Auto-Vectorizing a Large-scale Production Unstructured-mesh CFD Application

G.R. Mudalige
Oxford eResearch Centre,
University of Oxford,U.K.
gihan.mudalige@oerc.ox.ac.uk

I.Z. Reguly
Faculty of Information
Technology and Bionics
Pázmány Péter Catholic
University, Hungary
reguly.istvan@itk.ppke.hu

M.B. Giles
Oxford e-Research Centre,
University of Oxford, U.K.
mike.giles@maths.ox.ac.uk

ABSTRACT

For modern x86 based CPUs with increasingly longer vector lengths, achieving good vectorization has become very important for gaining higher performance. Using very explicit SIMD vector programming techniques has been shown to give near optimal performance, however they are difficult to implement for all classes of applications particularly ones with very irregular memory accesses and usually require considerable re-factorisation of the code. Vector intrinsics are also not available for languages such as Fortran which is still heavily used in large production applications. The alternative is to depend on compiler auto-vectorization which usually have been less effective in vectorizing codes with irregular memory access patterns. In this paper we present recent research exploring techniques to gain compiler auto-vectorization for unstructured mesh applications. A key contribution is details on software techniques that achieve auto-vectorisation for a large production grade unstructured mesh application from the CFD domain so as to benefit from the vector units on the latest Intel processors without a significant code re-write. We use code generation tools in the OP2 domain specific library to apply the auto-vectorising optimisations automatically to the production code base and further explore the performance of the application compared to the performance with other parallelisations such as on the latest NVIDIA GPUs. We see that there is considerable performance improvements with auto-vectorization. The most compute intensive parallel loops in the large CFD application shows speedups of nearly 40% on a 20 core Intel Haswell system compared to their non-vectorized versions. However not all loops gain due to vectorization where loops with less computational intensity lose performance due to the associated overheads.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]; C.1.2 [**Multiple Data Stream Architectures**]

WPMVP '16, March 13 2016, Barcelona, Spain

DOI: http://dx.doi.org/10.1145/2870650.2870651

Keywords

Automatic Vectorization, SIMD, OP2, Unstructured mesh, GPU

1. INTRODUCTION

The importance of achieving good vectorisation was not significant in previous generations of micro-processors. Vector lengths were short (128 bits or less) and compiler auto-vectorization was used to gain modest speedups if opportunities were present in the application code. Even then such compilers at best could only vectorize a few classes of applications that had very regular memory access and computation patterns such as from the structured-mesh or multimedia application domains. However, a significant portion of the capabilities of the latest processors depends on the utilization of their vector units. Particularly for modern x86 based CPUs with increasingly longer vector lengths, achieving good vectorization has become very important for gaining higher performance for *any* class of applications. This is even more significant on the next generation of Intel Skylake Xeon processors with 512 bit AVX vector units and novel accelerators such as the Intel Xeon Phi with the longest vector lengths currently found on any processor core [15].

In previous work [13] we presented research into gaining higher performance through vectorization on Intel CPUs and the Xeon Phi for unstructured-mesh applications. These applications have very irregular access patterns and are not easily vectorized. Using two unstructured mesh applications we explored different strategies to achieve good vectorization including (1) using vector intrinsics and (2) using OpenCL on CPUs, both of which can be executed on Intel CPUs and the Xeon Phi co-processor. Results showed that vectorization through the OpenCL SIMT model does not map efficiently to CPU vector units and had large threading overheads. Using SIMD vector intrinsics did result in efficient code and near-optimal performance but required very explicit and laborious manual hand-tuning. The optimisations also required a considerable re-write of the application. Such a re-write would be infeasible for a larger code base. As such, an open question remained as to whether automatic vectorisation (particularly with the recent advances in the Intel compiler suites) could be achieved for these applications through the compilers, to simplify development and if so whether it provides good performance.

In this paper we present recent research conducted to explore this open issue. A key contribution is to detail software techniques that achieve auto-vectorisation for a large pro-

duction grade unstructured mesh application so as to benefit from the vector units on the latest Intel processors without significant code re-writes. Following methods similar to that used in [13] we use code generation tools in the OP2 domain specific library [10] to apply the auto-vectorising optimisations automatically to this large application from the CFD domain. We further explore the performance of the auto-vectorized version of the code compared to the performance with other parallelisations generated with OP2 on the latest multi-core and many-core processors. The objective is to contrast the best performance that can be gained from two of the most popular parallel computing hardware platforms available today in light of the optimisations discovered from this research. Specifically we make the following contributions:

- We present the software strategies required to gain auto-vectorisation for unstructured mesh applications using the latest Intel C/C++ and Fortran compilers. Initially we explore auto-vectorisation with the Airfoil CFD benchmark. Airfoil was previously manually vectorized in [13], but we contrast that performance with the performance gained with auto-vectorisation. Results show how the auto-vectorized code is only about 4% slower than the best performance achieved with the previously developed vector intrinsics based application.

- The techniques developed with Airfoil are then applied to Hydra, a large-scale production CFD application used at Rolls-Royce plc., to explore if vectorization could be achieved and if so, to quantify the performance benefits (if any) of the optimisation. OP2's automatic code generation tools are used to apply the new optimisations to the production code base of over 50K LoC. We chart the issues encountered in this process including circumventing vectorisation inhibiting code in the production application.

- Finally, performance gained through vectorisation for the application is compared to that of other parallelisations achieved using OP2, particularly on GPUs. We present time to solution and achieved memory bandwidth performance for Hydra solving a standard unstructured mesh industrial problem called NASA Rotor 37 with 2.8 Million edges. Results show how the performance on the latest Intel CPUs, using vectorisation, matches that of the latest GPUs.

The rest of the paper is organised as follows: in Section 2 we present a brief summary of the unstructured-mesh applications used in this work and how OP2 is used for developing and parallelising these applications. In section 3 we present the key auto-vectorisation techniques explored in this paper with reference to both the C/C++ and Fortran based versions of the Airfoil CFD benchmark. Next, in Section 4 we attempt to apply the auto-vectorisation techniques to the Hydra CFD application from Rolls-Royce plc. Performance of key kernels in this application is benchmarked and contrasted with other parallelisations. Finally conclusions are drawn in Section 5.

2. BACKGROUND

Unstructured mesh applications, unlike structured mesh applications use explicit connectivity information to specify

```
1  call op_decl_set (nnode,nodes,'nodes' )
2  call op_decl_set (nedge,edges,'edges' )
3  call op_decl_set (ncell,cells,'cells' )
4
5  call op_decl_map (edges,nodes,2,edge,pedge,'pedge')
6  call op_decl_map (edges,cells,2,ecell,pecell,'pecell')
7
8  call op_decl_dat (nodes,2,'real(8)',x, p_x,'p_x' )
9  call op_decl_dat (cells,4,'real(8)',q, p_q,'p_q' )
10 call op_decl_dat (cells,1,'real(8)',adt,p_adt,'p_adt' )
11 call op_decl_dat (cells,4,'real(8)',res,p_res,'p_res' )
```

Figure 1: Example mesh declarations with the OP2 API

the mesh topology. They are used in a wide range of computational science and engineering applications, including the solution of partial differential equations (PDEs) in computational fluid dynamics (CFDs), structural mechanics, computational electro magnetics (CEM) and general finite element methods. In three dimensions, millions of elements are required for the desired solution accuracy leading to significant computational costs.

OP2 [8, 10] is a high-level framework designed to reduce the complexity of developing and parallelising unstructured mesh applications. The key idea is to describe the problem at a higher level with domain specific constructs using an API (embedded in C/C++ or Fortran) while leaving the parallel implementation to OP2. The parallel implementations are achieved by automated code generation techniques which transform the high-level problem declaration to specific parallel implementations. These currently include parallelisations for OpenMP, CUDA, OpenCL, OpenACC and their combinations with MPI. This enables high productivity, easy maintenance, and code longevity for the domain scientist, permitting them to focus on the scientific problem at hand and not the development of parallel code. At the same time, it enables the OP2 library developers to apply radical and platform-specific optimizations that help deliver near-optimal performance.

The OP2 abstraction involves breaking up the problem into four distinct parts: (1) sets, such as vertices or edges, (2) data on sets, such as coordinates and flow variables, (3) connectivity between sets, and (4) operations over sets. These form the OP2 API that can be used to fully and abstractly define any unstructured mesh problem. Unstructured mesh algorithms tend to iterate over different sets, accessing and modifying data indirectly on other sets via mappings; for example, flux computations often loop over edges accessing data on edges and neighbouring cells, updating flow variables indirectly on these cells. In a parallel setting, this leads to data races, the efficient handling of which is paramount for high performance.

Figure 1 and 2 illustrate OP2's Fortran API. Here, we have used the `op_set`s, `op_map`s and `op_dat`s that are involved in the `res_calc` loop of the Airfoil 2D non-linear finite volume benchmark. Airfoil solves the 2D Euler equations with scalar numerical dissipation and the algorithm iterates towards a steady state solution, in each iteration using a control volume approach. For example the rate at which the mass changes within a control volume is equal to the net flux of mass into the control volume across the four faces of a cell. In Figure 1 we see how the unstructured mesh is first declared with the sets involved in the computation – `op_set`s, the connectivity between the sets – `op_map`s and

```
1  SUBROUTINE res_calc(x1,x2,q1,q2,adt1,adt2,res1,res2)
2    IMPLICIT NONE
3    REAL(kind=8), DIMENSION(2), INTENT(IN) :: x1
4    REAL(kind=8), DIMENSION(2), INTENT(IN) :: x2
5    REAL(kind=8), DIMENSION(4), INTENT(IN) :: q1
6    REAL(kind=8), DIMENSION(4), INTENT(IN) :: q2
7    REAL(kind=8), INTENT(IN) :: adt1
8    REAL(kind=8), INTENT(IN) :: adt2
9    REAL(kind=8), DIMENSION(4) :: res1
10   REAL(kind=8), DIMENSION(4) :: res2
11   REAL(kind=8) :: dx,dy,mu,ri,p1,vol1,p2,vol2,f
12
13   !computations such as:
14   res(1) = res(1) + q1(1)*(x1(1) - x2(1))
15   ...
16 END SUBROUTINE
17
18 call op_par_loop_8 ( res_calc, edges, &
19 & op_arg_dat (p_x,  1, pedge,2,"real(8)",  OP_READ), &
20 & op_arg_dat (p_x,  2, pedge,2,"real(8)",  OP_READ), &
21 & op_arg_dat (p_q,  1, pecell,4,"real(8)", OP_READ), &
22 & op_arg_dat (p_q,  2, pecell,4,"real(8)", OP_READ), &
23 & op_arg_dat (p_adt,1, pecell,1,"real(8)", OP_READ), &
24 & op_arg_dat (p_adt,2, pecell,1,"real(8)", OP_READ), &
25 & op_arg_dat (p_res,1, pecell,4,"real(8)", OP_INC),  &
26 & op_arg_dat (p_res,2, pecell,4,"real(8)", OP_INC))
```

Figure 2: Declaring the `res_calc` loop with the OP2 API

the data associated with the sets – `op_dats`. In this case the `op_dec_map` API calls specifies the connectivity from the `op_sets` `edges` to `nodes` and `edges` to `cells` specified in the `edge` and `ecell` arrays respectively and returns two `op_maps` `pedge` and `pecell`. `op_dec_dat` specifies the double precision data defined on the given `ops_sets`, including their dimension (i.e. number of doubles per set element).

Figure 2 then demonstrates how a loop over the edges are specified in the `res_calc` loop (see full source on GitHub [11]). The `op_par_loop` call specifies a loop over a given set (`edges` in this case), executing a per-set-element kernel function (`res_calc`), which within OP2 is called a *user kernel*, on each element of the set passing in pointers to data based on arguments described as `op_arg_dats`. In an `op_arg_dat` we specify (1) the `op_dat`,(2) the index of the mapping used to access it, (3) the `op_map` to access the data if the `op_dat` is not specified on the `op_set` over which this loop is iterating over, (4) the type of the data and (5) a flag indicating the type of access, which can be one of read, write, increment, or read–write.

Figure 3 gives a simplified example of code generated to execute this computation using MPI on a distributed memory cluster of CPUs – note how information provided in the API is sufficient to generate this code. First, the host stub routine (`res_calc_host`, line 37) does the required MPI halo exchanges to update the boundary data held on an MPI process (line 41). Then a number of C to Fortran pointer conversions are done to bind the internal data structures held as C pointers in OP2 to Fortran pointers (e.g. bind the opArg1%data which is a C pointer to opDat1Local which is a Fortran pointer). This allows OP2 to use the per-element computation routine (`res_calc` in this case) without modification as a Fortran subroutine. The `op_wrap_res_calc` routine calls the elemental computation routine `res_calc` (lines 25-33) after setting up the indices used for indirectly accessing the data, via the mappings (lines 19-22).

In addition to the MPI parallelisation, our previous work on OP2 [8, 4, 3] shows how a combination of code gener-

```
1  SUBROUTINE op_wrap_res_calc(
2  & opDat1Local, opDat3Local, opDat5Local, opDat7Local, &
3  & opDat1Map, opDat1MapDim, opDat3Map, opDat3MapDim, &
4  & bottom,top)
5
6  real(8) opDat1Local(2,*)
7  real(8) opDat3Local(4,*)
8  real(8) opDat5Local(1,*)
9  real(8) opDat7Local(4,*)
10 INTEGER(kind=4) opDat1Map(*)
11 INTEGER(kind=4) opDat1MapDim
12 INTEGER(kind=4) opDat3Map(*)
13 INTEGER(kind=4) opDat3MapDim
14 INTEGER(kind=4) bottom,top,i1
15 INTEGER(kind=4) map1idx, map2idx, map3idx, map4idx
16
17 DO i1 = bottom, top-1, 1
18   ! set up the indirect access index
19   map1idx = opDat1Map(1 + i1 * opDat1MapDim + 0)+1
20   map2idx = opDat1Map(1 + i1 * opDat1MapDim + 1)+1
21   map3idx = opDat3Map(1 + i1 * opDat3MapDim + 0)+1
22   map4idx = opDat3Map(1 + i1 * opDat3MapDim + 1)+1
23   ! kernel call -- use the indirect index
24   ! to access the data in the ops_dat
25   CALL res_calc( &
26    & opDat1Local(1,map1idx), &
27    & opDat1Local(1,map2idx), &
28    & opDat3Local(1,map3idx), &
29    & opDat3Local(1,map4idx), &
30    & opDat5Local(1,map3idx), &
31    & opDat5Local(1,map4idx), &
32    & opDat7Local(1,map3idx), &
33    & opDat7Local(1,map4idx) )
34 END DO
35 END SUBROUTINE
36
37 SUBROUTINE res_calc_host( userSubroutine, set,
38 & opArg1, opArg2, opArg3, opArg4, opArg5, opArg6,
39 & opArg7, opArg8 )
40 ...
41 n_upper = op_mpi_halo_exchanges(...)
42 CALL op_mpi_wait_all(numberOfOpDats,opArgArray)
43
44 ! setup c to f pointers
45 CALL c_f_pointer(opArg1%data,opDat1Local,(/shape/))
46 CALL c_f_pointer(opArg1%map_data,opDat1Map,(/shape/))
47 ...
48 ...
49 CALL op_wrap_res_calc( &
50 & opDat1Local, opDat3Local, opDat5Local, opDat7Local, &
51 & opDat1Map, opDat1MapDim, opDat3Map, opDat3MapDim, &
52 & 0, n_upper)
53 ...
54 END SUBROUTINE
```

Figure 3: Code generated for MPI execution of `res_calc`

ation and run-time execution planning allows OP2 to generate a number of paralleisations that are able to run on a wide range of hardware platforms. At the highest level, we use distributed memory parallelization through message passing (MPI), where the mesh is split up into partitions using standard partitioners such as PT-Scotch [12], and an owner-compute approach is used in combination with halo exchanges to ensure correct execution. There are no potential race conditions, but redundant execution of certain set elements by different processes may be necessary. On multi-core CPUs, OpenMP threads are used and, through a pre-processing step, splits up the computational domain

```
#define SIMD_VEC 4
! user function -- modified for vectorisation
SUBROUTINE res_calc_vec(x1,x2,q1,q2,
           adt1,adt2,res1,res2,idx)
  !dir attributes vector :: res_calc_vec
  IMPLICIT NONE
  real(8), DIMENSION(SIMD_VEC,2), INTENT(IN) :: x1, X2
  real(8), DIMENSION(SIMD_VEC,4), INTENT(IN) :: q1, q2
  real(8), DIMENSION(SIMD_VEC,1), INTENT(IN) :: adt1, adt2
  real(8), DIMENSION(SIMD_VEC,4) :: res1, res2
  INTEGER(4) :: idx
  !computations such as:
  res(idx,1) = res(idx,1) + &
  &            q1(idx,1) * (x1(idx,1) - x2(idx,1))
END SUBROUTINE
SUBROUTINE op_wrap_res_calc( opDat1Loc, opDat3Loc, &
& opDat5Loc, opDat7Loc, opDat1Map, opDat1MapDim,
& opDat3Map, opDat3MapDim, bottom,top)
  ...
  real(8) dat1(SIMD_VEC,2), dat2(SIMD_VEC,2), &
  & dat3(SIMD_VEC,4), dat4(SIMD_VEC,4), dat5(SIMD_VEC,1),
  & dat6(SIMD_VEC,1), dat7(SIMD_VEC,4), dat8(SIMD_VEC,4)

  !loop SIMD_VEC number of iterations at a time
  DO i1 = bottom, ((top-1)/SIMD_VEC)*SIMD_VEC-1, SIMD_VEC
    !gather loop -- vectorized
    !DIR SIMD
    DO i2 = 1, SIMD_VEC, 1
      map1idx = opDat1Map(1+(i1+i2-1)*opDat1MapDim+0)+1
      map2idx = opDat1Map(1+(i1+i2-1)*opDat1MapDim+1)+1
      map3idx = opDat3Map(1+(i1+i2-1)*opDat3MapDim+0)+1
      map4idx = opDat3Map(1+(i1+i2-1)*opDat3MapDim+1)+1
      ...
      dat3(i2,1) = opDat3Loc(1,map3idx)
      dat3(i2,2) = opDat3Loc(2,map3idx)
      dat3(i2,3) = opDat3Loc(3,map3idx)
      dat3(i2,4) = opDat3Loc(4,map3idx)
      ...
      dat7(i2,1:4) = 0.0
      dat8(i2,1:4) = 0.0
    END DO
    !DIR SIMD
    DO i2 = 1, SIMD_VEC, 1
      CALL res_calc_vec(dat1,dat2,dat3,dat4,dat5,dat6, &
      & dat7, dat8, i2) ! -- vecotorized kernel call
    END DO
    ! scatter loop -- non-vectorized
    DO i2 = 1, SIMD_VEC, 1
      map7idx = opDat3Map(1+(i1+i2-1)*opDat3MapDim+0)+1
      map8idx = opDat3Map(1+(i1+i2-1)*opDat3MapDim+1)+1

      opDat7Loc(1,map7idx)=opDat7Loc(1,map7idx)+dat7(i2,1)
      opDat7Loc(2,map7idx)=opDat7Loc(2,map7idx)+dat7(i2,2)
      opDat7Loc(3,map7idx)=opDat7Loc(3,map7idx)+dat7(i2,3)
      opDat7Loc(4,map7idx)=opDat7Loc(4,map7idx)+dat7(i2,4)

      opDat7Loc(1,map8idx)=opDat7Loc(1,map8idx)+dat8(i2,1)
      opDat7Loc(2,map8idx)=opDat7Loc(2,map8idx)+dat8(i2,2)
      opDat7Loc(3,map8idx)=opDat7Loc(3,map8idx)+dat8(i2,3)
      opDat7Loc(4,map8idx)=opDat7Loc(4,map8idx)+dat8(i2,4)
    END DO
  END DO
  ! remainder - non-vectorized kernel call
  DO i1 = ((top-1)/SIMD_VEC)*SIMD_VEC, top-1, 1
    CALL res_calc( ... )
  END DO
END SUBROUTINE
```

Figure 4: Code generated for `res_calc` to facilitate auto-vectorisation

Table 1: Airfoil performance (20 MPI) on a 2 socket Intel Haswell E5-2650 v3 (2.30GHz, PeakBW 136 GB/s, STREAM 108 GB/s) system - non-vectorized

Loop	Dir/Ind	Time (sec)	Est. BW GB/sec	% runtime
`save_soln`	Direct	2.51	73.43	7.56
`adt_calc`	Indirect	10.47	39.62	31.54
`res_calc`	Indirect	11.64	75.21	35.04
`bres_calc`	Indirect	0.04	66.07	0.14
`update`	Direct	8.53	91.83	25.65
Total runtime		33.21	-	100

into mini-partitions, or blocks, which are colored based on potential data races [3]. Blocks of the same color can then be executed by different threads concurrently without any need for synchronization. The same technique is used to assign work to CUDA thread blocks or OpenCL work groups [4].

Table 1 presents the performance profile of the code in Figure 3 generated from the Fortran version of Airfoil. The results are for a 20-way MPI run solving a mesh with 2.8M edges in 1000 iterations, on a 2 socket (total of 20 cores) Intel Haswell E5-2650 v3 (2.30GHz) system with 128 GB of RAM. The compiler used is the Intel Fortran compiler version 16.0.0 20150815. We present the MPI performance first as a baseline to evaluate the speedups (if any) gained solely through SIMD vectorization. Later we present the performance of the application with SIMD vectorisation and also compare it to the performance on NVIDIA GPUs to give an indication of the best performance achievable for the application. As it can be seen from Table 1 the code takes about 33 seconds to complete a 2.8 Million edge computation. `adt_calc`, `res_calc` and `bres_calc` are indirect loops while `save_soln` and `update` are direct loops. Most of the time is spent in `adt_calc` and `res_calc`. The bandwidth figures are an estimate computed assuming perfect caching (i.e by only counting the number of bytes moved from RAM for each data element once). Although in reality this assumption does not fully hold, we can get an idea of an upper limit of achievable bandwidth for the application on this processor system. Note that the theoretical peak bandwidth of this two socket Haswell E5-2650 v3 system is 136 GB/sec (2×68GB/sec [5]). The sustained bandwidth reported by the STREAM benchmark [6] is 108 GB/sec.

3. AUTO-VECTORISATION

In our previous paper on vectorisation [13] we explored further optimizing the code generated for Intel CPUs and the Intel Xeon Phi using SIMD based vectorisation. Using two unstructured mesh applications we explored different strategies to achieve good vectorization including (1) using vector intrinsics and (2) using OpenCL on CPUs, both of which can be executed on Intel CPUs and the Xeon Phi co-processor. We showed that at the time we could only achieve efficient vectorization using vector intrinsics, however considering that the OP2 library does not have a full-fledged compiler, we also showed that there was no reliable way to automatically vectorize the computational kernels provided by the application developer - in fact it was necessary to alter the code to avoid any branching. Nevertheless, these results are important to demonstrate what can be achieved when using low-level vector intrinsics. Clearly, our former approach couldn't be applied to Fortran code (as vector intrinsics are

Table 2: Airfoil performance (20 MPI) on a 2 socket Intel Haswell E5-2650 v3 (2.30GHz, PeakBW 136 GB/s, STREAM 108 GB/s) system - auto-vectorized

Loop	Dir/Ind	Time (sec)	BW GB/sec	% runtime
`save_soln`	Direct	2.59	71.16	9.22
`adt_calc`	Indirect	5.76	72.02	20.53
`res_calc`	Indirect	10.96	79.89	39.10
`bres_calc`	Indirect	0.04	66.07	0.13
`update`	Direct	8.67	90.35	30.92
Total runtime		28.04	-	100

Table 3: Airfoil performance (20 MPI) on a 2 socket Intel Haswell E5-2650 v3 (2.30GHz, PeakBW 136 GB/s, STREAM 108 GB/s) system - vector intrinsics, C/C++

Loop	Dir/Ind	Time (sec)	BW GB/sec	% runtime
`save_soln`	Direct	2.58	71.42	9.58
`adt_calc`	Indirect	5.57	74.47	20.68
`res_calc`	Indirect	10.13	86.43	37.62
`bres_calc`	Indirect	0.04	66.07	0.15
`update`	Direct	8.59	91.19	31.90
Total runtime		26.93		100

not supported in Fortran), which underlines the importance of our current results; extending our methods to Fortran and supporting complicated user kernels by relying on compiler auto-vectorization.

In this section we explore code techniques that enable us to achieve auto-vectorisation for applications developed with OP2's Fortran based API (a similar technique has been used for the C/C++ API based applications which can be found in the OP2 GitHub repository [11]). The key issue prohibiting unstructured mesh applications from auto-vectorizing is accessing data via an indirection (i.e. through the mappings) in an indirect loop. More specifically given an iteration set (say edges) the objective of vectorization is to carry out the required computation per edge (which is specified in the user kernel) for multiple edges at the same time. Assuming a vector length of 4 doubles this would mean 4 edges will be computed in parallel using the vector registers. However when looping over the iteration set, edges in this case, to carry out the computation, the indirect accesses to other sets via the mapping (or connectivity tables) introduces data races. This is equivalent to the compiler not being able to prove that there is no aliasing which results in its refusal to vectorize the loop over edges. This essentially stops us from forcing line 17 (`DO i1 = bottom, top-1, 1`) of Figure 3 to be SIMD vectorized by placing a `!DIR$ SIMD` over it as vectorizing over this loop will also produce incorrect results due to the indirect access. The solution is to explicitly write gather-scatter code to pack the indirectly accessed data to local arrays and pass these to the user kernel. The specific changes in this case are illustrated in Figure 4.

To facilitate the packing and unpacking operations, we loop over the main iteration loop (line 25) in groups of 4 (`SIMD_VEC` is pre processor defined to be 4 in line 1, the vector length of the processor in doubles). Within the main iteration loop the data is gathered to local arrays using the indirections (mappings) (lines 34-37). Note that a zero value is initialized for the local variables for `op_dat`s that are incremented (i.e. `OP_INC`) indirectly. Now the gather loop could be vectorized with a `!DIR$ SIMD` directive as there are no

Table 4: Airfoil CUDA performance on 1/2 x NVIDIA K80 GPU (PeakBW 240 GB/s, BWTest 192 GB/s) system, compiled with PGI CUDA FORTRAN 15.1

Loop	Dir/Ind	Time (sec)	BW GB/sec	% runtime
`save_soln`	Direct	0.95	192.98	5.01
`adt_calc`	Indirect	2.61	158.93	13.68
`res_calc`	Indirect	11.37	77.03	59.59
`bres_calc`	Indirect	0.09	29.9	0.46
`update`	Direct	4.05	193.17	21.26
Total runtime		19.07	-	100

data dependencies (leading to data races) within these packing operations. The packed local arrays can now be passed on to the user kernel. However the user kernel needs to be slightly modified to allow for SIMD vectorisation. The need is to introduce an additional parameter (`idx`) that indicates the index of the element used within a given vector lane (see line 13/14). These modifications to the user kernel are fortunately trivial and could be achieved automatically through the OP2 code generator. OP2 generates the modified user kernel within the same file (see `res_calc_vec` on line 3). Finally any data that are incremented (`OP_INC`) or is read-/write (`OP_RW`) needs to be handled by an explicit unpacking operation (lines 52 - 60). Due to the indirect writing, this scattering cannot be vectorized on current generation hardware. The remainder of the iterations are handled by a non-vectorized loop (lines 64-66) calling the unmodified user kernel. Looking through the modifications required to achieve auto-vectorisation it is clear that the packing and unpacking code will cause an extra overhead. The question is whether the performance gained by vectorising the per-element computations would offset this cost to provide an overall performance improvement.

In contrast to indirect loops, direct loops do not have the above issue as all data (in `op_dat`s) are accessed in order of the loop iteration. By placing a `!DIR$ SIMD` over the main iteration loop we can instruct the compiler to make use of the vector registers to access 4 values (assuming a vector length of 4 doubles) at a time and pass them in to the user kernel as packed vector of values as before. In this case no change is required to the user kernel.

Table 2 presents the performance results for the auto-vectorized version of Airfoil. Comparing the result to that in Table 1 we see that overall runtime has improved by about 19% with vectorization. The key loops to gain speedups are `res_calc` (10%) and `adt_calc` (46%). `res_calc` does about 73 double precision computations per edge (iteration) while `adt_calc` does 5 `sqrt` calls in addition to about 56 double precision computations. Thus the high computational intensity allows these loops to gain considerable speedups with vectorisation, offsetting the pack/unpack overhead. However, vectorization appears to have been slightly detrimental to `update` and `save_soln` which although they are direct loops have very little computation per iteration.

For comparison Table 3 presents the performance of the Airfoil application vectorized using vector intrinsics (developed in our previous work [13]) for the 2.8 Million edge mesh. Note that these results are from the C/C++ version of the application. Comparing the overall runtime of the auto-vectorized and vector-intrinsics versions we see that the auto-vectorized version is only about 4% slower than the vector intrinsics version.

Table 5: Hydra performance (20 MPI) on a 2 socket Intel Haswell E5-2650 v3 (2.30GHz, PeakBW 136 GB/s, STREAM 108 GB/s) system - non-vectorized

Loop	Dir/Ind	Time (sec)	BW GB/sec	% runtime
`vfluxedge`	indirect	3.13	32.59	33.79
`wffluxedge`	indirect	0.17	12.28	1.79
`wvfluxedge`	indirect	0.15	12.57	1.66
`ifluxedge`	indirect	0.72	118.13	7.73
`edgecon`	indirect	0.68	176.51	7.32
`volapf`	direct	0.27	133.36	2.96
`srcsanode`	direct	1.02	40.98	11.07
`srck`	direct	0.21	172.57	2.28
`updatek`	direct	0.65	80.06	6.98
`accumedges`	indirect	0.82	50.30	8.91
`invjacs`	direct	0.12	92.34	1.27
Total runtime		9.25	-	100

For further comparison, Table 4 presents the performance profile of the CUDA parallelisation of the Airfoil code, generated with OP2 on an NVIDIA K80 GPU. Note that a K80 GPU card has two separate GPUs on it [9] and the results in the table are from executing on just one of these GPUs. The peak bandwidth of one GPU (ECC off) is 240 GB/sec while the NVIDIA CUDA bandwidthTest gives 192 GB/sec device to device sustained bandwidth. We see that the GPU performs about 30% faster than the two-socket Haswell system. Considering that the two socket Haswell processor has a TDP of 210W (2x105W [5]) and half of the K80 GPU had a TDP of 150W [9] the above comparison illustrate the performance per watt trade-off between two of the most popular parallel computing hardware platforms available today.

4. AUTO-VECTORISING HYDRA

Rolls-Royce's Hydra CFD application is a full-scale production code used for the simulation of turbomachinery such as Rolls-Royce's latest aircraft engines. It consists of several components to simulate various aspects of the design including steady and unsteady flows that occur around adjacent rows of rotating and stationary blades in the engine, the operation of compressors, turbines and exhausts as well as the simulation of behaviour such as the ingestion of ground vortices. The guiding equations solved are the Reynolds-Averaged Navier-Stokes (RANS) equations, which are second-order PDEs. By default, Hydra uses a 5-step Runge-Kutta method for time-marching, accelerated by multigrid and block-Jacobi preconditioning [7, 2]. The usual production meshes are in 3D and consist of tens of millions of edges, resulting in long execution times on modern CPU clusters. Hydra was originally designed and developed over 15 years ago at the University of Oxford and has been in continuous development since; it has become one of the main production codes at Rolls-Royce.

Originally Hydra was only parallelised to run on a distributed memory cluster of single threaded CPUs. However, recently Hydra was converted to use the OP2 API [14] and OP2 has been able to generate a number of parallelisations automatically that can now be executed on a wide range of multi-core and many-core hardware. These not only include distributed memory clusters of CPUs, but also multi-threaded CPUs, NVIDIA GPUs and their combinations with MPI. Our aim in this section is to chart the issues encountered in applying the auto-vectorising optimisations

Table 6: Hydra performance (20 MPI) on a 2 socket Intel Haswell E5-2650 v3 (2.30GHz, PeakBW 136 GB/s, STREAM 108 GB/s) system - vectorized

Loop	Dir/Ind	Time (sec)	BW GB/sec	% runtime
`vfluxedge`	indirect	2.20	46.37	27.68
`wffluxedge`	indirect	0.18	9.81	2.31
`wvfluxedge`	indirect	0.15	11.81	1.84
`ifluxedge`	indirect	0.80	100.35	10.07
`edgecon`	indirect	0.85	126.67	10.71
`volapf`	direct	0.28	115.04	3.48
`srcsanode`	direct	0.57	68.19	7.16
`srck`	direct	0.20	181.19	2.51
`updatek`	direct	0.66	84.90	8.28
`accumedges`	indirect	0.53	74.10	6.69
`invjacs`	direct	0.12	90.95	1.49
Total runtime		7.94	-	100

Table 7: Hydra performance on 1/2×NVIDIA K80 GPU (PeakBW 240 GB/s, BWTest 192 GB/s), compiled with PGI CUDA FORTRAN 15.1

Loop	Dir/Ind	Time (sec)	BW GB/sec	% runtime
`vfluxedge`	indirect	3.29	59.96	39.58
`wffluxedge`	indirect	0.11	25.04	1.36
`wvfluxedge`	indirect	0.08	34.57	0.91
`ifluxedge`	indirect	2.05	76.77	24.63
`edgecon`	indirect	3.21	54.70	38.54
`volapf`	direct	0.19	169.86	2.28
`srcsanode`	direct	0.39	97.18	4.65
`srck`	direct	0.26	148.79	3.07
`updatek`	direct	0.43	132.21	5.18
`accumedges`	indirect	2.59	29.52	31.09
`invjacs`	direct	0.28	36.26	3.42
Total runtime		8.32	-	100

to this large code-base using OP2's code generation techniques and explore the resulting performance.

We will use a configuration and input mesh of Hydra that model a standard application in CFD, called NASA Rotor37 [1]. It is a transonic axial compressor rotor widely used for validation in CFD. It is used for Rolls-Royce's HPC system procurements and validations, and as such is representative of other use cases of Hydra. For this problem over 90% of Hydra's runtime is spent within 11 key parallel loops. Their runtime for 20 iterations of a 20-way MPI job on 20 cores of the two socket Haswell system is detailed in Table 5.

In comparison Table 6 shows the performance of these key loops in Hydra with auto-vectorisation. The results illustrate the trade-offs in how we achieve vectorisation; the cost involved in setting up the vector registers and packing data and the benefit from the vectorised execution of the computations. Following this, there are two main factors that determine the cost/benefit; the complexity in data access patterns, which directly affects the cost of packing/unpacking the vector registers, and the amount and complexity of computations in the kernel.

On one end of this spectrum, the `edgecon` kernel accesses 71 double precision values from 11 different memory locations for each edge - when vectorised over 4 edges, this multiplies up to 44 non-contiguous locations in memory, due to the indirect access patterns that have poor locality across different edges; this packing and unpacking vector registers is very expensive. At the same time, this kernel does very few computations: 92 double precision floating-point operations per edge; and the bulk of the computations within

the kernel happen in a straightforward loop. We see that such a loop within the user kernel gets auto-vectorized by the compiler (which we note as intra-kernel vectorization to distinguish from the *inter*-kernel vectorization we have been attempting throughout this work). Thus for this kernel the overhead of packing and unpacking vector registers far outweighs the benefits of carrying out the operations in a vectorised way over the edges - partly because for the most part its internal computations gets vectorised already; overall the execution of `edgecon` slows down by about 25%. On the other end of this spectrum, we have `vfluxedge` kernel which accesses 104 double precision values from 17 different memory locations for each edge - when vectorised over 4 edges, this multiplies up to 68 non-contiguous locations in memory. Again due to the indirect access patterns it has poor data locality across different edges making the packing and unpacking vector registers expensive. However, the computation done within the loop is significant with over 700 double precision floating-point operations per edge. In fact `vfluxedge` has the largest user kernel of all the above 11 key loops and due to the higher computation per edge, vectorization gives a speedup of about 32%. Various other kernels lie in-between these two in terms of performance and the above two factors; `accumedges` accesses a lot of data indirectly, making packing/unpacking expensive, but performs a large number of operations, resulting in a 35% speedup, `ifluxedge` also accesses a lot of data indirectly but there are very few computations in the kernel, resulting in a 12% slowdown.

We see the above analysis holding for direct loops as well. `srcsanode`, moves a large amount of data (40 doubles) per node, but does so directly on the iteration set, therefore with very good locality across subsequent iterations. Furthermore there are a large number of operations in the kernel: around 216 per node with few branches and no loops; thus vectorisation across edges does improve computational performance significantly, resulting in a 45% speedup over the non-vectorised version. Direct loops with smaller computations such as `invjacs` and `updatek` shows a modest slowdown. `updatek` is further affected by global reductions which require separate (but similar) packing and unpacking for auto-vectorisation.

Overall, generating code that auto-vectorises helps Hydra significantly, improving overall performance by about 15%. However as shown there are computational kernels where inter-kernel vectorisation is actually detrimental to performance. Since OP2 can generate either code automatically, our strategy to achieve best performance is to generate both, and carry out an auto-tuning run that determines for which kernels it is advantageous to use the vectorised version. This is then fed back to the implementation that will query which implementation to use at run-time; improving overall speedup to 19% over the non-vectorised version.

Again for comparison we also present the performance profile of the CUDA parallelisation of Hydra in Table 7. Similar to Airfoil this is also generated with OP2 and is executed on just one of the two GPUs on a NVIDIA K80 card. We see that the GPU performs about 4.56% slower than the two-socket Haswell system.

5. CONCLUSIONS

The research detailed in this paper explored software techniques for achieving compiler SIMD auto-vectorisation for unstructured mesh based applications. This class of applications have very irregular memory access patterns and as such have previously been very difficult to vectorize on traditional x86 based CPUs. However in this paper, with the recent advances in the Intel compiler suites, we showed specific ways in which code could be written to force the compiler to automatically vectorize loops over collections of unstructured-mesh elements such as edges or nodes in a mesh. We showed how the indirect memory accesses in such loops, the main inhibitor for auto-vectorization, could be recoded with explicit packing and unpacking of data to local variables to avoid data races. This together with specific optimisations to the elemental operation carried out per iteration of the loop allowed automatic SIMD vectorisation.

The explicit details of the optimisations were presented with a standard CFD benchmark code called Airfoil previously developed and parallelized with the OP2 high-level domain specific framework. Performance results for Airfoil showed that performance with auto-vectorization is only about 4% slower than a previously hand-tuned vector intrinsics based version of the same application on a 20 core two socket Intel Haswell processor system.

Using OP2's automatic code generation capabilities the auto-vectorizing optimisations were applied to a large-scale production CFD application from Rolls-Royce plc. We see that there is considerable performance improvement with auto-vectorization where the most compute intensive parallel loops shows speedups of nearly 40% on the Intel Haswell system compared to their non-vectorized versions. However not all loops gain due to vectorization; loops with less computational intensity perform worse with vectorisation. Comparing auto-vectorized performance to that of the same application parallelized with CUDA using OP2 on one of the two GPUs on an NVIDIA K80 GPU card showed marginally better performance on the CPU system. These results demonstrated the best performance achievable within an equivalent power budget using two of the most popular parallel processing hardware platforms on offer today.

Future work will investigate a number of further optimisations to improve performance under auto-vectorization. These include the use of SoA on the CPUs for direct loops and tiling/loop fusion to combine smaller kernels to improve computational intensity. The later is expected to give greater benefits from vectorization. The full source of the Airfoil code used in this paper and the OP2 library is available online [11] as open source software.

Acknowledgements

This research has been funded by 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 grateful to John Pennycook at Intel for his invaluable advice during this research. We are thankful to David Radford, Leigh Lapworth, Paolo Adami and Yoon Ho at Rolls-Royce plc., for making the Hydra application and test cases available to us. We also would like to acknowledge the contributions made to the OP2 project by Paul Kelly, Carlo Bertolli, Graham Markall, Fabio Luporini, David Ham and Florian Rathgeber at Imperial College London, Lawrence Mitchell at the University of Edinburgh and Endre László at PPKE ITK Hungary.

6. REFERENCES

[1] Denton, J. D. Lessons from rotor 37. *Journal of Thermal Science 6*, 1 (1997), 1–13.

[2] Giles, M. B., Duta, M. C., Muller, J. D., and Pierce, N. A. Algorithm Developments for Discrete Adjoint Methods. *AIAA Journal 42*, 2 (2003), 198–205.

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

[4] Giles, M. B., Mudalige, G. R., Spencer, B., Bertolli, C., and Reguly, I. Designing OP2 for GPU Architectures. *Journal of Parallel and Distributed Computing 73* (November 2013), 1451–1460.

[5] Intel xeon processor e5-2650 v3 2.30 ghz, 2015. http://ark.intel.com/products/81705/Intel-Xeon-Processor-E5-2650-v3-25M-Cache-2_30-GHz.

[6] McCalpin, J. D. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec 1995), 19–25.

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

[8] Mudalige, G. R., Giles, M. B., Thiyagalingam, J., Reguly, I. Z., Bertolli, C., Kelly, P. H. J., and Trefethen, A. E. Design and initial performance of a high-level unstructured mesh framework on heterogeneous parallel systems. *Parallel Computing 39*, 11 (2013), 669 – 692.

[9] Nvidia launches tesla k80, gk210 gpu, 2014. http://www.anandtech.com/show/8729/nvidia-launches-tesla-k80-gk210-gpu.

[10] OP2 for Many-Core Platforms, 2011-2015. http://www.oerc.ox.ac.uk/projects/op2.

[11] OP2 github repository, 2011-2014. https://github.com/OP2/OP2-Common.

[12] Scotch and PT-Scotch, 2013. http://www.labri.fr/perso/pelegrin/scotch/.

[13] Reguly, I. Z., László, E., Mudalige, G. R., and Giles, M. B. Vectorizing unstructured mesh computations for many-core architectures. *Concurrency and Computation: Practice and Experience* (2015), n/a–n/a.

[14] Reguly, I. Z., Mudalige, G. R., Bertolli, C., Giles, M. B., Betts, A., Kelly, P. H. J., and Radford, D. Acceleration of a full-scale industrial cfd application with op2. *Parallel and Distributed Systems, IEEE Transactions on PP*, 99 (2015), 1–1.

[15] Intel xeon phi product family, 2015. http://www.intel.co.uk/content/www/uk/en/processors/xeon/xeon-phi-detail.html.