An abstraction for local computations on structured meshes and its extension to handling multiple materials

Daniel Becker, Istvan Z. Reguly
Faculty of Information Technology and Bionics
Pazmany Peter Catholic University
Budapest, Hungary
becker.daniel.balazs@hallgato.ppke.hu, reguly.istvan@itk.ppke.hu

Gihan R. Mudalige
Department of Computer Science
University of Warwick
Coventry, United Kingdom
g.mudalige@warwick.ac.uk

***Abstract*—Computations involving a neighbourhood on structured meshes represents a wide class of applications that includes the simulation of cellular automata, and the solution of partial differential equations (PDEs). In this paper we present an abstraction for describing such computations at a high level, allowing fast experimentation and productivity. The abstraction is designed such that it can be automatically converted to various high-performance implementations. A critical feature of this abstraction is an extension to support a varying number of materials, or species, at each grid point, enabling much more complex simulations.**

***Index Terms*—Structured meshes, stencils, PDEs, multimaterial**

I. Introduction

COMPUTATIONS based on a localised "stencil" have wide-ranging applications in scientific computing. As such, there are innumerable implementations done by scientists both for one-off experiments and for large software used in production. In many cases, the computational performance achieved by such codes is vital for scientific exploration, therefore many utilise modern many-core architectures such as multi-core CPUs or GPUs.

Today's wide variety of modern many-core architectures, with their differing programming styles, require considerable programming effort to run applications well. The programmer has to consider several levels of parallelism; even CPUs now include wide vector units in each core, as well as multiple cores, and complex memory hierarchies that often involve several levels of caches which may be explicitly programmable. If a certain piece of code needs to run well on multiple architectures, the coding effort is multiplied, and subsequently maintenance is more difficult as well: any modification has to be applied to multiple codebases.

Embedded Domain Specific Languages (eDSLs) [1], [2] target a specific domain of computations, and offer an abstraction that can be used to express computations in that domain using a familiar programming language (such as C++). Such an approach, while limiting in its scope, allows its users to describe their algorithms without having to describe how it needs to be parallelised and how data needs to be moved. In more complex cases it can even abstract the use of data structures as well. Developers of such DSLs can utilise domain-specific knowledge to address the challenges of productivity, performance and portability.

In this paper, we present such an eDSL for local computations on structured meshes, and discuss how it can be used to express a variety of computational patterns common in this problem domain. The current proof-of-concept implementation does not make use of parallelisation and only runs on one CPU core, but plans include code generation for multi-core CPUs and GPUs.

II. The abstraction

The main component of the abstraction is an N dimensional rectangular mesh. In the mesh, each cell corresponds to a portion of physical space. Each cell may contain zero, one - or in the more complex case - at most M materials. Once the abstract mesh is defined, we can associate data with it - commonly used to store *state variables*.

Datasets associated with the mesh may be:

- Defined by cell, for example the volume of each cell. The dimensionality of the dataset will correspond to the dimensionality of the mesh.
- Defined by material, for example the thermal capacity of each material, invariant of cells.
- Defined by cell and material, for example the fractional volume of each material in each cell. The dimensionality of this dataset will be one higher than that of the mesh: we can store values for each material in each cell.
- In addition, datasets that are neither defined by cell nor material may be used in the computations. These are not actually associated with the mesh. They can either be scalars or one-dimensional arrays.

The bulk of the code written will of course describe the algorithm: with our abstraction, an algorithm is a sequence of parallel loops over the mesh, reading values at any given point in the mesh, potentially also accessing neighbours using the pre-defined stencils, and writing values at the given grid point. The dimensionality of the loop is always defined by

the dimensionality of the data that is being written - when reading, accessing lower dimensional data is also allowed (e.g. accessing material data in a cell-material loop). Reductions are also supported.

Our proposed abstraction provides the following interface to manipulate the data.

First, it needs to be specified which part of the spatial mesh the computation will be performed on; the iteration space. This may be the whole mesh or an N dimensional rectangular subregion of it. This can for example be useful in the handling of boundary conditions if calculating the value of a cell needs to access neighbouring cells — choosing an appropriate subregion of the mesh guarantees that such neighbours exist.

Furthermore, each computation takes a number of datasets as parameters. There are seven kinds of parameters:

- IN parameters are read-only, it is not possible to mutate the dataset through this parameter.
- OUT parameters are for writing out the results of the computation at the given grid point.
- REDUCE parameters can only be datasets that are either defined only by cell or only by material, or neither. These are used to store the result of a reduction operation on a full-rank matrix (defined by both cell and material) either along the material dimension or all spatial dimensions. Reductions along individual spatial dimensions may be supported in the future. An aggregation function has to be provided along with the dataset. The aggregation function must be a binary commutative and associative function (summation is a good example of that).
- NEIGH parameters provide read-only access to a neighbourhood of the cells. The relative offsets of the neighbours to be used need to be specified along with the dataset.
- INDEX parameters provide access to the spatial coordinates of the cells. These may be useful in implementing complex boundary conditions. These parameters do not take arguments, and pass an array of integers to the *user kernel*.
- FREE_SCALAR parameters provide read-only access to scalar values that are not associated with the mesh.
- FREE_ARRAY parameters provide read-only access to one-dimensional arrays that are not associated with the mesh.

It is allowed to take a dataset both as an IN and an OUT parameter, but the effect of modifying it through the OUT parameter will not be visible when reading it through the IN parameter. To avoid making parallelisation much more difficult, it is forbidden to take the same dataset both as NEIGH and OUT.

After providing the datasets that will be used in the computation (we will call these *dataset parameters*), the user defines an operation to perform on them. This is done locally — the user only needs to write the part of code that will be applied to every cell and material pair. In other words, they do not need to explicitly write a loop, they only write what would be in the innermost loop body. Indeed, there is even no need to index the elements of the dataset arrays. All the user needs to do is provide a function (a *user kernel*) that takes a parameter for every dataset parameter. In the case of IN and OUT parameters, the parameters of the kernel are scalar values or references obtained by indexing the dataset with the given cell and material indices (in the case of datasets defined only by cell or only by material, the other index is ignored). For REDUCE dataset parameters, a special wrapper type is used as the corresponding kernel parameter to take care of the aggregation. In a similar manner, the kernel parameter corresponding to a NEIGH dataset parameter is a wrapper type providing access to neighbouring cell data. The kernel parameter corresponding to INDEX parameters is simply an N dimensional index type. The order of the kernel parameters is the same as the order of the corresponding dataset parameters.

III. Algorithms

In this section we will discuss some examples of using the proposed API. The following code demonstrates the application of an edge filter on a 128×128 mesh, which only has one material per cell. The computation is performed on the inner 127×127 region to ensure the required neighbours exist for every cell in the computation.

```
const std::size_t COLS = 128;
const std::size_t ROWS = 128;
const std::size_t MAT_N = 1;

Data<2> data({COLS, ROWS}, MAT_N);

CellData<2> x = data.new_cell_data();
CellData<2> y = data.new_cell_data();

Stencil<2> s9pt({{1,1},  {1,0},  {1,-1},
                 {0,1},  {0,0},  {0,-1},
                 {-1,1}, {-1,0}, {-1,-1}});

// Fill the datasets with data.

IndexGenerator<2> index_generator({1, 1},
                                  {127, 127});
Computation<2> computation(data,
                           index_generator);

computation.compute(
    [] (NeighProxy<CellData<2>> x, double& y)
    {
    y =
      -x[{1,1}]  - x[{1,0}]   - x[{1,-1}]
      -x[{0,1}]  + 8*x[{0,0}] - x[{0,-1}]
      -x[{-1,1}] - x[{-1,0}]
- x[{-1,-1}];
    },
    NEIGH<CellData<2>>(x, s9pt),
    OUT<CellData<2>>(y));
```

Once there are multiple materials per cell, there are additional types of loops. The most straightforward type is where we independently iterate over all cell-material pairs, for example to compute the mass of the material in that cell: $material\_density * material\_fractional\_volume\_in\_cell * cell\_volume$. In such a computation, density and fractional volume are defined on both cells and materials, but the cell volume is only defined on cells. Further types of computations include: accessing values of the same material on adjacent cells (e.g. locally averaged density), reduction of values along all spatial dimensions (e.g. total mass of each material on the whole mesh) and reduction of values along the material dimension (e.g. total mass of materials in each cell).

The following example demonstrates a multimaterial computation with reduction. Given a *density* and a *volume* state variable (both defined by both cell and material), we will calculate the total mass of the materials in each cell, summing the masses of the individual materials in each cell:

```
const std::size_t COLS = 200;
const std::size_t ROWS = 200;
const std::size_t MAT_N = 50;

Data<2> data({COLS, ROWS}, MAT_N);

CellMatData<2> density
    = data.new_cell_mat_data();
CellMatData<2> volume
    = data.new_cell_mat_data();

// Fill the datasets with data.

CellData<2> mass_by_cell
    = data.new_cell_data();

auto sum = [] (double left, double right) {
    return left + right;
};

IndexGenerator<2> index_generator(
                                {0, 0},
                                {COLS, ROWS});
Computation<2> computation(data,
                           index_generator);
computation.compute(
    [] (double density,
        double volume,
        ReduceProxy mass_by_cell)
    {
        mass_by_cell << density * volume;
    },
    IN<CellMatData<2>>(density),
    IN<CellMatData<2>>(volume),
    REDUCE<CellData<2>>(sum, mass_by_cell));
```

The next example illustrates the usage of mesh-invariant datasets.

```
const std::size_t COLS = 128;
const std::size_t MAT_N = 50;

Data<1> data({COLS}, MAT_N);

const double dt = 1e-2;

CellData<1> input = data.new_cell_data();
CellData<1> output = data.new_cell_data();

Stencil<1> neighs({{-1}, {0}, {1}});

// Fill the dataset with data.

IndexGenerator<1> index_generator({1}, {COLS - 1});
Computation<1> computation(data, index_generator);

computation.compute(
    [] (const double dt,
        const NeighProxy<CellData<1>> input,
        double& output) {
            output = dt * (
                - input[{-1}]
                + 2*input[{0}]
                - input[{1}]
                );
        },
        FREE_SCALAR<>(dt),
        NEIGH<CellData<1>>(input, neighs),
        OUT<CellData<1>>(output));
```

IV. Data structures

This section deals with the representation of datasets defined by both cell and material.

The most straightforward data structure to store such datasets is a two-dimensional array of size $N \times M$, where *N* is the total number of cells and *M* is the number of materials. Accessing values is easy and requires only simple pointer arithmetics. However, in practice, most cells usually contain only one or at most a few materials. If there are many cells and materials, such a full-matrix representation becomes extremely wasteful.

This is not only a problem of wasting memory space — also the locality of the data that we actually want to use deteriorates. On modern hardware with layered cache structures, accessing non-local data is potentially orders of magnitude more expensive than accessing local data.

Therefore, a more compact representation of the data is often desirable, which only stores data for materials that have non-zero fractional volume in a cell. We will briefly present one such data structure, taken from [3].

The central part of the data structure is a table that is stored with the mesh. For each cell, the table contains information about the number of materials (*nmats*) in that cell and an additional integer (*imaterial*). If there is only one material in

a given cell, the corresponding *nmats* value in the table will be -1 instead of 1, while in the case of multimaterial cells, the value is simply the number of materials in the cell. The *imats* value of single-material cells is a negative number, the absolute value of which is the id number of the material in that cell. The *imats* value of multimaterial cells is an index into a separate array (we will call it the linked list array) that stores the data of all multimaterial cells. The linked list array is also stored with the mesh, not with individual datasets.

The slots in the linked list array contain the following fields:

- *frac2cell*, which is the id number of the cell the slot belongs to
- *material*, which is the material id number
- *nextfrac*, which is the index of the next slot corresponding to the same cell as this slot. If no such slot exists, the value is -1. This field makes it possible to treat this structure as an array-backed linked list. This way, materials can be added to or removed from a cell later.

The actual values of the datasets are stored in the following way. For each dataset, an array is allocated that stores an element for each cell. For those cells that only contain one material, the corresponding value in the array is the value that belongs to that material. For cells that contain multiple materials, the value is unspecified.

For multimaterial cells, the dataset values are stored in arrays that run parallel to the linked list array. If a slot in the linked list array has index *i*, the corresponding value in the parallel array is also at index *i*.

In traditional multi-material codes, the management of the data structure is done by the user, intermixed with the science code — with our abstraction, this is completely hidden away, and may actually be changed easily to better support a given target architecture.

ACKNOWLEDGMENT

István Reguly was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Project no. PD 124905 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_17 funding scheme.

REFERENCES

[1] G. R. Mudalige, M. B. Giles, I. Reguly, C. Bertolli, and P. H. J. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *2012 Innovative Parallel Computing (InPar)*, May 2012, pp. 1–12.

[2] I. Z. Reguly, G. R. Mudalige, M. B. Giles, D. Curran, and S. McIntosh-Smith, "The ops domain specific abstraction for multi-block structured grid computations," in *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, Nov 2014, pp. 58–67.

[3] R. V. Garimella and R. W. Robey, "A comparative study of multi-material data structures for computational physics applications," Tech. Rep.