Accelerated keyword

Alessandro Fanfarillo and Damian Rouson

fanfarillo@ing.uniroma2.it damian@sourceryinstitute.org

August 4th, 2015

Introduction (to exascale)

In order to reach challenging performance goals, computer architecture will change significantly in the near future.

The main limitations to performance growth have been identified as:

- energy consumption
- high degree of parallelism
- fault resilience
- memory size and speed

From the point programmer's point of view, dealing with all these factors means writing a more "hardware-aware" code.

Energy consumption and its effects

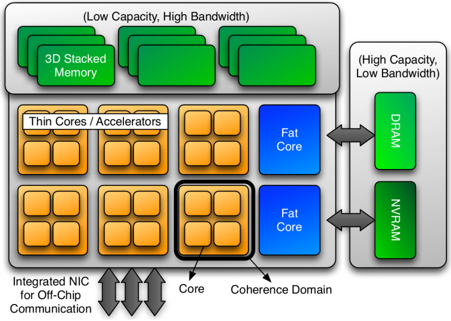
Energy consumption is one of the most important limiting factor for exascale computing.

- Many and slower cores must be used in parallel in order to keep energy consumption low.
- Off-chip data transfers must be reduced as much as possible (energy and technology reasons).
- Memory has to expose enough bandwidth in order to feed all the cores (memory wall).

In order to face all these effects, compute node's architecture will expose heterogeneous processing units, several levels of memory (based on different technologies) and as much integration as possible.

The more you will tell the compiler, the higher performance you will get.

Exascale compute node


(Low Capacity, High Bandwidth)
3D Stacked Memory
(High Capacity, Low Bandwidth)
Thin Cores / Accelerators
Fat Core
Fat Core
DRAM
NVRAM
Integrated NIC for Off-Chip Communication
Core
Coherence Domain

- Fat and thin cores.
- Two kinds of memory (fast and slow).
- Integrated NIC (lower power, higher message throughput).
- Coherence protocol limited to small groups of cores.

Image taken from *Abstract Machine Models and Proxy Architectures for Exascale Computing*

Intel Xeon Phi Knights Landing (KNL)

In 2014, NERSC announced that its next supercomputer, named "Cori", will be a Cray system based on a next-generation Intel MIC architecture; this machine will be a self-hosted architecture, neither a co-processor nor an accelerator.

Features	Edison (Ivy-Bridge)	Cori (Knights-Landing)
Num. physical cores	12 cores per CPU	72 physical cores per CPU
Num. virtual cores	24 virtual cores per CPU	288 virtual cores per CPU
Processor frequency	2.4-3.2 GHz	Much slower than 1 GHz
Num. OPs per cycle	4 double precision	8 double precision
Memory per core	2.5 GB	Less than 0.3 GB of fast memory per core and less than 2 GB of slow memory per core
Memory bandwidth	≈ 100 GB/s	Fast memory has $\approx 5\times$ DDR4

Near memory on KNL

On KNL, the “near” memory (fast, on-package memory), can work in 3 modes:

- Cache mode: the near memory works as a big L3 cache
- Flat mode: the near memory acts as a separate memory. The programmer has to allocate specifically into the fast memory.
- Hybrid mode: Portions of the near memory acts as L3 cache.

The mode is selected at system boot time.

Rumors on Internet tell that Intel is tweaking its compilers so Fortran can allocate into the near memory using the flat addressing mode.

Parallel Programming Models for Exascale

On an exascale machine, programs and programmers will deal with:

- heterogeneous processors
- high degree of parallelism
- unpredictable behaviours (faults and voltage/frequency throttling)

The high dinamicity exposed by such architectures will certainly require new parallel programming models.

PGAS languages are good candidates for dealing with highly dynamic parallel algorithms.

Fortran and Exascale

Fortran 2008 has already two features that can be used for improving performance on an exascale node: DO CONCURRENT and coarrays.

DO CONCURRENT tells the compiler that the loop is safe to parallelize/vectorize. Such statement can also be expressed using directives, defined by OpenMP or by a specific compiler.

On KNL, using vectorization is critical for getting high performance, but an efficient vectorization requires more conditions to be satisfied.

Non-aligned data are more expensive to load than aligned, padding can also play a significant role in getting better performance.

Coarrays provide the PGAS support suitable for exascale nodes.

Accelerated keyword

Providing a way to declare a specific variable as “suitable for acceleration” would certainly help the compiler to set the best conditions for an effective execution.

On KNL, declaring a variable as “accelerated” would tell the compiler to allocate space in the near memory when working in Flat Mode or Hybrid Mode.

Such declaration would also set all the conditions (data alignment, padding, unitstride, etc) for getting the best performance.

Having such expressivity inside the language would also allow to merge the concept of “computational variable” with the existing parallel features like coarrays and do concurrent.

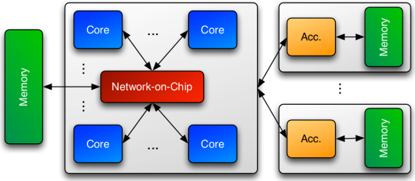
Language attribute vs. directive

Why should I use a language keyword instead of a directive?

There are several good reasons:

- The compiler (which knows the architecture) can pick the right set of optimizations to apply to the "special" variable (think about heterogeneous execution).
- The programmer is relieved from expressing architecture-related concepts.
- A directive could be hard to integrate with language features (coarrays).

Current heterogeneous architecture



Currently, accelerators and CPUs reside on different devices.

Anyway, they have the same memory address space and memory transfers can be performed transparently.

We used such architecture for simulating an exascale node and test the bahaviour of the accelerated keyword combining coarrays and CUDA.

Managed memory and Zero-copy memory in CUDA

Zero-copy memory was introduced in CUDA 2.0.

Such mechanims allows to “pin” some host memory and make it accessible to the GPU directly.

The transfer from/to host/device is started when the memory is accessed.

Managed memory has been introduced in CUDA 6.0.

It allows to allocate a portion of memory completely managed by the CUDA runtime system.

The transfer is totally transparent to the user and happens immediately before the launch and after the kernel termination.

Managed memory is much more complex than zero-copy and it can improve temporal and spatial locality.

Accelerated keyword in GNU Fortran

In order to use coarrays and CUDA we modified GFortran for allocating CUDA memory when a (allocatable) variable is declared as accelerated (passing through OpenCoarrays).

Since OpenCoarrays is based on MPI one-sided, we are implementing a one-sided device-to-device communication using native CUDA features.

Some MPI implementations, like OpenMPI, are CUDA-Aware and can be used for this purpose as well.

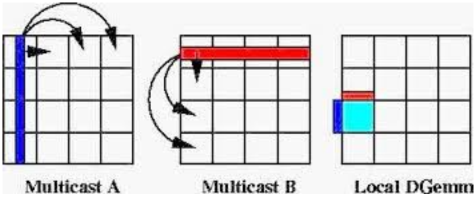
Implementing the “accelerated” keyword in GFortran took me 3 hours.

SUMMA Algorithm

SUMMA stands for Scalable Universal Matrix Multiplication Algorithm.

It is particularly suitable for PGAS languages because of the one-sided nature of the transfers involved.

Instead of performing inner products, SUMMA performs n partial outer products.



With coarrays no multicast is required; each image can get the row/column needed with a one-sided operation.

Machine Description and Test Configuration

We have run the tests on Eurora, a heterogeneous cluster provided by CINECA, equipped with Tesla K20 and Intel Xeon Eight-Core E5-2658.

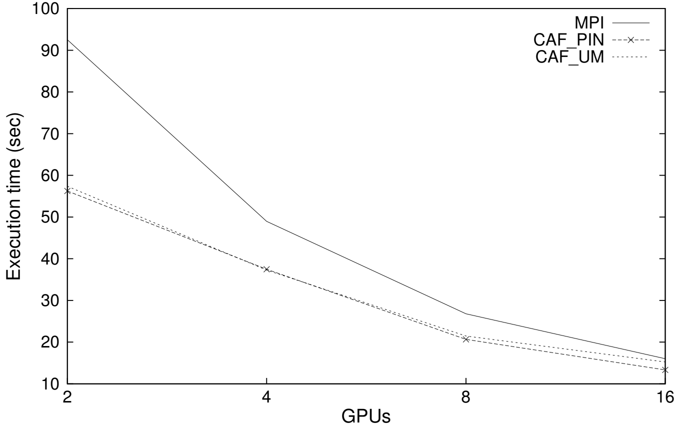
We used the pre-release GCC-6.0, with OpenCoarrays 0.9.0 and IntelMPI-5.

IntelMPI is the best MPI implementation provided on Eurora.

CublasDgemm provided by Nvidia has been used as computational CUDA kernel.

The test has been repeated 10 times using a matrix of size 4096x4096.

Results


100
90
80
70
60
50
40
30
20
10
Execution time (sec)
2
4
8
16
GPUs
MPI
CAF_PIN
CAF_UM

Test Conclusions

The integration of coarrays and accelerators can provide significant speedup.

The current implementation of Managed Memory is not always better than the usual zero-copy approach.

On systems with RDMA protocols (like Cray), Managed Memory does not work well.

Nvidia claims that they will improve Managed Memory in the next CUDA releases.

Conclusions

A language attribute can be useful on any heterogeneous architecture:

- It relieves the programmer from deeply knowing the architecture.
- It allows to run the code in the most efficient way on different devices (accelerated on CPU may have a different setting than on Intel MIC).
- It does not substitute directives, it can live with them without interfering.
- It is easy to implement for compiler vendors.

Thanks

Suggestions/Feedback/Questions?