Improving Locality of Unstructured Mesh Algorithms on GPUs

András Attila Sulyok[a,*], Gábor Dániel Balogh[a], István Zoltán Reguly[a], Gihan R Mudalige[b]

[a] *Faculty of Information Technology and Bionics, Pázmány Péter Catholic University, Hungary*
[b] *Department of Computer Science, University of Warwick, United Kingdom*

Abstract

To most efficiently utilize modern parallel architectures, the memory access patterns of algorithms must make heavy use of the cache architecture: successively accessed data must be close in memory (spatial locality) and one piece of data must be reused as many times as possible (temporal locality).

Unstructured mesh algorithms are notoriously difficult in this sense, especially due to computations that indirectly modify data, leading to race conditions. In this work we address this problem through a number of optimisations on GPUs, specifically the use of the shared memory and a two-layered colouring strategy to cache the data. We also look at different block layouts to analyse the trade-off between data reuse and the amount of synchronisation.

We developed a standalone library that can transparently reorder the operations done and data accessed by a kernel, without modifications to the algorithm by the user. Using this, we performed measurements on relevant scientific kernels from different applications, such as Airfoil, Volna, Bookleaf, Lulesh and miniAero; using Nvidia Pascal and Volta GPUs. We observed significant speedups (1.2–2.5×) compared to the original codes.

*Corresponding author
Email address: `sulyok.andras.attila@hallgato.ppke.hu` (András Attila Sulyok)

Keywords: finite volume, finite element, race condition, GPU

1. Introduction

To satisfy the computational needs of scientific algorithms, it is increasingly necessary to parallelise their solution on modern multi-core CPUs, and GPUs. In the past ten years since the first release of the CUDA language extension for C/C++ the GPUs became widely used for high performance and scientific computations. CUDA provides a low level abstraction commonly referred to as Single Instruction Multiple Threads (SIMT), which gives us fine grained control over GPU architectures.

A significant class of scientific applications operate on unstructured meshes, which are represented as sets and explicit connections between them. These applications are operating on large sets and usually execute similar computations for every set element. For processing the elements, they can access data on the set which they operate on, or, using indirections, data defined on other sets. In the latter case multiple threads may try to modify the same data, leading to race conditions. This algorithmic pattern is the focus of our work: operations on sets that read and most importantly *increment* data indirectly on other sets. These operations are common in numerical PDE solvers, such as Finite Volumes and Finite Elements. The adoption of GPUs for these kind of computations could lead to considerable speedups due to the highly parallel execution of the code. However, the straightforward parallelisation of the scientific algorithms may not lead to optimal performance on GPUs. The SIMT abstraction exposes a number of techniques that can improve performance, one of which is the explicit management of the memory system of the GPU.

The proper usage of the memory system for simulations is crucial to get good performance. The latency of global memory accesses is a bottleneck for a lot of kernels in modern scientific computations, thus we have to lower the impact of the memory transactions. Since we can't fully hide the latency with computations, we can either increase the number of memory transactions in flight (to more efficiently utilise bandwidth), or decrease the number of memory transactions with high latency. To achieve the latter goal we can use shared memory for CUDA thread blocks as an explicitly managed cache, because it has much lower latency than global memory. However, to get good performance we have to ensure that we can use these memories efficiently from all threads in the blocks.

Most codes do one of two things: they use a straightforward parallelisation relying on either colouring or large temporary datasets to avoid two threads that run simultaneously accessing the same data, but these end up with poor data locality for most of the cases: one cannot have good reuse in reading data, but no conflicts (and no reuse) in writing it. The only advantage of this approach is that the parallelisation is simple, which means that it has lower overhead at the beginning to plan the execution of the kernel. However, if we spend more time for planning the execution strategy for the kernel, for example by altering the order in which the elements are processed we can significantly improve data locality. Better memory locality means that we can use cache lines more efficiently (with one read transaction we can read multiple pieces of data that we can use for the simulation) so that we end up with fewer memory transactions in total, which means less time spent waiting for data.

Reordering and partitioning algorithms are already used for maximising data reuse in CPUs and minimising communication in distributed memory systems. In our work we use them to improve the data locality on GPUs, specifically within CUDA thread blocks, to get better performance. We make the following contributions:

1. We adopt a caching mechanism on the GPU that loads indirectly accessed elements into shared memory. We use hierarchical colouring to avoid data races.
2. We design a reordering algorithm based on partitioning that increases data reuse within a thread block, also increasing shared memory utilisation.
3. We implement a library that parallelises applications on unstructured meshes and is capable of reordering the threads to increase efficiency.
4. We analyse the performance of various parallelisations and data layout approaches on several representative applications and GPUs.

The rest of the paper is structured as follows: the rest of Section 1 introduces the basic concepts of unstructured meshes and also discusses some related works, Section 2 describes the used algorithms and the motivation behind them, Section 3 shows the structure of our library, then Section 4 shows the test-cases and the measurements we performed. Finally, in Section 5, we draw some conclusions from the measurements.

1.1. Background

1.1.1. Unstructured meshes

In the scientific and high performance communities concurrent operations on different meshes are commonly used; meshes can be used as discretisations to numerically solve partial differential equations (PDEs). Usually these simulations require millions of elements to get correct results. Structurally, we can differentiate between structured and unstructured meshes.

Structured meshes are defined by blocks. Inside the blocks we have elements and implicit connectivity between them. The blocks are organized in a structured manner meaning that for every element in a block we can get all the neighboring elements simply from indexing arithmetic without any further information.

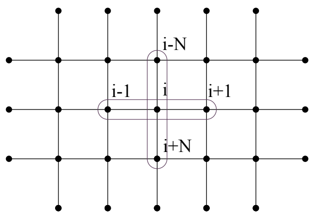

Figure 1: Structured mesh and stencil describing access pattern

Structured applications consist of loops over blocks which access datasets defined on blocks and the access patterns required for an element is described by stencils. A part of a block and a stencil describing access shown in Figure 1. Most operations on structured grids can be easily parallelised even on GPUs since all nodes can be processed simultaneously.

Unstructured meshes are defined by sets (e.g. nodes, edges, faces...), data defined on the sets, and explicit connectivity information between sets (mapping tables). For determining the neighbours of a set element the index isn't enough, we need external information. Thus there is an overhead associated with the reading of the mapping tables to determine the neighboring elements, but we gain significant performance when we need finer resolution for interesting parts of the mesh while a rougher resolution is sufficient for the most of the mesh (e.g. in the case of PDE-s where the gradient changes slowly). In such a case with structured meshes we need to compute unnecessary points otherwise we wouldn't get correct results in areas where detail is necessary.

For unstructured meshes the computations on the mesh are given as operations on sets. Basically a loop over the elements of a set, executing the same operations for every element, while accessing data directly on the iteration set or indirectly through a mapping. An example for a mesh and a mapping from edges to cells are shown in Figure 2. For kernels that only access data directly or only read data indirectly but write the data on the iteration set, the whole iteration space could run simultaneously. However, for kernels which indirectly

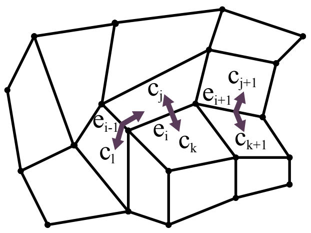

Figure 2: Unstructured mesh, the arrow represents the mapping tells e_i is connected to c_j and c_k.

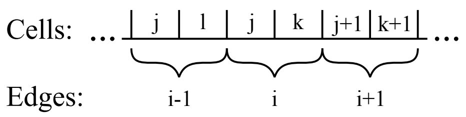

Figure 3: A part of the mapping from edges to cells.

increment data, we need to ensure that there are no race conditions during the execution.

The parallelisation of such unstructured mesh operations is much harder than for structured mesh codes: where the elements are writing data indirectly we can not tell which elements could be computed at the same time from compile time information, as it is driven by the mapping table. One of the most commonly used approaches is to use a runtime colouring of elements to parallelise the computation.

In our case the sets are represented as consecutive indexes from zero to the size of the set. The mapping between two sets are represented as a mapping table, an array which stores the index of set elements in the second set of the mapping (referred as to-set) for every set element of the first set (from-set). For the example in Figure 2 a part of the mapping from edges to cells is shown in Figure 3. In this way we can access the index of those elements which are connected to the current element of the from-set from other sets (the to-sets of the mappings).

In our case we used some restrictions for the applications, but out work is general enough that we are able to represent most applications in a way that doesn't violate our restrictions. The first is that there is no data access through multiple mappings. This means every data that we access is either accessed directly with the index of the set we iterate on or it is accessed with an index from one of to-sets of the accessible mappings get with using the index of the current element. This restriction does not exclude applications using nested indirections, since we can create a mapping that contains the indexes that we access through multiple mapping. The second restriction is that the result of the operations on the sets are independent from the order of processing the elements of the sets. This restriction is necessary to ensure that we can parallelise the operations, because there is no guarantee for execution order in the parallelised versions.

1.2. Related work

Algorithms defined on unstructured meshes have been around for a long time - discretisations such as finite volumes or finite elements often rely on these meshes to deliver high-quality results. Indeed there is a large number of papers detailing such algorithms, and a large number of commercial, government, and academic research codes; to name a few OpenFOAM [1], Rolls-Royce Hydra [2], FUN3D [3]. All of these codes use unstructured meshes in some shape or form, and since they are often used for large experiments, the efficiency of their implementation is an important concern. The problem of effectively parallelising applications using unstructured meshes is far from trivial and a number of solutions can be found in the literature.

There are a large number of papers discussing the efficient implementation and parallelisation in classical CPU architectures [4, 5], FPGAs [6, 7], and of course GPUs, which is the focus of our work.

There are libraries and domain specific languages that target unstructured mesh computations on GPUs, such as Liszt [8], OP2 [9], or more specialised ones such as the OCCA[10] open source library by Recacle et al. [11], which looks at efficiently solving elliptic problems on unstructured hexahedral meshes. Some of these libraries do use shared memory for improving data locality, however advanced techniques, such as reordering and partitioning studied in this work are not evaluated.

In work done by Castro et al. [12] on implementing path-conservative Roe type high-order finite volume schemes to simulate shallow flows, they also used auxiliary accumulators to avoid race conflicts while incrementing. Wu et al. [13] introduce caching using the shared memory with partitioning (clustering), but they do not use colouring, instead, they use a duplication method similar to that of Lulesh and miniAero, as described below. Fu et al. [14] also create contiguous patches (blocks) in the mesh to be loaded into shared memory, although they partition the nodes (the to-set) not the elements (the from-set of the mapping), further, they do not load all data into shared memory, only what is inside the patch. Writing the result back to shared memory is done by a binary search for the column index and atomic adds, which is inefficient on the GPU.

Mudalige et al. [15] work on the OP2 library, which is a basis of the present work. In the CUDA implementation, they use shared memory for caching with hierarchical colouring, but they do not reorder the threads, nor the data, to increase data reuse.

Parallel to these works the US Department of Energy labs have released a set of proxy applications that represent large internal production codes, showing some of the computational and algorithmic challenges that they face. In the Lulesh [16] and the miniAero [17] applications, two methods are proposed to address race conflicts: the first is to allocate large temporary arrays where the intermediate results (the increments) are placed, avoiding any race conditions, and to use a separate kernel to gather the results; the second is to use atomics. Both lead to increased warp divergence and high data latencies; and the use of the temporary array also leads to much more data being allocated and moved.

In our work, instead of porting any specific computation, we present a general approach to accelerate unstructured mesh applications, and in particular the indirect increment algorithmic pattern, on GPUs.

2. Parallelisation on GPUs

In this section we outline the techniques used to effectively optimise unstructured mesh applications on GPUs. We briefly show a naïve solution, then continue with describing various improvements others used, and then show our approach.

2.1. Traditional parallelisation approaches

On a GPU, groups of threads (warps) run at the same time, in lockstep, so it is not efficient to run computations of different length on different threads. Because of this, the usual practice is for each thread to take responsibility for one element on which computation is carried out (the iteration set or from-set), because then the number of computations is fixed: the amount of data involved is fixed in the dimension of the mapping and the data arrays.

Care must be taken when writing parallel code to avoid data races when different threads modify the same data. It could happen that the threads are scheduled so that both reads occur before the writes. For example, if the threads increment by 1 and 2, respectively, then the final result could be an increment of 3 (when one increment happens fully before the other), 2 (when the second thread writes last), 1 (when the first thread writes last). There are several approaches to tackle this.

A solution would be to colour each thread according to the indirect data it writes, so that no two threads with the same colour write the same data, and enforce ordering between colours using synchronisation. On the GPU, one would do multiple kernel launches corresponding to the colours, so there is no concurrent writes between threads in the same kernel. We call this the *global colouring* approach. The problem with this is that there is no data reuse: when multiple elements write the same data, they are scheduled for execution in different launches. Since these operations also tend to read data through the same mappings, there is no data reuse in the reads either. Worsening the issue is low cache line utilisation: elements of the same colour are not neighbours in the mesh, and therefore unlikely to be stored in consecutive memory locations.

Another approach is to serialise the indirect updates by means of locks or atomic additions. This is quite costly on the GPU, since the whole warp has to wait at the synchronisation step leading to warp divergence.

Yet another solution is the use of a large temporary array that stores the results for each thread separately, avoiding race conditions by formulation. But after the computation finishes, another kernel is required to gather the results corresponding to one data point. This suffers from the problem of not knowing how many values one thread has to gather, and the warps could diverge significantly, and memory access patterns are less than ideal (they can be good either for the write or the read, not both). Also, the size of the helper array is the number of elements multiplied by the dimension of the mapping. As a result, it can be quite large, for example, in LULESH, it is $8 \times 3 \times numElem$ in our measurements (where $numElem$ is the size of the from-set in Lulesh), compared to the array defined on nodes where these values will ultimately end up, which is roughly the same as the number of elements themselves.

2.1.1. Array-of-Structures (AoS) to Structure-of-Arrays (SoA) transformation

Because of the lockstep execution, consecutive threads in a warp are reading the memory at the same time, hence the layout of the data in the memory is an important factor. We used two different layouts in our measurements. The first is the Array-of-Structures layout which means that the data associated with one element is in consecutive places in the array (and thus in memory). The other layout is the Structure-of-Arrays where we store the components of elements consecutively e.g. the first data component of the elements are in the beginning of the array followed by the second, etc.
Although in most cases the SoA gives better performance on GPUs and better vectorisation on CPUs, the AoS layout is still commonly used on CPU architectures with large caches. In the case of the AoS layout, consecutive threads read data from strided addresses in memory and thus more cache lines are required to satisfy one transaction (which could be compensated by subsequently reading the other components, but caches on GPUs are small). However with the SoA layout the threads read data next to each other which means that the data needed by consecutive threads are probably in the same cache line and we get coalesced memory transactions. However, when indirections are involved, these access patterns become more complicated — even with the SoA pattern, consecutive threads may not be reading consecutive values in memory, and therefore cache line utilisation degrades. The choice of data layout in unstructured mesh computations is therefore highly non-trivial, as we show later.

2.2. Shared memory approach

Our approach exploits the shared memory on the GPU, which has much lower access latencies than the global memory, but is only shared within thread blocks. The idea is to collect the data required to perform the computations and load it into shared memory. During computation, the indirect accesses are to the shared memory, and the result is also stored there. After computation by all threads in the block have finished, the global memory is updated with the contents of the shared memory.

Note that fetching the data from the global memory (and also writing the result back) can be done by the threads independently of which thread will use which pieces of data for the computations. Particularly, fetching can be done in the order the data is laid out in memory, ensuring the utilisation of cache lines as much as possible. Also, if AoS layout is used, data can be read in contiguous chunks as large as the number of components in the structure.

To address data races, we use *two-layered colouring* or *hierarchical colouring*[15]: we colour the blocks and the threads within a block as well. The former is to avoid data races when thread blocks write the result back to global memory and the latter is to avoid threads writing their results into shared memory at the same time.

Note that the data that is loaded into shared memory is not necessarily only the data that is updated. The partitioning is done with regard to the the shared memory, the colouring with regard to the updated data. In fact, all of the indirectly accessed data can be loaded if it fits into shared memory (or does not cause such a low occupancy that offsets the benefits of high reuse).

The steps done by a kernel are described in Algorithm 1. All indirect data accessed by the block (this is identified during a preprocessing phase) is fetched from global to shared memory. This operation consists of two nested loops: an iteration over the data points, and within that, an iteration over the data corresponding to the data point. For the SoA layout, only the outer loop needs to be parallelised, since this will cause parallel read operations to access memory addresses next to each other. For the same reason, if the AoS layout is used, both parallel loops need to be parallelised (collapsed into one).

The data layout in shared memory is always SoA: our measurements showed a consistent degradation in performance when switching to AoS layout, due to the spatial locality described in Section 2.1.1: it leads to fewer bank conflicts.

After the data is loaded into shared memory, the results of the thread are computed by the main body of the kernel, and placed into registers. Next the threads update the result in shared memory with their increments. Finally the updated data is written back to global memory. Note that this is not the same data array as the one used as the input for the computations, since those are values calculated by a previous kernel or iteration.

Algorithm 1 Algorithm to use the shared memory to preload indirect data accessed within a thread block. global_indirect holds the data indirectly read, global_indirect_out holds the result of the iteration.

```
tid = blockIdx.x * blockDim.x + threadIdx.x
bid = blockIdx.x
for all d ∈ indirect_data do
    shared[shared_ind(d)] = global_indirect[d]
end for
__syncthreads()
result = computation(shared[mapping[tid]], global_direct[tid])
__syncthreads()
fill shared with zeros
__syncthreads()
for c = 1 ... num_thread_colours do
    if c == thread_colours[tid] then
        increment shared with result
    end if
    __syncthreads()
end for
for all d ∈ indirect_data do
    increment global_indirect_out with shared
end for
```

One other benefit from using the shared memory with hierarchical colouring is data reuse within the block: each piece of data has to be loaded from global memory only once, but can be used by multiple threads (e.g. data on a shared edge between two triangles). However, the greater the reuse, the more thread colours we have: the number of colours is no less than the number of threads writing the same data. Since the number of synchronisations also grows with the number of thread colours (more precisely, it is the number of colours plus two, one before and one after the computation if the input and the increment are stored separately in shared memory), there is a trade-off between the number of synchronisations and data reuse. Our measurements showed that if the kernel is memory-bound, the greater data reuse leads to increased performance, but the trade-off is non-trivial, as we will demonstrate in the Measurements section.

2.3. Increasing data reuse

The key contribution of our work is the reordering of the elements in the from-set (which map to the threads), so we can control how CUDA thread blocks are formed and how much data reuse can be achieved. We consider mainly the shared memory approach, where the benefit of data reuse is twofold: it decreases the number of global memory transactions and decreases the size of shared memory needed, which leads to greater occupancy.

We have looked at two different approaches: the first is the sparse matrix bandwidth reducing Gibbs-Poole-Stockmeyer algorithm, the second is partitioning.

2.3.1. Gibbs-Poole-Stockmeyer-based reordering

For serial implementations (typically on CPUs) of computations on graphs, the Gibbs-Poole-Stockmeyer (GPS, [18]) is a heuristic algorithm that increases spatial and temporal locality when traversing the nodes. It does this by renumbering the nodes, and with this, changing the order of traversal. (In this case, the edges are the elements of the from-set of the mapping, and the nodes form the to-set.)

It does this by going through the nodes in a breadth-first manner from two distant starting points, and then renumbers the nodes so that the levels

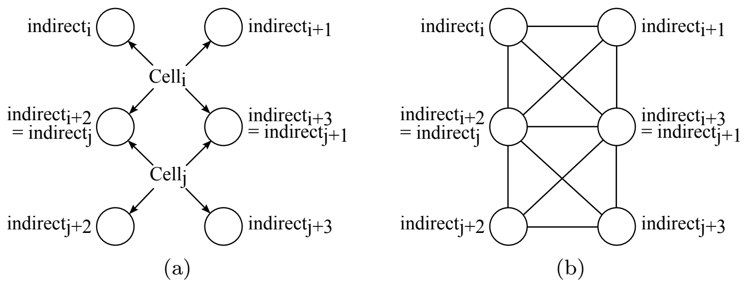

Figure 4: An example of converting a mesh (shown in (a), with mapping dimension 4) to a graph (on Figure (b)) for the GPS algorithm.

of the resulting spanning trees will constitute contiguous blocks in the new permutation.

After renumbering its points, which by design improves spatial locality, we order the edges of the graph lexicographically, so that consecutive threads (or spatial iterations in serial implementations) have a higher chance of accessing the same points, which improves temporal locality (data reuse).

The algorithm can be generalised to meshes by transforming each element into a fully connected graph of its points and then taking the union of these. An example of this is shown on Figure 4.

There are several straightforward generalisations to handle multiple sets and mappings (e.g. vertices, edges, cells and their connections). The first is to assume that all the mappings describe a similar topology, so the elements can be reordered based on one of the mappings (as described above), then reorder the points accessed through the other mappings by, for example, a greedy method. Another approach could be to reorder every data set separately, and then reorder the elements based on the new order of the accessed points, combining the separate data sets (and corresponding mappings) in some way. Since the mappings in the applications we measured are very similar topologically (in fact, except for Airfoil, there is only only one mapping in each application), we used the first method.

However, the algorithm fails to take into account that on the GPU the threads are grouped into blocks, and data reuse can only realistically be exploited within blocks. The next proposed algorithm will address this limitation.

2.3.2. Partition based reordering

To increase data reuse within a block is equivalent to decreasing data shared *between* the blocks, more specifically, to decrease the number of times the same data is loaded in different blocks. (With the sharing approach, data needs to be loaded only once per block.) So the task is to partition the elements into blocks of approximately the same size in such a way that when these blocks are assigned to CUDA thread blocks, the common data used (loaded into shared memory) by different blocks is minimised.

Let G_M be a graph constructed from the original mapping, where the points are the threads, and there is an edge between them if and only if they access the same data, and let $P_{G_M} = \{B_1, \dots, B_n\}$ be a partition of this graph with n blocks. This works even with multiple mappings.

If there is a set of blocks $B_{d_1}, \dots, B_{d_k}$ that access the same piece of data, then they form a clique in G_M in the sense that between any pair of blocks B_{d_i} and B_{d_j} (where $1 \leq i, j \leq k$), there is an edge of G_M between u and v such that $u \in B_{d_i} \land v \in B_{d_j}$. Note that the cliques have $0.5 \cdot (k^2 - k)$ edges, which is a monotone increasing function in k, since $k \geq 1$ (there is at least one block writing each data point, otherwise it is of no relevance).

That means that partitioning using the usual objective, ie. minimising the number of edges between blocks is a good heuristic for maximising data reuse within the blocks.

We chose the k-way recursive partitioning algorithm used by the METIS[19] library to partition the graph G_M. It is a hierarchical partitioning algorithm: it first coarsens the graph by collapsing nodes, then partitions using the recursive bisection algorithm, then, while progressively uncoarsening the graph, locally optimises the cuts.

The algorithm tries to maintain equal block sizes in the resulting partition, however, there are some differences. A tuning parameter is the load imbalance factor, which can be used to specify the tolerance for this difference. (It is called load imbalance because METIS is originally used for distributing computation, ie. load, in a distributed memory system.) It is defined as $l = n \max_j \{size(B_j)\}$, where n is the number of blocks and $size(B_j)$ is the size of the jth block. Due to the local optimisation in the uncoarsening phase, it is impractical to set this parameter to 1 (meaning the block sizes must be exactly the same). We found that a tolerance of 1.001 works well in practice for our needs.

In our library, the block size is a tuning parameter, which specifies the actual block size of the launched GPU kernels, naturally, the number of working threads cannot exceed it. Therefore, we calculate a new block size (S') and tolerance (l') with margins for the imbalance:

$$S' = \left\lfloor \frac{S}{l} \right\rfloor \tag{1}$$

$$l' = \frac{S + \epsilon}{S'}, \tag{2}$$

where S is the original block size, l is the original load imbalance parameter and ϵ is an empirical tuning parameter to create as large blocks (within the limit) as possible.

The variable number of working threads also incurs a slight overhead of two global loads for each thread: this number and the starting offset in the arrays. We found this overhead to be minimal in practice.

Due to the way global loads and stores work on the GPU, what actually affects performance is not the number of data points accessed, but rather the number of cache lines (of size 32 bytes) that are accessed. We used a simple heuristic reordering of data points that takes this into account.

The idea is to group data points together that are read/written by the same set of blocks, especially a set of more than one block, since any inefficiencies in that case will count more than once. To achieve this, we simply sort the data points by the number of partitions that write them, and within this, the indices of the partitions themselves.

2.4. Optimisations

There are a few optimisations we introduced to further improve performance.

During the load to and write from shared memory, in the case where the subsequent threads access addresses that are next to each other (the case when both loops are parallelised, as described in Subsection 2.2), we can make use of CUDA's built-in vector types (float2, float4 and double2) to load and write larger chunks (16 bytes) of data at a time.

To reduce warp divergence when updating the shared memory with the increments, the threads can be sorted within a block by their colours. After this, threads with the same colour will be next to each other, so warps will have fewer threads of different colours, hence less divergence on average.

To allow the compiler to apply some reordering optimisations that it wouldn't be able to do otherwise, the data pointers on the GPU is marked __restrict__ and const where applicable. The former tells the compiler that the pointers do not *alias* one another, ie. they do not point to the same memory space. The latter enables the compiler to place the data in texture cache that has lower latency than the global memory.

3. Library implementation

When implementing the library, the goal was to minimise the need for the users to modify their computation algorithms.

3.1. User kernels

The concept of a computation is in the form of a loop (or kernel) body. The same code can be used as in the serial algorithm, with the restrictions described in Section 1.1.1 and with the modification that, since the data layout is not necessarily the usual AoS, the accesses need to use the stride parameters (supplied to the kernel by the library) that are defined to be the distance between two consecutive data component (e.g. this is 1 in the case of AoS).

Four types of loops are supported: one serial and three parallel. The parallelisations are done by (1) OpenMP on the CPU, (2) CUDA with global colouring and (3) CUDA using the shared memory apporach with hierarchical colouring on the GPU. The CPU versions are not optimised and are only there for testing and verification purposes.

The user kernel consists of two main levels. In the first, the pointers to the data arrays are acquired, typecast and the result is written back into global (GPU or CPU) memory. This is similar in the differently parallelised loops (e.g. there is usually no difference between the serial and the OpenMP versions), but there are small variations in the use of GPU shared memory, the synchronisation steps and the calculation of the loop variable. (By loop variable, we mean the index from the from-set.) In our current implementation, the user creates this level from provided templates, but this can be easily automated by means of a source-to-source translator tool. The second level is the calculation itself that should be the same for all loop forms.

The data will be automatically copied to the device before the beginning of the loop when running on the GPU, and the result will be copied back to the host after. The pointers supplied to the user kernel also point to the location in the appropriate address space.

The directly accessed data is always in SoA form, while the layout of the indirectly accessed data is a user supplied compile time parameter. The shared memory in our kernels is in SoA form, but that is in the hand of the user. The layout of the mappings is AoS, except when using hierarchical colouring.

The synchronisation is done by the library using multiple kernel launches in the case of OpenMP and global colouring. The hierarchical colouring has some additional synchronisation steps, as described by Algorithm 1 in Section 2.2.

3.2. Execution planning

To avoid data races in the parallel loops, we colour the elements to avoid data races. Two kinds of colouring algorithms is used: global colouring is used for the OpenMP and the first CUDA parallelisation, while hierarchical or two-layer colouring is used for the shared memory approach.

The global colouring is a direct generalisation of the greedy graph colouring algorithm: for all elements of the from-set, its colour will be one of those that are used but none of its corresponding indirect points have. The function that chooses from the possible colours is a parameter; in the OpenMP and global colouring parallelisations, the colour with the least amount of from-set elements assigned to it (so far) is chosen.

The threads with different colours are started in different kernel launches. The mappings and the direct accessed data arrays are reordered according to their colour. This avoids introducing another indirection (which would map from their index in the current colour to the actual from-set element that it should be working on).

The hierarchical colouring reorders the threads according to the given partition and maps the threads to blocks. The block sizes are limited by a parameter (given by the user), if one block in the partition is larger, it is divided into two.

The same algorithm described above is then used to colour the thread blocks. Within the blocks we use heuristic of ordering the threads by removing the one with the minimum degree, placing it last in the ordering, then recursively ordering the other threads.

After colouring the threads are sorted according to their colour to reduce warp divergence, as described in Section 2.4.

The mappings and the direct accessed data are transformed into SoA layout to increase spatial locality. Similarly to the global colouring, these are then reordered according so they can be directly indexed by the loop variable.

3.2.1. Reordering

Before the execution of the loop an optional step is the reordering of from- or to-set elements using the GPS or the partitioning algorithms described in Section 2.3.

We use the Scotch library[20] for the GPS algorithm, and the METIS[19] library for the multilevel k-way partitioning algorithm (using 64 bit integers for indices). We also tried the recursive bisection algorithm in the METIS library, but the result was significantly worse, as well as the partitioning algorithm in the Scotch library, but that failed to stay within tolerance (and half of the blocks in the partition were empty, others were larger then requested).

It must be noted that these algorithms were developed for distributing workloads on computing clusters that typically have much larger block size to total size ratio. This also caused the reordering (partitioning) phase to be quite long: even minutes; but this is a one-off cost: the reordering can be saved from the library and reused many times later. Further improvement could be achieved by using the parallel versions of the METIS library: ParMETIS[21] and the alpha version mt-Metis[22] libraries. Partitioning algorithms targeting specifically small partition sizes required for CUDA thread blocks are a target of future research.

4. Measurements

4.1. Used applications

4.1.1. Airfoil

Airfoil is a benchmark application, representative of large industrial Finite Volume CFD applications. It is a non-linear 2D inviscid airfoil code that uses an unstructured grid and a finite-volume discretisation to solve the 2D Euler equations using a scalar numerical dissipation. The algorithm iterates towards the steady state solution, in each iteration using a control volume approach, meaning the change in the mass of a cell is equal to the net flux along the four edges of the cell, which requires indirect connections between cells and edges. Airfoil is implemented using the OP2 domain specific language [15], where two versions exist, one implemented with OP2's C/C++ API and the other using OP2's Fortran API [9, 23].

The application consists of five parallel loops: **save_soln**, **adt_calc**, **res_calc**, **bres_calc** and **update**. In our work we focus on **res_calc** since it has indirect increments and about the 70% of the time is spent in this kernel on GPUs with global colouring. The is the most complex loop with both indirect reads and writes; it iterates through edges, and computes the flux through them. It is called 2000 times during the total execution of the application and performs about 100 floating-point operations per mesh edge.
The tests are executed on a mesh containing 2.8 million cells.

4.1.2. Volna

Volna is a shallow water simulation capable of handling the complete life-cycle of a tsunami (generation, propagation and run-up along the coast) [24].

The simulation algorithm works on unstructured triangular meshes and uses the finite volume method. Volna is written in C/C++ and converted to use the OP2 library[15]. For Volna top three kernels where most time is spent: **computeFluxes**, **SpaceDiscretization** and **NumericalFluxes**. We focus on the **SpaceDiscretization** kernel since it has indirect increments and the 60% of the time spent inside this step on GPU with global colouring.

Tests are executed in single precision, on a mesh containing 2.4 million triangular cells, simulating a tsunami run-up to the US pacific coast. The kernel itself iterates over the edges and increments the cells.

4.1.3. BookLeaf

BookLeaf is a 2D unstructured mesh Lagrangian hydrodynamics application from the UK Mini-App Consortium [25]. It uses a low order finite element method with an arbitrary Lagrangian-Eulerian method. Bookleaf is written entirely in Fortran 90 and has been ported to use the OP2 API and library. Bookleaf has a large number of kernels with different access patterns such as indirect increments similar to increments inside **res_calc** in Airfoil. For testing we used the SOD testcase with a 4 million element mesh. The top three kernels with the highest runtimes are **getq_christiensen1**, **getacc_scatter**, **gather**. Among these there is only one kernel (**getacc_scatter**) with indirect increments so we tested our optimisations on this kernel.

4.1.4. MiniAero

MiniAero [17] is a mini-application for the evaulation of programming models and hardware for next generation platforms from the Mantevo suite [26]. MiniAero is an explicit (using 4th order Runge-Kutta) unstructured finite volume code that solves the compressible Navier-Stokes equations. Both inviscid and viscous terms are included. The viscous terms can be optionally included or excluded. For miniAero meshes are created in code and are simple 3D hex8 meshes. These meshes are generated on the CPU and then moved to the device. While the meshes generated in code are structured, the code itself uses unstructured mesh data structures and access patterns. This mini-application uses the Kokkos library [27].

For miniAero we tested the **compute_face_flux** kernel that computes the flux contributions of the faces and increments it with the appropriate cell flux values. The original code (depending on a compile time parameter) either uses the auxiliary **apply_cell_flux** kernel that does the actual incrementing by gathering the intermediate results from a large temporary array, or uses atomics to do it within the kernel. Both the atomics and the work of the auxiliary kernel was substituted in our code by colouring.

4.1.5. Lulesh

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH, [16]) represents a typical hydrocode representing the Shock Hydrodynamics Challenge Problem that originally defined and implemented by Lawrence Livermore National Lab as one of five challenge problems in the DARPA UHPC program and has since become a widely studied proxy application in DOE co-design efforts for exascale.

LULESH is a highly simplified application, hard-coded to only solve a simple Sedov blast problem that has an analytic solution [28] – but represents the numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications at the Lawrence Livermore National Laboratory. LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh.

Like miniAero, the mesh itself is structured (and generated in the code), but

	P100 (GP100)	V100 (GV100)
Streaming Multiprocessors (SM)	56	80
Max Thread Block Size	1024	1024
Max Warps / Multiprocessor	64	64
Max Threads / Multiprocessor	2048	2048
Max Thread Blocks / Multiprocessor	32	32
Max Registers / Thread	255	255
Shared Memory Size / SM	64 KB	64 KB
Max 32 - bit Registers / SM	65536	65536
Memory Size	16 GB	16 GB
L2 Cache Size	4096 KB	6144 KB

Table 1: Important informations about the NVIDIA Tesla P100 and V100 GPUs [29, 30]

the algorithm doesn't take this into account and accesses the data through an eight-dimensional mapping for the hex8 (brick) elements. In our measurements, we used a mesh where the the number of elements (from-set) was 4913000 and the number of nodes (to-set) 5000211.

For Lulesh, we tested the **IntegrateStressForElems** kernel that calculates the forces in the nodes. The original CUDA version of the code contracted this kernel with **CalcFBHourglassForceForElems**; the only modifications we did to this code for our measurements is to remove these parts from the kernel.

4.2. Experimental setup

For testing we used NVIDIA Tesla P100 and V100 GPUs, Intel(R) Xeon(R) CPU E5-1660 (3.20GHz base frequency, 1 socket with 8 cores) with Ubuntu 16.04. We used the nvcc compiler with CUDA 9.0 (V9.0.176). The parameters of the GPUs are shown in Table 1. We show mainly the results on the P100, however, the results were similar on the newer architecture.

When comparing performance of different versions, we used the achieved bandwidth that is the key performance metric with memory-bound applications such as our test applications. It is calculated by the following formula:

$$\frac{\sum_d w_d S_d}{T} \cdot I,$$

where d iterates over the datasets, w_d is 2 if the data is read and written, 1 otherwise, S_d is the size of the dataset (in bytes), T is the overall runtime of the kernel and I is the number of iterations.

We also collected other relevant metrics that describe the observations, such as

- data reuse factor (the average number of time an indirectly accessed data point is accessed),
- the number of read/write transactions from/to grobal memory, which is closely related to the data reuse factor but is affected by memory access patterns, and therefore cache line utilisation,
- the occupancy reflecting the number of threads resident on the SM versus the maximum - the higher this is, the better chance of hiding the latency of compute/memory operations and synchronisation
- the percentage of stalls occurring because of data requests, execution dependencies, or synchronisation,
- the number of block colours; the higher it is, the less work in a single kernel launch, which tends to lead to lower utilisation of the GPU,
- the number of thread colours; the higher this is the more synchronisations are required to apply the increments in shared memory — but also strongly correlates with data reuse,
- warp execution efficiency (ratio of the average active threads per warp to the maximum number of threads per warp).

Colouring	Global				Hierarchical			
Reordering	none		GPS	partition	no		partition	
Data layout	AOS	SOA	SOA	SOA	AOS	AOS	AOS	SOA
Bandwidth (GB/s)	71	94	105	67	212	227	270	225
Achieved Occupancy	0.63	0.45	0.45	0.45	0.44	0.52	0.59	0.42
Global Memory Read Transactions	52424k	45781k	41246k	66775k	21142k	21192k	13964k	14406k
Global Memory Write Transactions	14007k	14737k	13773k	20733k	5807k	57883k	3397k	3669k
Number of (Block) Colours	5	5	5	7	4	5	8	8
Number of Thread Colours	-	-	-	-	3	3	4	4
Reuse Factor	-	-	-	-	2	2	3.6	3.6
Issue Stall Reasons (Synchronization)	-	-	-	-	11%	9%	14%	14%
Issue Stall Reasons (Data Request)	-	-	-	-	69%	68%	61%	63%

Table 2: Collected performance metrics of the global and hierarchical colouring implementation of the **res_calc** kernel. Block size is 480.

Studying performance and these metrics help us understand and explain why ceratin variants are better than others.

4.3. Measurement results

4.3.1. Analysis of Airfoil

First we analyse the Airfoil application from the OP2 library — this is the most well understood and thoroughly studied example, therefore we go into more detail here — for later kernels and applications we then identify key differences.

Table 2 show the effect of reordering on the global colouring approach in the Airfoil application (**res_calc** kernel).

The SoA layout, as mentioned in Section 2.1.1, improves on the performance, since the threads in a warp access data addresses that are near each other. This can also be seen in the number of global memory read transactions as it is roughly 87% of that with AoS layout. This is even strengthened by GPS renumbering, which places data points that are accessed in consecutive threads close to each other (19% reduction in global read transactions compared to AoS).

The partition based reordering is primarily intended for the hierarchical colouring; it groups threads that access the same data together, while the global colouring puts them into different kernel launches, eliminating any chance for spatial reuse, therefore reducing performance.

The measurement on Figure 5 used the hierarchical colouring and the shared memory approach, that, coupled with the fact that it uses more registers (52 (SoA) and 48 (AoS)) compared to global colouring (48 and 40), leads to larger variations in occupancy and performance as the block size changes; for example, between block sizes 480 and 448 (44% versus 51%).

The key goal of this strategy is to better exploit data reuse by using the shared memory; the results of which show immediately in the number of global transactions as well: at block size 480, there is roughly 60% decrease in global read and write transactions, leading to three times the performance.

These also show that the reordering using partitioning is indeed effective. With a block size of 448, data reuse increased from 2 with the reference version, to 3.6, leading to the 19% performance gain over the version without reordering (AoS layout). This is also consistent with the number of global transactions: there is a 35% decrease in the number of reads and 41% decrease in the number of writes, and a decrease in the percentage of stalls occurring because of data requests: 61% with partitioning, 68% without.

With the increased reuse, the number of thread colours is also larger (4 versus 2.2) and this leads to more synchronisation: with reordering, 14% of the stalls were caused by synchronisation, compared to 9%.

This is further illustrated by Figure 6 that shows the relative speedup compared to the original OP2 version. In this case, the original version also used the shared memory approach, so the performance gain is caused by the reordering.

Although the number of block colours increased with reordering, it is only a

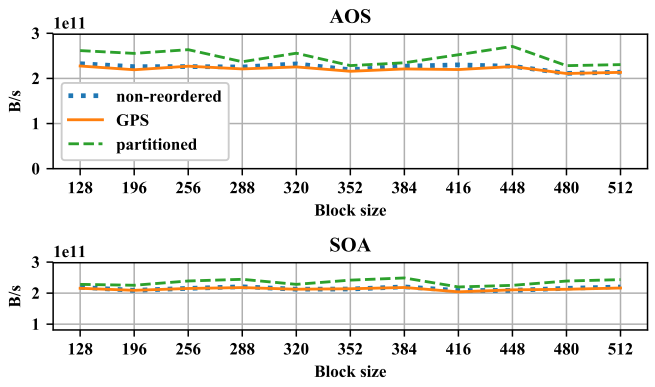

Figure 5: **res_calc** bandwidth on a dataset with 2880000 cells with hierarchical colouring. Note that the y-axis doesn't start at the origin.

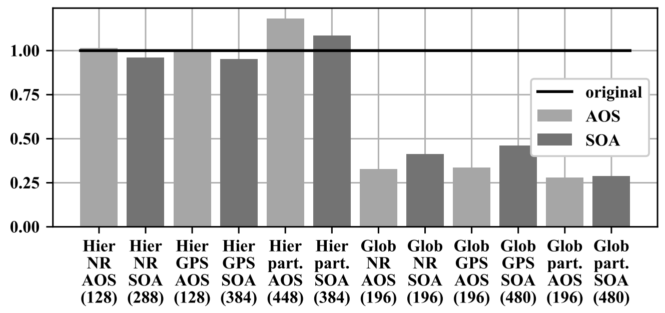

Figure 6: **res_calc** kernel speedup compared to the original code. Done on a dataset with 2800000 cells. The block sizes are shown in parentheses, the reordering algorithms are: the original reordering (NR), GPS reordering and partitioning (part.)

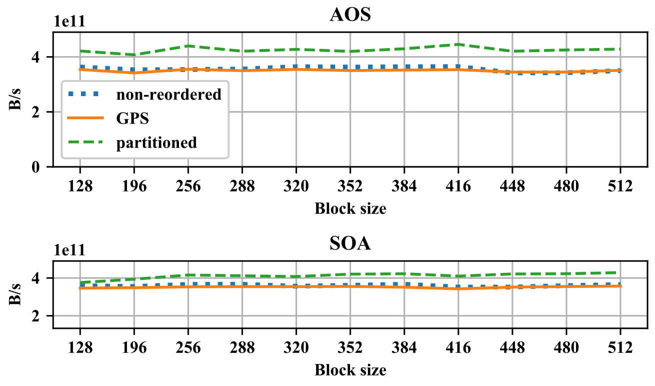

Figure 7: **res_calc** bandwidth on a dataset with 2880000 cells with hierarchical colouring, on Volta architecture. Note that the y-axis doesn't start at the origin.

problem when the size of the dataset is so small that it decreases the achieved occupancy (when there are not enough threads in one kernel launch to use the available resources on the GPU).

Despite the higher locality, in most cases, the AoS version also uses more registers (sometimes the compiler can optimise the register count of both versions to the same number), and this may lead to lower occupancy. Hence it is not always obvious which is the better choice.

When running on the newer Volta GPU architecture, the results look similar (Figure 7); the absolute value of the bandwidths are (understandably) higher.

4.3.2. Analysis of Volna

In measurements of the Volna application (**SpaceDiscretization** kernel) with hierarchical colouring (Figure 8) the reordering by partitioning again improves performance: it increases reuse from 1.5 to 2.8 and decreases the number of global transactions by 18% for reads and 37% for writes (Table 3). The larger reduction in writes can be explained by the fact that the calculation does not read the values of the indirect data from the previous iteration, only the direct data.

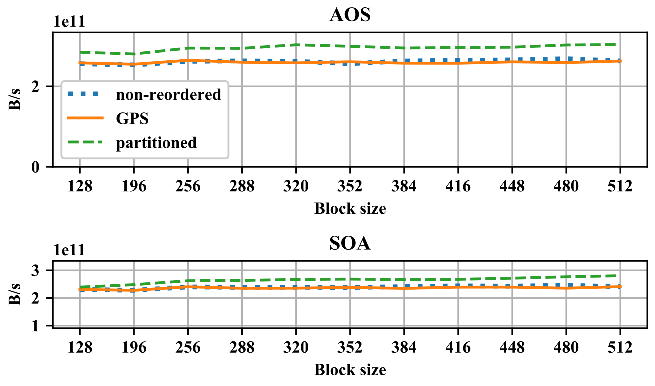

Figure 8: **SpaceDiscretization** kernel bandwidth on a mesh with 3589735 edges using hierarchical colouring. Note that the y-axis doesn't start at the origin.

Again, since the AoS version uses adjacent threads to load adjacent components of data points, and also because one thread loads 4 single precision values into shared memory using the built-in vector type float4, more data can be transferred at the same time, thus there is a 2% and 4% reduction in global memory transfers for reads and writes, respectively, leading to increased performance (292 GB/s versus 268 GB/s).

Due to the low register counts (28–32) and the fact that volna uses float (and int) datatypes compared to the doubles in Airfoil, the occupancy was quite high (around 80%). This explains the lack of dependence on the block size as shown by the figure.

Compared to Airfoil, the increase in the number of thread colours is less when partitioning: from 3 to 4, hence, the synchronisation overhead is also less: the percentage of stalls caused by synchronisation increases from 12% to just 15%. Of course, with high occupancy, the latency caused by synchronisation can be better hidden by running warps from other blocks.

As can be seen from Figure 9, our implementation is more than two times faster than the original OP2 version.

Reordering	no	no	partition	partition
Indirect data layout	AOS	SOA	AOS	SOA
Block size	448	448	448	448
Bandwidth (GB/s)	265	241	292	268
Achieved Occupancy	0.82	0.81	0.80	0.80
Warp Execution Efficiency	98%	98%	97%	97%
Global Memory Read Transactions	9114k	9166k	7493k	7617k
Global Memory Write Transactions	2438k	2512k	1542k	1640k
Issue Stall Reasons (Synchronization)	11%	12%	15%	15%
Issue Stall Reasons (Data Request)	51%	50%	46%	46%
Number of Block Colours	5	5	9	9
Reuse factor	1.5	1.5	2.8	2.8
Number of Thread Colours	3	3	4	4
Average cache lines/block	300	307	165	184

Table 3: Collected performance metrics of the hierarchical colouring implementation of the **SpaceDiscretization** kernel.

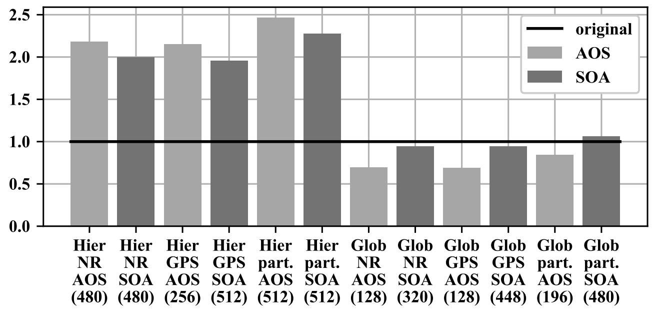

Figure 9: **SpaceDiscretization** kernel speedup compared to the original code, done on a mesh with 3589735 edges. The block sizes are shown in parentheses, the reordering algorithms are: the original reordering (NR), GPS reordering and partitioning (part.)

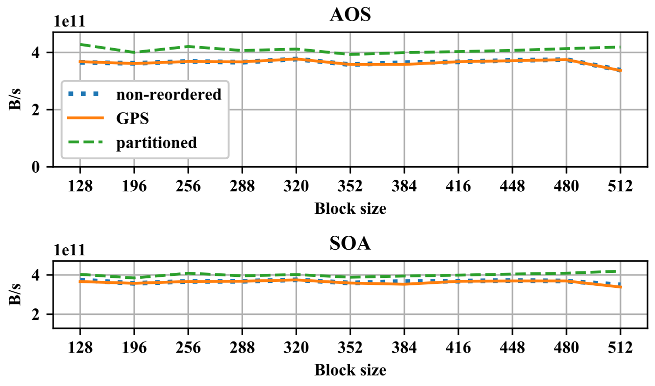

Figure 10: **getacc_scatter** kernel bandwidth on a mesh with 4000000 edges using hierarchical colouring. Note that the y-axis doesn't start at the origin.

4.3.3. Analysis of BookLeaf

The measurements on the BookLeaf application (specifically the **getacc_scatter** kernel), as can be seen on Figures 10 and 11, also benefits from the partitioning of the mesh.

The register count and occupancy are also similar to those with Airfoil (64 registers, achieving occupancy or around 40%), this leads to the variations in performance along different block sizes.

With partitioning, the number of thread colours increased from 2 to 5, this leads to the increased stalls from synchronisation: from 9% to 20%, while the reuse factor increases (from 2 to 3.5), which is comparable to that of Airfoil, this explains the smaller increase in performance (only 9%, compared to the 19% increase in Airfoil).

The higher data reuse leads to 14% and 41% decrease of the number of global transactions, for reads and writes, respectively. This large difference between reads and writes is also because, like **SpaceDiscretization**, **getacc_scatter** does not read values indirectly.

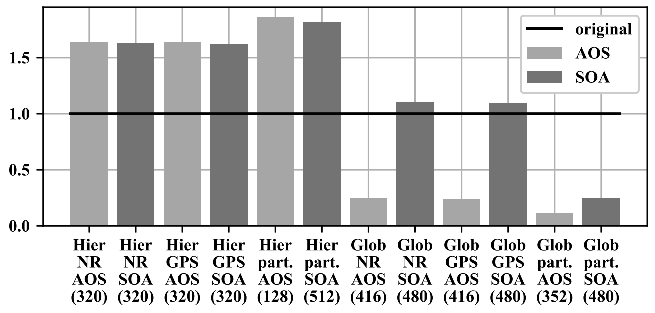

Figure 11: **getacc_scatter** kernel speedup compared to the original code, done on a mesh with 4000000 edges. The block sizes are shown in parentheses, the reordering algorithms are: the original reordering (NR), GPS reordering and partitioning (part.)

4.3.4. Analysis of LULESH

With LULESH (Figure 12), the partitioning approach actually worsened performance.

The **IntegrateStressForElems** kernel uses a mapping with 8 dimensions (compared to the 2 or 3 as in the case of the previous ones) that has high connectivity, so the number of colours is quite high: 8, 16 and 24 in the global coloring versions (for the different reorderings), and 4, 50 and 15 in the hierarchical colouring versions. The number of thread colours was also quite high: 4 in the non-reordered (4.5 in the GPS) and 11.6 in the partitioned version: this is a much higher increase compared to the previous applications (Table 4).

The other aspect in which LULESH is different is that it uses a high amount of registers (96), significantly decreases occupancy: with block size 320, the AoS version achieved 15% and the SoA version achieved around 30%.

Because of these two reasons, the synchronisation overhead (39% stalls were from synchronisation on the partitioned mesh) couldn't be hidden: there were no warps from other blocks to be scheduled in place of the stalled ones because there was only one block running on each multiprocessor.

In terms of reuse, although the gain (from 2.6 to 4.8) is larger compared

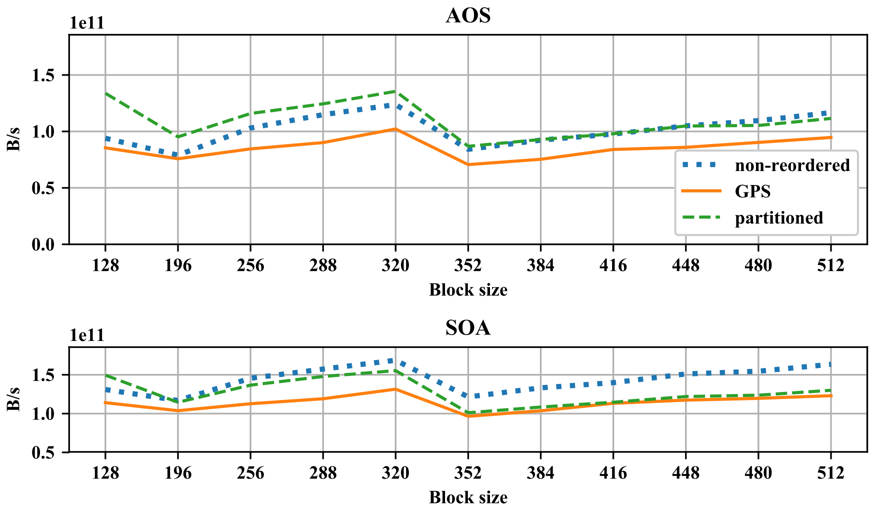

Figure 12: **IntegrateStressForElems** kernel bandwidth on a mesh with 4913000 cells using hierarchical colouring. Note that the y-axis doesn't start at the origin.

to Airfoil, it is still not enough to offset the overhead of synchronisation: in this case, the original ordering of the mesh is better. The difference in achieved occupancy also means that the SoA version with two blocks per multiprocessor performs better.

Nevertheless, as shown on Figure 13, our hierarchical colouring algorithm performs significantly better than the original, two-step implementation, and also uses much less memory. Although the original needs less synchronisation, the resulting warp divergence also cannot be hidden if the occupancy is this low.

4.3.5. Analysis of MiniAero

The **compute_face_flux** kernel is the most computationally intensive among the ones we tested: it uses 165 registers in hierarchical colouring (166 in SoA layout). Also, it achieves (with block size 384 and reordered by GPS) 15% of peak double precision efficiency, compared to the 6–7% in Airfoil (Table 5). It also uses 8 square root operations and several divides that can't efficiently fill

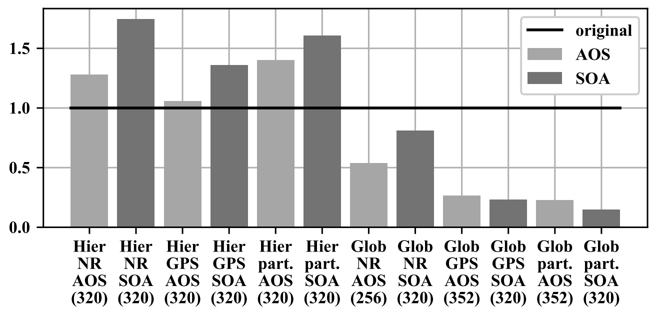

Figure 13: **IntegrateStressForElems** kernel speedup compared to the original (gathering) code, done on a mesh with 4913000 cells. The block sizes are shown in parentheses, the reordering algorithms are: the original reordering (NR), GPS reordering and partitioning (part.)

Reordering	no	no	partition	partition
Indirect data layout	AOS	SOA	AOS	SOA
Block size	320	320	320	320
Bandwidth (GB/s)	124	168	129	147
Achieved Occupancy	0.15	0.29	0.15	0.30
Warp Execution Efficiency	98%	97%	91%	90%
Global Memory Read Transactions (total)	33572k	35408k	22888k	25027k
Global Memory Write Transactions (total)	12673k	12704k	8047k	8701k
Issue Stall Reasons (Synchronization)	13%	19%	39%	42%
Issue Stall Reasons (Data Request)	64%	56%	35%	31%
Number of Block Colours	4	4	15	15
Reuse Factor	2.6	2.6	4.8	4.8
Average Cache Lines/Block	744	747	427	474
Number of Thread Colours	4	4	11.6	11.6

Table 4: Collected performance metrics of the hierarchical colouring implementation of the **IntegrateStressForElems** kernel.

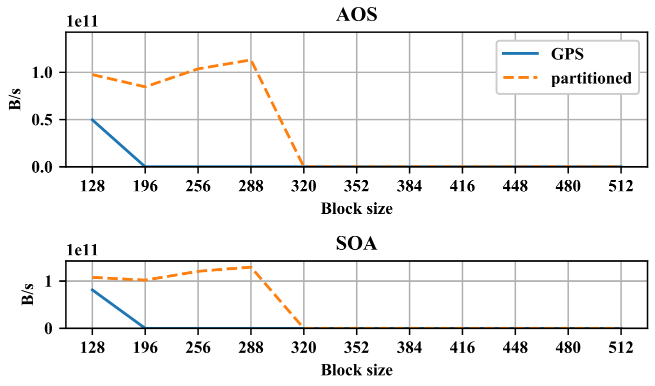

Figure 14: The **compute_face_flux** kernel bandwidth on a mesh with 6242304 faces. The kernel didn't run in the cases where the data reuse was not high enough because the large amount of shared memory needed; these are shown here with 0 bandwidth.

the pipelines at such low occupancy.

The amount of data indirectly accessed by the kernel is also large: each thread accesses 5 data points indirectly, each holding 28 double precision values. If all of that is loaded into shared memory, the size of it exceeds the hardware limits with block sizes larger than 288; it didn't run on the original mesh with any block size, and only with smaller block sizes on the reordered meshes (Figure 14). The other measurements were carried out by only loading the incremented data into shared memory.

The mesh also has a complex structure (18 and 15 block colours for GPS reordered and partitioned versions, respectively) and the original ordering was far from optimal: we couldn't run the non-reordered version, because the number of block colours exceeded the implementation limit of the library, which is 256.

As with LULESH, only one block was running at a time on each multiprocessor. Although the synchronisation overhead wasn't that high (3 and 6 thread colours in the GPS reordered and partitioned versions, respectively), the costly operations prevented high performance gains in the case of the partitioned ver-

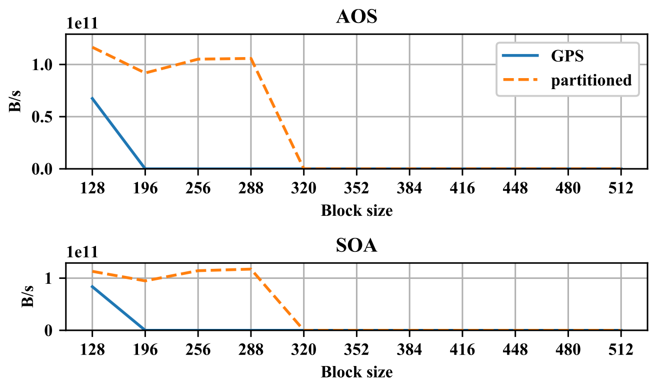

Figure 15: The **compute_face_flux** kernel bandwidth on a mesh with 6242304 faces. The shared memory was only used to cache the increments, reducing the need for large shared memory size. The kernel didn't fit into the shared memory with block size larger than 320 or if not reordered because the large amount of shared memory needed.

sion (Figure 15).

The original Kokkos implementation either used atomic adds or the two-step gathering approach depending on compilation parameters. Our implementation outperformed both (Figure 16), for the same reason as in case of LULESH.

4.3.6. Analysis of structured meshes

As mentioned in Sections 4.1.4 and 4.1.5, the meshes of miniAero and LULESH are generated by the code itself, therefore we can control the block-shapes explicitly by generating a partition with the mesh. This allows us to understand the compromise between high data reuse and few thread colours used more by creating 1D, 2D and 3D blocks (these will have an increasing amount of reuse and number of colours).

While both kernels operate on 3D Cartesian (hex8) meshes, the **IntegrateStressForElems** kernel uses a mapping from cells to their incident vertices, and the **compute_face_flux** kernel maps from (internal) faces to cells. In the former case, the blocks are created from the cells as rectangles.

In the second case, the creation starts the same way, but at the last axis (the

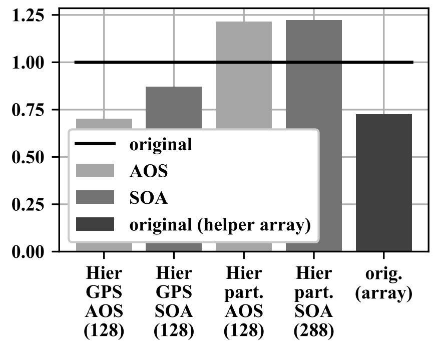

Figure 16: The **compute_face_flux** kernel speedup compared to the original code that was using atomic adds, on a mesh with 6242304 faces. The shared memory was only used to cache the increments, reducing the need for large shared memory size. The last bar shows the relative performance of the original code with the helper array approach.

Reordering	GPS	GPS	partition	partition
Indirect data layout	AOS	SOA	AOS	SOA
Block size	128	128	128	128
Bandwidth (GB/s)	83	157	107	136
Achieved Occupancy	0.06	0.06	0.12	0.12
Warp Execution Efficiency	91%	84%	88%	85%
Global Memory Read Transactions	73561k	82028k	53403k	77091k
Global Memory Write Transactions	9153k	10390k	6694k	9008k
Issue Stall Reasons (Synchronization)	4%	9%	15%	21%
Issue Stall Reasons (Data Request)	61%	35%	47%	35%
Issue Stall Reasons (Execution Dependency)	23%	33%	23%	23%
FLOP Efficiency(Peak Double)	5%	8%	10%	12%
Number of Block Colours	18	18	15	15
Reuse Factor	2.2	2.2	3.9	3.9
Average Cache Lines/Block	452	471	269	344
Number of Thread Colours	3	3	6	6

Table 5: Collected performance metrics of the hierarchical colouring implementation of the **compute_face_flux** kernel.

contiguous axis) three faces corresponding to one cell are added to the block at the same time. This way the blocks cover the whole mesh similarly to the previous case. This also means that when the block is long along the third axis, the reuse will be higher than it would be with the other two axes. Because of this, in our measurements, we ordered the dimensions of the blocks so that they are longest along this axis.

Figures 17 and 18 show the bandwidths, reuse factors and the number of thread colours across different block-shapes, along with the result of partitioning the same mesh using METIS. The size of the blocks is 128. These measurements were run on meshes with shape specifically taylored so that the handcrafted blocks can cover them without any gaps.

In **IntegrateStressForElems**, compared to the 134 GB/s bandwidth of the original ordering (which is a row major order, similar to ours when we use a block shape that is 128 cells long and only 1 cell thin in the other dimensions) and the 132 GB/s of the partitioned mesh (detailed in Section 4.3.4), we achieved 167 GB/s bandwidth using our handcrafted blocks, for the same block size (128). Note that using regular shapes is better for the thread colouring algorithm too: with METIS partitioning, the number of colours needed is higher than in the other cases.

In **compute_face_flux**, we achieved 167 GB/s bandwidth, compared to 101 GB/s achieved with partitioning.

Of course, using handcrafted blocks are "cheating", and can only be done when the mesh itself is not structured. However, these illustrate clearly that when the number of thread colours are the same, increased reuse leads to better performance. Also, there is an optimal number of thread colours for each application, and performance will suffer above that. The challenge lies in finding a partitioning algorithm that can either find the middle ground, or can be tuned

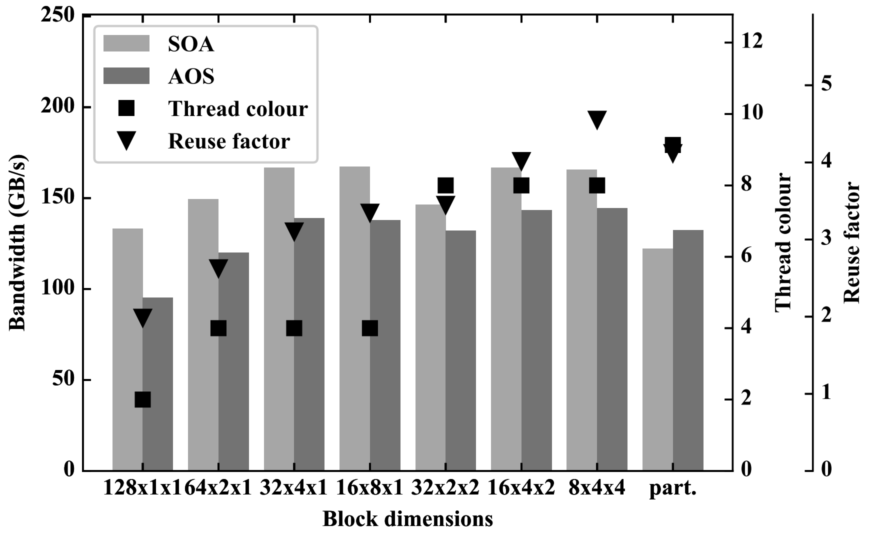

Figure 17: **IntegrateStressForElems** kernel with explicitly controlled partitioning. For comparison, the last column shows the result on the same mesh, partitioned by METIS.

along the amount of reuse it aims to achieve.

5. Conclusion

We investigated the methods of accelerating parallel scientific computations on unstructured meshes. We specifically looked at improving the performance of memory-bound implementations executing on GPUs.

We designed a reordering algorithm which uses k-way recursive partitioning to improve data reuse and with it, performance of unstructured mesh applications accelerated on GPU platforms.

We implemented a library that can automatically (without major modifications from the part of the user) parallelise serial user code, avoiding data races using global and hierarchical colouring. It uses optimisations specifically targeting GPUs, such as caching in shared memory, reordering by colours to reduce warp divergence and using vector types to more efficiently utilise available

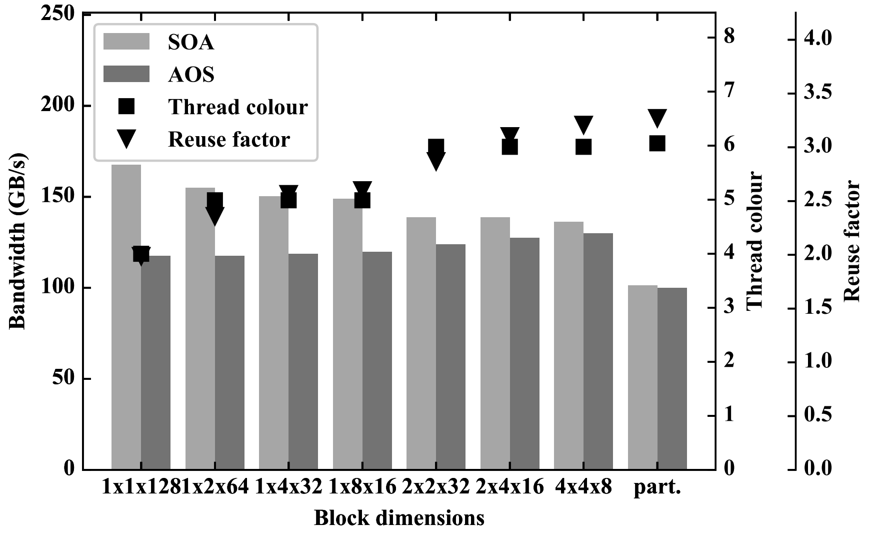

Figure 18: **compute_face_flux** kernel with explicitly controlled partitioning. For comparison, the last column shows the result on the same mesh, partitioned by METIS.

global memory bandwidth.

Using this library, we analysed the performance of our algorithms on a number of representative unstructured mesh applications varying a number of parameters, such as the different thread block sizes and data layouts (Array of Structs versus Struct of Arrays).

When comparing the performance of global and hierarchical colouring (shared memory caching) approach, the shared memory approach consistently performed better, since it could exploit the temporal locality in indirectly accessed data by avoiding data races in shared memory with synchronisation within thread blocks rather than different kernel launches.

We also analysed the performance of reordering based on GPS renumbering and partitioning. The former improves global colouring with increasing spatial reuse, while the latter can significantly improve the shared memory approach by increasing data reuse within thread blocks, which results in smaller shared memory and fewer global memory transactions.

We have shown that there is a trade-off between high data reuse and large numbers of thread colours in hierarchical colouring that is especially pronounced when the achieved occupancy is low: the more thread colours a block has, the more synchronisations it will need, the latency of which can be hard to hide when there are few eligible warps.

Using our methods, we were able to achieve performance gains of 10% (Airfoil), 140% (Volna), 75% (Bookleaf), 75% (Lulesh) and 25% (miniAero) over the original implementations. These results significantly advance the state of the art, demonstrating that the algorithmic patterns used in most current implementations (particularly in case of US DoE codes represented by LULESH and MiniAero) could be significantly improved upon by the adoption of two-level colouring schemes and partitioning for increased data reuse.

When carrying out this work, it had become clear that partitioning algorithms in traditional libraries such as Metis and Scotch weren't particularly well suited for producing such small partition sizes. As potential future work, we wish to explore algorithms that are better optimised for this purpose. The performance of these partitioning algorithms was also low - parallelising this could be another interesting challenge. Finally, we are planning to integrate these algorithms into the OP2 library, so they can be automatically deployed on applications that already use the OP2 library, such as Airfoil, BookLeaf, Volna or Rolls-Royce Hydra.

Acknowledgements

This paper was supported by the János Bólyai Research Scholarship of the Hungarian Academy of Sciences. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work http://dx.doi.org/10.5281/zenodo.22558. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

[1] OpenCFD, OpenFOAM - The Open Source CFD Toolbox - User's Guide, OpenCFD Ltd., United Kingdom, 1st Edition (11 Apr. 2007).

[2] P. Moinier, J. Muller, M. B. Giles, Edge-based multigrid and preconditioning for hybrid grids, AIAA journal 40 (10) (2002) 1954–1960.

[3] R. T. Biedron, J.-R. Carlson, J. M. Derlaga, P. A. Gnoffo, D. P. Hammond, W. T. Jones, B. Kleb, E. M. Lee-Rausch, E. J. Nielsen, M. A. Park, et al., Fun3d manual: 13.1.

[4] D. J. Mavriplis, Parallel performance investigations of an unstructured mesh navier-stokes solver, The International Journal of High Performance Computing Applications 16 (4) (2002) 395–407.

[5] H.-Q. Jin, M. Frumkin, J. Yan, The openmp implementation of nas parallel benchmarks and its performance.

[6] Z. Nagy, C. Nemes, A. Hiba, Á. Csík, A. Kiss, M. Ruszinkó, P. Szolgay, Accelerating unstructured finite volume computations on field-programmable gate arrays, Concurrency and Computation: Practice and Experience 26 (3) (2014) 615–643.

[7] T. Akamine, K. Inakagata, Y. Osana, N. Fujita, H. Amano, Reconfigurable out-of-order mechanism generator for unstructured grid computation in computational fluid dynamics, in: Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on, IEEE, 2012, pp. 136–142.

[8] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, et al., Liszt: a domain specific language for building portable mesh-based pde solvers, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, ACM, 2011, p. 9.

[9] M. Giles, G. Mudalige, I. Reguly, Op2 airfoil example.

[10] OCCA, `http://libocca.org/`.

[11] J.-F. Remacle, R. Gandham, T. Warburton, Gpu accelerated spectral finite elements on all-hex meshes, Journal of Computational Physics 324 (2016) 246–257.

[12] M. J. Castro, S. Ortega, M. De la Asuncion, J. M. Mantas, J. M. Gallardo, Gpu computing for shallow water flow simulation based on finite volume schemes, Comptes Rendus Mécanique 339 (2-3) (2011) 165–184.

[13] B. Wu, Z. Zhao, E. Z. Zhang, Y. Jiang, X. Shen, Complexity analysis and algorithm design for reorganizing data to minimize non-coalesced memory accesses on gpu, in: ACM SIGPLAN Notices, ACM, 2013, pp. 57–68.

[14] Z. Fu, T. J. Lewis, R. M. Kirby, R. T. Whitaker, Architecting the finite element method pipeline for the gpu, Journal of computational and applied mathematics 257 (2014) 195–211.

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

[16] I. Karlin, J. Keasler, R. Neely, Lulesh 2.0 updates and changes, Tech. Rep. LLNL-TR-641973 (August 2013).

[17] miniAero CFD Mini-Application, `https://github.com/Mantevo/miniAero`.

[18] J. Norman E. Gibbs, William G. Poole, P. K. Stockmeyer, An algorithm for reducing the bandwidth and profile of a sparse matrix, SIAM Journal on Numerical Analysis 13. doi:10.2307/2156090.

[19] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on scientific Computing 20 (1) (1998) 359–392.

[20] F. Pellegrini, J. Roman, Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: High-Performance Computing and Networking, Springer, 1996, pp. 493–498.

[21] G. Karypis, K. Schloegel, V. Kumar, Parmetis: Parallel graph partitioning and sparse matrix ordering library, Version 1.0, Dept. of Computer Science, University of Minnesota.

[22] D. LaSalle, G. Karypis, Efficient nested dissection for multicore architectures, in: European Conference on Parallel Processing, Springer, 2015, pp. 467–478.

[23] OP2 github repository, `https://github.com/OP2/OP2-Common`.

[24] D. Dutykh, R. Poncet, F. Dias, The volna code for the numerical modeling of tsunami waves: Generation, propagation and inundation, European Journal of Mechanics-B/Fluids 30 (6) (2011) 598–615.

[25] Uk mini-app consortium, `https://uk-mac.github.io`.

[26] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, R. W. Num-

rich, Improving Performance via Mini-applications, Tech. Rep. SAND2009-5574, Sandia National Laboratories (2009).

[27] H. C. Edwards, C. R. Trott, D. Sunderland, Kokkos: Enabling manycore performance portability through polymorphic memory access patterns, Journal of Parallel and Distributed Computing 74 (12) (2014) 3202 – 3216, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. doi:https://doi.org/10.1016/j.jpdc.2014.07.003. URL `http://www.sciencedirect.com/science/article/pii/S0743731514001257`

[28] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-490254.

[29] C. NVIDIA, Nvidia tesla p100, Tech. Rep. v01.1 (2016). URL `https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf`

[30] C. NVIDIA, Nvidia tesla v100 gpu architecture (2017). URL `https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf`

Appendix A. Parallelisation using the library

We implemented a library to automatically parallelise existing algorithms requiring only minimal modifications from the part of the user. What follows is a tutorial to using the library, to give you an idea of how general the necessary modifications are.

We followed a similar path to that of the OP-DSL family, only our “DSL” is so simple that it can be implemented using a few macro functions, and we did not use a source to source translator.

Supplying the algorithm to the library is done by modifying the `skeleton.hpp` and `skeleton_func.hpp` files in the `kernels` directory. All coding is done in the C++ language.

Appendix A.1. User function

The body of the parallel loop: the user function is defined using the USER_FUNCTION_SIGNATURE macro in the modified `skeleton_func.hpp` file. It can then call further inline functions.

It will be passed pointers to the first indirect and direct data it accesses and also the corresponding two strides. This way, it can for example access the ith component of the first indirect data point by calling indirect_data1[i $*$ indirect_stride].

The address of the output is also supplied, this is where the calculated indirect increment is placed. Only the increments are returned to allow controlled write-back of the results in different parallelisation methods.

All of the pointers should be marked with the RESTRICT macro that signals to the compiler that there is no overlap between the data accessed through these pointers.

Appendix A.2. Parallelisation administration

Before calling the user function, some administration overhead is needed, such as typecasting and extracting pointers from arrays, calculating the iteration variable and caching; this is implemented by modifying the `skeleton_func.hpp`.

The name of the file, the header guard and the enclosing namespace should be modified to something more descriptive of the application.

It is good practice to extract constants such as the dimension of the mesh and the number of components in the indirect and direct data points, these should be defined at the top of the namespace.

The different parallelisation methods (different kernels) are defined in different structs, all having one static call method that will be called by the library.

This has a boolean template parameter that is true if the indirect data is in Struct of Arrays layout (see Section 2.1.1). The direct data is always in SoA layout.

This method has arguments for the indirect input, indirect output, the directly accessed data points, the mappings. These pointers are of type void** (array of arrays structure), except for the mappings that are std :: uint32_t. The different arrays should be typecast to the appropriate types. There should also be a local array for the increments.

All kernels have different additional parameters. The sequential and OpenMP kernels have the iteration variable and the strides. With all these, the user function (user_func_host) can be called, then the results should be written back to the main memory.

The kernels running on the GPU have the same API as before, but the actual code is extracted into a separate function because CUDA does not support using a static method as kernel.

The kernel that runs on the GPU using the global colouring method is almost the same as the sequential, but the iteration variable can be extracted from the CUDA backend, hence the number of active threads (the number of threads that should do anything) are passed instead. After checking that the current thread id is less than the number of active threads, it proceeds as before, calling the GPU version of the user function (user_func_gpu).

The hierarchical kernel (the kernel that uses the hierarchical colouring method and caches into shared memory) has also as parameters the list of indirect data points to be cached (in Compressed Sparse Row format), the colours of the threads, the number of thread colours and the offsets of the blocks (because the variable block size).

The block offsets should be used to calculate the iteration variable, so instead of blockDim.x $*$ blockIdx.x $+$ threadIdx.x, it becomes block_offsets[blockIdx.x] $+$ threadIdx.x.

The skeleton for the cache-in and cache-out code is composed of nested conditionals that depend on the type of the data. Since the user knows the type, the structure can be simplified by removing any unneeded branch. Otherwise, only the variable names should be changed.

The computation code is similar to the sequential and global kernels, only the write-back of the increments is done in two steps: first the shared memory is cleared, then, in a for loop iterating over the thread colours, the shared memory is updated by the increments.

This is then written back to global memory.

Appendix A.3. Using the library

After the implementation of the algorithm, the library will take care of the parallelisation.

The data and mappings is read from specified input streams. To call the loop, the user calls on of the loopCPUCellCentred, loopCPUCellCentredOMP, loopGPUCellCentred, loopGPUHierarchical methods of the Problem class. The have a template parameter that is the struct implementing the kernel code.