OPTIMISING CASTEP ON INTEL'S KNIGHT'S LANDING PLATFORM TECHNICAL REPORT FOR ECSE11-17

April 1, 2019

Phil Hasnip[†], Ed Higgins[†] & Arjen[‡] Tamerus

[†] Dept of Physics, University of York

[‡] Research Computing Services, University of Cambridge

Abstract

CASTEP is a widely-used, UK-developed software package based on density functional theory, and capable of predicting the properties of materials from “first-principles”; that is, by solving quantum mechanical equations to determine what the behaviour is, without the need for adjustable parameters. CASTEP was designed from the beginning to run well on conventional parallel HPC machines, but in recent years a number of new computer architectures have emerged which do not follow the conventional trends for CPUs. One such architecture is Intel’s Knights Landing (KNL).

Knights Landing’s theoretical performance is very high, but much of its performance is delivered by long vector instructions on many low-power cores, and its performance profile differs considerably from that of a conventional CPU. In this work we profiled and analysed CASTEP’s performance on KNL, with particular attention to its vectorisation, and optimised the computational bottlenecks. The effects of hyperthreading and the KNL memory mode were also investigated. Substantial performance gains were realised in several key subroutines, and CASTEP performance on KNL was improved considerably.

Contents

Introduction

Methods for performing first-principles simulations of materials (solving the Schrödinger equation via parameter-free approximations) have had a profound, pervasive impact on science and technology, spreading from physics, chemistry and materials science to diverse areas including electronics, geology and medicine. Methods based on density functional theory (DFT) have led the way in this success, offering a favourable balance of computational cost and accuracy. However, DFT is certainly not a computationally 'cheap' method and it is essential to optimise codes not only for single-core performance but also for large parallel calculations.

CASTEP is a UK-based state-of-the-art implementation of DFT and a flagship code for UK HPC. It was designed from the outset according to sound software engineering principles and with HPC in mind. CASTEP describes the electronic states ("bands") using a plane-wave (Fourier) basis, necessitating the heavy use of parallel fast Fourier transforms (FFTs) for computational efficiency. CASTEP is widely-used by academia and industries across the world, and is available to all UK researchers under a free licence. It is a system-installed program on ARCHER, as it was on its predecessor HECToR, and is frequently in the 'Top 5' most used codes (as measured by aggregate core hours). CASTEP is written in modern Fortran 2003 and parallelised using MPI and OpenMP. Strict adherence to language/API standards is important, as CASTEP is widely ported to many systems worldwide, from PCs to Tier 0 HPC.

The work presented here is aimed at boosting the efficiency and scaling of CASTEP on modern CPUs, including Intel's KNL, thus enabling efficient calculations with greater use of concurrency; this in turn enables far more energy-efficient calculations, and provides some measure of "future-proofing" for CASTEP as KNL is an archetypical machine for Intel's future HPC platforms.

1 Optimising bottlenecks

1.1 Comparative benchmarking

A simple way to put the performance of the Knights Landing (KNL) hardware into perspective is to run a comparative benchmark on a single-node of ARCHER's KNL and standard Ivy Bridge compute partitions. CASTEP's `TiN` benchmark was chosen, as this is a relatively small calculation suitable for running on a single node, yet is large enough that the parallel efficiency remains high even with all 64 cores of the KNL. The relevant sizes are:

- 8 integration sampling points ('k-points')
  CASTEP parallelises almost perfectly over k-points, and both the Ivy Bridge and KNL have a multiple of 8 cores per node (24 and 64 respectively).

- 164 electronic states ('bands')
  These are sufficient to accommodate all of the electrons in the 33-atom simulation cell, as well as some extra conduction states. Having over 100 bands means that there is a reasonable amount of data and work per k-point, so a high compute:communication ratio is expected.

- 10,972 Fourier components ('plane-waves')
  CASTEP additionally distributes the data and workload over the plane-waves, and parallel efficiency typically remains high until there are substantially fewer than 1,000 plane-waves per MPI process.

These sizes suggest that high parallel efficiency should be achieved for up to 8-way k-point parallelism and up to 11-way G-vector parallelism, meaning calculations up to $8 \times 1 = 88$ MPI processes should be very efficient. Since this exceeds the single-node core counts of both architectures, communication overheads are expected to be low and parallel efficiency high.

1.2 Benchmark results

The results of running the `TiN` benchmark are shown in table 1, along with a selection of the per-routine timings from CASTEP's built-in trace module. The total execution time (as measured by wallclock time) is substantially higher for the KNL, and a full per-routine breakdown showed that of the 426 subroutines called and traced, the KNL was faster in only one: the matrix-multiplication routine `algor_matmul_cmplx_cmplx`. However many of the subroutines did not slow down substantially; rather, the increase in wallclock time on the KNL was dominated by two subroutines: `trace_entry` and `nlpot_calculate_d`.

1.2.1 `trace_entry`

This routine is not involved in any of the scientific simulation, but is CASTEP's main timing and profiling routine. Its operation involves a hash table look-up, traversal of a doubly-linked tree (using Fortran pointers) and a call to OMP_GET_WTIME for the timing itself. All three of these operations are substantially slower on the KNL compared to the Ivy Bridge. A simple work-around is simply to disable these calls, since they are only used for execution tracing and profiling and have no scientific function; however, since the overall aim of this project was to improve execution time, good profiling data is essential and disabling this functionality would be counterproductive. A pragmatic compromise was to disable the calls in all of the low- and mid-level subroutines (those subroutines in CASTEP's 'Utility' and 'Fundamental' modules). This sacrificed the detail of the profiling data, but recovered over 80% of the slow-down in wallclock time for this operation, compared to Ivy Bridge. This large reduction in the overhead is due primarily to the fact that the lower-level subroutines are called many more times than the high-level ones, often 10,000-100,000 times even in short calculations.

Operation	Ivy Bridge	KNL
Total calculation	246s	427s
`algor_matmul`	96s	72s
`nlpot_calculate_d`	10s	61s
`trace_entry`	2s	121s

Table 1: Comparative timings for 32 SCF cycles of the TiN benchmark on one node of ARCHER (CPU or KNL) using CASTEP timing routines to provide per-routine timing. The benchmark was performed using the baseline CASTEP 17.2 codebase.

1.2.2 `nlpot_calculate_d`

This subroutine computes the weights for CASTEP's non-local projectors, which is essential to determining the potential acting on the electrons in the simulation. Analysis of the operations showed that a large proportion of the time was spent interpolating the projectors themselves, and that this time was spent computing a set of 'Clebsch-Gordon' coefficients and applying spherical harmonics to the projectors. Each of these operations depends on several integer arguments (quantum numbers); although the operation involves looping over all valid values of the integers, the data in each case was computed on a per-element basis. This is slightly inefficient on a conventional CPU, but the operation is quick enough not to be significant; on the KNL, however, it is extremely time-consuming.

Refactoring the code so that the innermost loop was moved inside the lowest-level routine, and removing some redundant checks for argument validity, transformed the performance of these operations, and reduced the time taken by `nlpot_calculate_d` from 61s to 23s (a 2.7× speed-up).

1.3 Comparison conclusions

When compared with the Ivy Bridge CPUs, the KNL shows best performance on dense linear algebra operations, particularly matrix-matrix multiplications. Most of the other operations have similar, if slightly poorer, performance when comparing node-for-node performance. Operations which stress the branch prediction, for example conditionals inside tight loops, and system calls appear to be significantly slower, but in the case of CASTEP at least, many of these may be mitigated by relatively modest code refactoring.

Following implementation of the optimisations discussed so far, the `TiN` benchmark was re-run and the walltime for 32 SCF cycles was shown to have reduced from 427.24s to 287.13s, a speed-up of 1.49×.

2 Vectorisation

The ability to perform vector operations is a feature of almost all modern CPUs, but for KNL in particular it is key to achieving good performance. In a typical vector operation, a single instruction is used to perform an operation on multiple data (SIMD) concurrently. It is a requirement of these instructions that there are no data-dependencies, otherwise the concurrency could result in a race condition. It is not always possible for a compiler to determine whether or not data dependencies exist, and in such cases the compiler will assume the code cannot be vectorised.

While it is the programmer's responsibility to write vector-friendly code, exposing suitable parallelism is not always trivial nor intuitive. Even when the compile can identify vector-parallelisable code, the compiler will have to be conservative in its use of SIMD instructions to ensure the vector units operate on contiguous data and to avoid out-of-bounds memory accesses. We will discuss the methods used to identify suboptimal vectorisation, and the solutions that were implemented to resolve the issues that were found. We will particularly focus on the the issue of indirect memory copies, a heavily used operation in CASTEP which has shown to be a particularly complex task to vectorise.

2.1 Comparison of AVX instruction sets

A simple way to investigate the effects of vectorisation is to disable the longer vector instructions explicitly at compile-time. The results of running CASTEP compiled with three different levels of vector instruction set at shown in table 2, where the total run time is given alongside the timing breakdown for some notable subroutines; note that the instruction sets used by the libraries were not changed, so any calls to the BLAS/LAPACK or FFT will use the same instructions regardless of the setting.

It is clear from the data in table 2 that whilst CASTEP as a whole speeds up as each extra instruction set is included, the overall speed-up is not very large and the performance varies enormously from operation to operation. The matrix multiplication subroutine `algor_matmul_complex_complex` is a thin wrapper to a BLAS library call (ZGEMM), and since the library instruction set is unchanged the performance is approximately constant. One of the best-performing operations is `wave_zero_slice`, which is used to initialise instances of one of CASTEP's derived types, mostly by filling a large array with zeroes.

The remaining two routines highlighted show considerably poorer performance. The FFT routine `basis_recip_red_real_3d_coeffs` shows no benefit from the additional vector instructions at all, and `comms_transpose_exchange` actually slows down significantly as the vector lengths are increased. Each of these operations involves an indirect memory copy; this operation is the subject of the next section.

Routine name	AVX	AVX2	AVX512
`Total time`	1112.30	1089.14	1066.22
`comms_transpose_exchange`	169.02	177.43	192.29
`algor_matmul_complex_complex`	349.60	351.91	354.63
`wave_zero_slice`	100.56	65.83	38.64
`basis_recip_red_real_3d_coeffs`	27.82	28.05	27.60

Table 2: Comparison of the performance of notable CASTEP routines running the `Al3x3` on one node of KNL when compiled with different AVX instruction sets. Since CASTEP was linked to MKL for BLAS operations, routines such as `algor_matmul_complex_complex` were able to utilise AVX512 instructions in all runs.

2.2 Indirect memory copy

There are two common operations in CASTEP which involve an indirect memory copy:

- FFT data mapping
- Parallel data transposition

In each case the copy is of the form:

```
do n=1,max_n
  a(b(n)) = c(n)
end do
```

(or the inverse operation.)

Such operations can be difficult for compilers to vectorise, since it is not clear at compile-time whether the mapping is one-to-one. In fact in each case in CASTEP the indirection provides a unique mapping from element n of c, to element b(n) of a (and vice versa).

2.2.1 FFT data mapping

The wavefunction data in CASTEP is nonzero only inside a sphere in Fourier space. Furthermore, the diameter of this sphere is typically only half the length of the FFT grid. Thus for a cubic FFT grid of side 2L, the number of FFT grid points is $(2L)^3 = 8L^3$, but the nonzero elements lie in a sphere of points $\frac{4}{3}\pi(\frac{L}{2})^3 \approx \frac{1}{2}L^3$, meaning only $\frac{1}{16}$th of the FFT data is nonzero. Since the wavefunction data is one of the largest memory objects in a CASTEP calculation, the nonzero elements are stored in a contiguous array. When CASTEP needs to FFT the data to real-space, therefore, it must first be mapped onto the full FFT grid.

The Fortran code to do this is of the form:

```
do point=1,num_sphere_points
  FFT_grid(sphere_to_grid(point)) = sphere_data(point)
end do
```

The FFT operation may now be carried out on the data in FFT_grid.

Whenever the inverse operation is required, i.e. taking real-space data on the FFT grid and transforming it to reciprocal-space data within the sphere, the FFT is carried out on the FFT_grid to transform the data to Fourier-space, and then a similar indirect memory copy is used to only copy the data lying within the sphere:

```
do point=1,num_sphere_points
   sphere_data(point) = FFT_grid(sphere_to_grid(point))
end do
```

Although it is not necessarily obvious to the compiler, the array sphere_to_grid is a one-to-one mapping which simply takes the index of a data point in the sphere representation and returns the index in the full FFT grid representation. Since there are no data dependencies, this operation is inherently parallelisable and vectorisable.

2.2.2 Parallel data transposition

In parallel, CASTEP performs its 3D FFT as a series of 1D FFTs (along each direction x, y and z in turn) interspersed with explicit data transpositions. In Fourier space, the 3D data is distributed via a 2D decomposition over y and z, with each MPI process taking one or more "x-columns"; that is, all of the data with (x,y,z) coordinates from (1,y,z) to (N_x,y,z). This allows each process to perform its own 1D FFT in x, without requiring communication. In order to perform an FFT in y the data must be transposed (re-distributed) so that the process now hold "y-columns", i.e. columns of data from (x,1,z) to (x,N_y,z).

The parallel data transposition is carried out by `comms_transpose_exchange`, which uses `MPI_AlltoAllV` for the interprocess communication. As the name suggests, this involves all MPI processes communicating with all other processes, with each process setting up separate send and receive buffers. The send buffer must be arranged so that the data for each recipient process is contiguous, so `comms_transpose_exchange` uses an indirect memory copy to place its own FFT data (arranged contiguously in x) into the send-buffer (to be ordered contiguously in y).

An exactly analogous set of operations is then performed (by the same routines) to transpose the data again such that the MPI processes hold "z-columns", and the last FFT (along z) may be performed.

2.2.3 Compiler hints

For less complex operations than the indirect memory copy the compiler can often successfully detect the vectorisability of a section of code, but it cannot determine at compile time that the prerequisites for safe vectorisation are *guaranteed* to be met.

Fortran and OpenMP both contain constructs to signal to a compiler that code may be executed concurrently. The indirect memory copies are of the form:

```
do point=1,num_sphere_points
   sphere_data(point) = FFT_grid(sphere_to_grid(point))
end do
```

and in this case the relevant constructs are:

- Vector indexing (Fortran 90)
  This may be implemented in one of two (syntactically equivalent) ways. Method 1 is:

  ```
  sphere_data(:) = FFT_grid(sphere_to_grid(:))
  ```

  and method 2 is:

  ```
  sphere_data = FFT_grid(sphere_to_grid)
  ```

- forall (Fortran 2003)

  ```
  forall (point=1:num_sphere_points)
     sphere_data(point) = FFT_grid(sphere_to_grid(point))
  end forall
  ```

- do concurrent (Fortran 2008)

  ```
  do concurrent (point=1:num_sphere_points)
     sphere_data(point) = FFT_grid(sphere_to_grid(point))
  end do
  ```

- SIMD (OpenMP)

  ```
  !$OMP SIMD
  do point=1,num_sphere_points
     sphere_data(point) = FFT_grid(sphere_to_grid(point))
  end do
  ```

In order to investigate these different options, a stand-alone micro-benchmark code was developed which performed identical indirect memory copies using each of these options. The results are presented in table 3. Despite the similarity of these constructs in their intent, the actual performance differs considerably. The original, explicitly-coded loops show the worst performance (this is consistent across a range of problem sizes, architectures and compilers). The best performance is obtained with the 'do concurrent' construct, although the similar 'forall' also performs well. The performance of the remaining methods, namely the OpenMP SIMD and the two nominally-equivalent vector indexing methods, were somewhat poorer, though they still out-performed the original code (around 1.6 times faster). It is not clear why the formally-equivalent vector indexing methods show different performance, though this difference is consistent and repeatable. The explicit ':' syntax is more flexible, potentially allowing a subset of the data to be copied, and it may be that the compiler generates more general code in this case.

Additional compiler hints are often available via compiler directives. A particularly useful directive for the Intel compiler is the 'vector aligned' hint:

```
!DIR$ VECTOR ALIGNED
do concurrent (pw=1:npw)
```

This directive guarantees that the memory accessed in the loop following it is aligned to vector boundaries.

Method	Wall time (s)
Explicit loops	2.4651
Vector indexing (method 1)	1.5954
Vector indexing (method 2)	1.5512
Forall	1.4929
do concurrent	1.4755
OMP SIMD	1.5415

Table 3: Performance of the indirect memory copy in the micro-benchmark code, using a variety of different compiler hints. The copy was equivalent to a small simulation with 57,747 plane-waves, and the timing was reported for 33,908 repeated copies to reduce timing noise. Results are reported for a 3.7 GHz Intel Core i7-8700K CPU using a binary compiled with gfortran 5.4.

Analysing the memory pattern of the indirect copy showed that the indirection does not cause a random access pattern, but rather that the access is piecewise contiguous. The data in the sphere representation is distributed in parallel by columns parallel to the x-axis ('x-columns'), so that the data on each MPI process actually represents a collection of x-columns with different coordinates in y and z. When the copy is working on data within the same x-column, the copy from the sphere- to the FFT grid-representation is contiguous and the only jumps occur when the last element of an x-column is copied, and the next element to be copied belongs to a different x-column.

Understanding the memory access pattern allows a modification to the indirection. Instead of storing the sphere-to-grid mapping on an element-by-element basis, the mapping could instead be performed for the first element of each x-column, with a secondary array containing the information on how many elements are in this x-column. The actual memory access pattern is unchanged, but this change in the mapping data allows the code to be rewritten to make its piecewise contiguous nature explicit, e.g.

```
point = 1
col_point = 0
do col=1,num_columns
  do concurrent (column_start=0:num_points_in_column(col)-1)
    sphere_data(point+col_point) = FFT_grid(offset(col)+col_point)
  end do
  point = point + num_points_in_column(col)
end do
```

This refactoring showed improved performance for both GNU and Intel compilers, across a range of CPUs. Table 4 shows the performance obtained using GNU's gfortran 7 and Intel's ifort 19 on a Kaby Lake CPU. The original code is significantly slowed on gfortran compared to ifort. Not only is the optimised, piecewise-contiguous code the fastest on both compilers, its performance is also much more consistent between the compilers. Table 4 also shows the final speed-up achieved with respect to the original code (normalised separately for each compiler): the speed-up is 1.4× for gfortran and 1.24× for ifort.

Code	Performance per copy			
	gfortran 7		ifort 19	
Explicit loops	0.206s	1.00×	0.178s	1.00×
Vector indexing	0.157s	1.31×	0.153s	1.16×
do concurrent	0.157s	1.31×	0.152s	1.17×
OMP SIMD	0.175s	1.18×	0.152s	1.17×
Piecewise contiguous	0.147s	1.40×	0.143s	1.24×

Table 4: Performance of the indirect memory copy in the micro-benchmark code, for two different compilers on an Intel Kaby Lake CPU. The timing is reported per copy, and accompanied by the speed-up relative to the original code.

2.3 Identifying vectorisation problems

Despite the use of vectorisation hints and directives, unless explicit vector intrinsics are used the compiler will ultimately decide whether code will be vectorised or not. We used the Intel compiler's vectorisation report feature to investigate whether, and why, a loop was vectorised or not. The compiler estimates the computational efficiency for both unvectorised and vectorised version of a loop, and will only vectorise if the vector cost is lower than the scalar cost. The report will show hints to inform why a loop may not be (effectively) parallelised, such as data dependencies or unaligned memory accesses.

It is not uncommon for a vectorised loop to perform worse than a scalar version, even if the optimisation report estimated a significantly higher performance. To investigate the cause of this unexpected behaviour we performed traced execution of the code and mapped the results to the underlying assembly. This analysis showed that AVX512 gather and scatter instructions were issued in loops. These instructions appeared to stall the execution pipeline, causing the slowdown. By enforcing memory alignment the scatter and gather instructions were prevented from being issued, avoiding the execution stall.

2.4 Hubbard U

When a larger range of benchmarks was considered, vectorisation was found to be particularly poor in simulations which included a Hubbard U potential term (DFT+U simulations). The code in question contained multiply-nested loops with several conditionals, and several linear algebra operations written in explicit code. The subroutines were refactored and reworked to move the conditionals outside the loops where possible, precompute some data and transform the linear algebra into a form suitable for level 3 BLAS calls.

Not only was the optimised code more than twice as fast on the KNL, it was also found to be significantly quicker on other CPUs. Benchmarking on a 3 GHz Intel Skylake CPU (E3-1220 v5) showed a typical speed-up of 15% *for the entire calculation.* For example, a simulation of the Heusler alloy Co_2MnSi (16-atom unit cell) with a Hubbard U applied to Co and Mn took 34 seconds per SCF cycle with the original code (using 3 MPI processes), but this reduced to 29 seconds following the optimisations.

3 Memory Modes

One of the novel, and potentially high performance, aspects of the KNL processor is the on-chip high bandwidth memory (HBM) it provides. The KNL has 16 GB of high bandwidth memory on processor, that can provide ~4x-5x higher bandwidth than standard main memory, although at a slightly higher latency. This means that it can provide significantly higher performance for data that is reused or accessed in a predictable pattern (e.g. the STREAMS benchmark reports a bandwidth of around 450 GB/s on the ARCHER KNL processors using HBM compared to around 90 GB/s when only using main memory), but can reduce performance for random memory access patterns or small arrays (the latency

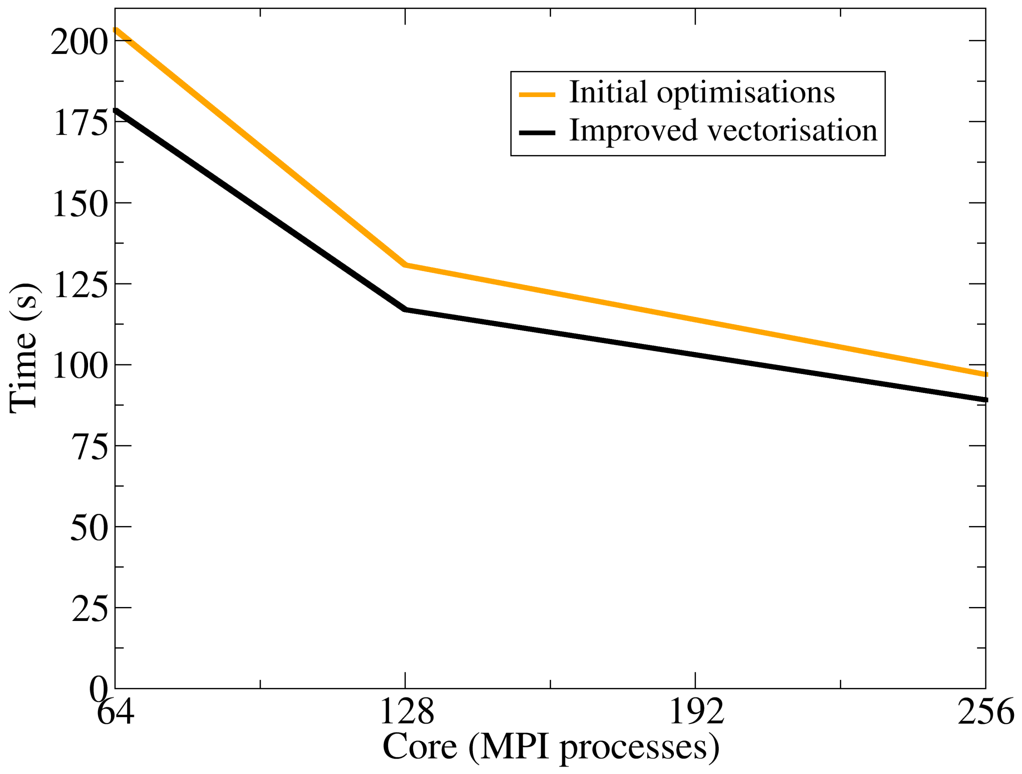

Figure 1: Performance of the TiN benchmark for CASTEP (version 17.21) with the initial optimisation of bottlenecks (see section 1) and the version with additional improvements to the vectorisation. The time is reported as the wall-clock time for 20 SCF cycles.

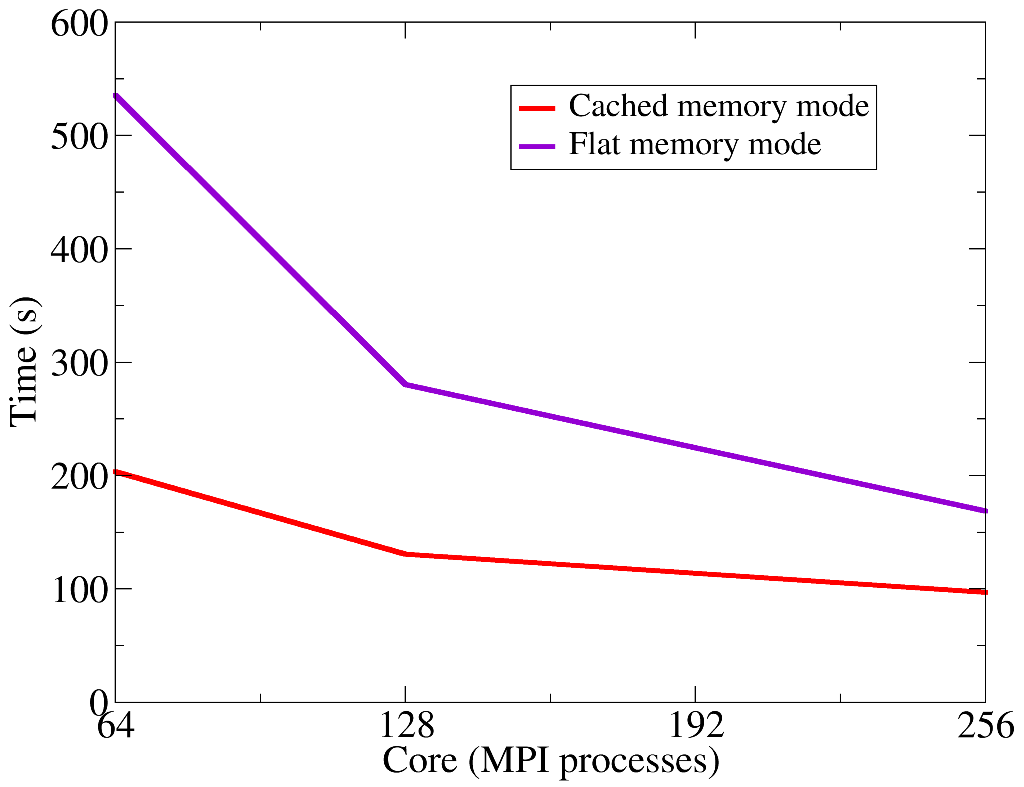

Figure 2: Performance of the TiN benchmark for cached memory mode and flat memory mode. The time is reported as the wall-clock time for 20 SCF cycles.

is 10%-20% higher than main memory).

To enable easy use of this memory the processor can be configured to use it as a very large last level cache, sitting between the processor and main memory and storing recently used data. This can provide good performance benefits for applications and does not require any source code changes to exploit this high bandwidth memory. However, this cache mode does not provide the full performance that can be achieved from the memory hardware (primarily because it requires data to be loaded or stored in two places at once rather than one). Therefore, the challenge for HBM is to be able to effectively place the data that will most benefit from its high bandwidth hardware.

3.1 Explicit memory allocation on the HBM

If it is known which data should be placed on the HBM at compile time, it should be possible to explicitly allocate the memory there and avoid the overhead caching and synchronising the data between the HBM and main memory. Using Intel Fortran compiler directives, it is possible to tell the compiler that allocatable variables should be allocated

Routine name	Flat	Explicit HBM	Cached
`Total time`	2305.15	1686.04	1066.22
`comms_transpose_exchange`	679.90	340.58	193.62
`algor_matmul_cmplx_cmplx`	792.60	634.48	359.34
`wave_zero_slice`	53.48	64.86	38.64
`wave_deallocate`	8.38	61.84	9.71

Table 5: Performance of a variety of operations in CASTEP using only main memory (flat), explicit allocation of the wavefunction coefficients on the HBM memory (explicit HBM), and automatic caching of data on the HBM memory (cached) in seconds for the `Al3x3` benchmark on one node of KNL.

on the HBM instead of in main memory:

```
real(kind=dp), allocatable :: Array(:)
!$ attributes fastmem      :: Array
```

However, since this can only be done during the declaration of allocatable objects, derived types containing such an object will *all* be allocated on the HBM since it is not possible to specify where memory is allocated during allocation.

In CASTEP, many of the most memory bandwidth intensive operations are performed on the wavefunction coefficients. For this reason, the performance of a selection of operations in CASTEP were compared for a calculation of the Al3x3 benchmark on one node of KNL. This was performed using:

- Only main memory in "flat" mode,
- The wavefunction coefficients explicitly allocated on the HBM, and
- Automatic caching of data to the HBM using "cached" mode.

The results of this are displayed in table 5. While explicitly allocating objects on the HBM does provide a significant performance improvement over flat mode, it provides a number of significant disadvantages. Firstly, not all operations are faster when using the HBM versus main memory. In particular, allocation/initialisation and deallocation of the wavefunctions (`wave_zero_slice` and `wave_deallocate` respectively) are significantly slower when the wavefunction coefficients are allocated on the HBM, compared to the same operations performed in either flat or cached mode. Secondly, the size of the HBM is significantly smaller than that usually available in main memory. Since the memory bandwidth intensive operations in CASTEP are performed on the largest objects allocated by CASTEP, this places a significantly smaller limit on the size of calculation that can be run than would be possible in either flat or cached mode.

Location of arrays	Memory bandwidth
Main memory	26.08 GB/s
HBM via the `fastmem` attribute	142.23 GB/s
HBM via `move_alloc`	36.40 GB/s

Table 6: Observed bandwidth for a contiguous memory copy between two 1.5GB arrays, both allocated on (a) main memory, (b) the HBM using the `fastmem` attribute and (c) the HBM using `move_alloc`.

3.2 Manually caching objects to the HBM

One solution to the limitations of explicit allocation of objects onto the HBM is to allow CASTEP to manually pre-fetch objects when they are needed. In order to avoid pervasive changes to the code base, the use of the `move_alloc` Fortran intrinsic subroutine was investigated, as this would allow pre-fetching and copying back objects to the HBM to be done as subroutine calls rather than re-implementation of the relevant routines with explicit use of the HBM.

For a given array `A` which has not been given the `fastmem` attribute, it is possible remap the array to the HBM as shown below:

```
real(kind=dp), allocatable :: A(:)
real(kind=dp), allocatable :: HBM_Buffer(:)
!$attributes fastmem HBM_Buffer

allocate(HBM_Buffer, source=A)
call move_alloc(HBM_Buffer, A)
```

Unfortunately, performance of this remapping suggests that this operation is not carried out as expected (see Table 6). Whilst the effective memory bandwidth is increased by a significant amount ($\sim 1.4\times$) this falls well below the bandwidth obtained by direct allocation of all arrays on the HBM ($\sim 5.5\times$). (NB even the HBM bandwidth falls well below the theoretical bandwidth, but this is to be expected as this experiment was performed on a single execution thread.)

The reason for the significant loss in performance using `move_alloc` has not been established. It is possible that `move_alloc` only moves the allocation of the data to the HBM, leaving the array metadata on main memory. This could result in a loss of performance, as witnessed here.

In summary, the effects of moving data to the HBM were investigated. Explicit allocation of performance critical objects on the HBM gave a significant performance improvement over main memory, however this placed limits on the size of calculation that could be run. Investigations into manual caching of objects onto the HBM were not able to realise these performance improvements without significant pervasive changes to the code base. In the end, the automatic "cached" mode was able to outperform any reasonable attempt to implement this manually.

4 Multithreading

Each of the 64 cores of the KNL in ARCHER has hardware support for up to 4 hyper-threads (HT), so in principle up to 256 execution threads are available for computation. The hardware support is for the most part not in the form of multiple functional units per core, but rather in providing support for multiple registers and fast context-switching, such that if one thread stalls (e.g. waiting for a memory or I/O request to complete) another thread can be switched in and executed with relatively little overhead.

Historically, enabling even 2-way hyperthreading on conventional Intel CPUs has been detrimental to performance. CASTEP's `al3x3` benchmark was run on a single KNL node using 64 MPI processes with 1-, 2- and 4-way hyperthreading enabled (1-way being equivalent to standard MPI-only execution). The additional threads were utilised with OpenMP. The results are shown in table 7.

The overall performance of CASTEP is worse when hyperthreading is enabled, with 4-way being worse than 2-way hyperthreading. However, looking at the per-subroutine timings it becomes clear that this slow-down is not universal, and some operations actually speed up significantly. The subroutine `pot_nongamma_apply_slice` performs the best in this context, showing a speed-up of $\sim 1.3\times$ going from no hyperthreading to 2-way hyperthreading, and a further speed-up of $\sim 1.3\times$ going to 4-way hyperthreading for a total speed-up of $\sim 1.7\times$. The workload in this routine is largely a level-2 BLAS call for matrix-vector multiplication, which is not very cache-efficient. If the cores are idling waiting for data from main memory, then this may explain the performance increase when hyperthreading is enabled, since it allows for fast context-switching and execution of another hyperthread can continue when one stalls.

At the other end of the spectrum, the matrix-multiplication routines perform substantially worse as the number of hyperthreads is increased. This operation is extremely cache-efficient, and often limited by the memory bandwidth as the data is streamed off the HBM. In this case, the memory bandwidth may be saturated, and it is not possible to keep all of the hyperthreads supplied with sufficient data. This is particularly problematic since all threads will be executing this code approximately simultaneously, i.e. each of the 2- or 4-hyperthreads for each of the 64 MPI processes will require data from main memory at the same time. In a similar vein, the data transposition subroutine is also limited by the memory bandwidth.

Routine name	1 threads	2 threads	4 threads
`Total time`	1099.69	1477.34	1788.48
`comms_transpose_exchange`	192.10	255.83	348.37
`algor_matmul_complex_complex`	356.09	627.98	715.16
`ion_beta_beta_recip_complex`	51.20	34.03	42.25
`pot_nongamma_apply_slice`	37.91	28.24	22.05

Table 7: Performance of a variety of operations in CASTEP running the `al3x3` benchmark. This was run with 64 MPI processes on a single node of KNL using 1, 2 and 4 hyperthreads per process.

The hyperthreading performance was also tested for a parallel run across multiple KNL nodes, both with the original CASTEP code and the code optimised as described in previous sections. For this test the `TiN` benchmark was used, and the results are shown in figure 3. The overall performance of the optimised code is about 10% better than the original, but no improvement to the relative hyperthreading performance is observed.

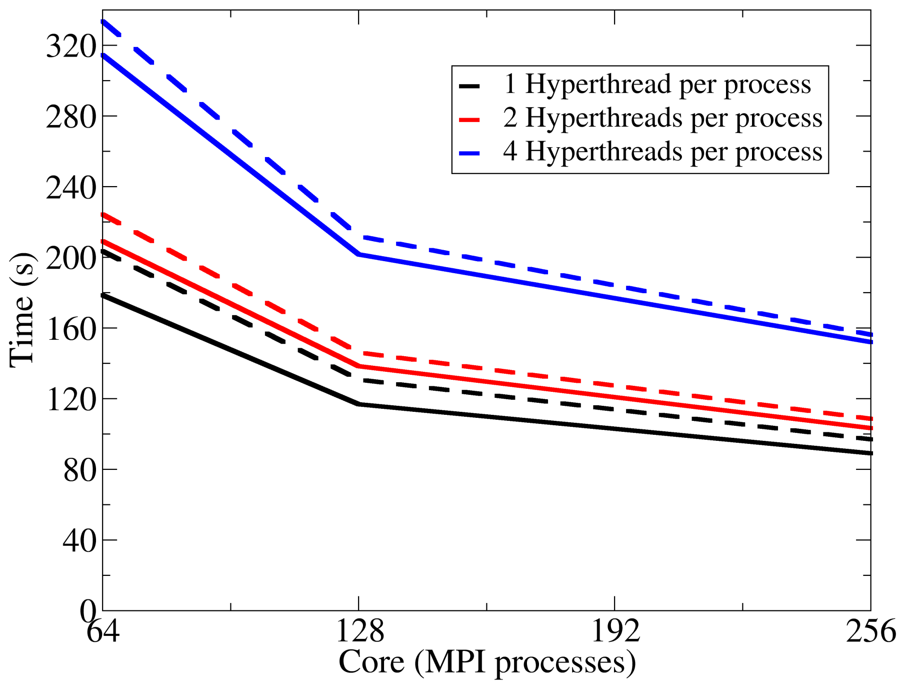

Figure 3: Performance of the TiN benchmark for 1, 2 and 4 hyperthreads per MPI process (1 MPI process per core). The performance is shown for the original (dashed lines) and optimised (solid lines) code. The time is reported as the wall-clock time for 20 SCF cycles.

Conclusion

In this work, the ability of CASTEP to use KNL efficiently was studied with respect to its per-subroutine performance on benchmarks, its vectorisation, use of hyperthreading and the KNL's memory mode. The performance of several key CASTEP subroutines was more than doubled, and the vectorisation and performance of several other subroutines was improved by around 20-40%. When CASTEP uses hyperthreading its overall performance degrades, but the profiling analysis showed that some subroutines did benefit from modest (2-way) hyperthreading.

When the KNL operated in "flat" memory mode, CASTEP initially performed poorly. By moving wavefunction slices into the fast HBM, the speed was improved substantially; however it was still quicker when run in cache mode.

The cumulative effect of this work is to speed CASTEP up enormously on the KNL platform in both memory modes. Even for the faster “cached” memory mode, CASTEP was sped up on the KNL by around 1.5 times. This is illustrated by the time taken for

CASTEP on a single KNL (64 MPI processes, 1 thread/process) to run 20 SCF cycles of the TiN benchmark: at the start of the project the time taken was 279 seconds, whereas the optimised code took only 179 seconds.

Phil Hasnip acknowledges support from an EPSRC Research Software Engineering Fellowship (grant no. EP/R025770/1). This work was funded under the embedded CSE programme of the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).