Comparison of Parallelisation Approaches, Languages, and Compilers for Unstructured Mesh Algorithms on GPUs

G. D. Balogh[1], I. Z. Reguly[1], and G. R. Mudalige[2]

[1] Faculty of Information Technology and Bionics, Pazmany Peter Catholic University, Budapest, Hungary,
balogh.gabor.daniel@hallgato.ppke.hu, reguly.istvan@itk.ppke.hu,
[2] Department of Computer Science, University of Warwick, Coventry, United Kingdom g.mudalige@warwick.ac.uk

Abstract. Efficiently exploiting GPUs is increasingly essential in scientific computing, as many current and upcoming supercomputers are built using them. To facilitate this, there are a number of programming approaches, such as CUDA, OpenACC and OpenMP 4, supporting different programming languages (C/C++ and Fortran mainly), and of course compilers (clang, nvcc, PGI, XL) support various combinations of these. In this study, we take a detailed look at some of the available options, and carry out a comprehensive analysis and comparison using computational loops and applications from the domain of unstructured mesh computations. Beyond runtimes and performance metrics (GB/s), we explore factors differentiating various combinations including register counts, occupancy, usage of different memory types, instruction counts, and algorithmic differences. We show clang's CUDA compiler frequently outperforming nvcc, the issues with directive-based approaches on complex kernels, and OpenMP 4 support maturing in clang and XL; currently around 10% slower than CUDA.

Keywords: compilers, CUDA, OpenACC, OpenMP, GPU, benchmarking

1 Introduction

The last ten years has seen the widespread adoption of Graphical Processing Units (GPUs) by the high performance computing community: for a great number of highly parallel workloads they offer unparalleled performance and efficiency. Programming techniques for GPUs have evolved significantly as well: the CUDA [1] language extensions to C/C++ and the OpenCL language [2] provide a low-level programming abstraction commonly referred to as Single Instruction Multiple Thread (SIMT) that gives fine-grained control over GPU architectures, allowing the exploitation of low-level features like scratch pad memory, warp operations, and block-level synchronization. However, converting existing applications to use CUDA or OpenCL is a significant undertaking that can be dis-

rupting to the source code, and getting good performance can entail significant work in orchestrating parallelism.

To simplify the adoption of GPUs, particularly for existing codes, high-level directive based programming abstractions were introduced. OpenACC [3] was one of the first supporting GPUs, and later on the OpenMP standard introduced support for accelerators with version 4 [4], and made subsequent refinements in 4.5 and 5.0. The evolution of directive based approaches was driven in particular by the acquisition of US DoE systems such as Titan and the upcoming Summit and Sierra systems: to be able to efficiently utilize these systems it was necessary that existing codes could be modified to support GPUs with a relative ease. Considering that many of these codes are written in Fortran, there is now compiler support for Fortran plus CUDA, OpenACC, and OpenMP in various compilers.

It is generally agreed that the best performance can be achieved by using CUDA, but the difference between CUDA and directive-based approaches vary significantly based on a multitude of factors: primarily the computation being parallelized, as well as the language being used (C or Fortran), and the compiler. This motivates the present study: for a number of parallel loops, coming from the domain of unstructured mesh computations, we wanted to get an idea of what performance looks like on different GPUs, different languages, and different compilers.

We evaluate some of the most commonly used compilers and parallelization approaches. CUDA C, compiled with nvcc, as well as with Google's recent clang based compiler [5]. On the CUDA Fortran side, the Portland Group (PGI, now owned by NVIDIA) has had support since 2010 [6], [7]. Additionally, as part of a recent push by IBM, preparing for the Summit and Sierra machines there has been support for CUDA Fortan with the XL compilers since v15.1.5 [8]. For OpenACC we use the PGI compilers which support both C and Fortran. There is also good support by for OpenACC by the cray compilers, however we did not have access to such machine and therefore will not be part of this analysis. For OpenMP 4 there are two compilers developed by IBM that have support on the C side: the XL compilers (since v13.1.5), and an extension to Clang [9]. There is also Fortran support by the XL compilers (since v15.1.5).

While there is a tremendous amount of research on performance evaluation of various combinations of languages and compilers, we believe our work is unique in its breadth: it directly compares C and Fortran implementations of the same code (Airfoil), and with three different parallelizations: CUDA, OpenACC, and OpenMP, and with five different compilers. We also present an in-depth study trying to explain the differences with the help of instruction counters and the inspection of low-level code. Specifically, we make the following contributions:

1. Using the Airfoil application, we run the same algorithms on NVIDIA K40 and P100 GPUs, with CUDA, OpenMP 4, and OpenACC parallelizations written in both C and Fortran, compiled with a number of different compilers.

2. We carry out a detailed analysis of the results with the help of performance counters to help identify differences between algorithms, languages, and compilers.
3. We evaluate these parallelizations and compilers on two additional applications, Volna (C) and BookLeaf (Fortran) to confirm the key trends and differences observed on Airfoil.

The rest of the paper is structured as follows: Section 2 discusses some related work, Section 3 briefly introduces the applications being studied, then Section 4 presents the test setup, compilers and flags. Section 5 carries out the benchmarking of parallelizations and the detailed analysis, and finally Section 6 draws conclusions.

2 Related Work

There is a significant body of existing research on performance engineering for GPUs, and compiler engineering, as well as some comparisons between parallelization approaches - the latter however is usually limited in scope due to the lack of availability of multiple implementations of the same code. Here we cite some examples, to show how this work offers a wider look at the possible combinations.

Work by Ledur et. al. compares a few simple testcases such as Mandelbrot and N-Queens implemented with CUDA and OpenACC (PGI) [10], Herdman et. al. [11] take a larger stencil code written in C, and study CUDA, OpenCL and OpenACC implementations, but offer no detailed insights into the differences. Work by Hoshino et. al. [12] offers a detailed look at CUDA and OpenACC variants of a CFD code and some smaller benchmarks written in C, and show a few language-specific optimizations, but analysis stops at the measured runtime. Normat et. al. [13] compare CUDA Fortran and OpenACC versions of an atmospheric model, CAM-SE, which offers some details about code generated by the PGI and Cray compilers, and identifies a number of key differences that let CUDA outperform OpenACC, thanks to lower level optimizations, such as the use of shared memory. Kuan et. al. [14] also compare runtimes of CUDA and OpenACC implementations of the same statistical algorithm (phylogenetic inference). Gonge et. al. [15] compare CUDA Fortran and OpenACC implementations of Nekbone, and scale up to 16k GPUs on Titan - but no detailed study of performance differences.

Support in compilers for OpenMP 4 and GPU offloading is relatively new [16] and there are only a handful papers evaluating their performance: Martineau et. al. [17] present some runtimes of basic computational loops in C compiled with Cray and clang, and comparisons with CUDA. Karlin et. al [18] port three CORAL benchmark codes to OpenMP 4.5 (C), compile them with clang, and compare them with CUDA implementations - the analysis is focused on runtimes and register pressure. Hart el. al. [19] compare OpenMP 4.5 with Cray to OpenACC on Nekbone, however the analysis here is also restricted to runtimes, the focus is more on programmability. We are not aware of academic papers studying

the performance of CUDA Fortran or OpenMP 4 in the IBM XL compilers aside from early results in our own previous work [20]. There is also very little work on comparing the performance of CUDA code compiled with nvcc and clang.

Thus we believe that there is a significant gap in current research: a comparison of C and Fortran based CUDA, OpenACC, and OpenMP 4, the evaluation of the IBM XL compilers, the maturity of OpenMP 4 compared to CUDA in terms of performance and a more detailed investigation into the reasons for the performance difference between various languages, compilers, and parallelization approaches. With the present study, we work towards filling this gap.

3 Applications

The applications being studied in this work come from the unstructured mesh computations domain solving problems in the areas of computational fluid dynamics, shallow-water simulation and Lagrangian hydrodynamics. As such, they consist of parallel loops over some set in the mesh, such as edges, cells or nodes, and on each set element some computations are carried out, while accessing data either directly on the iteration set, or indirectly via a mapping to another set. Our applications are all written using the OP2 domain specific language [21] embedded in C and Fortran, targeting unstructured mesh computations. While OP2 is capable of many things, its relevant feature for this work is that it can generate different parallelizations such as CUDA, OpenACC, and OpenMP 4, based on the abstract description of parallel loops.

A key challenge in unstructured mesh computations is the handling of race conditions when data is indirectly written. For the loops with indirect increments we use coloring to ensure that no two threads will write to the same memory at the same time. We can use a more sophisticated coloring approach for GPUs using CUDA as described in [22], where we create mini-partitions such that no two mini-partition of the same color will update the same cell. This allows mini-partitions of the same color to be processed by the blocks of one CUDA kernel. Within these mini-partitions, each assigned to a different CUDA thread block, each thread will process a different element, and thus is it necessary to introduce a further level of coloring. For an edges to cells mapping, we color all edges in a mini-partition so that no two edges with the same color update the same cell. Here, we first calculate the increment of every thread in the block, then we iterate through the colors and add the increment to the cell with synchronization between each color. The benefit of such an execution scheme is that there is a possibility that the data we loaded from the global memory can be reused within a block, which can lead to a performance increase due to fewer memory transactions. This technique is referred to as hierarchical coloring in the paper.

With other methods such as OpenACC and OpenMP4 there is no method for thread synchronization and data sharing in blocks, which is essential for the hierarchical coloring technique described above. Therefore a global coloring technique is used in case of these parallelization approaches. This technique is similar to the thread coloring inside the mini-partitions, but works on the full set. We assign colors to each thread in a way that no two edges of the same color update the same cell and threads from the same color can run parallel in a

separate CUDA kernel with synchronization between the kernels. This however excludes the possibility of the reuse of the data of the cells.

3.1 Airfoil

Airfoil is a benchmark application, representative of large industrial CFD applications. It is a non-linear 2D inviscid airfoil code that uses an unstructured grid and a finite-volume discretisation to solve the 2D Euler equations using a scalar numerical dissipation. The algorithm iterates towards the steady state solution, in each iteration using a control volume approach, meaning the change in the mass of a cell is equal to the net flux along the four edge of the cell. This requires indirect connections between cells and edges. Airfoil is implemented using OP2, where two versions exists, one implemented with OP2's C/C++ API and the other using OP2's Fortran API [21], [23].

The application consists of five parallel loops: **save_soln**, **adt_calc**, **res_calc**, **bres_calc** and **update** [24]. The **save_soln** loop iterates through cells and is a simple loop accessing two arrays directly. It basically copies every four state variables of cells from the first array to the second one. The **adt_calc** kernel also iterates on cells and it computes the local area/timestep for every single cell. For this it reads values from nodes indirectly and writes in a direct way. There are some computationally expensive operations (such as square roots) performed in this kernel. The **res_calc** is the most complex loop with both indirect reads and writes; it iterates through edges, and computes the flux through them. It is called 2000 times during the total execution of the application and performs about 100 floating-point operations per mesh edge. The **bres_calc** is similar to **res_calc** but computes the flux for boundary edges. Finally **update** is a direct kernel that includes a global reduction which computes a root mean square error over the cells and updates the state variables.

All test are executed with double precision on a mesh containing 2.8 million cells and with SOA data layout described in [24].

3.2 Volna

Volna is a shallow water simulation capable of handling the complete life-cycle of a tsunami (generation, propagation and run-up along the coast) [25]. The simulation algorithm works on unstructured triangular meshes and uses the finite volume method. Volna is written in C/C++ and converted to use the OP2 library[21]. For Volna we examined the top three kernels where most time is pent: **computeFluxes**, **SpaceDiscretization** and **NumericalFluxes**. In the **computeFluxes** kernel there are indirect reads and direct writes, in **NumericalFluxes** there are indirect reads with direct writes and a global reduction for calculating the minimum timestep and in **SpaceDiscretization** there are indirect reads and indirect increments.

Tests are executed in single precision, on a mesh containing 2.4 million triangular cells, simulating a tsunami run-up to the US pacific coast.

3.3 BookLeaf

BookLeaf is a 2D unstructured mesh Lagrangian hydrodynamics application from the UK Mini-App Consortium [26]. It uses a low order finite element method

with an arbitrary Lagrangian-Eulerian method. Bookleaf is written entirely in Fortran 90 and has been ported to use the OP2 API and library. Bookleaf has a large number of kernels with different access patterns such as indirect increments similar to increments inside **res_calc** in Airfoil. For testing we used the SOD testcase with a 4 million element mesh. For this case we examined the top five kernels with the highest runtimes which are **getq_christiensen1**, **getq_christiensen_q**, **getacc_scatter**, **gather**, **getforce_visc**. Among these there is only one kernel (**getacc_scatter**) with indirect increments (where coloring is needed), the **gather** and **getq_christiensen1** have indirect reads and direct writes as **adt_calc** in case of Airfoil, and the other two kernels have only direct reads and writes.

4 Test setup

For testing we used NVIDIA K40 and P100 GPUs in IBM S824L systems. We used nvcc in CUDA 8.0 and clang 5.0.0 (r308000) for compiling CUDA with C/C++. For compiling CUDA Fortran, we used PGI 17.4 compilers and IBM's XL compiler 15.1.6 beta 10 for Power systems. For OpenMP4, we tested clang version 4.0.0 (commit 6dec6f4 from the clang-ykt repo), and the XL compilers (13.1.6 beta 9). Finally, for OpenACC, we used the PGI compiler version 17.4. The specific compiler versions and flags are shown in Table 1.

To reviewers: please note that our P100 machine had a hardware failure two weeks before submission, therefore some numbers are old/outdated/missing, particularly ones with the XL compilers. We will add the updated values before camera-ready submission.

	Version	Flags
PGI	17.4-0	-O3 -ta=nvidia,cc35 -Mcuda=fastmath -Minline=reshape (-acc for OpenACC)
XL	13.1.6 beta 10	-O3 -qarch=pwr8 -qtune=pwr8 -qhot -qxflag=nrcptpo -qinline=level=10 -Wx,-nvvm-compile-options=-ftz=1 -Wx,-nvvm-compile-options=-prec-div=0 -Wx,-nvvm-compile-options=-prec-sqrt=0 (-qsmp=omp -qthreaded -qoffload for OpenMP4)
clang for OpenMP4	4.0	-O3 -ffast-math -fopenmp=libomp -Rpass-analysis -fopenmp-targets=nvptx64-nvidia-cuda -fopenmp-nonaliased-maps -ffp-contract=fast
clang for CUDA	5.0	-O3 –cuda-gpu-arch=sm_35 -ffast-math
nvcc	8.0.53	-O3 -gencode arch=compute_35,code=sm_35 –use_fast_math

Table 1. Compiler flags used on K40 GPU (for P100 cc60 and sm_60 is used)

5 Benchmarking

5.1 Airfoil

The run times of different versions of Airfoil on the K40 and P100 GPUs are shown in Figure 1. The hierarchical coloring is used in **res_calc** and **bres_calc**, because these have indirect increments and in the case of other kernels we don't need coloring because they have only direct updates. The versions using the

hierarchical coloring scheme have the best performance, due to the huge performance gains in **res_calc** thanks to data reuse. The main differences between versions with the same coloring strategy is in the run times of the **res_calc** and **adt_calc** kernels, where most of the computation is performed. In the following, we examine performance in detail on all five kernels.

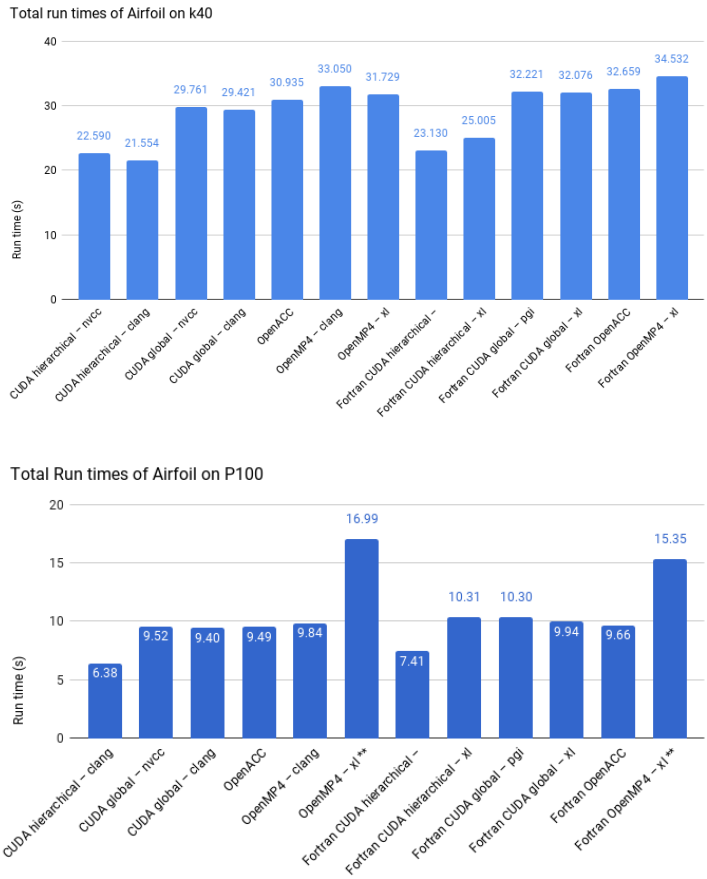

Fig. 1. Measured run times of versions on the K40 and P100 GPU

save_soln : **save_soln** is a really simple kernel with only direct reads and writes. In Table 2 the runtimes of the **save_soln** kernel are shown: all versions have approximately the same performance. In case of C/C++, OpenMP4 and OpenACC versions have about $6-7\%$ better runtimes and bandwidth than other versions even though the OpenMP4 versions are the only versions that have only 75% occupancy. The reason for this is that in case of CUDA we used 200 blocks so each thread processes more than one cell to save on integer instructions. However this leads us to a for loop inside the kernel, increasing control instructions, and slowing performance. If we run one thread per cell and delete this loop from the kernel the performance of CUDA matches the performance of OpenMP4. In

case of Fortran the OpenACC and OpenMP4 versions also perform better than CUDA with multiple cell per thread, but the difference is only 2-5%.

	K40			P100		
	Run Time (s)	Bandwidth (GB/s)	Register count	Run Time (s)	Bandwidth (GB/s)	Register count
nvcc - CUDA	1.052	175.136	24	0.359	512.726	24
clang - CUDA	1.052	175.078	24	0.359	512.741	24
OpenACC	1.006	183.234	26	0.350	526.508	29
XL - OpenMP4	0.985	187.173	17	0.491	374.931	19
clang - OpenMP4	0.984	187.628	35	0.355	519.094	32
PGI - F_CUDA	1.060	173.823	32	0.362	509.509	32
PGI - F_OpenACC	1.011	182.244	24	0.349	528.592	29
XL - F_CUDA	1.053	175.011	32	0.370	498.404	32
XL - F_OpenMP4	1.038	177.528	26	0.390	473.186	26

Table 2. save_soln run times and register count

adt_calc :

	K40			P100		
	Run Time (s)	Bandwidth (GB/s)	Register count	Run Time (s)	Bandwidth (GB/s)	Register count
nvcc - CUDA	2.932	141.477	46	0.879	472.149	46
clang - CUDA	2.768	149.844	40	0.874	474.401	44
OpenACC	4.070	101.925	86	0.988	419.683	96
XL - OpenMP4	3.803	109.085	64	1.266	327.756	72
clang - OpenMP4	4.440	93.442	88	1.079	384.620	96
PGI - F_CUDA	3.754	110.497	64	0.959	432.571	96
PGI - F_OpenACC	4.340	95.574	86	1.059	391.683	56
XL - F_CUDA	4.063	102.108	86	1.112	373.036	64
XL - F_OpenMP4	4.109	100.955	85	1.204	344.419	88

Table 3. adt_calc run times and register counts

In case of **adt_calc** the loop iterates on cells and reads data indirectly from the nodes while updating the values on the cell. The operation contains some expensive square root calculations. For **adt_calc** CUDA versions use fewer registers and reach better occupancy than other versions, as shown in Table 3 (CUDA versions with C/C++ use 40-46 registers, which means 62.5-75% in terms of occupancy, instead of 31.25-43.75% for OpenACC and OpenMP4). In case of Fortran, only the CUDA version compiled with the PGI compiler is able to achieve 50% occupancy all other versions have 31.25% because of the high register counts. Another source of performance difference in case of OpenMP4 comes from how the memory system is used; with clang OpenMP4 this kernel does not use texture caches, while with the XL compiler we get high numbers

of memory transactions (20.6M transaction at L2 level for XL, 11.6M for clang OpenMP4, and 8.6M in case of nvcc, and 33% more device memory transactions for XL compared to nvcc). Also in terms of instruction counts OpenACC and OpenMP4 versions execute 10-20% (45-50% for Fortran) more integer instructions than what we get with nvcc. The CUDA version compiled with clang performs 17% fewer floating point instructions and 23% fewer integer instructions compared to nvcc. The Fortran CUDA version with the PGI compiler has 28% lower performance than the C/C++ CUDA versions because it executes twice as many control instructions and 1.5 times as many integer instructions. Although with the XL compiler for CUDA Fortran we get 7% fewer floating point instructions, we get twice as much integer instructions compared to nvcc.

res_calc : In case of **res_calc** we have indirect updates, therefore we need coloring to avoid race conditions. The runtime and bandwidth results are shown in Table 4 for hierarchical coloring, and for global coloring in Table 5. In case of hierarchical coloring all versions perform within 7% of nvcc's performance except the XL compiler for CUDA Fortran which is 12% slower due to low occupancy (caused by register pressure) and it also executes about 20% more integer instructions. The instruction counts for these versions are shown in Table 6. The memory usage of CUDA nvcc and clang versions is mostly similar but in case of clang, 35% fewer Shared Load instructions are executed for the kernel. Another source of difference between the performance of nvcc and clang is due to a 15% difference in the number of integer instructions and 64 bit floating point instructions.

For global coloring, CUDA with nvcc has 3% fewer shared memory transactions than clang, but clang has 13% fewer floating point and 16% fewer integer instructions as shown in Table 7. Differences in shared memory usage for both coloring approaches come down to how values in shared memory used repeatedly are cached in registers. For global coloring, every OpenACC and OpenMP4 version is within 10% of nvcc's performance, with OpenMP4 Fortran with the XL compiler using high number of registers and executing high numbers of integer and control instructions. In case of C/C++ the OpenMP4 versions have 2-3% more global memory transactions, and about 2.5 times more shared memory transactions than nvcc and the OpenMP4 clang version doesn't use the texture cache. The OpenACC version executes 1% more global memory instructions, but there is no difference in the usage of shared memory and texture cache compared to nvcc. As in the case of the previous kernels, the number of integer instructions executed by the OpenMP4 and OpenACC versions are about 10-30% higher than the CUDA versions, but they have the same amount of floating point instructions.

bres_calc : The **bres_calc** kernel also has indirect reads and writes, so we need coloring as in case of **res_calc**. In this case the versions using hierarchical coloring performs equally good except the Fortran CUDA version compiled with the PGI compiler as shown in Table 8. The Fortran versions have lower

	K40			P100		
	Run Time (s)	Bandwidth (GB/s)	Register count	Run Time (s)	Bandwidth (GB/s)	Register count
nvcc - CUDA	13.881	63.073	58	3.824	228.989	50
clang - CUDA	12.960	67.556	50	3.607	242.703	56
PGI - F_CUDA	13.639	64.197	64	4.451	196.715	78
XL - F_CUDA	15.422	56.772	79	-		

Table 4. res_calc run times and register count in case of hierarchical coloring

	K40			P100		
	Run Time (s)	Bandwidth (GB/s)	Register count	Run Time (s)	Bandwidth (GB/s)	Register count
nvcc - CUDA	21.219	41.262	48	6.748	129.744	48
clang - CUDA	21.045	41.603	44	6.629	132.072	48
OpenACC	21.478	40.765	72	6.621	132.229	88
XL - OpenMP4	22.336	39.199	71	7.750	112.973	80
clang - OpenMP4	22.343	39.186	96	6.647	131.725	96
PGI - F_CUDA	22.709	38.555	87	7.304	119.875	88
PGI - F_OpenACC	22.997	38.072	87	6.723	130.227	96
XL - F_CUDA	22.295	39.271	96	7.0746	123.759	104
XL - F_OpenMP4	23.820	36.755	110	11.203	78.149	47

Table 5. res_calc run times and register count in case of global coloring

	nvcc	clang	fortran PGI	fortran XL
integer instructions	205569K	173366K	181360K	241244K
floating point (64 bit) instructions	88698K	77183K	91001K	88698K

Table 6. Average number of instructions performed in **res_calc** kernel with hierarchical coloring on k40 GPU

	fp (64 bit)	integer	control
nvcc - CUDA	93555K	116583K	1439K
clang - CUDA	92115K	94994K	1439K
OpenACC	96433K	159763K	1439K
XL - OpenMP4	93555K	142490K	1439K
clang - OpenMP4	93555K	119462K	1439K
PGI - fortran CUDA	96433K	171278K	2879K
PGI - fortran OpenACC	96433K	146809K	1439K
XL - fortran CUDA	93555K	191428K	2879K
XL - fortran OpenMP4	93555K	171278K	7198K

Table 7. Average number of instructions performed in **res_calc** kernel with global coloring on K40 GPU

	K40			P100		
	Run Time (s)	Bandwidth (GB/s)	Register count	Run Time (s)	Bandwidth (GB/s)	Register count
nvcc - CUDA	0.065	31.043	44	0.031	64.919	42
clang - CUDA	0.068	29.795	45	0.032	63.065	46
PGI - F_CUDA	0.081	24.923	53	0.028	71.894	78
XL - F_CUDA	0.066	30.736	78	-		

Table 8. bres_calc run times and register count in case of hierarchical coloring

	K40			P100		
	Run Time (s)	Bandwidth (GB/s)	Register count	Run Time (s)	Bandwidth (GB/s)	Register count
nvcc - CUDA	0.072	28.053	40	0.033	61.527	40
clang - CUDA	0.072	28.320	37	0.033	62.424	40
OpenACC	0.073	27.751	71	0.034	59.914	56
XL - OpenMP4	0.088	23.066	72	0.040	51.181	80
clang - OpenMP4	0.080	25.367	86	0.036	55.568	94
PGI - F_CUDA	0.095	21.196	56	0.038	55.684	80
PGI - F_OpenACC	0.075	27.016	102	0.036	56.773	88
XL - F_CUDA	0.080	25.437	94	0.040	50.404	70
XL - F_OpenMP4	1.098	1.849	54	0.446	4.549	47

Table 9. bres_calc run times and register count in case of global coloring

occupancy (56.25% and 37.5% for PGI and XL respectively versus 62.5% for C/C++ side for both clang and nvcc). In this case nvcc has about 10% fewer memory instructions than clang which has some spilled registers. In case of Fortran, the version compiled with PGI has spilled registers and fewer texture loads than with XL (7k texture loads with PGI and 22k with XL) also with PGI we get 20% more global memory transactions, about 50% more transactions on L2 level and twice as much shared memory transaction than any other version. In terms of instruction counts the clang version does best; it has 10% fewer floating point instructions than the other versions, 13% fewer integer instructions and 5% fewer control instructions than nvcc. However the spilled registers and higher number of memory transactions lead to lower performance than with nvcc. On the Fortran side the PGI compiled version has 21% fewer control instructions than nvcc on C/C++ side, but has 10% more integer instructions, while with the XL compiler we get 40% more integer instructions and 18% fewer control instructions than with nvcc. In case of CUDA Fortran with PGI we face the same situation than with clang on C side, it has better instruction counts than XL but the spilled registers and memory transactions slow down the execution. However on the P100 GPU the PGI compiled version uses higher number of registers and we get about 10% better performance than C/C++ versions, because there is no spilled registers (the clang compiled C/C++ version is still uses spilled registers). With global coloring, the C/C++ CUDA versions performed equally as shown in Table 9, and the OpenACC and OpenMP4 versions got 10-20% loss in performance due to high register counts (resulting in lower occupancy)

with both GPUs. In this case CUDA clang has about 10% fewer texture reads, and about the same amount of shared memory and global memory transactions as nvcc. For this kernel the OpenMP4 versions don't use texture cache, while for Fortran XL with OpenMP4 have high number of memory transactions because of spilled registers. The CUDA Fortran XL version performs almost the same as nvcc on C/C++ side, but with PGI we get fewer texture cache reads (5k with PGI instead of 19k with nvcc on C/C++ side) and have 15% more global memory transactions in case of K40. The XL compiler in this case has generated extremely high numbers of control and integer instructions for the Fortran OpenMP4 version (20 times more integer and 100 times more control instructions as other Fortran versions).

	K40			P100		
	Run Time (s)	Bandwidth (GB/s)	Register count	Run Time (s)	Bandwidth (GB/s)	Register count
nvcc - CUDA	4.485	174.674	29	1.503	521.368	32
clang - CUDA	4.483	174.728	32	1.504	520.803	32
OpenACC	4.308	181.841	36	1.498	522.766	38
XL - OpenMP4	4.518	173.381	32	7.447	108.190	40
clang - OpenMP4	5.205	150.499	86	1.721	455.198	86
PGI - F_CUDA	4.601	170.241	79	1.638	478.148	36
PGI - F_OpenACC	4.235	184.963	37	1.492	524.998	40
XL - F_CUDA	4.586	170.831	61	1.701	460.456	54
XL - F_OpenMP4	5.481	142.919	80	2.109	371.525	70

Table 10. update run times and register count (On P100 GPU the OpenMP4 XL version uses an earlier compiler version which has issues with reduction)

update For **update** we have direct reads and writes, so we don't need coloring. In addition, there is a global reduction in this kernel. For CUDA versions we run only 200 blocks as in the case of **save_soln** and in this case this approach helps us minimize the cost of the global reduction (i.e. larger chunks of the reduction can be computed in a single block's shared memory), less data moved to the host. In Table 10 the runtimes and the register counts are shown. For the global reduction the OpenACC versions generated an extra kernel where it computes the result (this add another 110ms for both C/C++ and Fortran on the K40, and 80ms for C/C++ and 86ms for Fortran on the P100 GPU). On the C/C++ side all versions except clang OpenMP4 perform about the same in terms of occupancy. The Fortran versions have lower occupancy because of the high register usage (the OpenACC version has a separate kernel for reduction thus have lower register count for the bulk of the kernel). In terms of memory transactions CUDA versions are perform equally on the C/C++ side. The C/C++ OpenMP4 versions don't use texture caches, and use 2-3 times more integer instructions than CUDA, but overall performance is similar because the kernel is bandwidth limited. On the Fortran side all versions perform within 2%

of nvcc's performance (except OpenMP4 with XL which behaves similarly as the C/C++ version) but all versions have spilled registers (which introduce about 10k-20k local load and store transactions). The CUDA clang version has 6-7% fewer floating point and integer instructions than nvcc. The OpenMP4 versions have 30% more floating point instructions, 3 and 2 times more integer and 7 and 4 times more control instructions than nvcc with clang and XL respectively. The OpenACC version has 5% more floating point instructions and 55% more integer instructions compared to nvcc. On the Fortran side the OpenMP4 version has 40% more floating point instructions than on C/C++ side but the C/C++ version has 4 times more integer and about 20 times more control instructions. The CUDA Fortran versions have 5-10% fewer floating point instructions and 5-15% more integer instructions compared to nvcc. The OpenACC Fortran version executes about the same amount of instructions as the C/C++ OpenACC version.

Effect of tuning the number of registers per thread In case of the Airfoil application, the key performance limiter is the latency of accesses to global memory. To achieve high bandwidth, we need many loads in flight, and for that we should increase the occupancy, which is limited by the number of registers used in these kernels. To get better occupancy we can limit the maximum number of registers per thread during the compilation. With both K40 and P100 with 128 thread per block the jumps in occupancy values (where occupancy decreases if we use one more register per thread) are the same register counts and we tested all versions with the three values which leads to occupancy increase in case of that version. In case of hierarchical coloring the shared memory required by the kernel could be the bottleneck for occupancy. In Figure 2 and 3 the runtime of limited versions relative to the original version in percentage are shown. The shared memory requirement of **res_calc** and **bres_calc** is roughly 4KB per block which limits the occupancy at 68.8% on the K40, meaning that we cannot reach better occupancy by further reducing the maximum register count (reducing the count to 48 would lead to 62.5% and to 40 would lead to 75% occupancy). On the P100 GPU this requirement maximizes the occupancy at 94% thanks to more available shared memory. For most language-compiler combinations, limiting the register count only affects the **adt_calc**, **res_calc** and **bres_calc** kernels. In the OpenMP4 - clang, Fortran OpenMP4 - XL, and Fortran CUDA - PGI combinations, **update** is also affected by the limiting because of the high register count as shown in Table 10.

With the increased occupancy, we do get better run times in most cases (a limit of 56 in case of C/C++ and CUDA and 80 for other versions), except for the Fortran OpenMP4 version with the XL compiler. However further limitation of register counts leads to performance degradation, with the exception of CUDA Fortran code compiled with XL. The reason for the loss of performance is the increasing number of spilled registers, and the latency introduced by the usage of these registers.

The main differences lie in the run times of **res_calc** and **adt_calc**In case of **res_calc** on C/C++ side limiting the register count increases the performance

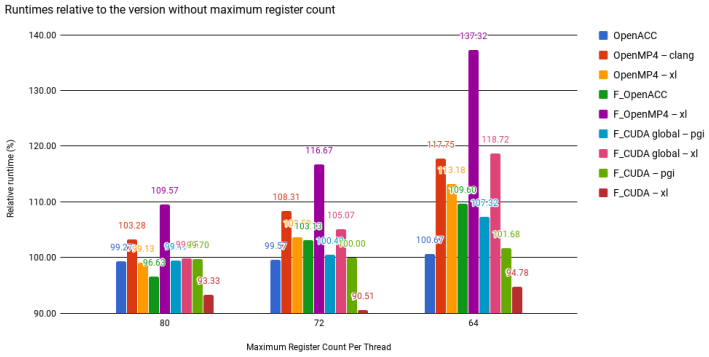

Fig. 2. Runtime of C OpenACC/OpenMP 4 and Fortran versions with limited register per threa

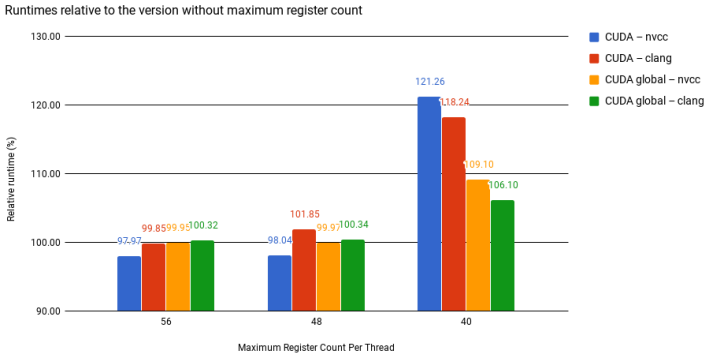

Fig. 3. Runtime of CUDA C versions with limited register per thread relative to original versions measured on K40. Lower is better.

by 5% in case of CUDA nvcc (with hierarchical coloring), the OpenMP4 XL compiler and OpenACC versions also get better run times by 1-2% but the other versions get significantly higher run times. In case of CUDAFortran with the XL compiler the runtime is decreased by 12% using a limit of 72 registers, but the OpenMP4 version runs 13% slower than the original if we limit the register usage. The other versions performed within 2% of the original for the first step and the run time is increased further for subsequent limits. In case of **adt_calc** all versions have better performance due to the increased occupancy, except for CUDAclang and CUDAFortran with PGI, which show no significant change. Other Fortran versions and the OpenMP4 version with clang have significant performance increase if we limit the register usage (9%, 18%, 17% and 15% for OpenMP4 Fortran XL, CUDA Fortran XL, OpenACC Fortran PGI, OpenMP4 clang respectively), the runtime of other versions improved by up to 6%.

5.2 Volna

For Volna the **SpaceDiscretization** kernel has a huge impact on runtime (half of the time is spent in this kernel when using global coloring), and so the hierarchical coloring leads to significant overall performance gain. Overall, with global

coloring the CUDA clang and OpenACC versions are within 10% of the nvcc's performance and OpenMP4 is just 14% slower than nvcc, as shown in Figure 4. In terms of occupancy, the OpenACC and OpenMP4 versions have lower occupancy for the **computeFluxes** kernel, and as in case of Airfoil the OpenMP4 versions have lower occupancy in most of the kernels as shown in Table 12. The CUDA clang version performs better in terms of instructions executed, as in the case of Airfoil. In the **computeFluxes** kernel clang executes 30% fewer floating point instructions and 20% fewer integer and control instructions than the nvcc version, but there are about 20 million local load and store instructions which leads to an overall performance loss. In case of other kernels clang executes about 15-20% fewer integer instructions and the same amount of control instructions compared to nvcc (except in **NumericalFluxes** where clang executes 10% more integer and control instructions than nvcc) and there are no local memory transactions. The OpenACC and OpenMP4 versions have higher instruction counts in every kernel than CUDA versions. Another reason for the high runtime of OpenMP4 versions is that they have 2-8 times more control instructions than other versions.

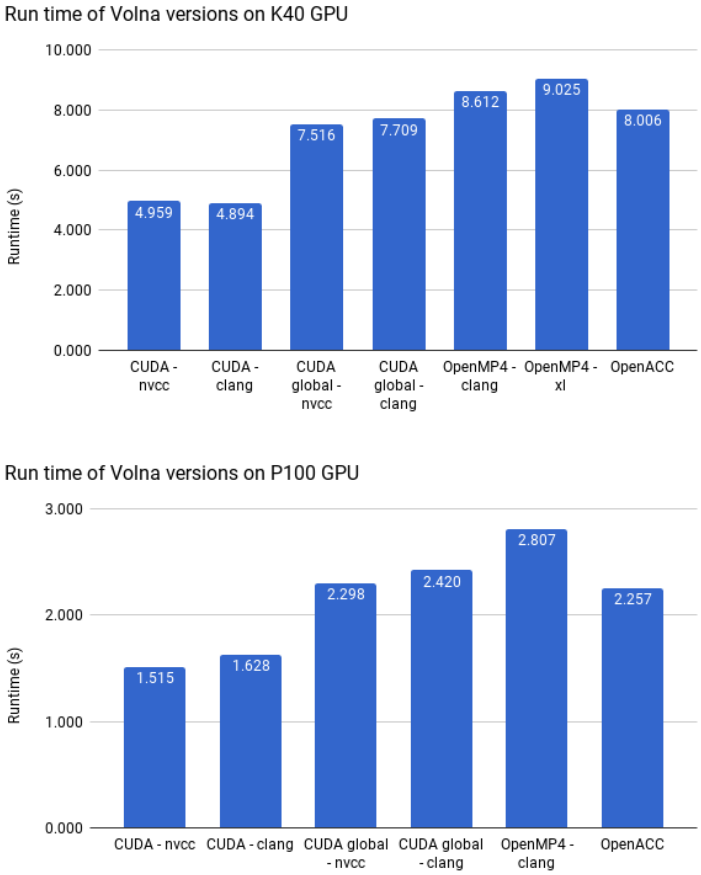

Fig. 4. Measured run times of Volna versions on the K40 and P100 GPU

	nvcc CUDA	clang CUDA	nvcc CUDA global	clang CUDA global	clang OpenMP4	XL OpenMP4	OpenACC
compute Fluxes	1.361	1.681	1.326	1.682	2.164	1.669	1.625
Space Discretization	2.040	1.869	4.725	4.700	3.979	5.341	4.762
Numerical Fluxes	0.433	0.430	0.414	0.416	0.801	0.572	0.468
Evolve Values RK2_2	0.316	0.316	0.316	0.316	0.411	0.338	0.302
Evolve Values RK2_1	0.449	0.371	0.449	0.371	0.641	0.390	0.338

Table 11. Run times of the five most time consuming Volna kernels on the K40 GPU

5.3 BookLeaf

Since in case of BookLeaf most of the time is spent in direct kernels, there isn't as much difference between hierarchical and global coloring versions in total run

	nvcc CUDA	clang CUDA	nvcc CUDA global	clang CUDA global	clang OpenMP4	XL OpenMP4	OpenACC
compute Fluxes	58	60	46	43	93	78	77
Space Discretization	36	39	22	22	64	30	30
Numerical Fluxes	27	27	26	20	40	30	33
Evolve Values RK2_2	33	25	28	25	80	25	28
Evolve Values RK2_1	28	26	33	26	86	32	33

Table 12. Register counts of the five most time consuming Volna kernels on the K40 GPU

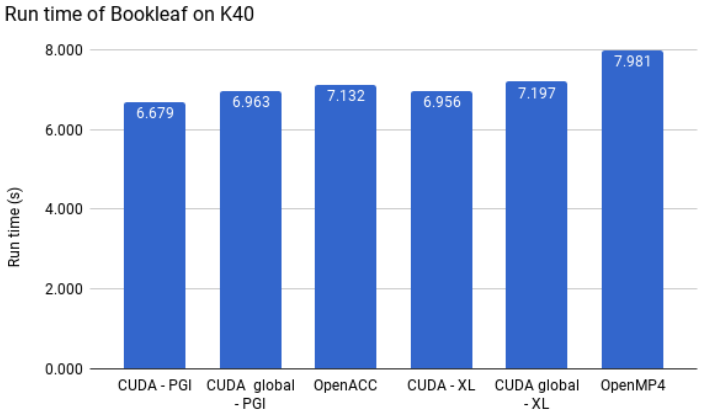

Fig. 5. Measured run times of BookLeaf versions on k40 GPU

time, as shown in Figure 5. However in case of **getacc_scatter**, which is the only kernel with indirect increments among the top five most time consuming kernels, the runtime of the hierarchical coloring is at least 70% better than that of the global coloring versions. All versions except OpenMP4 are within 8% of the performance of the best version which is Fortran CUDA with hierarchical coloring compiled with the PGI compiler. As in case of airfoil, the Fortran CUDA code compiled with XL have higher register counts (thus lower occupancy) than the PGI versions as shown in Table 14.

6 Conclusions

In this paper we have carried out a detailed study of some of the most popular parallelization approaches, programming languages, and compilers used to

	CUDA - PGI	CUDA global - PGI	OpenACC	CUDA - XL	CUDA global - XL	OpenMP4
getq_christiensen1	0.857	0.935	1.035	1.013	1.183	0.915
getq_christiensen_q	0.932	0.929	0.976	1.017	1.016	0.759
getacc_scatter	0.450	0.798	0.917	0.495	0.763	0.748
gather	0.495	0.526	0.525	0.542	0.540	0.592
getforce_visc	0.494	0.493	0.484	0.488	0.488	0.396

Table 13. Run times of the five most time consuming BookLeaf kernel on K40 GPU

	CUDA - PGI	CUDA global - PGI	OpenACC	CUDA - XL	CUDA global - XL	OpenMP4
getq_christiensen1	72	78	77	94	104	78
getq_christiensen_q	86	86	143	135	135	70
getacc_scatter	79	44	28	86	58	52
gather	24	30	23	35	31	23
getforce_visc	40	40	32	48	48	40

Table 14. Register counts of the five most time consuming Bookleaf kernel on K40 GPU

program GPUs, on a number of parallel loops coming from the domain of unstructured mesh computations. OpenMP4 and OpenACC are high level models using directives on loops in order to utilize GPUs, while CUDA use a lower level Single Instruction Multiple Threads model.

In this class of applications, a key common computational pattern is the indirect incrementing of data: to avoid race conditions we explored the use of coloring. The high level models must use global coloring of the iteration set to ensure that no two thread writes the same value when running simultaneously, whereas with lower-level models (CUDA) it is possible to apply a "two-level" coloring approach permitting better data reuse.

In case of Fortran, the CUDA versions with global coloring and OpenACC versions are within 10% of each other's performance. However the OpenMP4 versions use higher number of registers per thread, leading to low occupancy, as well as high numbers of local memory transactions when registers are spilled. OpenMP4 versions also use higher numbers of integer and control instructions.

On the C/C++ side, CUDA code compiled with the clang compiler performs 2-5% better in terms of runtime and perform 20% fewer integer instructions compared to nvcc. While OpenACCś performance is within 10% of CUDA's performance, OpenMP4 code compiled with clang is about 15% slower than its CUDA equivalent, due to lower occupancy, 15-50% more integer instructions, and a more infrequent use of the texture cache. The OpenMP4 versions compiled with XL are 7-20% slower than CUDA for similar reasons: lower occupancy, more control and integer instructions, less texture cache usage.

We have also shown that using CUDA one can handle race conditions more efficiently thanks to block-level synchronization; this in turn enables an execution approach with much higher data reuse. Kernels with indirect increments using hierarchical coloring have significantly better performance than the versions using global coloring; in case of Airfoil hierarchical coloring leads to about

35% better overall performance, in case of Volna about 50% and with BookLeaf about 6%.

In summary, we have demonstrated that support for C is only slightly better than for Fortran, for all possible combinations, with a 3-10% performance gap. Our work is among the first ones comparatively evaluating the clang CUDA compiler and IBM's XL compilers; clang's CUDA support is showing great performance already, often outperforming nvcc. Even though the XL compilers are only about one year old, they are already showing competitive performance and good stability - on the OpenMP 4 side often outperforming clang's OpenMP 4 and PGI's OpenACC. Directive based approaches demonstrate good performance on simple computational loops, but struggle with more complex kernels due to increased register pressure and instruction counts - lagging behind CUDA on average by 5-15%, but in the worst cases by up to 60%. It still shows that OpenMP 4 GPU support isn't yet as mature as OpenACC, nevertheless, they are within 5-10%. Our results also demonstrate how CUDA allows for more flexibility in applying optimizations that are currently not possible with OpenACC or OpenMP 4.

Acknowledgements

The authors would like to thank the IBM Toronto compiler team, and Rafik Zurob in particular, for access to beta compilers and help with performance tuning. Thanks to Carlo Bertolli at IBM TJ Watson for help with the clang OpenMP 4 compiler. This paper was supported by the János Bólyai Research Scholarship of the Hungarian Academy of Sciences. The authors would like to acknowledge the use of the University of Oxford Advanced Research Computing (ARC) facility in carrying out this work `http://dx.doi.org/10.5281/zenodo.22558`. The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013).

References

[1] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730. DOI: `10.1145/1365490.1365500`. [Online]. Available: `http://doi.acm.org/10.1145/1365490.1365500`.

[2] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *IEEE Des. Test*, vol. 12, no. 3, pp. 66–73, May 2010, ISSN: 0740-7475. DOI: `10.1109/MCSE.2010.69`. [Online]. Available: `http://dx.doi.org/10.1109/MCSE.2010.69`.

[3] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc: First experiences with real-world applications," in *Proceedings of the 18th International Conference on Parallel Processing*, ser. Euro-Par'12, Rhodes Island, Greece: Springer-Verlag, 2012, pp. 859–870, ISBN: 978-3-642-32819-0. DOI: 10.1007/978-3-642-32820-6_85. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32820-6_85.

[4] *OpenMP 4.5 specification*, http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.

[5] J. Wu, A. Belevich, E. Bendersky, M. Heffernan, C. Leary, J. Pienaar, B. Roune, R. Springer, X. Weng, and R. Hundt, "Gpucc: An Open-source GPGPU Compiler," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16, Barcelona, Spain: ACM, 2016, pp. 105–116, ISBN: 978-1-4503-3778-6. DOI: 10.1145/2854038.2854041.

[6] *The Portland Group*, http://www.pgroup.com.

[7] G. Ruetsch and M. Fatica, *Cuda fortran for scientists and engineers: Best practices for efficient cuda fortran programming*. Elsevier, 2013.

[8] *Getting Started with CUDA Fortran programming using XL Fortran for Little Endian Distributions*, http://www-01.ibm.com/support/docview.wss?uid=swg27047958&aid=11.

[9] *Clang with OpenMP 4 support*, https://github.com/clang-ykt.

[10] C. L. Ledur, C. M. Zeve, and J. C. dos Anjos, "Comparative analysis of openacc, openmp and cuda using sequential and parallel algorithms," in *11th Workshop on parallel and distributed processing (WSPPD)*, 2013.

[11] J. Herdman, W. Gaudin, S. McIntosh-Smith, M. Boulton, D. A. Beckingsale, A. Mallinson, and S. A. Jarvis, "Accelerating hydrocodes with openacc, opencl and cuda," in *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, IEEE, 2012, pp. 465–471.

[12] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "Cuda vs openacc: Performance case studies with kernel benchmarks and a memory-bound cfd application," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, May 2013, pp. 136–143. DOI: 10.1109/CCGrid.2013.12.

[13] M. Norman, J. Larkin, A. Vose, and K. Evans, "A case study of cuda fortran and openacc for an atmospheric climate kernel," *Journal of computational science*, vol. 9, pp. 1–6, 2015.

[14] L. Kuan, J. Neves, F. Pratas, P. Tomás, and L. Sousa, "Accelerating phylogenetic inference on gpus: An openacc and cuda comparison.," in *IWBBIO*, 2014, pp. 589–600.

[15] J. Gong, S. Markidis, E. Laure, M. Otten, P. Fischer, and M. Min, "Nekbone performance on gpus with openacc and cuda fortran implementations," *The Journal of Supercomputing*, vol. 72, no. 11, pp. 4160–4180, 2016.

[16] S. F. Antao, A. Bataev, A. C. Jacob, G.-T. Bercea, A. E. Eichenberger, G. Rokos, M. Martineau, T. Jin, G. Ozen, Z. Sura, *et al.*, "Offloading support for openmp in clang and llvm," in *Proceedings of the Third Workshop on LLVM Compiler Infrastructure in HPC*, IEEE Press, 2016, pp. 1–11.

[17] M. Martineau, J. Price, S. McIntosh-Smith, and W. Gaudin, "Pragmatic performance portability with openmp 4.x," in *OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings*, N. Maruyama, B. R. de Supinski, and M. Wahib, Eds. Cham: Springer International Publishing, 2016, pp. 253–267, ISBN: 978-3-319-45550-1. DOI: 10.1007/978-3-319-45550-1_18. [Online]. Available: https://doi.org/10.1007/978-3-319-45550-1_18.

[18] I. Karlin, T. Scogland, A. C. Jacob, S. F. Antao, G.-T. Bercea, C. Bertolli, B. R. de Supinski, E. W. Draeger, A. E. Eichenberger, J. Glosli, H. Jones, A. Kunen, D. Poliakoff, and D. F. Richards, "Early experiences porting three applications to openmp 4.5," in *OpenMP: Memory, Devices, and Tasks: 12th International Workshop on OpenMP, IWOMP 2016, Nara, Japan, October 5-7, 2016, Proceedings*, N. Maruyama, B. R. de Supinski, and M. Wahib, Eds. Cham: Springer International Publishing, 2016, pp. 281–292, ISBN: 978-3-319-45550-1. DOI: 10.1007/978-3-319-45550-1_20. [Online]. Available: https://doi.org/10.1007/978-3-319-45550-1_20.

[19] A. Hart, "First experiences porting a parallel application to a hybrid supercomputer with openmp4.0 device constructs," in *IWOMP*, 2015.

[20] I. Z. Reguly, A.-K. Keita, R. Zurob, and M. B. Giles, "High performance computing on the ibm power8 platform," in *International Conference on High Performance Computing*, Springer, 2016, pp. 235–254.

[21] *OP2 github repository*, https://github.com/OP2/OP2-Common.

[22] M. B. Giles, G. R. Mudalige, Z. Sharif, G. Markall, and P. H. Kelly, "Performance analysis and optimization of the op2 framework on many-core architectures," *The Computer Journal*, vol. 55, no. 2, pp. 168–180, 2011.

[23] M. Giles, G. Mudalige, and I. Reguly, "Op2 airfoil example," 2012.

[24] G. Mudalige, M. Giles, I. Reguly, C. Bertolli, and P. Kelly, "Op2: An active library framework for solving unstructured mesh-based applications on multi-core and many-core architectures," in *Innovative Parallel Computing (InPar), 2012*, IEEE, 2012, pp. 1–12.

[25] D. Dutykh, R. Poncet, and F. Dias, "The volna code for the numerical modeling of tsunami waves: Generation, propagation and inundation," *European Journal of Mechanics-B/Fluids*, vol. 30, no. 6, pp. 598–615, 2011.

[26] *Uk mini-app consortium*, https://uk-mac.github.io.