LSST:UK Consortium

LSST:UK Science Centre

Technical Report on ImSim

Project Acronym	LUSC-A
Project Title	UK Involvement in the Large Synoptic Survey Telescope
Document Number	LUSC-A-06

Submission date	8/NOV/19
Version	1.0
Status	Published
Author(s) inc. institutional affiliation	James Perry, University of Edinburgh
Reviewer(s)	Bob Mann (Edinburgh), George Beckett (Edinburgh)

Dissemination level	
Public	

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	15/NOV/17	Initial version	James Perry
0.2	12/SEP/19	Added details of OpenMP parallelisation work	James Perry
0.3	24/OCT/19	Minor changes to address comments from Bob Mann	James Perry, Bob Mann
1.0	8/NOV/19	Document finalised and published	George Beckett

Table of Contents

Index of Tables

1 Executive Summary

This report describes my work on the ImSim image simulation software[1] for DESC, part of the agreed UK contribution to DESC operations. My first task was to understand the performance of ImSim on the KNL[3] nodes of the NERSC supercomputer Cori and to look for ways to improve it. Subsequently, I parallelised parts of GalSim[2] (an open source galaxy simulation code used internally by ImSim) using OpenMP to improve its performance on parallel systems.

I investigated all the different execution modes offered by the KNL hardware, concluding that the choice of mode makes no significant difference to ImSim's performance, however the significant start-up cost incurred by switching to a non-default mode makes the default (quad, flat) mode the best choice.

I profiled ImSim to find out which parts of the code are taking the most time, and also to see if the profile was significantly different between the Haswell and KNL architectures. I found the profile to be relatively flat with no significant hotspots, and also to be relatively similar between Haswell and KNL. The low percentage of time taken by the GalSim `drawImage` function was surprising to many at DESC.

I investigated why ImSim was running 6-8x slower on KNL than on Haswell, when many DESC members were expecting a ratio closer to 2-3x. I tested ImSim, its backend library GalSim in isolation, and very simple computationally intensive Python and C programs. The result was a 6-8x slowdown between KNL and Haswell for everything CPU intensive, though I/O bound codes (such as very short GalSim runs) showed much less of a difference (1-4x).

I tested running multiple ImSim processes on a single KNL node, in order to make better use of the resources available. I found that up to 48 processes can be successfully run simultaneously per node, and that this is a far more efficient use of resources, even when contention for memory bandwidth is taken into account.

I attempted to roughly predict the CPU time and memory required for an ImSim run given the size of the instance catalogue used as input. Such a prediction would be useful for grouping similar processes together when assigning them to nodes. I discovered that the run time scales roughly linearly with catalogue size and that the memory usage is more constant but always well below 2GB. This suggests that assigning catalogues of similar size to the same node, and running up to 48 processes per node, would be a good strategy.

I introduced OpenMP[4] directives to the `Silicon::accumulate` function in GalSim, allowing it to benefit from shared memory parallelism when available. This significantly improves the performance on both KNL and Haswell systems, up to around 64 threads on Haswell and around 128 on KNL.

2 Introduction

2.1 Purpose

This document describes my work testing ImSim on the Cori system at NERSC, and exploring how to make the best use of the resources available for future data challenges. It also describes the OpenMP parallelisation of the GalSim library used internally by ImSim. This work was part of the agreed UK contribution to DESC operations.

2.2 Glossary of Acronyms

DESC - Dark Energy Science Collaboration

LSST – Large Synoptic Survey Telescope

KNL – Knight's Landing (an Intel Xeon Phi architecture)

MCDRAM – Multi-Channel Dynamic Random Access Memory

NERSC – National Energy Research Scientific Computing Centre

NUMA – Non-Uniform Memory Access

3 ImSim Performance Testing

3.1 Background

For their Data Challenge work (the data challenges are large scale data processing exercises intended to simulate the processing of LSST data, in order to understand how best to prepare resources and software for the telescope coming online), DESC have been allocated time on the Cori system at NERSC. Most of this time is on the KNL partition of the machine, however Data Challenge 1 highlighted some problems with running ImSim on the KNL:

- ImSim is single threaded, and in order to get reasonable performance from the KNL processors, many threads are required.
- Even when only a single KNL core is used, NERSC users are still charged for all of the cores during the time of the run. Since there are 68 cores per KNL chip, this is not an efficient use of the time allocated.
- ImSim appears to run approximately 6-8x slower on KNL than it does on the Haswell partition of Cori. This was unexpected as many members of DESC were expecting a slowdown closer to only 2-3x.

My task was to measure and understand the performance of ImSim on Cori, investigate why it runs so much slower on KNL than on Haswell, and find out how to make more efficient use of the KNL hardware.

Most of my testing was carried out using an instance catalogue (an input file for ImSim, consisting mostly of a list of objects) from the Data Challenge 1 data set. Specifically, instcat_40336_R_2_3_S_2_2.txt was chosen because of its relatively small size (4MB) – it can be processed in about an hour by ImSim running on KNL, whereas running the larger instance catalogues would have been prohibitively slow. Unless otherwise noted the runs referred to in this document used this instance catalogue.

I used the central installation of ImSim which was set up by Heather Kelly and supported both KNL and Haswell architectures.

3.2 KNL Modes

The KNL hardware can be configured into several different execution modes. These relate to the configuration of the NUMA domains, and whether caching is enabled or not. Specifically, the NUMA modes are as follows:

- quad – the KNL chip is divided into four virtual quadrants, but appears to the OS as a single NUMA domain
- snc2 – the chip is analogous to a 2 socket Xeon
- snc4 – the chip is analogous to a 4 socket Xeon

The cache modes available are:

- cache – the fast MCDRAM is used as a last-level cache
- flat – the fast MCDRAM is used as normal addressable memory

The modes can be selected using parameters in Slurm job scripts.

In order to get the best from the different execution modes, code changes are required. Unfortunately there was insufficient effort available for this, so I was limited to testing the standard ImSim code on each different mode to see which mode was the best fit for it. The results were as follows:

KNL Mode	Run Time
quad, cache	1:03:15
quad, flat	1:02:16
snc2, cache	1:02:48
snc2, flat	1:02:32
snc4, cache	1:03:52
snc4, flat	1:03:01

Table 1 Performance of ImSim in different KNL modes

As seen from the table, the actual run time is almost the same regardless of the mode selected. However, there is a delay of approximately 30 minutes before the code runs when selecting any mode other than the default (quad, cache), because the node needs to be rebooted. Therefore, the default mode is the most suitable.

3.3 Profiling

In order to better understand the performance of the code, I profiled it using the standard Python cProfile tool. Both KNL and Haswell runs were profiled so that the profiles could be compared across different hardware platforms. The run took 4,245s on KNL and 559s (7.6x faster) on Haswell. The profile revealed the following:

- The profile is fairly "flat". There is no single hotspot that takes the vast majority of the time as there is in many codes. Instead there are numerous functions that take moderate amounts of time.
- The function that takes the most time on both processors is setupCCMab from LSST photUtils. This accounts for 737s (17%) on KNL and 107s (19%) on Haswell.
- The function that takes the second most time on KNL is _transformSingleSys (from afw), accounting for 279s (7%) of the run time.
- The function that takes the second most time on Haswell is drawImage (from galsim), accounting for 32s (6%) of the run time.
- The relative time taken by the various functions is similar between both processors, but not identical.
- Looking at a selection of functions, most appear to be 6-12x slower on KNL than on Haswell. However, I/O related functions (such as posix.stat) are generally only 3-4x slower. These functions are probably not CPU bound.

The lack of a single intensive hotspot makes optimisation of the code more difficult, and the similarity of the KNL and Haswell profiles gives little clue as to the reason for the 6-8x slowdown. Several people expressed surprise that GalSim’s drawImage function only accounted for 6% of the run time (70% has been observed on other machines). To investigate whether this was due to start-up overheads when using a relatively small input file, I profiled with a larger 15MB instance catalogue (instcat_40336_R_1_2_S_0_0.txt). In this case, drawImage accounted for 12% of the total run time, twice as much as before but still less than expected.

3.4 Investigating the 6-8x slowdown

After analysing the profile of ImSim on Cori, I suspected that the 6-8x slowdown observed between KNL and Haswell might simply be due to Python being slow on KNL, rather than anything specific to ImSim. In order to test this I wrote a very simple but compute-intensive Python script:

```
sum=0
i=0
while i < 1000000000:
    sum = sum + i
    i = i + 1
print("sum=",sum)
```

I then ran this script on both KNL and Haswell nodes on Cori, using the default Python 2.7 environment, the Python 3 LSST environment used for my other ImSim tests, and finally the environment given by running `module load python/3.6-anaconda-4.4`. The results are below:

Python Environment	KNL runtime	Haswell runtime	Performance difference
Default Python 2.7	602s	101s	6x
LSST stack	917s	155s	5.9x
Anaconda	844s	154s	5.5x

Table 2 Performance of Python test program on Haswell and KNL CPUs

Since all three versions of Python showed a significant slowdown between Haswell and KNL even for a trivial test program, I decided to also test a trivial compute intensive C program:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
  long sum = 0, i;
  for (i = 0; i < 100000000000; i++) {
    sum += i;
  }
  printf("sum=%d\n", sum);
  return 0;
}
```

This program, compiled with gcc using the –O3 switch, showed an even more marked difference between Haswell and KNL: running on KNL took 7.2x longer. This suggested to me that 6-8x is simply the baseline performance difference between Haswell and KNL for single core, CPU intensive codes that have not been optimised for KNL.

It was also suggested that I investigate whether a similar slowdown occurs when running GalSim directly. (GalSim is used as a backend library by ImSim). Running the samples that came with GalSim resulted in only a 1x-4x performance difference between KNL and Haswell. However, these are very small examples and therefore likely to be dominated by I/O and start-up overhead. Running a larger, more realistic example resulted in a 6.6x slowdown between Haswell and KNL, much more similar to that seen with ImSim.

3.5 Running multiple processes per node

It was clear from the outset that making use of more than one core per KNL node was going to be crucial in order to get reasonable speed and efficiency for the ImSim runs. Unfortunately ImSim was strictly a single core program with no provision for parallelism, and there was insufficient effort available to embark on parallelising the code.

However, I did investigate the possibility of running multiple independent ImSim processes at once on the same KNL node. Since the data challenges require the same processing steps to be run on large numbers of different inputs, this could be a feasible way of exploiting more of the KNL's compute cores, both to reduce overall runtimes and to make better use of the resources allocated.

I experimented with running multiple ImSim processes simultaneously on a single KNL node. All processes were running the same instance catalogue (the one used in most of my other tests) so that timings could be meaningfully compared with the single process runs. A very simple system was used for this: all processes were launched in the background from a single Slurm script and the Bash `wait` command was used to wait for them all to complete.

Initially I was only able to run up to 15 processes simultaneously, before encountering a threading related error message. It turned out that the OpenBLAS library used internally by ImSim was parallelised with Pthreads. The number of threads used by OpenBLAS can be controlled by the `OPENBLAS_NUM_THREADS` environment variable. After experimenting with this it became clear that there was no benefit at all to ImSim in using multiple threads, in fact the threads were actually interfering with running multiple processes at once, so I set the variable to 1 for the remainder of my tests.

I was then able to run up to 48 processes simultaneously. The run times are given below:

Number of Processes	Total Runtime
1	1:03:15
2	1:04:42
4	1:05:32
8	1:08:44
16	1:09:43
32	1:16:38
48	1:24:24

Table 3 Performance of multiple ImSim processes on single KNL node

As can be seen, running 2 or 4 ImSim processes simultaneously only increases the runtime very slightly over that of a single process. When the number of processes is increased further, the runtime increases, probably due to memory contention between the cores. However, even with 48 processes the runtime has still increased by less than half of the single process runtime, so the efficiency is still far better than when running a single process on the node.

3.6 Estimating CPU and memory requirements

If the proposed solution of allocating multiple ImSim processes to a single KNL node is to be adopted, it would be advantageous to be able to roughly estimate the CPU and memory requirements of a given instance catalogue in advance, for two reasons:

1. To ensure that the node has enough memory to run all of the processes assigned to it.
2. To group together runs that are likely to take roughly the same time to complete, which is likely to be desirable for workload management.

In order to determine whether it was possible to estimate the CPU and memory resources required from the instance catalogue size, I ran ImSim with several instance catalogues of various sizes and recorded the time taken and high water mark memory usage.

Instance Catalogue	Size (bytes)	Time taken	Maximum memory used (GB)
R_1_0_S_0_2	2133703	5:54	1.22
R_2_3_S_2_2	3924779	7:59	1.29
R_1_1_S_0_0	7325162	14:19	1.53
R_1_1_S_0_1	11840568	23:03	1.56
R_1_2_S_0_0	15146110	29:02	1.64
R_0_1_S_0_1	24371410	49:09	1.59

Table 4 CPU and memory requirements of different sized instance catalogues

The results show that the time taken scales roughly linearly with catalogue size (although this scaling is not so strong for the smallest instance catalogues, probably due to start-up and I/O overhead), suggesting that grouping together catalogues of similar sizes would be a sensible strategy.

The maximum memory usage also tends to increase with catalogue size, but much more slowly, and the highest usage observed with any of the catalogues was only 1.64GB. This suggests that running 48 processes per node should be safe, as each node has 96GB of memory.

As well as testing this experimentally, I also talked to some of ImSim's developers to 'sanity check' this result. Their response was that this is to be expected and is likely to remain true for the instance catalogues used in DC2.

4 OpenMP parallelisation of GalSim

Although the many cores provided by KNL processors can be exploited to some extent by running multiple ImSim processes, it is also highly desirable to be able to run multiple threads per process within the critical sections of the code. This provides for a finer grained, lower overhead parallelism, since all memory is shared between the various threads.

Within ImSim, the actual drawing is performed by GalSim, an open source galaxy simulation toolkit. GalSim is mostly written in C++ with a thin Python wrapper around it. For typical ImSim workloads, the critical function within GalSim is the `accumulate` method of the `Silicon` class. This method simulates a number of photons being fired at a silicon image sensor. Since the same operation is repeated multiple times over a potentially large number of photons, this appeared to be a good candidate for parallelisation.

4.1 Parallelising the code

Whilst the main operation of adding the photons to the image was relatively straightforward to parallelise, there were a few subtleties that had to be addressed in order for the parallelisation to work satisfactorily:

- A subroutine called `updatePixelDistortions` took up a significant portion of the run time, so it also had to be parallelised. This function involves changing the positions of the pixel boundaries very slightly based on the photons that had previously hit each pixel, mimicking the behaviour of real silicon image sensors.
- Some operations (for example, adding the updated flux to the existing image) had to be made atomic to avoid the danger of race conditions if multiple threads attempt to update the same value simultaneously. An alternative method of having a separate delta image for each thread and merging them at the end of the pixel loop was also tested, but the atomic version was found to be significantly faster.
- The `updatePixelDistortions` function is called whenever the total flux added to the image since the last call reaches a certain threshold. In the original serial version of the code, this was done within the main photon loop, however in the parallel version this was not a suitable place for it, since that would result in it being called from every thread instead of just once. Instead, the loop condition was changed so that the inner (parallelised) loop would exit when a call to `updatePixelDistortions` was required. To ensure that exactly the same number of photons would be processed before each call as in the serial version, the parallel version pre-calculates how many photons are required to take the added flux over the threshold and uses this number as the inner photon loop bound. Fortunately the flux value for each photon is readily available in the photon structure, so this is a fairly lightweight operation.
- `Silicon::accumulate` makes extensive use of random number generators, so care has to be taken to avoid a different random sequence (and hence a different resulting image) being produced when running in parallel. To maintain reproducibility, the parallel code generates all the required random numbers in an array before entering the parallel loop. A slight modification to the logic was required because one of the random number generation calls in the original code was conditional, therefore the total number of random numbers required to process each photon could not be predicted in advance. To make this predictable, the parallel code instead generates this number for every photon, whether it is required or not. As a result, although the parallel code will produce the same result regardless of the number of threads in use, it does not produce an identical result to the old serial code.

More details of the parallelisation process are given in the relevant GitHub issue (https://github.com/GalSim-developers/GalSim/issues/1008).

4.2 Performance tests

During development, the parallel version of GalSim was tested using one of the existing GalSim test scripts, namely `test_sensor.py`. This was useful for verifying that the code was still producing correct results and that performance was improved, but it uses a relatively small problem size, resulting in the performance improvement tailing off as the number of threads increases. A new script called `test_silicon_accumulate.py` was developed to allow more realistic problem sizes to be tested. This uses a 1000x1000 pixel image size and fires 10,000,000 photons at random locations throughout the image.

Number of Threads	Total Runtime
1	341.95
2	189.04
4	99.63
8	56.86
16	34.85
32	26.4
64	24.79

Table 5 Performance of parallelised GalSim code on Haswell

Number of Threads	Total Runtime
1	3017.62
2	1526.92
4	777.83
8	403.57
16	215.28
32	119.35
64	71.25
128	52.49
256	49.8

Table 6 Performance of parallelised GalSim code on KNL

Performance of this test problem with various thread counts is shown in Table 5 and Table 6 for Haswell and KNL respectively. It can be observed that in both cases the code scales well up to around 8 threads, and increasing the thread count continues to provide significant benefits up to 32 threads on Haswell and 128 on KNL. The performance eventually plateaus at around 13x faster than the serial code on Haswell, and around 60x faster on KNL. This is consistent with the number of cores available on these processors. The plateauing is likely due to contention for shared resources (such as memory) becoming problematic at higher thread counts.

4.3 Further Work

At the time of writing, the parallelised code has only been tested in isolation using pure GalSim examples. It should also be tested as part of larger ImSim runs. In particular, since parallelism can now be exploited both at the process level in ImSim and at the thread level in GalSim, experiments should be done to find out the optimal way to distribute work among the cores available.

5 References

[1] https://github.com/LSSTDESC/imSim

[2] https://www.lsst.org/scientists/simulations/galsim

[3] Sodani, Avinash, et al. "Knights landing: Second-generation intel xeon phi product." *Ieee micro* 36.2 (2016): 34-46.

[4] Dagum, Leonardo, and Ramesh Menon. "OpenMP: An industry-standard API for shared-memory programming." *Computing in Science & Engineering* 1 (1998): 46-55.