Locale Models

Chapel Team, Cray Inc.

Chapel version 1.16

October 5, 2017

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

- APU Locale Model
- Locale Model deinit()
- Numa Locale Model

APU Locale Model
(contributed by Mike Chu, AMD)

APU Locale Model

Background:

- APU: Accelerated Processing Unit
  - AMD term for co-packaged CPU and GPU
- GPU can access CPU's virtual memory through MMU and cache
  - allows shared pointers
  - avoids copying data
- AMD is interested in having Chapel support APUs

This Effort:

- Added an APU locale model, consistent with other locale models

APU Locale Model

Impact:

- Foundation of APU support is in place

Status:

- Tested at AMD
- The first of a series of contributions to support APUs
  - needs runtime additions to interface with Radeon Open Compute Platform
  - needs compiler changes to generate GPU executable code
  - both additions function in AMD fork, currently working on merging to master

Next Steps:

- Incorporate AMD runtime and compiler changes
- Establish regular testing
- Consider GPU-related language extensions (CHIP 22)
  - allow users to specify the natural width of a forall loop

Locale Model deinit()

Locale Model deinit()

Background:

- Locale model data structures were not torn down upon Chapel exit
  - existing locale models had no special clean-up requirements
- Some GPUs require tear-down to leave hardware in a sane state

This Effort:

- Started deleting locale model data structures on exit
  - added deinit() to existing cases (flat, numa, knl)

Impact:

- Supplies a hook for any special hardware actions at exit
- Used by the APU locale model
  - provides a place to reset the GPU hardware

Next steps:

- Ensure that any future locale models supply deinit()

Numa Locale Model

Numa: Background

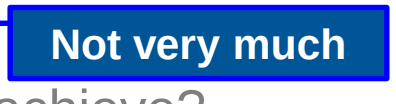
- **Introduced ‘multi-ddata’ arrays for locModel=numa in 1.15**
  - Achieved perfect affinity (i.e. matched execution locality) for arrays
  - Performance was great for some iteration styles
  - Horrible for others
- **Not addressed: pre-existing locality**
  - Memory with undesirable NUMA localization before being allocated (previously allocated and thus localized, then freed, then re-allocated)
  - Only option seems to be migration (copying) to desired NUMA domain
- **Prompted a step back to review options and paths forward**

Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?**
- **If retaining multi-ddata:**
  - reduce array access overhead; how much can we achieve?
  - provide for programmer specification re: single- vs. multi-ddata
- **If reverting to single-ddata:**
  - how (whether) to handle pre-existing locality?
  - how to handle NIC-registered heaps?
- **Improve NUMA-aware memory allocation**
  - do more sublocale-aware allocation (task stacks, etc.)
  - reduce migration by allocating memory with proper locality

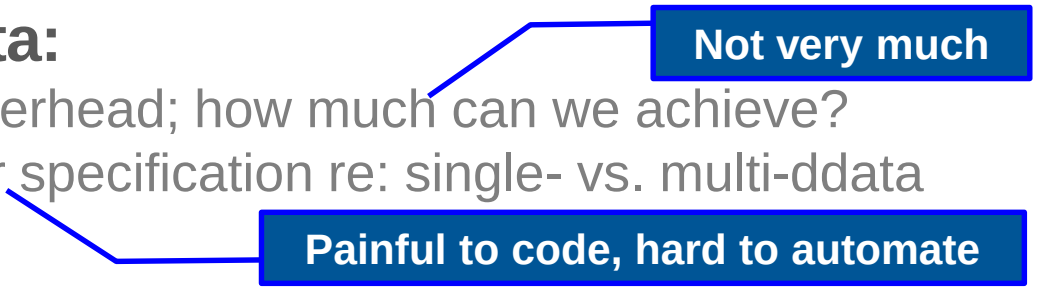
Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?**
- **If retaining multi-ddata:**
  - ✗ reduce array access overhead; how much can we achieve?
  - provide for programmer specification re: single- vs. multi-ddata
- **If reverting to single-ddata:**
  - how (whether) to handle pre-existing locality?
  - how to handle NIC-registered heaps?
- **Improve NUMA-aware memory allocation**
  - do more sublocale-aware allocation (task stacks, etc.)
  - reduce migration by allocating memory with proper locality



Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?**
- **If retaining multi-ddata:**
  - ✗ reduce array access overhead; how much can we achieve?
  - ✗ provide for programmer specification re: single- vs. multi-ddata
- **If reverting to single-ddata:**
  - how (whether) to handle pre-existing locality?
  - how to handle NIC-registered heaps?
- **Improve NUMA-aware memory allocation**
  - do more sublocale-aware allocation (task stacks, etc.)
  - reduce migration by allocating memory with proper locality



Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?** no
- **If retaining multi-ddata:**
  - ✗ reduce array access overhead; how much can we achieve?
  - ✗ provide for programmer specification re: single- vs. multi-ddata

Not very much

Painful to code, hard to automate

- **If reverting to single-ddata:**
  - how (whether) to handle pre-existing locality?
  - how to handle NIC-registered heaps?
- **Improve NUMA-aware memory allocation**
  - do more sublocale-aware allocation (task stacks, etc.)
  - reduce migration by allocating memory with proper locality

Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?** no
- ~~**If retaining multi-ddata:**~~
  - ✗ reduce array access overhead; how much can we achieve?
  - ✗ provide for programmer specification re: single- vs. multi-ddata

Not very much

Painful to code, hard to automate

- **If reverting to single-ddata:**
  - how (whether) to handle pre-existing locality?
  - how to handle NIC-registered heaps?
- **Improve NUMA-aware memory allocation**
  - do more sublocale-aware allocation (task stacks, etc.)
  - reduce migration by allocating memory with proper locality

Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?** no
- ~~**If retaining multi-ddata:**~~
  - ✗ reduce array access overhead; how much can we achieve?
  - ✗ provide for programmer specification re: single- vs. multi-ddata
- **If reverting to single-ddata:**
  - ? how (whether) to handle pre-existing locality?
  - how to handle NIC-registered heaps?
- **Improve NUMA-aware memory allocation**
  - do more sublocale-aware allocation (task stacks, etc.)
  - reduce migration by allocating memory with proper locality

Not very much

Painful to code, hard to automate

Don't know

Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?** no
- ~~**If retaining multi-ddata:**~~
  - ✗ reduce array access overhead; how much can we achieve? — **Not very much**
  - ✗ provide for programmer specification re: single- vs. multi-ddata — **Painful to code, hard to automate**
- **If reverting to single-ddata:**
  - ? how (whether) to handle pre-existing locality? — **Don't know**
  - ✓ how to handle NIC-registered heaps? — **Dynamic registration**
- **Improve NUMA-aware memory allocation**
  - do more sublocale-aware allocation (task stacks, etc.)
  - reduce migration by allocating memory with proper locality

Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?** no
- ~~**If retaining multi-ddata:**~~
  - ✗ reduce array access overhead; how much can we achieve?
  - ✗ provide for programmer specification re: single- vs. multi-ddata
- **If reverting to single-ddata:**
  - ? how (whether) to handle pre-existing locality?
  - ✓ how to handle NIC-registered heaps?
- **Improve NUMA-aware memory allocation**
  - ✓ do more sublocale-aware allocation (task stacks, etc.)
  - reduce migration by allocating memory with proper locality

Not very much

Painful to code, hard to automate

Don't know

Dynamic registration

Extendable heap

Numa: This Effort (quoting 1.15 'Next Steps')

- **Major 1.16 decision: stay with multi-ddata?** no
- ~~**If retaining multi-ddata:**~~
  - ✗ reduce array access overhead; how much can we achieve?
  - ✗ provide for programmer specification re: single- vs. multi-ddata
- **If reverting to single-ddata:**
  - ? how (whether) to handle pre-existing locality?
  - ✓ how to handle NIC-registered heaps?
- **Improve NUMA-aware memory allocation**
  - ✓ do more sublocale-aware allocation (task stacks, etc.)
  - ? reduce migration by allocating memory with proper locality

Not very much

Painful to code, hard to automate

Don't know

Dynamic registration

Extendable heap

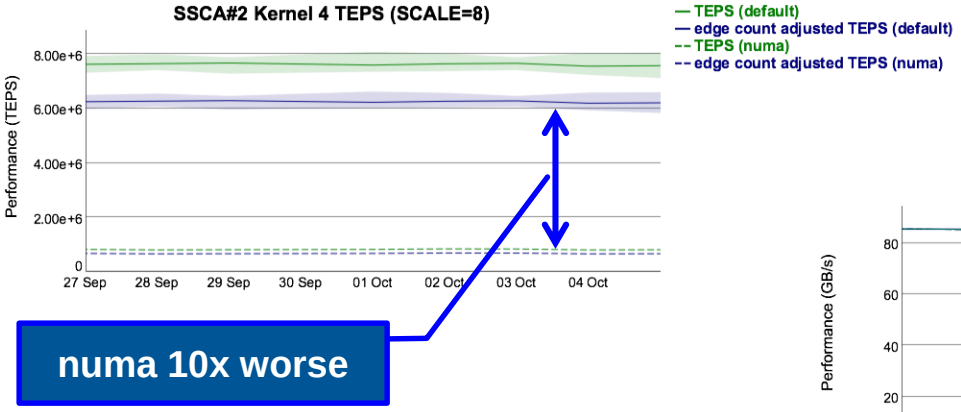
Not needed?

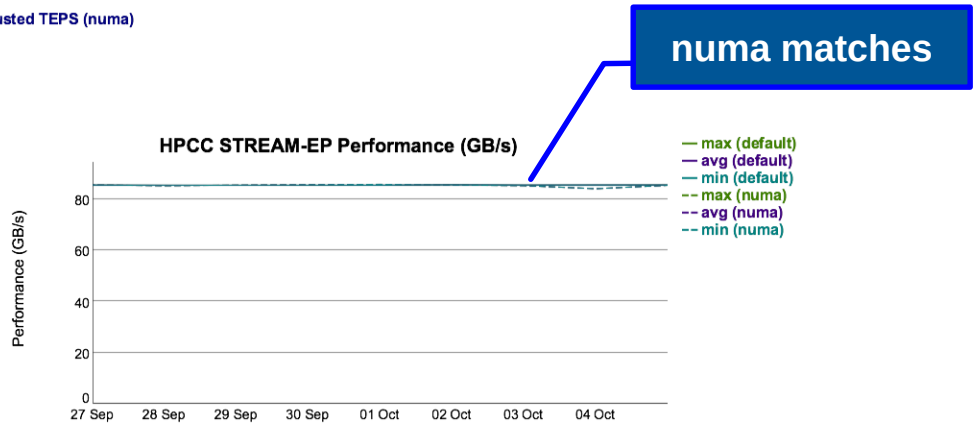
Numa: Status

- **Numa locale model was on hold for this release cycle**
- **Focused on single-ddata improvements**
- **Dynamic NIC registration helps flat LM as much as numa**
  - See ugni comm layer slides in Runtime/Third Party deck for details

Numa: Next Steps

- **Can we get rid of the numa locale model altogether?**
  - No; sometimes we want architectural NUMA domains to be explicit
  - But performance needs to be nearly on par with flat, and often is not
    - Due to wide pointer overhead, Qthreads scheduler issues, likely others





- **Explore:**
  - Locality queries in place of wide pointers
  - Qthreads scheduler improvements
  - Other things?

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

THE SUPERCOMPUTER COMPANY