Portability and Platform-Specific Work

Chapel Team, Cray Inc.

Chapel version 1.13

April 7, 2016

Safe Harbor Statement

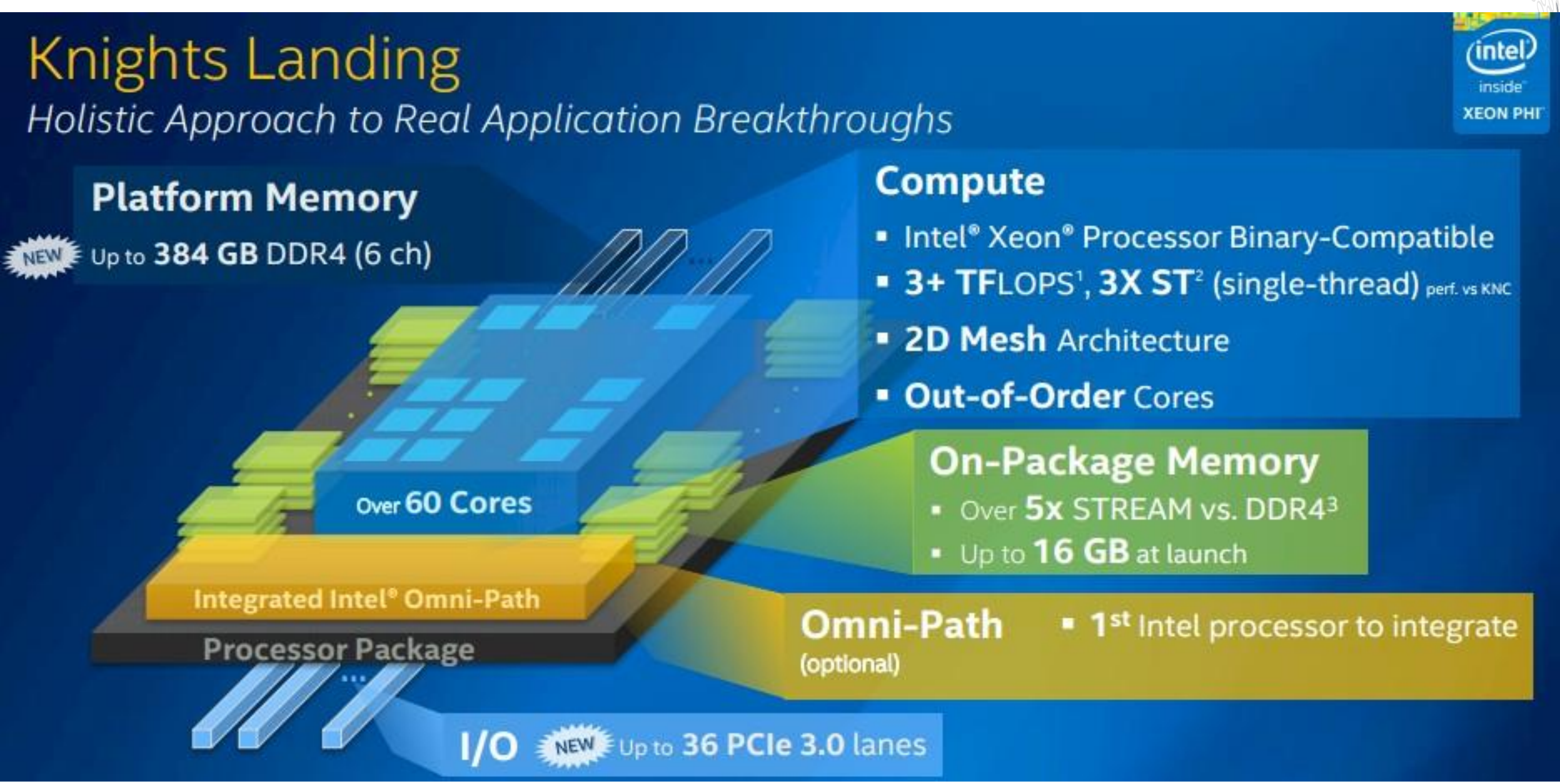
This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Outline

- **Chapel on Intel® Xeon Phi™ "Knights Landing" (KNL)**
  - Chapel Support for KNL High-Bandwidth Memory (HBM)
- **Unification of Cray (CCE) Configuration**
- **Python 2/3 Compatibility**
- **Other Portability Work**

Chapel on Intel® Xeon Phi™ "Knights Landing" (KNL)

Chapel on KNL: Background



Source: [KNL HBM Usage Overview]

Chapel on KNL: Background

- **Previous releases included Knights Corner (KNC) support**

- **Targeting KNC with Chapel has several limitations**
  - requires a special build of the module
  - only supports Intel as the target compiler
  - all communication has to be bounced through the host processor
    - cannot use "native" communication layers (e.g., ugni/gasnet-aries)
  - limited amount of memory

- **KNC was a good starting point for becoming "KNL-ready"**
  - allowed us to explore running on a many-core architecture
    - 60+ cores (4 threads per core)
      - each core is relatively slow compared to a traditional Xeon
    - wide (512 bit) vector units

Chapel on KNL: Background

- **Targeting KNL addresses several KNC-specific limitations:**
  - binary-compatible with Xeon Processors
  - support from Intel, GNU, and Cray target compilers
  - can run as a self-hosted processor
  - can support native communication layers
  - substantially higher memory capacity

- **That said, KNL remains a unique many-core architecture**
  - 60+ cores (with 4 threads per core)
  - support for AVX-512 (wide vector units)
  - high-performance on-package memory (MCDRAM)
    - a.k.a. high-bandwidth memory (HBM)
  - several "clustering" modes available
    - all-to-all, quadrant, sub-numa
  - for more information, see: https://software.intel.com/en-us/articles/what-disclosures-has-intel-made-about-knights-landing

Chapel on KNL: Background

- **Several KNL-based systems will be coming online soon**
  - e.g., Cori and Trinity: two large systems being manufactured by Cray
    - each system will have over 9,000 self-hosted KNL nodes
    - will be running a new version of Cray's OS and system management stack

- **Main KNL goal for 1.13 was to simply get up and running**
  - intentionally modest, due to limited access to hardware/systems

- **Secondary goal: start looking at performance**
  - improve multi-threaded performance using Qthreads tasking
  - add a locale model that supports targeting HBM
    - ongoing work, reported on in following section
  - start exploring vectorization performance

Chapel on KNL: This Effort

- **Default to qthreads tasking on KNC**
  - previously defaulted to fifo
    - got poor performance with qthreads in early days of KNC work
  - subsequent work with qthreads has resulted in better performance
    - using different scheduler, affinity options, etc.
  - had to disable guard pages for a few tests
    - concurrent calls to mprotect() appear to be expensive on Xeon Phi
  - stream performance is now on par with reference for KNC
    - early results on KNL also look promising

- **Numerous bug fixes for CLE 6.X and KNL**
  - primarily related to differences with new Cray OS
    - updates to ugni comm layer memory registration
    - fixes to hwloc for /proc/mounts overflow bug
    - updates to module build process
    - several miscellaneous bug fixes

Chapel on KNL: Impact

- **Achieving good multi-threaded performance on KNC**
  - and on KNL for the limited testing we've had the chance to do

- **Correctness test suite passing for all KNL configurations**
  - matrix of nightly testing over comm/tasking layers vs. target compilers:

Configuration Matrix	cray	intel	gnu
none			
gasnet-aries			
gasnet-mpi			
ugni-muxed			
ugni-qthreads			

(blue means "clean run" in our test system)

Chapel on KNL: Next Steps

- **Retire Chapel's support for KNC**
  - was a useful stepping-stone towards KNL
  - we believe it is no longer important to Chapel's end-users
    - please let us know if we're mistaken!

- **Add KNL-specific locale model to support targeting HBM**

- **Verify that Qthreads provides good performance on KNL**
  - and see if guard page overhead can be lowered

- **Work on vectorization improvements/optimizations**
  - and explore any potential KNL-specific optimizations

Chapel Support for KNL High-Bandwidth Memory (HBM) (an ongoing effort)

KNL HBM: Background

- **KNL supports two memory systems:**
  - off-package DDR memory
    - 90+ GB/s and up to 384 GB
  - high-performance on-package memory (a.k.a. MCDRAM or HBM)
    - 400+ GB/s and up to 16 GB
- **MCDRAM can operate in different modes:**
  - partly or entirely as a cache to system memory
  - allocating all data in the MCDRAM
  - as a memory that is independent from system memory
    - programs use it by allocating explicitly from the MCDRAM
- **KNL will support dividing into 4 distinct NUMA domains**

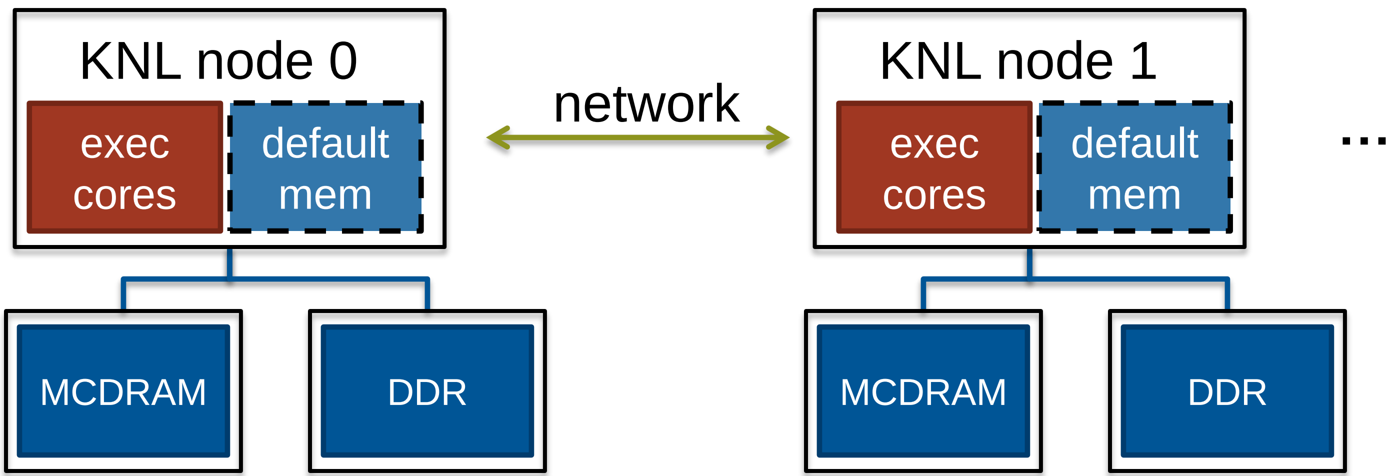
Sources: Intel KNL Disclosures KNL HBM Usage Overview

KNL HBM: Representing Using Locales

- **Represent MCDRAM as part of a Chapel locale model**
  - our first example of a *memory-only* sub-locale
  - such locales can be used like any other:
    - you can target them using on-statements
      - to store data
      - to execute tasks
    - you can store them in an array and target them with distributions
- **The following slides propose our default policy for KNL**
  - since it's part of a locale model, none of this is baked into the compiler
    - i.e., if you want a different KNL policy, you can create one
    - (or you can advocate for changes to ours)
- **This is all work-in-progress**

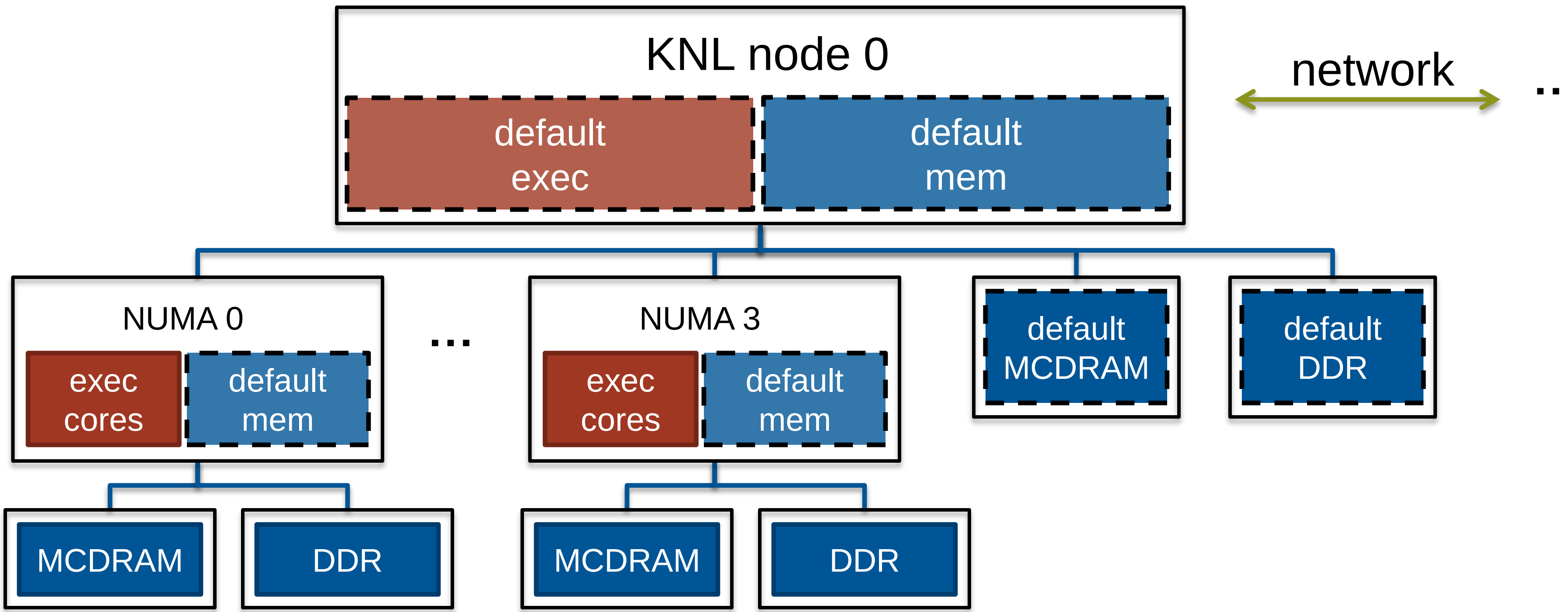
KNL HBM: Proposed Policy

- **Tasks on *memory locales* execute on the parent**
  - but with a constraint: any allocations are from the designated memory
- **Tasks on top-level KNL locales use a default child memory**



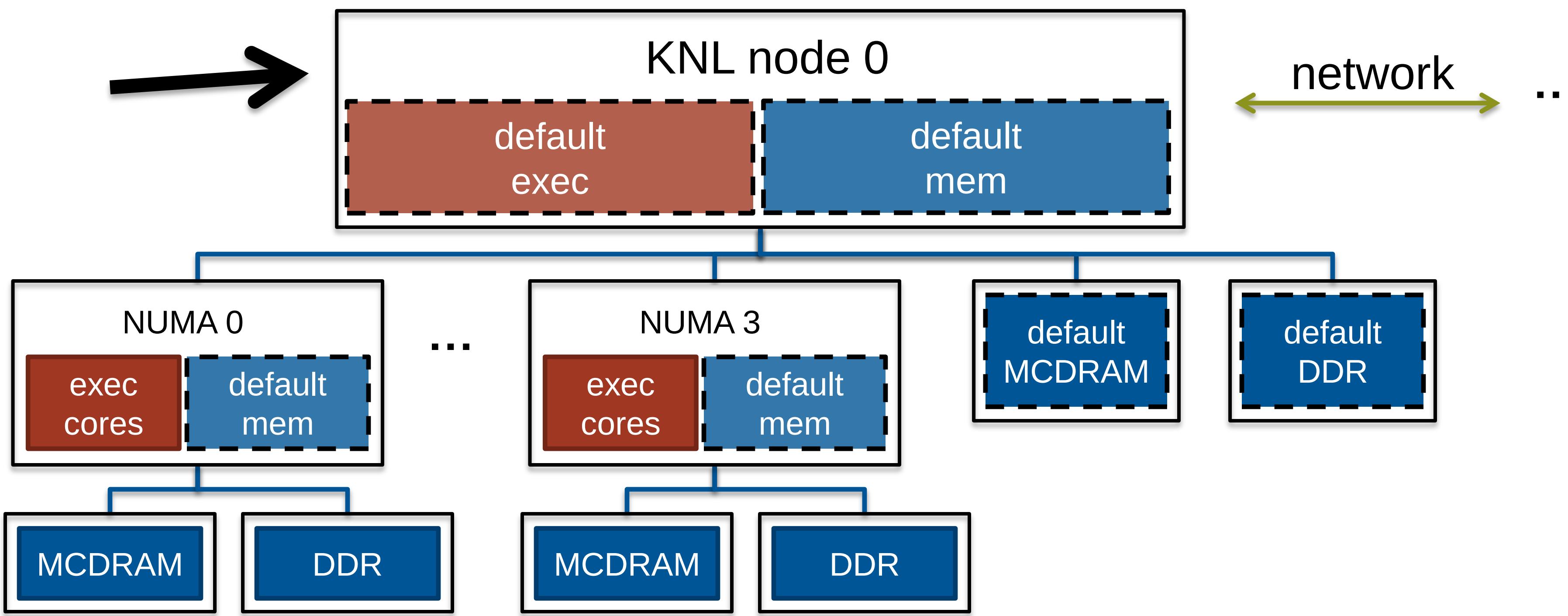
KNL HBM: Representing NUMA Locales

- **Challenge: How to represent distinct NUMA domains?**
  - give memory sub-locales to each top-level KNL locale, as before
  - also give each KNL locale multiple NUMA sub-locales
  - give each NUMA sub-locale its own memory sub-locales



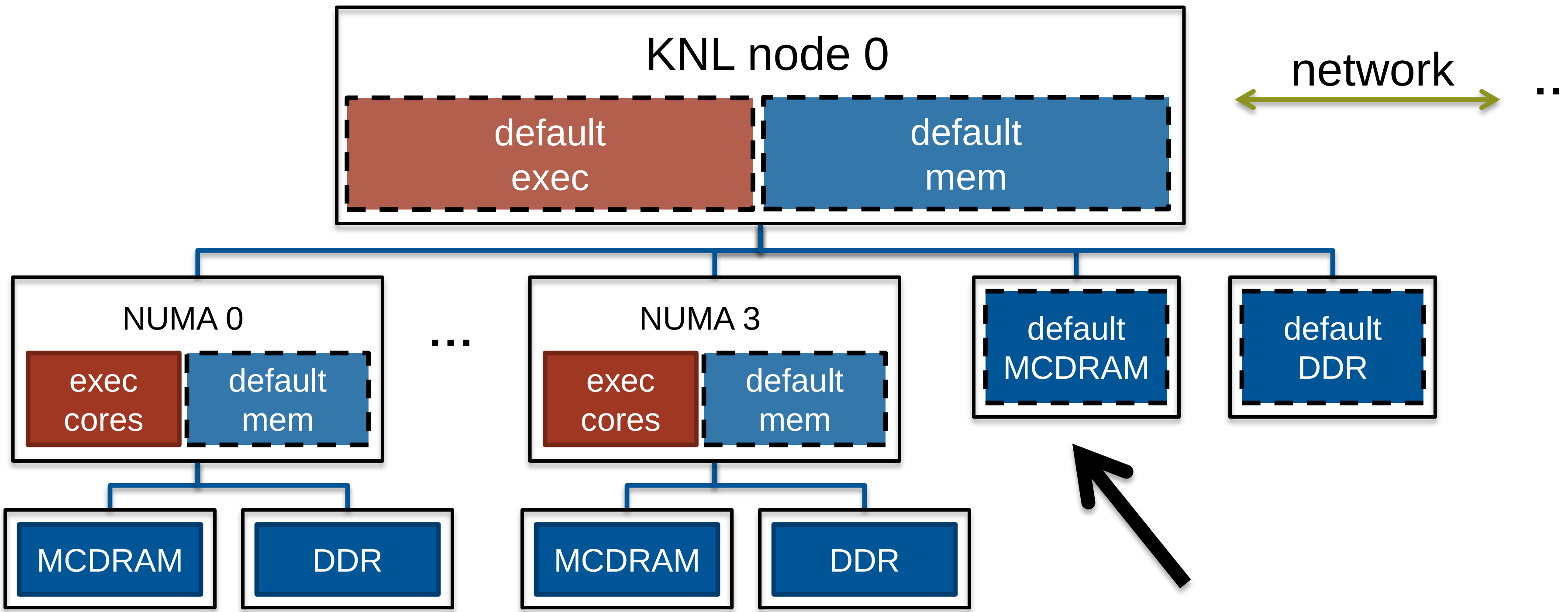
KNL HBM: NUMA Locales

- **Executing on a top-level KNL locale will:**
  - choose among the NUMA sub-locales for where to run the task (as in the NUMA locale model)
  - use a default memory type for its allocations



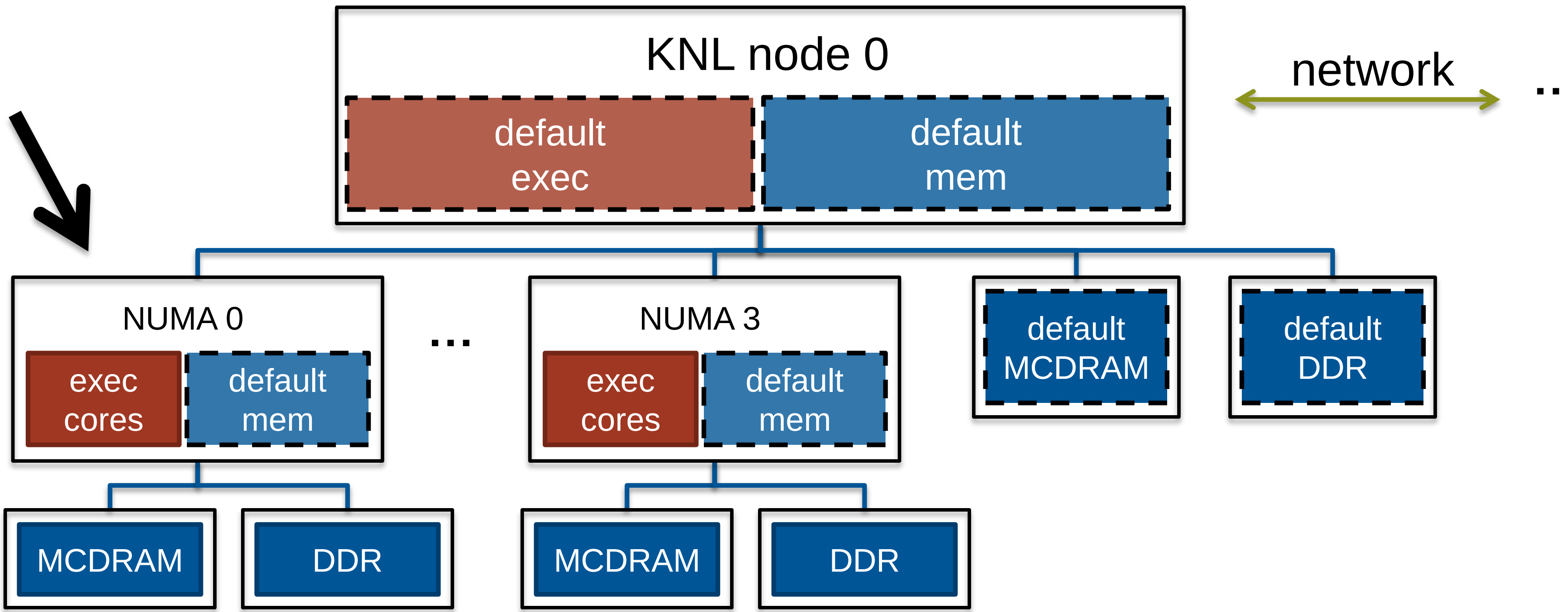
KNL HBM: NUMA Locales

- **Executing on a KNL locale's memory sub-locale will:**
  - choose among the NUMA sub-locales for where to run the task (as in the previous case)
  - use the specified memory type for its allocations



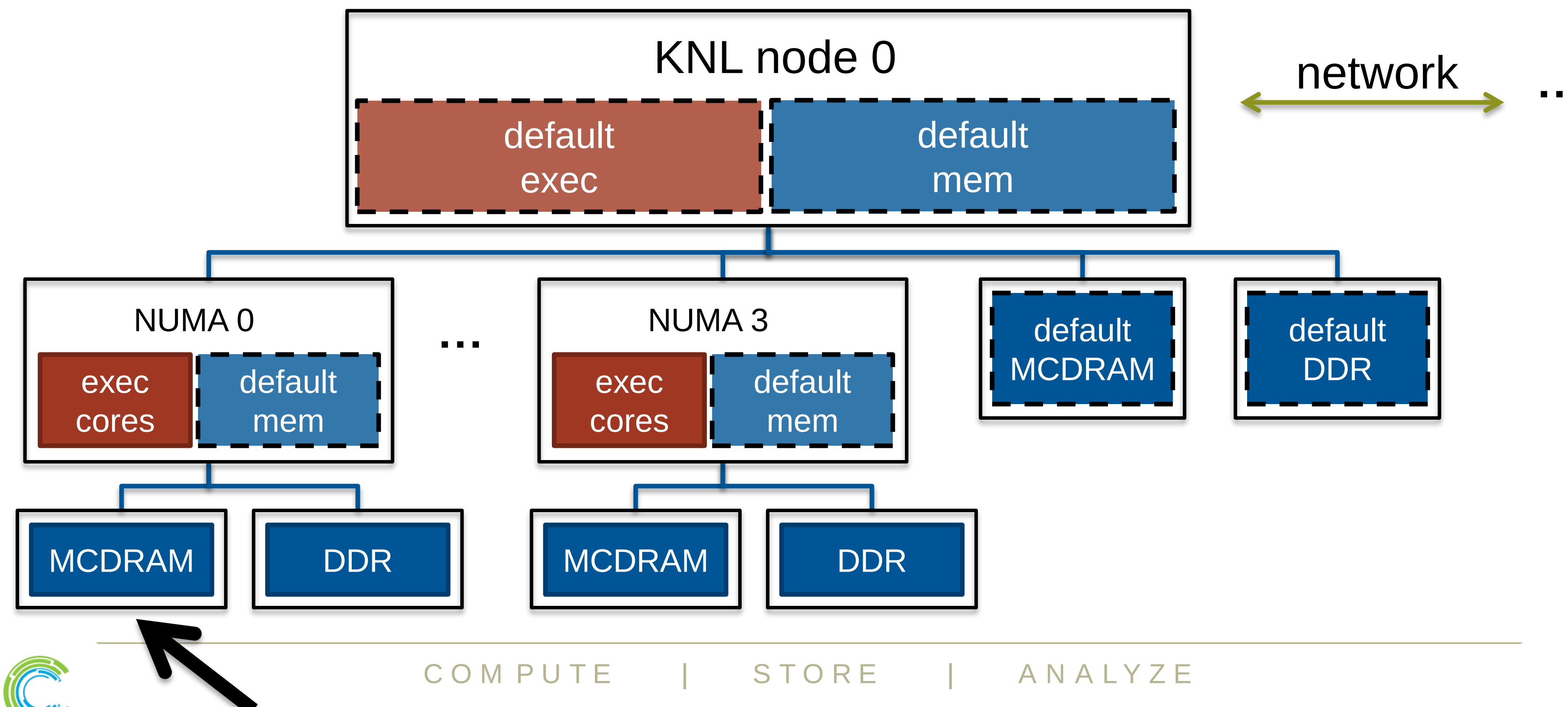
KNL HBM: NUMA Locales

- **Executing on a KNL locale's NUMA sub-locale will:**
  - run the task on one of the cores within the NUMA domain
    - and limit child tasks to those same cores, unless an on-clause is used
  - use a default memory type for its allocations



KNL HBM: NUMA Locales

- **Executing on a NUMA domain's memory sub-locale will:**
  - run the task on one of the cores within the NUMA domain
    - and limit child tasks to those same cores, unless an on-clause is used
  - use the specified memory type for allocations



KNL HBM: Mapping forall loops to KNL

- **Memory locales are locales like any other**
  - Yet, we don't want forall loops to run tasks on memory sub-locales
- **Default foralls currently use simple locale model methods:**

```
proc getChildCount(): int

proc getChild(idx:int) : locale
```

- **We'll need to update this interface to avoid the memories**

KNL HBM: Allocation Experiments

- **Developed prototype KNL locale model**
  - targeted *hbw_malloc* functions provided by memkind library
- **Demonstrated allocating explicitly as follows:**

```
// get HBM with a temporary interface: LocaleModel.hbm
var hbm = (here:LocaleModel).hbm;

on hbm {
  // demonstrated this array is allocated with hbw_malloc
  var A: [1..1000] int;
  A = 1;
}
```

- **In-progress: Support for Block distribution over HBM**

KNL HBM: Impact, Status, and Next Steps

Impact:

- Well on our way to MCDRAM support for KNL
  - Chapel programs will be able to explicitly program the memory
  - Arrays will be able to be distributed over many MCDRAMs in a system

Status:

- early prototype demonstrated (not included in v1.13)

Next Steps:

- evolve Locale interface to support HBM
- support capability-oriented interfaces for users for portability, e.g.:
  - “get locale ancestor running the operating system”
  - “get a fast-memory sub-locale”
- complete implementation of KNL Locale Model
- investigate NUMA support on KNL
- investigate performance

Unification of Cray (CCE) Configuration

CCE Target Compiler: Background

- **On Crays we support Cray, GNU, and Intel target compilers**

- **Cray (CCE) hasn't supported all runtime configurations**
  - used fifo tasking — couldn't build qthreads or muxed (inline ASM)
  - used gasnet comm — couldn't build ugni (missing atomics)
  - used dlmalloc memory — couldn't build tcmalloc (C++, compiler flags, etc)
  - lacked regexp support — couldn't build re2 (Makefiles used GNU flags)

- **As a result, using the CCE back-end limited performance**
  - and did not support regular expressions

- **CCE 8.4.0 allows us to improve on this by adding…**

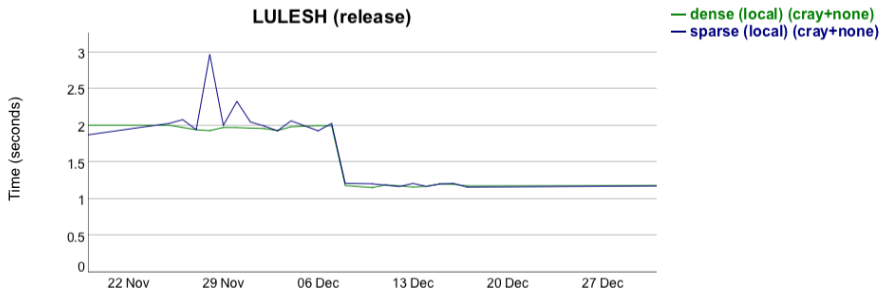
  ... inline ASM support
  …additional GNU compatability
  ...missing atomics

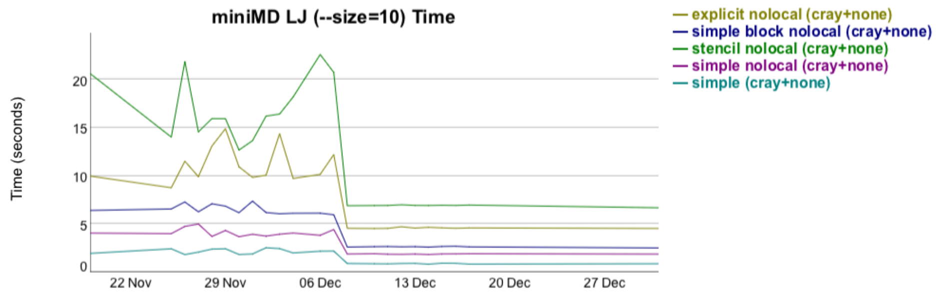
CCE Target Compiler: This Effort

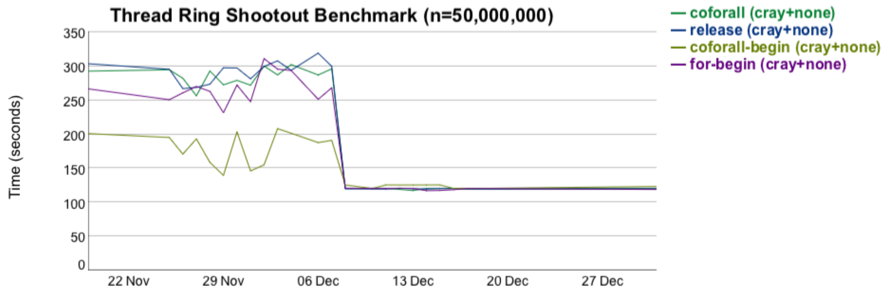
- **Build qthreads+hwloc with CCE**
  - required working around broken ffs() for hwloc
  - required CCE 8.4.0 for Qthreads (inline ASM)
- **Build muxed with CCE**
  - required CCE 8.4.0 (inline ASM)
- **Build jemalloc with CCE**
  - required building with –hipa2 to work around inlining bug
  - had to avoid throwing GNU-specific flags
- **Build ugni with CCE**
  - required building with –hipa0 to work around inlining bug
  - required jemalloc, since getting tcmalloc to build proved impractical
- **Build re2 with CCE**
  - edited Makefiles to avoid GNU-specific flags

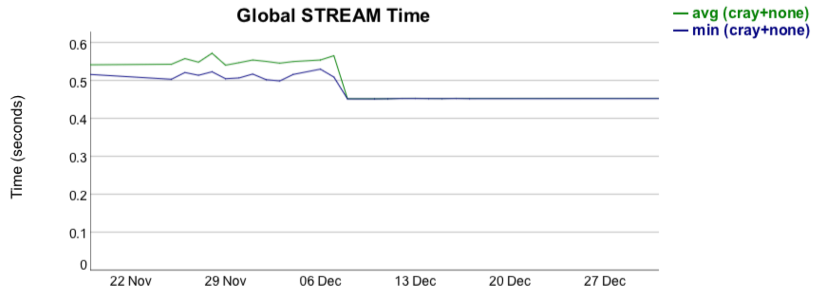
CCE Target Compiler: Impact

- **CCE back-end now supports the full runtime**
  - significantly better performance due to qthreads+hwloc, jemalloc, ugni
  - regular expression support









Python 2/3 Compatibility

Python 2/3 Compatibility: Background

Background:

- Building Chapel required Python 2.x, but Python 3 is used widely
- Had become our most frequent barrier to successful end-user installs

This Effort:

- Modified Chapel configuration scripts to support Python 3.x as well

Impact:

- Users can now build Chapel with Python 2 or 3

Status:

- Nightly testing checks compatibility with Python 2.6, 2.7, and 3.5

Next Steps:

- Make the testing infrastructure Python 3-compatible
  - e.g., `start_test`

Other Portability Work

Other Portability Work

- **Switched PGI C++ compiler from pgCC to pgc++**
- **Worked around code base incompatibilities with gcc 5.1**
  - seemingly a gcc 5.1 compiler bug, as unlikely as that seems…
- **Fixed portability issues to newer Cygwin installs**
- **Removed stale Xcode support**

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.

CRAY CHAPEL

CRAY

THE SUPERCOMPUTER COMPANY