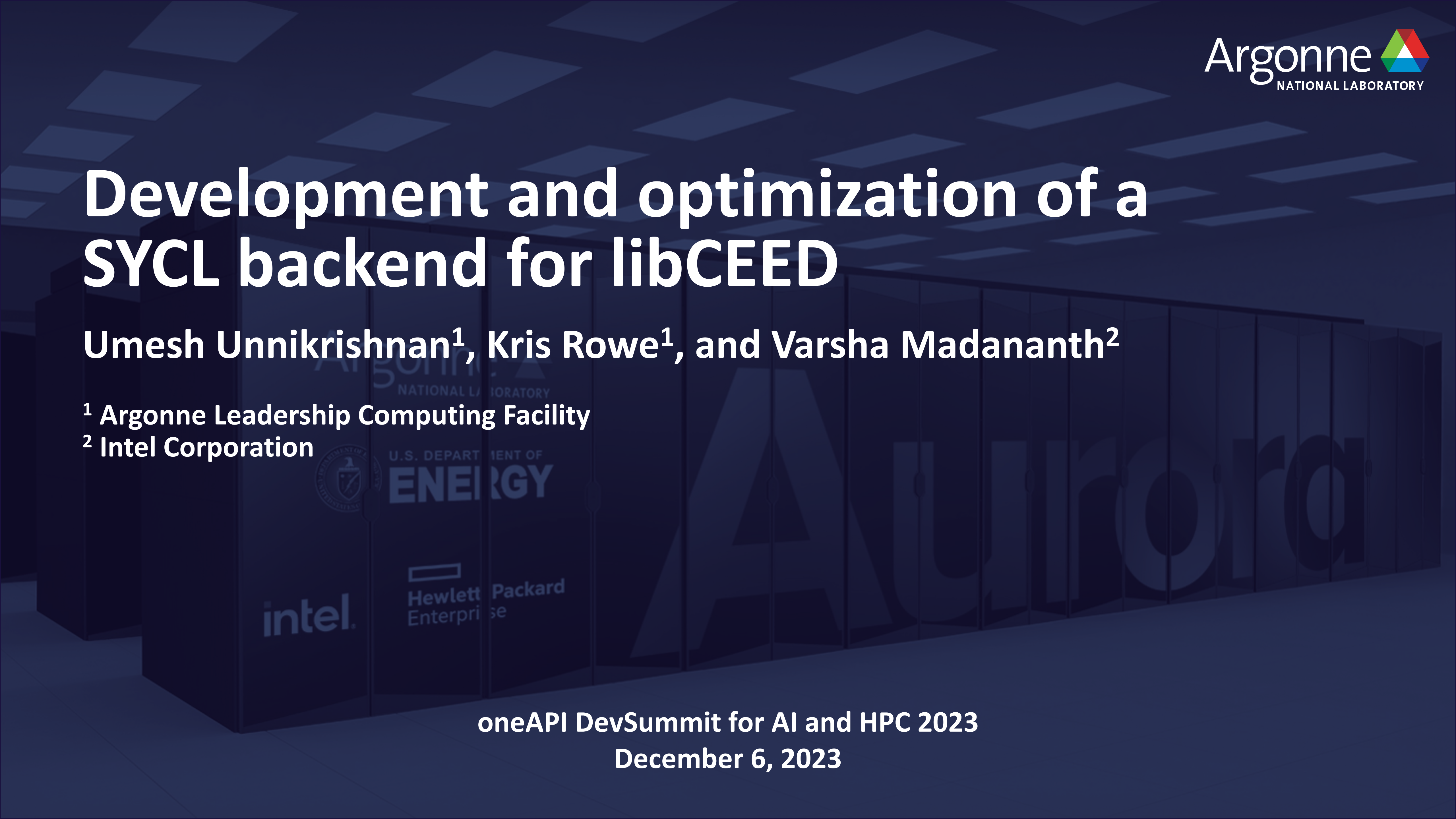


Development and optimization of a SYCL backend for libCEED

Umesh Unnikrishnan[1], Kris Rowe[1], and Varsha Madananth[2]

[1] Argonne Leadership Computing Facility
[2] Intel Corporation

oneAPI DevSummit for AI and HPC 2023
December 6, 2023

Outline

- ➢ Software design of libCEED
- ➢ SYCL online compiler
- ➢ Optimization of hotspot kernels
  - ❑ Specialization constant
  - ❑ Tuning workgroup sizes/barriers
  - ❑ SIMD size/register width
- ➢ Summary and Future work

Disclaimer: This work was done on a pre-production supercomputer with early versions of the Aurora software development kit.

libCEED

Developed through CEED co-design center as part of the ECP
courtesy: Jed Brown, Natalie Beams and others.

https://github.com/CEED/libCEED

Library

- C language
- Element-based discretizations
- low & high-order

Interfaces

- Fortran
- Python
- Julia
- Rust

Backends

- Runtime selection
- CPU—serial, AVX, LIBXSMM
- GPU (Native)—CUDA, HIP
- MAGMA, OCCA**

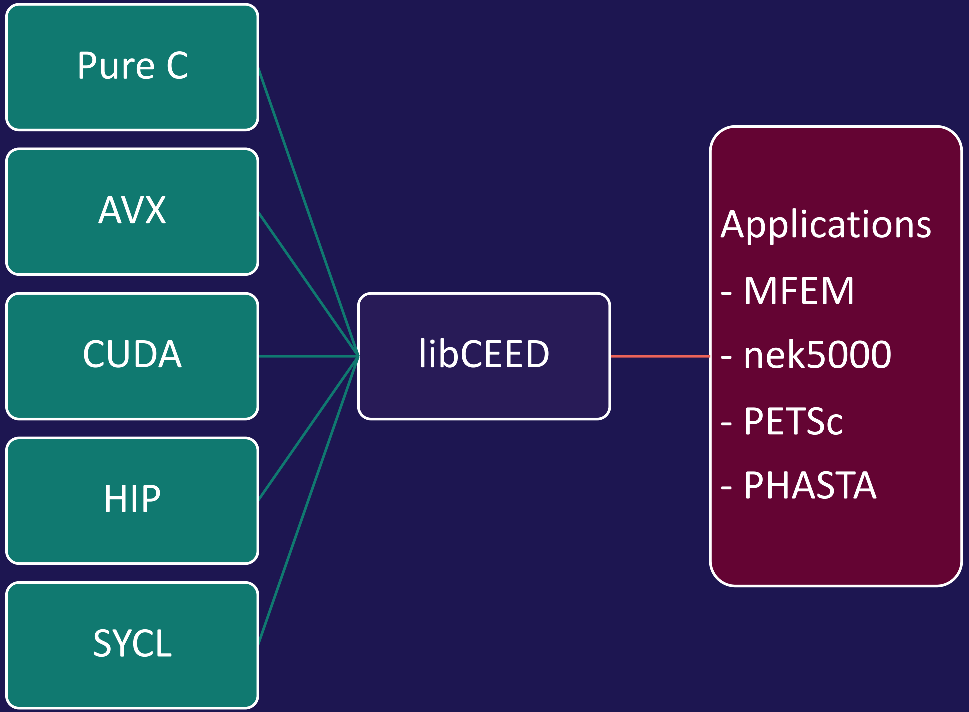
Implementation

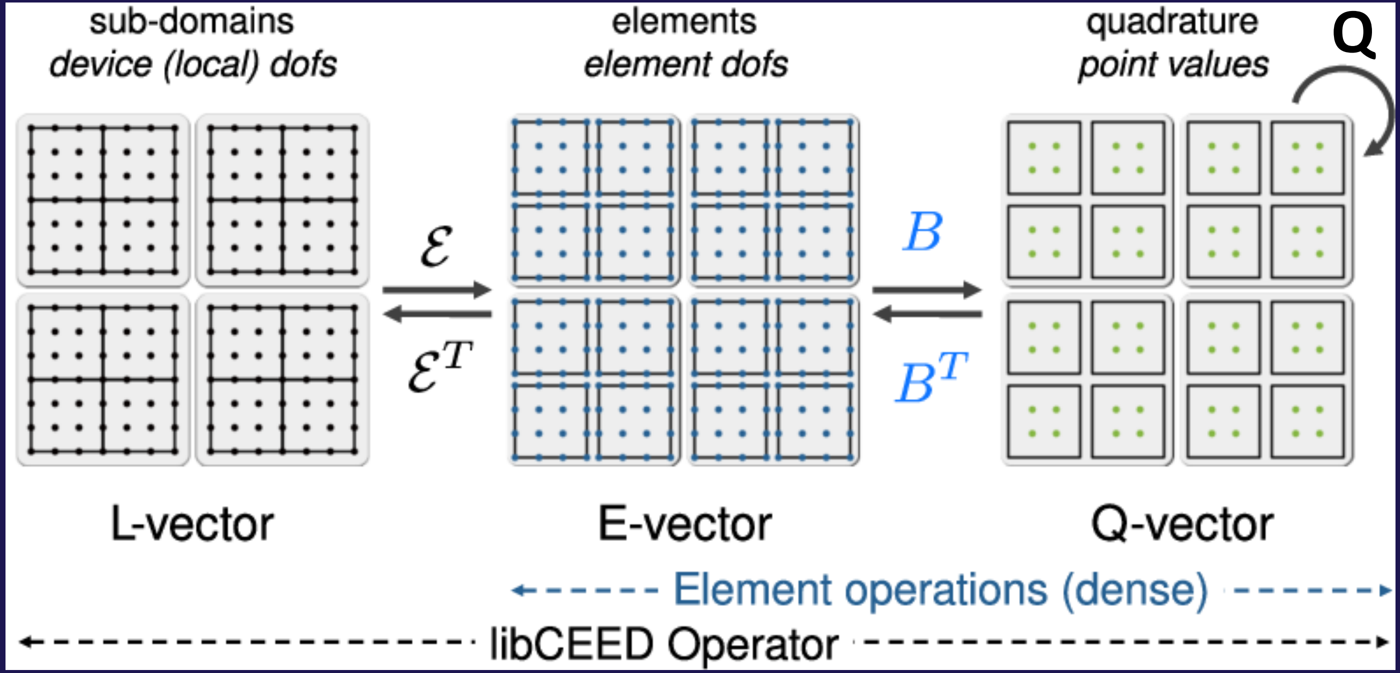
- PIMPL Idiom
- Host code is serial
- User handles MPI

Overview of libCEED operations

- Portable library that provides an API for applications to share efficient kernels for element-based discretizations.

$$\boxed{A = E^T B^T Q B E}$$





https://github.com/CEED/libCEED

libCEED Runtime Compilation Usage

Problem Parameters	Constants	polynomial order
		spatial dimension
	Determine loop-bounds	element count
		node count
User Functions	Operate pointwise at mesh nodes	Boundary conditions
		Forces
	Injected into kernels	Problem-dependent
		Avoids memory traffic

Runtime code compilation in libCEED

- libCEED host application generates device code at runtime based on user input (the code is in stringstream) – compiles and runs on GPU devices.
- CUDA backend uses nvrtc for runtime compilation of generated kernels.
- Two possible solutions for SYCL –
  - ➢ Using specialization constant (a partial alternative)
  - ➢ Using Intel's oneAPI online compilation extension (experimental early access) –

    https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_intel_online_compiler.asciidoc

Existing CUDA Backend

AOT Compilation

- Uses nvcc
- Some (few) kernels

Online Compilation

- Uses NVRTC
- Kernel source string is generated
- Boilerplate code is provided (e.g., read/writes with global mem)
- User source code is loaded from file
- Create BP→UserDefinedFunc→BP "sandwich"
- Load Cumodule, use CUfunction

Designing the SYCL Backend

Problem Parameters	Specialization constants
	Define kernels using lambdas or functors
	CUDA module can be replaced by SYCL kernel bundle
User Functions	How to compile source code string? (Ideal)
	How to link compiled user functions? (Fallback)

libCEED SYCL Online Compilation

- The libCEED CUDA and HIP backends use their respective vendor runtime compilation libraries (e.g., nvrtc, hiprtc).
- The current SYCL spec doesn't prescribe similar functionality.
- An Intel extension for SYCL allows for the runtime compilation of OpenCL C source.
  - Provided to SYCL API as a std::string
- Restriction to OpenCL C required some workarounds when porting libCEED jit source.
  - E.g., No templates, no function pointers
  - Also required some workarounds in QFunction implementation.
- We are working with Intel compiler team to help drive the online compiler extension forward.
- Future versions may support runtime compilation of SYCL source

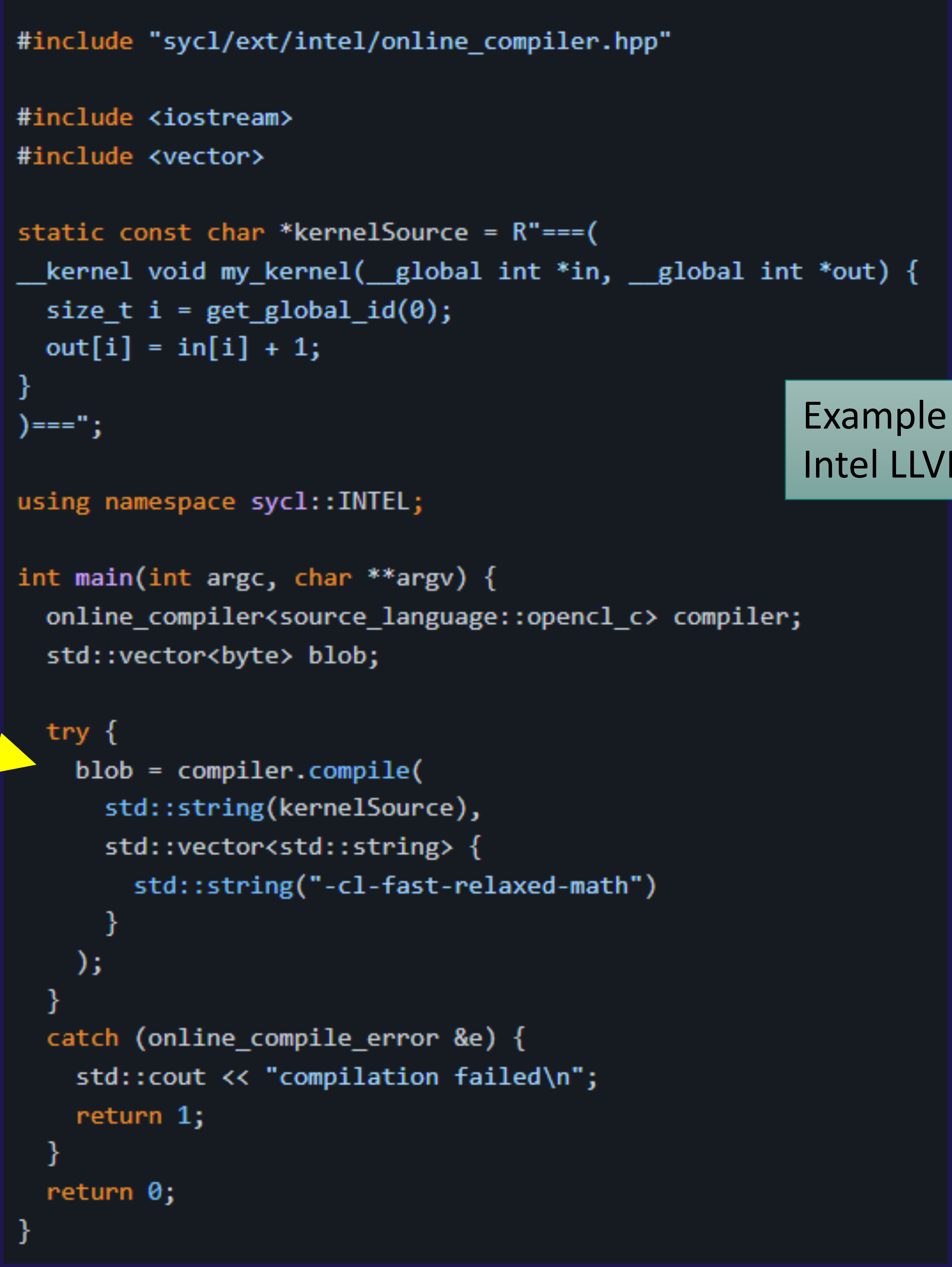
  https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/experimental/sycl_ext_oneapi_kernel_compiler.asciidoc

Intel Online Compiler SYCL Extension

- Experimental extension
- Calls ocloc library API (libigc)
- Compile OpenCL string to “binary blob”
- How to use “binary blob”?

Alternatives are not ideal

- OpenCL plugin only + interop
- Use Level Zero + libigc directly

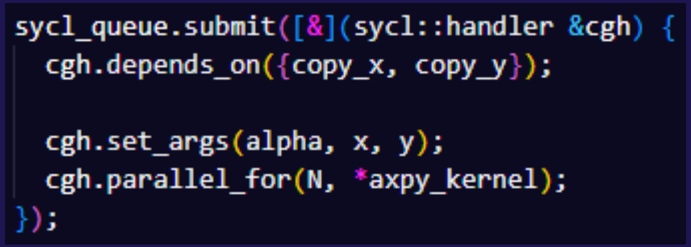


```
#include "sycl/ext/intel/online_compiler.hpp"

#include <iostream>
#include <vector>

static const char *kernelSource = R"===(
__kernel void my_kernel(__global int *in, __global int *out) {
  size_t i = get_global_id(0);
  out[i] = in[i] + 1;
}
)===";

using namespace sycl::INTEL;

int main(int argc, char **argv) {
  online_compiler<source_language::opencl_c> compiler;
  std::vector<byte> blob;

  try {
    blob = compiler.compile(
      std::string(kernelSource),
      std::vector<std::string> {
        std::string("-cl-fast-relaxed-math")
      }
    );
  }
  catch (online_compile_error &e) {
    std::cout << "compilation failed\n";
    return 1;
  }
  return 0;
}
```

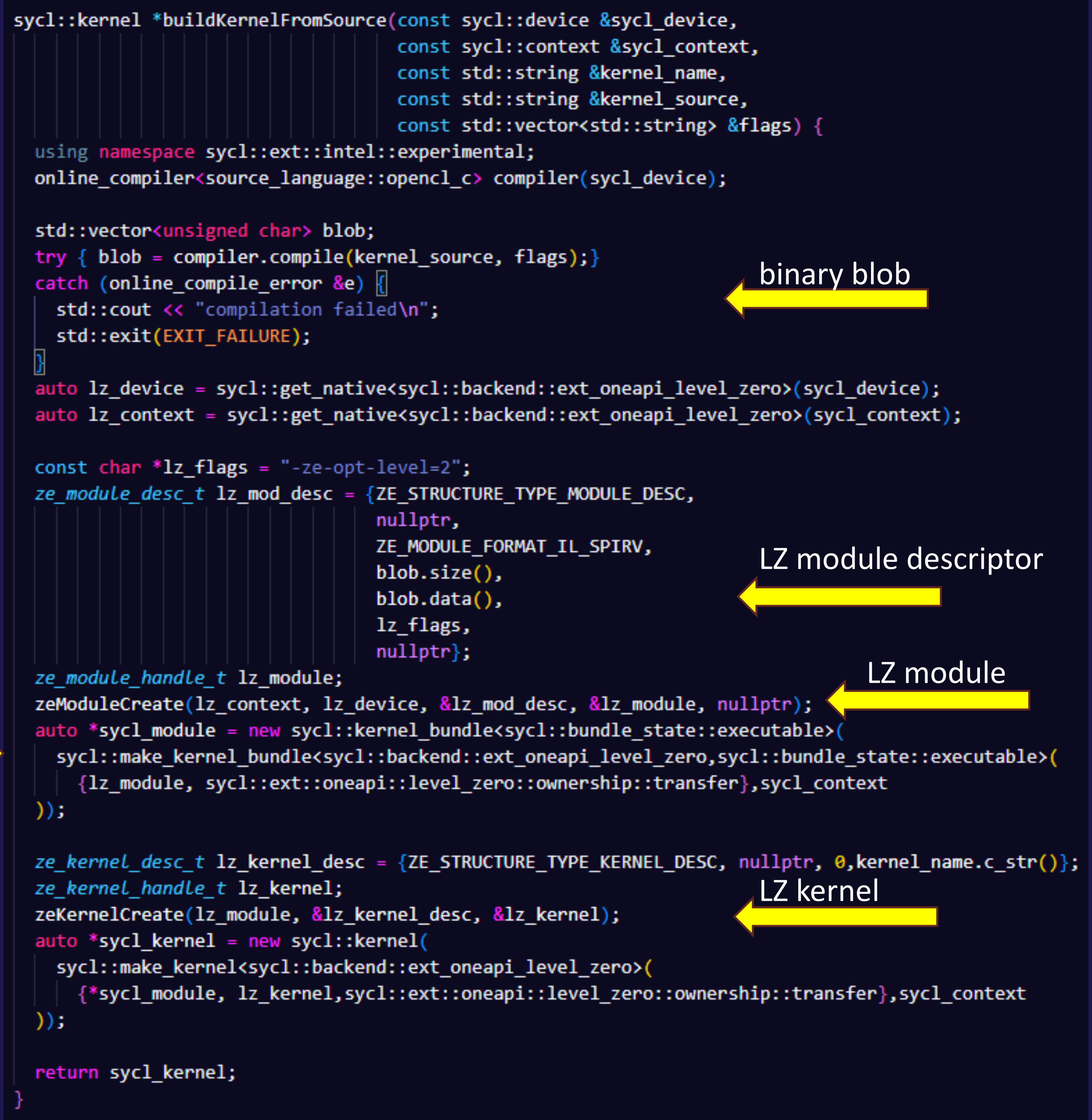
Example from Intel LLVM docs

Example

Online Compiler + Level Zero



```
sycl_queue.submit([&](sycl::handler &cgh) {
  cgh.depends_on({copy_x, copy_y});

  cgh.set_args(alpha, x, y);
  cgh.parallel_for(N, *axpy_kernel);
});
```



```
sycl::kernel *buildKernelFromSource(const sycl::device &sycl_device,
                                    const sycl::context &sycl_context,
                                    const std::string &kernel_name,
                                    const std::string &kernel_source,
                                    const std::vector<std::string> &flags) {
  using namespace sycl::ext::intel::experimental;
  online_compiler<source_language::opencl_c> compiler(sycl_device);

  std::vector<unsigned char> blob;
  try { blob = compiler.compile(kernel_source, flags);}
  catch (online_compile_error &e) {
    std::cout << "compilation failed\n";
    std::exit(EXIT_FAILURE);
  }
  auto lz_device = sycl::get_native<sycl::backend::ext_oneapi_level_zero>(sycl_device);
  auto lz_context = sycl::get_native<sycl::backend::ext_oneapi_level_zero>(sycl_context);

  const char *lz_flags = "-ze-opt-level=2";
  ze_module_desc_t lz_mod_desc = {ZE_STRUCTURE_TYPE_MODULE_DESC,
                                  nullptr,
                                  ZE_MODULE_FORMAT_IL_SPIRV,
                                  blob.size(),
                                  blob.data(),
                                  lz_flags,
                                  nullptr};
  ze_module_handle_t lz_module;
  zeModuleCreate(lz_context, lz_device, &lz_mod_desc, &lz_module, nullptr);
  auto *sycl_module = new sycl::kernel_bundle<sycl::bundle_state::executable>(
    sycl::make_kernel_bundle<sycl::backend::ext_oneapi_level_zero,sycl::bundle_state::executable>(
      {lz_module, sycl::ext::oneapi::level_zero::ownership::transfer},sycl_context
  ));

  ze_kernel_desc_t lz_kernel_desc = {ZE_STRUCTURE_TYPE_KERNEL_DESC, nullptr, 0,kernel_name.c_str()};
  ze_kernel_handle_t lz_kernel;
  zeKernelCreate(lz_module, &lz_kernel_desc, &lz_kernel);
  auto *sycl_kernel = new sycl::kernel(
    sycl::make_kernel<sycl::backend::ext_oneapi_level_zero>(
      {*sycl_module, lz_kernel,sycl::ext::oneapi::level_zero::ownership::transfer},sycl_context
  ));

  return sycl_kernel;
}
```

binary blob

LZ module descriptor

LZ module

SYCL kernel bundle

LZ kernel

SYCL kernel

libCEED Implementation

```cpp
using SyclModule_t = sycl::kernel_bundle<sycl::bundle_state::executable>;
```

```cpp
int CeedJitGetKernel_Sycl(Ceed ceed, const SyclModule_t *sycl_module, const std::string &kernel_name, sycl::kernel **sycl_kernel) {
  Ceed_Sycl *data;
  CeedCallBackend(CeedGetData(ceed, &data));

  // sycl::get_native returns std::vector<ze_module_handle_t> for lz backend
  // https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/supported/sycl_ext_oneapi_backend_level_zero.md
  ze_module_handle_t lz_module = sycl::get_native<sycl::backend::ext_oneapi_level_zero>(*sycl_module).front();

  ze_kernel_desc_t   lz_kernel_desc = {ZE_STRUCTURE_TYPE_KERNEL_DESC, nullptr, 0, kernel_name.c_str()};
  ze_kernel_handle_t lz_kernel;
  zeKernelCreate(lz_module, &lz_kernel_desc, &lz_kernel);

  *sycl_kernel = new sycl::kernel(sycl::make_kernel<sycl::backend::ext_oneapi_level_zero>(
      {*sycl_module, lz_kernel, sycl::ext::oneapi::level_zero::ownership::transfer}, data->sycl_context));

  return CEED_ERROR_SUCCESS;
}
```

SYCL queue synchronization for external libraries

- Streams in CUDA and HIP are in-order.
- A global default stream is available - simplifying the coordination with external libraries.
- Initial implementation of SYCL backends used in-order queues.
- CEED-PHASTA uses PETSc
  - For Intel GPUs, PETSc uses Kokkos-SYCL backend
  - Kokkos-SYCL uses out-of-order SYCL queues
- For synchronization with PETSc, libCEED SYCL inputs SYCL queue from PETSc.
- To use out-of-order queues with the libCEED SYCL implementation, the Intel SYCL extension for enqueueing (asynchronous, device-side) enqueue barriers were used.
- Future development will look to use SYCL events to explicitly express data dependencies within libCEED.

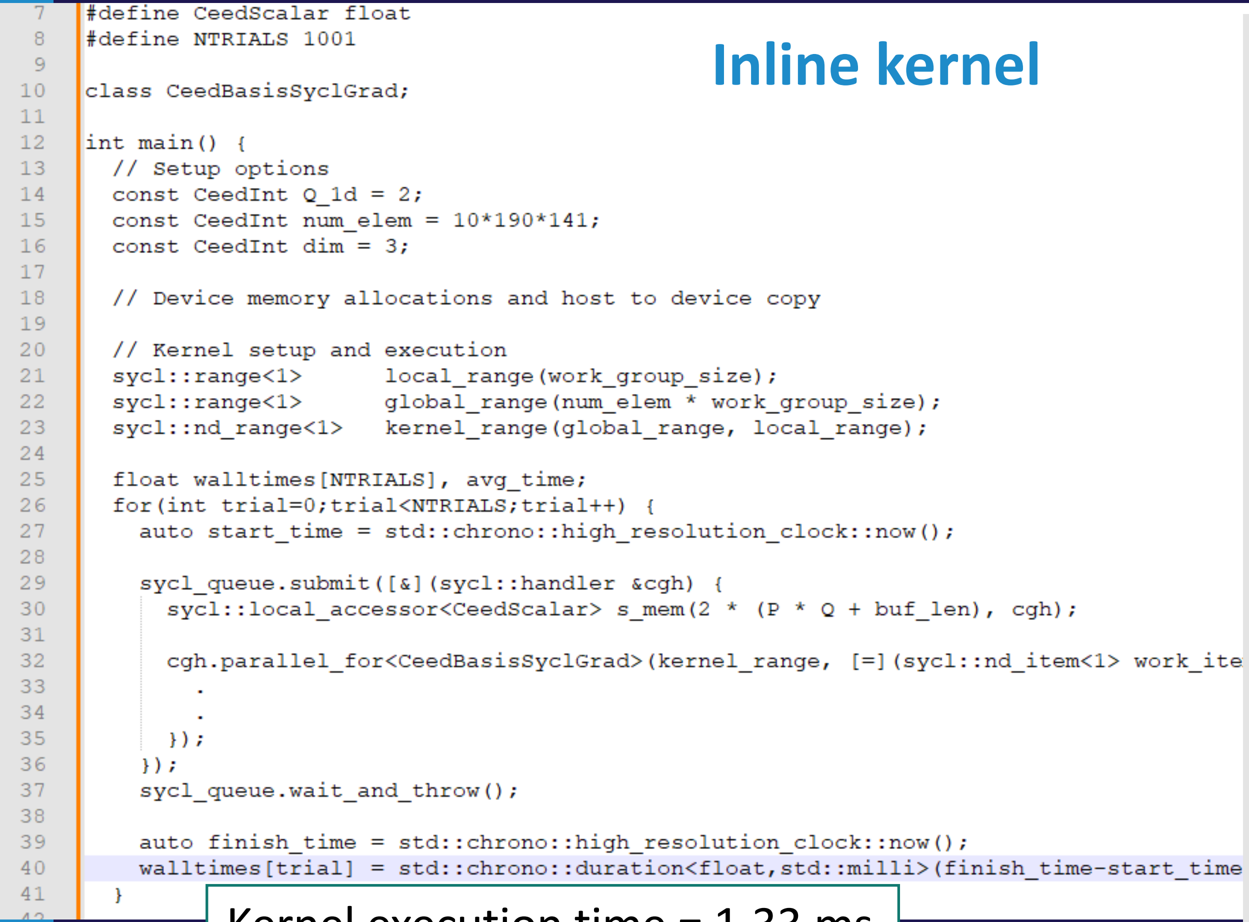
Performance Baseline

- sycl-fluids examples using Blasius test case.
  - Overall time per iteration on A100 was 3.7 sec
  - Overall time per iteration on PVC was 70 seconds
- 3 hotspots identified for both A100 and PVC.
  - CeedBasisSyclGrad – SYCL kernel
  - CeedBasisSyclInterp – SYCL kernel
  - IJacobian (OpenCL kernel which is online compiled)

Kernel	**PVC (1 Tile)**	**A100 (CUDA)**
IJacobian	29 ms	0.92 ms
Interp	10.7 ms	0.11 ms
Grad	197 ms	1.3 ms

BASIS GRAD KERNEL

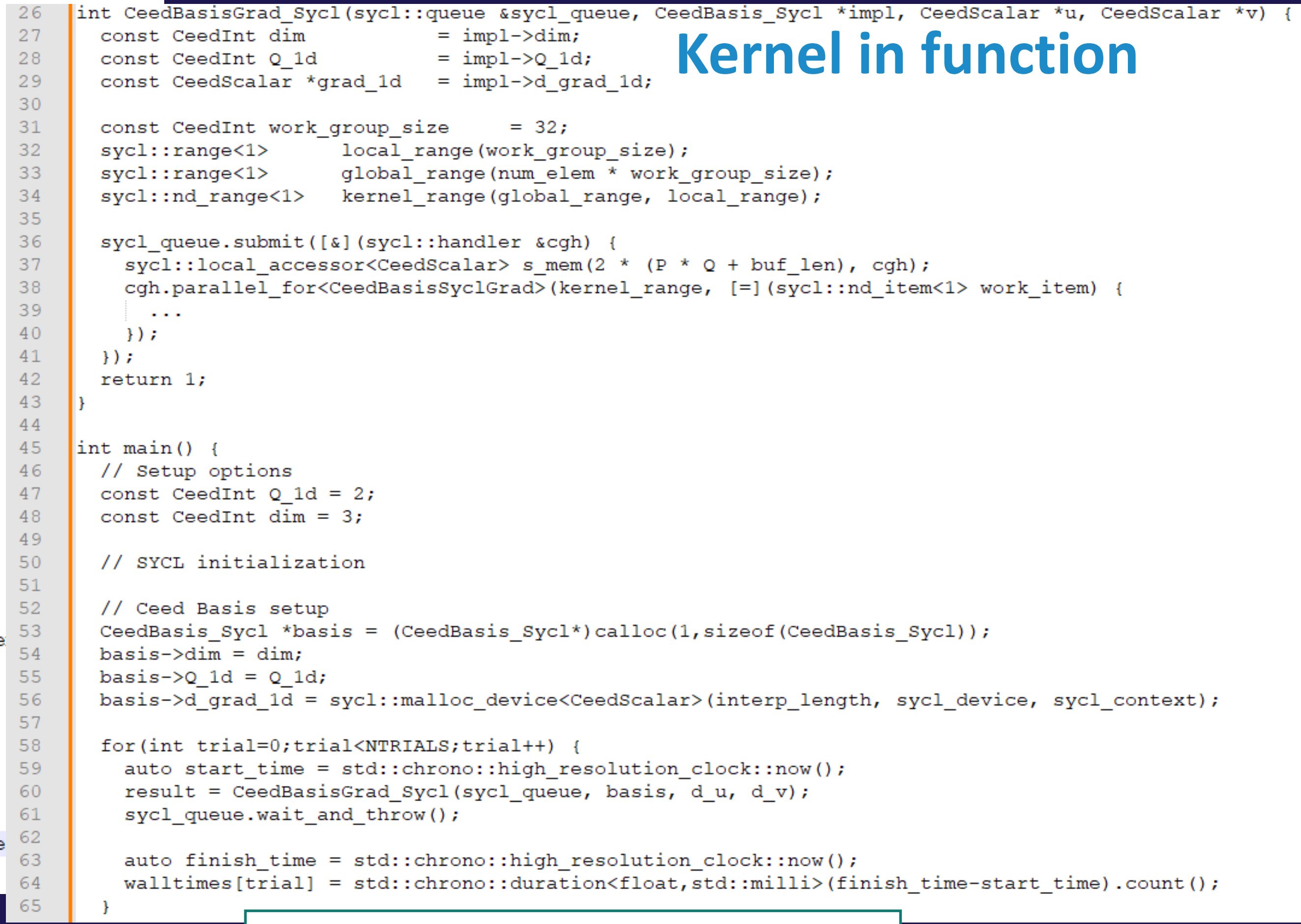
Using microbenchmarks of the kernel



Inline kernel

```
#define CeedScalar float
#define NTRIALS 1001

class CeedBasisSyclGrad;

int main() {
  // Setup options
  const CeedInt Q_1d = 2;
  const CeedInt num_elem = 10*190*141;
  const CeedInt dim = 3;

  // Device memory allocations and host to device copy

  // Kernel setup and execution
  sycl::range<1>      local_range(work_group_size);
  sycl::range<1>      global_range(num_elem * work_group_size);
  sycl::nd_range<1>   kernel_range(global_range, local_range);

  float walltimes[NTRIALS], avg_time;
  for(int trial=0;trial<NTRIALS;trial++) {
    auto start_time = std::chrono::high_resolution_clock::now();

    sycl_queue.submit([&](sycl::handler &cgh) {
      sycl::local_accessor<CeedScalar> s_mem(2 * (P * Q + buf_len), cgh);

      cgh.parallel_for<CeedBasisSyclGrad>(kernel_range, [=](sycl::nd_item<1> work_ite
        .
        .
      });
    });
    sycl_queue.wait_and_throw();

    auto finish_time = std::chrono::high_resolution_clock::now();
    walltimes[trial] = std::chrono::duration<float,std::milli>(finish_time-start_time
  }
```

Kernel execution time = 1.33 ms

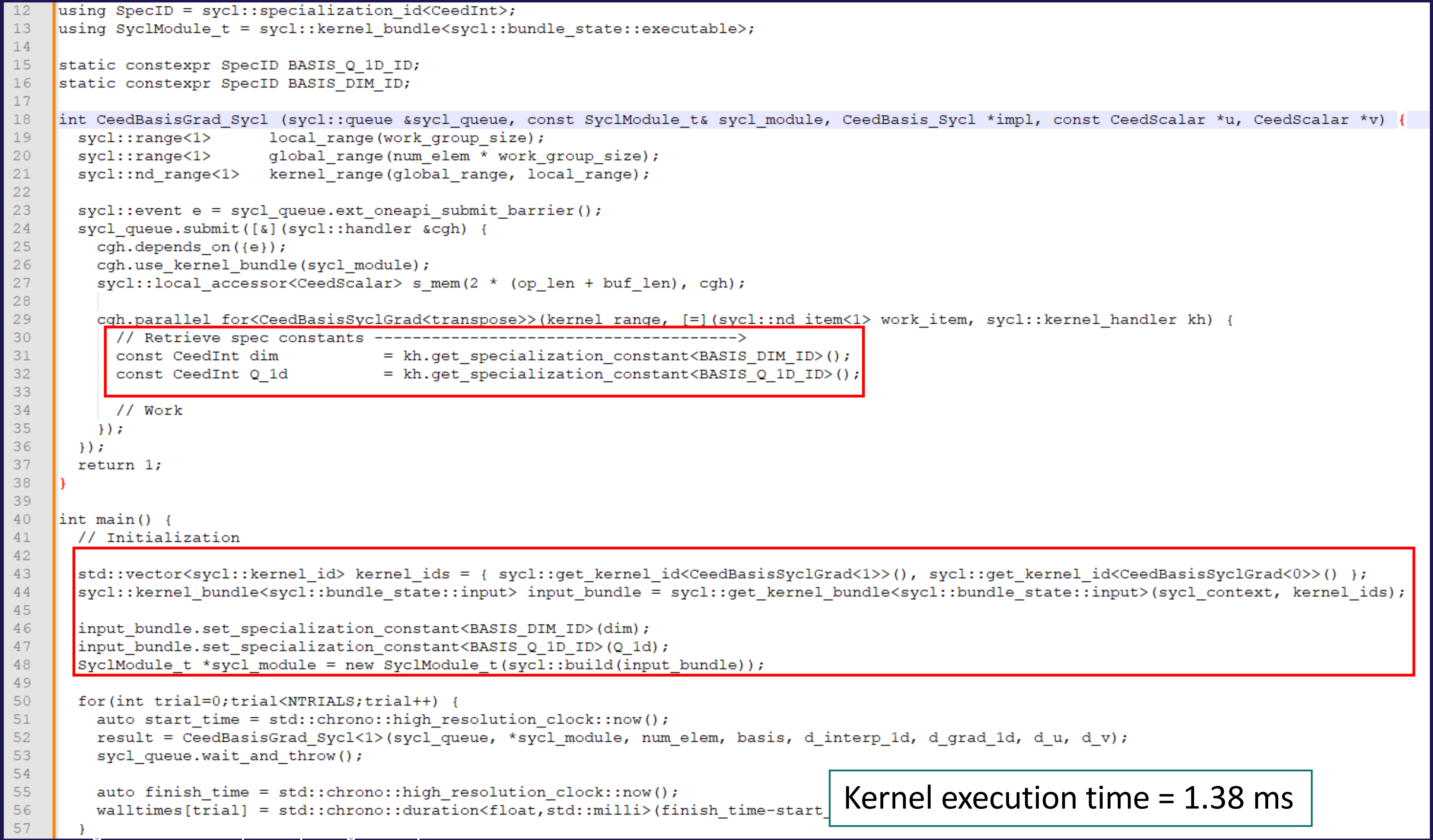


Kernel in function

```
int CeedBasisGrad_Sycl(sycl::queue &sycl_queue, CeedBasis_Sycl *impl, CeedScalar *u, CeedScalar *v) {
  const CeedInt dim            = impl->dim;
  const CeedInt Q_1d           = impl->Q_1d;
  const CeedScalar *grad_1d    = impl->d_grad_1d;

  const CeedInt work_group_size     = 32;
  sycl::range<1>      local_range(work_group_size);
  sycl::range<1>      global_range(num_elem * work_group_size);
  sycl::nd_range<1>   kernel_range(global_range, local_range);

  sycl_queue.submit([&](sycl::handler &cgh) {
    sycl::local_accessor<CeedScalar> s_mem(2 * (P * Q + buf_len), cgh);
    cgh.parallel_for<CeedBasisSyclGrad>(kernel_range, [=](sycl::nd_item<1> work_item) {
      ...
    });
  });
  return 1;
}

int main() {
  // Setup options
  const CeedInt Q_1d = 2;
  const CeedInt dim = 3;

  // SYCL initialization

  // Ceed Basis setup
  CeedBasis_Sycl *basis = (CeedBasis_Sycl*)calloc(1,sizeof(CeedBasis_Sycl));
  basis->dim = dim;
  basis->Q_1d = Q_1d;
  basis->d_grad_1d = sycl::malloc_device<CeedScalar>(interp_length, sycl_device, sycl_context);

  for(int trial=0;trial<NTRIALS;trial++) {
    auto start_time = std::chrono::high_resolution_clock::now();
    result = CeedBasisGrad_Sycl(sycl_queue, basis, d_u, d_v);
    sycl_queue.wait_and_throw();

    auto finish_time = std::chrono::high_resolution_clock::now();
    walltimes[trial] = std::chrono::duration<float,std::milli>(finish_time-start_time).count();
  }
```

Kernel execution time = 5.8 ms

Use of SYCL specialization constants



```cpp
using SpecID = sycl::specialization_id<CeedInt>;
using SyclModule_t = sycl::kernel_bundle<sycl::bundle_state::executable>;

static constexpr SpecID BASIS_Q_1D_ID;
static constexpr SpecID BASIS_DIM_ID;

int CeedBasisGrad_Sycl (sycl::queue &sycl_queue, const SyclModule_t& sycl_module, CeedBasis_Sycl *impl, const CeedScalar *u, CeedScalar *v) {
  sycl::range<1>       local_range(work_group_size);
  sycl::range<1>       global_range(num_elem * work_group_size);
  sycl::nd_range<1>    kernel_range(global_range, local_range);

  sycl::event e = sycl_queue.ext_oneapi_submit_barrier();
  sycl_queue.submit([&](sycl::handler &cgh) {
    cgh.depends_on({e});
    cgh.use_kernel_bundle(sycl_module);
    sycl::local_accessor<CeedScalar> s_mem(2 * (op_len + buf_len), cgh);

    cgh.parallel_for<CeedBasisSyclGrad<transpose>>(kernel_range, [=](sycl::nd_item<1> work_item, sycl::kernel_handler kh) {
      // Retrieve spec constants ---------------------------------------->
      const CeedInt dim           = kh.get_specialization_constant<BASIS_DIM_ID>();
      const CeedInt Q_1d          = kh.get_specialization_constant<BASIS_Q_1D_ID>();

      // Work
    });
  });
  return 1;
}

int main() {
  // Initialization

  std::vector<sycl::kernel_id> kernel_ids = { sycl::get_kernel_id<CeedBasisSyclGrad<1>>(), sycl::get_kernel_id<CeedBasisSyclGrad<0>>() };
  sycl::kernel_bundle<sycl::bundle_state::input> input_bundle = sycl::get_kernel_bundle<sycl::bundle_state::input>(sycl_context, kernel_ids);

  input_bundle.set_specialization_constant<BASIS_DIM_ID>(dim);
  input_bundle.set_specialization_constant<BASIS_Q_1D_ID>(Q_1d);
  SyclModule_t *sycl_module = new SyclModule_t(sycl::build(input_bundle));

  for(int trial=0;trial<NTRIALS;trial++) {
    auto start_time = std::chrono::high_resolution_clock::now();
    result = CeedBasisGrad_Sycl<1>(sycl_queue, *sycl_module, num_elem, basis, d_interp_1d, d_grad_1d, d_u, d_v);
    sycl_queue.wait_and_throw();

    auto finish_time = std::chrono::high_resolution_clock::now();
    walltimes[trial] = std::chrono::duration<float,std::milli>(finish_time-start_
  }
```

Kernel execution time = 1.38 ms

Further optimizations on CeedBasisGrad

- Adjust WG size from 1024 to 32.
- Replace workgroup barrier with nd_item barrier

```
sycl::group_barrier(work_group)  → work_item.barrier(sycl::access::fence_space::local_space)
```

- Remove IGC runtime memory checks by using specialization constant.

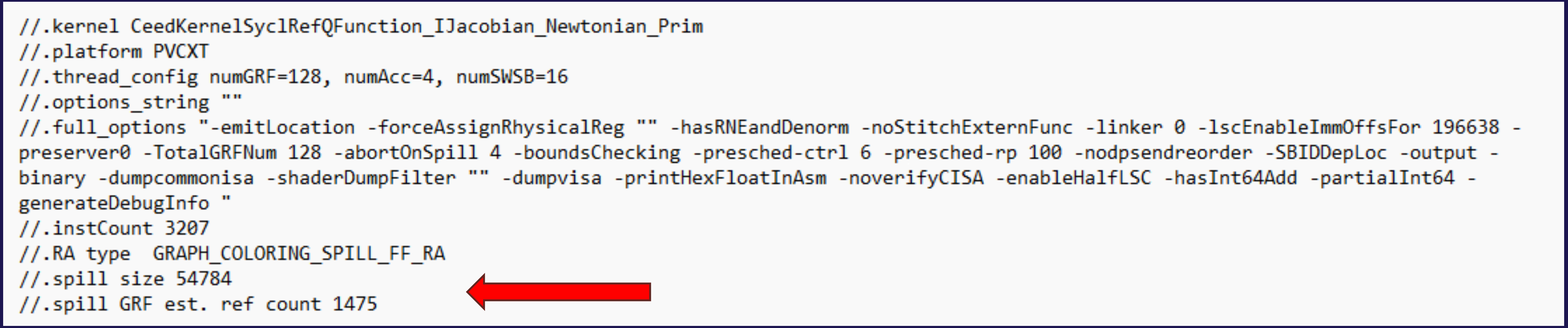
```
warning: from dir:/home/vmadananth/PHASTA_aesp_CNDA/CEED-PHASTA/libCEED/backends/sycl-ref from file:ceed-sycl-ref-basis.sycl.cpp line:100 :Adding additional control flow due to presence of generic address space operations
warning: from dir:/home/vmadananth/PHASTA_aesp_CNDA/CEED-PHASTA/libCEED/backends/sycl-ref from file:ceed-sycl-ref-basis.sycl.cpp line:185 :Adding additional control flow due to presence of generic address space operations
warning: from dir:/home/vmadananth/PHASTA_aesp_CNDA/CEED-PHASTA/libCEED/backends/sycl-ref from file:ceed-sycl-ref-basis.sycl.cpp line:187 :Adding additional control flow due to presence of generic address space operations
warning: from dir:/home/vmadananth/PHASTA_aesp_CNDA/CEED-PHASTA/libCEED/backends/sycl-ref from file:ceed-sycl-ref-basis.sycl.cpp line:0 :Adding additional control flow due to presence of generic address space operations
```

Kernel	**PVC (1 Tile)**	**A100**
IJacobian	29ms	0.9ms
Interp	1.75ms	0.11ms
Grad	4.8ms	1.3ms

Optimizations performed on IJacobian

- IJacobian kernel was using default range. The compiler was setting the workgroup size to 32.
  - Changing to default nd_range and setting WG size to 384 (CUDA) improved performance slightly.
- Reducing register spills
  - Building with AOT did not show any warnings on spills as this was OpenCL kernel.
  - Inspected assembly by setting the following env variables –
    - IGC_DumpToCurrentDir=1, IGC_DumpToCurrentDir=1
    - For more IGC env variables - https://github.com/intel/intel-graphics-compiler/blob/master/documentation/configuration_flags.md
    - Generates asm for all kernels. To search for kernel names –

```
for f in ./*.asm; do echo "------------"; echo $f; cat $f | grep "\/\/.kernel"; done
```



```
//.kernel CeedKernelSyclRefQFunction_IJacobian_Newtonian_Prim
//.platform PVCXT
//.thread_config numGRF=128, numAcc=4, numSWSB=16
//.options_string ""
//.full_options "-emitLocation -forceAssignRhysicalReg "" -hasRNEandDenorm -noStitchExternFunc -linker 0 -lscEnableImmOffsFor 196638 -
preserver0 -TotalGRFNum 128 -abortOnSpill 4 -boundsChecking -presched-ctrl 6 -presched-rp 100 -nodpsendreorder -SBIDDepLoc -output -
binary -dumpcommonisa -shaderDumpFilter "" -dumpvisa -printHexFloatInAsm -noverifyCISA -enableHalfLSC -hasInt64Add -partialInt64 -
generateDebugInfo "
//.instCount 3207
//.RA type  GRAPH_COLORING_SPILL_FF_RA
//.spill size 54784
//.spill GRF est. ref count 1475
```

Reducing spills

- By default , PVC has 128 64-byte registers allocated per thread.
- Registers spills can be expensive. Improve register usage per thread by
  - Increasing the registers available per thread to 256
  - Running in lower SIMD width
- Passing large register file with online compilation.

```
ze_module_desc_t lz_mod_desc = {ZE_STRUCTURE_TYPE_MODULE_DESC,
                                  nullptr,
                                  ZE_MODULE_FORMAT_IL_SPIRV,
                                  il_binary.size(),
                                  il_binary.data(),
                                  "-ze-opt-large-register-file",    // flags
                                  nullptr};
```

- Enforcing SIMD16 with OpenCL kernel

  __attribute__(intel_reqd_sub_group_size(16))

  https://registry.khronos.org/OpenCL/extensions/intel/cl_intel_required_subgroup_size.html

Where does performance stand currently

Kernel	A100	PVC (before optimization)	PVC (after optimizations)
IJacobian	0.9 ms	29 ms	1.1 ms
Grad	1.3 ms	6.4 ms	4.5 ms
Total time per timestep	3.6 s	70 s	11 s

*This is still a work in progress, and not a representative of performance between two hardware

Key Takeaways

Intel online compiler SYCL extension

- Allows for NVRTC-like runtime compilation
- Currently restricted to use of OpenCL C/LevelZero kernels (interop)
- In future, likely to extend support for SYCL kernel code

Optimization Strategies

- Specialization constants - useful for optimizing code with parametric values. Runtime increases with number of spec. constants
- Use of appropriate workgroup sizes and barriers.
- For register-heavy kernels – large GRF + smaller SIMD width

Future Work

- Systematic profiling of online compiled code for further insights on performance bottlenecks.
- Use of the updated SYCL kernel compiler with SYCL user functions.

Acknowledgements

Team members:	Jed Brown (UC Boulder)
	Natalie Beams (UT Knoxville)
	Kenneth Jansen (UC Boulder)

*This research used resources of the Argonne Leadership Computing Facility, a U.S. Department of Energy (DOE) Office of Science user facility at Argonne National Laboratory and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

***This work was done on a pre-production supercomputer with early versions of the Aurora software development kit.**