Compiler Optimizations for Industrial Unstructured Mesh CFD Applications on GPUs[⋆]

C. Bertolli[1], A. Betts[1], N. Loriant[1], G. R. Mudalige[2], D. Radford[5], D. Ham[1,4], M. B. Giles[2,3], and P. H. J. Kelly[1]

[1] Dept. of Computing, Imperial College London,
{c.bertolli,a.betts,n.loriant,david.ham,p.kelly}@imperial.ac.uk
[2] Oxford e-Research Centre, University of Oxford,
{gihan.mudalige}@oerc.ox.ac.uk
[3] Mathematical Institute, University of Oxford, {mike.giles}@maths.ox.ac.uk
[4] Grantham Institute for Climate Change, Imperial College London
[5] Rolls Royce Plc., David.Radford@Rolls-Royce.com

Abstract. Graphical Processing Units (GPUs) have shown acceleration factors over multicores for structured mesh-based Computational Fluid Dynamics (CFD). However, the value remains unclear for dynamic and irregular applications. Our motivating example is HYDRA, an unstructured mesh application used in production at Rolls-Royce for the simulation of turbomachinery components of jet engines. We describe three techniques for GPU optimization of unstructured mesh applications: a technique able to split a highly complex loop into simpler loops, a kernel specific alternative code synthesis, and configuration parameter tuning. Using these optimizations systematically on HYDRA improves the GPU performance relative to the multicore CPU. We show how these optimizations can be automated in a compiler, through user annotations. Performance analysis of a large number of complex loops enables us to study the relationship between optimizations and resource requirements of loops, in terms of registers and shared memory, which directly affect the loop performance.

Keywords: Computational Fluid Dynamics, Unstructured Meshes, Graphical Processing Units, Compiler

1 Introduction

Unstructured mesh (or grid) applications are widely used in Computational Fluid Dynamics (CFD) simulations when complex geometries are involved. They achieve a higher degree of correctness by enabling critical components of the geometry to be finely discretized.

This comes at the cost of increased difficulty in achieving high memory system utilization. In structured mesh applications, compilers can leverage the topology

[⋆] This research is partly funded by EPSRC (grant reference numbers EP/I00677X/1, EP/I006079/1), the UK Technology Strategy Board, and Rolls Royce plc through the SILOET programme.

of the mesh which is explicit in the program structure. In contrast, in unstructured mesh applications, the mesh topology is not known at compile-time. It may include elements (e.g. triangular faces) of widely different sizes to reflect the modeller's interest in specific sections of the geometry, and consequentially the adjacency relationship is non-uniform. To support this flexibility, implementations depend on indirections between adjacent mesh elements, which prevent many structured mesh compiler optimizations. A typical instance of this behaviour is when a program visits all edges of the mesh and accesses data associated to vertices. To do so, it uses a *mapping* between edges and vertices, which represents the grid structure itself and expresses a non-affine access to arrays holding mesh data.

In this paper we consider a motivating example – HYDRA, an unstructured mesh finite-volume CFD application used at Rolls Royce for the simulation of inner turbomachinery components of jet engines. It consists of 50,000 lines of code, including more than 1,000 parallel loops over the mesh, and it supports the simulation of a wide range of CFD problems, including linear, non-linear and adjoint cases.

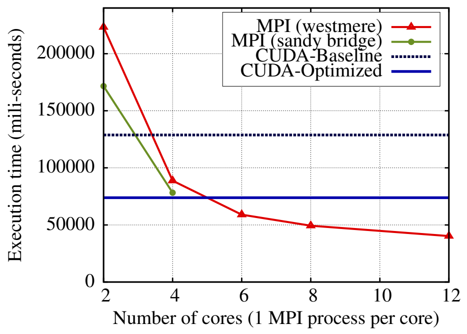
Our research aim is the acceleration of HYDRA through both strong and weak scaling, i.e. decreasing simulation times and increasing the size of the geometries modelled. For this purpose, HYDRA has been modified to use our unstructured mesh library, called OP2, which is supported by a compiler and run-time library. OP2 supports a wide range of architectures, including clusters of CPUs and GPUs. In this paper we focus on the acceleration of HYDRA on a single GPU node.

In a preliminary optimization phase, we studied the performance of HYDRA on a single multicore node using MPI, against that of a single GPU node. Our results showed that a baseline GPU implementation, featuring only standard unstructured mesh optimizations, is not sufficient to achieve performance comparable to the execution on a single CPU. To improve this situation, we identified pathological patterns in the HYDRA code. We used three optimizations to address those patterns: loop fission, an improved colouring strategy, and loop-specific tuning of partition size and CUDA thread block size. We applied these optimizations manually to four specific loops of HYDRA having low performance on a GPU. These results are shown in Figure 1: the execution of HYDRA on Intel Westmere and Sandybridge processors, using different numbers of cores using MPI, are compared to execution on an NVIDIA Fermi C2070.

In this paper we build on the experience and techniques gathered from our preliminary optimization steps. The described optimizations are automated in the compiler by extending the OP2 language with annotations. These are used by the programmer to signal the compiler that the optimizations can be applied to the annotated loops. This reduced significantly the compiler design complexity, as it does not need to analyze the entire user kernel code, but only the loop parameters and annotations.

As the described optimizations are composable for the same loop, the compiler is given tools to select the best combination of optimizations to be applied

Fig. 1: Comparison of HYDRA performance on single CPU and GPU. The starting point for this paper is the lower, manually-optimised performance.



to each loop. We take a step forward understanding what (composition of) optimizations actually deliver better performance. By taking advantage of the large number of complex OP2 loops available in HYDRA, we can put the performance improvement due to optimizations into relation with the loop features, and their resource requirements for GPUs. This represents a key step towards a fully-automatic optimizing compiler for unstructured mesh applications. The contributions of this paper are the following:

– We present an annotation-based scheme that allows our compiler to split complex loops over unstructured meshes in a way that optimises effective use of shared memory.
– We present performance analysis for this and other optimisations, separately and in combination, on a wide variety of OP2 loops from a substantial application case study.
– From this experimental work we characterise the properties of loops that most impact their performance. We show how OP2's access descriptors, in combination with quantitative compile-time metrics of shared-memory and register requirements, can be used to determine where these optimisations are valid and profitable.

2 Related Work

A huge literature exists related to optimisations for unstructured grid applications, or, in more general terms, for irregular applications. Most optimizations attempt to improve data locality through mesh renumbering, with the goal of improving cache usage (e.g. [1, 2]). Our run-time library is currently able to use either PT-Scotch [3], and METIS [4]. However, the performance results shown in this paper is based on well-ordered meshes. The optimisations that we present do not require the analysis of the iteration order, and they are based on input program transformations, alternative code syntheses, and run-time tuning.

The optimization strategy that we aim at developing shares similar goals with the work presented by Strout et al. in [5]. This introduces a framework for composing optimisations for irregular applications, where examples of such optimisations include iteration and data re-ordering, and loop tiling. The framework enables modelling of optimizations at compile-time in terms of undefined functions, which are then applied at run-time by analysing the mesh in the *inspection phase*. The result of the mesh inspection is the instantiation of the loop *execution phase*, with improved performance as a consequence of optimisations. The compiler framework allows sound composition of the undefined optimization functions, effectively providing an abstraction for composing optimizations.

A number of development projects include elegant abstractions for parallel computing on unstructured meshes using MPI. The most prominent research effort targeting intra-node parallelisation is theg Liszt project [6], which has many similarities with our work. Liszt is a domain specific language for programming unstructured mesh applications, and it targets performance portability across multiple different architectures. Unlike the OP2 compiler, the Liszt compiler synthesizes stencil information by analyzing user kernels, with the aim of applying platform-specific optimizations. Performance results from a range of systems (GPU, multi-core CPU, and MPI based cluster) executing a number of applications written using Liszt have been presented in [6]. We are not aware of any industrial applications developed using Liszt.

3 The OP2 Library

In this section we give a brief description of the mesh abstraction that is exposed by OP2, and we relate it to its user interface. The reader is invited to refer to [7, 8] for a full description of the interface. A mesh is modelled as a graph and it includes a collection of interconnected sets. In a typical CFD program, the mesh includes the following sets: edges, vertices, and cells. A set is a programming abstraction of the OP2 library (op_set) and it is used to build an iteration space. To declare an op_set, the user is provided with the op_decl_set call, which requires the iteration space cardinality (or size), i.e. the number of elements in the set.

The connectivity between sets expresses the mesh topology, and it specifies how a generic element of a set maps to elements in another set. For instance, the user can specify for each edge what are the incident vertices. This translates in OP2 with the op_map data structure and with a call for declaring it (op_decl_map). The call takes as input: the *from* and *to* sets of the mapping; the arity (or dimension) of the mapping, i.e. the number of elements in the *to* associated to each element in the *from* set (this number must be homogeneous for all mapped elements); an array of indices implementing the mapping.

Data in OP2 is associated to mesh sets. A dataset associates a tuple to each element of a set, and is abstracted in OP2 through the op_dat data structure and declared with the op_decl_dat function. This function takes as input the set to which the op_dat is associated, the cardinality (or dimension) of the tuples (i.e. the number of data items associated to each set element, that must be homogeneous), and the array of tuples. For instance, an op_dat contains the 3D spatial coordinates for each vertex.

Fig. 2: Example of user kernel and OP2 op_par_loop. The first line is an annotation extension which we will describe in Section 4.

```
@op_inc_id(v1Data, v2Data)
void incrVertices (double * eData, double * v1Data, double * v2Data) {
  ...
  *v1Data += t;
  *v2Data += t;
}
op_par_loop (incrVertices, edges,
  op_arg_dat (edgeData, -1, OP_ID, OP_READ),
  op_arg_dat (vertData, 0, edges2Verts, OP_INC),
  op_arg_dat (vertData, 1, edges2Verts, OP_INC));
```

In OP2, computation is expressed through parallel loops, which apply a user-programmed kernels to all elements of a chosen iteration op_set. An example of a user kernel, which reads data associated to an edge and modifies the data associated with the two connected vertices, is illustrated in Figure 2. We also show the related op_par_loop call, expressing the application of the user kernel to all edges. The first two arguments of the op_par_loop are the user kernel and the iteration set. Then, the user is required to specify how datasets are accessed to instantiate actual arguments for the user kernel. For this purpose, the op_par_loop takes as input further arguments (called op_args), one for each parameter of the user kernel. For each op_arg, the user specifies:

– The dataset, or op_dat, from which the actual argument is to be retrieved (first parameter).
– If the dataset is associated with the iteration set (edges in Figure 2), then no indirection is necessary. In this case the second and third parameters assume the values `-1` and `OP_ID`. For a given iteration, the dataset is accessed *directly* using the iteration identifier.
– If the dataset is associated to a set different from the iteration set, then an op_map is needed. The third parameter is a reference to the op_map to be used to translate iteration set identifiers to op_dat associated set identifiers. The second parameter is an integer specifying which mapped element is to be considered. For instance, for the mapping from edges to vertices the user has to specify 0 or 1, to address the first or second vertex associated to each edge.
– The access modality: read (`OP_READ`), write (`OP_WRITE`), increment (`OP_INC`), read and write (`OP_RW`). The user kernel must reflect the access modality expressed for the op_par_loop parameters.

This information is called the access descriptor and it exposes the loop's data-access pattern to the OP2 compiler. It is important to notice that an access descriptor implicitly contains information related to the cardinality of the involved op_dat and the arity of the op_map used (if any). This information is extracted by the compiler by analysing op_dat and op_map declarations, and can be used to compute the memory requirements for a specific iteration of an op_par_loop.

To maximise parallelism for op_par_loops, OP2 assumes that the loop iteration ordering does not influence the final result. Some combinations of access descriptors, i.e. when indirectly modifying a dataset, might incur data races if not properly controlled. The OP2 implementation guarantees data race avoidance when incrementing (`OP_INC`) a dataset accessed indirectly. For all other cases (`OP_WRITE`, `OP_RW`) it is responsibility of the user to express parallelism control by constructing and, if necessary, partitioning the mesh to ensure no conflicts exist.

3.1 Compiler and Run-Time Support

The current implementation of OP2 includes: a source-to-source translator, that maps a program using OP2 to multiple target languages, such as CUDA, OpenMP, OpenCL and MPI; a run-time library which performs standard unstructured mesh optimizations, such as mesh partitioning and coloring (see below). We give a description of the CUDA implementation of OP2.

For GPUs, the size of the mesh is constrained to be small enough to fit entirely within the GPU's device memory. This means that for non-distributed memory implementations (i.e. single node back-ends) data transfer only happens at the time of data declaration, and when collecting results at the end of the computation. For CUDA, the compiler parallelizes an op_par_loop by partitioning its iteration set and assigning each partition to a Streaming Multiprocessor (SM)[6]. In this section we discuss two main features of the implementation: coalescing memory accesses and a coloring strategy to prevent data races. The implementation distinguishes between op_par_loops that use at least one op_map, called indirect loops, and those that do not use indirections, called direct loops.

For direct op_par_loops, we partition the iteration set in chunks of the same size, and each thread in a CUDA thread block works on at most $\lceil \frac{n}{m} \rceil$ elements of the partition, where m and n are the sizes of the thread block and partition, respectively. Observe that this execution model is sufficient to avoid data races because, by definition, none of the data is accessed indirectly and therefore each thread can only update data belonging to its iteration set elements. The main concern is to avoid non-coalesced accesses into device memory. This is achieved by staging data between device memory and the shared memory, in two stages. (1) Before the user kernel executes, any dataset read whose cardinality per set element exceeds one is brought into the shared memory. The rationale behind this is that unary data will be accessed during execution through a naturally coalesced transfer. (2) After the user kernel is complete, any modified dataset is moved back from shared into device memory.

For indirect op_par_loops, we chose a different strategy, where we distinguish between op_dats accessed directly or indirectly. Indirectly accessed op_dats are staged between device and shared memory. The data can be scattered in the device memory because of mappings, even if proper renumbering algorithms are used to minimise the dispersion of data and to build clusters of coalesced data. For contiguos regions, memory accesses are coalesced. The stage in phase coalesces device memory data into shared memory locations mapping successive memory addresses into successive thread identifiers. Directly accessed op_dats are instead left in device memory. This reduces the shared memory requirements for the CUDA kernel and relies on the L1 cache.

Additionally to memory access coalescing, for indirect op_par_loops gaining good performance is somewhat restricted by the need to avoid data races between threads. That is, allowing threads to operate on distinct elements of the iteration set does not guarantee an absence of data dependencies due to indirect accesses, as previously discussed. The implementation is based on *coloring* the iteration set in an inter- and intra-partition fashion to resolve this issue. The inter-partition coloring is used to avoid conflicts between the data shared at partition boundaries. Since the library ensures partitions with the same color do not share elements retrieved through a mapping, these can proceed in parallel. Intra-partition coloring is needed to prevent threads in the same thread block from data race conflicts. In OP2 increments are computed in a fully-parallel way by threads in the same block using local private thread variables. Colors are followed when applying the increments to the shared memory variables, to prevent conflicts.

4 Optimizations

An application programmer writing an OP2 loop is insulated from the details of the implementation on the back-end architectures which OP2 supports. As such, there is no restriction on how many sets, maps and datasets are used in the loop, their size or access pattern. Thus, given specific back-end hardware, the OP2 code transformation framework needs to take into consideration not only how an op_par_loop can be optimized, but also the limitations of the underlying hardware that degrade performance. This is a key issue that we encountered when utilizing OP2 for accelerating HYDRA.

We consider an example loop of HYDRA, called EDGECON, which is representative of the key loops that make up over 90% of the runtime in HYDRA on a GPU. EDGECON computes the gradient contribution on edges, by iterating over edges accessing datasets associated to both edges and vertices (using a mapping from edges to vertices). This scheme is common in CFD code, and its pattern is shown in Figure 2. The input of the loop includes both indirectly and directly accessed op_dats. Each iteration of the loop accesses directly 24 bytes (1 op_dat), and indirectly a total of 544 bytes (10 op_dats). Of these, two op_dats are accessed indirectly and incremented, and their total size is 384 bytes per iteration. As these incremented op_dats are allocated to shared memory and local thread variables, they represent a main source of shared memory and register pressure. The elemental user kernel used by EDGECON is made of 102 double precision floating point operations, and about 200 integer operations. The PGI compiler reports the use of 41 registers per thread, which is larger than the 31 available for double precision on the NVIDIA C2070 GPU. The largest iteration size available for execution is 64, which requires 34KB of shared memory (a Fermi GPU supports 48KB).

From this analysis it can be noted that the loop suffers from two main issues when mapped to a GPU. Firstly, as the partition size is small, the available parallelism within each SM is limited. To improve this, shared-memory requirements need to be reduced. For instance, to employ partitions of 128 iterations, we need to fit all indirectly accessed arguments into shared memory. When the iterations in a same partition do not share any data (i.e. in the worst case), this requires a partition with 64 iterations to use no more than 24KB in shared memory, as the shared memory requirements roughly double with the partition size. Conversely, an effect of high shared memory requirements is a poor CUDA block occupancy. Secondly, the registers required for each iteration are more than the maximum available on a Fermi GPU. This hardware resource shortage prevents the dynamic SM scheduler from allocating all 32 threads per warp.

4.1 Compiler Support for Loop Fission

Loop fission is an effective means to address the high register pressure and high shared memory requirements exhibited in the EDGECON loop. Splitting a loop manually requires the developer to analyze the kernel for valid splitting points and to explicitly refactor the kernel to pass data across sub-kernels. This task is tedious and error-prone.

As discussed, the EDGECON loop follows a widely used loop scheme in unstructured mesh CFD. It iterates over edges of the mesh (line 9) and increments two arguments through an indirection (lines 11 and 12). The user kernel typically computes a unique contribution value on a local function variable (*i.e.,* `t`), which is then used to apply the increment through the indirection to the two vertices of the edge (lines 5 and 6). This scheme is widely used in HYDRA by all performance-critical loops, which together account for the 90% of execution time. It also exists in a few variants: for example, one vertex data value is incremented while the other is unchanged or decremented.

Because this scheme is critical and opens up a natural splitting point between the contribution computation and the *dispatch* to the vertices, we extended the OP2 abstractions with annotations to categorize kernels. Using these, developers can drive the OP2 compiler as depicted in Figure 2, line 1. We extended the OP2 compiler to leverage annotated code with automatic OP2 to OP2 loop splitting. Our transformation replaces the original loop with three loops, depicted in Figure 3, with equivalent semantics:

- The first loop computes the contributions for each edge and stores them into a new op_dat associated with the edges (lines 14-16). The kernel code is obtained from the original kernel by moving the argument corresponding to the second vertex data to a local variable (line 3). In doing so, the second increment no longer has any effect and can be safely eliminated by the compiler.
- The second and third loops iterate over edges and apply the increment, passed in the new op_dat, to the vertices (lines 17-22). The corresponding kernel code (lines 9-12) is generated according to the kernel annotation and the types of the vertex data.

Fig. 3: Source to source loop fission of the example shown in Figure 2.

```
void incrVerticesAlt (double * eData, double * v1Data) {
  double v2Data [1];
  ...
  *v1Data += t;
  *v2Data += t;
}
void incrVerticesCpy (double * e, double * v) {
  *v += *eData;
}
op_par_loop ( incrVerticesAlt, edges,
  op_arg_dat ( edgeData, -1, OP_ID, OP_READ),
  op_arg_dat ( incrVerticesTemp, -1, OP_ID, OP_WRITE));
op_par_loop ( incrVerticesCpy, edges,
  op_arg_dat ( incrVerticesTemp, -1, OP_ID, OP_READ),
  op_arg_dat ( vertData, 0, edges2Verts, OP_INC));
op_par_loop ( incrVerticesCpy, edges,
  op_arg_dat ( incrVerticesTemp, -1, OP_ID, OP_READ),
  op_arg_dat ( vertData, 1, edges2Verts, OP_INC));
```

This distribution of one loop into three allows the number of op_args to be reduced for each loop w.r.t. the original loop. For the first loop, it also transforms two indirect op_arg accesses into a single directly accessed op_arg. As directly accessed op_args for indirect loops are not staged into shared memory, this reduces the shared memory requirements for the first loop. The increment loops are generally simpler loops, with a smaller number of input parameters and small kernels, and can be thus easily accelerated.

4.2 Alternative Coloring Schemes

In Section 3 we showed that OP2 guarantees absence of data races by using a two-level coloring technique. The optimization presented here provides an alternative strategy for intra-partition coloring. In the current OP2 implemention the following scheme is applied. First, the user kernel is evaluated for all iterations in the partition. The parallelism degree for this stage is the minimum between the number of iterations in the partition, and the number of threads in the CUDA block. In this phase, the increment contributions are not applied directly to the op_dat in shared memory, to prevent data races between threads, but local *private* thread variables are used to store the increments. There is one such variable for each thread and for each incremented op_dat. After executing the user kernel, the increments are applied to the shared memory by following colors. This strategy maximizes the parallelism when evaluating the user kernel, at the cost of a larger register pressure due to the additional thread private variables.

An alternative to this is to program threads to compute and apply the increments to shared memory when executing the user kernel. The user kernel is passed a shared memory address, instead of private thread variable references.

To prevent data races while executing the user kernel, the thread execution must follow colors. This reduces the amount of total parallelism when evaluating the kernel, but it also reduces the register requirements due to the eliminated private variables. The implementation of this alternative strategy is confined to the CUDA kernels synthesized for OP2 loops. This behavior can be selected by the user through annotations on the loop, similar to the ones used for fission. However, this is a short term solution, and we aim at understanding when this alternative synthesis actually delivers better performance. In Section 5 we discuss how this can be deduced by combining information from access descriptors and CUDA compiler reports.

4.3 Tuning Partition and Thread Block Size

The final optimization that we extensively applied to HYDRA loops is the tuning of the partition and thread block size. These two parameters are inter-dependent: the partition size is the number of iterations that are mapped to the same SM, while the thread block size is the number of threads that are used in the CUDA program to execute the iterations.

Both the partition and thread block size represent an upper bound on the amount of parallelism that can be achieved by a SM when executing a partition. The execution consists in the following phases, as discussed in Section 3: *(i)* stage in of input data from device to shared memory, one dataset at a time; *(ii)* execution of the user kernel; *(iii)* stage out from shared to device memory, one dataset at a time. When executing the user kernel the maximum parallelism achievable is equal to the number of iterations in the partition; in the staging phases the parallelism is instead limited by the number of elements to be staged, multiplied by the dataset cardinality. With no data re-use, this is equal to the partition size multiplied by the cardinality of the dataset. Using a larger CUDA thread block size permits more parallelism in the staging phases, without losing the coalescing property. As a general rule, a larger partition size, constrained by the shared memory size, is always preferred to provide more parallelism to the SM. However, the optimal block size depends on the size of the op_dats and the kind of access. Section 5 studies the relation between multiple loops with different access descriptors and the optimal block size.

5 Experiments

In this section we show the results of performance analysis of the optimizations described in the previous section applied to several loops in HYDRA. The simulation used is a standard CFD test problem used for validation of correctness and performance called NASA Rotor 37, that models a blade of a turbomachinery component. In our tests we replicate the blade twice to set up a mesh size including: 2.5M edges, 860K vertices, and 54K wall edges. The simulation solves the Navier-Stokes equation to evaluate the flow through one passage of the two NASA Rotor 37 blades and it is the same application used in the performance graph of Section 1. In previous tests, we also used a larger mesh, including 4 NASA Rotor 37 blades, and we obtained similar performance results of the case used here, scaled by a factor of 2.

The configuration of HYDRA required for this simulation uses 33 op_par_loops some of which are extremely complex. In our tests, we used an NVIDIA Fermi C2070 including 14 SMs (i.e. 448 CUDA cores), 6 GB of main memory and 48/16 KB of shared memory and L1 cache, respectively. The full simulation runs for tens of minutes, but we limited performance analysis to 30 time steps to reduce the total performance analysis time. To compile the CUDA code generated by the OP2 compiler we used the PGI Fortran CUDA compiler version 12.2, with CUDA 4.0, and the NVIDIA CUDA compiler version 4.0.17. The optimization options are, respectively, -O4 and -O3. The experiments focus on the effects of optimizations on all loops involved in the simulation. For each loop, there are a limited number of optimizations that can be applied, and that can be composed together. We analyze the performance of each loop when applying the optimizations individually, and in all their possible compositions. The aim of these experiments is to put into relation: (1) The features of an op_par_loop in terms of its input arguments including: the type and cardinality (or dimension) of the related op_dat ; the arity (or dimension) of the related op_map, if used. (2) The GPU resource requirements, in terms of the number of registers needed for each thread and the shared memory size required given a specific partition size. (3) The performance in terms of execution time for the CUDA kernel section.

5.1 Fission, Block Tuning and Coloring

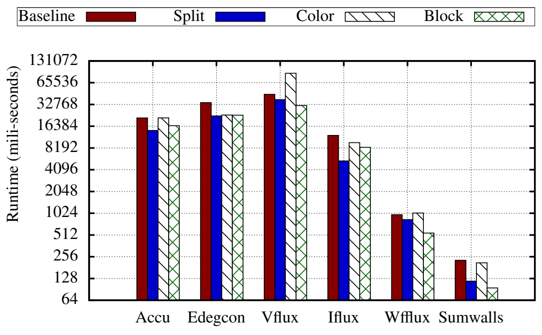
A number of OP2 loops in NASA Rotor 37 can be subject to all the three optimizations discussed in the previous section. For space reasons, we study the following relevant loops: accumulation of contributions on edges (ACCU), gradient contribution on edges (EDGECON), viscous flux calculation (VFLUX), inviscid flux calculation (IFLUX), viscous wall flux calculation (WFFLUX), and summation of near wall edge contributions (SUMWALLS). The first five loops adhere to the requirements of the fission optimization, by incrementing equivalent amounts to two op_dats. Unlike the previous loops, the last one has a single incremented op_dat. We used this case as an experiment to understand if loop fission increases performance. This explores the general idea that smaller kernels are always better parallelized on a GPU than larger ones.

	ACCU	EDGECON	VFLUX	IFLUX	WFFLUX	SUMWALLS
Iteration set	edges	edges	edges	edges	wall edges	wall edges
No. of op_arg_dats	13	11	19	9	15	8
No. of indirect op_arg_dats	12	10	18	8	12	5
Size of op_arg_dats (bytes)	712	568	776	296	628	228
Size of increments (bytes)	200	288	96	96	96	48

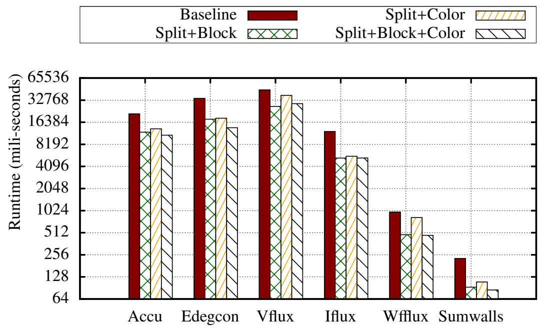
Table 1: Loop properties resulting from access descriptor analysis for loops which can be subject to fission and alternative coloring.

Table 1 illustrates the main features of the loops, by inspecting their access descriptors. All loops feature an average to large number of input op_dats, each with a large cardinality, resulting in a large amount of shared memory required to execute each iteration. The first four loops iterate over the largest mesh set (edges), while the last two iterate on the wall edge set that is two orders of magnitudes smaller. This is reflected in the average performance of the loops, as we detail below. The size of the input data for each iteration can be used to define the maximum permitted partition size that a loop can use. As a partition is mapped to a single streaming multiprocessor (SM), all iteration data for that partition must fit into shared memory, i.e. into 48KB on the C2070 GPU. The run-time profiling of OP2, which analyses the mesh, computes the average data re-use, and with these results, the kernel configuration can be tuned to maximize the partition size.

Fig. 4: Performance results when applying optimizations alone and in composition. The Y-axis is in log-2 scale.



(a) Single



(b) Compound

Figure 4 shows the results of applying each optimization to the described loops. Table 2 shows the resource requirements for each loop when applying different optimization schemes. For each optimization, we always choose the maximum partition size achievable or the one delivering better performance. For all cases, except the block tuning optimizations, the CUDA thread block size is equal to the partition size: this assigns one thread per iteration.

The analysis of the results shows:

- Splitting a loop reduces both shared memory and register pressure, and should thus be applied extensively. In some cases, it also permits larger partition sizes to be achieved, thus improving the available parallelism.
- For split loops, the alternative coloring strategy delivers slightly better performance in nearly all cases. This is related to a reduction in the average number of colors for split loops. If applied to original loops, this strategy can deliver significantly worse performance, when associated with a larger number of intra-partition colors. Thus, it should only be used in tandem with splitting.
- Block and partition tuning improves the performance for all loops, both split and original ones, and should be applied extensively.

As highlighted, the alternative coloring strategy does not necessarily reduce register usage, but it sometimes increases it slightly. This is somewhat unexpected, and we believe that it is related to the way in which the low-level CUDA compiler treats different control-flow organizations.

As expected, loop fission improves performance by a large factor, even when the user kernel includes a relatively small number of floating point operations. Also, the choice of the alternative colouring strategy should be taken when register requirements are actually reduced. We can do this by synthesizing the two versions at OP2 compile-time, with and without alternative coloring strategy, and by choosing the best one by looking at the register requirements for the two kernels as reported by the low-level compiler.

5.2 Tuning Partition and Block Size

The final optimisation involves the tuning of seven loops of NASA Rotor 37. These loops are generically small, in terms of number of input op_dats and kernel size, and their contribution to the total performance is much lower than the six loops discussed above. However, our goal is to understand what is the best configuration of these two parameters. Table 3 shows the results, including the configuration parameter values and the obtained performance. In the table, we can notice that the first four loops obtain higher performance with the largest achievable partition and block sizes (512), while the remaining three loops perform better with a lower value (128). This can be explained by analysing the access descriptors. All loops take as input a number of op_dats between 4 and

(a) Register Usage

Loop	Baseline, Block	Color	Split (+block)	Split + Color (+block)
ACCU	63	63	63, 28, 28	63, 18, 23
EDGEC.	41	46	37,32, 32	37, 34, 34
VFLUX	63	63	63, 27, 31	63, 29, 34
IFLUX	63	63	63, 28, 29	63, 29, 32
WFFLUX	63	63	63, 28, 28	63, 18, 18
SUMW.	37	41	34, 28	34, 15

(b) Shared Memory Usage

Loop	Baseline, Color, Block	Split (all compounds)
ACCU	43	36, 25, 25
EDGEC.	34	20, 24, 24
VFLUX	47	41, 6, 6
IFLUX	34	22, 6, 6
WFFLUX	36	30, 12, 12
SUMW.	24	18, 6

Table 2: Resource usage for OP2 optimizations. In 'Split' columns there are 3 values as each loop is split into 3 loops.

6, but only the first four loops have all input data accessed through indirection. The remaining three loops only access a single input through an indirection, while the remaining op_dats are accessed directly.

As described in Section 3, indirect op_dats are staged into shared memory for indirect loops, while directly accessed data is left in device memory. The block size parameters strongly influences the staging performance. As the first four loops spend more time in staging data than the remaining three loops, the block size increase plays a dominant role in the performance of the loops. Also, the number of data items to be staged directly depends on the number of data values per mesh element and the partition size. The first four loops have either a larger partition size, or input op_dats with larger dimension, and can thus benefit of larger block sizes.

5.3 Discussion

The analysis of the performance results shown in this section led us to the following conclusion which can be adopted as a general optimization strategy in the compiler:

- A main source of performance degradation on GPUs for OP2 loops are small partition sizes. This is a consequence of having a large number of op_dats which sum up to a large number of input data for each iteration, resulting in larger shared memory requirements. This condition — having larger shared memory requirements — can be checked at compile-time by inspecting the access descriptors. The compiler addresses this issue by splitting the loops which have been annotated by the user.
- When a loop is split, the resulting loops can be further optimized if the alternative coloring strategy actually reduces the number of registers needed per thread. This can be achieved at compile-time by first generating two versions, each using a different coloring strategy, and then choosing the best version by feeding the OP2 run-time with register pressure information returned by the CUDA compiler. This removes the burden on the programmer to annotate loops which should be implemented using the alternative coloring strategy.
- Once the partition size is optimized, a loop exposes sufficient parallelism inside its partitions. However, the real parallelism that can be obtained on a GPU depends on the resource constraints of each thread, in terms of register requirements. This requirement directly influences the warp allocation strategy for the SM dynamic scheduler: if each thread requires a large number of registers, then a smaller number of threads can be allocated in the same warp. This condition must be checked also for loops with relatively small

	$Loop_1$	$Loop_2$	$Loop_3$	$Loop_4$	$Loop_5$	$Loop_6$	$Loop_7$
Part. and Block size	(64,64)	(64,64)	(64,64)	(64,64)	(64,64)	(64,64)	(64,64)
Perf. (millisec.)	15.70	41.94	19.58	35.30	9.53	6.13	7.46
Part. and Block size	(64,512)	(128,512)	(256,512)	(256,512)	(128,128)	(128,128)	(128,128)
Perf. (millisec.)	11.52	14.67	8.66	18.99	8.68	5.44	6.64

Table 3: Partition and block tuning for seven indirect loops.

input op_dats, but with high register pressure. For this kind of loop, splitting and the alternative coloring strategy can be applied to reduce register pressure.

6 Conclusion

In this paper we have demonstrated and evaluated the effect of applying three optimizations for unstructured mesh programs to a wide number of HYDRA loops. The optimizations: (1) permit transforming the input OP2 program to optimize shared memory requirements; (2) provide a kernel-tailored code synthesis minimizing register requirements; (3) tune configuration parameters to optimize data staging for each loop. We have shown how these three optimizations can be automatically implemented by the compiler by extending the OP2 language with loop annotations. This reduces significantly the compiler complexity, as it does not need analysing the user kernels associated to each loop. Finally, in the experiment section we presented a full performance analysis showing the optimization effects on the performance of the loops, and on their resource requirements. This enabled us to derive a general optimization strategy for the complier, based on the composition of the described optimizations.

References

1. Han, H., Tseng, C.W.: A comparison of locality transformations for irregular codes. In: 5th Intnl' Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers, Springer (2000) 70–84
2. Burgess, D.A., Giles, M.B.: Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. Adv. Eng. Softw. **28**(3) (April 1997) 189–201
3. Chevalier, C., Pellegrini, F.: PT-Scotch: A tool for efficient parallel graph ordering. Parallel Comput. **34**(6-8) (July 2008) 318–331
4. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J. Sci. Comput. **20**(1) (December 1998) 359–392
5. Strout, M.M., Carter, L., Ferrante, J.: Compile-time composition of run-time data and iteration reorderings. In: Procs. of the PLDI '03. (June 2003)
6. DeVito, Z., Joubert, N., Palacios, F., Oakley, S., Medina, M., Barrientos, M., Elsen, E., Ham, F., Aiken, A., Duraisamy, K., Darve, E., Alonso, J., Hanrahan, P.: Liszt: a domain specific language for building portable mesh-based PDE solvers. In: Procs. SC'11, New York, NY, USA, ACM (2011) 9:1–9:12
7. Giles, M.B.: OP2 User's Manual (April 2012) http://people.maths.ox.ac.uk/gilesm/op2/user.pdf.
8. Giles, M.B., Mudalige, G.R., Sharif, Z., Markall, G., Kelly, P.H.: Performance analysis and optimization of the OP2 framework on many-core architectures. The Computer Journal **55**(2) (2012) 168–180