A programming framework for data streaming on the Xeon Phi

A programming framework for data streaming on the Xeon Phi

S Chapeland[1] for the ALICE collaboration

[1]CERN, Physics Department, Geneva, Switzerland

Sylvain.Chapeland@cern.ch

Abstract. ALICE (A Large Ion Collider Experiment) is the dedicated heavy-ion detector studying the physics of strongly interacting matter and the quark-gluon plasma at the CERN LHC (Large Hadron Collider). After the second long shut-down of the LHC, the ALICE detector will be upgraded to cope with an interaction rate of 50 kHz in Pb-Pb collisions, producing in the online computing system (O^2) a sustained throughput of 3.4 TB/s. This data will be processed on the fly so that the stream to permanent storage does not exceed 90 GB/s peak, the raw data being discarded. In the context of assessing different computing platforms for the O2 system, we have developed a framework for the Intel Xeon Phi processors (MIC). It provides the components to build a processing pipeline streaming the data from the PC memory to a pool of permanent threads running on the MIC, and back to the host after processing. It is based on explicit offloading mechanisms (data transfer, asynchronous tasks) and basic building blocks (FIFOs, memory pools, C++11 threads). The user only needs to implement the processing method to be run on the MIC. We present in this paper the architecture, implementation, and performance of this system.

1. Introduction

1.1. ALICE and the O^2 project

ALICE [1] is the heavy-ion detector designed to cope with very high particle multiplicities to study the physics of strongly interacting matter at the CERN LHC. It is optimized to study the properties of the deconfined state of quarks and gluons produced in such collisions known as quark-gluon plasma [2]. It is also well-suited to study elementary collisions such as proton-proton and proton-nucleus interactions.

The detector will be upgraded [3] in the next LHC long shutdown, planned 2019-2020, and will produce an increased data throughput reaching 3.4 TB/s. The Online-Offline system, named O^2, will be in charge of reading out these data and processing them on-the-fly, in order to reduce the volume down to 90 GB/s initially recorded to storage. These demanding processing and compression steps will be handled by a computing farm consisting of ~250 nodes for readout and ~1500 nodes for online reconstruction.

In order to build the O^2 system, different computing platforms are considered, including classical dual CPU sockets servers, GPUs accelerators, and MIC. Some evaluations have already been performed, as described in the O^2 Technical Design Report (TDR) [4], in order to select the most suitable and cost-effective hardware for the target tasks. This study will be further completed with latest hardware assessment before final purchase decisions are taken.

1.2. The Intel Xeon Phi platform
At the time of the TDR study, a new computing platform was just launched by Intel, targeting high-performance computing: the Intel Xeon Phi or MIC (Many Integrated Cores). It was released initially in 2013 with name Knights Corner (KNC) [5], followed by a newer version in 2016 named Knights Landing (KNL) [6]. It comes in two form factors: first, as a PCI-e "co-processor" card (KNC, possibly KNL later); second, as a standalone system with integrated fabric (KNL). Some of the specification highlights include: 57 to 72 physical x86 cores, 4-way multithreading, moderate core clocking (1.1-1.5GHz) but large vectors (512 bits operations), fast on-chip MCDRAM (8 to 16 GB for KNL), and limited power consumption (200-300W). Programming and execution can be done in different ways, in particular: compilation of the code for native execution on the MIC (it runs Linux, so can be accessed by ssh even on the PCI-e version); use of Intel language extensions for parallelism (OpemMP, Cilk); use of the Intel Performance Libraries making use of the hardware capabilities (Math Kernel Library, Thread Building Blocks); and finally, explicit data and processing offloading with language extensions (by mapping variables between CPU and MIC memory address spaces and calling offload commands for data copy and code execution on the MIC).

1.3. Rationale
The MIC hardware features sounded potentially interesting for our purpose, and we started to evaluate the KNC in 2014. In particular, we focused on the explicit offloading model, as it looked the one giving the most flexibility to control streams of data, as is needed in our busy data-acquisition nodes where data movements are critical. For evaluation purpose, we implemented an algorithm to be used as benchmark. We noticed during this first experimentation with the MIC several points to be addressed.

First, the offload can be quite intrusive in the processing code to be tested, as it needs special control blocks to handle data movements and execution on the MIC. This resulted in a prototype code mixing both control flow and algorithm. We would like to keep them separate, in order to reuse the same (hard-to-optimize) offload mechanisms for different algorithms, and if possible avoid any change to the algorithms code to be tried.

Also, in our application, the incoming streams of data can naturally be divided in chunks (their size varying from a few Kilo Bytes to few Mega Bytes) that can be processed independently. This provides a trivial way to parallelize processing. However there was no high-level mechanism matching well this data pattern for MIC offload.

Additionally, tuning was needed to make good use of the MIC resources: PCI-e transfers, memory allocations, and workload distribution on many cores had to be carefully managed globally to actually use all MIC cores in I/O busy workloads.

All in all, substantial development time was spent on getting the data on the MIC efficiently, which prevented us to fully focus on the algorithm and MIC-specific code optimization. The work presented in this paper is a by-product of the TDR benchmark study. It was revived in 2016 with the announcement of the KNL platform, for its evaluation. The key offload and workload distribution mechanisms of the first prototype have been extracted and optimized, in order to build a general purpose data streaming framework able to run code on MIC coprocessor boards, primarily for the benchmarking of different physics algorithm that will be needed in O^2.

2. The framework

2.1. Purpose of the framework
The purpose of the framework presented in this paper is to provide a simple and efficient way to distribute data and run processing code on MIC coprocessor boards. It aims at minimizing the transport and threading offload overhead, both in in terms of performance (lightweight, so that it scales well) and code complexity (easy integration of new algorithms). The intent is to allow measuring MIC performance with a realistic data pattern for our future system, i.e. a sustained stream of input data chunks which can be processed and reduced in parallel.

2.2. Run time processing pipeline

The run time processing pipeline is made essentially from two simple building blocks:

- First-In-First-Out (FIFO) buffers: queue optimized for 1-to-1 lockless communication (one side pushing data in, the other side reading data out). Each structure stores pointers.
- Threads: each thread is an object having a *start()* and *stop()* method called once when starting or stopping, and a *doLoop()* method called iteratively (in its own system thread) when active at run time.

These base components have been chosen based on previous positive experience in the development of different multi-threaded server processes in the current ALICE Data-Acquisition system [7].

They are arranged and connected together to create a pool of processing threads, called the thread pool engine, as shown in Figure 1. Each thread reads from an input FIFO, and writes to an output FIFO, with data flowing from one component to the next. The number of processing threads is configurable and should be adapted to the number of cores available on the target hardware (in benchmarks, it is usually variable to check how the system scales up with the number of threads from one test run to the other). There are also two special threads part of the data flow: one to dispatch incoming data to the pool of threads (depending on space availability in their respective FIFO), and one collecting (serializing) the output of each processing thread FIFO to a single output queue. The processing threads methods are overloaded with the relevant code to implement a given processing algorithm. The framework code (including the threads creation) is written using the native C++11 constructs.

The thread pool engine is used to build the MIC processing pipeline, but can also be used standalone to run tests on standard CPU servers.

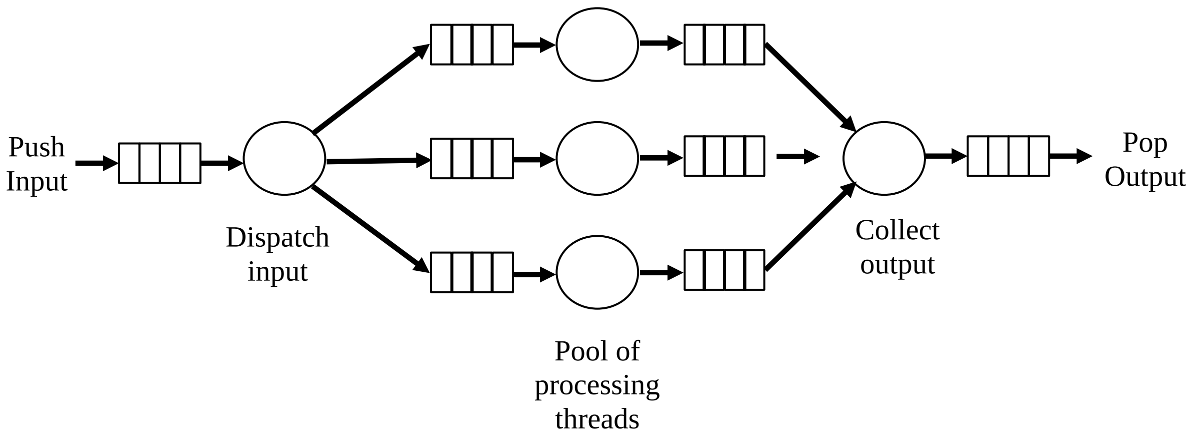

Figure 1- Sketch of the Thread Pool Engine

On systems with a MIC board, the complete data flow is shown in Figure 2. The framework implements a MIC processing engine based on the generic thread pool engine class and some extra I/O handlers. One instance of the thread pool engine is launched persistently on the MIC. It needs a few additional components to transfer data back and forth from the host CPU memory (where the data to be processed resides and where the results should eventually go). It is therefore completed with two threads running on the MIC, handling data input and output for the thread pool engine running on the MIC. Additionally, one thread running on the CPU host takes care of all offload commands to move data to and from the MIC. Each of these three threads are also associated with a queue. This splitting of tasks leaves some flexibility to the operating system to schedule work with minimal time spent in polling, as threads are made idle sleeping for a short time when their buffer is empty. To avoid repeated memory allocations, all data blocks going through the FIFOs belong to memory pools pre-allocated both on CPU

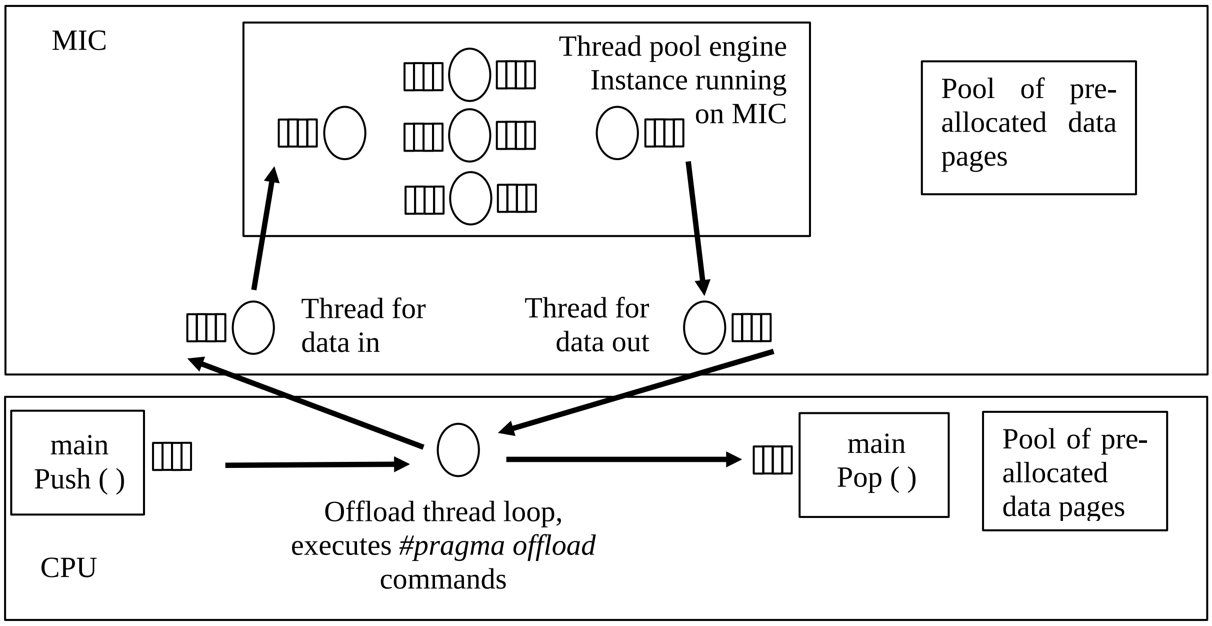
MIC
Thread pool engine Instance running on MIC
Pool of pre-allocated data pages
Thread for data in
Thread for data out
main Push ( )
main Pop ( )
Pool of pre-allocated data pages
Offload thread loop, executes *#pragma offload* commands
CPU

Figure 2- Sketch of the MIC processing engine internal data flow

and MIC. They are key components of the data moves optimization between host CPU and MIC coprocessors, each having a separate memory system.

2.3. Implementing code for run time processing

From user's perspective, adding a new processing algorithm to the system and stream it with data at run time is fairly simple with the framework. It is based on a C++ interface.

First, a user's class should be added for the specific processing to be done, derived from the framework's *tProcessData* base class. A single *processData()* method is needed, which takes as input a pointer to data (coming from the main input FIFO), and providing a pointer to the result (pushed by the framework to the main output FIFO). The framework also requires the declaration of a factory function, so that instances of this new class can be created at run time according to the size of the thread pool.

Then, the MIC processing engine can be instantiated in the main executable, requiring as argument only a *tProcessData* factory function to be used and the number of threads to be created. The framework takes care of creating the necessary threads and FIFOs, to which the user is connected by two single input and output queues. From that point, data can be moved in and out in a loop, by a *push()* or *pop()* call on the input or output FIFO of the MIC processing engine. Data will be transparently moved to the MIC and processed there with the custom *processData()* method.

2.4. Practical aspects of writing code for MIC

It is actually not much different than writing code for the CPU. Among things which should be taken into consideration, we have experienced the following. First, it is necessary to use the good compiler options to get the code (or part of it) built for MIC native execution (e.g. *–mmic* for the Intel compiler ICC). Adequate compiler hints can also be added to source code with:

- *__declspec__ target(mic)* before a single declaration
- or with *#pragma offload_attribute(push,target(mic)) … #pragma offload_attribute(pop)* for a block of code.

External libs can be used on the MIC, provided they have been recompiled with appropriate flags to generate MIC code (which can be a long iterative process if there are many hierarchical dependencies to be themselves recompiled). In practice, we have seen good portability of the components we tried to use, as it was usually done just by adding some compiler and linker option, provided that the package build rules can be edited easily. For example, for the BZIP library [8], it consisted of only 3 lines of code changed in a Makefile in order to be able to call BZIP calls from MIC routines:

```
CC=icc
AR=xiar –qoffload-build
CFLAGS=-qoffload-attribute-target=mic
```

When necessary, it is also possible to simply isolate code in the source code with *#ifdef __MIC__* … *#endif* in the case of portability issues, or custom handling for the MIC.

2.5. MIC offload optimization
While optimizing the framework for the KNC boards and PCI-e offload, we found the following aspects to be critical for the best usage of the Host-KNC communication. Although hidden in the framework implementation to the end users, these points could be useful for the programmer interested to use MIC explicit offload mechanisms.

Memory should be pre-allocated whenever possible, as it is time consuming to allocate memory on the MIC. If blocks have to be allocated at transfer time, we measured block transfer rate to top at 1400Hz, which means that transfer is 6x slower for blocks of size 256 kB compared to blocks of same size pre-allocated on MIC (and it gets even much worse as block size decreases). Reusing the same blocks is lighter than reallocating new ones continuously.

Memory should be aligned according to guidelines for the variables to be transferred, in order to get maximum Host-MIC copy performance. We measured that copying blocks up to 16 kB allocated with *malloc()* without alignment specifier is 30% slower than same blocks allocated on 4 kB boundaries with *_mm_malloc()*.

For indirect memory mapping between the CPU and MIC address space, the use of *#pragma offload into targetptr* proved to be a useful construct when the two memory spaces do not match (e.g. the large size of the pool allocated for data coming from the DAQ readout boards in CPU memory prevents direct mapping, as it is much bigger than the total memory available on the MIC, and one wants to pre-allocate them for performance). Also, for lower chunks size, transfers can be grouped together (i.e. in an array of pointers) to reduce the number of offload commands.

Asynchronous offload transfer and compute can be alternated to avoid idling during transactions (like in the double buffering example provided with the MIC documentation [9]).

As all data moves are initiated from CPU side (one cannot have a MIC thread pushing data out to CPU memory), there is an asymmetry in the communication, which usually is easier to be done by a single task to avoid contention.

Concerning the thread scheduling, it is possible to start persistent threads on the MIC by launching them from an asynchronous offload task with *#pragma offload signal()*. Special tuning needs to be done for polling loops, as the parameters optimal on the CPU side may not work best on the MIC (for example, *delay, sleep* or *yield* should be selected appropriately, with parameters to be tuned by practice).

3. Performance

3.1. Base components
We measured the performance of one thread sending data to another thread through a FIFO, as it is the main mechanism used to stream data within the framework. Two x86 systems were tested: a Skylake desktop CPU, and a dual-socket SandyBridge server. The first thread pushes in the FIFO pointers to data at a predefined rate, and the other reads them out from the FIFO. Both tasks are asynchronous and work independently. As shown in Figure 3, transfer is using less than 1% of a CPU core up to 50 kHz,

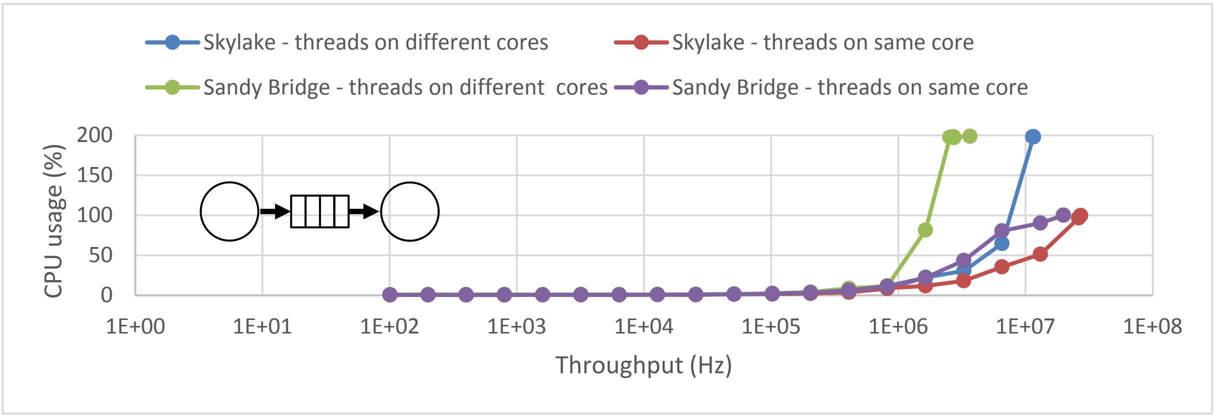

Figure 3- CPU usage as function of FIFO throughput

and less than 10% at 1MHz rate. It reaches a maximum rate of 10 to 30 MHz for a full core per thread. CPU resources used are therefore well controlled in the target rates of O2, usually less than 100 kHz.

For the data offload to MIC, the raw I/O speed was measured on KNC, as seen on Figure 4. It was found to correspond to the maximum speed offered by PCIe, around 6.2 GB/s for one-way transfers when data chunks are large enough, i.e. more than ~250 kB. When getting data in and out of the MIC, the bandwidth is shared, and we get 3 GB/s in and 3 GB/s out, with a limit of 100-200k transfers per second for smallest packet sizes. This test was done by moving data of fixed size and address back and forth between the host and MIC memories.

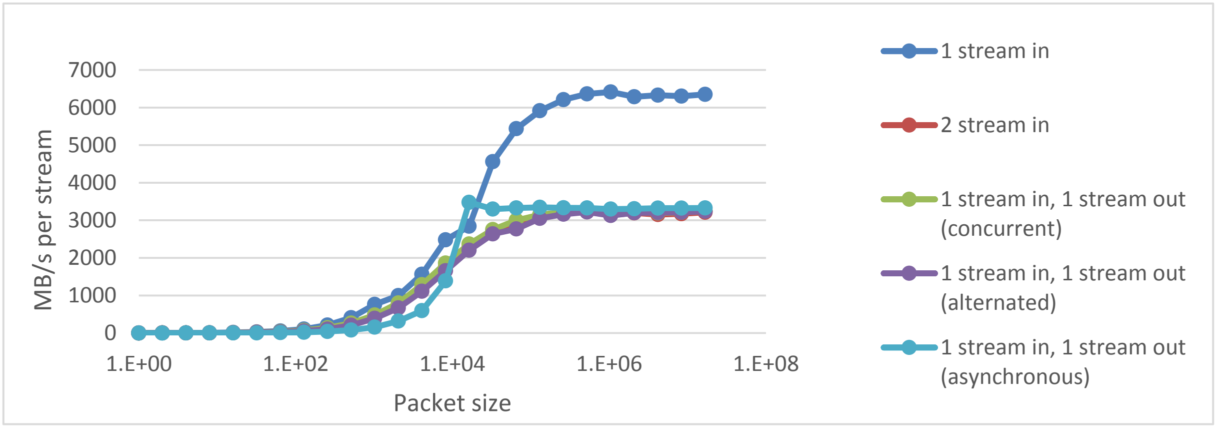

Figure 4 – Data transfer speed between CPU Host and KNC as a function of packet size

(MIC version 31S1P – PCIe 2.0)

Running the framework with asynchronous processing on the MIC, throughput is lower, around 1.2 GB/s each direction (or 2.4 GB/s total), as the data streaming complexity is higher. Indeed, one extra step is needed to first query available results from the MIC (addresses and sizes of data chunks) before moving them back to CPU. It can probably be optimized further, but is however good enough for our current needs, and is not the processing bottleneck: it is 50x faster than the typical algorithms we tried.

3.2. Use case: BZIP

To check the capabilities of the framework, we chose to implement a workflow compressing data with BZIP. It represents a task of similar nature than some ALICE online algorithm, like the ITS cluster finder [10] (i.e. lots of memory I/O and little floating-point computations). The code being readily available in a library, no algorithm needed to be developed. Using the method described in 2.3. , a single class was implemented, simply calling *BZ2_bzBuffToBuffCompress()*.The main program preloads in memory a file of simulated data samples, and pushes the chunks to be compressed to the processing engine FIFO interface, while reading out the output FIFO when available. We added a few counters and clocks to time the execution and convert results to throughput performance values. We were able, with minimal code development (the code not provided by framework, i.e. the algorithm part), to exercise a realistic workflow processing a continuous stream of data coming from memory, able to saturate the cores on different systems, both with standalone Xeon CPUs and Xeon Phi offload boards. System showed a good scale-up linearity as a function of the number of threads, as can be seen in Figure 5. It reaches the expected extrapolated performance of the one-core standalone measurement multiplied by number of physical cores (and even 20-30% more when oversubscribing Xeon CPUs, thanks to hyper-threading).

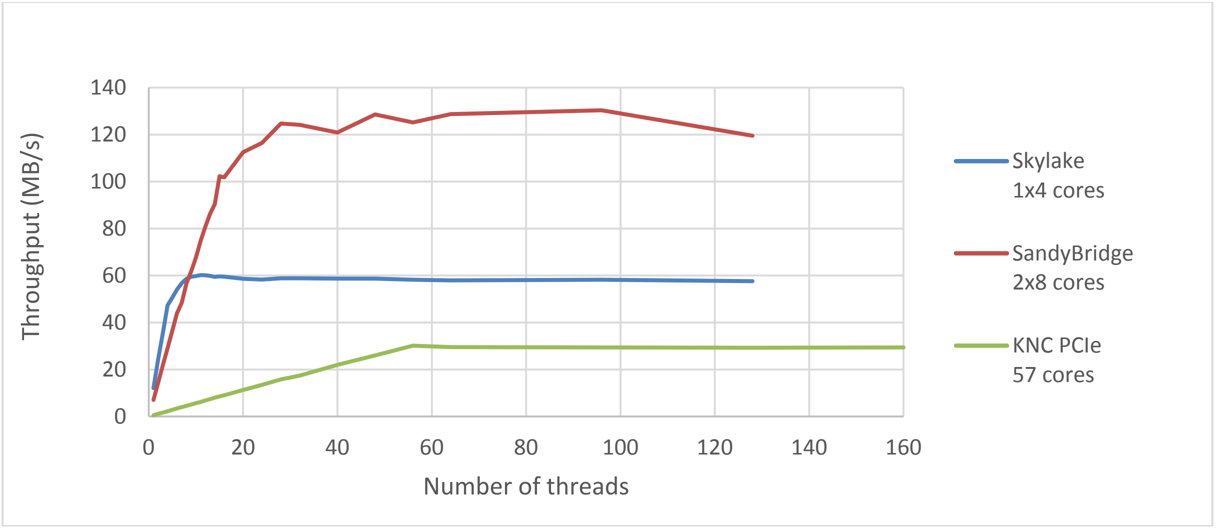

Figure 5- BZIP benchmark with framework processing engine

Unfortunately, the offload processing engine could not be tried on KNL, as it is not yet available as a PCI-e offload board. However, the thread pool engine was tried with success on a standalone KNL, showing the same good scale-up behaviour with the number of cores as on the other systems. When more than half of the cores are used, we notice a small decrease of per-core performance, which is expected given the underlying KNL hardware architecture and I/O intensive application, as the low-level memory cache is physically shared by two cores. Performance is shown in separate Figure 6, as this is an earlier measurement not done with exactly the same code and data as for Figure 5. Although not part of this study, we note that the example BZIP processing algorithm is (as could be expected) not the most suitable task to run on KNC and KNL platforms (which may outperform Xeon systems rather with algorithms having highly vectorised code and floating point operations, this not the case here).

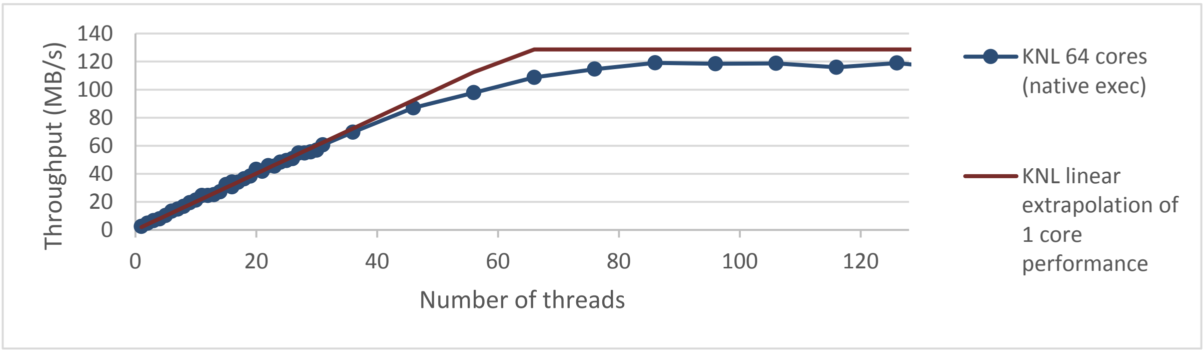

Figure 6 - KNL scaling up of an I/O intensive processing task

4. Conclusions

In the context of hardware benchmarks for the ALICE O2 project, we built a generic framework simplifying offload processing of a sustained stream of data to MIC co-processor board. The framework can also be used on standalone CPU systems to distribute a stream of data chunks to the available processing cores, all threads belonging to the same process. We measured the framework performance on KNC, and showed that the framework allows to transparently use MIC offload processing features. It also shows good thread scalability on other systems, providing a simple integration mechanism of new algorithms in the pipeline.

Due to uncertainties about the availability of a PCI-e version of the latest generations of Xeon Phi, further developments have been suspended for the time being. Should a PCI-e version of KNL be released (which is still unclear at the time of writing), the framework will readily help with benchmarks of the MIC by taking care of the offloading mechanism and workload distribution.

References

[1] The ALICE Collaboration et al. 2008 The ALICE experiment at the CERN LHC JINST **3** S08002
[2] Cabibbo N and Parisi G 1975 Phys. Lett. B **59** pp 67-69
[3] The ALICE Collaboration et al. 2014 Upgrade of the ALICE Experiment: Letter of Intent J. Phys. **G 41** 087001
[4] Buncic P, Krzewicki M and Vande Vyvre P 2015 Technical Design Report for the Upgrade of the Online-Offline Computing System Technical Design Report ALICE **19**
[5] http://ark.intel.com/products/codename/57721/Knights-Corner
[6] http://ark.intel.com/products/codename/48999/Knights-Landing
[7] Carena F et al. 2014 The ALICE data acquisition system Nuclear Instruments and Methods in Physics Research A **741** pp 130–162 http://dx.doi.org/10.1016/j.nima.2013.12.015
[8] http://bzip.org/
[9] http://software.intel.com/en-us/articles/asynchronous-offload-c-code-examples
[10] Chapeland S et al. 2014 Benchmarks Based on a Pixel Cluster Finder Algorithm for the Future ALICE Online Computing Upgrade Proceedings of the 19th IEEE-NPSS Real Time Conference p 333