# Designing OP2 for GPU Architectures

M.B. Giles[a], G.R. Mudalige[a,*], B. Spencer[b], C. Bertolli[c], I. Reguly[d]

[a] *Oxford e-Research Centre, University of Oxford, U.K.*
[b] *Dept. of Computer Science, University of Oxford, U.K.*
[c] *Dept. of Computing, Imperial College London, UK*
[d] *Pázmány Péter Catholic University, Hungary*

**Abstract**

OP2 is an "active" library framework for the solution of unstructured mesh applications. It aims to decouple the specification of a scientific application from its parallel implementation to achieve code longevity and near-optimal performance through re-targeting the back-end to different multi-core/many-core hardware. This paper presents the design of the current OP2 library for generating efficient code targeting contemporary GPU platforms. In this we focus on some of the software architecture design choices and low-level optimizations to maximize performance on NVIDIA's Fermi architecture GPUs. The performance impact of these design choices is quantified on two NVIDIA GPUs (GTX560Ti, Tesla C2070) using the end-to-end performance of an industrial representative CFD application developed using the OP2 API. Results show that for each system, a number of key configuration parameters need to be set carefully in order to gain good performance. Utilizing a recently developed auto-tuning framework, we explore the effect of these parameters, their limitations and insights into optimizations for improved performance.

*Keywords:* Performance, GPU, CUDA, Unstructured mesh applications, auto-tuning

## 1. Introduction

OP2 is an "active" library framework for the solution of unstructured mesh applications. It utilizes code generation to exploit parallelism on heterogeneous multi-core/many-core architectures. The "active" library approach uses program transformation tools, so that a single application code written using the OP2 API is transformed into the appropriate form that can be linked against a target parallel implementation (e.g. OpenMP, CUDA, OpenCL, AVX, MPI, etc.) enabling execution on different back-end hardware platforms.

Such an abstraction enables application developers to focus on solving problems at a higher level and not worry about architecture specific optimizations. This splits the problem space into (1) a higher application level where scientists and engineers concentrate on solving domain specific problems and write code that remains unchanged for different underlying hardware and (2) a lower implementation level, that focuses on how a computation can be executed most efficiently on a given platform by carefully analyzing the data access patterns. This paves the way for easily integrating support for any future novel hardware architecture.

Currently, unstructured mesh applications developed using the OP2 API can be transformed into code that can be executed on a single multi-core and/or multi-threaded CPU node (using OpenMP) or a single GPU (using NVIDIA CUDA). This paper presents the key design features of the current OP2 library for generating efficient code targeting GPUs based on NVIDIA's current Fermi architecture. Based on an assessment of the actual bandwidth relative to the theoretical peak, we show that we can achieve near-optimal performance for a significant application. This requires a number of design optimizations based on a through understanding of the GPU architecture. The advantage of a library such as OP2 is that it provides non-expert application developers with these benefits with minimal extra effort.

More specifically, we make the following contributions:

- We present, and discuss, key design issues in developing optimized unstructured mesh applications for GPU architectures. These consists of (1) the impact of different data layouts (array of structs vs. struct of arrays) and (2) techniques for managing data dependencies. OP2's design choices are detailed and justified, with quantitative insights into their contributions to performance. To our knowledge, this is the first detailed account of near-optimal code design for this class of applications on GPUs.
- The performance impact of the above design choices are quantified on a range of NVIDIA GPUs using the end-to-end performance of an industrial representative CFD application (Airfoil) developed using

*Corresponding author, Address: Oxford e-Research Centre, University of Oxford, 7 Keble Road, Oxford. OX1 3QG, U.K., Tel: +44 (0)1865 610787, Fax: +44 (0)1865 610612

*Email addresses:* mike.giles@maths.ox.ac.uk (M.B. Giles), gihan.mudalige@oerc.ox.ac.uk (G.R. Mudalige), benjamin.spencer@balliol.ox.ac.uk (B. Spencer), c.bertolli@imperial.ac.uk (C. Bertolli), reguly.istvan@itk.ppke.hu (I. Reguly)

the OP2 API. The OP2 code generation tools are used to generate NVIDIA CUDA code to execute on these systems. Benchmarked systems include, two NVIDIA GPUs; a popular consumer graphics card (GTX560Ti) and a high performance computing card (Tesla C2070). The GPU results are also contrasted with the equivalent multi-core CPU results, generated through OP2's OpenMP code generation back-end, on a CPU node system with two high-end 6-core Intel Westmere processors.

- Results show that for each system, a number of configuration parameters needs to be set in order to gain good performance. For GPUs the "optimum point" of performance falls within a narrow range of parameter values and thus each needs to be carefully chosen, in order to avoid significantly sub-optimal performance. To support this task, we develop a novel auto-tuning framework to configure an application, and apply it to explore key performance aspects of the Airfoil code.

The rest of this paper is organized as follows: Section 2 details related work including development of abstraction frameworks such as OP2 for multi-architecture platforms; Section 3 provides a description of the unstructured mesh application class and details the OP2 API; Section 4 presents in detail OP2's current design for generating executables for GPUs; Section 5 present performance figures for the execution of Airfoil on a range of NVIDIA GPU systems, quantifying the performance impact of key design choices. Section 6 investigates the optimal configuration of the Airfoil code using an auto-tuning framework. Finally Section 7 concludes the paper.

## 2. Related Work

OP2 is the second iteration of OPlus (Oxford Parallel Library for Unstructured Solvers) [8], a research project that had its origins in 1993 at the University of Oxford. OPlus provided an abstraction framework for performing unstructured mesh based computations across a distributed-memory cluster of processors. It is currently used as the underlying parallelization library for Hydra [19, 14] a production-grade CFD application used in turbomachinery design at Rolls-Royce plc. OP2 builds upon the features provided by its predecessor but develops an "active" library approach with code generation to exploit parallelism on heterogeneous multi-core/many-core architectures.

Although OPlus pre-dates it, OPlus and OP2 can be viewed as an instantiation of the AEcute (access-execute descriptor) [18] programming model that separates the specification of a computational kernel with its parallel iteration space, from a declarative specification of how each iteration accesses its data. The decoupled Access/Execute specification in turn creates the opportunity to apply powerful optimizations targeting the underlying hardware.

A number of related research projects have implemented similar programming frameworks. The most comparable of these is LISZT [11] from Stanford University.

LISZT is a domain specific language (embedded in Scala [6]) specifically targeted to support unstructured mesh application development. The aim, as with OP2, 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. A Liszt application is translated to an intermediate representation which is then compiled by the Liszt compiler to generate native code for multiple platforms. 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 [11]. The Navier-Stokes application in [11] is most comparable to the Airfoil application and shows similar speed ups to those gained with OP2 in our work.

Related work in the solution of unstructured mesh applications on GPUs, particularly in the CFD domain, has also appeared elsewhere. In [10], techniques to implement an unstructured mesh inviscid flow solver on GPUs are described. Average speed-ups of about 9.5× are observed during the execution of the GPU implementation on an NVIDIA Tesla 10 series card against an equivalent optimized 4-thread OpenMP implementation on a quad-core Intel Core 2 Q9450. The authors, later in [9] also describe a semi-automatic script based method of porting a large CFD code written in Fortran/OpenMP to NVIDIA CUDA. Similarly [7] reports the GPU performance of a Navier-Stokes solver for steady and unsteady turbulent flows on unstructured/hybrid grids. The computations were carried out on NVIDIA's GeForce GTX 285 graphics cards (in double precision arithmetic) and speed-ups up to 46× (vs a single core of two Quad Core Intel Xeon CPUs at 2.00 GHz) are reported.

We believe the research in this paper differs from the above works and advances the state-of-the-art with a number of novel contributions. Firstly, to our knowledge, this paper is the first of its kind to quantitatively present the performance impact of design choices for developing unstructured mesh applications on GPUs. The insights presented as part of this work are especially important for gaining good performance on the NVIDIA Fermi architecture with its new L1/L2 cache. Secondly, we believe that the use of auto-tuning methods for optimally configuring GPU code is also novel, particularly applied to a non-trivial unstructured mesh application such as Airfoil. Finally, research in GPU acceleration often cites speed-ups, relative to a hand-coded CPU implementation - sometimes even comparing to a single-core. In this paper we compare the end-to-end performance of a representative application on contemporary flagship platforms (GTX560Ti, NVIDIA C2070, Intel 6-core Westmere). For each architecture (Fermi and x86) OP2 generates highly-optimized code, using the same application code, allowing for a direct performance comparison.

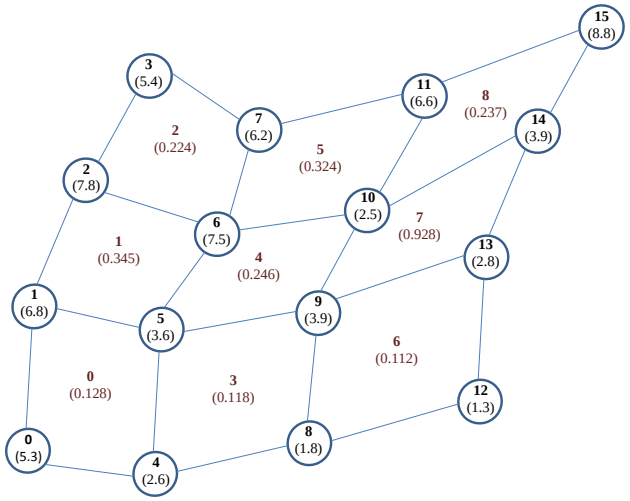



Figure 1: An example mesh with node and quadrilateral cell indices, and associated data values in parenthesis

## 3. Background

Unstructured grids/meshes have been and continue to be used over a wide range of computational science and engineering applications. They have been applied in the solution of partial differential equations (PDEs) in computational fluid dynamics (CFD), computational electromagnetics (CEM), structural mechanics and general finite element methods. Usually, in three dimensions, millions of elements are often required for the desired solution accuracy, leading to significant computational costs.

Unstructured meshes, unlike structured meshes, use connectivity information to specify the mesh topology. The OP2 approach to the solution of unstructured mesh problems involves breaking down the algorithm into four distinct parts: (1) sets, (2) data on sets, (3) connectivity (or mapping) between the sets and (4) operations over sets. These leads to an API through which any mesh or graph can be completely and abstractly defined. Depending on the application, a set can consist of nodes, edges, triangular faces, quadrilateral faces, or other elements. Associated with these sets are data (e.g. node coordinates, edge weights) and mappings between sets which define how elements of one set connect with the elements of another set.

Figure 1 illustrates a simple quadrilateral mesh that we will use as an example to describe the OP2 API. The mesh can be defined by two sets, nodes (vertices) and cells (quadrilaterals). There are 16 nodes and 9 cells, which can be defined using the OP2 API as follows:

```
int nnode = 16;
op_set nodes = op_decl_set(nnode, "set_nodes");

int ncell = 9;
op_set cells = op_decl_set(ncell, "set_cells");
```

The connectivity is declared through the mappings between the sets. The integer array `cell_map` can be used to represent the four nodes that make up each cell.

```
int cell_map[36] = { 0,1,5,4, 1,2,6,5, 2,3,7,6,
                     4,5,9,8, 5,6,10,9, 6,7,11,10,
                     8,9,13,12, 9,10,14,13, 10,11,15,14};

op_map mcell = op_decl_map(cells, nodes, 4,
                cell_map,"cell_to_node_map");
```

Each element of set `cells` is mapped to four different elements in set `nodes`. The `op_map` declaration defines this mapping where `mcell` has a dimension of 4 and thus its index 0,1,2,3 maps to nodes 0,1,5,4, index 4,5,6,7 maps to nodes 1,2,6,5 and so on. When declaring a mapping we first pass the source set (e.g. `cells`) then the destination set (e.g. `nodes`). Then we pass the dimension of each map entry (e.g. 4; as `mcell` maps each cell to 4 nodes).

Once the sets and connectivity are defined, data can be associated with the sets; the following are some data arrays that contain double precision data associated with the cells and the nodes respectively. Note that here a single double precision value per set element is declared. A vector of a number of values per set element could also be declared (e.g. a vector with three doubles per node to store the X,Y,Z coordinates).

```
double cell_data[9] = {0.128, 0.345, 0.224, 0.118,
                       0.246, 0.324, 0.112, 0.928,
                       0.237};
double cell_data_u;
cell_data_u = (double *)malloc(sizeof(double)*9);

double node_data[16] = {5.3, 6.8, 7.8, 5.4, 2.6, 3.6,
                        7.5, 6.2, 1.8, 3.9, 2.5, 6.6,
                        1.3, 2.8, 3.9, 8.8 };

op_dat dcells   = op_decl_dat(cells, 1, "double",
                         cell_data, "data_on_cells");

op_dat dcells_u = op_decl_dat(cells, 1, "double",
                cell_data_u, "updated_data_on_cells");

op_dat dnodes   = op_decl_set(nodes, 1, "double",
                         node_data, "data_on_nodes");
```

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. If the loop involves indirection through a mapping we refer to it as an indirect loop; if not, it is called a direct loop.

The OP2 API provides a parallel loop declaration syntax which allows the user to declare the computation over sets in these loops [13]. Consider the following sequential loop, operating over each cell in the mesh illustrated in Figure 1. Each of the cells updates its data value using the data values held on the four nodes connected to that cell.

```
void seq_loop(int ncell, int *cell_map,
              double *cell_data_u,
              double *cell_data,
              double *node_data)
{
  for (int i = 0; i<ncell; i++){
    cell_data_u[i] = cell_data[i] +
                     node_data[cell_map[4*i]] +
                     node_data[cell_map[4*i+1]] +
                     node_data[cell_map[4*i+2]] +
                     node_data[cell_map[4*i+3]];
  }
}
```

An application developer declares this loop using the OP2 API as follows, together with the "elemental" kernel function.

```
void kernel(double* cell_u, double* cell,
     double* n0, double* n1, double* n2, double* n3){
  *cell_u = *cell + n0[0] + n1[0] + n2[0] + n3[0];
}

op_par_loop(kernel,"kernel", cells,
   op_arg(dcells_u,-1,OP_ID, 1, "double", OP_WRITE),
   op_arg(dcells,  -1,OP_ID, 1, "double", OP_READ),
   op_arg(dnodes,  0, mcell, 1, "double", OP_READ),
   op_arg(dnodes,  1, mcell, 1, "double", OP_READ),
   op_arg(dnodes,  2, mcell, 1, "double", OP_READ),
   op_arg(dnodes,  3, mcell, 1, "double", OP_READ));
```

OP2 handles the architecture specific code generation and parallelization. The elemental kernel function takes six arguments in this case and the parallel loop declaration requires the access method of each to be declared (OP_WRITE, OP_READ, etc). OP_ID indicates that the data in `dcells` and `dcells_u` is to be accessed without any indirection (i.e. directly). `dnodes` on the other hand is accessed through the `mcell` mapping using the given index (0, 1, 2, and 3 respectively). The dimension of the data (in this example 1, for all data) is also declared. Complete details of the API can be found in [13].

OP2's general decomposition of unstructured mesh algorithms, imposes no restrictions on the actual algorithms, it just separates the components of a code. However, OP2 makes an important restriction that the order in which elements are processed must not affect the final result, to within the limits of finite precision floating-point arithmetic. This constraint allows the program to choose its own order to obtain maximum parallelism. Moreover the sets and mappings between sets must be static and the operands in the set operations cannot be referenced through a double level of mapping indirection (i.e. a mapping to another set which in turn uses another mapping to access data associated with a third set).

The straightforward programming interface combined with efficient parallel execution makes it an attractive prospect for the many algorithms which fall within the scope of OP2. For example the API could be used for explicit relaxation methods such as Jacobi iteration; pseudo-time-stepping methods; multi-grid methods which use explicit smoothers; Krylov subspace methods with explicit preconditioning; semi-implicit methods where the implicit solve is performed within a set member, for example performing block Jacobi where the block is across a number of PDE's at each vertex of a mesh. However, algorithms based on order dependent relaxation methods, such as Gauss-Seidel or ILU (incomplete LU decomposition), lie beyond the capabilities of the API.

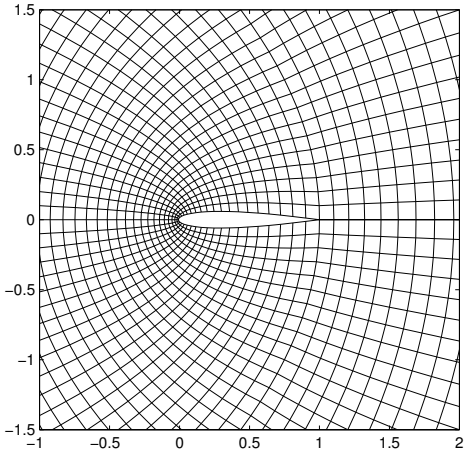



Figure 2: Rendering of a simple 120x60 mesh used in Airfoil

The example application used in this paper, Airfoil, is a non-linear 2D inviscid airfoil code that uses an unstructured grid [15]. It is a much simpler application than the Hydra [19, 14] CFD application used at Rolls-Royce plc. for the simulation of turbomachinery, but is representative of a production grade unstructured mesh application. A rendering of a smaller (120×60) unstructured mesh similar to the one used in Airfoil is illustrated in Figure 2. The actual mesh used in our experiments is of size 1200×600, which is too dense to be reproduced here. This consists of over 720K nodes, 720K cells and about 1.5 million edges. The code consists of five parallel loops: `save_soln`, `adt_calc`, `res_calc`, `bres_calc`, `update`. The most compute intensive loop `res_calc` has about 100 floating-point operations performed per mesh edge and is called 2000 times during total execution of the application. `save_soln` and `update` are direct loops while the other three are indirect loops.

## 4. OP2

The original OPlus library [8], was developed over 10 years ago for MPI/PVM based distributed memory execution of unstructured mesh algorithms written in Fortran. Its second iteration, OP2, is designed to leverage emerging multi-core and many-core hardware (GPUs, AVX etc.) on top of distributed memory parallelism allowing the user to execute on either a single multi-core/many-core node (including large shared memory systems), or a cluster of multi-core/many-core nodes. At the moment OP2 only

supports code development in C/C++, but a Fortran API is being developed with similar functionality.

The OP2 strategy for building executables for different back-end hardware consists of firstly generating the architecture specific code by pre-processing the user code, which is written using the OP2 API, and then secondly linking the generated code with the appropriate parallel implementation (e.g. OpenMP, CUDA, MPI, etc.). For example, when generating back-end code for executing on NVIDIA GPUs, the user's main program is parsed through the code generation tools, producing a modified main program and a CUDA file. The CUDA file *includes* a separate file for each of the kernel functions. These are then compiled and linked to the `op_lib.cu` library using a C compiler (e.g. gcc) and the NVIDIA CUDA compiler (nvcc), controlled by a Makefile. The design of the `op_lib.cu` library and exploring the performance impact of design choices forms the main theme of this work. The results presented in this paper come from code produced by a pre-processor written in MATLAB which only parses the individual OP2 routine calls. A new pre-processor is being developed by collaborators at Imperial College U.K. using the ROSE compiler framework [5]; this will parse the entire user code, allowing simplification of the API.

Currently, OP2 only supports generating parallel code based on CUDA and OpenMP. This will be later extended to generate code based on OpenCL and Intel AVX, thus supporting a wide range of CPU and GPU hardware. OP2 will also include support for the above to be executed on distributed memory CPU and GPU clusters in conjunction with MPI. To this end, a prototype implementation that allows to execute a user application on a cluster of CPUs using MPI is now nearing completion.

### *4.1. Data Dependencies*

A key design issue for developing parallel unstructured mesh applications is managing data dependencies encountered when incrementing indirectly referenced arrays. For example, in a mesh with edges and nodes, with a loop over edges updating nodes, a potential problem arises when two edges update the same node. A solution at a coarse grained level would be to partition the nodes such that the *owner* of the nodal data would carry out the computation. The drawback in this case is redundant computation when the two nodes for a particular edge have different *owners*. At the finer grained level, we could assign a "color" for the edges so that no two edges of the same color update the same node. This allows for parallel execution for each color followed by a synchronization. The disadvantage in this case is a possible loss of data reuse and loss of some parallelism. A third method would be to use *atomic* instructions, which combine read/add/write into a single operation.

The first method is applied at the distributed memory level where OP2 will partition the data so that the partition within each MPI process owns some of the set elements i.e. some of the nodes and edges. These partitions only perform the calculations required to update their own elements. However, it is possible that one partition may need to access data which belongs to another partition; in that case a copy of the required data is provided by the other partition. This follows the standard "halo" exchange mechanism used in distributed memory message passing parallel implementations. As the partition size becomes large, the proportion of "halo" data becomes very small.

At the distributed memory level (such as on clusters of CPUs and GPUs) the partition size is large. However, within a CPU or a GPU, operations are to be performed on a finer granularity on each processing unit. For a multicore CPU the processing units are processor cores (each based on a traditional heavy-weight core architecture such as x86 or IBM POWER). For NVIDIA GPUs the processing units are a number of relatively lightweight stream multiprocessors (SMs) each consisting of a number of stream processors (SPs) that share control logic, an instruction cache and a block of shared memory [2]. The new NVIDIA Fermi architecture also provides an L1 cache per SM.

Both coloring and atomics can be used to resolve the data dependency conflicts within a GPU. However, atomic operations (especially on doubles) are not present on all the hardware platforms we are interested in (atomics are an optional extension in OpenCL), and performance varies between platforms. Thus the current OP2 design uses coloring as the general solution. Atomic operations are available on the current Fermi architecture based NVIDIA GPUs and we are currently exploring its performance within OP2.

To resolve the data dependency issues with coloring on GPUs, indirect data from GPU global memory is loaded into each SM's shared memory space forming a local mini-partition. Each mini-partition is assigned to an SM and they are executed in parallel. The SPs within an SM, execute the mini-partition utilizing a number of threads (called a thread block). The threads are executed 32 at a time in parallel (called a *warp* in CUDA). During the execution of an indirect loop, the loop receives data from shared memory instead of global memory, maximizing data reuse and minimizing the traffic between global memory and shared memory. Thus on the GPU, updating the same node could occur either (1) by multiple threads within a single processing unit (an SM) updating data held in its shared memory (i.e. mini-partition) or (2) when the shared memory is written back to the GPU global memory which is used by other processing units. In OP2 thread coloring is used for the former and a block coloring is used for the latter.

Figure 3 illustrates coloring of cells for updating nodes connected to each. Nodes are located at the four corners of each cell. Cells are colored such that no two cells update the same node at the same time. As a result, the cells with the same color can be processed in parallel by different threads. However, on the GPU, synchronized execution of the same thread color across mini-partitions (which are assigned to different SMs) would have very poor per-

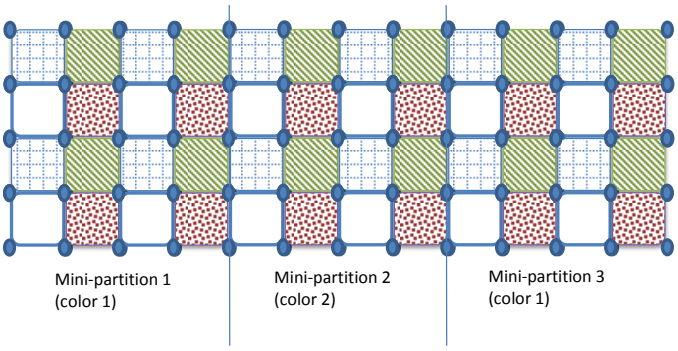



Figure 3: An example coloring of cells

formance. Thus a mini-partition level coloring scheme is used so that no two mini-partitions that share a common element gets the same color. In the figure, mini-partition 1 and 3 can be executed in parallel (assigning it the same color), while mini-partition 2 (which shares boundary values with mini-partitions 1 and 3) is executed after updated data from mini-partition 1 and 3 has been written back to global memory, and there has been a global synchronization.

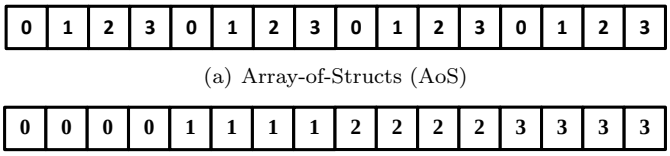
### *4.2. Data Layout in Memory*

The second key design issue for generating efficient GPU code is the form in which data should be organized when there are multiple components for each element. For example, in our Airfoil test case each cell has 4 variables; should these 4 components be stored contiguously for each cell (a layout which is sometimes referred to as an array-of-structs, AoS) or should all of the first components be stored contiguously, then all of the second components, and so on (a layout which is sometimes referred to as a struct-of-arrays, SOA)? Figure 4 illustrates the two options. The array-of-structs (AoS) approach views the 4 flow variables as a contiguous item, and holds an array of these. The struct-of-arrays (SoA) approach has a separate array for each one of the flow variables.

The SoA layout was natural in the past for vector supercomputers which streamed data to vector processors, but the AoS layout is natural for conventional cache-based CPU architectures for two reasons. Firstly, if there is a very large number of elements then each component for a particular element will be on a different virtual page, and if there are a lot of components this could be a problem[1].

The second is due to the fact that an entire cache line must be transferred from the memory to the CPU even if only one word is actually used. This is a particular problem for unstructured grids with indirect addressing; even with renumbering of the elements to try to ensure that neighboring elements in the grid have similar indices, it is often the case that only a small fraction of the cache line

[1]On an IBM RS/6000 workstation in the 1990's, one of the authors experienced a factor 10 drop in performance due to the limited size of the Translation Look-aside Buffer which holds a cache of the virtual memory address tables.




(a) Array-of-Structs (AoS)



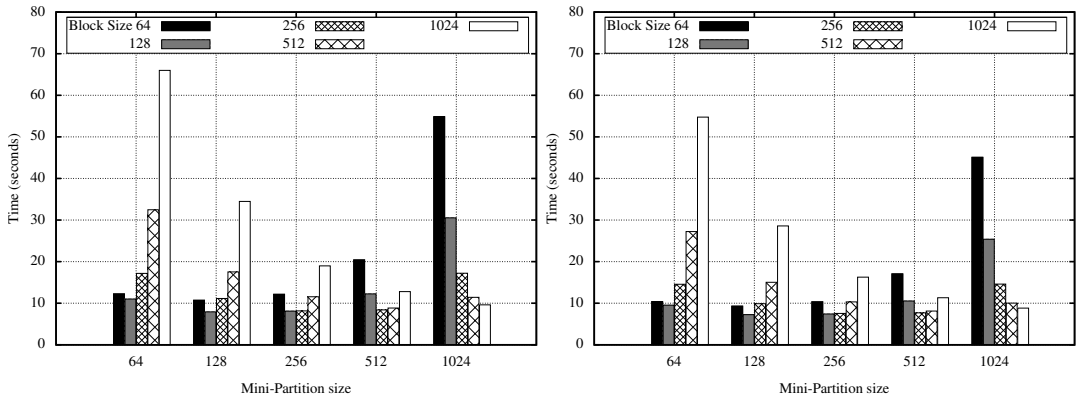

(b) Struct-of-Arrays (SoA)

Figure 4: AoS vs SoA data layouts

is used (vector supercomputers circumvented this problem by adding gather/scatter hardware to the memory subsystem). This problem is worse for SoA compared to AoS, because each cache line in the SoA layout contains many more elements than in the AoS layout. In an extreme case, with the AoS layout each cache line may contain all of the components for just one element, ensuring perfect cache efficiency, assuming that all of the components are required for the computation being performed.

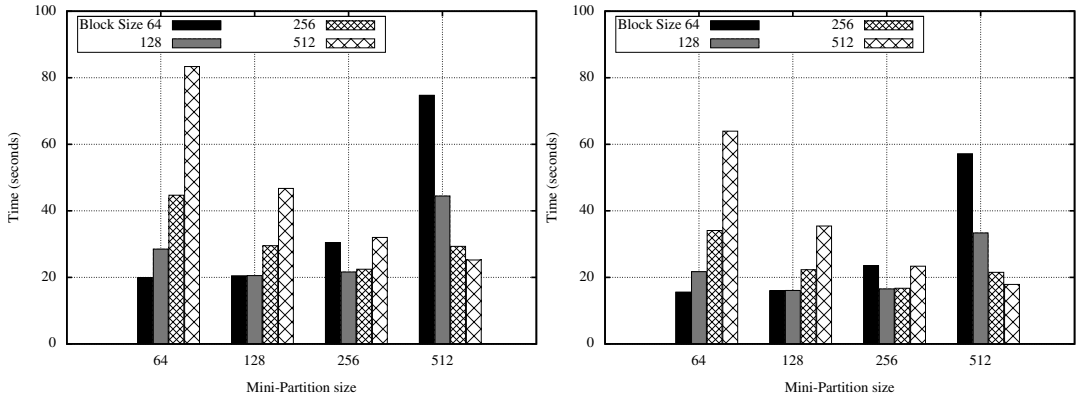
Until recently, NVIDIA GPUs did not have caches and most applications have used structured grids. Therefore, most researchers have preferred the SoA data layout which leads to natural memory transfer "coalescence" giving high performance. However, the latest NVIDIA GPUs based on the Fermi architecture have L1/L2 caches with a cache line size of 128 bytes, twice as large as used by Intel's latest Westmere CPUs [4]. This leads to significant problems with cache efficiency, especially since there is only 48kB of local shared memory and so not many elements are worked on at the same time. For example, in our Airfoil application, in the `res_calc` loop four floating-point values are computed on and within the Fermi architecture with a 128 bytes cache line, this corresponds to 32 floating-point values in single precision. When data is accessed indirectly, the SoA layout can lead to a worst-case scenario in which only 1/32 of the cache line is used. But with the AoS layout the worst case is only 1/8. Hence, in extreme cases with almost random addressing, the AoS layout could be 4 times more efficient than the SoA layout. The savings could be even larger for applications with more data per set element. Consequently, the AoS layout is used in OP2.

To ensure good coalescence in the data transfers requires some more complicated programming [12], but that is again a benefit of a library; it takes care of the complexity without burdening the application programmer. For example, during our optimization efforts, different thread numbering schemes for improved coalesced memory accesses were implemented for indirect loops. While, performance gains were observed for the Airfoil code, it was not clear whether a given thread numbering scheme was better in general for all unstructured mesh applications. This remains an open optimization that perhaps will be better implemented as a user configuration option at the code generation time. We also investigated the use of the `float4` data type to directly load a block of memory in an SMs registers. However, in this case there were no notable performance gains. Memory bank conflicts [2] were investigated due to the `res_calc` loop accessing shared




(a) GTX560Ti (b) Tesla C2070

Figure 5: Runtime of Airfoil on NVIDIA GPUs on a range of mini-partition sizes and thread-block size configurations - Single Precision




(a) GTX560Ti (b) Tesla C2070

Figure 6: Runtime of Airfoil on NVIDIA GPUs on a range of mini-partition sizes and thread-block size configurations - Double Precision

Table 1: GPU specifications

| GPU | Cores | Clock | Glob. Mem. | Shared Mem. /SM | Driver version (Comp. |
|---|---|---|---|---|---|
| | | GHz | GB | kB | Cap.) |
| GeForce GTX560Ti | 384 | 1.6 | 1.0 | 48 | 4.0 (2.1) |
| Tesla C2070 | 448 | 1.15 | 6.0 | 48 | 3.2 (2.1) |

Table 2: 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 | ICC 11.1 [-O2 -xSSE4.2] |

memory in four floating-point value blocks. We implemented padding to avoid bank conflicts but again did not observe significant benefits. We suspect that this is due to `res_calc` having sufficient computing demand so that any increased memory access latency is hidden from the critical path of execution.

## 5. GPU Performance

In this section, we present quantitative results exploring the performance impact of the above design features of OP2. Performance results are extracted from executing the Airfoil code on two NVIDIA GPUs based on their current flagship Fermi architecture. The specifications of these two GPUs - a consumer grade GTX560Ti and the high performance computing capable Tesla C2070 - are given in Table 1. On both GPUs the NVIDIA CUDA compiler (nvcc) was built using gcc 4.4.5 and the CUDA code generated for Airfoil by OP2 was compiled using `-O3 -arch=sm_20 -Xptxas -Xptxas=-v -dlcm=ca -use_fast_math` compiler flags. On both the GPUs the error correcting codes (ecc) were switched off.

Recall that the thread-blocks and mini-partitions are key features in the OP2 design for efficiently distributing work and operating in parallel over the mesh/grid elements on a GPU. The first set of results (Figure 5 and Figure 6) explores the performance trends due to selecting different mini-partition and the thread block sizes. The CUDA code generated by OP2 allows for these to be overridden at runtime. The thread-block and mini-partition sizes for the overall application or a different value for each individual loop could be set using command-line arguments.

The above figures presents the performance when the overall application's thread-block and mini-partition sizes are varied. Optimum configurations for each individual loop are discussed in Section 6.

In single precision (SP) the best runtime on both GPUs is just under 8 seconds, in both cases, achieved with a mini-partition size of 128 and a thread-block size of 128. In double precision (DP) the GTX560Ti runs in about 20 seconds, while the C2070 takes just under 16 seconds, when the application on both are configured to run with a mini-partition size of 64 and a thread-block size of 64. In DP, both cards failed to execute the application with a mini-partition size of 1024 as well as a thread block size of 1024 due to the large amount of shared memory required. Qualitatively both GPUs have similar performance behavior when the mini-partition and thread-block sizes are varied. In general we see that using a thread-block size equal to the mini-partition size gets close to the best performance achievable for each mini-partition size. However in some cases having a number of spare threads may be useful for carrying out more memory loads simultaneously with computation, but only when the GPU occupancy limits [2] are not overrun. Thus for example if having a larger thread-block size than the mini-partition size utilizes more registers than the maximum available number of registers per SM, then we see a performance degradation due to register spillage into global memory.

Given a thread-block size equal to the mini-partition size reducing the mini-partition size, decreases the amount of shared memory used and thus the GPU is able to execute multiple thread-blocks at the same time on each SM. This is advantageous as now one thread-block can be loading data into shared memory while another block is doing the computation. On the other hand, smaller mini-partitions results in less data re-use as the ratio of boundary/interior nodes and cells increases. Smaller mini-partitions also decreases cache efficiency. These conflicting trends gives rise to the run times we see in the above figures.

Next, we attempt to quantify and compare the amount of data transferred with GPU global memory due to the two different data layouts discussed in Section 4.2. The amount of data transfered during each indirect loop (`adt_calc`, `res_calc` and `bres_calc`) can be calculated as follows. Consider the case when a loop over elements (each containing a number of variables) of an indirectly accessed set is performed; for example the loop over cells (each with 4 flow variables) in `res_calc`. Then if the AoS data layout is used, we can compute the total number of bytes transferred from global memory to shared memory by counting the number of times a new cache line is loaded (assuming that the first element of each array is cache-aligned). For each new cache line loaded, the amount of data transferred is incremented by the cache line size if the access is only a read operation. If the access is a write operation then, we increment the amount of data transferred by two cache line sizes to account for the write back. The number of cache lines loaded will need to be increased if the variables making up an element takes more storage space than a single cache line size. Thus for example an element with 28 flow variables (each a double precision floating point value, i.e each of 8 bytes) will require 224 bytes of memory space in total. This is larger than the 128 byte cache line size on the NVIDIA Fermi architecture.

Table 3: Ratio of data transfer rates (SoA/AoS)

Single Precision

| Loop | Mini-partition size | | | | |
|---|---|---|---|---|---|
| | 64 | 128 | 256 | 512 | 1024 |
| `adt_calc` | 1.07 | 1.03 | 1.02 | 1.01 | 1.00 |
| `res_calc` | 2.21 | 1.87 | 1.56 | 1.33 | 1.18 |
| `bres_calc` | 2.65 | 2.64 | 2.63 | 2.62 | 2.62 |

Double Precision

| Loop | Mini-partition size | | | | |
|---|---|---|---|---|---|
| | 64 | 128 | 256 | 512 | 1024 |
| `adt_calc` | 1.04 | 1.02 | 1.01 | 1.01 | - |
| `res_calc` | 1.99 | 1.65 | 1.39 | 1.22 | - |
| `bres_calc` | 2.49 | 2.48 | 2.48 | 2.47 | - |

Alternatively consider the SoA data layout. Now, given a set with $N$ elements each with $v$ variables, then the distance between the first variable and the second variable (and so on) of each element will be $N\times$ `sizeof(double)` bytes. This number of bytes is significantly larger than a cache line of a GPU (or CPU), due to the size of $N$. Thus loading each element will mean that $v$ number of new cache lines needs to be transferred from global memory to access all the variables for that element. In addition to the data values transferred, we also include in our calculation the bytes transferred due to loading mapping tables that are used to perform the indirect accessing of data.

The ratio of data transfer rates (SoA/AoS) for both SP and DP are given in Table3. The results indicate that the AoS data layout is always better and for a number of cases reduces the data transfer between global memory and the GPU by over 50%. The ratios for the DP runs are less than the ratios from the SP runs, as the number of elements held within a cache line is smaller in DP for each cache line, if the data is organized in the AoS format.

## 6. Performance Analysis with Auto-tuning

Breaking down the runtime into the time taken by the five parallel loops reveals that the optimum mini-partition size and the thread-block size differs for each loop. For example, in DP, `res_calc` runs best when configured with a mini-partition size of 64 and a thread-block of 64 on the C2070, while `adt` is optimized at a mini-partition size of 128 and a thread-block of 64. Thus it is apparent that further runtime improvements could be gained by simply configuring each parallel loop to be executed on its optimum mini-partition size and thread-block size settings. However it is also apparent that incorrectly "guessing" the mini-partition and thread-block size could result in a significant performance loss.

Table 4: Auto-tuned results from a Tesla C2070

Single Precision

| Loop | Mini part size | block size | Time (sec) | BW useful (GB/s) | BW cache (GB/s) |
|---|---|---|---|---|---|
| save_soln | n/a | 512 | 0.22 | 104.30 | |
| adt_calc | 256 | 128 | 1.07 | 75.44 | 76.69 |
| res_calc | 128 | 128 | 4.85 | 37.64 | 61.81 |
| bres_calc | 64 | 256 | 0.07 | 7.38 | 27.94 |
| update | n/a | 128 | 0.95 | 103.11 | |
| Total min time (sec) | | | | | 7.16 |
| STD (sec) | | | | | $3.58 \times 10^{-3}$ |
| CV | | | | | $4.99 \times 10^{-4}$ |

Double Precision

| Loop | Mini part size | block size | Time (sec) | BW useful (GB/s) | BW cache (GB/s) |
|---|---|---|---|---|---|
| save_soln | n/a | 512 | 0.44 | 104.94 | |
| adt_calc | 128 | 64 | 2.62 | 52.87 | 53.83 |
| res_calc | 64 | 64 | 10.35 | 30.45 | 50.77 |
| bres_calc | 64 | 128 | 0.08 | 11.05 | 27.61 |
| update | n/a | 256 | 1.88 | 104.33 | |
| Total min time (sec) | | | | | 15.36 |
| STD (sec) | | | | | $5.11 \times 10^{-3}$ |
| CV | | | | | $3.32 \times 10^{-4}$ |

Table 5: Auto-tuned results from two, 6-core Intel Westmere CPUs

Single Precision - 24 OMP threads

| Loop | Mini part size | Time (sec) | BW useful (GB/s) | BW cache (GB/s) |
|---|---|---|---|---|
| save_soln | n/a | 1.15 | 20.09 | |
| adt_calc | 128 | 7.54 | 10.73 | 10.89 |
| res_calc | 512 | 12.21 | 14.13 | 15.21 |
| bres_calc | 128 | 0.10 | 4.81 | 10.80 |
| update | n/a | 3.61 | 27.10 | |
| Total min time (sec) | | | | 24.62 |
| STD (sec) | | | | 3.45 |
| CV | | | | 0.12 |

Double Precision - 24 OMP threads

| Loop | Mini part size | Time (sec) | BW useful (GB/s) | BW cache (GB/s) |
|---|---|---|---|---|
| save_soln | n/a | 2.35 | 19.57 | |
| adt_calc | 512 | 8.95 | 15.46 | 15.49 |
| res_calc | 1024 | 17.93 | 15.01 | 15.30 |
| bres_calc | 64 | 0.14 | 5.94 | 9.36 |
| update | n/a | 8.51 | 23.01 | |
| Total min time (sec) | | | | 37.89 |
| STD (sec) | | | | 1.97 |
| CV | | | | 0.05 |

Our observations has been that the scarce resources on the GPUs and their conflicting utilizations have significantly narrowed the "optimum-spot" of performance. Results illustrated on Figure 5 and Figure 6 also display an increased complexity in identifying the best parameter values. On the one hand there appear to be multiple optimums, while on the other hand straying even a little away from these points will result in considerably poor performance. The above issue could become further complicated by having to choose values/settings for other parameters such as compiler options, environmental variables and hardware configurations (e.g. whether to use a 16/48 or 48/16 split for the L1 cache/shared memory). Even the underlying design/implementation could be parameterized in the future (e.g. options to choose between the AoS or SoA data layouts) to allow the user to generate code with greater flexibility for a range of contrasting CPU/GPU architectures. Thus it is important that a flexible and efficient mechanism is used to arrive at the optimum configuration.

For the remainder of this paper, we use one such system - an auto-tuning framework - to further investigate the performance of the Airfoil application under optimum configuration on NVIDIA GPUs. Given a number of parameters and possible values for each, our auto-tuning framework (written in Python) selects the optimum combination of parameters by an exhaustive "brute force" search of the state space. However, the system allows to indicate which parameters can be optimized independently (e.g. the parameters for one parallel loop do not affect the execution of another parallel loop). This information is exploited to greatly reduce the number of configurations which need to be tested and is essential to making such an exhaustive search viable. Input specification for auto-tuning includes parameters and possible values, a mechanism to compile the code, perhaps using some of the parameter values and a mechanism to run the code, again perhaps using some of the parameter values. By default, the run-time is used as the "figure-of-merit" to be optimized. This can be overridden by other figures of merit as required. The framework can run the application under auto-tuning multiple times for each combination of parameter values and report the average, minimum and maximum together with other statistics such as the standard deviation (STD).

Table 4 presents the auto-tuned results for the C2070 GPU in both SP and DP. These results were obtained by configuring the auto-tuning framework to execute 10 runs for each parameter combination, selecting the minimum out of the 10 runs as the figure of merit for choosing the overall optimum parameter combination. The STD suggests that the run times were extremely consistent across multiple runs and executions on the C2070 GPU had negligible noise and/or other perturbations.

The last two columns of the table indicate the memory bandwidth utilization of the GPU. The first bandwidth column figures were obtained by counting the total amount

of useful data bytes transferred with global memory during the execution of a parallel loop and dividing it by the runtime of the loop. The second column takes into account the size of the whole cache line for the C2070 in the memory bandwidth calculation of indirect loops. For both SP and DP runs, the useful memory bandwidth utilization on the C2070, is up to 70% of the available upper limit of 144GB/s [3] on `save_sol`, but remains relatively low on `res_calc`. However the memory bandwidth figures taking in the cache line loading is considerably higher, with between 30% to 50% of the 144GB/s utilized for the indirect loops.

Given the mesh size, we can approximately compute the SP floating-point performance achieved on the C2070 during the most compute intensive loop, `res_calc`. The mesh consists of approximately 1.5 million edges, each responsible for 100 floating-point operations in `res_calc`. This routine is in turn called 2000 times giving $30 \times 10^{10}$ floating-point operations in total. The total time spent in the `res_calc` loop is about 4.85 seconds. This is about 60 GFlops/sec. In DP, the total run times spent in the `res_calc` loop is 10 seconds. This translates to a double precision floating-point performance of about 29 GFlops/sec on the C2070. Thus we see that only a fraction of the peak SP and DP floating-point performance on the GPU is achieved [3].

A key concern in determining whether GPUs are suitable for main-stream high performance computing and production scientific work is how their performance compares against the traditional mainstream processors. Thus, for comparison, the performance of the equivalent auto-tuned OpenMP version of the Airfoil application (also generated by OP2) on two 6-core Intel Westmere (2.67 GHz with SMT) processors is given in Table 5. The specifications of the CPU node system is detailed in Table 2. The key parameters for auto-tuning in this case were the number of OpenMP threads (OMP_NUM_THREADS = 12, 16, 18, 20, 24) and the mini-partition size. Each mini-partition is executed by a single OpenMP thread and mini-partitions are colored to stop multiple blocks trying to update the same data in the main memory simultaneously [17]. The optimum number of OpenMP threads for both SP and DP executions on this system was 24. The higher standard deviations observed on the CPU runs, are believed to be due to operating system noise and other perturbations that affect the system's CPU more readily than a discretely attached GPU. The observed memory bandwidth figures on the Westmere system is close to 50% of the maximum available bandwidth (32 GB/s [1]) during the most time consuming `res_calc` loop. Other loops such as `update` get much closer to saturating the available memory bandwidth (particularly in SP) on the CPU. Thus we suspect that the memory bandwidth of the single node system may become the bottleneck limiting future thread scalability of multi-core CPUs. Comparing Table 4 and Table 5 we see that the C2070 provides a 3.5× and 2.5× speed-up in SP and DP respectively. These results suggests competitive performance by the GPUs for this class of applications at a production level.

## 7. Conclusions

The OP2 "active" library framework, for the development of unstructured mesh applications, aims to decouple the scientific specification of an application from its parallel implementation to achieve code longevity and near-optimal performance through re-targeting the back-end to different hardware. This paper presented key design features of the current OP2 library for generating efficient code to be executed on contemporary GPUs.

Unlike in structured meshes, in unstructured meshes connectivity information is needed to specify the mesh topology. The majority of the solution is then spent in looping over indirect data. Thus two important design considerations is the data layout for efficient indirect data access and managing data dependencies during these accesses. This paper presented OP2's design to address these key issues and quantified the performance impact of our choices during the execution of a representative industrial CFD application (Airfoil) written using the OP2 API. Benchmarked systems consisted of two NVIDIA Fermi based GPUs - a GTX560Ti and a Tesla C2070.

Results show that utilizing an array of structs (AoS) data layout, as opposed to the struct of arrays data layout (SoA) reduces (in some cases by over 50%) the total amount of memory transferred to and from GPU global memory. We believe that these design insights and their relative performance merits will be applicable not only to cache-based GPUs but also to emerging CPU architectures as they become more SIMD-like with longer vector units.

We also investigated the performance trends due to selecting different mini-partition and thread-block sizes. We see that the scarce resources on the GPUs and their conflicting utilizations have significantly narrowed the "optimum spot" of performance on these devices. Guessing the right combination is not entirely straightforward and a bad set of selected values can lead to significantly sub-optimal performance. To obtain the best configurations we developed a novel auto-tuning framework and applied it to explore key performance aspects of the Airfoil code.

Comparing the auto-tuned runtime of the Airfoil code on the C2070 with the equivalent auto-tuned runtime of the OpenMP version of the code on two 6-core Intel Westmere processors show that the GPU executes 3.5× and 2.5× faster in single precision and double precision arithmetic respectively. These results suggest competitive performance by the GPUs for this class of applications at a production level.

The OP2 framework is currently being extended to support execution on distributed memory clusters based on MPI. GPU clusters using MPI across nodes and CUDA (or OpenCL) within GPUs will be supported as well as single and multi-threaded CPU clusters using MPI and OpenMP. The global mesh will be partitioned using standard graph partitioning techniques among the compute

nodes of a cluster, and import/export halos will be constructed for MPI message-passing. The respective CUDA (and in the future OpenCL) and OpenMP back-ends will be utilized for parallelizing the execution within each MPI process. Additionally extensions to the library will include AVX vectorization for future multi-processors supporting longer vector units for higher performance.

With reference to methods that are useful for obtaining a near-optimal configuration, such as auto-tuning used in this paper, we note that in the future it may be possible to recall previous configurations to provide improved starting points which should reduce the time for the auto tuning. Additionally techniques of machine learning and performance modeling could also be employed.

The full OP2 source, the Airfoil test case code and the auto-tuning framework are available as open source software [16] and the developers would welcome new participants in the OP2 project.

### Acknowledgements

Funding for this research has come from the UK Technology Strategy Board and Rolls-Royce plc. through the Siloet project, and the UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on Multi-layered Abstractions for PDEs. We are thankful to Leigh Lapworth, Yoon Ho and David Radford at Rolls-Royce, Paul Kelly, Graham Markall, David Ham and Florian Rathgeber, at Imperial College London, Jamil Appa and Pierre Moinier at BAE Systems, Tom Bradley at NVIDIA and Nick Hills at the University of Surrey for their contributions to the OP2 project.

### References

[1] Intel Xeon Processor X5650 specifications, 2010. http://ark.intel.com/Product.aspx?id=47922.

[2] NVIDIA CUDA C Programming Guide, 2010. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf.

[3] NVIDIA Tesla C2050 / C2070 GPU Computing Processor, 2010. http://www.nvidia.com/object/product_tesla_C2050_C2070_us.html.

[4] Intel Xeon X5650 processor, 2011. http://www.cpu-world.com/sspec/SL/SLBV3.html.

[5] The ROSE Compiler, 2011. http://www.rosecompiler.org/.

[6] Scala - general purpose programming language, 2011. http://www.scala-lang.org/.

[7] V.G. Asouti, X.S. Trompoukis, I.C. Kampolis, K.C. Giannakoglou, Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on graphics processing units, International Journal for Numerical Methods in Fluids 67 (2011) 232–246.

[8] D.A. Burgess, P.I. Crumpton, M.B. Giles, A parallel framework for unstructured grid solvers, in: S. Wagner, E. Hirschel, J. Periaux, R. Piva (Eds.), Proceedings of the Second European Computational Fluid Dynamics Conference, John Wiley and Sons, Stuttgart, Germany, 1994, pp. 391–396.

[9] A. Corrigan, F. Camelli, R. Löhner, F. Mut, Semi-automatic porting of a large-scale fortran cfd code to gpus, International Journal for Numerical Methods in Fluids 69 (2012) 314–331.

[10] A. Corrigan, F. Camelli, R. Löhner, J. Wallin, Running unstructured grid-based CFD solvers on modern graphics hardware, in: 19th AIAA Computational Fluid Dynamics Conference, Grand Hyatt Hotel, San Antonio, Texas, pp. 1–11.

[11] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E. Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, P. Hanrahan, Liszt: a domain specific language for building portable mesh-based pde solvers, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 9:1–9:12.

[12] M.B. Giles, OP2 Developer's Manual, 2011. http://people.maths.ox.ac.uk/gilesm/op2/dev.pdf.

[13] M.B. Giles, OP2 User's Manual, 2011. http://people.maths.ox.ac.uk/gilesm/op2/user.pdf.

[14] M.B. Giles, M.C. Duta, J.D. Muller, N.A. Pierce, Algorithm developments for discrete ad-joint methods, AIAA Journal 42 (2003) 198–205.

[15] M.B. Giles, D. Ghate, M.C. Duta, Using automatic differentiation for adjoint CFD code development, Computational Fluid Dynamics Journal 16 (2008) 434–443.

[16] M.B. Giles, G.R. Mudalige, OP2 for Many-Core Platforms, 2011. http://www.oerc.ox.ac.uk/research/op2.

[17] M.B. Giles, G.R. Mudalige, Z. Sharif, G. Markall, P.H. Kelly, Performance analysis of the OP2 framework on many-core architectures, SIGMETRICS Perform. Eval. Rev. 38 (2011) 9–15.

[18] L.W. Howes, A. Lokhmotov, A.F. Donaldson, P.H.J. Kelly, Deriving efficient data movement from decoupled access/execute specifications, in: A. Seznec, J. Emer, M. O'Boyle, M. Martonosi, T. Ungerer (Eds.), High Performance Embedded Architectures and Compilers, volume 5409 of *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, 2009, pp. 168–182.

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