2014 First Workshop on Accelerator Programming using Directives

Achieving Portability and Performance Through OpenACC

J. A. Herdman, W. P. Gaudin, and O. Perks
High Performance Computing
AWE plc
Aldermaston, UK
Email: {Andy.Herdman,Wayne.Gaudin,Olly.Perks}@awe.co.uk

D. A. Beckingsale, A. C. Mallinson and S. A. Jarvis
Performance Computing and Visualisation
Department of Computer Science
University of Warwick, UK
Email: {dab,acm}@dcs.warwick.ac.uk

***Abstract*—OpenACC is a directive-based programming model designed to allow easy access to emerging advanced architecture systems for existing production codes based on Fortran, C and C++. It also provides an approach to coding contemporary technologies without the need to learn complex vendor-specific languages, or understand the hardware at the deepest level. Portability and performance are the key features of this programming model, which are essential to productivity in real scientific applications.**

OpenACC support is provided by a number of vendors and is defined by an open standard. However the standard is relatively new, and the implementations are relatively immature. This paper experimentally evaluates the currently available compilers by assessing two approaches to the OpenACC programming model: the "parallel" and "kernels" constructs. The implementation of both of these construct is compared, for each vendor, showing performance differences of up to 84%. Additionally, we observe performance differences of up to 13% between the best vendor implementations. OpenACC features which appear to cause performance issues in certain compilers are identified and linked to differing default vector length clauses between vendors. These studies are carried out over a range of hardware including GPU, APU, Xeon and Xeon Phi based architectures. Finally, OpenACC performance, and productivity, are compared against the alternative native programming approaches on each targeted platform, including CUDA, OpenCL, OpenMP 4.0 and Intel Offload, in addition to MPI and OpenMP.

I. Introduction

The OpenACC Application Program Interface (API) [1] is a high-level programming model based on the use of directives. By applying directives to original Fortran, C or C++ source code it aims to provide increased architecture portability with minimal code modification. This increase of portability is offered without compromising code maintainability, a key consideration for existing complex industrial applications. At the time of writing three compiler vendors: CAPS, Cray and PGI[1] support the initial OpenACC release.

In this paper we present both a portability and performance study of OpenACC using the hydrodynamic mini-application (mini-app) CloverLeaf, which can be found as part of the R&D 100 [2] award winning Mantevo mini-app suite [3]. To study portability we evaluate the relative performance across a wide range of supported architectural platforms for each OpenACC compliant compiler. For the performance study, each compliant compiler's implementation of OpenACC's "parallel" and "kernels" constructs are compared and contrasted. Each vendors highest performing construct is in turn compared against one another. Performance differences are investigated by analysing the application at the kernel level.

[1]As of July 2013, PGI was acquired by NVIDIA

Performance comparisons are presented and discussed, comparing the OpenACC implementation against alternative programming methodologies for those particular architectures. This is demonstrated on a kernel by kernel basis where appropriate.

The authors believe that this paper differs from other studies in open literature in that it provides a comprehensive and objective evaluation of the current implementations of commercially available OpenACC compilers, using both OpenACC's "parallel" and "kernels" constructs. Additional, these evaluations are carried out on a range of diverse and competing hardware architectures. Additionally, the OpenACC implementations available are compared against the best established native programming alternatives. No one else has compared all such variations.

In addition we discuss performance portability, that is whether OpenACC provides a level of abstraction that is essential for enabling existing large code bases to exploit emerging multicore architectures, whilst being simple and non-intrusive enough to be viable in a production environment.

The remainder of this paper is organised as follows: Section II discusses related published work in this field; Section III gives a brief overview of the CloverLeaf mini-app; Section IV details the various commercially available OpenACC implementations available and the differences in the two compute OpenACC constructs, namely "kernels" and "parallel"; Section V details the architectural details of the hardware platforms used in the study; the results of the study are presented in Section VI, and finally, Sections VII and VIII offer some discussions and conclude the paper.

This paper would not have been possible without the considerable assistance from each of the three compiler vendors: Cray, PGI and CAPS. In particular John Levesque and Alistair Hart from Cray, Inc; Doug Miles, Craig Toepfer and Michael Wolfe of The Portland Group (PGI) and Romain Dolbeau of CAPS Enterprise.

II. Related Work

Our previous work [4] describes a first comparison between OpenACC, OpenCL and CUDA for the CloverLeaf mini-application in terms of performance over multiple GPUs. The productivity, by way of development metrics, for the three programming alternatives is also considered. However our OpenACC comparison is only made under a single vendor implementation, Cray's CCE, and a single hardware architecture, a Cray XK6 with NVIDIA X2090 "Fermi" GPUs.

This was extended in [CUG PAPER] to investigate extreme scale across four generations of Cray platforms, showing the utility of hybrid MPI with OpenMP, CUDA, OpenCL and OpenACC under both PGI and Cray compilers.

An increasing number of application code developers are utilising OpenACC's directive approach to allow established industrial codes to take advantage of accelerator-enabled architectures due to the low barrier of entry. Levesque et al. demonstrate the approach to take S3D, a current MPI application, and expose greater levels of parallelism using OpenACC as the vehicle to exploit the GPUs on Oak Ridge National Laboratory's (ORNL) Titan supercomputer [5]. Whilst illustrative of OpenACC's capabilities they fail to provide a comparative performance analysis of alternative OpenACC implementations or alternative approaches to acceleration.

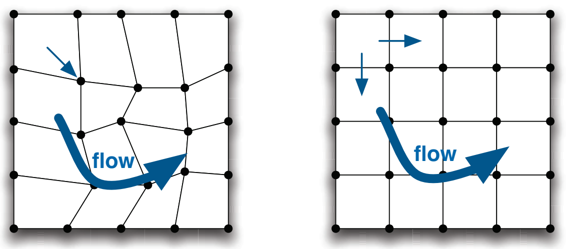
Baker et al. [6] look at a hybrid approach of OpenSHMEM and OpenACC for the BT-MZ benchmark application. The focus of the research is on hybridising the application rather than an assessment of the OpenACC implementation used (a beta version of CAPS 3.3), and results are focused on a single architecture, namely ORNL's Titan.

There are a small number of case studies presenting direct comparisons of OpenACC against alternative programming models. Reyes et al. present a direct comparison between hiCUDA [7], PGI's Accelerator model and OpenACC using their own novel implementation of OpenACC: accULL [8]. Again, this focuses on a single type of accelerator, and a single instance of an architecture: the NVIDIA Tesla 2050.

Comparison of OpenCL against OpenACC can be found in [9] by Wienke et al. The paper compares OpenACC against PGI Accelerator and OpenCL for two real-world applications, demonstrating OpenACC can achieve 80% of the performance of a best effort OpenCL for moderately complex kernels, dropping to around 40% for more complex examples. The study only uses Cray's CCE compiler and OpenACC's "parallel" construct. Also, it is limited to a single hardware architecture, an NVIDIA Tesla C2050 GPU.

III. CloverLeaf

CloverLeaf, co-authored by AWE and University of Warwick, is part of the R&D 100 [2] award winning Mantevo test suite [10], is an explicit Eulerian hydro mini-app that solves the compressible Euler equations, a series of equations describing the conservation of energy, mass and momentum in a system. The equations are solved on a Cartesian grid in two dimensions. Each grid cell stores three quantities: energy, density and pressure, and each cell corner, or node, stores a velocity vector. CloverLeaf solves the equations with second-order accuracy, using an explicit finite-volume method.



(a) Node movement during the Lagrangian step.

(b) Material movement during advective remap.

Fig. 1: Lagrangian-Eulerian hydrodynamics cycle

As depicted in Figure 1, each cycle of the application consists of two steps: (i) a Lagrangian step advances the solution in time using a predictor-corrector scheme, distorting the cells as they move with the fluid flow, (ii) an advection step is used to restore the cells to their original positions.

The computational intensive sections of CloverLeaf are implemented via twelve individual kernels. In this instance, we use kernel to refer to a self contained function which carries out one specific aspect of the overall hydrodynamics algorithm.

The authors have written CloverLeaf with the purpose of assessing emerging hardware and programming models. The mini-app represents performance and algorithmic characteristics of larger industry-based codes, which are themselves difficult to experiment with, and are commercially sensitive. The initial implementation of CloverLeaf was in Fortran 90 and was used to develop an optimised and highly vectorisable, hybrid MPI/OpenMP code. The authors ported this version to an OpenACC implementation using Cray's CCE compiler, on an NVIDIA accelerated Cray XK6. This initial OpenACC version was then used as the basis of an implementation that would compile and perform under the PGI and CAPS OpenACC compilers. This ultimately led to the two OpenACC implementations presented here, the "parallel" and "kernels" versions.

CloverLeaf's C, MPI/OpenMP hybrid, OpenCL [11], CUDA [12], Intel's Heterogeneous Offload model [13] and OpenMP 4.0 [14] implementations have been utilised, hardware permitting, to compare each of the OpenACC variants. Copies of all of CloverLeaf's implementations are available for download, and can be found under the download section of the Mantevo Suite [10]

A. Test Cases

Two representative square-shock benchmark test cases have been used from CloverLeaf's input suite. These equate to 960^2 cell and $3{,}840^2$ cell meshes respectively, simulating a small, high-density region of ideal gas expanding into a larger, low-density region of ideal gas, causing a shock front to form. Execution is for 2,955 and 87 timesteps respectively, which for their respective sizes, gives sufficient compute time to ensure reliable timing measurements.

Although CloverLeaf is capable of multiple accelerated node runs, the nature of this study is focused on single card performance.

IV. OpenACC Implementations

Prior to the support of a common OpenACC Standard, Cray, PGI and CAPS each had their own bespoke set of accelerator directives from which their implementations of OpenACC stem. A brief overview of each vendors implementation, along with limitations, follows.

Cray originally proposed accelerator extensions to the OpenMP standard [15] to target GPGPUs, through their Cray Compilation Environment (CCE) compiler suite. These evolved into the "parallel" construct in the OpenACC standard. Rather than creating a CUDA source for the kernels, CCE translates them directly to NVIDIA's low-level Parallel Thread Execution (PTX) [16], a pseudo-assembly language subsequently compiled by the graphics driver into binary code. CCE is currently only available on Cray architectures, restricting portability.

As of version 10.4 of their compiler, PGI supported the PGI Accelerator model [17] for NVIDIA GPUs. This provided their own bespoke directives for acceleration of regions of source code. In particular their "region" construct evolved into their implementation of the OpenACC "kernels" construct.

Initially, CAPS provided support for the OpenHMPP directive model [18], which served as their basis for the OpenACC standard. A major difference with CAPS is the necessity to use a host compiler. In the case of this paper the Intel compiler is used as the host. Code is directly translated into the application developers choice of either CUDA or OpenCL [11]. In the case of the latter, this increases the range of architectures which can be targeted.

OpenACC provides two constructs "parallel" and "kernels" to launch, or execute, accelerated regions. The main differences are down to how they map the parallelism in the region to be accelerated to the underlying hardware [19]. In the case of the "parallel" construct this is explicit, requiring the programmer to additionally highlight loops within the region, while in the case of the "kernels" construct this is carried out implicitly.

In the case of single loops, as in CloverLeaf's kernels, the two constructs are virtually interchangeable, with the compiler being able to automatically generate vector code for the "parallel" construct variant. With the simple re-structuring required for the PGI and CAPS implementations, this produced a single source code capable of compilation of "parallel" and "kernels" construct implementations with all three compiler vendors.

V. Targeted Architectures

Five test architectures are employed in this study:

Chilean Pine is a 40 node Cray XK6 hosted at the Atomic Weapons Establishment (AWE). Each node consisting of one 16-core AMD Opteron 6272 CPU and one NVIDIA "Fermi" X2090 GPU, with 512 cores clocked at 1.15 GHz. Although a previous generation platform, *Chilean Pine* is the only Cray architecture available to the authors containing a full range of OpenACC compilers.

Swan is primarily a Cray XC series system, provided by Cray's Marketing Partner Network. A subset is configured as an XK7 consisting of 8 nodes, each with an 8 core Intel Xeon E5-2670 and an attached NVIDIA Kepler K20X, with 2,688 732 MHz cores and 6 GB of memory. OpenACC is available on Swan via CCE 8.3.0 and PGI 13.10. Alternative approaches available to application acceleration on the system are native implementations in OpenCL and CUDA.

Shannon is a 32 node, dual socket oct-core Intel Xeon E5-2670 with either 2 NVIDIA Kepler K20Xs per node, each with 2,688 cores clocked at 732 MHz, or 2 NVIDIA K40s per node, each with 2,880 cores clocked at 745 MHz. As part of NNSA's Advanced Simulation and Computing (ASC) project, it is one of a number of advanced test-bed architectures based at Sandia National Laboratories (SNL). OpenACC implementations available on Shannon are PGI 13.9.0 and CAPS 3.3.4. Alternative approaches available to application acceleration on the system are native implementations in CUDA.

Also based at SNL is *Teller*: a cluster of AMD, second generation "Trinity", Accelerated Processing Unit (APU) processors. Each APU consists of an AMD A10-5800K (Piledriver) 3.8GHz Quad-core with one Radeon HD-7660D (Northern Islands) with on-die integration containing 384 800 MHz cores. OpenACC is provided via CAPS 3.3.3, the only alternative for exploiting the Radeon is via raw OpenCL using AMD's APP SDK 2.8.0.

Hosted at AWE, *PillowB* consists of 40 nodes of dual 2.1 GHz Intel Xeon E5-2450 processors, each node having two Intel Xeon Phi 5110P cards. OpenACC can be used on both the Xeons and the Xeon Phis via CAPS 3.3.2, using Intel OpenCL SDK v1.2.3.0. On the Xeon, alternative methodologies for comparison are OpenCL and hybrid MPI/OpenMP implementations of the min-app; while on the Xeon Phi, comparisons can also be made with OpenCL and hybrid MPI/OpenMP, additionally with the release of Intel's Fortran Composer XE 2013 Update 2 (compiler version 13.1) Intel's Heterogeneous Offload model and Intel's implementation of OpenMP 4.0's new features for controlling execution on coprocessors can also be assessed.

The targeted architectures and their system software stacks are summarised in Table I.

TABLE I: System Software Stack

Platform	Chilean Pine (XK6)	Swan (XK7)	Shannon	Pillow B	Pillow B	Teller
Architecture	X2090	K20x	K20x/K40	E5-2450	5110P	A10/7660D
OpenACC	CCE 8.1.7 PGI 13.7 CAPS 3.3.2	CCE 8.3.0 PGI 13.10	PGI 13.9.0 CAPS 3.3.4	CAPS 3.3.2	CAPS 3.3.2	CAPS 3.3.3
CUDA	5.0	5.5	6.0	N/A	N/A	N/A
OpenCL	AMD APP SDK v2.7	CUDA 5.5	CUDA 5.0.0	Intel OpenCL SDK 1.2.3.0	Intel OpenCL SDK 1.2.3.0	AMD APP SDK 2.8.0
Offload	N/A	N/A	N/A	N/A	Intel 13.1.3	N/A
OpenMP 4.0	N/A	N/A	N/A	N/A	Intel 13.1.3	N/A
Hybrid (MPI/OMP)	N/A	N/A	N/A	Intel MPI 4.1.1	Intel MPI 4.1.1	N/A

VI. Results

A. Experimental Studies

With the exception of the *Teller* AMD APU cluster (where memory constraints prevented execution of the larger test case), platform results are presented for both the 960^2 and $3{,}840^2$ test cases.

For each OpenACC implementation available on the targeted platforms, results are presented for both the OpenACC "parallel" and "kernels" construct. This allows comparison of not only each vendor's best OpenACC implementation, but also any gaps between a vendor's best and worst implementation. Additionally it highlights those implementations where collaborative efforts can be focuses with vendors, or at least implementations to be avoided.

For each alternative programming model available on the architecture in question, comparable performance figures are presented. This gives an OpenACC performance baseline against alternative approaches. Additionally comparative performance of hand-coded CUDA and OpenCL implementations on the XK6, XK7, K20x and K40, and the AMD APU are presented.

Finally, performance at the compute kernel level is examined. The aim is to identify under-performing kernels and determine if particular OpenACC features have performance issues under a particular vendor's implementation. The aim is to identify if particular kernels exhibit good or poor performance rather than the application as a whole. Subsequent analysis of these kernels can hopefully lead to particular OpenACC features that require vendor attention in their implementations.

B. Analysis

Figures 2, through 5 show the comparative performance of each OpenACC implementation, the alternative programming models available and the hand-coded CUDA and OpenCL implementations on the XK6, XK7, K20x & K40, and the AMD APU respectively.

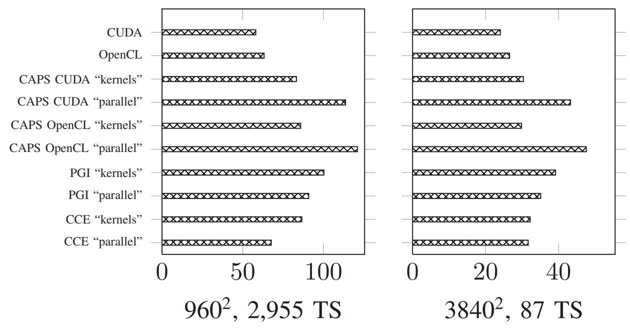


Fig. 2: XK6 Runtimes (s)

These results are summarised in Table II.

With the exception of the XK6, which uses older compiler revisions from each vendor, comparing each vendors best OpenACC implementations (be it "parallel" or "kernels" constructs, with either CUDA or OpenCL backends) performance differences are within 13%. However, choosing the least optimal OpenACC implementation can result in significant differences up to 84% when using the "parallel" construct via the CAPS compiler.

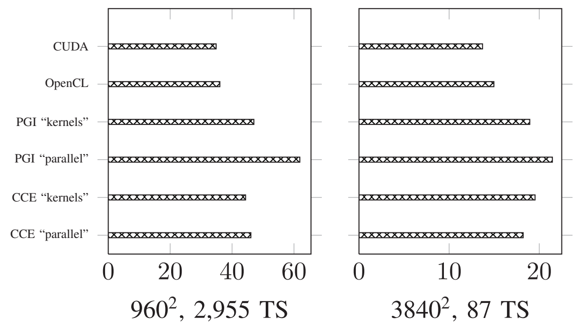


Fig. 3: XK7 Runtimes (s)

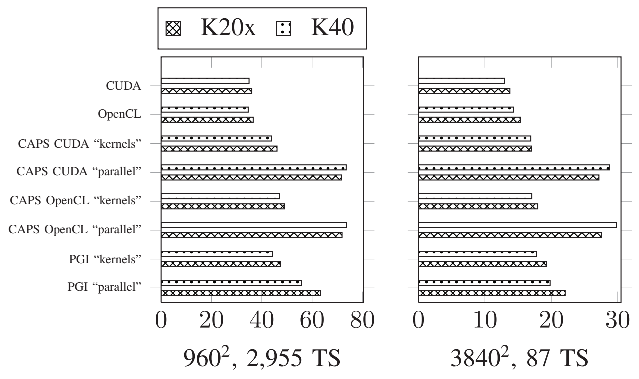


Fig. 4: K20x & K40 Runtimes (s)

As would be expected, the raw CUDA and OpenCL versions do out perform any OpenACC implementation; although only by 15% to 20% performance increase for CUDA and 10% to 20% for OpenCL. The exception to this is raw OpenCL on the APU, where differences are only around 1%. This is attributed to the raw OpenCL not being optimised for either the APU's A10 CPU or HD-7600D GPU.

Figure 6 depicts the same comparisons for the Xeon E5-2450 and the Xeon Phi 5110P. For both platforms a traditional MPI / OpenMP hybrid versions of the code (running natively on the card in the case of the 5110P) can be compared on the architecture. Additionally, in the case of the 5110P, offload models of OpenMP 4.0 and Intel's Offload are also presented.

Although the study focuses on OpenACC directives applied to Fortran source code, the perceived poor performance observed on the 5110P of the Fortran based MPI / OpenMP hybrid lead to the coding and comparison of a C-based implementation. This showed significant differences between C and Fortran which have been attributed to the fact that the Intel compiler is not vectorising the Fortran source code when compiled for the 5110P, despite accomplishing this when compiling for the E5-2450. This is also true for Intel's Offload model when using Fortran, hence in the summary of the results, the C implementations on the 5110P are taken as reference

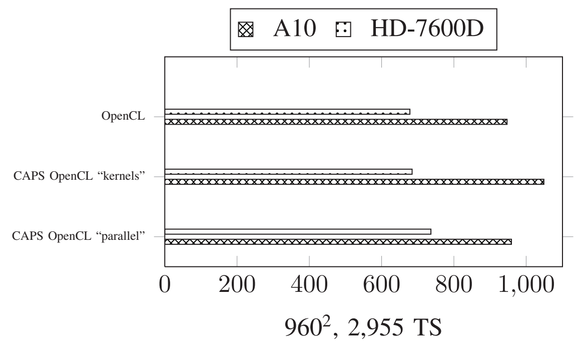


Fig. 5: A10 & HD-7600D Runtimes (s)

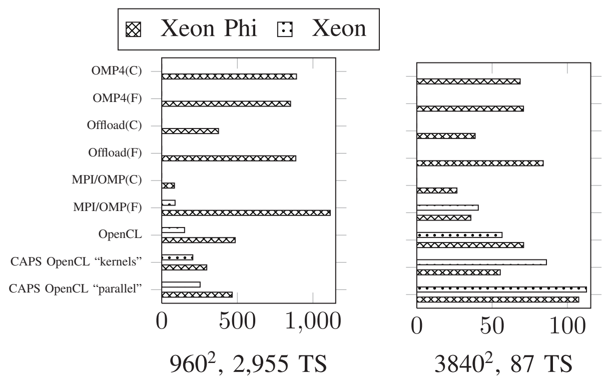


Fig. 6: E5-2450 & 5110P Runtimes (s)

TABLE II: OpenACC Performance Comparisons

Platform / Problem	Best OpenACC	Slowest Best Alt OpenACC % diff	Slowest OpenACC % diff	CUDA % Diff	OpenCL % Diff
XK6 / 960^2	CCE "parallel"	PGI "parallel" -34.31	CAPS OpenCL "parallel" -78.88	14.19	11.41
XK6 / 3840^2	CAPS CUDA "kernels"	PGI "parallel" -17.62	CAPS OpenCL "parallel" -56.79	19.46	10.95
XK7(K20x) / 960^2	CCE "kernels"	PGI "kernels" -6.04	PGI "parallel" -34.34	21.46	18.73
XK7(K20x) / 3840^2	CCE "parallel"	PGI "kernels" -4.00	PGI "parallel" -17.78	24.75	17.84
K20x / 960^2	CAPS CUDA "kernels"	PGI "kernels" -3.13	CAPS OpenCL "parallel" -56.09	21.79	20.56
K20x / 3840^2	CAPS CUDA "kernels"	PGI "kernels" -13.17	CAPS OpenCL "parallel" -61.79	19.11	9.82
K40 / 960^2	CAPS CUDA "kernels"	PGI "kernels" -0.80	CAPS CUDA "parallel" -54.40	20.41	21.16
K40 / 3840^2	CAPS CUDA "kernels"	PGI "kernels" -9.57	CAPS OpenCL "parallel" -84.13	19.83	11.55
A10 / 960^2	CAPS OpenCL "parallel"	CAPS OpenCL "kernels" -9.36	CAPS OpenCL "kernels" -9.36	N/A	1.27
HD-7600D / 960^2	CAPS OpenCL "kernels"	CAPS OpenCL "parallel" -7.60	CAPS OpenCL "parallel" -7.60	N/A	0.89

points rather than the original Fortran implementations.[2]

Table III shows the summation of results. The only OpenACC available on the Intel chipsets is via CAPS using its OpenCL backend. Hence the only OpenACC-to-OpenACC comparison that can be made is between the constructs "parallel" and "kernels". This shows a significant performance difference for both test cases, on both architectures. For the E5-2450 this is 23% for the 960^2 cell problem rising to 30% for the $3{,}840^2$ case, while the 5110P gives differences of 56% rising to 94% for the 960^2 and $3{,}840^2$ problems respectively. As previously observed the CAPS "parallel" implementation performance is significantly worse, hence with this being the only comparison these differences are to be expected.

Comparing the optimal OpenACC implementation against the raw OpenCL code on the Intel chipsets shows a very

[2]This issues has been reported as resolved in Intel 15.0

TABLE III: Xeon and Xeon Phi OpenACC Performance Comparisons

Platform / Problem	Best OpenACC	Slowest Alt OpenACC % diff	Best Alt Offload Model % Diff	OpenCL % Diff	MPI/OpenMP Hybrid % Diff
Xeon E5-2450 / 960^2	CAPS OpenCL "kernels"	CAPS OpenCL "parallel" -23.53	N/A	26.49	57.02
Xeon E5-2450 / 3840^2	CAPS OpenCL "kernels"	CAPS OpenCL "parallel" -30.87	N/A	34.22	52.56
Phi 5110P / 960^2	CAPS OpenCL "kernels"	CAPS OpenCL "parallel" -56.55	Offload -26.13 (C)	-63.18	71.75 (C)
Phi 5110P / 3840^2	CAPS OpenCL "kernels"	CAPS OpenCL "parallel" -93.97	Offload 30.09 (C)	-27.95	51.94 (C)

different pattern between the E5-2450 and the 5110P. On the E5-2450 the raw OpenCL gives a gain of either 26% to 34% depending on problem size, while on the 5110P the OpenCL gives performance *degradation* of over 60%. In the case of the 5510P this is most likely attributable to the performance portability of the hand-coded OpenCL, in that it has not been optimised for the 5510P. However, given that the OpenACC is a single source and no code modifications have taken place here either, which would point to CAPS producing an efficient Xeon Phi OpenCL backend.

All offload models perform poorly when compared against a hybrid MPI / OpenMP implementation. On the E5-2450 this is between 52% and 57%, depending on problem size, while the gains on the 5110P are 52% to 72% when using the C implementation as a reference point.

All the above analysis considers the code as a whole; that is the overall execution time, with each of its twelve of kernels accelerated. Further insight is gained if each of those kernels performance is observed in isolation. What is observed is the relative performance of each kernel can vary substantially depending on construct and vendor implementation.

Figure 7 shows the percentage difference of cumulative runtimes of each of the twelve CloverLeaf kernels: i) *Timestep*, ii) *IdealGas*, iii) *Viscosity*, iv) *PdV*, v) *Revert*, vi) *Accelerate*, vii) *Fluxes*, viii) *CellAdvection*, ix) *MomAdvection*, x) *Reset*, xi) *Halo*, and xii) *Summary*. These are based on the 960^2

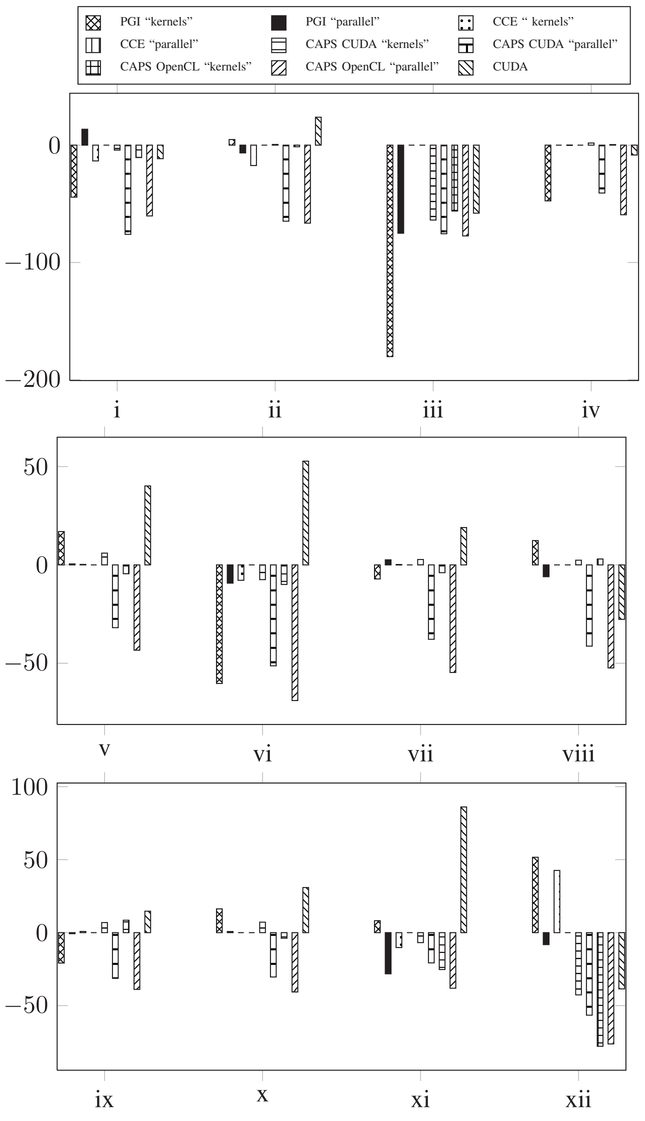


Fig. 7: Relative Kernel Performance Normalised to CCE "parallel" Construct

cell problem running on one of Chilean Pine's X2090 GPUs having all been normalised to the performance of the Cray CCE "parallel" construct time for that particular kernel. Hence, a positive percentage value indicates a shorter time spent in the kernel, and a negative a longer execution time for that particular compiler / construct pair.

Data is also available for the larger 3,840^2 cell problem, but as they show indistinguishable behaviour, they are omitted in the interests of brevity.

This closer analysis backs up some of the previous observations: CAPS implementation of the "parallel" construct underperforms on all twelve kernels irrespective of a CUDA or OpenCL backend, Cray's "parallel" implementation is equal to or outperforms its "kernels" construct on all kernels. However it also reveals that CAPS "kernels" implementation, which is the best performing OpenACC on all of the non-Cray architectures, makes these gains as it performs well on those computationally intense kernels: *MomAdvection*, *PdV*, *CellAdvection*, and *Accelerate*, which account for 60% of kernel execution time, while under-performing on those with a lesser contribution.

PGI outperforms all implementations in seven of the twelve kernels, however, these seven: *Timestep*, *IdealGas*, *Revert*, *Fluxes*, *CellAdvection*, *Reset*, *Halo* and *Summary* account for less than a quarter of execution time. There are a subset of kernels where PGI performs badly: *viscosity*, *PdV*, *accel*, *mom_advec*, which account for over 85% of kernel time, hence showing PGI's under-performance when comparing the code as a whole. Indeed, on some of these kernels the relative performance is surprisingly low. Consider the viscosity kernel, where PGI's "kernels" construct shows more than a 180% penalty compared to CCE's "parallel" implementation.

Such a disparity warrants further investigation. Both PGI and CCE provide a means to provide information regarding what the compiler has done to OpenACC accelerator regions of code. By use of the "-Minfo=accel" compile time flag with PGI and the "-ra" flag with CCE, information is sent to stderr, and a source listing file respectively.

The following snippet shows this output from both compilers for the viscosity kernel:

```
  53, Generating NVIDIA code
  55, Loop is parallelizable
  57, Loop is parallelizable
      Accelerator kernel generated
  55, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
  57, !$acc loop gang, vector(64)! blockidx%x threadidx%x


ftn-6413 ftn: ACCEL File = viscosity_kernel.f90, Line = 53
A data region was created at line 53 and ending at line 96.

ftn-6401 ftn: ACCEL File = viscosity_kernel.f90, Line = 55
A loop starting at line 55 was placed on the accelerator.

ftn-6430 ftn: ACCEL File = viscosity_kernel.f90, Line = 55
A loop starting at line 55 was partitioned across the
                                          thread blocks.

ftn-6430 ftn: ACCEL File = viscosity_kernel.f90, Line = 57
A loop starting at line 57 was partitioned across the
                         128 threads within a threadblock.
```

It indicates a difference in the maximum vector length the loop iterations are executed over. The "vector" clause is one such optional clause that is allowable on the OpenACC "kernels" construct, however in the case of the CloverLeaf kernels this clause is absent, hence the values reported by the compilers are default / assumed values. In the case of the viscosity kernel, the compiler information indicated a difference in the assignment of vector length. Further investigation across all of the twelve kernels gives a similar account for those six which PGI has disproportionate performance. Explicitly adding the "vector" clause to the "kernels" construct in these cases does now show a match between the compiler outputs, although only makes a minor change in kernel execution time.

VII. Discussions and Observations

A. Productivity

At the time of writing, the "deep copy" required to support Fortran 90 derived types was not supported in either the PGI or CAPS compilers. Hence some minor code reconstruction was required to remove derived types and explicitly pass data through the relevant Fortran routines. This also required relocating the directives higher up the call tree to reflect this reconstruction. Also under the PGI compiler, compile time for the “parallel” construct variant is significantly longer than that for the “kernels” construct.

The different basis of each vendor implementation manifested itself most notably in the varying levels of support for OpenACC’s two compute constructs “kernels” and “parallel”. Cray targeted the “parallel” construct with a direct mapping from its OpenMP “parallel” extension, while PGI and CAPS targeted the “kernels” construct with a direct mapping to the PGI Accelerator “region” construct.

It is worthy of note that the two constructs are not mutually exclusive, and can be mixed and matched depending on individual accelerator region performance, and optimal constructs could be used depending on compiler and architecture as part of an optimisation strategy.

An artefact of study highlights Fortran issues with the Intel compiler on Xeon Phi 5110P, resulting in having to use C implementation to give a more realistic comparison. Also, all offload models perform poorly when compared to hybrid MPI/OpenMP.

The authors previous work [4] assessed the programmer productivity offered by measurement of the number of words of code (WOC) added for each version, and considered whether computational kernels needed rewriting. This revealed OpenACC superior to both OpenCL and CUDA, requiring the addition of only 184 OpenACC pragmas (1,510 words of code), compared to OpenCL and CUDA versions of the code requiring an additional 17,930 and 13,085 words of code respectively. However, of the 17,930 words required by OpenCL, 3,958 can be attributed to the static class created to manage OpenCL objects. This static class could be used in other similar applications with little modification, meaning the total amount of OpenCL code unique to CloverLeaf is 13,972 words. Additionally, OpenCL and CUDA both required extra work to re-write the computational kernels in C-style code.

VIII. Conclusions

In this work, the authors set out to evaluate the performance and portability of OpenACC. This was done by comparing OpenACC’s ”parallel” and ”kernels” constructs in each vendors implementation; performance comparison between vendors; how OpenACC compares with alternative programming methodologies whilst balancing performance against programmer productivity.

By evaluating both the ”parallel” and ”kernels” constructs the results indicate that vendors have primarily focused efforts on one of the two constructs. This is most apparent in CAPS implementation of the ”parallel” construct, with 84% difference in runtime over their ”kernels” implementation.

When comparing each vendors quickest construct, differences in execution time are within 13% when the latest compiler versions are invoked. This indicates, as long as the application developer is aware of a vendors relative performance on the ”parallel” and ”kernels” construct, the choice of vendor is not so crucial.

Each vendor does have their relative merits: the CAPS’s implementation of the “parallel” construct is deficient, and their focus has concentrated on the “kernels” construct. However, the CAPS compiler is the most prevalent on the diverse architectures tested, and in some cases is the only current option to compile and execute OpenACC applications. CCE is the best performing compiler, in all but one case, on all architectures where it is available. However, it is only available on Cray systems, limiting its accessibility. Looking at overall runtimes PGI trails the other vendor’s compilers. However, when broken down into the individual kernels, PGI’s “kernels” construct outperforms the rest on more occasions than any other implementation. This is counterbalanced by its significant poor performance for one particular kernel: (*viscosity*). At 175% and 225% slower that the best implementations for the 960^2 cell and $3{,}840^2$ cell meshes respectively, this impacts when the code is considered as a whole.

Comparing to alternative offload models on the Intel Xeon Phi and Intel Xeon architectures, OpenACC outperforms both Intel’s Heterogeneous Offload model and their current OpenMP 4.0 implementations in all but the large $3{,}840^2$ test case when using the former. Although in the case of these two architectures, a hybrid MPI/OpenMP implementation outperforms all offload based programming models. On the three NVIDIA GPU based architectures a native CUDA implementation outperforms the best OpenACC by 15% to 20%. OpenCL is the only alternative programming model to OpenACC that spans all the target architectures in this study. Comparing performance, OpenCL outperforms OpenACC by 10% to 20% with the exception of the AMD APU and the Intel Xeon Phi where is performance is on par on the former and a significant under-performance of over 60% against OpenACC on the latter.

Empirically, these performances differences are more than acceptable when offset against programmer productivity measure in the number of words of code. While performance is important, for a new programming standard like OpenACC, the convergence of the standard on a range of compilers is the most important factor for portability.

CloverLeaf now has a portable single source for the “parallel” and the “kernels” construct versions that work without modification on all compilers, and a diverse range of architectures. This is major step forward and would indicate that OpenACC provides good level of abstraction.

CAPS Enterprise announced that as of June 27, 2014 they are to cease trading. This loss of one of the three vendors, and hence future CAPS OpenACC compiler support, reduces the options for OpenACC developers. However, OpenACC v1.0 support has been implemented into a branch of GCC Fortran [20], which as open source, bodes well for the standard.

Irrespective of OpenACC’s future, the development effort expended in producing a portable OpenACC implementation of an application is easily translated into alternative pragma-based offload models such as OpenMP 4.0’s accelerator offload support constructs, should this become the de-facto standard.

OpenACC has matured significantly in both its portability and performance. The ability to use a higher level language like C, C++, or Fortran on CPUs, Xeon Phi's, GPUs and APUs is a major step forward in future proofing production codes.

Acknowledgements

The authors would like to thank each of the OpenACC compiler vendors: CRAY, CAPS, PGI and NVIDIA. Thanks also goes to the Test Bed Team members at Sandia National Laboratory for their system support on the Shannon and Teller platforms. Likewise, thanks goes to Cray for their Marketing Partner Network support on the Swan platform and finally AWE's HPC system administrators for their continued support on their test bed platforms: Chilean Pine and Pillow B.

CloverLeaf is released as part of the Mantevo project hosted at Sandia National Laboratories, it can be accessed at *http://mantevo.org/*. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] "The OpenACC Application Programming Interface," http://www.openacc-standard.org, November 2011.

[2] R&D Magazine, "2013 R&D 100 Award Winners," http://www.rdmag.com/award-winners/2013/07/2013-r-d-100-award-winners, 2013.

[3] R. Barrett, P. Crozier, D. Doerfler, M. Heroux, P. Lin, H. Thornquist, T. Trucano, and C. Vaughan, "Assessing the Role of Mini-Applications in Predicting Key Performance Characteristics of Scientific and Engineering Applications," *Journal of Parallel and Distributed Computing, To appear (2014)*, 2014.

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

[5] J. Levesque, R. Sankaran, and R. Grout, "Hybridizing S3D into an Exascale Application using OpenACC," *SC12, November 10-16, 2012, Salt Lake City, Utah, USA (2012)*, 2012.

[6] M. Baker, S. Pophale, J.-C. Vasnier, H. Jin, and O. Hernandez, "Hybrid Programming using OpenSHMEM and OpenACC," *OpenSHMEM, Annapolis, Maryland. March 4-6, 2014*, 2014.

[7] D. Tianyi and T. Abdelrahman, "hiCUDA:High-level GPGPU programming," *IEEE Transactions on Parallel and Distributed System , vol. 22, pp. 7890, 2011.*, 2012.

[8] R. Reyes, I. Lopez, J. Fumero, and F. de Sande, "Directive-based Programming for GPUs: A Comparative Study," *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*, 2012.

[9] S. Wienke, P. Springer, C. Terboven, and D. Mey, "OpenACC - First Experiences with Real-World Applications," *Euro-Par 2012, LNCS, Springer Berlin/Heidelberg(2012)*, 2012.

[10] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, H. Edwards, A. Williams, M. Rajan, E. Keiter, H. Thorn-quist, and R. Numrich, "Improving Performance via Mini-applications," http://mantevo.org/.

[11] Khronos OpenCL Working Group, "The OpenCL Specification Version 1.2," http://www.khronos.org/opencl/, 2008.

[12] ——, "CUDA C Programming Guide v5.5," http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, July 2013.

[13] Intel Developer Zone, "The Heterogeneous Programming Model," http://software.intel.com/en-us/articles/the-heterogeneous-programming-model, 2012.

[14] ——, "OpenMP 4.0 Features in Intel Fortran Composer XE 2013," http://software.intel.com/en-us/articles/openmp-40-features-in-intel-fortran-composer-xe-2013, 2013.

[15] "The OpenMP API Specification for Parallel Programming," http://openmp.org/wp/.

[16] NVIDIA Corporation, "Parallel Thread Execution ISA Application Guide v3.2," July 2013.

[17] "PGI Fortran & C Accelerator Compilers and Programming Model," http://www.pgroup.com/lit/pgiwhitepaperaccpre.pdf.

[18] CAPS Enterprise, "HMPP: A Hybrid Multicore Parallel Programming Platform," http://www.caps-entreprise.com/en/documentation/caps_hmpp_product_brief.pdf.

[19] M. Wolfe, "PGI Insider: OpenACC Kernels and Parallel Constructs," http://www.pgroup.com/lit/articles/insider/v4n2a1.htm, 2012.

[20] Phoronix, "Samsung Brings OpenACC 1.0+ Support To GCC Fortran," http://www.phoronix.com/scan.php?page=news_item&px=MTU4MjQ, 2014.