ITEA

INFORMATION TECHNOLOGY FOR EUROPEAN ADVANCEMENT

PERFCLOUD

H4H (ITEA2 09011)

Optimize HPC Applications on Heterogeneous Architectures

Deliverable: D5-4.1.1

Report on costs and benefits of adopting a heterogeneous architecture.

Version: V1.1

Date: January 2014

Authors: Bull, CEA, UVSQ

Status: Final

Visibility: Public

HISTORY

Document version #	**Date**	**Remarks**	**Author**
V1.0	January 2015	Final Draft	Bull, CEA-DAM & UVSQ
V1.1	January 2015	After review	

TABLE OF CONTENTS

1. Introduction 4

2. Programming models 5

2.1 NVidia GPU: KEPLER generation 5

2.2 Intel Xeon Phi 6

2.2.1 Different environments adapted for XeonPhi 6

2.2.2 Standards supported by different manufacturers 11

2.2.3 Comparison: OpenACC vs the others 11

2.2.4 Programming models summary 12

3. Architectures and Performances 14

3.1 Memory benchmark 15

3.1.1 Memory bandwith 15

3.1.2 Memory latency 16

3.2 Mathematics benchmarks 16

3.2.1 Dgemm 16

3.2.2 Linpack 16

3.3 Cryptographic benchmarks 17

3.3.1 Sha-1cryptography 17

3.3.2 Aes cryptography 18

3.4 Bandwith bus PCI 18

3.5 More tests on Intel XeonPhi 19

3.5.1 Vectorization is key for the KNC 19

3.5.2 A scalable architecture but… 20

3.5.3 Programming models Performances on XeonPhi 20

4. Conclusion 23

4.1 Programming models 23

4.2 Architecture and Performance 24

4.2.1 Benchmarks results 24

4.2.2 Optimization on KNC has a positive impact on standard Xeon processors 25

4.2.3 More studies are needed to understand this architecture 26

5. Abreviations and Acronyms 27

6. Bibliography 27

1. Introduction

This document analyses the use cases carried out during the H4H WP5.0 project to experiment with the Xeon-Phi type of accelerated architecture, identify respective costs and benefits, and make a synthesis of the results integrating some comparison with the GPU architecture.
This report is cut in two points of view: a first one about programming models and tools used to help the developments of optimized codes for these hybrid architectures, and a second one about architecture and performance.

As mentioned in the first delivered document (D5-1.2.1) of this workpackage 5.1 dedicated to the Xeon Phi accelerators, the competition is still strong between the GPUs made by NVIDIA and the Xeon Phi from Intel. Even similarly packaged as a PCIe device, the Xeon Phi architecture differs significantly from current GPU architectures.
This new era of multi- and many-core computing has been disruptive to the software industry, as it requires that existing applications be redesigned to exploit parallelism (rather than clock speed) to achieve high application performance on this new parallel hardware.

Bull offered SandyBridge then IvyBridge blades hosting 2 GPUs Nvidia or 2 Xeon Phi boards. We have measured the performance of these blades, their power consumption, their efficiency, and compared them. We will also consider the different ways of programming these two objects, and make a comparison.

CAPS integrated the support of Intel Xeon Phi in their programming models and libraries (openHMPP).

CEA realized a set of tests mainly based on the use of the HydroC benchmark [Github], in order to understand the value of an hybrid architecture and more specifically for Xeon-Phi processors.

UVSQ did studies and developments in its tool MAQAO to support Intel Xeon Phi processors.

2. Programming models

The type of machine that we consider in this report is heterogeneous. Generally, these machines have several types of processor architectures: generalist processors (for example Intel Xeon) and processors more specialized for intensive computing (Intel Xeon Phi, GPU).

Accelerators considered in this document are all extremely parallel. For example, programs optimized for NVIDIA GPUS of KEPLER generation must operate thousands of cores. Those created for the Xeon Phi must exploit vector instructions as well as hundreds units of calculation.

Memory areas available for generalist processors and accelerators are generally disjointed. Movements of data should therefore be launched to feed an accelerator and to repatriate results calculated by an accelerator.

In this model, accelerators are dependent on the central processor. It is the central processor that leads the program flow, and offloads during its run the highly parallel parts of the program.

Thus, to operate a heterogeneous architecture, 3 major issues should be treated:

- Parallelization of code: accelerators are extremely parallel
  - Vectorization: some accelerator as the Xeon Phi have vector calculation units
- Code offload: parts of the application must be able to be deported on an accelerator
- Data transfers: disjointed memory areas involve movements of data between accelerator's memory and main memory.

2.1 NVidia GPU: KEPLER generation

This document makes a focus on Xeon Phi studies, even if there are some comparisons and benchmarks realized with KEPLER accelerators in this document.

See the previous H4H studies and deliverables to have more details on NVIDIA GPUs programming models and tools.

2.2 Intel Xeon Phi

Xeon Phi Knight Corner (KNC generation) accelerators have two modes of operation: they can be considered as accelerators, and therefore are controlled by the central processor.

They can also be in native mode. In this mode, Xeon Phi is more than an accelerator, it becomes a full compute node. In this mode, the only essential issue is parallelizing and vectorizing the code.

This native mode will be predominant in the next generation of type Knight Landing (KNL). Indeed, these Xeon Phi may be used as a central processor: an application can only operate with Xeon Phi.

It should be noted also that manufacturers like NVIDIA provide architectures where generalist (type ARM) processors share memory with the GPU. Thus, the problem of data transfers will dramatically change with the next generation of accelerators.

In this chapter, we focus on this type operating mode. In some cases, we will deal with the particularities of the native mode of the Xeon Phi.

To program these architectures, the programming model is crucial. It must be able to manage the heterogeneity of computing resources and effectively use this considerable number of cores.

Much of environments have been proposed to facilitate the use of these hybrid architectures with strong parallelism. We are interested in this document to evaluate one of them: Open-ACC, which has the advantage of being open and provide a programming generic interface usable on all processors and accelerators. The portability of code provided by OpenACC, as well as its support by a large family of compilers and tools are major criteria to promote the adoption by a large application community.

Rather than describe in detail OpenACC, we have chosen to confront it to other parallel programming models.

2.2.1 Different environments adapted for XeonPhi

2.2.1.1 OpenACC

OpenACC is a parallel programming model for heterogeneous architectures based on a set of classic CPUs (Intel Xeon processor for example), and on a set of accelerators (GPUs, Xeon Phi).

The execution model is based on the offload of code. This means that an OpenACC application runs on the CPU and some very parallel execution parts are deported to the available accelerators.

OpenACC is based on compile-time directives. Thus, like OpenMP, the programmer annotates a sequential code. It says, for example, where a memory transfer should be launched, what loops should be parallelized, or when a portion of code should be executed on an accelerator.

The OpenACC provides C, C++, and Fortran language support.

2.2.1.2 Intel LEO (Language Extensions for Offload)

LEO is an extension, based on compilation directives, made by Intel to its compilers v.2013. LEO allows, within a Fortran, C, C++ program to handle transfers memory between CPU and accelerator, and to indicate functions to compile for Xeon Phi. This allows for example to encapsulate OpenMP 3.x or Cillk constructs in a LEO structure and run on Xeon Phi.
Intel LEO has the same memory model and the same execution model compared to OpenACC. But the programming model is not fixed. Within a construction LEO, the programmer can directly use

directives OpenMP 3.x, Cilk, threads posix or any other programming environment directives. LEO takes care to prepare and set up the storage areas on Xeon Phi, to launch the execution of code on the Xeon Phi, and the rest can be chosen by the programmer.

It is therefore possible to quite easily port a MPI/OpenMP program on Xeon Phi code: functions to send on Xeon Phi must be marked with LEO directives, and the compiler generates automatically on the Xeon Phi code for these portions of code.

2.2.1.3 OpenMP 4.0

The new OpenMP standard, announced in July 2013, adds a new set of compiler directives to OpenMP to exploit architectures based on accelerators. Intel is one of the first manufacturers to support OpenMP 4.0 for the Xeon Phi.

With the addition of extensions `target` in OpenMP 4.0, it is now possible to have an OpenMP code which supports accelerators. Like OpenACC or LEO, OpenMP is based on a memory model that exposes the separation between CPU memory and accelerator memory.

In order to address the new heterogeneous hardware, it was added to OpenMP 4.0 directives roughly equivalent to those available on OpenACC to control the disjoint memory spaces. Clauses, which allow defining a set of threads have been added (clause team).

In OpenACC loops are parallelized with threads and vectors; compiler can choose, for example, to match the vectors the vector units and the threads to the different cores available.

In OpenMP 4.0, the vectorization of a loop must be explicitly requested using the directive! $omp simd.

However, on Xeon Phi, this structuring is useless, and a simple OpenMP loop allows the addressing of all the hearts of the Xeon Phi. OpenMP 4.0 for Xeon Phi code can therefore be much simpler than a OpenMP 4.0 code for GPU.

This is a sample to illustrate that OpenMP (native mode) is easy to implement on a KNC. Using the same source code compiled for the different architectures and with the same test case, Figure 2 gives the performance result on a standard Xeon and Figure 3 the result on a KNC. The comparison of the two shows first, that the results are identical, second that the KNC has still a performance gap to bridge to significantly beat the Xeon family even if portability of the code is there. This is why the expectations on the next architecture (KNL) are so high.

```
env OMP_NUM_THREADS=16 time ./hydroc -i input.nml
HydroC: allocated time 1800s time guard 300s
Centered test case : 1002 1002
Forcing tilesize to 60
HydroC starting run with 1156 tiles on cirrus50
CPU name: Intel(R) Xeon(R) CPU E5-2650 0 @ 2.00GHz
+-------------------+
|GlobNx=2000        |
|GlobNy=2000        |
|nx=2000            |
|ny=2000            |
|ts=60              |
|nt=1156            |
|morton=0           |
|numa=1             |
|tend=100.000       |
|nstepmax=10        |
|noutput=0          |
|dtoutput=0.000     |
|dtimage=0.000     |
+-------------------+
Hydro: OpenMP max threads 16
Hydro: OpenMP num threads 1
Hydro: OpenMP num procs   16
Hydro: MPI is present with 1 tasks
 Initial dt 1.33631e-03
Iter      1 Time 0.00133631    Dt 0.00133631    (0.158710 25.203125 Mc/s 1.545460 GB)  1499.470539
Iter      2 Time 0.00267261    Dt 0.00133631    (0.161334 24.793353 Mc/s 1.545460 GB)  1499.309063
Iter      3 Time 0.00570914    Dt 0.00303653    (0.159744 25.040135 Mc/s 1.545460 GB)  1499.149215
Iter      4 Time 0.00874568    Dt 0.00303653    (0.160121 24.981039 Mc/s 1.545460 GB)  1498.988987
Iter      5 Time 0.0124942     Dt 0.00374854    (0.160982 24.847552 Mc/s 1.545460 GB)  1498.827907
Iter      6 Time 0.0162428     Dt 0.00374854    (0.158892 25.174270 Mc/s 1.545460 GB)  1498.668910
Iter      7 Time 0.0206309     Dt 0.00438811    (0.160178 24.972152 Mc/s 1.545460 GB)  1498.508631
Iter      8 Time 0.025019      Dt 0.00438811    (0.160238 24.962863 Mc/s 1.545460 GB)  1498.348297
Iter      9 Time 0.02957       Dt 0.004551      (0.160226 24.964758 Mc/s 1.545460 GB)  1498.187970
Iter     10 Time 0.034121      Dt 0.004551      (0.159670 25.051726 Mc/s 1.545460 GB)  1498.028199
End of computations in 1.61 s  (00:00:01.60) with 1156 tiles using 16 threads and 1 MPI tasks maxMEM 1.55GB
Total simulation time: 00:00:01.60 in 1 runs
Average MC/s: 25
```

Figure 2 : reference run of HydroC on a standard Xeon platform using 16 OpenMP threads (tiled version)

```
HydroC: allocated time 14400s time guard 900s
Centered test case : 1002 1002
Forcing tilesize to 60
HydroC starting run with 1156 tiles on cirrus10002
CPU name: 0b/01
+-------------------+
|GlobNx=2000        |
|GlobNy=2000        |
|nx=2000            |
|ny=2000            |
|ts=60              |
|nt=1156            |
|morton=0           |
|numa=1             |
|tend=100.000       |
|nstepmax=10        |
|noutput=0          |
|dtoutput=0.000     |
|dtimage=0.000     |
+-------------------+
Hydro: OpenMP max threads 240
Hydro: OpenMP num threads 1
Hydro: OpenMP num procs   240
 Initial dt 1.33631e-03
Iter      1 Time 0.00133631    Dt 0.00133631    (0.131905 30.324837 Mc/s 16.616894 GB)  13498.173688
Iter      2 Time 0.00267261    Dt 0.00133631    (0.131116 30.507413 Mc/s 16.616894 GB)  13498.041560
Iter      3 Time 0.00570914    Dt 0.00303653    (0.130354 30.685744 Mc/s 16.616894 GB)  13497.910324
Iter      4 Time 0.00874568    Dt 0.00303653    (0.130828 30.574517 Mc/s 16.616894 GB)  13497.778639
Iter      5 Time 0.0124942     Dt 0.00374854    (0.130423 30.669421 Mc/s 16.616894 GB)  13497.647205
Iter      6 Time 0.0162428     Dt 0.00374854    (0.131112 30.508245 Mc/s 16.616894 GB)  13497.515262
Iter      7 Time 0.0206309     Dt 0.00438811    (0.130269 30.705682 Mc/s 16.616894 GB)  13497.384151
Iter      8 Time 0.025019      Dt 0.00438811    (0.130152 30.733356 Mc/s 16.616894 GB)  13497.253161
Iter      9 Time 0.02957       Dt 0.004551      (0.129721 30.835312 Mc/s 16.616894 GB)  13497.122670
Iter     10 Time 0.034121      Dt 0.004551      (0.131054 30.521676 Mc/s 16.616894 GB)  13496.990770
End of computations in 1.32 s  (00:00:01.32) with 1156 tiles using 240 threads maxMEM 16.6GB
Total simulation time: 00:00:01.32 in 1 runs
Average MC/s: 30.7
```

Figure 3 : run of HydroC on a Xeon Phi platform using 240 OpenMP threads (tiled version)

2.2.1.4 OpenCL

OpenCL, based on the C language, allows the programmer to expose a massive parallelism to the compiler. Compared with OpenMP 4 or LEO, OpenCL is not based on compilation directives but exposes a real language for parallel programming. Intel supports OpenCL for its XeonPhi.
As OpenACC, OpenCL exposes the different heterogeneous architectures memory spaces. The data transfers are explicit and left to the user. Rather than using compilation directives, an API is defined.
Like other environments we've seen so far, the execution model is based on the code offloading: the programmer creates kernels, and decides at which time they shall be offloaded on an accelerator. The reader can refer to the OpenCL standard for more details on these features.
As with CUDA, OpenCL programmer exposes to the compiler a structured parallelism. Elementary, serial, code located in work-items, all of which are independent, and these work-items are organized (optionally) in work-group that are the equivalents of the blocks and threads CUDA, gangs and the vectors OpenACC. At launch, the programmer determines the number of work-groups and the number of work-items by work-groups: what are these work-items that run in parallel, on the different execution cores available on the hardware. The work-items are all running the same code.
The compiler can be decisive on Xeon Phi. Dimension 0 of the work-groups is indeed automatically vectorized by the Intel's OpenCL compiler. Dependency problems, which can occur when one vectorizes the loop are here naturally avoided since the work-items are independent by construction.

Like OpenACC, OpenCL supports a third level of parallelization, which are vector instructions. The OpenCL standard indeed provides vector data types (for example groups of four floating, called float4), which explicitly allow to perform vector operations. This possibility does not seem to be supported by Intel: Intel prefers to leave its OpenCL compiler automatically vectorizes the code and application programmers to avoid trying to manually vectorize the code.

This is the sample of HydroC test realized with OpenCL:

```
Hydro:  OpenCL compute unit type = ACC
+-------------------+
|nx=2000            |
|ny=2000            |
|nxystep=128        |
|tend=1600.000      |
|nstepmax=10        |
|noutput=0          |
|dtoutput=0.000     |
+-------------------+
Hydro starts in double precision.
Nb platform : 1
[0] Profile : FULL_PROFILE
[0] VERSION : OpenCL 1.2 LINUX
[0] NAME : Intel(R) OpenCL
[0] VENDOR : Intel(R) Corporation
[0]   EXTENSIONS   :   cl_khr_icd   cl_khr_global_int32_base_atomics   cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics     cl_khr_local_int32_extended_atomics     cl_khr_byte_addressable_store
cl_khr_spir cl_khr_fp64
(0)  :: device  maxcu  1  mxwkitdim  3  mxwkitsz  8192  8192  8192  mxwkgsz  8192   mxclockMhz  2100  mxmemallocsz
12057 (Mo) globmemsz 48230 (Mo) type 2 [CPU]
   extensions:     cl_khr_icd     cl_khr_global_int32_base_atomics     cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics     cl_khr_local_int32_extended_atomics     cl_khr_byte_addressable_store
cl_khr_spir cl_intel_exec_by_local_thread cl_khr_depth_images cl_khr_3d_image_writes cl_khr_fp64
Device 0 supports double precision floatting point
   prefered vector size: c=1 s=1 i=1 l=1 f=1 d=1
(1) :: device maxcu 236 mxwkitdim 3 mxwkitsz 8192 8192 8192 mxwkgsz 8192  mxclockMhz 1052 mxmemallocsz
1924 (Mo) globmemsz 5773 (Mo) type 8 [ACCELERATOR]
   extensions:     cl_khr_icd     cl_khr_global_int32_base_atomics     cl_khr_global_int32_extended_atomics
cl_khr_local_int32_base_atomics     cl_khr_local_int32_extended_atomics     cl_khr_byte_addressable_store
cl_khr_spir cl_khr_fp64
Device 1 supports double precision floating point
   prefered vector size: c=1 s=1 i=1 l=1 f=1 d=1
Hydro: 000 has 0 GPU
Hydro: 000 has 1 ACC
Hydro: 000 uses ACC 0
[0] : nbdevices = 2
Building an ACC version
Build           OpenCL           (opts="-cl-mad-enable           -DNVIDIA           -DHASFP64           -
I/ccc/ghome/ocre/coling/Github/Hydro/HydroC/oclHydroC_2D/Src ") OK.
Centered test case : 1002 1002
Hydro 0: initialize acc 0.496170s
--> step=1     1.33631e-03, 1.33631e-03  (0.364s) *
--> step=2     2.67261e-03, 1.33631e-03  (0.304s)
--> step=3     5.70914e-03, 3.03653e-03  (0.361s) *
--> step=4     8.74568e-03, 3.03653e-03  (0.301s)
--> step=5     1.24942e-02, 3.74854e-03  (0.332s) *
--> step=6     1.62428e-02, 3.74854e-03  (0.308s) (13.0 MC/s)
--> step=7     2.06309e-02, 4.38811e-03  (0.333s) (12.0 MC/s) *
--> step=8     2.50190e-02, 4.38811e-03  (0.301s) (13.3 MC/s)
--> step=9     2.95700e-02, 4.55100e-03  (0.336s) (11.9 MC/s) *
--> step=10    3.41210e-02, 4.55100e-03  (0.302s) (13.2 MC/s)
Hydro ends in 00:00:09.834s(9.834) without init: 3.258s. [DP]
    GATCON     CONPRI     EOS        SLOPE      TRACE      QLEFTR     RIEMAN     CMPFLX     UPDCON     COMPDT
MAKBOU    ALLRED
PE0 0.577700   0.166104   0.129874   0.174436   0.365882   0.264103   0.571189   0.169403   0.587714   0.201799
0.034090   0.000000
%    17.817641 5.123032   4.005629   5.380016   11.284665 8.145553   17.616830 5.224789   18.126475 6.223969
1.051401   0.000000
Average MC/s: 12.7
```

Figure 4 : test case using the OpenCL version of HydroC on a KNC (no tiles).

2.2.2 Standards supported by different manufacturers

The OpenACC standard is supported by NVIDIA and OpenMP 4 by Intel. NVIDIA is part of the initiators of the OpenACC standard and recently acquired PGI, a pioneer in the operation of OpenACC. Intel is one of the first editors of compiler to support the OpenMP 4 extensions for accelerators.

OpenACC is clearly designed for GPU architectures, where parallelization, inherited from CUDA, is carried out in a structured form (gang, vector, workers). The design of OpenACC approach seems to be bottom-up: it was started from hardware architecture GPU to create a language suitable for it.

Accelerator extensions to OpenMP 4 were rather designed the other way: we begin from a 15 years model old to adapt it to GPU architectures without however breaking the entire model. Where optional gang and distribute directives for the GPU.
For the Xeon Phi architecture however, which is much more classic than a GPU (cache coherent, standard multicore processors) OpenMP code doesn't have to use these new directives.

Thus, for the moment, OpenMP 4 seems natural on Xeon Phi and OpenACC on GPU.

2.2.3 Comparison: OpenACC vs the others

Although OpenACC supports the compiling function to an accelerator (directive routine), OpenACC is much less generic than LEO.
The genericity of LEO is related to the Intel compiler and the characteristics of the Xeon Phi. Indeed, because Intel has entirely enabled its compilers for Xeon Phi, any code supported by Intel can be compiled to Xeon Phi, and thus integrated into a LEO section.
For GPU architectures, much less generic than the Xeon Phi, it seems extremely difficult to operate LEO without adding a lot of constraints. Thus, LEO is probably exploitable on Xeon Phi only.
As LEO treated only the problem of the code offload and memory transfers, there is no interest when a Xeon Phi is used natively. It becomes also unnecessary in the case of the KNL, when used as the central processor.

The difference of parallelism between OpenMP and OpenACC involves a few semantic differences.
In OpenMP, it is possible to have variables shared by thread. Thus, in line 11 of the OpenMP page 12 example, some data are defined local to each thread (clause private).
In OpenACC, the private clause exists but applies, not to the threads, but gangs. Thus, the workers or the vectors of a gang share the same memory private zone. For OpenMP semantics, it is necessary to ensure that each gang creates a thread. To do this, an equivalent code OpenACC to our example OpenMP would be:

```
1.      !$acc parallel  present(x,y,kstrten,fil1,fil2)   num_workers(1) vector_length(1)
2.      !$acc loop  collapse(2)  private(locarra1,locarra2)
3.        do i3=0,n3
4.          do i2=0,n2
```

In line 1, we force a single worker and a single vector by gang : the private array defined on line 2. becomes local to each thread.
However, this approach is not ideal and prevents, for example, effectively use of GPU, since we need multiple threads by gang for these architectures.

Another difference concerns the synchronizations. With OpenACC, after a loop construction there is no implied barrier as in OpenMP. It is possible to add an end parallel directive to force a synchronization barrier.
Because OpenACC was designed for the specific type GPU architectures, synchronization directives are much more limited than in OpenMP.
One of the advantages of the approaches by directives such as OpenACC is the progressive porting of applications to accelerators. It is indeed possible to add gradually different compile-time directives to optimize more and more the code. For example, starting by the parallelization of the loop and then deeply optimize memory transfers, while maintaining a close code compared to the original code base.

The intrusiveness level in the source code is therefore in favour of the solutions by directives. Nevertheless, memory transfers are the responsibility of the programmer, and in realistic cases, where such transfers should be avoided as possible, or covered by calculations, a very complex code must be added; a large number of directives is necessary and involves a large intrusiveness levels in the code. For memory transfers, the benefits of the directves are not clear compared with the addition of some OpenCL function calls.

It should also be noted that to port an application on a multi-core architecture, it can be difficult to port a loop only by annotating it. Other, perhaps more important factors are to be taken into consideration.
Indeed, to effectively exploit hundreds of cores in a compute node, restructuring of algorithm must sometimes be considered.
To give an example, SPECFEM3D designers have had to add a coloring during the phase that creates the mesh to break dependencies in the loops of calculation [4] and therefore to effectively use a multi-core architecture.
In these extreme cases, one wonders if an approach of gradual application porting is possible. Having to deeply modify an algorithm might be the opportunity to spend a true parallel language, such as OpenCL.

2.2.4 Programming models summary

We have in this chapter 4 compared OpenACC OpenMP 4, LEO and OpenCL.

OpenCL is supported by a wide range of materials: classics, NVIDIA GPU, GPU AMD processors, Xeon Phi... However, it seems difficult that a same kernel is also efficient for a different set of hardware. Optimizations must be made in the OpenCL kernels to target different hardware.

Optimizations should be also performed in approaches by compile-time directives: for OpenMP 4 or OpenACC, it may be useful to define values for the team/gang clauses in order to exploit the material.

Thus, for the moment, it does not appear that a single programming environment can allow targeting different hardware from the same source code with a highly optimization level. Manual optimizations are still needed, either for OpenACC or OpenCL.

OpenACC has directives suitable for GPU architectures through structuring of parallelism into three levels. It is possible to configure elements of low level, such as the size of the gang, and use features of these architectures such as management, - sometimes manual memory caches (cache directive).

OpenACC has been influenced by CUDA, the structuring of parallelism, essential element of CUDA, was reused in OpenACC, which makes it suitable for NVIDIA GPU architectures.
OpenMP 4 is an evolution of the OpenMP standard. An OpenMP code can easily be extended to an accelerator architecture. However this facility depends on the considered target. The Xeon Phi, close to classical multicore architecture, makes it naturally adapted to OpenMP 4 buildings. For a GPU it is necessary to use new buildings to get a parallelization for these architectures.

OpenMP 4 is very generic, one may wonder if all OpenMP 4 codes can be compiled to more constraint architectures, such as GPU. With OpenACC, which the GPU hardware constraints are present in the language, it should be possible to more easily ensure that a valid OpenACC code will run on GPU. The current lack of compiler OpenMP 4 for GPU prevents us from doing this type of analysis.
LEO, meanwhile, allows to separate the problems of memory transfers / offloading, and the parallelization of code. It makes it extremely interesting. However, LEO is only available with the Intel compilers.
Finally, OpenCL defines a different approach: rather than deduct a parallelism from loops, OpenCL asks the programmer to directly express a massive parallelism. This allows you to more finely control the parallelization of the code but involves a somewhat larger intrusiveness levels in code against approaches by directives.
However, an application where porting code is complex, i.e. which require modifications to algorithm or reorganization of code, taking the opportunity to exploit a true parallel language can be a good option.

3. Architectures and Performances

The material studied as well as versions of software components are collected in the following table:

SKU	nVidia K20m, 2496 cores 0.71 GHz	nVidia K20Xm, 2688 cores 0.732 GHz	nVidia K40m , 2880 cores 0.88 GHz
	5 GB GDDR5 at 2.6 GHz ECC enabled	6 GB GDDR5 at 2.6 GHz ECC enabled	12 GB GDDR5 at 2.6 GHz ECC enabled
Host	Sandy Bridge E5-2470 16 cores 2 chips 2.30 GHz	Sandy Bridge E5-2470 16 cores 2 chips 2.30 GHz	Sandy Bridge E5-2470 16 cores 2 chips 2.30 GHz
	48 GB at 1.600 GHz	48 GB at 1.600 GHz	48 GB at 1.600 GHz
RedHat Distribution	RHEL 6.3 Santiago	RHEL 6.3 Santiago	RHEL 6.4 Santiago
Linux Kernel	2.6.32-279.el6.x86_64	2.6.32-279.el6.x86_64	2.6.32-358.el6.x86_64
BIOS	InsydeH2O Version 3.72.35BIOS_I3GPU_2030. 03.06	InsydeH2O Version 3.72.35BIOS_I3GPU_2030. 03.06	BIOSX06.036.00.101
Driver Version	304.64	304.64	319.60
Cuda version	5.0	5.0	5.0

SKU	Xeon Phi 3110P 57 cores, 1.1 GHz	Xeon Phi 5110P 60 cores, 1,05 GHz	Xeon Phi 7210X 61 cores, 1,238 GHz
	3 GB GDDR5 at 2.75 GHz ECC enabled	8 GB GDDR5 at 2.75 GHz ECC enabled	8 GB GDDR5 at 2.75 GHz ECC enabled
MPSS	2.1. 6720-13	2.1. 6720-13	2.1. 6720-13
Host	Sandy Bridge E5-2470 16 cores 2 chips 2.30 GHz	Sandy Bridge E5-2470 16 cores 2 chips 2.30 GHz	Ivy Bridge E5-2697 v2 24 cores 2 chips 2.70 GHz
	48 GB at 1.600 GHz	48 GB at 1.600 GHz	64 GB at 1.8 GHz
RedHat Distribution	RHEL 6.3 Santiago	RHEL 6.3 Santiago	RHEL 6.3 Santiago
Linux Kernel	2.6.32-279.el6.x86_64	2.6.32-279.el6.x86_64	2.6.32-279.el6.x86_64
BIOS	InsydeH2O Version 3.72.35BIOS_I3GPU_2030. 03.06	InsydeH2O Version 3.72.35BIOS_I3GPU_2030 .03.06	BIOSX03.033.202

The following chapters describes the results of different kind of benchmarks realized comparatively on Xeon, XeonPhi and NVIDIA GPUs.
These different benchmarks enable us to compare and show strengths and weaknesses of these architectures.

3.1 Memory benchmark

3.1.1 Memory bandwith

Bandwith during transfer of buffers on the GDDR5 or on DDR3 memory:

Bandwith	K40m 0.88 GHz	K20Xm 0.73 GHz	K20m 0.71 GHz	Xeon Phi 7120 1.238 GHz	Haswell E5-2697 v3 2.6 GHz
Threads	15*192=2880	14*192=2688	13*192=2496	61*44 = 244	28
Memory	11 GB 3.004 GHz	6 GB 2.6 GHz	5 GB 2.6 GHz	8 GB 2.75 GHz	128 GB 2.133 GHz
Engraving	28 nm	28 nm	28 nm	22 nm	22 nm
BW Read	347 GB/s	372 GB/s	359 GB/s	168 GB/s	119 GB/s
BW Write	157 GB/s	157 GB/s	115 GB/s	46 GB/s	49 GB/s

Stream mccalpin bandwith during transfer of buffers on the GDDR5 or on DDR3 memory:

Streams mccalpin	Xeon Phi 3110P 1.10 GHz	Xeon Phi 5110P 1.053 GHz	Xeon Phi 7210P 1.238 GHz	Sandy Bridge E5-2680 2.7 GHz	IvyBridge E5-2697v2 2.7 GHz	Haswell E5-2697 v3 2.6 GHz	K40m* 0.88 GHz
Threads	57*4=228	60*4=240	61*4=244	16	24	28	15*192=2880
Memory	3 GB 2.5 GHz	8 GB 2.75 GHz	16 GB 2.75 GHz	64 GB 1.6 GHz	64 GB 1.8 GHz	128 GB 2.133 GHz	11 GB 3.004 GHz
Engraving	22 nm	22 nm	22 nm	32 nm	22 nm	22 nm	28 nm
StreamsTriad	106 GB/s	137 GB/s	130 GB/s	79 GB/s	99 GB/s	115 GB/s	152 GB/s
StreamsTriad (Intel mesure)	119 GB/s	147 GB/s					

All measurements are in GB, i.e. 10^9Byte.

* : for K40m card, Streams Triad are measured with another implementation of test triad.

The bandwidth of Xeon Phi cards is comparable to that of the GPU and much better than most of the Intel Xeon processors.

3.1.2 Memory latency

Latency	Xeon Phi 3110P 1.10 GHz	Xeon Phi 5110P 1.053 GHz	Xeon Phi 7210X 1.238 GHz	Sandy Bridge E5-2680 2.7 GHz	IvyBridge E5-2697v2 2.7 GHz	Haswell E5-2697 v3 2.6 GHz
Local	190 ns	190 ns	178 ns	78 ns	78 ns	70 ns
Far (inter cpus)	-	-	-	123 ns	138 ns	180 ns

The memory latency is more than two times greater than the memory latency of a classic cpu.

3.2 Mathematics benchmarks

3.2.1 Dgemm

Mathematical libraries are necessary for this bench. With cuda, CBLAS library is used, and for Xeon and XeonPhi, the MKL library. These libraries are provided by the hardware vendors.

Performance	Xeon Phi 3110P 1.10 GHz	Xeon Phi 5110P 1.053 GHz	Xeon Phi 7120X 1.238 GHz	Sandy Bridge E5-2680 2.7 GHz	IvyBridge E5-2697v2 2.7 GHz	Haswell E5-2697 v3 2.6 GHz
Core Number	57	60	61	16	24	28
Dgemm theoretical (GFlops)	1003	1011	1208	396	595	1165
Dgemm measured (GFlops)	792 (79%)	769 (76%)	957 (79%)	360 (90%)	531 (89%)	992 (85%)
Dgemm measured Intel (GFlops)	762 (76%)	775 (77%)	-	-	-	-

The performance for this bench is below expectations.
In addition, performance with nVidia Kepler cards gave the following results:

Performance	K40m 0.88 GHz	K20Xm 0.73 GHz	K20m 0.71 GHz
Threads	15*192 = 2880	14*192 = 2688	13*192 = 2496
Dgemm theoretical (GFlops)	2534	1962	1772
Dgemm measured (GFlops)	1151 (45%)	1130 (58%)	870 (49%)

These results show the good performance of these cards in the calculation of floating double-precision numbers.

3.2.2 Linpack

We give the measures of this bench for equipment with XeonPhi cards on the one hand, and NVIDIA Kepler cards on the other. For best performance, this bench is customized specifically for each concerned material.

We use a binary provided by Intel, optimized for use in both two SandyBridge EP processors and two cards KNC.

NVIDIA provided us an optimized version of the hpl benchmark allowing to use both cards K20 to the best of their capacities, in parallel with the EP SandyBridge CPUs.

The results obtained are the following:

Performance	Xeon Phi 3110P 1.10 GHz	Xeon Phi 5110P 1.053 GHz	Xeon Phi 7210X 1.238 GHz	K20Xm 0.73 GHz	K20m 0.71 GHz	K40m 0.88 GHz	Haswell E5-2697 v3 2.6 GHz (2 sockets)
Linpack (GFlops)	1418	1644	2036	2100	2022	2305	830

We find the same ratio as for the dgemm test, **Intel Xeon Phi cards have a performance of 75% of nVidia Kepler cards**.

3.3 Cryptographic benchmarks

3.3.1 Sha-1cryptography

A bench that implements cryptographic sha-1 algorithm has been generated for the cards XeonPhi and NVIDIA. The results of cryptographic encryption flow :

Performance	K40m 0.88 GHz	K20Xm 0.73 GHz	K20m 0.71 GHz	Xeon Phi 7120X 1.238 GHz	Xeon Phi 5110P 1.053 GHz	Xeon Phi 3110P 1.10GHz
Threads	2080	2688	2496	244	240	228
mesure (Hash/s)	27000	25412	24000	12400	10350	10450

In comparison, on Xeon hardware the best performance obtained is 17012 Hash/s with a Haswell E5 2697 v3 2.6 GHz 28 cores AVX2 instructions and 256 bits vectors.

The best performances for logic operations with integers are provided by NVIDIA Kepler cards. The best Xeon Phi card is less performant than a recent bi-socket processor with many cores.

3.3.2 Aes cryptography

This test consists of encryption of buffers by a set of aes 128-bit keys and a measure of rate. An assembler version using vector Xeon 64 bits instructions has been developed to handle 8 buffers at the same time. We have ported this version of code, also in assembler, on Xeon Phi with 512 bits vectors.

The contribution of this technology is measured in the following table:

Performance	MIC 3110P 1.10 GHz	Ivy Bridge E5-2697 v2 2.7 GHz	Haswell E5-2697 v3 2.6 GHz
Test aes	46.94 Mtests/s	7.06 Mtests/s	9.35 Mtests/s
Test aes vectorized	156.13 Mtests/s	18.44 Mtests/s	22.08 Mtests/s
Enhancement ratio	3.33	2.61	2.36

The instruction set and vectorization of the Phi Xeon brings all its profit in this bench.
However, porting efforts were substantial. The instruction set of Xeon Phi is not compatible with that of xeon. In fact, in vectorize mode, 64 bits shift instructions, and byte manipulation instructions such as byte compare and byte shuffle do not exist with Xeon Phi. It took us three months to carry out this work which gives an idea of the difficulty of coding with Xeon Phi.

Otherwise, processors that support the « aes encryption new instructions ».AES-NI- give good results, so with the above Haswell E5-2697 v3, the result is 76.26 Mtests/s.

A cuda version with aes_cuda lib for NVIDIA equipment is not efficient and does not exceed 12.38 Mtests/s with a K40m card.

3.4 Bandwith bus PCI

Flow of copy buffers between the host and the device, in both senses:

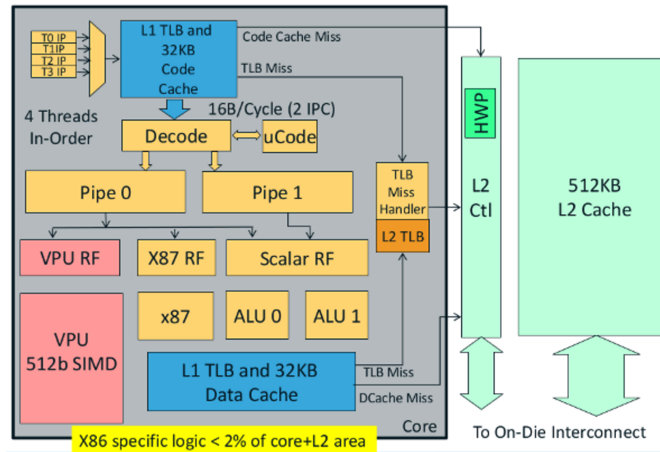
Bandwidth	K40m 0.88 GHz	K20Xm 0.73 GHz	K20m 0.71 GHz	Xeon Phi 5110P 1.053 GHz	Xeon Phi 3110P 1.10GHz
Host to Device	10.01 GB/s	5.75 GB/s	5.75 GB/s	6.6 GB/s	6.6 GB/s
Device to Host	9.49 GB/s	6.38 GB/s	6.38 GB/s	7.0 GB/s	6.9 GB/s

XeonPhi cards seem to best manage the PCIe Gen2 bus 16x width than NVIDIA Kepler cards, apart from the K40m card that manages PCI Gen3 bus.

3.5 More tests on Intel XeonPhi

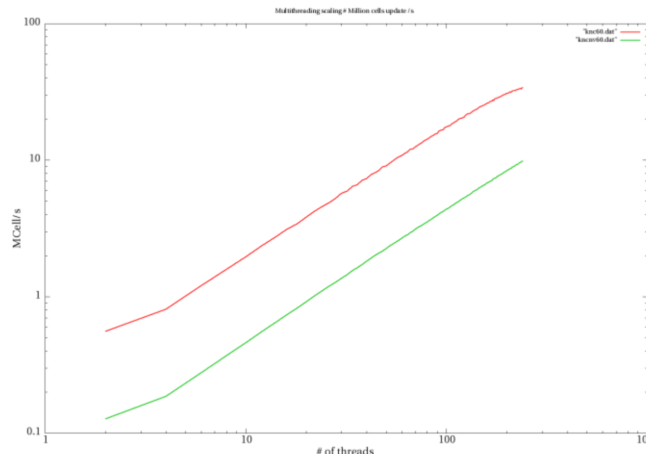
3.5.1 Vectorization is key for the KNC

The KNC architecture uses a vector unit to deliver the bulk of its performances as shown in Figure 5. While being multithreaded (4 ways), the sequential part of the core is fairly weak (an old design of 2002). Therefore most of the programming effort must be placed on vectorization.



• Figure 5 : KNC core internal

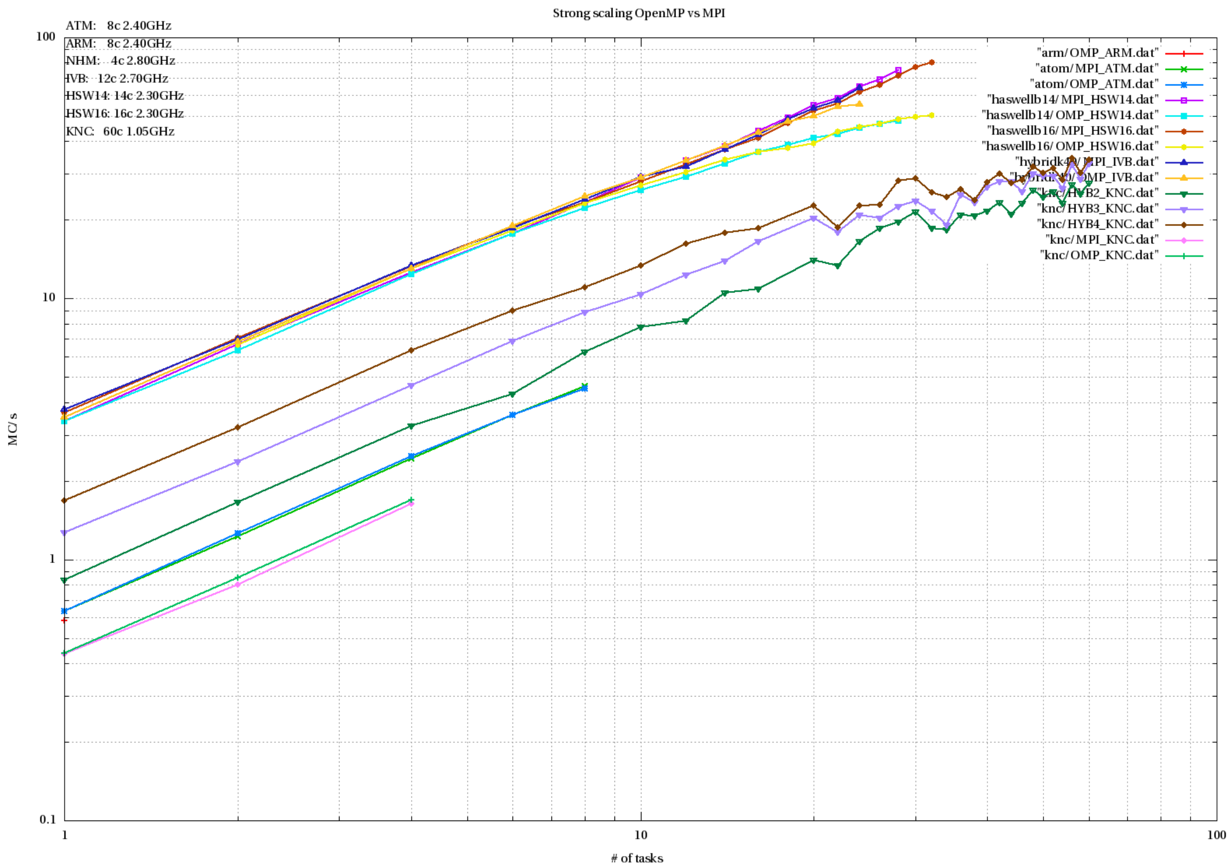
To demonstrate this fact, we did the following test illustrated in Figure 6: for different number of threads, we run the same test case with two versions of HydroC. The first version is compiled with "-O3" (which implies auto-vectorization) and the second one is compiled with "-no-vec" which inhibits vectorization. The figure shows clearly that vectorization is needed to get decent performances.



• Figure 6 : Influence of the vectorization of the HydroC benchmark. The higher the curve, the better is the performance.

On the other hand, the lower line of the following Figure 7 shows that the core is indeed not very powerful.

3.5.2 A scalable architecture but…



• Figure 7 : Scaling of HydroC on different architectures using either MPI or OpenMP

Since the KNC has 60 cores, it was important to start studying the impact of this large core count on the performances and the scalability of the programs. In Figure above, we study the strong scaling of HydroC. While the standard Xeon versions have a pretty regular behavior, the KNC versions (either OpenMP or MPI) show strange oscillations at large core count. This remark has no real explanation as of today.

3.5.3 Programming models Performances on XeonPhi

The objective of this section is to compare the performance of an OpenACC code to an equivalents code for OpenMP 4 and LEO on Xeon Phi. We use for this the CAPS HMPP compiler to compile a code OpenACC to Xeon Phi, and Intel fortran v.14 compiler to compile OpenMP 4 and LEO codes to Xeon Phi.

We use a convolution code from the BigDFT software that has been optimized for Xeon Phi in OpenMP, the heart of the calculation corresponds to the following code:

```
!$omp  parallel default (private) shared(y,x,kstrten,cx,cy,cz,n1,n2,n3),firstprivate(fil1,fil2)
```

```
!$omp do collapse(2)
  do i3=0,n3
     do i2=0,n2

<mise à jour d'un tableau local>

    do i1=0,n1
           tt1=locarra1(14 + i1)*cx
           tt2=locarra2(14 + i1)*cx

           do l=lowfil,lupfil
              j=i1+ l
              tt1=tt1+locarra1(14 + j)*fil1(l,1)-locarra2(14 + j)*fil2(l,1)
              tt2=tt2+locarra2(14 + j)*fil1(l,1)+locarra1(14 + j)*fil2(l,1)
           enddo

           y(i1,i2,i3,1)=y(i1,i2,i3,1)+tt1
           y(i1,i2,i3,2)=y(i1,i2,i3,2)+tt2

           kstrt1=kstrt1+tt1*locarra1(14 + i1)+tt2*locarra2(14 + i1)
        enddo
     enddo
end do
(…)
```

We show only a loop. In the original code, there are three which follow. Each loop treats a different dimension of a 3D matrix double precision. In this code, each OpenMP thread is expected to have a local array with the directive private.

We used strategies of parallelism for OpenACC, OpenMP 4.0 and LEO identical to those presented in the previous sections. Thus, in OpenACC, we forced the creation of a vector and a worker by gang, in order to have a local table by thread. Furthermore, in OpenACC, we have added guidelines “acc_end_parallel” after each loop to force a synchronization barrier. These adaptations were necessary to have three codes producing the same numerical results.
However, the codes are not exactly identical: in LEO, the full function of the convolution is remote.
Although it is provided by the standards OpenMP 4.0 and OpenACC, we could not manage to deport completely the convolution function in these two environments. For example, the Intel compiler stopped on an internal error during our attempts.
As a result, with OpenMP 4.0 and OpenACC only the function loops are deported. Thus, some scalar variables like loop counters are copied to each offset loop in OpenMP 4.0 and OpenACC. These small transfers are avoided with LEO.
In using CAPS HMPP, the convolution is performed on a matrix of size 135 x 140 x 145. For each cell of the table, the time shown is the average of 1000 executions. The standard deviations are small and are not indicated.

	OpenMP 4	OpenACC	LEO
Time (s)	0.06	0.055	0.026

The numerical results produced by the three versions of the convolution are of course identical.
As mentioned in previous section, the LEO version is more performant because some memory transfers were avoided.

LEO produced the best performance. Compiler OpenACC of CAPS and OpenMP 4.0 from Intel produce comparable performances.
However, the results should be interpreted with caution: in OpenACC and OpenMP 4.0, compilers forced us to allow a few additional copies which can explain differences in time.
In addition, the code of the convolution is small enough.
Concerning programming efforts to port the initial code OpenMP to Xeon Phi, LEO has been the easiest. It is quite natural to have a language that handles the code offload, and another the parallelization.
It also simple enough to port an OpenMP code to Xeon Phi using OpenMP 4.0.
The most difficult work was to port OpenMP code to OpenACC. Some differences, like the private clause of OpenMP or the absence of synchronization after the directives *!$acc* loop have required some adjustments in the code.

4. Conclusion

4.1 Programming models

We (Bull and CAPS) have presented the new OpenACC standard and compared it with other environments to operate hybrid architectures. We also conducted preliminary performance experiments on a same code using OpenACC, OpenMP 4 and LEO.

On a Xeon Phi type KNC, OpenMP 4.0, OpenCL and LEO work properly. We also managed to run code OpenACC on Xeon Phi with HMPP. Its performance is comparable to an OpenMP 4.0 code.

In all of the technologies described in this report, the memory transfers and code offload management are essentially identical. The change is the parallelization of the code. Thus, on the next single memory architectures, as the Xeon Phi KNL, LEO becomes useless (since it only manages transfers and code offload) but the question of choosing between OpenMP 4.0, OpenACC or OpenCL remains.

OpenMP 4.0 and OpenACC relate to a similar purpose: have a portable, easily embeddable code in applications that can operate a wide range of hardware architectures. These technologies are fairly recent. There are only a small number of compilers that support them. When the situation will stabilize, it will be relevant to assess in detail how an OpenACC code behaves on Xeon Phi, and how behaves an OpenMP 4.0 on GPU code.

However, with today information, it is clear that the architecture of the Xeon Phi naturally allows the use of OpenMP 4.0 code: an OpenMP program that is intended to be used on Xeon Phi needs only few changes to be functional. Conversely, changes are needed to bring an OpenMP code to OpenACC.

The design of OpenACC makes certainly more suitable for GPU architectures.

The OpenCL standard is also intended to create usable codes on different materials. In contrast to approaches by directives, it exposes the programmer a real language for parallel programming. Thus, the parallelization of code is not left to a compiler, but to the programmer. It is an approach more generic which allows parallelization of elements more complex than a simple loop. However, it is more intrusive in a source code of an existing application.

For code intended to be portable, i.e. code that can exploit GPU and Xeon Phi, the advantage should go to OpenACC or OpenCL. Indeed, OpenACC constructions and the OpenCL language are more constraint than OpenMP 4.0, and it seems appropriate to think that an OpenACC or OpenCL code has more chance to run efficiently on different accelerators than an OpenMP 4.0 code.

For a code designed only for the Phi Xeon architecture, OpenMP 4.0 seems the appropriate candidate.

Moreover, UVSQ had provided two major pieces of feedback, based on studies with MAQAO:

The first one comes from the first phase of the project, during the port of MAQAO on the Xeon Phi architecture. In order to evaluate the features and potential of the architecture, we have run a microbenchmark whose aim is to assess these. We have observed a real issue in terms of performance when dealing with last level cache. Performances were way far from the theoretical ones. That is how we concluded that only applications tailored to use L1 cache could really get the most out of the accelerator.

The second aspect was to evaluate the potential of a single core. We had the opportunity to test it when we provided help to the CEA-LIST on optimizing a "Hamming distance" computation kernel. On Xeon Phi, getting the best performance means being able to use vectorized code. We were never able to make the compiler produce an optimal code. So the only option left was to use intrinsics which are not portable (only on Xeon PHI). Once again, such an approach will not benefit the majority of applications.

So, maintain a different code for different types of architectures is still an extremely difficult task for application designers. This problem of portability of performance is a major problem to be addressed to prevent applications from being dependent on a certain architecture, and other dependent applications to other architectures. New programming models, associated with performant runtimes appear to be indispensable to deal effectively with the new problems brought by the characteristic of the new hardware architectures.

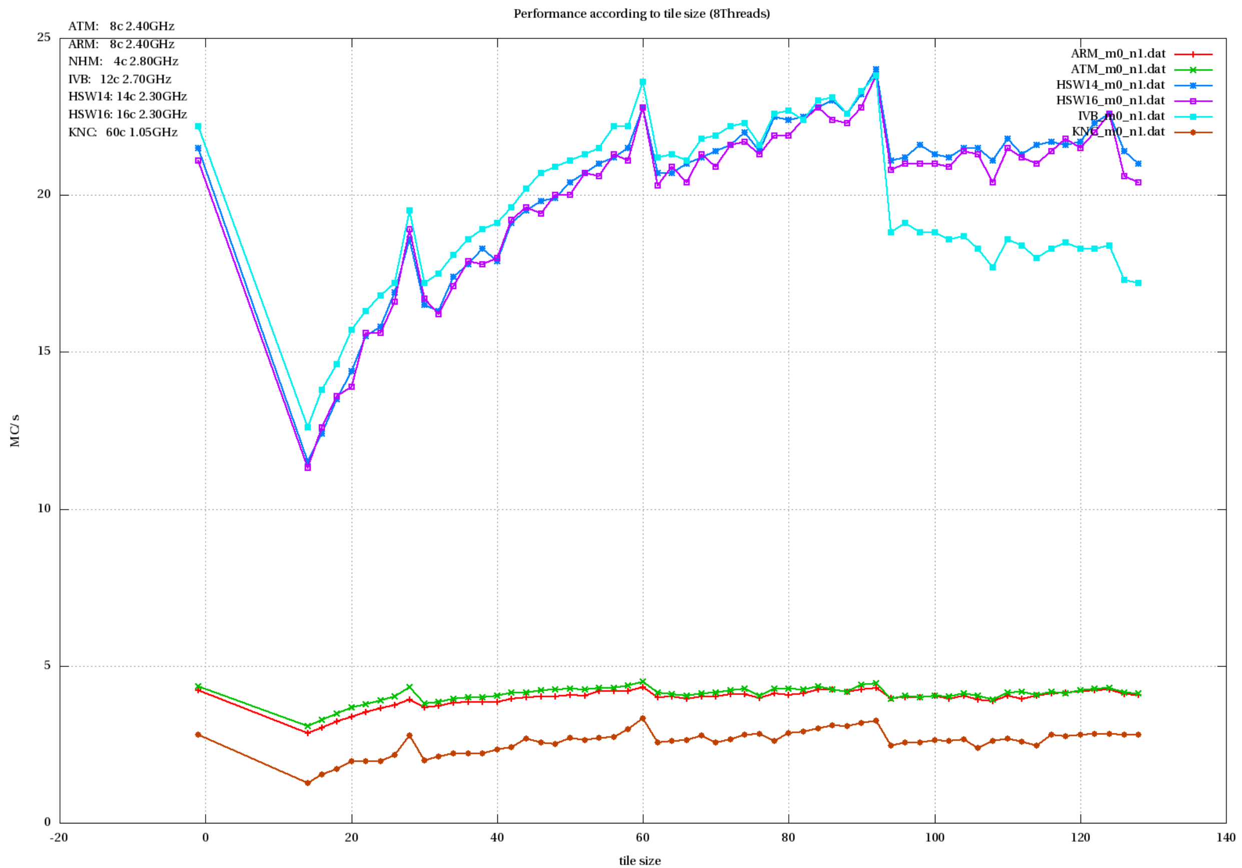
4.2 Architecture and Performance

4.2.1 Benchmarks results

The performances of Xeon Phi for programs that require access to memory are less good than NVIDIA Kepler cards.
Similarly, calculations with instructions with floating double-precision are less efficient than Kepler cards or with the 64-bit Xeon processors.
However, with Xeon Phi and the intrinsic, the programming on the host and the card are similar and easier to implement than with CUDA.
In fact, with CUDA, it is necessary to translate the code into independent threads while synchronizing the threads in time, what is not always easy. Certainly it is possible to use memory shared between threads but it is also risky.
On one hand, sometimes it is necessary to adjust the number of threads and registers used in a very fine way, by using tools such as « occupancy calculator » or the profiler nvvp.
On the other hand, Intel offers tools for debug and profiling constituting an uniform environment favorable to the development of the code and its improvement.

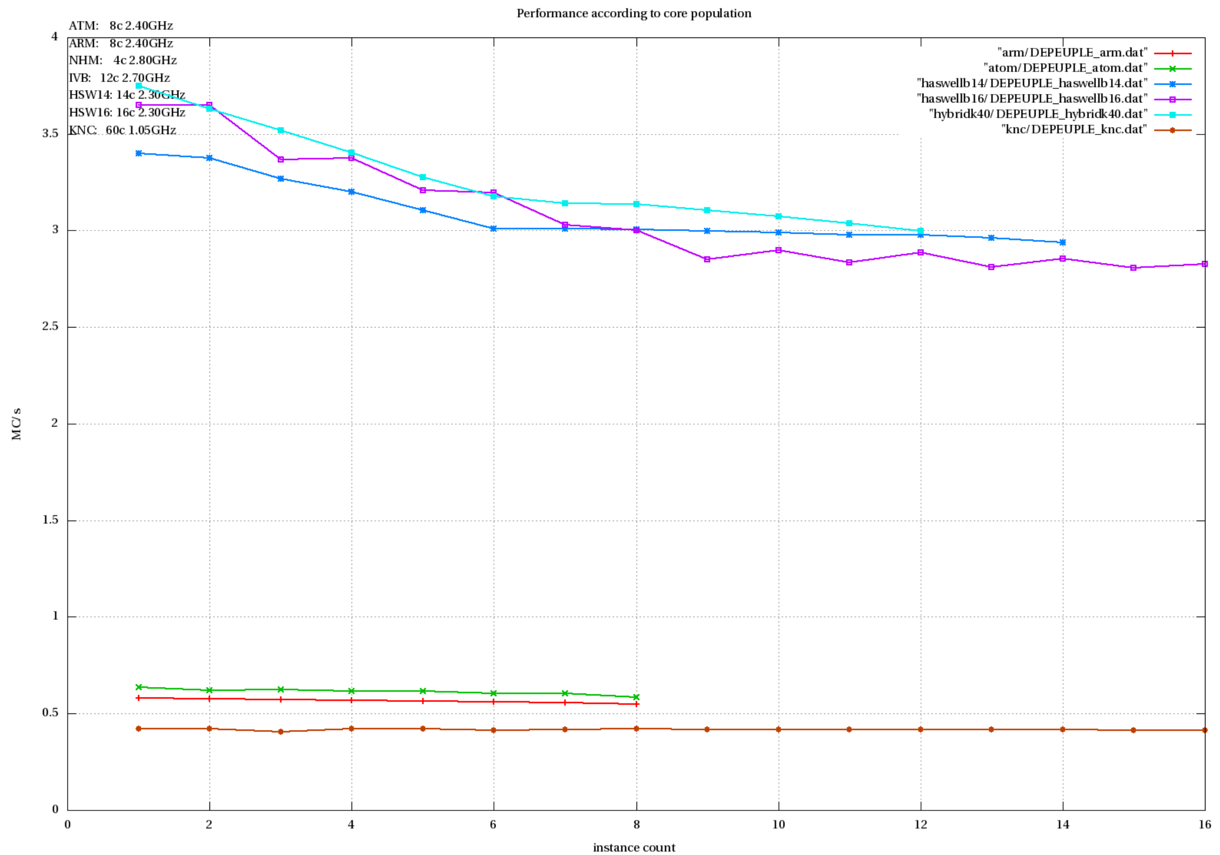
4.2.2 Optimization on KNC has a positive impact on standard Xeon processors

The study made in collaboration with Intel on HydroC [HPPP2014] demonstrates that memory access on the KNC needs to be carefully studied. In particular, it is shown that a sub-decomposition of the computed domain in “tiles” boosts the performances by a factor close to 3 (compare the final lines of HydroC OpenMP and OpenCL samples (Figure 3 and Figure 4). Figure 8 shows that the tile size (hence the memory layout used by the program) has a big impact on the performance of a given test case, emphasizing that memory access pattern is indeed a problem.



• Figure 8 : tile size impact on performances on different architectures using 8 threads in each case.

4.2.3 More studies are needed to understand this architecture



• Figure 9 : study of the memory behavior with an increase load of the compute element

To understand a given processor architecture, many tests are needed. A classical approach to study the memory subsystem is to gradually increase the pressure on the memory access and see how the chip reacts to this pressure. To produce Figure 9, we incrementally replicated a test case (one instance per core) and measure the performance of the runs. What is expected is to see the global memory bandwidth being shared by the instances and thus notice a regular degradation of the performance per instance. It is the case for the regular Xeon (upper curves). The KNC (lower line) has a flat profile telling us that the runs are not affected by the increased memory pressure. More experiments are then required to understand why the Xeon would show saturation and not the KNC. Many reasons are possible. TLB misses, L2 cache issues, impact of the internal topology of the KNC are potential culprits.

As a summary, Figure 7 and Figure 9 exhibit behaviors that obviously need further investigations. The KNC internal architecture is (rather) complex and corner cases are not unexpected. The time allocated to the project and the unavailability of the next generation KNL were not enough to investigate in depth this new exciting architecture. This will be done in the coming months allowing the partners to get ready to study the KNL when it will show up.

5. Abreviations and Acronyms

- **aes**: The Advanced Encryption Standard is a specification for the encryption of electronic data
- **ATM**: Atom ® processor from Intel
- **ARM**: Architecture of processors designed under the license of the ARM company
- **CUDA**: *Compute Unified Device Architecture,* a parallel programming framework by NVIDIA
- **GPU**: Graphic Processing Unit
- **HSW14:** Haswell-EP ® from Intel featuring 14 cores @ 2.3GHz (here a bi-socket)
- **HSW16**: Haswell-EP ® from Intel featuring 16 cores @ 2.3GHz (here a bi-socket)
- **IVB**: Ivy Bridge ® from Intel featuring 12 cores @ 2.7Ghz (here a bi-socket)
- **KNC**: Xeon Phi ® from Intel featuring 60 cores @ 1.05GHz
- **MAQAO**: Modular Assembly Quality Analyzer and Optimizer: Performance analyse tool of UVSQ (open source)
- **MIC=XeonPhi**: Many Integrated Core, Intel's many-core processor architecture
- **NHM**: Nehalem ® from Intel featuring 4 cores @ 2.8GHz (here a bi-socket)
- **OpenCL**: Open Computing Language for heterogeneous highly parallel systems
- **OpenACC**: provides a simple programming model for accelerators that can work in the presence or absence of an accelerator
- **DGEMM** and SGEMM: Double and Simple precision General Matrix Multiply
- **Linpack:** The LINPACK benchmarks appeared initially as part of the LINPACK user's manual. Parallel LINPACK benchmark implementation called HPL (High Performance Linpack) is used to benchmark and rank supercomputers for the TOP500 list.
- **sha-1**: SHA-1 is a cryptographic hash function.

6. Bibliography

- All PERFCLOUD deliverables of Workpackage 5.1.
- The OpenACC Application Programming Interface, 2013
- The OpenCL Specification, 2011. Khronos OpenCL Working Group,
- Documentation Intel composerxe v.2013, 2013. Intel
- High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster, 2010. Dimitri Komatitsch and Gordon Erlebacher and Dominik Goddeke and David Michea.
- One OpenCL to Rule Them All? Romain Dolbeau, François Bodin, Guillaume Colin de Verdière.
- Github https://github.com/HydroBench/Hydro.git
- HPPP2014 Towards an efficient Godunov's scheme on Phi, chap 2; in High Performance Parallelism Pearls, Reinders and Jeffers, isbn-9780128021187, Morgan Kaufmann.