Effective Resource-Driven Loop Splitting for Large Unstructured Mesh Applications on GPUs

R. Tohid
Louisiana State University

Fabio Luporini
Imperial College, London

Carlo Bertolli
IBM Research

Istvan Reguly
Gihan Mudalige
Oxford e-Research Centre

Adam Betts
Florian Rathgeber
Graham Markall
David A. Ham
Imperial College London

J. Ramanujam
Louisiana State University

Michael B. Giles
University of Oxford

Paul H.J. Kelly
Imperial College, London

Abstract

Unstructured mesh applications are widely used in science and industry for simulating phenomena as diverse as turbomachinery components of jet engines and blood flow in arteries. These are examples of irregular applications that are difficult to optimize for accelerator targets such as GPUs. Splitting loops is a standard technique used for optimizing GPU applications. It breaks down large complex parallel loops into smaller units whose performance is improved, due to reduced shared memory and register requirements. In this paper we introduce a general loop splitting methodology for unstructured meshes, which is able to split a complex loop into multiple simpler loops. A given loop can be split in different ways, depending on the loop features and the target GPU hardware. Unlike previous contributions, the introduced technique permits synthesizing alternative implementation strategies, without the need of transforming the input program. Experiments on a series of complex loops from an industrial CFD code show the efficacy of our solution both for NVidia Fermi GPUs and Intel multicore CPUs. The results show that the version obtained after loop splitting always performs better on a GPU compared to the original version. The opposite result is instead obtained for the CPU, as the original unsplit version performs better when using large numbers of threads.

1. Introduction

Computational Fluid Dynamics (CFD) applications using unstructured meshes dominate the workload of many industrial and academic HPC systems. Examples include the modeling of blood flow in the human body, air flow over aircraft and ocean circulation. The use of unstructured meshes is often essential to correctly simulate complex geometries in CFD and they are widely used in those cases in which structured meshes are unable to provide a suitable modeling abstraction.

Despite their attractiveness from a simulation viewpoint, unstructured mesh applications represent a "hard" case in terms of realizing computing performance. This is a result of the extensive use of pointers between mesh elements (e.g., edges to vertices) used to express the mesh structure, which renders data layout and data movement a complex problem. This performance issue has been the subject of a number of research works [3, 5, 6, 8, 9].

In this paper we focus on optimizing the performance of unstructured mesh applications on GPUs. Unlike previous papers addressing this issue (see [6, 8, 9]), we focus on CFD codes that access large amounts of data, in which GPU performance is hard to achieve and CPUs typically deliver better results. This derives from two well-known limitations of current GPU technology, namely, the limited size of shared memory and the small number of registers available to each GPU thread.

Existing implementations of unstructured mesh applications introduce several optimizations for GPUs (such as those from NVIDIA) [6, 8, 9]. One effective optimization improves data locality by mapping all or part of the mesh data onto shared memory. This exploits temporal locality; for instance, when iterating over edges and accessing vertex data stored in shared memory, two linked edges will access the same vertex. It also improves spatial locality as data scattered in global memory due to the presence of pointers (e.g., between edges and vertices) is stored in a contiguous chunks of shared memory. Other optimizations aim to improve global memory access coalescing, by renumbering the mesh using standard software (e.g., METIS [11]). Both optimizations are included in the leading stencil code generation tools for unstructured meshes, namely LISZT [6] and OP2 [8].

Despite the effectiveness of these "standard" GPU optimizations on many small applications and benchmarks (e.g., those in [6, 8]), there are cases in which this is not sufficient to achieve good performance. An example is industry-strength CFD simulations that have large loops where each iteration needs to access a large amount of data. The limitation in size of the shared memory on current GPU technology forces one to partition the iteration set into small chunks, whose required data can fit into shared memory. However, this significantly reduces the performance of a GPU, since the amount of available parallelism is small and is extremely

HiStencils 2015
Second International Workshop on High-Performance Stencil Computations
January 20, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://www.exastencils.org/histencils/2015/

difficult to overlap global memory accesses with computation using typical GPU optimization techniques [3].

To optimize the performance of large loops in unstructured mesh computations on GPUs, we introduce a general loop splitting technique. We consider the case of parallel loops over the mesh applying some user-defined kernel (typically called *user kernel*). Using this technique we can synthesize an implementation of a large parallel loop in which the user kernel is split into multiple functions (sub-kernels) of equivalent semantics. Each sub-kernel is carefully outlined from the original user kernel in such a way that its shared memory requirement is smaller than the original user kernel. Using this knowledge, the implementation only stages into shared memory the strictly necessary data for executing each sub-kernel. This permits maximizing the partition size of the whole loop.

A fully general approach to loop splitting (often called loop fission) would be to try to find an optimum cut of the user kernel's dataflow graph. This paper focuses on a simpler but very effective technique which applies to a very common class of loops over the edges or cells of an unstructured mesh—where the user kernel computes values which are then "pushed," via indirections, to increment data associated with vertices.

We validate the effectiveness of our approach by studying multiple loops derived from an industrial CFD application for simulating the air flow in turbomachinery components of jet engines. We report on the performance of loop splitting when executing on a GPU and a CPU. The aim is to understand the possible benefits of using this approach also when dealing with architectures with larger caches. In addition, we present a discussion of the possible impacts of the presented technique when using clusters of GPUs and CPUs. In our experiments we use the software developed by the OP2 project as its source-to-source translator is based on ROSE[1], which provides automatic code outlining that can be used to split the user kernel into multiple functions.

The main contributions of this paper are as follows.

1. We present an approach to splitting a loop into multiple loops each with smaller data requirements. From a single complex loop we can derive alternative implementations (through code synthesis) by splitting in different ways to achieve optimal performance. The technique does not require modification of the user code.
2. We show experimental results of loop splitting on a complex industrial application on two architectures: an NVIDIA C2070 GPU using CUDA and two Intel Westmere X5660 multicores with 12 processors using OpenMP. Our aim is to show the impact of loop splitting on architectures with larger caches.

The rest of this paper is organized as follows. We discuss related work in Section 2 and the OP2 implementation on GPUs in Section 3. Loop splitting is discussed in Section 4. Experimental results are presented in Section 5 and we conclude with a summary in Section 6.

2. Related Work

While a number of works on GPU acceleration have focused on structured-mesh problems [4, 10, 12, 14, 15], there has been relatively little work on unstructured mesh codes.

Two major projects have targeted unstructured mesh applications on diverse target parallel architectures. Liszt [6] is an embedded domain-specific language from Stanford for the solution of unstructured mesh based partial differential equations (PDEs). A Liszt description is translated to an intermediate representation which is then compiled by a dedicated compiler to generate native code for multiple platforms. The aim is to exploit information about the structure of data and the nature of the algorithms in the code and to apply aggressive and platform specific optimizations. Performance results from a range of systems (GPU, multicore CPU and MPI-based cluster) executing a number of applications written using Liszt have been presented in [6].

The OP2 project [8] provides a library interface for C/C++ and FORTRAN, and it includes the following: declaration of an unstructured mesh in terms of sets (e.g., vertices, edges) and pointers (or maps) between sets (e.g., how edges are linked to vertices); computation over the unstructured mesh, expressed in terms of visit of all elements of a set applying some user-defined kernel. OP2 is an active library that is translated into several languages, including CUDA [9], OpenMP, OpenCL, MPI and their combinations like MPI+CUDA. A restricted version of loop splitting for OP2 loops was presented in a previous paper [3]. In contrast, in this paper, we present a generalized loop splitting technique. We use OP2 in this paper because: (i) we can take advantage of its ROSE compiler infrastructure which already supports source code outlining [1]; and (ii) we can easily compare our results to previous attempts at optimizing OP2.

Loop splitting has been used for structured grids and other computations, amenable to compile-time analysis. But, to the best of our knowledge, this is the first work to present loop splitting for unstructured meshes that are examples of irregular computations. Exploiting locality is the key issue for unstructured meshes and requires an effective runtime solution for achieving loop splitting, which is what have proposed.

3. The OP2 Implementation on GPUs

In this section, we give a brief description of the relevant OP2 interface and implementation, including all the necessary details required for understanding the contributions of this paper. The interested reader can refer to the user manual [7] and other references giving a full description of the implementation [8].

The OP2 approach to the solution of unstructured mesh problems involves breaking down the problem into four distinct parts: (1) sets, (2) data on sets, (3) connectivity (or *mapping*) between the sets and (4) operations over sets. This leads to an API through which any mesh or graph can be completely and abstractly defined. Depending on the application, a set can represent nodes, edges, polygonal faces or other elements. Associated with these sets are data (e.g., node coordinates, edge weights, etc.) and mappings between sets that define how elements of one set connect with the elements of another set. In this paper, we omit a full description of the OP2 interface for space reasons. Instead, we focus on the parallel loop implementation.

The use of an *active library* provides application programmers with the ability to express complex abstractions through an API, analogous to classical software libraries, but with the benefit of compiler support to optimize those abstractions accordingly. An application written once using the API, which is hosted in C/C++ or FORTRAN, can be translated using the source-to-source compiler tools provided to deliver performance portability across a diverse range of multicore and many-core architectures, including CUDA, MPI, OpenMP, and, in progress, OpenCL. In this section we briefly discuss the library API, its compile- and run-time infrastructure before presenting the main contributions of this paper.

3.1 Mesh Declarations

We discuss the OP2 programming model by coding up the simple unstructured mesh depicted in Fig. 1 that consists of vertices v_1 through v_6, edges e_1 through e_{10}, and five triangular cells (unnumbered in the figure). The programmer first defines the topology of the mesh by declaring sets and the relations between these sets.

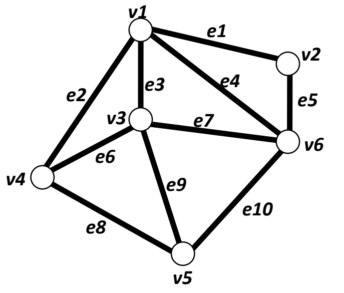

Figure 1. Example mesh

Declaration of these sets using the C API can be detailed as follows[1]:

```
op_set triangles = op_decl_set (5);
op_set vertices = op_decl_set (6);
op_set edges = op_decl_set (10);
```

To relate the vertices to which each edge is incident, the programmer defines an array encoding the relation and informs the library of the sets to which it is applicable:

```
int map [][2] = {{1, 2}, {1, 4}, {1, 3},
  {1, 6}, {2, 6}, {4, 3}, {3, 6}, {4, 5},
  {3, 5}, {6, 5}};
op_map edgesToVertices =
  op_decl_map (edges, vertices, 2, map);
```

Position i in `map` corresponds to edge $\lfloor \frac{i}{2} \rfloor$. The first and second parameters in `op_decl_map` state the source and destination sets, respectively, while the third parameter defines the dimension (or cardinality) of the relation. The next task of the programmer is to associate data to the sets of the mesh over which the parallel computation operates. Assume that each vertex in this example contains two double-precision floats (e.g., representing their coordinates) and that each edge contains one single-precision float (e.g., representing their weight). A dataset is declared on a set through an array of the required size, including an appropriate initializer as follows:

```
double coordinates [6][2] = {...};
op_dat vertexData =
  op_decl_dat (vertices, 2, coordinates);

float weights [10] = {...};
op_dat edgeData =
  op_decl_dat (weights, 1, edges);
```

The second parameter in `op_decl_dat` informs the library of the cardinality of the dataset per element of the set. Generally, the implementation does not operate directly on the passed array, since certain targeted back-end architectures utilize different memory spaces. For example for the CUDA back-end implementation of the `op_decl_dat`, the input data array is transferred from host to device memory, where it will reside for the rest of the computation, unless the user explicitly requires a copy back to the host through a call to the `op_get_dat` function. This means that there is no data transfer between host and device, or vice-versa, during the computation.

In effect, the data of a program is *sliced* into two subsets: that which belongs to the library (the mesh datasets) and that which belongs to the sequential program segments. To initialize an `op_dat`, the user first initializes a user-level array, i.e., an array outside the library logical space. Then, the `op_decl_dat` transfers the array data to library logical space, where it is initialized. This means that the user is now no longer able to modify the data encapsulated in an `op_dat` without using a library call.

[1] For clarity of exposition, we omit certain non-essential parameters in all the API calls. Details of the API are documented elsewhere [7]. Also, *pml* in the library calls stands for "parallel mesh library" which is used in this version for double-blind review.

3.2 Parallel Loops in OP2

All the numerically intensive computations in the unstructured mesh application can be described as operations over sets. Within an application code, this corresponds to loops over a given set, accessing data through the mappings (i.e., one level of indirection), performing some calculations, then writing back (possibly through the mappings) to the data arrays. The OP2 API provides a parallel loop declaration syntax (op_par_loop) which allows the user to declare the computation over sets in these loops. The computation per element is defined by the user by providing *kernel* functions—written in regular C/C++ or Fortran syntax—which operate on a single element (i.e., they are *element-wise*) of a given set, the so-called loop *iteration set*.

Let us suppose that, in our running example, we wish to increment the coordinates of each vertex by the weight of each edge incident to it, and also calculate the maximum weight across all edges. Below are functions implementing this functionality per vertex:

```
void update (double coordinates [],
       double coordinates [],
       float weights []){
          coordinates[0] += weights[0];
          coordinates[1] += weights[0];
}
void maxWeight (float *weights, double *max){
  if ( *weights > *max ) *max = *weights;
}
```

The op_par_loop declaration is then used to specify the data access pattern for the computation where parallelization is achieved:

```
op_parallel_loop (update, edges,
    op_arg_dat(coordinates, 0, map, OP_INC),
    op_arg_dat(coordinates, 1, map, OP_INC),
    op_arg_dat(weights, -1, OP_ID, OP_READ));

op_parallel_loop (maxWeight, edges,
    op_arg_dat(weights, -1, OP_ID, OP_READ),
    op_arg_gbl(&max, OP_MAX));
```

Each parallel loop call takes $n+2$ arguments: the first is a pointer to the user kernel function; the second is the relevant iteration set; there are n remaining arguments where n is the number of parameters in the user kernel and argument i corresponds to parameter $i-2$. Furthermore, each argument either relates to a mesh dataset (`op_arg_dat`) or to a reduction (`op_arg_gbl`).

For a dataset argument, the parameters convey the following information to the library:

- Access modality of the corresponding parameter. There are four cases: read-only `OP_READ`, write-only `OP_WRITE`, both read and write `OP_RW`, or increment `OP_INC`. We indicate `coordinates` to be incremented inside `update` through `OP_INC`, whereas `weights` is a read-only variable and its access is described through `OP_READ`.
- Whether the data is accessed via an indirection. Either the data is *directly* linked to the iteration set, i.e., it is associated to the iteration set used by the loop, or it is *indirectly* accessed, in which case a mapping must be supplied. For instance, in the first parallel loop above, `coordinates` is an indirect dataset since it is attached to `vertices`, but the mapping `map` determines how vertices can be retrieved while sweeping through edges. On the other hand, `weights` is directly accessed and the mapping is the identify function, indicated by `OP_ID`. When all datasets

are directly accessed, the parallel loop is denoted as *direct*, otherwise it is *indirect*. Thus, the first parallel loop is indirect whereas the second parallel loop is direct.

- If the argument is indirectly accessed, which element of the relation should be utilized. For instance, since there are two vertices per edge, the value 0 in the first op_arg_dat specifies the first vertex. When there is no mapping, a sentinel value of -1 is used, as for the op_arg_dat in the second parallel loop.

3.3 Implementation of Parallel Loops

An application written using the library API is parsed through its compiler and will produce back-end specific code specializing the implementation of each parallel loop present in the input program. It will also modify the input program to call the specialized implementations. We are here interested in the GPU implementation produced by the OP2 compiler, through CUDA. We modify this implementation in Section 4. The result of this first compilation phase is then compiled again using a CUDA compiler (e.g., nvcc or the PGI FORTRAN CUDA compiler) and linked against platform specific back-end libraries to generate the final executable.

For GPUs, the size of the mesh (i.e., of its datasets and maps) is constrained to fit entirely within the GPU's global memory. This means that for the non-distributed memory implementations (i.e., single node back-ends) that we consider in this paper, the only data exchange between the GPU and the host CPU is for the initial transfer of data to the GPU and the final results out to the host CPU. No implicit data transfers are issued by the implementation.

The compiler parallelizes an op_par_loop by partitioning its iteration set and assigning a thread block to each partition. In unstructured mesh applications the meshes to which a program is applied are not available at compile-time. The OP2 compiler produces code that: (i) inspects the actual mesh at run-time, to produce execution information; (ii) executes the parallel loop using the produced information. In the next subsections we give details on these two phases.

How the partitioned data are managed and the execution proceeds depend on whether the parallel loop is direct or indirect.

For direct parallel loops the iteration set is divided in partitions of equal size. Each thread in a thread block works on at most $\lceil \frac{n}{m} \rceil$ elements of the assigned partition, where m and n are the sizes of the thread block and partition, respectively. That is, when the partition size exceeds the number of threads in the block executing it, the threads operate in successive steps to cover the entire partition.

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. Thus, it is possible to instantiate a number of thread blocks equal to the number of partitions. This is obviously limited by the maximum number of blocks that can be launched in parallel on a GPU.

3.4 Partitioning and Scheduling for Indirect Loops

Also for indirect loops the partitions in which the iteration set is partitioned are of homogeneous size. Gaining good performance is in this case 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. OP2 supports data race absence only in the case when an indirectly accessed data item is incremented. For instance, when iterating over edges, two threads that are assigned two different edges, but which are connected by a same vertex, might incur in a data race when incrementing the data associated to the same vertex.

OP2 performs coloring to prevent data races. There are two levels of coloring in its implementation: inter- and intra-mini-partition. The inter-partition coloring is used to avoid conflicts between the data shared at partition boundaries. This can happen if two iterations assigned to different partitions increment a same data accessed through a map. 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 incurring data race conflicts. Again, two threads assigned to the same partition could otherwise increment the same data accessed through indirections.

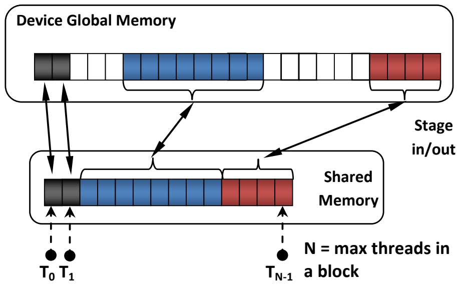

Figure 2. Data staging phase in CUDA kernels. The threads in a block coalesce dataset accesses between device and shared memory.

3.5 Staging Data into Shared Memory

OP2's compiler optimizes for both temporal and spatial locality by staging data between global (device) and shared memory, before and after user kernel execution.

Temporal locality is achieved when the user kernel accesses multiple times the same data allocated into shared memory during a partition execution. Spatial locality is obtained by loading contiguous dataset regions for each partition, which is a consequence of mesh locality itself (i.e., when elements in the same partition are interconnected). Spatial locality also introduces coalescing of device memory accesses, through a proper thread coordination scheme, as detailed below. Data staging includes two phases.

1. Before the user kernel executes, any dataset read whose cardinality per set element exceeds one is brought into the shared memory. This access to the shared memory maximizes parallelism by mapping successive threads identifiers to successive shared memory addresses: a thread identified with i accesses the address base $+ i$, where *base* is the base address of the dataset. This is shown in Figure 2. Unary data accesses in direct loops are already coalesced in global device memory, because the threads in a block access successive coalesced global memory addresses. For this case, datasets are not staged into shared memory. After this first staging phase, all ordered accesses to datasets performed by threads when executing the user kernel are coalesced in shared memory.
2. After the user kernel invocation section is complete, any dataset written is moved back into global memory from the shared memory, so that executions on the next color of mini-partitions, or the next parallel loop, have visibility to the computations. Again, this uses the same coalescing mechanisms discussed in point (1).

An exception to this rule is when datasets are accessed directly in an indirect loop. OP2's implementation avoids staging data with this kind of access, to minimize the requirements of shared memory. Instead, indirectly accessed datasets are always staged into shared memory. Aggressive loop splitting, as described in Section 4, could require a re-thinking of this strategy when splitting a loop into

```
1  loop over colors C //seq
2    loop over partitions P in C //par
3      stage in all data for partition p in P
4      loop over edges e in P //par
5        tmp = contributions (e)
6        update ( e(v1), tmp)
7        update ( e(v2), tmp)
8      stage out all data for partition p in P
9    end loop
10 end loop
```

Figure 3. Original op_par_loop implementation.

many much simpler loops. This should be subject of future investigations.

Data that is incremented through an indirection has a further staging phase, that uses local thread variables and is introduced to maximize the parallelism when executing the user kernel. For each incremented data value, OP2 allocates a local thread array variable whose size corresponds to the dimension of the dataset. Each thread uses its own variables to store the increments to be successively applied to shared memory variables. As these local variables are private to the thread, there is no concurrency control problem, and user kernels can be executed in parallel by threads. Finally, the threads apply the increments stored in the local variables to the appropriate shared memory variables. This step requires concurrency control, as shared memory data is effectively shared between threads. This phase is executed by following a coloring scheme in order to avoid data conflicts.

4. Loop Splitting

The goal of splitting is to reduce the shared memory requirements of the original code on a GPU. This is achieved by splitting the user kernel into multiple successive functions (or stages). Each of such functions must be chosen in such a way that each accesses only a preferably small subset of the input parameters. Consequently, less data needs to be allocated to shared memory for each stage. This results in a smaller overall average shared memory requirement, which permits fitting more parallel loop iterations into the same partition. That is, partitions with larger sizes can be allocated to the same streaming multiprocessor (SM) on a GPU, effectively increasing the parallelism achievable by threads within the same thread block. This improves overlapping of global memory accesses, and specifically targets large CFD loops, as discussed in Section 1.

In this section we show how loop splitting is implemented in OP2 using the pseudo-code of the loop implementation. We start by showing the original OP2 implementation of parallel loops (as described in [8]), and then we incrementally add splitting. First, we consider a simple loop splitting technique that takes advantage of a user kernel property. As discussed, a similar technique was discussed in [3] but that required an OP2 source to OP2 source transformation step; our solution does not require such a transformation step. Next, we explain the general loop splitting implementation, following which we discuss the assumptions and optimality issues.

Figure 3 shows the original implementation of OP2, without splitting, as discussed in Section 3. Note that we have excluded some irrelevant details for the description here. The iteration set is partitioned and partitions are colored to prevent data races. Partitions of the same color are executed in parallel on the GPU (line 2), while partitions of different colors are serialized (line 1). Every partition is scheduled to an SM and all the needed data is staged into shared memory (line 3). The threads execute the iterations of the partition in parallel (line 4) - we assume here the case in which we are iterating over edges and we increment adjacent vertices, a common and general pattern in OP2 and CFD codes. Any other

```
1  loop over colors C //seq
2    loop over partitions P in C //par
3      stage in all data for contrib
4      loop over edges e in P //par
5        tmp = contributions (e)
6     stage out all tmp data
7    end loop
8  end loop
9
10 loop over colors C //seq
11   loop over partitions P in C //par
12     stage in all data for update1 (tmp)
13     loop over edges e in P //par
14       update ( e(v1), tmp)
15    stage out all data for partition p in P
16   end loop
17 end loop
18
19 loop over colors C //seq
20   loop over partitions P in C //par
21     stage in all data for update2 (tmp)
22     loop over edges e in P //par
23       update ( e(v2), tmp)
24    stage out all data for partition p in P
25   end loop
26 end loop
```

Figure 4. Split op_par_loop by generating three successive loops. This requires modification of the user code.

OP2 loops can be reduced to this or to a simpler pattern along with simple code movement.

For each iteration, a thread computes a contribution (line 5) and then it applies it to the two vertices (lines 6 and 7). Finally, all modified datasets are staged back from shared memory to global memory. Notice that using a given edge, the same contribution is applied to the two vertices, as it is often the case in CFD programs. The alternative, where two different contributions are computed, requires no different support from the point of view of splitting.

The work in [3] describes a splitting technique in which the single loop is split into three loops, as shown in Figure 4. Consider that the user kernel can be split into three phases: (1) computation of the contribution; (2) update of the first vertex with the contribution; and, (3) update of the second vertex with the contribution. We can thus derive three loops with equivalent semantics of the original one which share a common dataset associated to edges holding the contribution. Unfortunately, this scheme requires us to stage the contributions three times between global and shared memory, resulting in high overhead.

In contrast to the scheme in [3], we describe here the case in which the initial loop is not transformed into multiple loops; the splitting happens as an alternative code synthesis. In the improved implementation of the loop, shown in Figure 5, the partition loop body (lines 1 and 2) now alternates the three phases (contribution calculation, first and second updates). Unlike the previous splitting technique, the contribution temporary dataset is kept in global memory and accessed directly during the computation of a partition. That is, it is not staged between global and shared memory.

This scheme reduces the shared memory requirements for the loop because strictly only the data needed in each step is staged into shared memory. For instance, as the incremented dataset is not used during the contribution calculation (OP_INC semantics), it can be omitted during the initial stage in phase (line 3). Also, after computing the contribution, all data not required in the update steps can be simply overwritten, and we can use the whole shared memory for the vertex data alone.

```
loop over colors C //seq
  loop over partitions P in C //par
    stage in all data for contrib
    loop over edges e in P //par
      tmp(e) = contrib (e)
    stage in all data for update1
    loop over edges e in P //par
      update ( e(v1), tmp(e) )
    stage out all data for update1
    stage in all data for update2
    loop over edges e in P //par
      update ( e(v2), tmp(e) )
    stage out all data for update2
  end loop
end loop
```

Figure 5. Split of contribution calculation and vertex updates.

If the contribution for the two vertices is different, then their computation falls in the first step. Note that the temporary data has now been promoted from a local user kernel variable, to an array associated to edges. As our target is the reduction of shared memory requirements, this is allocated into global memory.

Under this new scheme we do not need to allocate the vertex data to be updated into shared memory while computing the contribution. Also, all data needed for the contribution need not to be allocated during the update. This means that the shared memory requirements for the three steps is smaller than the original loop.

The last version that we describe in Figure 6 splits the contribution calculation in successive loops. We assume that the function *contrib* can be split into multiple functions, say contrib1, . . . , contribN. This is done using automatic support such as that provided by ROSE [1] for outlining code sections into functions. The splitting is similar to the previous case, where at each stage we stage into shared memory only the data required by each successive contribution function. For instance, we initially stage in all data for the evaluation of the first contribution at line 3. The computed temporary data after each contribution is associated to edges and stored, as above, into global memory. This temporary data can be produced by a contribution calculation, and then reused by a successive calculation, or in the vertex update step.

The goal of this is to further reduce the shared memory requirements for each of the contribution calculation, which results in larger total partition sizes. Of course this comes at the cost of the need to manage temporary datasets (associated to edges) which are stored in global memory. The resulting trade off between reducing shared memory requirements (thus increasing the partition size) and the added global memory traffic is what decides the particular best loop splitting strategy. Note that this strategy, in general, depends on the application characteristics along with parameters of the target GPU architecture. In this paper, we present a methodology to address this issue for the case of large unstructured mesh applications.

5. Experiments

In this section, we present the performance results for the first simple loop splitting technique. Our aim is the study of an optimal loop splitting strategy to be applied in a fully-automated optimizing compiler. For this purpose, we study the effects of loop splitting on two different architectures.

We used a CFD simulation software developed at Rolls-Royce for the simulation of turbomachinery engines, called HYDRA. Performance studies of HYDRA have been reported in [3]. We apply the loop splitting technique to six loops resulting from the simulation of a standard CFD test case, called NASA Rotor 37.

```
1   loop over colors C //seq
2     loop over partitions P in C //par
3       stage in all data for contrib1
4       loop over edges e in P //par
5         tmp1(e) = contrib1 (e); tmp(e) = tmp1(e);
6       stage in all data for contrib2
7       loop over edges e in P //par
8         tmp2(e) = contrib2 (e); tmp(e) += tmp2(e);
9       ...
10      stage in all data for contribN
11      loop over edges e in P //par
12        tmpN(e) = contribN (e); tmp(e) += tmpN(e);
13      ...
14      stage in all data for update1
15      loop over edges e in P //par
16        update ( e(v1), tmp(e) )
17      stage out all data for update1
18      stage in all data for update2
19      loop over edges e in P //par
20        update ( e(v2), tmp(e) )
21      stage out all data for update2
22    end loop
23  end loop
```

Figure 6. Split of contribution calculation in multiple functions.

For the simulation, we use a triangular mesh with approximately 2.5 million edges and the simulation is based on double-precision floating point operations. The studied loops all iterate over an edge set (edges or boundary edges), and they follow the described CFD loop pattern: they compute a contribution which is then applied to the two adjacent vertices (same contribution). We study four loops and we report here the size in byte of indirectly and directly accessed datasets because of its relevance with respect to the loop splitting technique:

- Accumulation over edges (Accu), which access 680 bytes of datasets indirectly for each iteration. The incremented datasets are 416 bytes in size per iteration.
- Contribution on edges (Edgecon), which accesses indirectly 528 bytes and directly 24 bytes for each iteration. The incremented datasets are 384 bytes in size per iteration.
- Viscous or smoothing fluxes calculation (Vflux), which accesses indirectly 808 bytes and directly 24 bytes for each iteration. The incremented datasets are 96 bytes in size per iteration.
- Inviscid flux calculation (Iflux), which accesses indirectly 328 bytes and directly 24 bytes. The incremented datasets are 96 bytes in size per iteration.

The involved user kernels for the described loops can be as complex as including up to 600 double precision floating point operations.

We performed experiments using an NVIDIA GPU C2070, running the original and the split versions of each loop. Similar results are also shown in [3] and we do not expect relevant differences. Unlike previous contributions, in this paper we run the same experiments on an dual 6-core Intel Westmere X5660, to understand the applicability of the studied technique on different architectures. Table 1 gives details of the used architectures. The aim of the comparison is to understand if loop splitting is a desirable feature also on architectures with large caches.

For the GPU experiments we maximized the partition size in order to maximize parallelism on each SM on the NVIDIA GPUs. On the CPU we tested two partition sizes for every loop: 128 and 512. We expect loop splitting to influence data locality, and the choice of two highly different partition sizes permit us to study this effect. To maximize performance and stability of execution

Table 1. Single node CPU system specifications

Node System	Cores/node (Clock/core)	Mem. /node	Compiler [flags]
2×Intel Xeon X5650 (Westmere)	12 [24 SMT] (2.67GHz)	24 GB	IFORT 11.1 -openmp -O2 -parallel
Tesla C2070	448 (1.15GHz)	6.0 GB	pgfortran 12.2 -O4 nvcc 4.1 -O3

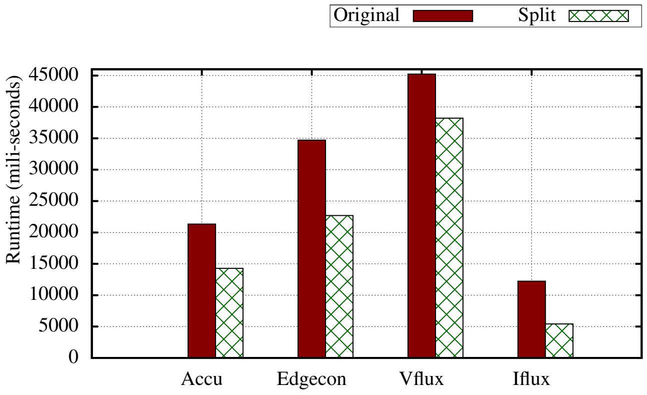

Figure 7. Loop performance of NASA Rotor 37 with 2.5 million edges on a GPU. This version of the loops is not tuned w.r.t. the block size.

times we used the Intel thread affinity support in *scatter* mode, using the two Westmere nodes as a single 12-core processor. As we execute in parallel, only partitions that do not share data, on different threads, expect this to maximize main memory bandwidth. The results are for experiments with parallelism degree (number of OpenMP threads) equal to 1, 2, 6 and 12. We do not show the remaining degrees for space reasons.

Figures 7 and 8 show the results for the baseline and split versions of the studied loops. The two graphs differentiate in the fact that the first one does not optimize w.r.t. the CUDA block, while the second one does. As it can be noticed, the impact of loop splitting is higher when the CUDA block is not optimized. This means that loop splitting alleviates the slowdown given by a wrong choice the CUDA run-time configuration parameters for the studied loops. The maximum reported improvement of loop splitting is 34.5% when the CUDA thread block is not optimized, and 22.5% when the CUDA thread block is optimized.

For the CPU experiments we show results in Figure 9 (1 thread), Figure 10 (2 threads), Figure 11 (6 threads), and Figure 12 (12 threads). As expected, the impact of loop splitting is smaller on CPUs compared to GPUs. We can notice that we obtain some performance improvements for the Vflux and Iflux loops with small parallelism degrees. This is due to the large number of bytes required for each iteration of these two loops compared the size of incremented datasets; thus, splitting these loops highly improves the two increment loops as we can now run them with a large partition size compared to cases in which large datasets need to be incremented. By increasing the parallelism degree and by scattering the threads over the two multicore nodes we amortize the impact of loop splitting for the two loops as more threads access less and different data. From these results we can conclude that, except for some larger and specific cases when using small parallelism degrees, a fused version of loops is always to be preferred. An optimizing compiler should do its best to maximize loop fusion, e.g. by using mesh-independent techniques (e.g. see [2]) or by applying sparse tiling [13].

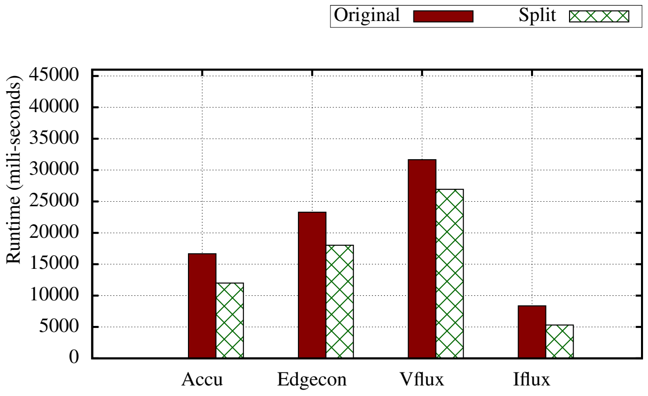

Figure 8. Loop performance of NASA Rotor 37 with 2.5 million edges on a GPU. This version of the loops is tuned w.r.t. the block size.

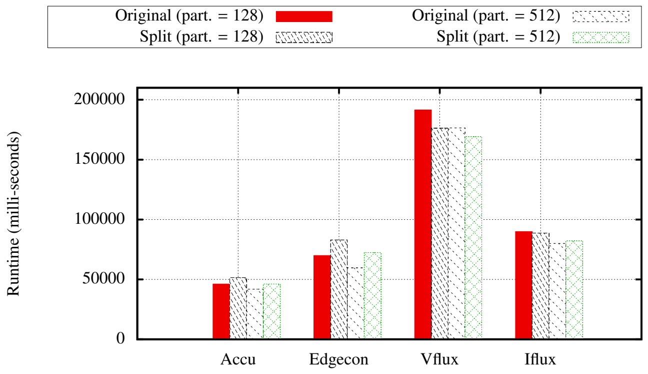

Figure 9. Loop performance of NASA Rotor 37 with 2.5 million edges on a CPU with 1 OpenMP thread. The graph shows execution time of each loop in its original and split version

6. Conclusion

In this paper, we have described our study on general loop splitting techniques for unstructured mesh applications. The aim is the optimization of the shared memory requirements for large loops, i.e., loops accessing large quantities of data for each iteration.

Based on the OP2 implementation of parallel loops over an unstructured mesh, we derived multiple versions. The first technique is a simple splitting of the original loop into three loops. It is based on a common user kernel property in CFD applications in which the same contribution is applied to multiple indirectly accessed datasets. In this case, the contribution calculation and the update of the indirect datasets can be re-mapped to successive loops. However, this requires user code modification and multiple staging of contributions between global and shared memory on a GPU.

We have shown that this can be avoided by synthesizing a single loop implementation for the original parallel loop, but following a split behavior. The first loop splitting technique alternates contribution calculation and updates while executing a same partition. As a consequence, the input user code does not require modification, and the contributions can be kept in global memory and we can rely on the L1 cache on a GPU. The general loop splitting technique assumes that the contribution calculations can be split into multiple successive functions. The related code synthesis splits the contribution calculation for each partition further by staging into shared memory only the necessary data for each sub-function in which the contribution is divided. The key is that each function requires a

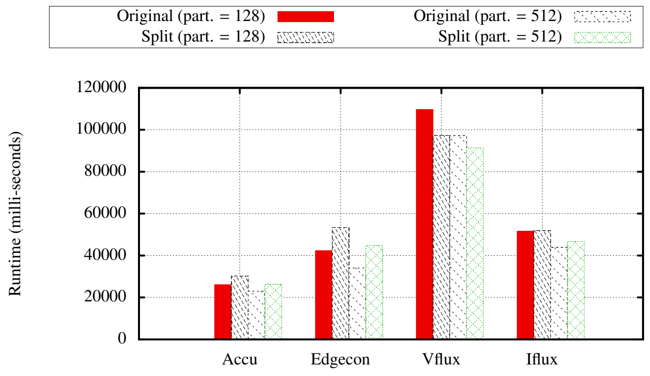

Figure 10. Loop performance of NASA Rotor 37 with 2.5 million edges on a CPU with 2 OpenMP threads. The graph shows execution time of each loop in its original and split version

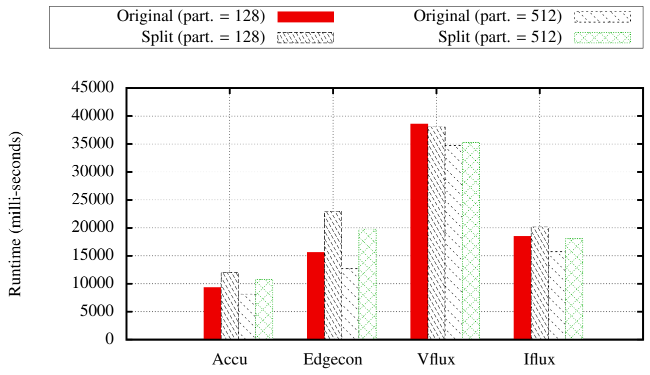

Figure 11. Loop performance of NASA Rotor 37 with 2.5 million edges on a CPU with 6 OpenMP threads. The graph shows execution time of each loop in its original and split version

smaller number and size of indirectly addressed data, to be stored into shared memory. This permits maximizing the overall partition size, as less data is required to be allocated on the shared memory at the same time.

We have presented experimental results for four complex loops for the first simple splitting on a GPU to validate the efficacy of our approach. On GPUs, we obtained improvements of up to 34.5% over the baseline implementation. We have also studied the effect of loop splitting for the same loops on a CPU, featuring larger caches, to understand the strategy that is to be followed by an optimizing compiler on these architectures. These results demonstrate that, except in some corner cases with small parallelism degrees, the fused version of loops always performs better that the split ones on CPUs.

Acknowledgments

This research is funded by the UK Technology Strategy Board and Rolls-Royce plc. through the Siloet project, the UK EPSRC (EP/I006079/1, EP/I00677X/1, EP/J010553/1 and EP/L000407/1), the Future of Computing E9RCTDO Project funded by the Oxford Martin Foundation, by the U.S. National Science Foundation through grants 0811457, 0926687 and 1059417, by the U.S. Army through contract W911NF-10-1-000, and by Louisiana State University. We are grateful to Leigh Lapworth, Yoon Ho and David Radford at Rolls-Royce.

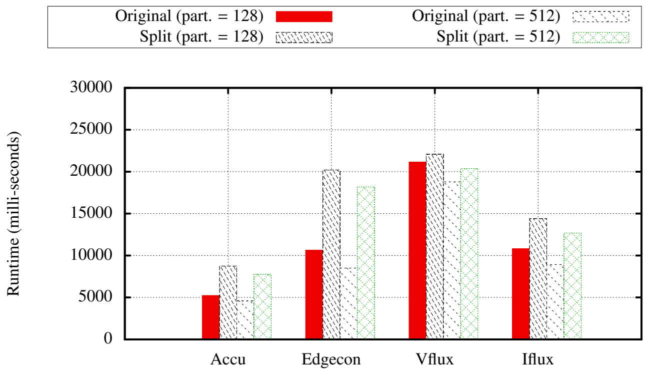

Figure 12. Loop performance of NASA Rotor 37 with 2.5 million edges on a CPU with 12 OpenMP threads. The graph shows execution time of each loop in its original and split version

References

[1] The ROSE compiler. `rosecompiler.org/`.

[2] C. Bertolli, A. Betts, P. H. Kelly, G. R. Mudalige, and M. B. Giles. Mesh independent loop fusion for unstructured mesh applications. In *Proceedings of CF'12*, pages 43–52, 2012.

[3] C. Bertolli, A. Betts, N. Loriant, G. Mudalige, D. Radford, D. Ham, M. Giles, and P. Kelly. Compiler optimizations for industrial unstructured mesh CFD applications on gpus. In H. Kasahara and K. Kimura, editors, *LCPC*, volume 7760 of *LNCS*, pages 112–126. Springer, 2013.

[4] T. Brandvik and G. Pullan. SBLOCK: A framework for efficient stencil-based PDE solvers on multi-core platforms. In *CIT'10*, pages 1181–1188, 2010.

[5] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *J. Parallel Distrib. Comput.*, 22(3):462–478, Sept. 1994.

[6] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *SC'11*, pages 9:1–9:12, 2011.

[7] M. B. Giles. OP2 User's Manual, November 2014. `http://intranet.oerc.ox.ac.uk/personal-pages/gihan/op2-documentation/C-_Users_Guide.pdf`.

[8] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly. Performance analysis and optimization of the OP2 framework on many-core architectures. *The Computer Journal*, 55(2):168–180, 2012.

[9] M. B. Giles, G. R. Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing OP2 for GPU architectures. *J. Parallel Distrib. Comput.*, 73(11):1451–1460, Nov. 2013.

[10] J. Holewinski, L. Pouchet, and P. Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *ICS'12*, pages 311–320, 2012.

[11] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998. ISSN 1064-8275.

[12] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *Proceedings of SC'11*, pages 11:1–11:12, 2011.

[13] M. M. Strout, L. Carter, J. Ferrante, and B. Kreaseck. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications*, 18(1):95–114, February 2004.

[14] Y. Tang, R. A. Chowdhury, B. C. Kuszmaul, C.-K. Luk, and C. E. Leiserson. The pochoir stencil compiler. In *Proceedings of SPAA'11*, pages 117–128, 2011.

[15] D. Unat, X. Cai, and S. B. Baden. Mint: Realizing CUDA performance in 3D stencil methods with annotated C. In *ICS '11*, pages 214–224, New York, NY, USA, 2011.