Optimization of IFS Subroutine LAITRI on Intel Knights Landing

Oisín Robinson\textsuperscript{a,*}, Alastair McKinstry\textsuperscript{b}, Michael Lysaght\textsuperscript{a}

\textsuperscript{a}ICHEC, Dublin, Ireland
\textsuperscript{b}ICHEC, Galway, Ireland

Abstract

The landscape of HPC architectures has undergone significant change in the last few years. Notably, the Intel Xeon Phi architecture features a 512 bit wide vector register allowing fine-grained parallelism, with a marked improvement in memory/cache speeds in the Knights Landing variant over the original Knights Corner version. Complexity of optimization is increased due to the great variety of hardware features, where cache considerations become fundamentally important. The optimization process is greatly facilitated in this regard by the Intel Advisor tool, which not only allows traditional roofline analysis, but also new roofline variations specific to each cache level - the ‘cache-aware’ roofline model (CARM). The Exascale-driving ESCAPE project is motivated by reducing time/energy costs of numerical weather prediction in part by investigation of ‘mini-app/dwarf’ performance on emerging hardware. The LAITRI subroutine of IFS is one of the ‘seven dwarves of HPC weather codes’. It accounts for about 4% of the total runtime of IFS in production. We investigate various optimization strategies (using coarse and fine-grained parallelism among other ideas) for speeding up LAITRI. The CARM implemented by the Intel Advisor allows realistic evaluation of actual/potential performance gain.

1. Introduction

It is expected that near-future extreme-scale European HPC systems will reflect an increasing level of node-level heterogeneity as well as increasingly complex processor architectures, with more cores, wider SIMD/SIMT units, deeper memory hierarchies and the possibility of reconfiguring hardware logic. While this diversity is important to fulfil existing and emerging computational needs, it imposes challenges on the European HPC community to fully exploit the performance potential of such systems. Any model that provides insights into system performance capabilities is a valuable tool to assist in the development and optimisation of European HPC applications as well as future architectures as we face into the challenges of Exascale computing.

In order to model the interrelation between architecture capabilities and application characteristics, it is usually required to develop architecture-specific testing and simulation environments, which result in accurate but complex, difficult to develop and hard to use models. However, simpler “bound and bottleneck” approaches can provide valuable insights into the primary factors that affect system performance, and give useful guidelines for improving applications and architectures. In particular, the roofline model [2] provides insights into inherent architectural bottlenecks and potential application optimizations.

Modern multi/many core systems can be represented as a set of central processing units (cores) with an on-chip memory hierarchy connected to a DRAM memory. Hence, the original roofline model (ORM) shows the maximum attainable performance of a computer architecture as a line in the form of a “roof”. The roofline method relates the peak floating-point (FP) performance (Fp in FLOPS/s), the peak DRAM memory bandwidth (B in Beta/s), and the application’s operational intensity (I in FLOPS/Beta), where Beta represents the data traffic in DRAM bytes accessed. However, it has been recently demonstrated that this original roofline modelling concept is usually not sufficient to fully describe the performance of modern applications and architectures which rely on exploiting the on-chip memory hierarchy of multi/many core platforms. To address this issue, a new Cache-Aware Roofline Model (CARM) has been introduced [3], where a different core-centric concept involving FP operations, data traffic and memory bandwidth at different levels are perceived from a consistent architectural point of view, i.e. as seen by the core. These changes introduce a fundamentally different view on the attainable performance and behaviour of modern applications and architectures, thus providing more insightful guidelines for application optimisation. Moreover, the proposed model is a single-plot model that translates into an increase of the maximum attainable performance when directly compared to the original DRAM roof. In contrast to the ORM that ties FP performance with the bandwidth and data traffic at a

\* Corresponding author: e-mail. oisin.robinson@ichec.ie tel. +0-000-000-0000 fax. +0-000-000-0000
single memory hierarchy level, the proposed CARM concept differs in how memory traffic and bandwidth are observed. There are two key differences: (1) all memory operations including accesses to the different cache levels are accounted for; and (2) the bandwidth depends on the accessed memory levels and it is defined relative to the core. Since the proposed model considers the complete volume of memory traffic, i.e. the total number of transferred bytes, the operational intensity in CARM is uniquely defined for all levels of the memory hierarchy, thus resulting in a single-plot model.

As of 2017, the Intel Advisor tool (Beta version) offers a means of applying the CARM on Intel platforms including Intel Xeon and Intel Xeon Phi systems. This new feature provides insight beyond vectorisation (as previously provided by Intel Advisor), such as memory usage and the quality of algorithm implementation. To expose cache usage effectiveness, the Intel Advisor implementation of CARM introduces separate rooflines for all cache levels on the Intel platform through automated micro-benchmarks that are transparent to the user of the tool. A loop positioned above a particular roofline on a roofline plot means that most of the loop data fits into caches of the appropriate level. In other words, the Intel Advisor counts memory operations analysing actual instructions, not DRAM traffic. Following the general CARM methodology, optimising cache usage without algorithm rework does not move your application to the right in the roofline plot, as on ORM. In terms of counting FLOPS, the Intel Advisor uses a classic approach for computing FLOPS: (1) It counts FP operations as one operation and FMA as two operations. (2) For a SIMD instruction, it multiplies the FP operations count for that instruction by the number of vector elements used. (3) It does not count single- and double-precision FLOPS into separate metrics; it computes a single cumulative metric. (4) It counts actual executed instructions.

In order to investigate the CARM as facilitated by the Intel Advisor tool, we have chosen to work on LAITRI, one of the dwarves from IFS/Harmonie as defined by the European Horizon 2020 FET-HPC ESCAPE project. The Integrated Forecast System (IFS) is the European global numerical weather prediction (NWP) suite. Current operational grid resolution is typically 9km. A key objective of the ESCAPE (Energy-efficient SCalable Algorithms for weather Prediction at Exascale) is to improve the spatial resolution to below 2km. This is predicated on a move to Exascale computing infrastructure. Exascale computing in turn will depend on emerging hardware improving on the metrics of energy/temporal/monetary cost. For a given scientific weather model, ESCAPE adopts a ‘divide-and-conquer’ approach to algorithmic development - NWP software such as IFS is partitioned into a relatively small number of ‘mini-apps’ which are then optimised heavily. Typically, a number of target platforms, usually consisting of new/emerging hardware, are chosen and then mini-apps are specifically optimised on each platform. A further objective is to maintain platform independence as far as possible. These latter requirements are not necessarily mutually exclusive - for example, the ‘grid-tools’ software initiative aims to allow a degree of hardware-specific optimization, while not detracting from the freedom to maintain code at a more algorithmic/scientific level.

While it is clear that weather/climate codes/models are of significant interest to the European HPC community and have featured heavily as part of WP7 (Application Enabling and Support) investigations over the year, it should be emphasised that the CARM and Intel Advisor tool can target any HPC application (with the Intel Advisor implementation only available on Intel processors) and, as such, should be of interest to PRACE WP7 application experts, PRACE users and the wider European HPC community as part of efforts around extreme-scale performance.

This work is an investigation of optimization strategies for the IFS mini-app LAITRI (semi-LAgrangian Interpolation, TRI-linear) where the target platform is the Intel Xeon Phi processor. LAITRI is an interpolation subroutine which accounts for about 4% of the total time of an operational run of IFS. This is comparatively high for a single subroutine (there are many more), thus its status as a mini-app of special interest.

The structure of this paper is as follows: In section 2, we describe the function of LAITRI in sufficient detail to frame the context of the optimization, as well as give at least some sense of the subroutine’s imperviousness to algorithmic improvement. In section 3 we describe the optimization strategies used. In section 4 we give the results of our optimization attempts on all three platforms considered, namely Xeon Phi Knights Corner (KNC), the KNC host (an Intel Ivy Bridge processor), and KNL, and make comparisons. Also, we describe the Intel Vectorization Advisor and how it aids greatly the optimization process on Xeon Phi Knights Landing (KNL) in particular. In section 5 we summarize our findings and make recommendations based on our results.

2. LAITRI Function

The function of LAITRI ([1], p. 53) is to interpolate the value of a 3-dimensional physical variable (e.g. temperature/wind speed) based on boundary values in an ‘advection cube’ - the vertical projection of a mesh square from a global grid (see Figs 1, 2, 3, 4). It is a Fortran-90 program. It was originally written in 1992 and has undergone few changes, but notably in 2009 vectorization was incorporated, targeting NEC supercomputers.

2.1. LAITRI 32 points 3D interpolation

LAITRI uses linear, 4-point cubic, bi-linear and a form of 3D interpolation involving these first three in addition to ‘12-point horizontal interpolation’ to produce output quantities. The main interpolation function of LAITRI
is a hybrid of the these four methods. Referring to Fig. 4, LAITRI merely computes the interpolated value of a central point in the advection cube via a 4-point cubic interpolation of four separate stencil interpolations, corresponding to the four vertical levels in the advection cube. At level $l$ and level $l + 3$, bilinear interpolation of four central values is done, and at levels $l + 1$ and $l + 2$, 12-point horizontal interpolation is done (using the black nodes at levels $l + 1$ and $l + 2$ in Fig. 4).

2.2. LAITRI program

The Fortran 90 source file implementing LAITRI is named laitri.F90. We will examine the source, with a breakdown explaining how the various interpolation methods are implemented (see Fig. 5 for an excerpt of the main source file).

The data for the variable being interpolated originate from PXSL, a very large one-dimensional array which represents values in a three-dimensional grid, ordered to reside in a one-dimensional data structure. To facilitate fast access to this array, a separate three-dimensional array of indices KL0 is maintained, which effectively allows locating one of four western points in a stencil level of a particular advection cube - the first two dimensions of KL0 represent the vertical and horizontal advection cube numbers, and the third (fixed) dimension specifies western point 0, 1, 2, or 3 (this would correspond to points $A_0$, $A_1$, $A_2$, $A_3$ in Fig. 4). Individual point values in a stencil are accessed by a corresponding offset from the western point. There are 12 offset variables, which represent the needed horizontal positions (shared by a number of points in each case) of stencil points in a cube - in total, 2 positions in stencil level 0, a further 4 positions in each of stencil level 1 and stencil level 2, and lastly 2 positions in stencil level 3. These variables are named, for example, as IV2L1, where in this case the stencil level is 2 and the longitude is at position 1.
Recall the linear interpolation formula

\[ f(x_0) = f(a) + \left( \frac{x_0 - a}{b - a} \right) (f(b) - f(a)) \]

which computes the linearly interpolated value of the function \( f \) at the point \( x_0 \) from \( f(a), f(b) \) (the values of \( f \) at \( a \) and \( b \)), and \( a, b \). If we examine laitri.F90 to see how it relates to the notation our formula for linear interpolation, we can see that the line

\[
Z10(JROF)=PXSL(KL0(JROF,JLEV,1)+IV0L1)+PDLO(JROF,JLEV,1) \& \\
\& *(PXSL(KL0(JROF,JLEV,1)+IV0L2)-PXSL(KL0(JROF,JLEV,1)+IV0L1))
\]

accomplishes a linear interpolation where

\[ PXSL(KL0(JROF,JLEV,1)+IV0L1) = f(a), \]
\[ PXSL(KL0(JROF,JLEV,1)+IV0L2) = f(b) \]

and

\[ PDLO(JROF,JLEV,1) = \frac{x_0 - a}{b - a}. \]

In fact, the array \( PDLO \) contains the precomputed linear interpolation weight corresponding to the central longitude for a given western point. These weights are referred to as ‘zonal weights’ of a given number (western point). For example, for western point 1 of the first linear interpolation at stencil level 0, the value of \( PDLO(JROF,JLEV,1) \) will be

\[ PDLO(JROF,JLEV,1) = ZDLO1 = \frac{\Lambda_O - \Lambda_{B1}}{\Lambda_{C1} - \Lambda_{B1}}. \]

where \( \Lambda \) is a longitude on the computational sphere corresponding to a particular point in the stencil.

We note that IFS is a very large suite of routines and for our optimization efforts we chose to isolate LAITRI from the rest of the system, which required simulating its operational usage via a small driver program. The objective was to give a reliable indicator of performance in a real run of IFS by suitably loading LAITRI with similar input array sizes/inner loop trip counts/cache usage.

Fig. 5: Excerpt of source code of laitri.F90

3. Optimization Strategies

We investigate strategies based on parallelism and correct use of hardware features. In terms of parallelism, we use both coarse-grained parallelism via breaking an outer loop in the driver into multi-threaded calls via OpenMP, and fine-grained parallelism by the use of the vectorization capabilities of the target platforms, and the automatic vectorization capability of the Intel compiler. Improved hardware vectorization is a major feature of the Intel Xeon Phi platform, which for the first time features a 512-bit wide vector register, and in the case of KNL not just one but two of these, supporting the full AVX-512 instruction set (note that KNC has a similar
in terms of other optimization based on hardware features, we have considerations such as correct ‘first touch’ of program data used by OpenMP threads. This means that the data used by a particular thread should be ‘nearby’ in physical memory. This can usually be arranged by enclosing the initialization of data which is subsequently used by an OpenMP while loop in an OpenMP region in the same way. This increases memory locality in execution and usually has a noticeable effect on performance. Another example is the variety of compiler switches defining different hardware configurations. For one such example, we found that setting the KMP_AFFINITY=scatter was preferable to KMP_AFFINITY=compact.

To determine the actual impact of fine-grained parallelism via automatic compiler vectorization, we tried a number of switches. Firstly, it is possible to disable compiler vectorization altogether via the -no-vec compiler switch. We did this to allow a realistic evaluation of whether vectorization actually makes a difference. Secondly, vectorization performance is aided by correct data alignment, which is ensured via a combination of code changes - we use the -align array64byte compiler switch in conjunction with !DIR$ ATTRIBUTES ALIGN : 64 :: VAR directive for a variable VAR and the appropriately located !DIR$ VECTOR ALIGNED directive to indicate that data is properly aligned for vectorization (this gives the compiler increased freedom for aggressive optimization). Another concern here is to minimize the use of peel and remainder loops as far as possible, and the various compiler reports will inform whether the speedup due to vectorization is optimal.

A featured method of performance evaluation was to measure the total time taken for multiple iterations of calls to LAITRI. This was achieved via the built-in Fortran timer CALL SYSTEM_CLOCK(COUNT,COUNT_RATE), placed at the start of the outer loop of the driver, while a second call to this function, placed after the timed loop, populates the COUNT variable with the time taken. This timer has sufficient resolution at the timescale of interest, which is of the order of milliseconds.

We remark that LAITRI has a built-in ‘vectorization’ section, which originally targeted NEC supercomputers some years ago, but as this section is well-suited algorithmically to vectorization on Xeon Phi (it amounts to a doubly-nested loop which is easy to work with), we simply enabled this section exclusively for our work.

LAITRI is typically called for each material variable (e.g. temperature/pressure), and on modern systems the magnitude of the loop trip counts can range from 10-200 or so, when atmospheric chemistry modules are used. In current codes (IFS, Harmonie) LAITRI is called sequentially for each variable, but scope exists for parallelisation of these calls via OpenMP.

4. Results

In this section we provide a performance analysis of the LAITRI mini-app running on three separate Intel platforms: (1) 2 x 10-core Intel Xeon E5-2610v2 Ivy Bridge (IVB) node, (2) 1 x 60-core 5120P Knights Corner (KNC) coprocessor and (3) 1 x 64-core 7210 Knights Landing (KNL) processor. We begin by showing the strong-scaling of the LAITRI mini-app over increasing OpenMP thread count on each of the platforms in Figures 7, 8, 9, where it can be seen that the LAITRI mini-app scales well over increasing thread count. It should be pointed out that while we have endeavoured to ensure that our investigations on strong scaling reflect realistic problem sizes, due to the limited nature of mini-apps in replicating real-world performance challenges, we cannot guarantee that the LAITRI mini-app faithfully mirrors LAITRI’s compute and memory footprint as found within IFS and other weather codes. Nevertheless, the LAITRI mini-app results as reported here do offer some guidance on where performance challenges lie for LAITRI within full-scale weather codes, particularly with regard to vectorisation efficiency on the single core level. In Fig. 9, the KNL scaling graph, we show an extra detail - the effect of increasing the OpenMP thread count per core, where it can be seen that exploiting the extra hardware threads per core on the KNL leads to measurable performance improvement.

Next, we compare the best time-to-solution data, for various different hardware settings, and with thread count being selected constant across each platform, at the optimum point of the scaling plots. We experimented with the following settings: without automatic vectorization (by compiling with the -no-vec switch in ifort), with vectorization, and with data alignment at 64 byte boundaries.

![Fig. 6: Time to Solution for Platforms](image-url)
We now turn to the central focus of our investigation of the emerging Intel Advisor CARM tool. In Fig. 10 we see the CARM plot generated by the Intel Advisor tool for a given run of the LAITRI mini-app on the KNL platform configured for flat memory mode and quadrant cluster mode (note that these are not mutually exclusive. Flat memory mode just means that the MCDRAM on the KNL is not used as cache. Cluster mode relates to memory ‘address affinity’ and can be set to take advantage of memory access patterns). The red dot corresponds to the measured operational intensity of the loop under examination. Comparing Figs 10, 11, 12, we see the difference that using all available cores makes to the actual operational intensity - as expected, it is orders of magnitude higher for the many-core case (100 GFlops/byte vs 1 GFlop/byte). One useful feature of the Intel Advisor CARM tool is its ability to generate ceilings for the single-thread case as well as for multiple threads, so that we can analyse the actual single thread LAITRI performance relative to realistically obtainable single thread platform ceilings. In Fig. 11, we see that these single thread (diagonal) ceilings correspond to: DRAM-L2 bandwidth (8.85 GB/sec), MCDRAM-L2 bandwidth (14.54 GB/sec), L2-L1 bandwidth (51.65 GB/sec) and L1-core bandwidth (155.19 GB/sec) measurements. We also see single thread (horizontal) ceilings corresponding to peak performance achievable for optimal scalar addition, vector addition and fused multiply-add (FMA) benchmarks.

In Fig. 11, we also see the measured operational intensity of the loop of interest within our LAITRI mini-app, seen as the red dot on the CARM plot, corresponding to an L1 operational intensity of 0.11 FLOPS/Bytes and a throughput performance of 1.44 GFLOPS/sec. Importantly, unlike the ORM, which accounts for the memory traffic for a specific memory level (e.g. DRAM), CARM accounts for the total memory traffic as seen by the core. As a result, unlike in the ORM, where the operational intensity of a given function/loop will shift according the data set size or memory optimisation, the operational intensity as measured in the CARM methodology will remain fixed on the roofline plot. Instead, memory optimisations (or the effects of varying data size) will result only in the throughput performance changing in the vertical direction, relative to the ceilings associated with each of the memory levels, giving a much clearer view of the actual limiting factors of performance than via ORM.

In Fig. 11, we have highlighted (in bold) the ceilings associated with the scalar addition and L2 bandwidths on the KNL platform, as our measured point lies below each of these ceilings. The fact that our measured point lies closer to the vector addition ceiling indicates that performance could first be improved by increasing the vectorisation efficiency of the loop in question. This is unsurprising, since the interpolation loop is dominated...
by indirect array accesses resulting in performance-limiting gather operations. Beyond this, the next ceiling to break through is the L2 ceiling, which would require better data reuse in L1.

Fig. 10: Intel Advisor screenshot

Fig. 11: Intel Advisor screenshot, single-thread case
In Fig 12, we show the CARM plot as generated by the Intel Advisor tool for the multithreaded case, where we run on 64 cores, with 4 OpenMP threads per core. Firstly, observe the new multithreaded ceiling values used as bounds instead of the single thread ceilings. These correspond to DRAM-L2 bandwidth (75.77 GB/sec), MCDRAM-L2 bandwidth (409.83 GB/sec), L2-L1 bandwidth (1653.18 GB/sec) and L1-core bandwidth (9930.25 GB/sec) measurements. Secondly, as expected from the CARM, the operational intensity of the loop in question remains the same as in the previous single threaded run at 0.11 FLOPS/Byte. It can be seen that the throughput performance is now measured at 102 GFLOPS/sec and once again the limiting factors are the vectorisation efficiency and L2 bandwidth. On the other hand, with the overall limiting factor being L2 bandwidth, the achievable performance as indicated by the CARM would be approximately 220 GFLOPS/sec.

Fig. 12: Intel Advisor screenshot, multithread case

<table>
<thead>
<tr>
<th>Platform</th>
<th>Clock [GHz]</th>
<th>No. cores</th>
<th>Threads/core</th>
<th>L1 Cache</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
<th>DRAM Type</th>
<th>SKU</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 × Ivy Bridge</td>
<td>2.20</td>
<td>20</td>
<td>2</td>
<td>32kb</td>
<td>256kb</td>
<td>25Mb</td>
<td>64Gb DDR3</td>
<td>E5-2660v2</td>
</tr>
<tr>
<td>1 × Knights Corner</td>
<td>1.053</td>
<td>60</td>
<td>4</td>
<td>32kb</td>
<td>512Kb</td>
<td>-</td>
<td>8Gb DDR3</td>
<td>5110P</td>
</tr>
<tr>
<td>1 × Knights Landing</td>
<td>1.3</td>
<td>64</td>
<td>4</td>
<td>32kb</td>
<td>30Mb*</td>
<td>16Gb†</td>
<td>≤384Gb DDR4</td>
<td>7210</td>
</tr>
</tbody>
</table>

* shared, † MCDRAM

Table 1: Optimization platform hardware specifics

5. Summary

We have presented our optimization strategy for the heavily-used IFS subroutine LAITRI on the Intel Xeon Phi platform, with a comparison to Ivy Bridge, with a particular focus on the Cache-Aware Roofline Model as implemented by the Intel Advisor tool, as a means of determining actual/potential performance gain. The results show that improvements in time-to-solution are incremental for any given optimization on a fixed platform, with no particular setting giving considerable speedup compared to others. However, there is a marked performance boost on the KNL platform over both KNC and the Intel Ivy Bridge Xeon processors tested. This is a turnaround from previous work comparing KNC to Ivy Bridge, where it was found (with similar results elsewhere in HPC) that the KNC performance was disappointing. Not only does KNL surpass KNC (by an order of magnitude in our tests), it also outperforms Ivy Bridge by 2 to 1 in time to solution. The settings to achieve this are straightforward, involving 3 main ideas - correct use of OpenMP (e.g. ensuring correct ‘first touch’ of data), suitable compiler data alignment directives to facilitate good vectorization performance, and prudent setting of runtime variables such as KMP_AFFINITY and KMP_HW_SUBSET. In contrast to a paradigm that is much more difficult to arrange like data preconditioning, for example, which would not only involve local code changes, but possible higher-level structural code changes also, making that approach in particular unrealistic, although
we have dealt mainly with hardware specifics, it should not be difficult to integrate these kinds of settings into grid-tools.

Acknowledgements

This work was supported by the PRACE project funded in part by the EU’s Horizon 2020 research and innovation programme (2014-2020) under grant agreement 653838. The work was achieved using the PRACE Research Infrastructure resources at the CURIE (GENCI), the FERMI (CINECA), and the SuperMUC (LRZ) supercomputers.

References