A programming framework for data streaming on the Xeon Phi

Sylvain Chapeland (CERN)
for the ALICE Collaboration

22nd International Conference on Computing in High Energy and Nuclear Physics, CHEP 2016
Outline

• About ALICE O²
• About the Xeon Phi
• Purpose of the framework
• Features and implementation
• Use case
ALICE Online-Offline project ($O^2$)

- ALICE detector to be upgraded
  - LHC long shutdown 2019-2020
- Increased data throughput
  - Demanding processing and compression
  - Estimated farm size:
    - Readout: ~250 nodes
    - Online reconstruction: ~1500 nodes
  - Computing platforms to be selected: CPU, GPU, MIC, ...
  - Some hardware evaluations already done (c.f. TDR)
  - To be completed with latest hardware for final selection

- See other ALICE $O^2$ talks at CHEP’16
  #114, #119, #120, #182, #230, #314, #378

A programming framework for data streaming on the Xeon Phi - CHEP’16
What’s a Xeon Phi?

- Device from Intel
  - Intel Many Integrated Cores (MIC)
  - KNC (2013), KNL (2016)

- Form factor
  - (big) PCI-E “co-processor” (KNC, KNL soon?)
  - Standalone systems, integrated fabric (KNL)

- Specifications highlights
  - 57-72 physical cores
  - x86 cores, based on Pentium (KNC) / Atom (KNL)
  - 4-way multithreading
  - Moderate core clocking (1.1-1.5 GHz)
  - Large vectors (512b)
  - Fast on-chip MCDRAM (KNL, 8-16GB)
  - Power 200-300 W

- Potentially interesting for some processing tasks

```bash
> cat /proc/cpuinfo | grep processor | wc
256
```
How to use a Xeon Phi?

• Compilation of the code for MIC native execution
  • scp + ssh (MIC runs Linux)

• Language extensions for parallelism
  • OpenMP, Cilk

• Intel Performance libraries
  • Math Kernel Library, Thread Building Blocks

• Explicit data and processing offloading
  • Mapping of variables between CPU and MIC
  • #pragma offload commands for I/O and execution
Rationale

• While evaluating KNC in 2014, we noticed that:
  • Offload usually intrusive in the processing code to be tested
  • Lack of high-level offload mechanism matching well our data pattern
  • Tuning needed to make good use of the MIC resources
    • PCIe transfers
    • Workload distribution on many cores
  • Substantial development time spent on getting the data on the MIC efficiently
  • Prevents to fully focus on algorithm / MIC-specific code optimization

• Work presented here is a by-product of this past benchmark study
• Paused for a while, revived with the arrival of the KNL(for its evaluation)
Purpose of the framework

• Provide a (hopefully simple and efficient) way to...
  • distribute data and run processing code on the MIC
  • minimize transport/threading offload overheads
    • performance and code complexity
  • measure MIC performance with realistic data pattern
    • sustained stream of input data chunks which can be processed in parallel

• Typical usage
  • Implement some data processing algorithms
  • Benchmark to see how they fit on MIC
  • Run same code on CPU and MIC for comparison
Building blocks for processing pipeline

- **FIFO buffers**
  - Push(), Pop()
  - 1-to-1, lock-free
  - Store pointers

- **Threads**
  - Start(), Stop(), doLoop()
  - Sleep/yield when idle

- These components have been chosen based on previous positive experience in multi-threaded server processes
Architecture of a thread pool engine

A programming framework for data streaming on the Xeon Phi - CHEP'16
Adaptation of thread engine to MIC PCIe

A programming framework for data streaming on the Xeon Phi - CHEP'16
From user’s perspective

- Implement a derived tProcessData class (one single method + factory)
  
  ```cpp
  processData(data* in, data * &out) { ... }
  ```

- Use the pool in main()
  - Instantiate a thread pool
    ```cpp
    new tProcessThreadPool(processDataFactory, nThreads)
    ```
  - Move data in and out in a loop
    ```cpp
    tProcessThreadPool ->Push (dataIn)
    tProcessThreadPool ->Pop (dataOut)
    ```
An example use case

• Goal: measure performance of BZIP
  • Because it is a task of similar nature than e.g. ITS cluster finder
  • Readily available from the MIC-recompiled bzip library

• Implementation
  • processData () is a single call to BZ2_bzBuffToBuffCompress( )

• Runtime main
  • Create a data set from local files
  • Instantiate the MIC thread pool with given number of threads
  • Loop pushing data in/ pulling data out
  • Measure throughput
Framework use case: BZIP benchmark (native exec)

Throughput (MB/s) vs. Number of threads for different processors:

- Sandybridge 2x 8 cores
- Sandybridge linear
- Skylake 1x 4 cores
- Skylake linear
- KNC 57 cores (native exec)
- KNC linear
- KNL 64 cores (native exec)
- KNL linear

Early framework prototype, Pseudo-random data

A programming framework for data streaming on the Xeon Phi - CHEP'16
Framework use case: BZIP benchmark (MIC PCIe)

Thread engine scalability (BZIP)

Latest framework prototype, ITS pixel data

A programming framework for data streaming on the Xeon Phi - CHEP'16
Writing native code for MIC

• Not necessarily different than for CPU!
• Use Intel compiler: ICC
  • Option –mmic to build native MIC code executable
  • And/or compiler hints in source code: __declspec__ target(mic)
    or #pragma offload_attribute (push,target(mic)) ... #pragma offload_attribute (pop))
• External libs can be used on MIC
  • need to recompile with appropriate flag to generate MIC code
  • might get messy if many hierarchical dependencies
• Good portability / cross-compile seen so far
  • Mostly, just add compiler/linker option
    For example, to build BZIP library, 3 lines changed in the Makefile
  • Little or no change in the code
    • If necessary, can be isolated with #ifdef __MIC__ ... #endif
    • e.g. MIC does not know how to sleep()...
MIC pitfalls and hints ... things to consider

• I/O optimization
  • Memory pre-allocation / alignment
  • Mapping / handling of persistent buffers in CPU and MIC memory
  • For indirect memory mapping between CPU and MIC, use `#pragma offload into targetptr`
  • Group data transfer (e.g. 32 at a time)
  • Asynchronous offload (transfer / compute) to avoid idling during transactions (like double buffering)
  • All data moves are initiated from CPU side, asymmetric handling of transfers

• Thread scheduling
  • Delay vs. sleep vs. yield
  • Start persistent threads on MIC with `#pragma offload signal()`

• Debugging offload sometime difficult with wild data pointers
  • Same piece of source code runs on CPU and MIC... match address to variables not always easy

• A framework taking care of these issues is good value for end-users!
Summary

• MIC offload optimization is not easy but possible

• Programming and runtime framework available
  • Provides simple way to implement and run user code on MIC
  • Provides distribution of data stream chunks to the cores
  • Running on many cores (MIC or CPU)
  • Low framework overhead, good scalability measured
  • Hides MIC offload tricks
  • Allows to focus on the processing algorithms optimization on target hardware

• Status
  • Prototype, good enough to perform benchmarks and test different hardware
  • Will be used to evaluate KNL PCIe for ALICE O2
  • Might be extended further (lots of ideas on the “to-do list”: NUMA, etc)
Backup
**Performance of the building block**

CPU usage as function of FIFO throughput

- Skylake - threads on different cores
- Skylake - threads on same core
- Sandy Bridge - threads on different cores
- Sandy Bridge - threads on same core

CPU distribution with `numactl`

- CPU < 1 % up to 50KHz
- CPU < 10 % up to 1MHz
- 10-30 MHz maximum

A programming framework for data streaming on the Xeon Phi - CHEP'16
MIC i/o

- 6.4 GB/s in or out when data chunks big enough (250kB)
- Maximum 100-200k transfers/s

A programming framework for data streaming on the Xeon Phi - CHEP'16
Offload thread loop

- All data moves are initiated from CPU side
  1. Get list of buffers available at output
  2. Move CPU→MIC a block of buffers available at input
  3. Move MIC→CPU a block of buffers available at output

- 1-2-3 instead of 2-1-3: 1 needs MIC exec (in-exec-out), can be done asynchronously while 2 (data-transfer in only)