Under the Hood of SYCL – An Initial Performance Analysis With an Unstructured-mesh CFD Application

Istvan Z. Reguly[1,2], Andrew M.B. Owenson[2], Archie Powell[2], Stephen A. Jarvis[3], and Gihan R. Mudalige[2]

[1] Faculty of Information Technology and Bionics, Pazmany Peter Catholic University, Budapest, Hungary
reguly.istvan@itk.ppke.hu
[2] University of Warwick, Coventry, UK
{a.m.b.owenson, a.powell.3, g.mudalige}@warwick.ac.uk
[3] University of Birmingham, Birmingham, UK
s.a.jarvis@bham.ac.uk

Abstract. As the computing hardware landscape gets more diverse, and the complexity of hardware grows, the need for a general purpose parallel programming model capable of developing (performance) portable codes have become highly attractive. Intel's OneAPI suite, which is based on the SYCL standard aims to fill this gap using a modern C++ API. In this paper, we use SYCL to parallelize MG-CFD, an unstructured-mesh computational fluid dynamics (CFD) code, to explore current performance of SYCL. The code is benchmarked on several modern processor systems from Intel (including CPUs and the latest Xe LP GPU), AMD, ARM and Nvidia, making use of a variety of current SYCL compilers, with a particular focus on OneAPI and how it maps to Intel's CPU and GPU architectures. We compare performance with other parallelisations available in OP2, including SIMD, OpenMP, MPI and CUDA. The results are mixed; the performance of this class of applications, when parallelized with SYCL, highly depends on the target architecture and the compiler, but in many cases comes close to the performance of currently prevalent parallel programming models. However, it still requires different parallelization strategies or code-paths be written for different hardware to obtain the best performance.

1 Introduction

With the switch to multi-core processors in 2004, the underpinning expectation of commercial hardware developers and vendors has been that performance improvements of applications could be maintained at historical rates by exploiting the increasing levels of parallelism of emerging devices. However, a key barrier that has become increasingly significant is the difficulty in programming them. The hardware architectures have become highly complex with massively parallel and heterogeneous processors, deep and multiple memory hierarchies and complex interconnects. Consequently extensive parallel programming knowledge is required to fully exploit the potential of these devices.

A wide range of parallel programming models, extensions and standards have been introduced to address this problem. Over the years these have included proprietary extensions such as CUDA, TBB, Cilk and OpenACC as well as evolving open standards such as OpenMP, OpenCL, and MPI. However, as observed by David Patterson in 2010 [17], industry, academia and stakeholders of HPC have still not been able to provide an acceptable and agile software solution for exploiting the rapidly changing, massively parallel diverse hardware landscape. On the one hand, open standards have been slow to catch up with supporting new hardware, and for many real applications have not provided the best performance achievable from these devices. On the other hand, proprietary solutions have only targeted narrow vendor-specific devices resulting in a proliferation of parallel programming models and technologies. As a result, we have seen and continue to see a golden age of parallel programming software research. A primary target of most such research has been achieving performance portability, where software techniques and methods are developed to enable an application to achieve efficient execution across a wide range of HPC architectures without significant manual modifications.

The most recent addition to the myriad array of parallel programming technologies and software suites is Intel's OneAPI. The need for a single application programming interface (API) to program their divergent hardware products – the currently dominant Xeon multi-core CPUs, recently announced Xe GPUs and Intel's FPGA devices – is driving this development. OneAPI is based on SYCL [2], a C++ abstraction layer for programming parallel systems, initially based on OpenCL, but as of the latest version of the standard, decoupled from it [20] to allow for different backends (e.g. CUDA, OpenMP). With the advent of OneAPI and the emerging vendor support for SYCL, it has been touted as one possible open standard for addressing the HPC performance portability problem. As such the objective of the research presented in this paper is to explore the performance of SYCL with a view to evaluate its performance portability, contrasting achieved performance to more established programming models on a range of modern multi-core and many-core devices.

We carry out this work building on the OP2 Domain Specific Language(DSL) [14], which already has wide-ranging capabilities to target modern architectures. OP2 uses source-to-source translation and automatic code generation to produce multiple parallellizations of an application written using the OP2 high-level API. It is currently able to generate parallel code that use SIMD, OpenMP, CUDA and their combinations with MPI together with different optimizations for each version to obtain the best performance from different hardware. In this work we extend these capabilities to also rapidly generate different variants of highly optimized SYCL code and apply it to a recently developed, representative unstructured-mesh CFD application [16] that is written with the OP2 API. We generate SYCL paralleizations for this application, and explore its performance, allowing for a fair and direct comparison of performance, including comparisons with other parallelisations generated through OP2. The work aims to provide a preliminary performance evaluation using current state-of-the-art SYCL. More specifically we make the following contributions:

- We explore how an implementation of the unstructured-mesh parallel motif can be achieved using the SYCL programming model. The main aspect for efficient parallelization is on handling the race-conditions of indirect array increments/updates which we do through coloring and atomics schemes implemented with SYCL.

- The SYCL parallelization is used to develop a new target source generator for OP2. This is used to automatically generate optimized SYCL code for a representative CFD application called MG-CFD. Performance of the SYCL-based MG-CFD parallelization is benchmarked on a range of single-node hardware platforms and compared to the same application parallelized through OP2 using currently established programming models, including SIMD, OpenMP, CUDA and their combinations with MPI.
- Finally, we present a detailed performance analysis of all the parallelizations explored above, contrasting the SYCL implementation with other parallelizations.

The use of an unstructured mesh application, which is characterized by their indirect memory accesses leads to an interesting benchmarking study as such an irregular motif is difficult to parallelize. This we believe will provide a more contrasting evaluation of SYCL, complementing previous work [9] on regular parallel motifs such as structured-mesh applications. Furthermore, the use of OP2's source-to-source translator to automatically produce SYCL parallelizations enabled us to rapidly explore the design space and various optimizations without needing to manually modify MG-CFD's 25 loops. We also show that the use of OP2 does not impact the best achievable performance from SYCL for this application. Given the range of modern and emerging multi-core and many-core architectures benchmarked, the different parallelizations explored for each, together with the use of multiple SYCL compilers, makes this study, to our knowledge, the most comprehensive performance investigation into a non-trivial, representative application developed with SYCL to-date.

Details of OP2's performance and portability for existing parallelizations along with the benefits and limitations of such a DSL-based approach have been extensively studied and presented in previous publications [15, 19, 11, 18]. As such we focus on the performance portability of SYCL. As this work uses OP2, we also do not draw conclusions with respect to the usability and maintainability of SYCL, as it is fully hidden from the users of the OP2 library.

The rest of this paper is organized as follows: in Section 2 we present an introduction to unstructured mesh applications and the key challenges in parallelizing this class of applications. Next, in Section 3 we briefly detail the OP2 API and the target SYCL parallelizations developed for subsequent code-generation through OP2. In Section 4 we present empirical performance results of our main benchmark application MG-CFD, parallelized using SYCL, compared to other parallelizations generated with OP2. In Section 5 we present a bottleneck analysus of the systems benchmarked and the achievable performance of each parallelization. Finally, conclusions are presented in Section 6.

2 Parallelizing Unstructured-mesh Applications

The key characteristic of the unstructured-mesh motif is the use of explicit connectivity information between elements to specify the mesh topology and consequently to access data defined on neighboring elements [7]. This is in contrast to the use of stencils in structured-mesh applications where the regular geometry of the mesh implicitly provides the connectivity information. As such, iterations over unstructured meshes lead to highly irregular patterns of data accesses over the mesh, due to indirections. For example, computations over the mesh involve iterating over elements of a set (e.g. faces), performing the same computations, on different data, accessing/modifying data on the set which they operate on (e.g. fluxes defined on the faces), or, using indirections accessing/modifying data defined on other sets (such as data on connected nodes). These indirect accesses are particularly difficult to parallelize. For example, parallelizing a loop over mesh edges, updating data on each of the two nodes connected to an edge will lead to multiple edges updating the same nodal data simultaneously, unless explicitly handled by the programmer.

Several strategies exists for handling data races depending on the target hardware and parallel programming model. SIMD vectorization on CPUs parallelize the iterative loop over the mesh elements, stepping through it in strides of the SIMD vector length of the processor. On a processor such as the current generation Intel Xeon – Skylake or Cascade Lake processors this will be a vector length of 8 with double precision arithmetic. Thus the computation over edges will proceed by computing over 8 edges simultaneously at a time, updating values on the two nodes connected to each edge. One way to handle data races within each step is to implement explicit gather-scatter operations to apply the indirect increments [15]. A gather will collect indirectly accessed data, to a local SIMD-length sized array, then carrying out a computation as SIMD-vector operations on this local data. Finally a scatter will serially apply the increments to the indirectly accessed data.

For multi-threading on CPUs, the parallelization should make sure that multiple edges assigned to threads do not update the same node simultaneously. With an OpenMP parallelization, one way to avoid data races is to color the edges such than no two edges of the same color update the same node [14]. Coloring can be similarly used for parallelizing on GPUs. Given the larger number of threads executable on GPUs, and the availability of GPU shared memory, different variations of coloring can be used [21]. For distributed memory parallelizations, such as using MPI, explicitly partitioning the mesh and assigning them to different processors leads to a decomposition of work that only have the potential to overlap at the boundaries of the partitions. An owner compute model with redundant computation can be used in this case to handle data races [14]. Other strategies published for parallelizing unstructured-mesh applications have included the use of a large temporary array [1] and atomics [21]. Using a large temporary array entails storing the indirect increments for the nodes in a staging array, during the edge loop, for example, and then a separate iteration over the nodes to apply the increments from the temporary array on to the nodal data. Atomics on the other hand simply allow for updates to be done one increment at a time with the use of hardware-locks.

3 SYCL Parallelizations with OP2

The best performance we have observed with multi-threading on CPUs and SIMT on GPUs has been through the use of coloring and atomics, respectively. As such, for the SYCL implementation, we solely explore these strategies as appropriate to the target hardware. To ease the development of multiple parallelizations with a range of different optimizations, we make use of the OP2 DSL [4, 14].

```
/* ----- elemental kernel function ----------------*/
void res(const double *edge,
         double *cell0, double *cell1 ){
  //Computations, such as:
  cell0 += *edge; *cell1 += *edge;
}

/* ----- main program -----------------------------*/
// Declaring the mesh with OP2
// sets
op_set edges = op_decl_set(numedge, "edges");
op_set cells = op_decl_set(numcell, "cells");
// mppings -connectivity between sets
op_map edge2cell = op_decl_map(edges, cells,
                   2, etoc_mapdata,"edge2cell");
// data on sets
op_dat p_edge = op_decl_dat(edges,
                1,"double",edata,"p_edge");
op_dat p_cell = op_decl_dat(cells,
                4,"double",cdata,"p_cell");

// OP2 parallel loop declaration
op_par_loop(res,"res", edges,
  op_arg_dat(p_edge,-1,OP_ID    ,4,"double",OP_READ),
  op_arg_dat(p_cell, 0,edge2cell,4,"double",OP_INC ),
  op_arg_dat(p_cell, 1,edge2cell,4,"double",OP_INC));
```

Fig. 1. Specification of an OP2 parallel loop

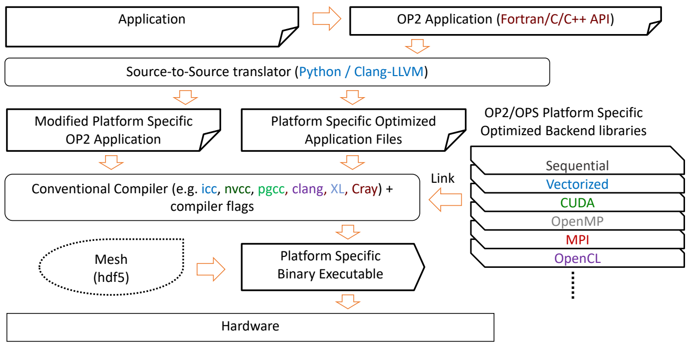
Application
OP2 Application (Fortran/C/C++ API)
Source-to-Source translator (Python / Clang-LLVM)
Modified Platform Specific OP2 Application
Platform Specific Optimized Application Files
OP2/OPS Platform Specific Optimized Backend libraries
Sequential
Vectorized
CUDA
OpenMP
MPI
OpenCL
Link
Conventional Compiler (e.g. icc, nvcc, pgcc, clang, XL, Cray) + compiler flags
Mesh (hdf5)
Platform Specific Binary Executable
Hardware

Fig. 2. Developing an Application with OP2

OP2 allows to define the unstructured-mesh problem in four abstract components: (1) sets (e.g. nodes, edges, triangular faces, quadrilateral faces), (2) data on sets (e.g. node coordinates, edge weights, cell fluxes), (3) explicit connectivity (or mapping) between the sets and (4) operations over sets declared as kernels iterating over each element of the set, accessing indirectly via mappings. A simple example illustrating the OP2 API is presented in Fig.1. The loop is over the set of edges, carrying out the computation per edge defined by the function `res`, accessing the data on edges, `p_edge`, directly and updating the data held on the two cells, `p_cell`, adjacent to an edge, indirectly via the mapping `edge2cell`. The `op_arg_dat` provides details of how an `op_dat`'s data is accessed in the loop. Its first argument is the `op_dat`, followed by its indirection index, `op_map` used to access the data indirectly, arity of the data in the `op_dat` and the type of the data. The final argument is the access mode of the data, read only, increment and others (such as read/write and write only, not shown here). The `op_par_loop` call contains all the necessary information about the computational loop

```
op_par_loop(compute_flux_edge_kernel,
  "compute_flux_edge_kernel", op_edges,
  op_arg_dat(vars,0,en,5,"double",OP_READ),
  op_arg_dat(vars,1,en,5,"double",OP_READ),
  op_arg_dat(edwgts,-1,OP_ID,3,"double",OP_READ),
  op_arg_dat(fluxes,0,en,5,"double",OP_INC),
  op_arg_dat(fluxes,1,en,5,"double",OP_INC));
```

Fig. 3. MG-CFD's `compute_flux_edge_kernel` loop

to perform the parallelization. It is clear that due to the abstraction, the parallelization depends only on a handful of parameters such as the existence of indirectly accessed data or reductions in the loop, plus the data access modes that lends to optimizations.

Parsing a code written in the above API, OP2's automatic code generator can produce a wide range of parallelizations following the development flow illustrated in Fig.2. When generating a platform specific parallel implementation of a loop specified by an `op_par_loop`, the code generator, essentially a source-to-source translator, makes use of a base template (or skeleton) that has been hand-crafted by the OP2 developers. Each of these skeletons make use of the best optimizations and platform specific configurations for the target hardware and programming models. For example, to produce the CUDA implementation, the OP2 code generator simply modifies the appropriate CUDA skeleton with the declared parameters of the `op_par_loop` to produce the concrete implementation of the loop [6]. Given the different parallelization strategies and optimizations that can be used even when using a single parallel programming model, different skeletons are maintained and reused together with configuration flags for further customizing what code they will generate. However, for a domain scientist developing an unstructured-mesh application, the process of generating a concrete parallel implementation will be automatic.

In this work, we formulate the unstructured-mesh problem using OP2's domain-specific abstraction and then extend its automatic code generation tools to produce an optimized SYCL parallelization. Given the range of hardware and SYCL compilers available, multiple parallelization strategies were investigated to ascertain the best performance. On CPUs, we found that a hierarchical coloring strategy (with 1 work item per workgroup) to apply the indirect increments produced the best performance. On the NVIDIA GPUs benchmarked, the best performance with SYCL was achieved with atomics. However, a SYCL compiler exposing non-standard atomics (fp64) is required to take advantage of the hardware support available on these many-core devices.

3.1 Coloring

As noted before, coloring can be applied to resolve the data races in unstructured-mesh computations. This parallelization strategy can be generally implemented on any shared memory multi-threaded system, including CPUs and GPUs without any restrictions due to hardware capabilities. Different variations of coloring have been implemented within OP2 as detailed in previous works [21]. Figure 4 details an excerpt of the SYCL code generated by OP2 for the most time-consuming parallel loop `compute_flux_edge_kernel` in MG-CFD. The OP2 API declaration of this loop is listed in Figure 3. This loop iterates over the set of mesh edges, `op_edges`, indirectly reading DP floating-point data held in the node-indexed array `vars`, using the mapping from edges to nodes, `en`;

```
/*Cast OP2 dats, maps and coloring plan as SYCL buffers*/
arg0_buffer = ... ; // vars - indirectly accessed
arg3_buffer = ... ; // fluxes - indirectly accessed
map0_buffer = ... ; // en - mapping table
arg2_buffer = ... ; // edwgts - directly accessed

col_reord_buffer = ... ; // coloring array

for ( int col=0; col<Plan->ncolors; col++){//for each color
  int start = Plan->col_offsets[0][col];
  int end = Plan->col_offsets[0][col+1];
  int nblocks = (end - start - 1)/nthread + 1;

  // enqueue arguments and elemental kernel
  op2_queue->submit([&](cl::sycl::handler& cgh) {
    ind_arg0 = (*arg0_buffer).. // enqueue vars
    ind_arg1 = (*arg3_buffer).. // enqueue fluxes
    opDat0Map = (*map0_buffer).. //enqueue mapping en
    arg2 = (*arg2_buffer).. // enqueue edwgts

    // enqueue coloring array
    col_reord = (*col_reord_buffer)..
    // enqueue any global constants used in the kernel
    ...
    //elemental kernel function as lambda - enqueue it
    auto compute_flux_edge_kernel_gpu = [=](
       const double *var_a, const double *var_b,
       const double *edwgts,double *fluxes_a,
       double *fluxes_b)
       { .../*body of kernel*/... };

    // setup kernel work items
    auto kern = [=](cl::sycl::item<1> item) {
      int tid = item.get_id(0);
      if (tid + start < end) {
        int n = col_reord[tid + start];
        int map0idx; int map1idx;

        // get the indirect index via mapping
        map0idx = opDat0Map[n+set_size*0];
        map1idx = opDat0Map[n+set_size*1];

        //user-supplied kernel call
        compute_flux_edge_kernel_gpu(
        &ind_arg0[map0idx*5], &ind_arg0[map1idx*5],
        &arg2[n*3],
        &ind_arg1[map0idx*5], &ind_arg1[map1idx*5]);
      }
    };
    // execute kernel
    cgh.parallel_for
    <class compute_flux_edge_kernel>
    (cl::sycl::range<1>(nthread * nblocks), kern);
  }); // end of enqueue arguments and elemental kernel
}
```

Fig. 4. Global coloring parallelization generated by OP2 for the `compute_flux_edge_kernel` loop in MG-CFD

it also directly reads DP floating-point data held on edges from array `edwgts`. The resulting flux contributions are indirectly incremented onto the output node-indexd array `fluxes`, again via the edges to nodes mapping.

There are two fundamental coloring schemes and execution strategies, which are illustrated in Figure 5. The simpler one, called *global coloring*, performs a single level of greedy coloring of set elements (in this case edges) based on a mapping (here edges

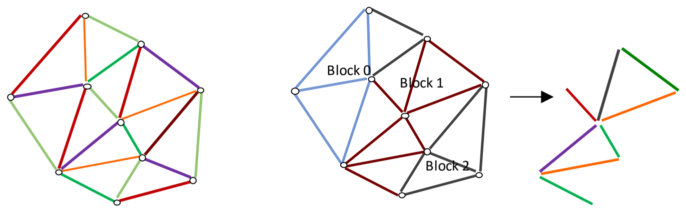

Fig. 5. Coloring strategies in OP2 – global and hierarchical coloring

to nodes). As the first drawing in Figure 5 shows, this gives us edge colors, where no two edges with the same color share a node. In terms of execution, edges with the same color can now be executed in parallel, with a synchronization between colors.

The second strategy, called *hierarchical coloring*, performs two levels of coloring. First, the mesh is split into blocks of edges, the blocks themselves are colored, so that no two blocks with the same color share any node. Second, edges within the block are greedily colored. In terms of execution, blocks of the same color can be executed in parallel, and within blocks there is further parallelism, so edges of the same color can be executed in parallel. This hierarchical scheme maps to architectures with hierarchical parallelism, for example blocks map to OpenMP threads or thread blocks in CUDA, and intra-block parallelism maps to vector units or CUDA threads. We map this hierarchical scheme to `nd_range` parallel for loops in SYCL.

The SYCL parallelization with global coloring starts by extracting the SYCL typed buffers from OP2's data structures (Figure 4, lines 1-5). The iteration set, in this case the mesh edges, has been colored by OP2, with coloring information stored in internal struct `Plan`. For SYCL execution, this coloring information is also stored in a SYCL integer buffer. An outer loop over colors initiates parallel execution across edges of the same color (line 9). Edge indices are held in the `col_reord` array, with edges of the same color stored consecutively. The current color determines the loop range `start` to `end`, read from `Plan->col_offsets`, determining which edges of `col_reord` to iterate through for that particular parallel execution.

Similar to the setup required for executing an OpenCL kernel, the arguments for the execution kernel, the kernel itself and any global constants referenced in the kernel are enqueued (lines 15-30). The kernel itself is specified as a lambda function (lines 25-30). Next, the SYCL kernel is set with flat parallelism, so that `nthread*nblocks` work items are launched (lines 51-53). The indirections are resolved by using the edge index `n` to access the indices held in the mapping table `opDat0Map` (lines 40-41). The elemental kernel is called with these indices, together with the directly accessed data as arguments (lines 44-47).

The advantage of global coloring is its simplicity – it can be easily expressed in any parallel programming environment. The main disadvantage with global coloring is the lack of data-reuse where multiple edges that write to the same mesh node have different color, and temporal locality is therefore poor. A further disadvantage is the low cache utilization where elements of the same color are not neighbors in the mesh, and therefore unlikely to be stored in consecutive memory locations.

The hierarchical coloring scheme maps well to GPU architectures, and in principle to CPU threads and vector units as well. However, the OpenMP-based implementations (hipSYCL) have a mismatch between the abstraction and the implementation, leading to poor performance; they need to launch one thread per work item when using two-level parallelism (nd_range). Intel's OneAPI compilers can optimise and map this better to hardware, yet despite achieving vectorization, as we show in Section 4, performance was poor. To address these issues, we implemented a variation of the hierarchical execution scheme in SYCL where each work group consists of a single work item, which then iterates through the edges in that block sequentially. This proved to perform better on all CPU platforms with all compilers. This implementation now matches the execution scheme used by OP2's OpenMP execution scheme. However, it prevents vectorization by construction.

All coloring based executions add a one-time setup cost to the total runtime for creating the colored execution scheme. For production applications that iterate over many cycles, this setup cost becomes negligible or indeed could be pre-computed if the mesh is known before runtime.

3.2 Atomics

In contrast to coloring, atomics-based parallelizations enable the indirect updates (i.e. increments) to be applied sequentially using hardware atomic operations. The disadvantage is that not all hardware has fast DP implementations of atomics, and that the SYCL 1.2.1 standard does not include them, however hipSYCL has support for them on NVIDIA GPUs. Figure 6 details an excerpt of the SYCL code generated by OP2 for `compute_flux_edge_kernel` loop using atomics targeting the hipSYCL compiler (which has support for DP atomics). This code is similar to the global coloring scheme for much of the setup. The `start` and `end` now points to the full iteration range over the mesh edges. The key difference with atomics is the use of local arrays `arg3_l` and `arg4_l` to hold the indirect increments and apply them using atomics (lines 44-53). This results in an extra 10 floating point operations per edge. In contrast to coloring schemes, there is no setup cost for coloring plan construction when using atomics.

4 Performance

In this section, we generate SYCL parallelizations with OP2 for MG-CFD [16]. MG-CFD is a 3D unstructured multigrid, finite-volume computational fluid dynamics (CFD) mini-app for inviscid-flow. Developed by extending the CFD solver in the Rodinia benchmark suite [8, 5], it implements a three-dimensional finite-volume discretization of the Euler equations for inviscid, compressible flow over an unstructured grid. It performs a sweep over edges to accumulate fluxes, implemented as a loop over all edges. Multi-grid support is implemented by augmenting the construction of the Euler solver presented in [8] with crude operators to transfer the state of the simulation between the levels of the multi-grid. Initially written as a standalone CPU only implementation [16], MG-CFD has now been converted to use the OP2 API. It is available as open-source software at [3]. This repository also contains the concrete parallel implementations generated through OP2 for SIMD, OpenMP, CUDA, OpenMP4.0, OpenACC and their combinations with MPI. The branch `feature/sycl` contains the generated SYCL versions of the application used in our performance investigation.

Our aim is to compare the performance of the SYCL implementations to that of other parallel versions for MG-CFD and explore how similar execution strategies can

```
/*Cast OP2 dats, maps and coloring plan as SYCL buffers*/
arg0_buffer = ... ; // vars - indirectly accessed
arg3_buffer = ... ; // fluxes - indirectly accessed
map0_buffer = ... ; // en - mapping table
arg2_buffer = ... ; // edwgts - directly accessed

if (end-start>0) {
  int nblocks = (end-start-1)/nthread+1;
  // enqueue arguments and elemental kernel
  op2_queue->submit([&](cl::sycl::handler& cgh) {

    ind_arg0 = (*arg0_buffer).. // enqueue vars
    ind_arg1 = (*arg3_buffer).. // enqueue fluxes
    opDat0Map = (*map0_buffer).. //enqueue mapping en
    arg2 = (*arg2_buffer).. // enqueue edwgts
    // enqueue any global constants used in the kernel
    ...
    //elemental kernel function as lambda
    auto compute_flux_edge_kernel_gpu = [=](...)
    {  .../*body of kernel*/... };

    // setup kernel work items
    auto kern = [=](cl::sycl::nd_item<1> item) {
      //local variables for holding indirect increments
      double arg3_l[5], arg4_l[5];
      for ( int d=0; d<5; d++ ){ arg3_l[d] = 0.0;}
      for ( int d=0; d<5; d++ ){ arg4_l[d] = 0.0;}
      int tid = item.get_global_linear_id();
      if (tid + start < end) {
        int n = tid+start;
        int map0idx; int map1idx;

        // get the indirect index via mapping
        map0idx = opDat0Map[n + set_size * 0];
        map1idx = opDat0Map[n + set_size * 1];

        //elemental kernel call
        compute_flux_edge_kernel_gpu(
        &ind_arg0[map0idx*5], &ind_arg0[map1idx*5],
        &arg2[n*3],
        arg3_l, arg4_l);

        //apply indirect increments using atomics
        {cl::sycl::atomic<double> a
        {cl::sycl::global_ptr<double>
        {&ind_arg1[0+map0idx*5]}};
        a.fetch_add(arg3_l[0]);}
         ...
         ...
        {cl::sycl::atomic<double> a
        {cl::sycl::global_ptr<double>
        {&ind_arg1[4+map1idx*5]}};
        a.fetch_add(arg4_l[4]);}
      }
    };
    // execute kernel
    cgh.parallel_for
    <class compute_flux_edge_kernel>
    (cl::sycl::nd_range<1>(nthread*nblocks,nthread),kern);
  }); //end of enqueue arguments and elemental kernel
}
```

Fig. 6. Atomics-based paralelization generated by OP2 for `compute_flux_edge_kernel` loop in MG-CFD

be expressed using SYCL. For benchmarking we make use of a number of systems based on currently prevalent and emerging processor architectures. A summary of the

Table 1. Benchmark systems specifications: GPUs

GPU	NVIDIA V100	NVIDIA A100	AMD Radeon VII	Intel Iris XE MAX
Bus protocol	PCI-e 3.0	SXM4	PCI-e 3.0	PCI-e 4.0
Cores	5120	6912	3840	768
Clock (MHz)	1245-1380	1410	1400-1750	300-1650
TFLOPS/s compute	7	9.7	3.46	2.53 (single)
Bandwidth (GB/s)	900	1600	1024	68
Measured BW (GB/s)	789.3	1268.7	668.2	53.5
Memory size (GB)	16	40	16	4
TDP (W)	250	400	300	25

Table 2. Benchmark systems specifications: CPUs

System	AWS c5d.24xlarge	AWS c5a.24xlarge	AWS c6g.16xlarge
Node Architecture	Intel Xeon Platinum 8275CL @ 3.00GHz (**Cascade Lake**)	AMD Epyc (**Rome**) 7R32 @ 3.20GHz	AWS **Graviton2** ARM v8.2 @ 2.5GHz
Procs × cores	2×24 (2 SMT/core)	1×48 (2 SMT/core)	1×64 (1 thread/core)
CPU Vector Length (Instructions)	512 bits (AVX-512)	256 bits (AVX-2)	128 bits (NEON)
Cache Hierarchy	32KB L1D/core, 1MB L2/core, 35.75MB L3/socket	32KB L1D/core 512KB L2/core 256MB L3/socket	64KB L1/core 1MB L2/core 32MB L3/socket
CPU Main Memory	192 GB	192 GB	128 GB
Measured BW (GB/s)	109.3 (per socket)	131.6	173.6
O/S	Ubuntu 20.04	Ubuntu 20.04	Ubuntu 20.04
TDP per CPU	~240 W	~220W	~100W

Table 3. Compilers and Compiler Flags

Compiler	Version	Compiler Flags
Intel OneAPI Compilers `icc, icpc, dpcpp`	Beta 9 Backend: OpenCL	`-O3 -xHOST -inline-forceinline -restrict -qopenmp|-fsycl`
`nvcc`	CUDA 10.2	`-O3 -restrict` V100 : `-gencode arch=compute_70,code=sm_70` A100 : `-gencode arch=compute_80,code=sm_80`
HipSYCL Compiler `syclcc-clang`	Based on Clang 9 Backend: OpenMP Backend: CUDA Backend: CUDA Backend: HIP	`-O3` All CPUs: `-hipsycl-platform=cpu` V100 : `-hipsycl-gpu-arch=sm_70` A100 : `-hipsycl-gpu-arch=sm_80` Radeon VII : `-hipsycl-gpu-arch=gfx906`
GNU `gcc,g++`	9.3	AMD Rome, AWS Graviton2 `-Ofast -fopenmp`

key specifications of these systems are detailed in Table 1 and Table 2.

We use the NVIDIA V100 and AMD Radeon VII GPUs of our local cluster. The A100 GPUs used were in AWS (Amazon Web Services) `p4d.24xlarge` instances, and the Iris XE MAX (based on XE LP architecture) GPUs were accessed through Intel's DevCloud. To benchmark CPUs, we have evaluated three high-end machines available through AWS instances. The `c5d.24xlarge` instance has a dual-socket Intel

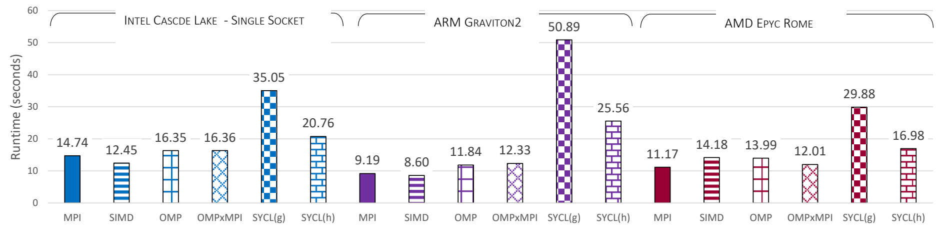

Fig. 7. MG-CFD, Runtime (seconds) on single socket CPUs – 8M edges, 25 MG Cycles

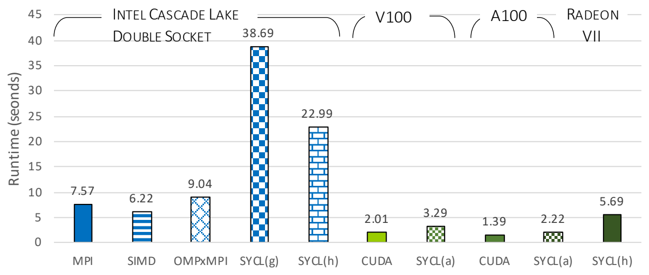

Fig. 8. MG-CFD, Runtime (seconds) on a dual socket CPUs and GPUs - 8M edges, 25 MG Cycles

Xeon Cascade Lake Platinum, each with 24 cores and 2 threads per core (SMT-2). The `c5a.24xlarge` has a single-socket AMD Epyc Rome with 48 physical cores and SMT-2. The `c6g.16xlarge` has a single-socket AWS Graviton2 ARM with 64 cores. While all of these were virtual machines, these contain some of the latest hardware architectures available, and achieve the same fraction of peak performance as our internal, less powerful, Intel-based bare-metal systems.

Figures 7 and 8 presents the runtime of the main time-marching loops of MG-CFD on the above systems, solving a NASA Rotor 37 [10] benchmark mesh consisting of 8 million edges on the finest level. The time to solution of 25 multi-grid cycles are reported here. The time for initial I/O (mesh loading and partitioning) are not included, given that these are a one-off setup cost and depends on other libraries such as HDF5, ParMetis/PTScotch etc., whose performance is not the subject of this paper. The figure presents the runtime of the application without any coloring plan time construction overheads, which was discussed in Section 3.1. Given these setup costs are one-off or indeed can be computed a priori if the mesh is known before runtime, we believe the figure provides a fairer comparison between the actual performance of each architecture and parallelization model. Compilers and compiler flags used for each version of the parallelizations are detailed in Table 3.

Reference performance numbers were collected using existing parallelizations in OP2: plain MPI which does not auto-vectorize, an MPI+SIMD version which uses explicit gathers and scatters to enable auto-vectorization, OpenMP that does not auto-vectorize, hybrid MPI+OpenMP, and for NVIDIA GPUs using CUDA with atomics.

4.1 CPU results

On the Intel Cascade Lake CPU, we used the OneAPI compilers to compile SYCL. The global colouring variant *(g)* uses a flat parallel for loop, however due to poor memory access patterns it performs the worst. To improve memory locality, we utilize the hierarchical execution scheme *(h)* – mapping blocks to threads as done in case of OpenMP, and optionally elements within blocks to vector lanes. The performance reported in Figure 7 (SYCL (h)) uses the same number of blocks as the OpenMP version and only uses a single work item per workgroup, which mirrors the behavior of our OpenMP execution scheme, and overall runtime is 26% slower. We also evaluated using 8 work items (AVX512 with FP64) and utilized Intel's subgroup extension to perform safe colored updates, however performance further degraded. An examination of the generated assembly for the most expensive loop provides two key insights : (1) computations did vectorize (when using more than 1 work item per workgroup), closely matching our SIMD variant, although no fused multiply-add instructions were generated, resulting in a 32% increase of floating point operations, and (2) the number of memory movement operations were significantly larger (approx. $2\times$).

On the ARM Graviton2 and AMD Epyc, we used the hipSYCL implementation, which uses OpenMP underneath. For flat parallel loops, hipSYCL will map computations to a flat OpenMP parallel loop, however when using hierarchical parallelism it will launch one thread per work item to guarantee that barriers can be handled correctly. The global coloring execution scheme clearly performs poorly, due to the lack of memory locality. For the hierarchical execution scheme, we used a single work item per workgroup, mirroring the OpenMP execution scheme. On the Graviton2 SYCL performance is $2.16\times$ worse than plain OpenMP – largely due to the relative immaturity of ARM support in Clang (used by hipSYCL) versus GNU g++ (used with flat OpenMP). On the AMD Epyc however, SYCL performs only 21% slower compared to plain OpenMP.

4.2 NVIDIA and AMD GPU results

For the comparison with GPUs, we also ran on both sockets of the Intel Cascade Lake CPU, and observed over 95% scaling efficiency for pure MPI and MPI+SIMD, though only 80% for MPI+OpenMP. SYCL however did not improve when running over both sockets due to NUMA issues – OP2 does not yet have an MPI+SYCL backend, which would address issue. For both NVIDIA and AMD GPUs, we utilized the automatic Array-of-Structs $\rightarrow$ Struct-of-Arrays data layout conversion feature of OP2. On NVIDIA GPUs, we used the atomics versions and compiled with hipSYCL – this showed a $60-64\%$ slowdown compared to the atomics version of CUDA. These differences are likely due to the immaturity of the hipSYCL compiler, resulting in higher register pressure and lower occupancy. The AMD Radeon VII GPU does not have hardware support for double precision atomics, and therefore we utilized the hierarchical coloring execution scheme with 128 work items per workgroup. OP2 does not have support for either HIP or OpenCL, therefore we could not compare this to a reference implementation.

4.3 Intel Iris XE MAX performance

To further explore OneAPI, we have evaluated the recently released Intel datacenter GPU built on the XE LP (low-power) platform. As the specifications in Table 1 show, this is a GPU in an entirely different class to the others tested. It has a $10-16\times$ lower TDP, and a similarly lower maximum bandwidth (53 GB/s measured), yet a relatively high maximum computational throughput – though it has to be noted that the card does not support double precision. This makes the platform have the highest ratio of FLOPS to bandwidth among all the tested hardware.

We prepared a single precision version of MG-CFD to evaluate performance – without considering the implications on accuracy and convergence at the moment. These GPUs also do not support single precision atomics, therefore we compared the various colored execution schemes. Intel's GPUs also support the subgroups extension of SYCL, and indeed are vital for good performance. We found the best performing combination is the hierarchical coloring execution scheme with 16 work items per workgroup, and 16 work items per subgroup (i.e. one subgroup per workgroup), and relied on subgroup barriers to perform the colored updates. The automatic AoS to SoA data layout conversion did not improve performance on this GPU. The best runtime was 22.37 seconds – for context, we compared performance to a single-socket Platinum 8256 CPU (4 cores, 51 GB/s STREAM bandwidth), which ran MPI+SIMD in 21.62 seconds and pure OpenMP in 42.37.

5 Bottleneck Analysis

To gather further insight into the performance profile of the application, we selected the most time-consuming kernel (`compute_flux_edge`), responsible for over 50% of total runtime, to carry out a bottleneck analysis. This kernel operates on edges, for each performing about 150 floating point instructions, reading 5 values from each node, 3 values from the edge, then indirectly incrementing 5 further values on each node. Due to the indirect accesses it is not a trivially vectorizable kernel, and it is highly sensitive to data reuse. For each platform we collected the relevant compute (GFLOPS/sec) and bandwidth (GB/s) measures onto the rooflines of each platform using the Berkeley Empirical Roofline Tool [12] and the STREAM [13] benchmark.

Table 4. Floating point operations per edge with different compilers

	Scalar		SIMD			CUDA
	Intel	ARM/AMD	AVX 512	AVX	ARM	
FLOPs/edge	150	165	216	165	164	323

To calculate the operation counts (FLOPs) per edge, we inspected the assembly generated for the computation of each edge for different implementations (see Table 4, which shows operations, not instructions). There are over 150 floating point operations per edge (with minor variations between compilers), and 13 of these are `sqrt` and `div` operations. It is important to note here that on CPUs there are separate assembly instructions for division and square root operations (though with much lower throughput than multiply or add), whereas on NVIDIA GPUs, these are mapped to a sequence of multiplications and additions – hence the much higher FLOPS per edge for CUDA shown in Table 4. Furthermore, depending on the compiler and the kinds of vectors used (scalar, AVX or AVX512) we get different FLOP counts; AVX generates precise divisions (single instruction), whereas AVX512 generates approximate reciprocals followed by additional multiply and add operations. With SIMD and hierarchical execution schemes, we stage increments in a local array, then apply them separately, adding a further 10 add instructions per edge. Finally, as reported before, Intel's SYCL version does not use FMA instructions, therefore even though the number of floating point operations is the same, the number of instructions is 32% higher.

The achieved computational throughput of `compute_flux` is shown in Table 5, with the highest fraction of peak achieved on the V100 GPU at 26%. While the maximum throughput is not representative particularly on CPUs due to the long-latency sqrt and division instructions, it is nevertheless clear that operational throughput – particularly for vectorized and GPU versions - is not a bottleneck. On CPU architectures, it is on the ARM platform, where the highest fraction of peak is achieved: 22% with the MPI+SIMD variant.

Table 5. Achieved computational throughput (GFLOPS/sec) of `compute_flux`

	Intel CSX	AMD EPYC	ARM Graviton2	NVIDIA V100	NVIDIA A100	AMD Radeon VII
Max compute	1150	1421	845.2	6953	9539	3301
MPI	101	139	171			
MPI+SIMD	215	104	190			
OpenMP	98	117	138			
SYCL	74	88	67	1140	1269	517
CUDA				1836	2480	

Table 6. Amount of data moved (in GB) from/to off-chip RAM with various parallelizations

MPI	OpenMP hierarchical	SYCL global	SYCL hierarchical	CUDA atomics	AMD SYCL hierarchical
448	778	2856	818	381	1190

To calculate the amount of data moved, and to determine how much `compute_flux` is bound by available memory bandwidth, we have created a stripped-down version of the kernel with negligible compute operations, but the same memory accesses, called `unstructured_stream`. Then, we instrumented this `unstructured_stream` kernel using LIKWID [22], and used the `MEM` performance counter group to determine the amount of data that is read from and written to off-chip DRAM. For the GPU architectures, we used NVIDIA Nsight Compute tool and ROCm's `rocprof` tool to gather the same information. The collected results are shown in Table 6, and it highlights a very important aspect of unstructured mesh computations: the execution method used to avoid race conditions has enormous implications in terms of the amount of data moved, and consecutively performance.

1. MPI – with distributed memory parallelism, each process iterates sequentially over the elements it owns, and the additional cost is in the explicit communications between processes.
2. Hierarchical coloring – when we break the edges into blocks, then color the blocks, and execute blocks of the same color in parallel, then by construction there will be

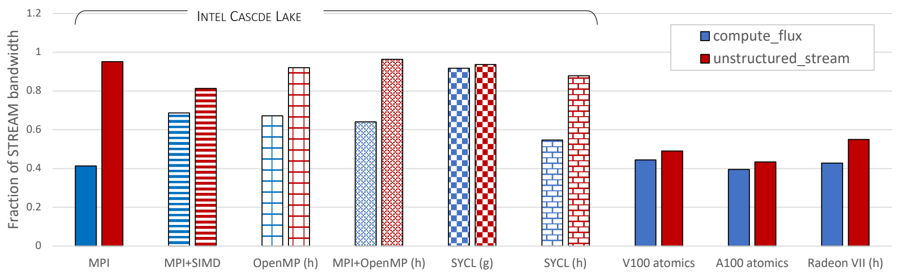

Fig. 9. Fraction of STREAM bandwidth achieved by unstructured_stream and compute_flux

no data reuse between blocks, but there will be reuse within blocks. On average 26 colors are required. With OpenMP and SYCL running on the CPU, we use blocks of size 2048, when running on the AMD GPU, we use a block size of 128. Correspondingly, these execution schemes move $1.73-2.65\times$ the amount of data compared to MPI.

3. Global coloring – when edges are colored based on potential race conditions, then all edges with the same color are executed in parallel, then there is no data reuse between edges by construction. On average 22.8 colors are required. This approach requires moving the most data; $6.25\times$ the amount compared to MPI.
4. CUDA Atomics – flat parallelism is used, and there is no need for explicit communication between the processes or thread as in the case of MPI. Therefore this approach incurs the least overhead in terms of data movement.

The performance of `unstructured_stream` can then be directly contrasted with STREAM; in Figure 9, we show the fraction of peak bandwidth (as measured by STREAM) achieved by both `unstructured_stream` and `compute_flux`. It is clear that on the Intel CPU, all parallelisations are bandwidth bound – MPI+SIMD is slower than the others due to the overhead of explicitly packing and unpacking registers. On the GPU platforms, performance is reported to be limited by L1/Texture cache throughput as well as atomics on NVIDIA and block synchronization overhead (required for colored updates) on AMD cards.

When comparing the achieved bandwidth of `compute_flux`, the performance differences of different parallelisations are seen from a different perspective; how much performance is bottlenecked by data movement, and how much computations are interleaved with data movement. The MPI and MPI+SIMD variants move the same amount of data, VTune reports 99.7% vector capacity usage for the computational body of the SIMD version, while the plain MPI version does not vectorize at all. Despite good vectorization with SIMD, there is poor overlap between computations and data movement, reducing its efficiency to 69%. When enabling Hyper-threading (which is shown in the results), performance of `compute_flux` is improved by 17% compared to only using 1 thread per core (but makes no difference for `unstructured_stream`), which supports the conclusion of poor overlap. This is even more obvious on the non-vectorized MPI version, where the cost of scalar computations reduces the efficiency of `compute_flux` to 41%, and is therefore just as much bound by the throughput of scalar operations.

The lack of vectorization is much less impactful on other parallelizations; even though neither OpenMP nor hierarchical SYCL versions vectorize (SYCL's vectorized hierarchical version performs even worse as discussed above), they still achieve over 55% of peak bandwidth – due to having to move $1.7 - 1.8\times$ more data. The lack of overlap between compute and data movement is responsible for the rest of the gap to STREAM. With global coloring, the SYCL implementation does vectorize, yet the cost of data movement dwarfs that of computations.

On the GPUs, only 43–62% of peak bandwidth is achieved by `unstructured stream`, but `compute_flux` also achieves 39–44% of peak as well. Computations and data movement is much better overlapped thanks to the massive parallelism, but while `unstructured_stream` achieves 100% occupancy, `compute_flux` only has 35%, leading to worse overlap.

6 Conclusion

The results shown indicate that the SYCL API brings comparable performance (within a factor of 1.3-2.0$\times$) overall for both CPUs and GPUs from different vendors in this application. The SYCL ecosystem is rapidly closing the performance gap with other parallel programming models. This is an essential quality of any new parallel API, so the fact that SYCL already achieves this shows that it is a good foundation for Intel's OneAPI software suite. In addition, as the standard is further developed, performance parity with other models is expected as software and hardware vendors optimise.

However, as with other portable parallelization approaches, there is still the need to write different parallelizations within the code to achieve the best runtimes. In the case of this unstructured mesh application, that entailed writing a coloring parallelization for CPUs and Radeon GPUs, and an atomics version for NVIDIA GPUs. Thus, the idea of SYCL abstracting away device specific code may not be entirely representative of real world use cases. This is especially true for irregular classes of applications, such as unstructured-mesh as opposed to the more commonly explored regular applications.

If this disparity continues, then it could lead to SYCL being seen as yet another industry standard, being grouped together with existing compute frameworks which offer similar levels of performance portability. For example, OpenMP is a far more mature standard which can also be written for all devices that SYCL currently supports, not to mention code-bases that do not use modern C++ (e.g. Fortran), which then cannot use SYCL. The DSL-based code generator used in this work, OP2, has been able to keep up with such changes by adding new code generators which can produce code based on emerging standards and models. However, for applications which are not based on frameworks and require a rewrite, developers could be hesitant to adopt SYCL for these reasons.

Nevertheless, SYCL is a major step forward, in that it presents a modern, succinct C++ API (in contrast to e.g. OpenCL), capable of targeting an impressively wide set of parallel architectures (in contrast to vendor-specific extensions, e.g. CUDA), that allows fine grained control over parallelism, and is reasonably capable of exposing low-level features of various architectures. Given the improving performance of compilers, we do recommend SYCL to application developers who want a unified parallel programming framework.

As a continuation of this work, we are developing multi-device and distributed memory support with MPI+SYCL in OP2, and we are evaluating performance with a range of further applications already using OP2. We also intend to explore the support for targeting FPGAs using SYCL.

Acknowledgment

References

1. Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Tech. Rep. LLNL-TR-490254
2. C++ Single-source Heterogeneous Programming for OpenCL (accessed 2019), `https://www.khronos.org/sycl/`
3. MG-CFD-OP2 GitHub Repository (accessed 2019), `https://github.com/warwick-hpsc/MG-CFD-app-OP2`
4. OP2 github repository (accessed 2019), `https://github.com/OP-DSL/OP2-Common`
5. Rodinia: Accelerating Compute-Intensive Applications with Accelerators (accessed 2019), `https://rodinia.cs.virginia.edu/`
6. Balogh, G., Mudalige, G., Reguly, I., Antao, S., Bertolli, C.: OP2-Clang: A Source-to-Source Translator Using Clang/LLVM LibTooling. In: 2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). pp. 59–70 (Nov 2018). https://doi.org/10.1109/LLVM-HPC.2018.8639205
7. Colella, P.: Defining Software Requirements for Scientific Computing (2004), (Presentation)
8. Corrigan, A., Camelli, F., R.Löhner, Wallin, J.: Running Unstructured Grid CFD Solvers on Modern Graphics Hardware. In: 19th AIAA Computational Fluid Dynamics Conference. No. AIAA 2009-4001 (June 2009)
9. Deakin, T., McIntosh-Smith, S.: Evaluating the Performance of HPC-Style SYCL Applications. In: Proceedings of the International Workshop on OpenCL. IWOCL '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3388333.3388643
10. Denton, J.: Lessons from rotor 37. Journal of Thermal Science **6**(1), 1–13 (1997)
11. Giles, M., Mudalige, G., Spencer, B., Bertolli, C., Reguly, I.: Designing OP2 for GPU architectures. Journal of Parallel and Distributed Computing **73**(11), 1451–1460 (2013). https://doi.org/https://doi.org/10.1016/j.jpdc.2012.07.008, `https://www.sciencedirect.com/science/article/pii/S0743731512001694`, novel architectures for high-performance computing
12. Lo, Y.J., Williams, S., Van Straalen, B., Ligocki, T.J., Cordery, M.J., Wright, N.J., Hall, M.W., Oliker, L.: Roofline model toolkit: A practical tool for architectural and program analysis. In: International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems. pp. 129–148. Springer (2014)
13. McCalpin, J.D.: Memory Bandwidth and Machine Balance in Current High Performance Computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (Dec 1995)
14. Mudalige, G., Giles, M., Reguly, I., Bertolli, C., Kelly, P.: OP2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures. 2012 Innovative Parallel Computing, InPar 2012 (2012). https://doi.org/10.1109/InPar.2012.6339594, `http://dx.doi.org/10.1109/InPar.2012.6339594`

15. Mudalige, G., Reguly, I., Giles, M.: Auto-vectorizing a Large-scale Production Unstructured-mesh CFD Application. In: Proceedings of the 3rd Workshop on Programming Models for SIMD/Vector Processing. pp. 5:1–5:8. WPMVP '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2870650.2870651, `http://doi.acm.org/10.1145/2870650.2870651`
16. Owenson, A., Wright, S., Bunt, R., Ho, Y., Street, M., Jarvis, S.: An Unstructured CFD Mini-Application for the Performance Prediction of a Production CFD Code. Concurrency Computat: Pract Exper (2019). https://doi.org/10.1002/cpe.5443, `https://doi.org/10.1002/cpe.5443`
17. Patterson, D.: The Trouble with Multi-Core. IEEE Spectrum **47**(7), 28–32, 53 (July 2010). https://doi.org/10.1109/MSPEC.2010.5491011
18. Reguly, I.Z., Giles, D., Gopinathan, D., Quivy, L., Beck, J.H., Giles, M.B., Guillas, S., Dias, F.: The VOLNA-OP2 tsunami code (version 1.5). Geoscientific Model Development **11**(11), 4621–4635 (2018). https://doi.org/10.5194/gmd-11-4621-2018, `https://gmd.copernicus.org/articles/11/4621/2018/`
19. Reguly, I.Z., Mudalige, G.R., Bertolli, C., Giles, M.B., Betts, A., Kelly, P.H.J., Radford, D.: Acceleration of a Full-Scale Industrial CFD Application with OP2. IEEE Transactions on Parallel and Distributed Systems **27**(5), 1265–1278 (2016). https://doi.org/10.1109/TPDS.2015.2453972
20. Reyes, R., Brown, G., Burns, R., Wong, M.: SYCL 2020: More than Meets the Eye. In: Proceedings of the International Workshop on OpenCL. IWOCL '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3388333.3388649, `https://doi.org/10.1145/3388333.3388649`
21. Sulyok, A., Balogh, G., Reguly, I., Mudalige, G.: Locality Optimized Unstructured Mesh Algorithms on GPUs. Journal of Parallel and Distributed Computing **134**, 50 – 64 (2019). https://doi.org/https://doi.org/10.1016/j.jpdc.2019.07.011
22. Treibig, J., Hager, G., Wellein, G.: LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In: 2010 39th International Conference on Parallel Processing Workshops. pp. 207–216 (2010). https://doi.org/10.1109/ICPPW.2010.38