High performance mesh abstractions

Connor Ward

October 10, 2022

0.1 Introduction

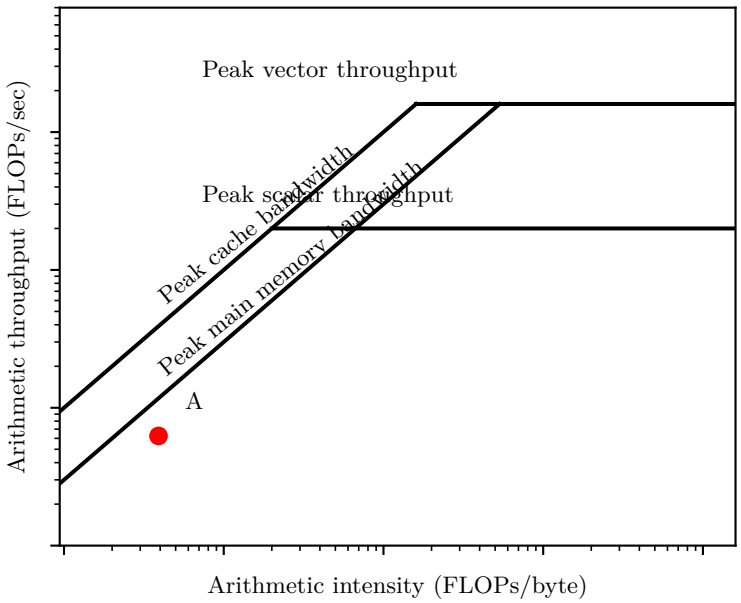
In scientific computing, the composition of appropriate (software) abstractions is essential for scientists to write portable and performant simulations in a productive way (the three P's). Having suitable abstraction layers allows for a separation of concerns whereby numericists can reason about their 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 mesh 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?

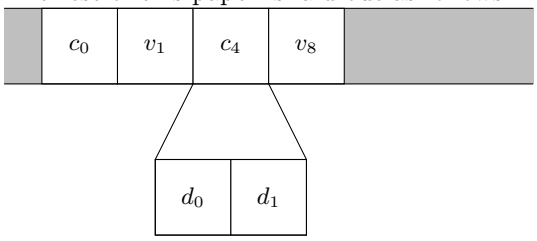
To begin, the wall-clock time of a program, assuming good algorithmic/par-

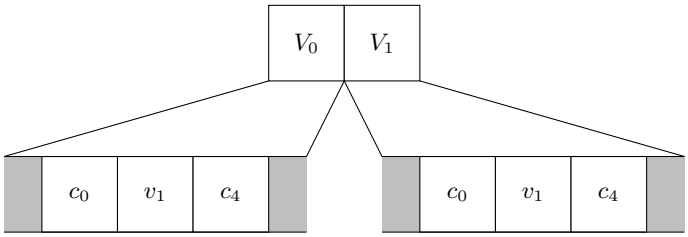


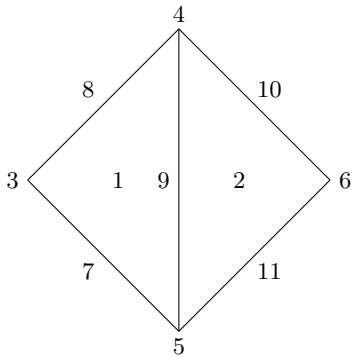
allel design, is usually limited by some combination of the maximum throughput of the processor and the cost of moving data to and from said processor. Given that the principle role for `pyop3`is to marshall data for the kernel we focus on codes where the cost of moving data is the bottleneck. Any effort 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.

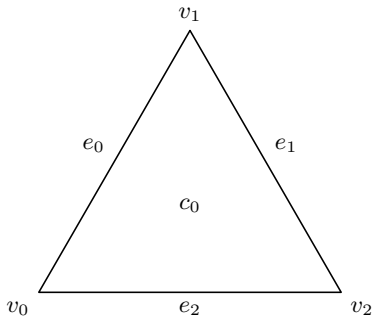
In this work, we present `pyop3`…

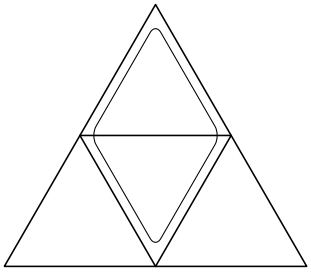
The rest of this paper is laid out as follows: …

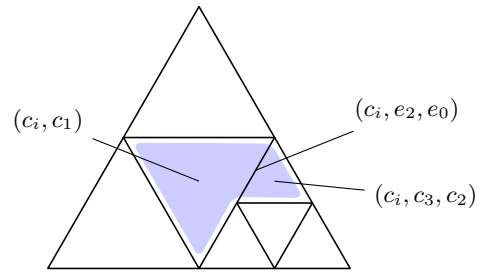


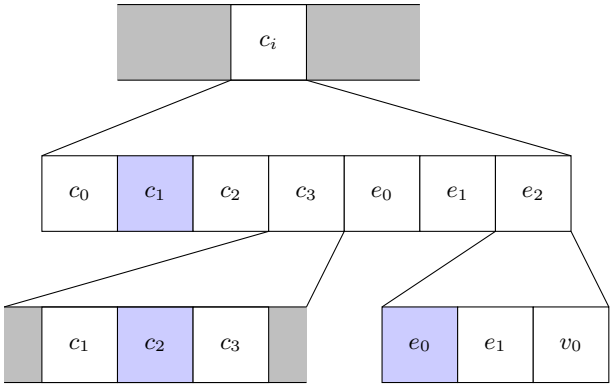

V_0
V_1
c_0
v_1
c_4
c_0
v_1
c_4

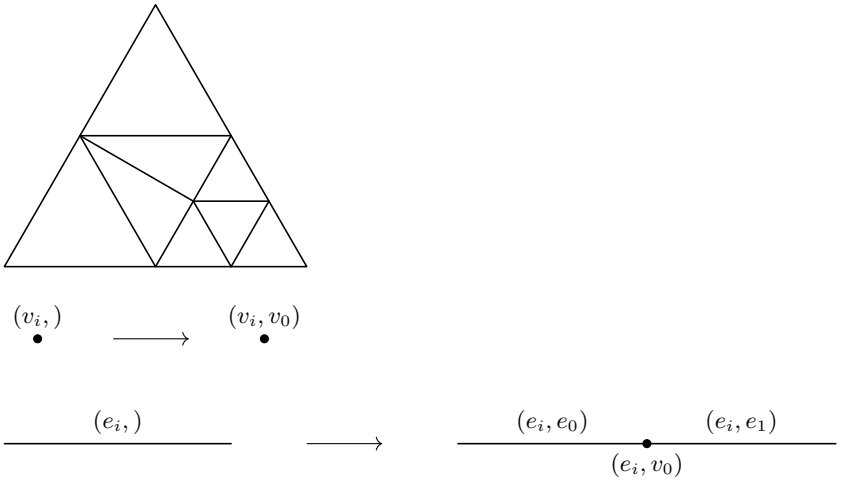

4
8
10
3
1
9
2
6
7
11
5

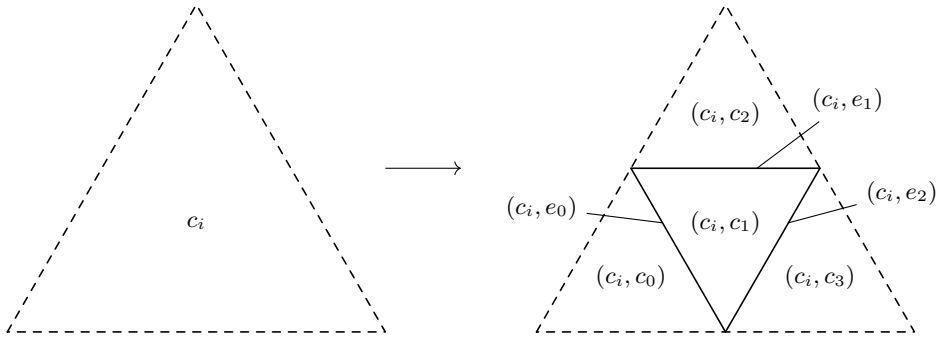

v_1
e_0
e_1
c_0
v_0
e_2
v_2




(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)


c_i
(c_i, c_2)
(c_i, e_1)
(c_i, e_0)
(c_i, c_1)
(c_i, e_2)
(c_i, c_0)
(c_i, c_3)