# Optimise HYPO4D on HECToR X2 machine

by **Ning Li**
HECToR CSE Team

## Background Information:

HYPO4D is a computational code developed at UCL using Lattice Boltzmann method to solve the governing equations of fluid flows for turbulence studies.

The code is written in C language and has been used extensively on the XT4 part of HECToR and is reported to scale extremely well (linearly) to up to 4096 cores. The developer has taken part in our X2 early-user programme aiming to port this code onto X2 vector machine to take advantage of the large amount of shared memory available.

The developer approached the CSE team after what he described as “completely unsuccessful” initial attempt to port the code with “segmentation fault right at the beginning with no clear origin”. The bug was traced to an undefined reference caused by the developer accidentally deleting some source code when porting the code. Efforts were then dedicated to improve the performance of the code on X2.

## Utilities Used:

The following utilities were used when assessing the performance of this code:

- Cray compiler's reporting function:
  Cray C compiler '**cc**' has an excellent 'loopmark listing' function that at compile-time generates annotated source code together with information regarding code optimisations. To turn on this, pass '**-h list=a**' flag to the compiler. An equivalent feature in Cray Fortran compiler is '**-R a**'.
- Cray 'explain' utility:
  Under X2 programming environment, type 'explain CC-????' for more details related to certain error/warning/information messages. Here ???? is a message ID given by the compiler.
- CrayPat:
  The Cray Performance Analysis Tools is first used to locate the most time-consuming routines. Only 'sampling' experiment is needed for this purpose. More detailed 'tracing' experiment is needed at later stage to study performance-specific issues such as cache usage and vector register usage of the code.

## Base Case:

The original source code compiled using standard '-O3' level optimisation is considered as the base case for comparison. A 256*128*128 mesh is used for all tests running on 8 cores – this is a suitable problem size as it contains sufficient workload and completes in a couple of minutes on TDS, the internal testing system. The timing results reported here are from the code's internal timing routine. These results are seen to show very small variations.

The base case completed in about **448 seconds**. According to the developer, this is almost twice the XT4 runtime for the same configuration. Clearly there are plenty of rooms for improvement. A second performance measurement widely used in Lattice Boltzmann codes is called site-updates-per-second

(SUPS). The SUPS value for the base case is **1.87E+06**.

A set of Cray recommended options '-O3 -h fp3 -h cache2' gave no obvious performance gains.

**Pointers Inhibiting Vectorisation:**

First of all the two most time-consuming routines (according to user's experience on XT4) were examined and a major problem was identified. The code consistently uses C pointers to operate big data arrays. This may be a very elegant programming approach (or not elegant at all depending on your perspective), however it is a major problem for vector compilers if not using properly. The compiler repeatedly reported messages like the following (try '`explain CC-6290`' for more information):

```
CC-6290 CC: VECTOR File = hypo4d_advection.c, Line = 31
  A loop was not vectorized because a recurrence was found between "lattice"
  and "lattice2" at line 35.
```

Here 'lattice' and 'lattice2' are two multiple-dimensional arrays of certain data type occurring in a loop. When they are referenced using pointers, the compiler has no way to know if their memory spaces overlap therefore the loop containing them doesn't get vectorised. A simple code snippet demonstrate the above problem looks like:

```
void pntr(int *a, int *b) {
   int i;
   for (i=0;i<64;i++)
      b[i] = i * a[i];
}
```

If one calls the above function using something like `pntr(&caller[0], &caller[10])` then the result would be incorrect if the loop operations are performed in parallel. The Cray compiler can spot this danger and simply generate scalar code to avoid potential problems.

Luckily in practice, it rarely programmers' intention to use overlapped memory space and the two pointers are often accessing independent arrays, as it is the case in this code. One way to inform the compiler that there are no data dependencies is to use 'restrict' keywords as defined by C99 standard, i.e. the above function definition becomes `void pntr(int * `**`restrict`**` a, int * `**`restrict`**` b)`. To avoid excessive source-code changes, one can pass '**-h restrict=a**' flag to the Cray C compiler to instruct that all pointers used in this manner can be safely vectorised.

The runtime was reduced to about **390 seconds** after using the above compiler flag, giving a SUPS value of **2.15E+06,** a speed-up of **115%**.

At this stage, a CrayPat report was generated to identify the most expensive routines for the next stage of optimisation. The results are:

```
 Samp % |  Samp |    Imb. |   Imb. |Experiment=1
        |       |    Samp | Samp % |Group
        |       |         |        | Function
        |       |         |        |  PE='HIDE'

 100.0% | 36558 |      -- |     -- |Total
|------------------------------------------------
|  93.8% | 34297 |      -- |     -- |USER
```

```
||-----------------------------------------------
||  47.0% | 17181 |  669.00 |   4.3% |lbe_step
||  35.5% | 12993 | 2829.88 |  20.4% |uniform_ABC_force
||   6.5% |  2379 |   97.62 |   4.5% |advection
||   2.5% |   897 | 1180.25 |  64.9% |ForceABC
||   1.4% |   498 |  548.75 |  59.9% |LBForceTerm
||===============================================
|   5.9% |  2151 |      -- |     -- |ETC
||-----------------------------------------------
||   5.4% |  1973 |  888.25 |  35.5% |__cis
|================================================
```

Clearly, function `lbe_step` and `uniform_ABC_force` should get more attention next.

**I/O Statements Inhibiting Vectorisation:**

The `lbe_step` routine is very time-consuming because it goes through 4-dimensional loops doing floating-point calculations. It contains a non-desirable feature – it checks some sort of stability condition of the numerical algorithm within the loops. According to the developer, once the code is set up properly the unstable conditions are seldom met. By removing these frequent checks, some calculations also become unnecessary (therefore removed from the machine code generated by the compiler), making the code significantly faster (**252 seconds**, SUPS=**3.32E+06**, a **178%** speed-up from the base case).

The developer is advised to change the way stability checking is performed (possibly less frequent check in the scope of an outer loop). If the checks are seldom needed in production runs, the developer is then advised to use preprocessor directives so that the checking part of the code can be conditionally filtered out at compile time to aid the run-time performance.

A related issue is that using any I/O statements in the inner-most loop will inhibit the vectorisation of that loop on X2. When this happens, a warning message like the following is issued by the compiler.

```
CC-6287 CC: VECTOR File = hypo4d_fequilibrium.c, Line = 39
  A loop was not vectorized because it contains a call to function "printf"
on line 114.
```

These I/O statements, even not located within the inner-most loop to prevent vectorisation directly, may have other negative impact on code performance. For example:

```
CC-6205 CC: VECTOR File = hypo4d_advection.c, Line = 518
  A loop was vectorized with a single vector iteration.

CC-6004 CC: SCALAR File = hypo4d_advection.c, Line = 529
  A loop was fused with the loop starting at line 518.
```

Here the first loop at line 518 is properly vectorised by the compiler. Because the second loop at line 529 has the same loop count, it can be fused (meaning the bodies of the two loops merged into a single bigger loops). Larger loop body may give the compiler better opportunities to perform optimisation. By adding unnecessary I/O statements in between the two loops, these additional opportunities get lost.

A second CrayPat report shows that the `lbe_step` routine is no longer the most expensive function after the above optimisation.

```
 Samp % |  Samp |    Imb. |   Imb. |Experiment=1
        |       |    Samp | Samp % |Group
        |       |         |        | Function
        |       |         |        |  PE='HIDE'

 100.0% | 23879 |      -- |     -- |Total
|-----------------------------------------------
|  83.1% | 19838 |      -- |     -- |USER
||----------------------------------------------
||  42.8% | 10212 | 6849.75 |  45.9% |uniform_ABC_force
||  19.3% |  4602 |  621.62 |  13.6% |lbe_step
||  10.2% |  2432 |  342.12 |  14.1% |advection
||   4.7% |  1121 |  715.88 |  44.5% |ForceABC
||   4.6% |  1101 |  779.62 |  47.4% |LBForceTerm
||   1.3% |   303 |   49.38 |  16.0% |halo_exchange
||==============================================
|  16.6% |  3971 |      -- |     -- |ETC
||----------------------------------------------
||  16.0% |  3825 | 4636.00 |  62.6% |__cis
|===============================================
```

Similar unnecessary checks are performed in  other part of the code, particularly in expensive functions such as `uniform_ABC_force` function. The developer was advised to properly change these too.

## Inlining and Vectorisation:

There are calls to smaller functions such as `ForceABC` within the nested loops in function `uniform_ABC_force`. These functions have to be 'inlined' in order for the nesting loop to be vectorised. Cray compilers' default setting is not particularly convenient when inlining functions. The smaller functions have to be defined in the same source file as the calling function for them to be picked up by the compiler. Or users must use '**-h ipafrom=**' flag for C code (or '**-O ipafrom=**' for Fortran) to explicitly specify the files containing the functions to be inlined. A third options is to use compiler directives to fine-tune the inlining behaviour, which isn't quite straight-forward.

The optimisation earlier using '**-h restrict=a**' flag actually has a negative impact in term of inlining. The compiler reported the following:

```
CC-3148 CC: INLINE File = hypo4d_advection.c, Line = 496
  Routine ForceABC was not inlined because a formal parameter is a restricted
  pointer and its corresponding actual argument is not a restricted pointer.
```

In order to fix this problem, a new build script was created which doesn't apply '**-h restrict=a**' flag to this one particular source file. As a result, the compiler reported:

```
CC-3001 CC: INLINE File = hypo4d_advection.c, Line = 496
  The call to ForceABC was textually inlined.
```

This change to source file `hypo4d_advection.c` actually slowed down the code to **305 seconds** initially. The **restrict** keyword had to be added explicitly to some pointer definitions to instruct the compiler to safely vectorise certain loops, reducing the time to **231 seconds**. Commenting out a few more I/O statements further brought down the run-time to **163 seconds**, or SUPS= **5.14E+06**, a **275%** speed-up in comparison to the base case.

## Unrolling the inner-most loop:

The most time-consuming part of the code often involves 4-dimensional loops. The 3 levels of outer loops are over spatial indices which are normally quite large (128 and 256 in the test case); and the inner-most loop is over a relatively small number of 'lattice vectors' and this number is fixed (=19) at compile time for this particular application. By default the inner-most loop gets vectorised unless the compiler is smart enough to swap the loops. Because in X2 hardware the vector registers can hold up to 128 words at a time, it makes sense to completely unroll the inner-most loop so that the larger nesting loop gets the chance to be vectorised. This proved to be very helpful in this case. For example, the source code `hypo4d_fequilibrium.c` contains several such loops. By default the compiler vectorise the inner-most loop, generating information like:

```
CC-6315 CC: VECTOR File = hypo4d_fequilibrium.c, Line = 35
   A loop was not vectorized because the target array (f_eq) would require
rank expansion.

CC-6315 CC: VECTOR File = hypo4d_fequilibrium.c, Line = 37
   A loop was not vectorized because the target array (f_eq) would require
rank expansion.

CC-6315 CC: VECTOR File = hypo4d_fequilibrium.c, Line = 39
   A loop was not vectorized because the target array (f_eq) would require
rank expansion.

CC-6205 CC: VECTOR File = hypo4d_fequilibrium.c, Line = 42
  A loop was vectorized with a single vector iteration.
```

Here line 42 is the inner-most loop getting vectorised into a single vector iteration but did not take advantage of the full vector length of the hardware. By adding a directive '**#pragma _CRI unroll 19**' in the source code before the inner-most loop, the compiler completely unrolled it (also called **unwound** by the compiler), i.e. making 19 copies of the loop body and deleting the loop itself. The 'loopmark listing' then gave:

```
CC-6294 CC: VECTOR File = hypo4d_fequilibrium.c, Line = 35
  A loop was not vectorized because a better candidate was found at line 39.

CC-6294 CC: VECTOR File = hypo4d_fequilibrium.c, Line = 37
  A loop was not vectorized because a better candidate was found at line 39.

CC-6204 CC: VECTOR File = hypo4d_fequilibrium.c, Line = 39
  A loop was vectorized.

CC-6008 CC: SCALAR File = hypo4d_fequilibrium.c, Line = 44
  A loop was unwound.
```

It can be seen that the larger loop at line 39 was vectorised instead, generating much more efficient code because more capacity of the hardware can be used. This improved the code performance significantly. With similar changes made in several other key functions (`uniform_ABC_force`, `advection`, `LBForceTerm`) through out the code, the run-time was reduced to only **99 seconds** (SUPS=**8.49E+06**). Now the code is **454%** faster than the base case.

Using CrayPat to examine the code performance again (tracing experiment this time), one can see two major areas of improvement. Because of the vectorisation of bigger outer loop, the average vector length in use has increased from 8.91 to 64, indicating much effective use of the hardware. A bonus is that the data cache usage has also improved significantly, from 83% to 100%.

```
========================================================================
USER / lbe_step
------------------------------------------------------------------------
  PAPI_VEC_INS              127.314M/sec    7969178998 instr
......
  Utilization rate                              100.0%
  Instr per cycle                                 0.36 inst/cycle
  HW FP Ops / Cycles                              1.07 ops/cycle
  HW FP Ops / User time     857.580M/sec   41313895012 ops        3.3%peak
  HW FP Ops / WCT           857.580M/sec
  HW FP Ops / Inst                              297.8%
  Avg VL                                          8.91 ops
  Data cache refs             0.001M/sec         54998 refs
  D cache hit ratio                              83.3%
  MIPS                      2303.97M/sec
  MFLOPS                    6860.64M/sec
  Instructions per LD ST                     252268.21 inst/ref
  LD & ST per D1 miss                             6.00 refs/miss

========================================================================
USER / lbe_step
------------------------------------------------------------------------
  PAPI_VEC_INS              127.314M/sec     819199998 instr
......
  Utilization rate                              100.0%
  Instr per cycle                                 0.20 inst/cycle
  HW FP Ops / Cycles                              9.05 ops/cycle
  HW FP Ops / User time    7242.970M/sec   46604654208 ops       28.3%peak
  HW FP Ops / WCT          7242.970M/sec
  HW FP Ops / Inst                             4425.6%
  Avg VL                                         64.00 ops
  Data cache refs            12.548M/sec      80738192 refs
  D cache hit ratio                             100.0%
  MIPS                      1309.29M/sec
  MFLOPS                   57943.76M/sec
  Instructions per LD ST                         13.04 inst/ref
  LD & ST per D1 miss                          5877.37 refs/miss
```

Before unrolling the inner-most loops, the compiler generates a mixture of vector and scalar operations, therefore the average vector length recorded by CrayPat (8.91) is much smaller than the loop count (19); after unrolling the inner-most loops, every operations are in vector mode, so the average vector length is identical to the loop count (64). The total number of vector instructions are also seen to be reduced by nearly 90% - from roughly 8 billion to 800 million.

Because the vector registers in the hardware can hold up to 128 words, it is natural to extend the above idea further by changing the order of the spatial loops so that the inner-most dimension corresponds to the largest mesh count. Recall that the problem size of the test case is 256*128*128 (or 128*64*64 for each core) and the third dimension doesn't have the largest loop count. In a further test of a different case over a mesh of 128*128*256 was performed (although this represents a different problem physically, the computational cost is the same), the total computational time has been reduced to **93 seconds** (SUPS=**9.03E+06**, **483%** speedup). The developer has since confirmed that the domain decomposition routine by default partitions the computational domain assuming nx >= ny >= nz. This is an arbitrary choice and nz should indeed be given the largest possible number. A slight change of the domain decomposition routine should benefit all future applications of this code on vector machines.

**Summary:**

The performance of HYPO4D code on HECToR X2 vector machine was significantly improved by

- instructing the compiler of data dependency issues.
- removing unnecessary stability checks and I/O statements within loops.
- explicitly instructing the compiler to inline key functions.
- manually unrolling loops to facilitate vectorisation of larger outer loop.

A 483% performance gain was achieved. Further study of the compiler 'loopmark listing' messages revealed additional opportunities for code improvement. One example is a more complex data dependency issue which can possibly be solved by restructuring of the source-code.