High performance mesh abstractions

Connor Ward

October 17, 2022

1 Introduction

In scientific computing, the composition of appropriate (software) abstractions is essential for scientists to write portable and performant simulations in a productive way (the three P's). Having suitable abstraction layers allows for a separation of concerns whereby numericists can reason about their problem from a purely mathematical point-of-view, and computer scientists can focus on low-level performance. Each discipline is presented with a particular interface from which the problems of interest can be expressed in the clearest possible way. This makes the application of particular optimisations straightforward as problem-specific information can be explicitly enumerated, rather than requiring inference to determine.

When it comes to writing software there are effectively three choices of approach. For many problems, generic library interfaces introduce too much overhead to be viable options for writing programs. Similarly, hand-written codes, though extremely fast, require a substantial effort to maintain and extend and the codebase can be very large. Code generation is an appealing solution to these problems. Given an appropriate abstraction, high-performance code can be automatically generated, compiled and run. This offers an advantage over library interfaces because problem-specific information can be exploited to generate faster code (e.g. commonly used operations can be memoized for fast lookups), and the task of actually writing the code is offloaded to a compiler rather than being hand-written. With a code generation framework, the key questions now become: What is an appropriate abstraction for capturing all of the behaviour I wish to model? What performance optimisations are nicely expressed at this layer of abstraction?

In this work, we present `pyop3`, a library for the fast execution of mesh-based computations. In accordance with the principles described above, `pyop3`deals mainly in 3 abstractions: Firstly, the user interface is motivated by the fact that many operations relating to the solution of partial differential equation (PDE)s can be expressed as the operation of some 'local' kernel over a set of entities in the mesh where only functions with non-zero support on this entity are considered in the calculation. A classic example of this sort of calculation occurs with finite element assembly where the cells of the mesh are the iteration set and the kernel uses degrees-of-freedom (DoFs) from the cell and enclosing edges and vertices. This first abstraction layer therefore presents an interface to the user where they may straightforwardly express the operation of local kernels within loops over mesh entities, specifying the requisite restrictions for the data. The second abstraction, intended as an internal representation for the developer, describes the data layout and the third is a polyhedral loop model that is the target for the code generation.

Having motivated the need for a mesh traversal abstraction, we now move on to motivating the need for a *high-performance* mesh traversal abstraction. In other words, are there reasonable use-cases where this traversal constitutes a substantial fraction of the overall program runtime?

To begin, the wall-clock time of a program, assuming good algorithmic/parallel design, is usually limited by some combination of the maximum throughput of the processor and the cost of moving data to and from said processor. Given that the principle role for `pyop3`is to marshall data for the kernel we focus on codes where the cost of moving data is the bottleneck. Any effort on minimising the number of floating point operations per second (FLOPs) would be wasted as such optimisations would only be useful inside the 'hot' loops of the kernel. Also, hardware developments have seen a general trend where computing power is increasing at a more rapid rate than memory access speed. As such, the bottleneck in an increasing number of codes is going to be the memory accesses.

In this work, we present `pyop3`…

The rest of this paper is laid out as follows: …

2 Background

In this chapter we review: existing software abstractions for mesh computations, common strategies for optimising performance, and mesh-specific optimisations.

2.1 Existing software abstractions

2.2 Stencil languages

Given the ubiquity of stencil operations in simulations, a number of libraries exist providing convenient interfaces for stencil applications.

Ebb, Simit and Liszt follow the approach of providing a domain-specific language for the expression of stencil problems... They all provide high performance execution on GPUs as well as CPUs.

OP2 is another approach [6, 5]. Rather than using a domain-specific language, OP2 provides a simple API to the user for specifying the problem. The key entities in the OP2 data model are: sets, data on sets, mappings between sets, and operations applied over these sets. Having provided these inputs, the OP2 compiler is then called and transforms to source code to a high performance implementation of the traversal for a specific architecture.

Another library for the application of stencil operations, specifically high-order matrix-free kernels for the finite element method (FEM), is libCEED.

PyOP2 is a domain-specific language for expressing computations over unstructured meshes. It is the direct precursor, and inspiration for, `pyop3`.

It distinguishes itself from OP2, a library depending on the same abstractions, by using run-time code generation instead of static analysis and transformation of the source code.

PyOP2's mesh abstraction is formed out of the following main components:

In PyOP2, data is defined on *sets* and these are related to one another using *mappings*. Importantly, this abstraction does not contain any concept of the underlying mesh and instead all of the required information is encoded in the maps.

Computations over the mesh are expressed as the execution of some local kernel over all entities of some iteration set via a construct called a parallel loop, or *parloop*. The kernel is written using Loopy, a library for expressing array-based computations in a platform-generic language [3]. This intermediate representation allows for interplay between the local kernels enabling optimisations such as inter-element vectorisation [10].

At present, PyOP2 only works on distributed memory, CPU-only systems (although some work has been done to permit execution on GPUs [4]). During the execution of a parloop, each rank works independently on some partition of the mesh. To avoid excessive communication between ranks, each rank has a narrow *halo* region that overlaps with neighbouring ranks that is executed redundantly. The halos are split into *owned*, *exec*, and *non-exec* regions to indicate the data's origin and the communication direction between the neighbouring processes.

2.2.1 Mesh representations

In software, a mesh is typically represented by a collection of sets of entities (e.g. cells or faces), coupled with adjacency relations between these sets. Possible abstractions capturing this behaviour include databases (ebb, moab) or hypergraphs (simit). In this work we focus on DMPlex, the unstructured mesh abstraction used in PETSc. In contrast with Ebb, Simit or Liszt, DMPlex is a more general purpose mesh abstraction and so has a more substantial feature set.z

In DMPlex, the mesh is represented as a *CW-complex*, an object from algebraic topology that describes some topological space. In such a complex, all topological entities (e.g. cells, vertices) are simply referred to as *points* and the connectivity of the mesh can be expressed as the edges of a directed acyclic graph (DAG) with the vertices being the points of the mesh. More specifically, the points and relations form a partially-ordered set (poset) such that the mesh can be visualised using a Hasse diagram (Figure 1).

It is important to note that DMPlex works for arbitrary dimension.

Stencil queries are natural to express at this level. For instance, the classical finite element request of "give me all of the DoFs that have local support" is

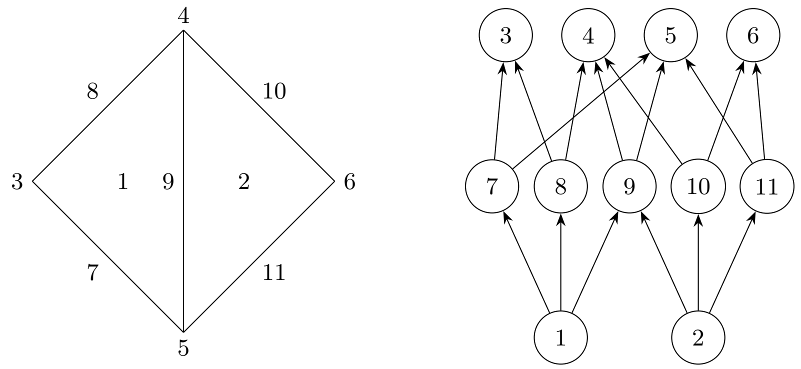
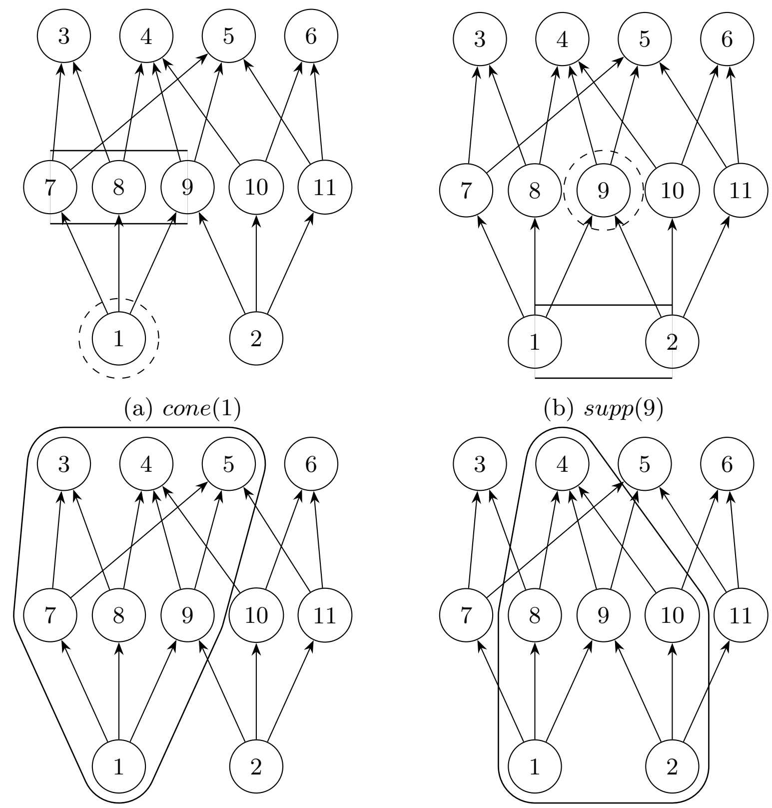


Figure 1: ...



(a) *cone*(1)

(b) *supp*(9)

(c) *cl*(1)

(d) *star*(4)

Figure 2: ...

simply expressed as the closure of a given cell. Another example useful for finite volume calculations: "what are my neighbouring cells?" is supp(cone(c)). One can also do clever patch things.

2.2.2 Orienting degrees-of-freedom

One significant challenge stencil applications face when applied to PDEs is on agreeing on a consistent orientation of the DoFs for shared entities.

To demonstrate the issue, consider the DoF arrangements shown in Figure 3. Figures 3a and 3c show the reference DoF arrangements for an arbitrary space on triangles, with scalar or vector DoFs on the edges respectively. Likewise, Figures 3b and 3d show the resulting DoF arrangements for the same spaces but with one of the edges flipped. In either flipped case, were the DoFs to be naively packed into a local temporary and passed to a local kernel, the results would be incorrect as the DoFs would be passed in in the wrong order.

For many cell types, the orientation problem can be avoided through renumbering the mesh such that adjacent entities agree on the orientations of any shared facets or edges. In particular this has been shown to work for simplices [7], as well as quadrilaterals and (some) hexes [1, 2]. ...Hexes are annoying because they cannot be oriented in parallel and some meshes (e.g. Mo(..)bius strip) are not orientable. But. We want them for tokamaks (cite).

However, for more complicated cell types with more complex symmetry groups (e.g. hexes or pyramids), the issue of orientation cannot be avoided by a simple renumbering and DoF transformations are needed to be able to collect the DoFs in a suitable reference order. For scalar-valued DoFs, one simply needs to permute the order in which DoFs are loaded into the local temporary. This can easily be done in advance and be encapsulated by, for example, a PyOP2 `Map`. This approach is insufficient for vector-valued DoFs though as components may still be pointing in the wrong direction. This is demonstrated in Figure 3d where one can see that simply permuting the DoFs on the flipped edge would not be enough. One also needs to multiply the values by -1 in order to get the vectors pointing in the right direction. The situation is further complicated in 3D where one could have two tangent vectors per DoF on each face, requiring the application of a 2×2 rotation matrix to reach consensus.

The general solution to orienting DoFs for stencil application is therefore as follows: First, one loads the (permuted) DoFs associated with a particular entity, along with a bitarray encoding the entity's orientation. Then, one can apply appropriate transformations to the loaded DoFs such that they can be correctly passed through to the local kernel. This is the approach used by Basix [9, 8], part of the FEniCSx finite element software suite []. However, to our knowledge, this is not performed by any existing stencil library.

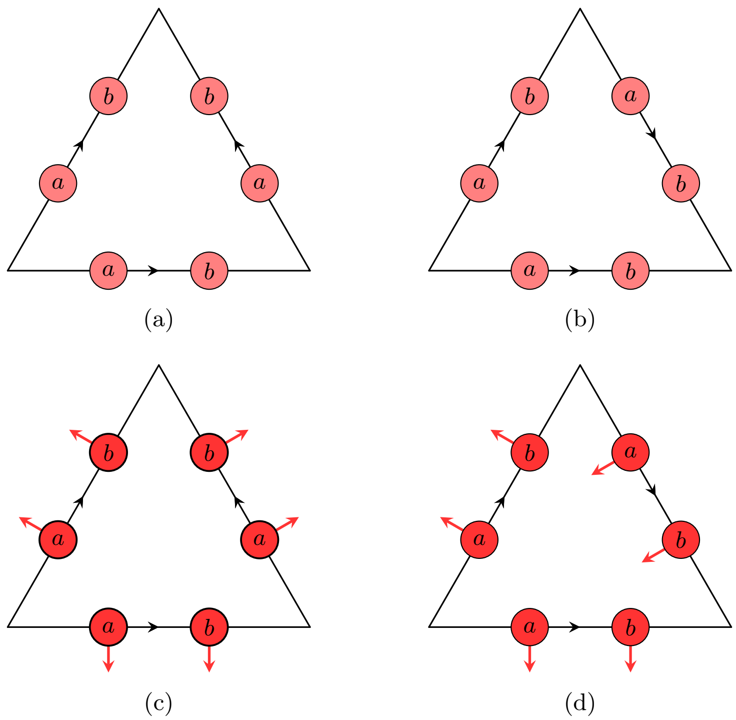
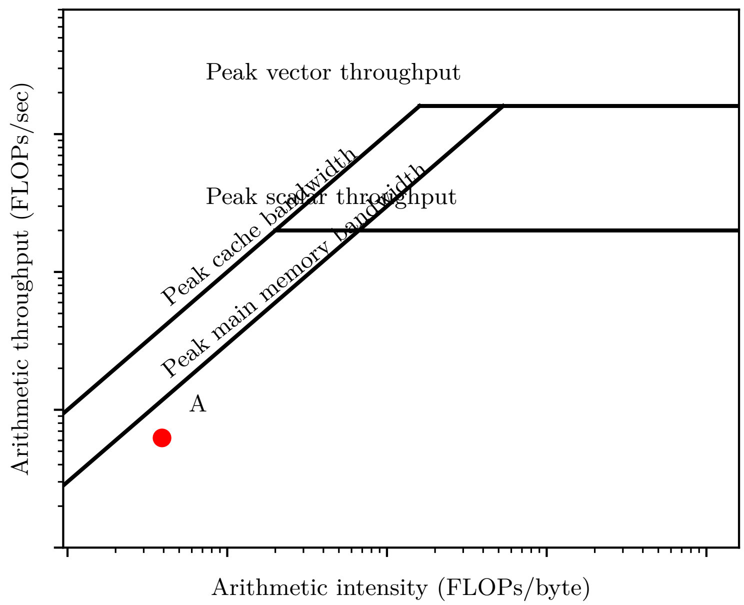


Figure 3

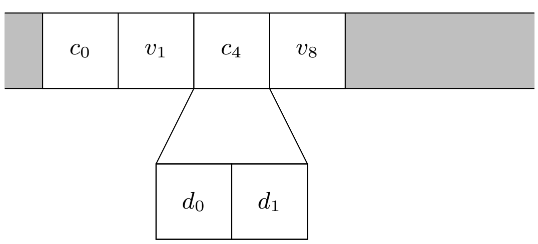


2.3 Methods for performance optimisation

2.3.1 Roofline model

2.4 Optimisations for mesh computations

3 Implementation

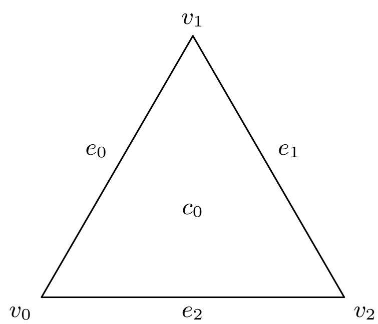


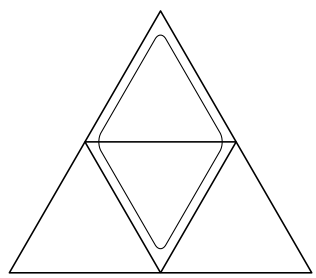
4 Future work

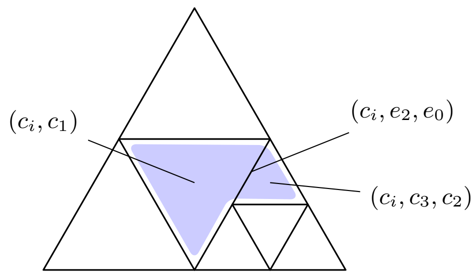
4.1 Direct addressing for partially structured meshes

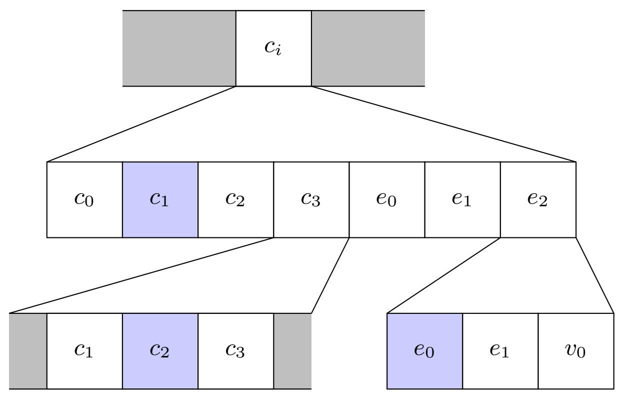
4.1.1 Extruded meshes

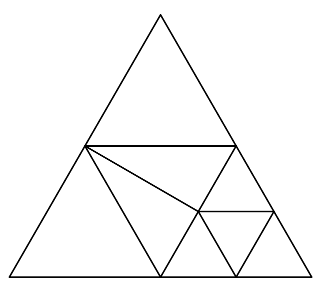
4.1.2 Regular mesh refinement

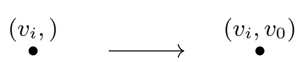


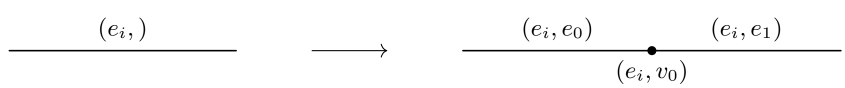


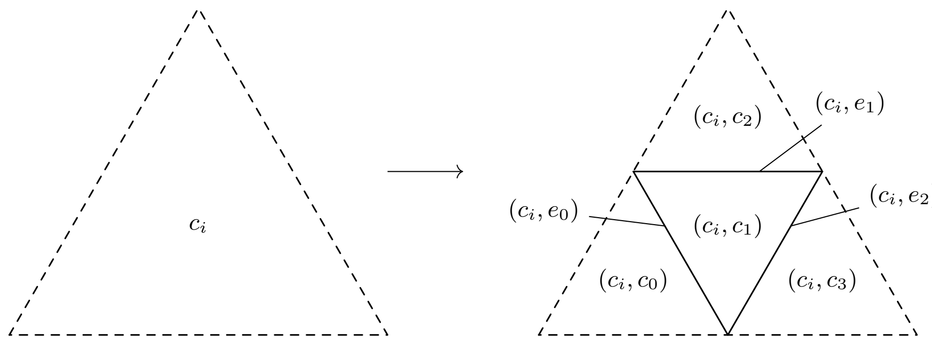

(c_i, c_1)
(c_i, e_2, e_0)
(c_i, c_3, c_2)


c_i
c_0 c_1 c_2 c_3 e_0 e_1 e_2
c_1 c_2 c_3
e_0 e_1 v_0

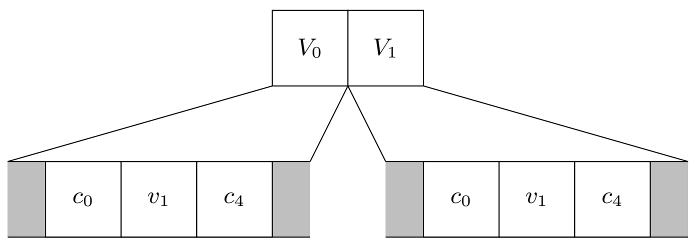



$(v_i,)$
(v_i, v_0)


$(e_i,)$
(e_i, e_0)
(e_i, e_1)
(e_i, v_0)



4.2 Data layout transformations



5 Conclusions

References

[1] Rainer Agelek et al. "On Orienting Edges of Unstructured Two- and Three-Dimensional Meshes". In: *ACM Transactions on Mathematical Software* 44.1 (July 24, 2017), pp. 1–22. ISSN: 0098-3500, 1557-7295. DOI: `10.1145/3061708`. URL: `https://dl.acm.org/doi/10.1145/3061708` (visited on 07/01/2021).

[2] M. Homolya and D. A. Ham. "A Parallel Edge Orientation Algorithm for Quadrilateral Meshes". In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016), S48–S61. ISSN: 1064-8275, 1095-7197. DOI: `10.1137/15M1021325`. URL: `http://epubs.siam.org/doi/10.1137/15M1021325` (visited on 06/29/2021).

[3] Andreas Klöckner. "Loo.Py: Transformation-Based Code Generation for GPUs and CPUs". In: *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming - ARRAY'14* (2014), pp. 82–87. DOI: `10.1145/2627373.2627387`. arXiv: `1405.7470`. URL: `http://arxiv.org/abs/1405.7470` (visited on 10/14/2020).

[4] Kaushik Kulkarni and Andreas Kloeckner. “UFL to GPU: Generating near Roofline Actions Kernels”. In: *Proceedings of FEniCS 2021, Online, 22–26 March*. Ed. by Igor Baratta et al. 2021, p. 302. DOI: 10.6084/m9.figshare.14495301. URL: http://mscroggs.github.io/fenics2021/talks/kulkarni.html.

[5] G.R. Mudalige et al. “Design and Initial Performance of a High-Level Unstructured Mesh Framework on Heterogeneous Parallel Systems”. In: *Parallel Computing* 39.11 (Nov. 2013), pp. 669–692. ISSN: 01678191. DOI: 10.1016/j.parco.2013.09.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167819113001166 (visited on 10/12/2020).

[6] G.R. Mudalige et al. “OP2: An Active Library Framework for Solving Unstructured Mesh-Based Applications on Multi-Core and Many-Core Architectures”. In: *2012 Innovative Parallel Computing (InPar)*. 2012 Innovative Parallel Computing (InPar). San Jose, CA, USA: IEEE, May 2012, pp. 1–12. ISBN: 978-1-4673-2633-9 978-1-4673-2632-2 978-1-4673-2631-5. DOI: 10.1109/InPar.2012.6339594. URL: http://ieeexplore.ieee.org/document/6339594/ (visited on 01/28/2021).

[7] Marie E. Rognes, Robert C. Kirby, and Anders Logg. “Efficient Assembly of $H(\mathrm{div})$ and $H(\mathrm{curl})$ Conforming Finite Elements”. In: *SIAM Journal on Scientific Computing* 31.6 (Jan. 2010), pp. 4130–4151. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/08073901X. URL: http://epubs.siam.org/doi/10.1137/08073901X (visited on 03/18/2022).

[8] Matthew W. Scroggs et al. “Basix: A Runtime Finite Element Basis Evaluation Library”. In: *Journal of Open Source Software* 7.73 (May 25, 2022), p. 3982. ISSN: 2475-9066. DOI: 10.21105/joss.03982. URL: https://joss.theoj.org/papers/10.21105/joss.03982 (visited on 10/17/2022).

[9] Matthew W. Scroggs et al. “Construction of Arbitrary Order Finite Element Degree-of-Freedom Maps on Polygonal and Polyhedral Cell Meshes”. Mar. 23, 2021. arXiv: 2102.11901 [cs, math]. URL: http://arxiv.org/abs/2102.11901 (visited on 06/29/2021).

[10] Tianjiao Sun et al. “A Study of Vectorization for Matrix-Free Finite Element Methods”. In: *The International Journal of High Performance Computing Applications* 34.6 (Nov. 2020), pp. 629–644. ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342020945005. URL: http://journals.sagepub.com/doi/10.1177/1094342020945005 (visited on 10/13/2020).