NektarIR

Towards code-generation of highly efficient finite element kernels for computational fluid dynamics using the MLIR compiler infrastructure

Edward Erasmie-Jones[(1)], Giacomo Castiglioni[(2)], David Moxey[(1)]

(1) King's College London, United Kingdom. (2) CSCS, Switzerland

Background

The **spectral/hp element method** is an example of a **high-order finite element method,** used for solving the weak form of **partial differential equations (PDEs),** in areas such as fluid dynamics, aeronautics and renewable energy. The method involves partitioning the solution domain into a **mesh of elemental regions** and approximating the local (elemental) solution to the weak PDE using a **polynomial expansion,** given by a series of known **basis functions** and unknown **expansion coefficients** that need to be determined [1]. The basis functions are often given by **high-order (or degree) polynomials,** giving the method a **high arithmetic intensity,** defined as FLOPs per byte of memory, that is ideal for numerical schemes on modern high-performance hardware [2]. The local approximation, and therefore the basis functions, are only evaluated at a particular set of points, known as **quadrature points**. These are chosen according to a **Gaussian quadrature rule** to facilitate numerical integration. The discretization of the PDE within each element yields an **elemental operator** and reduces the problem to **solving a system of linear equations for the unknown local expansion coefficients**. A set of **"building block" finite element operators** that act on alike elements are used to construct the elemental operator, and these are common across a range of PDEs. An example of a building block operator is the **backward transformation**, which constructs the approximation from the coefficients and basis functions (discussed further below).

Having performant implementations of the building block finite element operators available for a range of hardware architectures is a crucial part of the development of a hardware extensible PDE solver using the spectral/hp element method. In this poster, we present the initial stages of the development of **NektarIR**, an MLIR dialect for the generation of high-performance and hardware extensible kernels for finite element operators.

Goals and upcoming work

- **Abstraction of the method:**
  Develop an MLIR dialect representing a high-level abstraction of various finite element operators in the spectral/hp element method.
- **JIT compilation and automatic optimization:**
  Facilitate the just-in-time compilation of optimized kernels for the building block finite element operators for heterogeneous hardware architectures using LLVM.
- **CPU and GPU:**
  We are close to performance testing our operators for CPUs, both in a scalar case and for AVX2/AVX512 instruction sets. Exploring how to implement optimizations for these operators using MLIR is an immediate concern. Extending support to GPU is also important next step for the project.

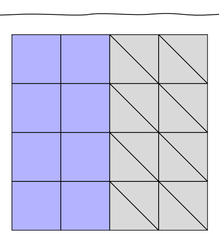
Nektar++

- Nektar++ is an open-source framework for the development of high-fidelity PDE solvers using the spectral/hp element method, with support for both CPU and GPU hardware architectures [2,3].
- As a well-established framework, it is an ideal reference point for testing and verifying the NektarIR implementations.
- One of the goals of NektarIR is to complement the existing support for heterogenous hardware in Nektar++ and to provide a JIT compiled option for the finite element operator kernels that are already available.

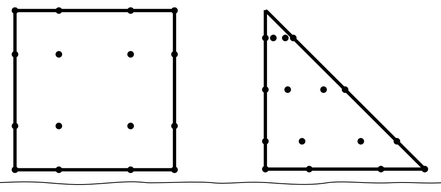
The NektarIR dialect

- **Types:**
  We have a custom type, the block type, that is an abstraction of a single element or collection of alike elements in a mesh, encoding its basis type or quadrature rule and geometric shape through attribute parameters. This is a tensor-like type.
- **Operations:**
  Our operations, for instance for the backward transformation, act on block types to perform the finite element operation on alike elements in the mesh. An IR example is given below. For inserting and extracting from a block, we use a destination-passing style and aim to mirror the tensor.insert_slice and tensor.extract_slice operations.
- **Passes:**
  Our entry-point to upstream MLIR is into the affine dialect. We lower our finite element operations to affine loop nests that loop over alike elements and try to mirror the bufferization dialect for conversion between the block type and memrefs.

Elements from a mesh with the same shape, quadrature rule and number of quadrature points, basis type and polynomial order are described by the !nir.block type



A simple mesh of quadrilateral and triangular elements.



Standard quadrilateral and triangular elements with quadrature points, chosen according to a Gaussian quadrature rule

An example of our IR for the backward transformation on a triangle

```
!coeffBlockType = !nir.block<Fields: [u], SEShape : tri, Deformed: false, Basis:(modified, modified), Size:1x1x7x7xf64, Layout: (d0,d1, d2) -> (d0,d1,d2)>
!physBlockType = !nir.block<Fields: [u], SEShape : tri, Deformed: false, Quadrature: (gll, gl), Size:1x1x20x20xf64, Layout: (d0,d1,d2) -> (d0, d1 ,d2)>
#map = affine_map<(d0,d1) -> (d0+20*d1)>
module{
   func.func @bwdtri(%uhat: memref<1x1x28xf64>, %b0: memref<20x7xf64, #map>, %b1:memref<20x28xf64, #map>, %u:memref<1x1x400xf64>)
    attributes {llvm.emit_c_interface}
    {
       %b0t = bufferization.to_tensor %b0 restrict : memref<20x7xf64, #map>
       %b1t = bufferization.to_tensor %b1 restrict : memref<20x28xf64, #map>

      %coeffBlock = nir.block_from_memref [
         Data: %uhat : memref<1x1x28xf64>
         Fields : [u]
         Shape: tri
         Basis: (modified, modified)
         BlockSize: [7,7]
         Deformed: false] -> !coeffBlockType

       %out = nir.elemental_bwd [
         Block: %coeffBlock : !coeffBlockType
         Bases: %b0t, %b1t: tensor<20x7xf64>, tensor<20x28xf64>] -> !physBlockType

      nir.materialize_in_destination %out in restrict writable %u: (!physBlockType, memref<1x1x400xf64>) -> ()
      return
   }
}
```

This operation creates a nir block type from the memref "%uhat" which contains the expansion coefficients. The Shape and Basis attribute parameters encode some sparsity. This operation folds away after our first pass into standard MLIR, leaving the memref.

The nir.elemental_bwd constructs the elemental approximation from the expansion coefficients and basis tensors evaluated at the quadrature points. It is a kind of polynomial interpolation. We lower this to affine, with affine loops, loads and stores. For vectorization, we also use vector.transfer_read/write. We use arith for addition and multiplication.

The approximation evaluated at the quadrature points

Basis functions

$$u(\xi_{1i}, \xi_{2j}) = \sum_{p=0}^{P} \sum_{q=0}^{Q-p} \hat{u}_{pq} \tilde{\psi}_p^a(\xi_{1i}) \tilde{\psi}_{pq}^b(\xi_{2j}), \quad \forall i, j$$

Expansion coefficients

Acknowledgements

EEJ would like to thank the LLVM foundation for their generous support through the LLVM student travel grant.
DM would like to acknowledge the Royal Academy of Engineering under their research chair scheme.

References

[1] Karniadakis, George & Sherwin, Spencer. (1999). Spectral/hp Element Methods for CFD.
[2] Moxey, D., Amici, R., & Kirby, M. (2020). Efficient matrix-free high-order finite element evaluation for simplicial elements. *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, *42*(3), C97-C123. https://doi.org/10.1137/19M1246523
[3] C.D. Cantwell, D. Moxey, A. Comerford, A. Bolis, G. Rocco, G. Mengaldo, D. De Grazia, S. Yakovlev, J.-E. Lombard, D. Ekelschot, B. Jordi, H. Xu, Y. Mohamied, C. Eskilsson, B. Nelson, P. Vos, C. Biotto, R.M. Kirby, S.J. Sherwin, Nektar++: An open-source spectral/hp element framework, Computer Physics Communications, Volume 192, 2015, Pages 205-219, ISSN 0010 4655, https://doi.org/10.1016/j.cpc.2015.02.008.