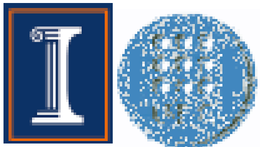
CIGAR: Application Partitioning for a CPU/Coprocessor Architecture

John H. Kelm[*] , Isaac Gelado[†] , Mark J. Murphy[*] , Nacho Navarro[†] ,
Steve Lumetta[*] , Wen-mei Hwu[*]

*Center for Reliable and
High-Performance Computing
University of Illinois at Urbana-Champaign

† Computer Architecture Department
Universitat Politecnica de Catalunya (UPC)



Application Partitioning for a CPU/coprocessor Architecture

Sequential Code

Determine Application Partitioning

Data Layout and Communication Design

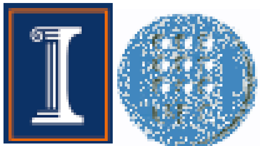
Application Modification

Coprocessor Development

Introduction

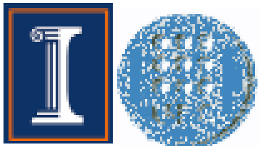
- Context for CIGAR
  - Applications with data parallelism prevalent
  - Opportunities for data-parallel coprocessors
  - Partitioning sequential apps/data is difficult
- The CIGAR three-tiered (holistic) approach:

1. **Extend the machine model**
2. **Provide a partitioning Methodology**
3. **Allow fast Prototyping**



Outline

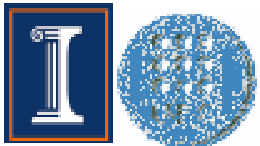
- Introduction
- Coprocessor landscape and current issues
- **Extend:** Data structure hosting and CUBA
- **Methodology:** CIGAR, a methodology for mapping applications into CUBA
- **Prototyping:** Rapid debug/development platform
- Conclusions



CPU/Coprocessor Design Issues Today

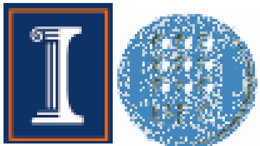
- Why **extend** the machine model?
  - High communication overhead
- Why develop a **methodology?**
  - Developers rewrite/modify sequential apps
  - Inconsistencies between code and design process:

    CPU-only vs. CPU/coprocessor code
  - Discovering code/data to map to coprocessor difficult
- Why build a **prototyping** platform?
  - Late evaluation of ***correctness***
  - Visibility and speed for debugging



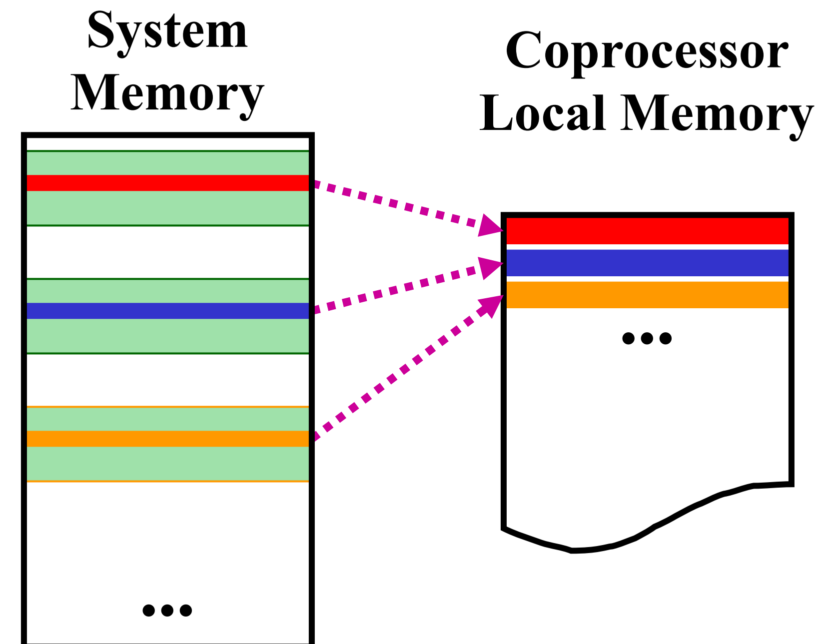
Coprocessor Taxonomy

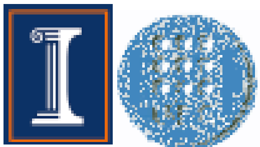
	Fine-grained Accelerators	**Coarse-grained Coprocessors**
Input/Output:	Registers, small datasets, data not persistent	DMA/MMIO, Large, persistent datasets
Execution:	Primitive operations (e.g., SIMD FMAC)	Complex functions (e.g., Motion Estimation)
Location:	Embedded in CPUs	External device
Examples:	x87 (FPU), Intel's MMX (multimedia), SSE*x* (SIMD), Stretch (reconfigurable)	Cray XD1 (FPGA), NVIDIA G80 (GPU), Ageia's PhysX (Physics Processor)



Data Transmission

- Communication cost critical
- Data marshalling:
  1. Select
  2. Aggregate
  3. Transfer
- Data layout
- Asymmetric access latency
- **Goals**:
  - Remove need to marshal data
  - Low-latency access for both CPU and coprocessor





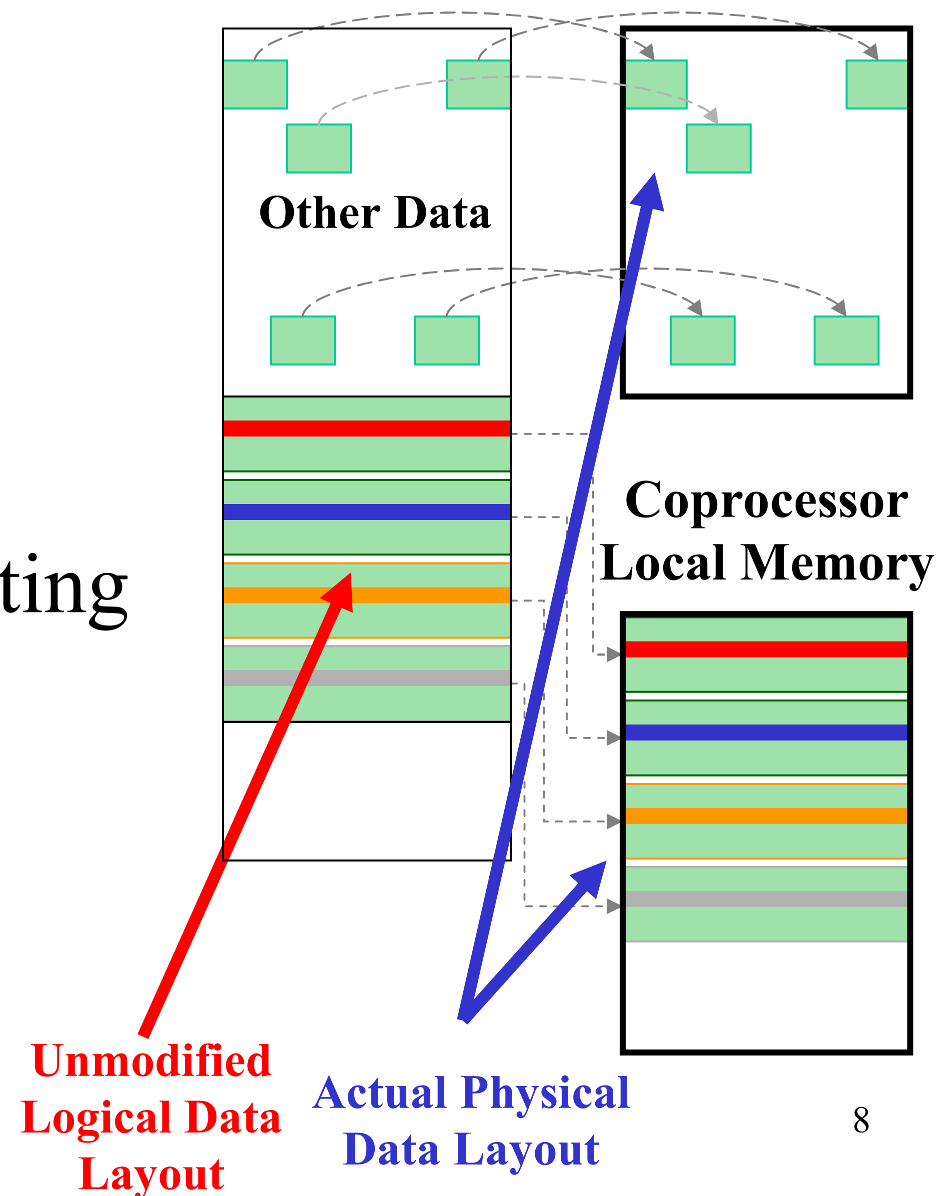
Data Structure Hosting

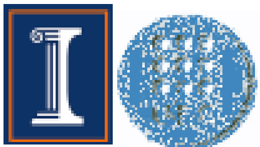
Problem: Want to Avoid…

- Marshalling data
- Changing code
- Remapping pointers
- Modifying data layout

Solution: Data Structure Hosting

- Only one (persistent) copy
- Allocate whole structure in coprocessor local memory
- Same data layout for app

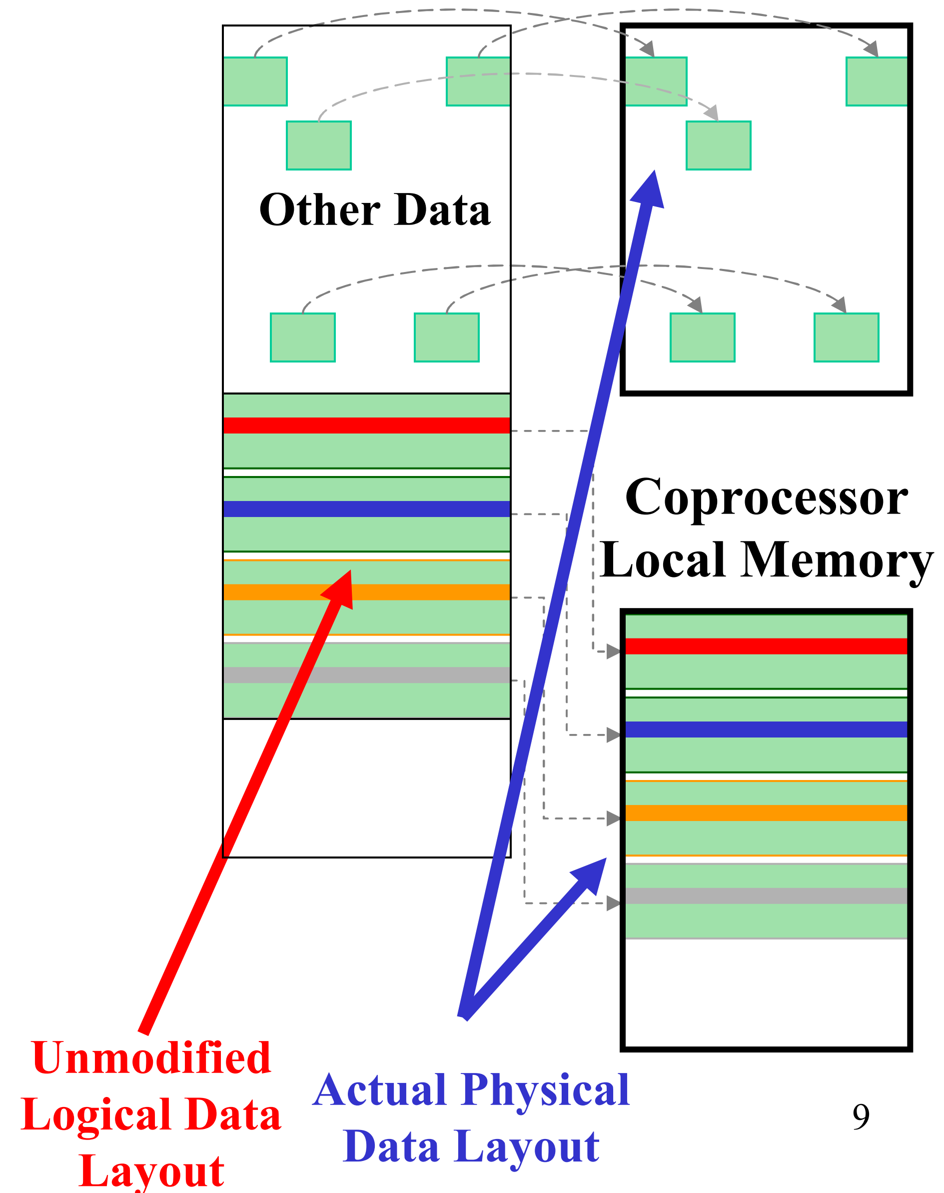


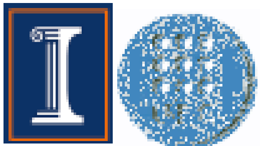


Data Structure Hosting

- Consistent view of data structures
- Not a replacement programming model
- Enables low-latency access for CPU and coprocessor

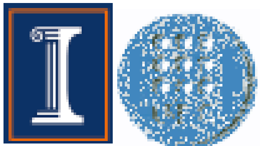
- **Tradeoff**: Simpler, consistent machine abstraction **vs.** memory efficiency





Parameter Passing Semantics

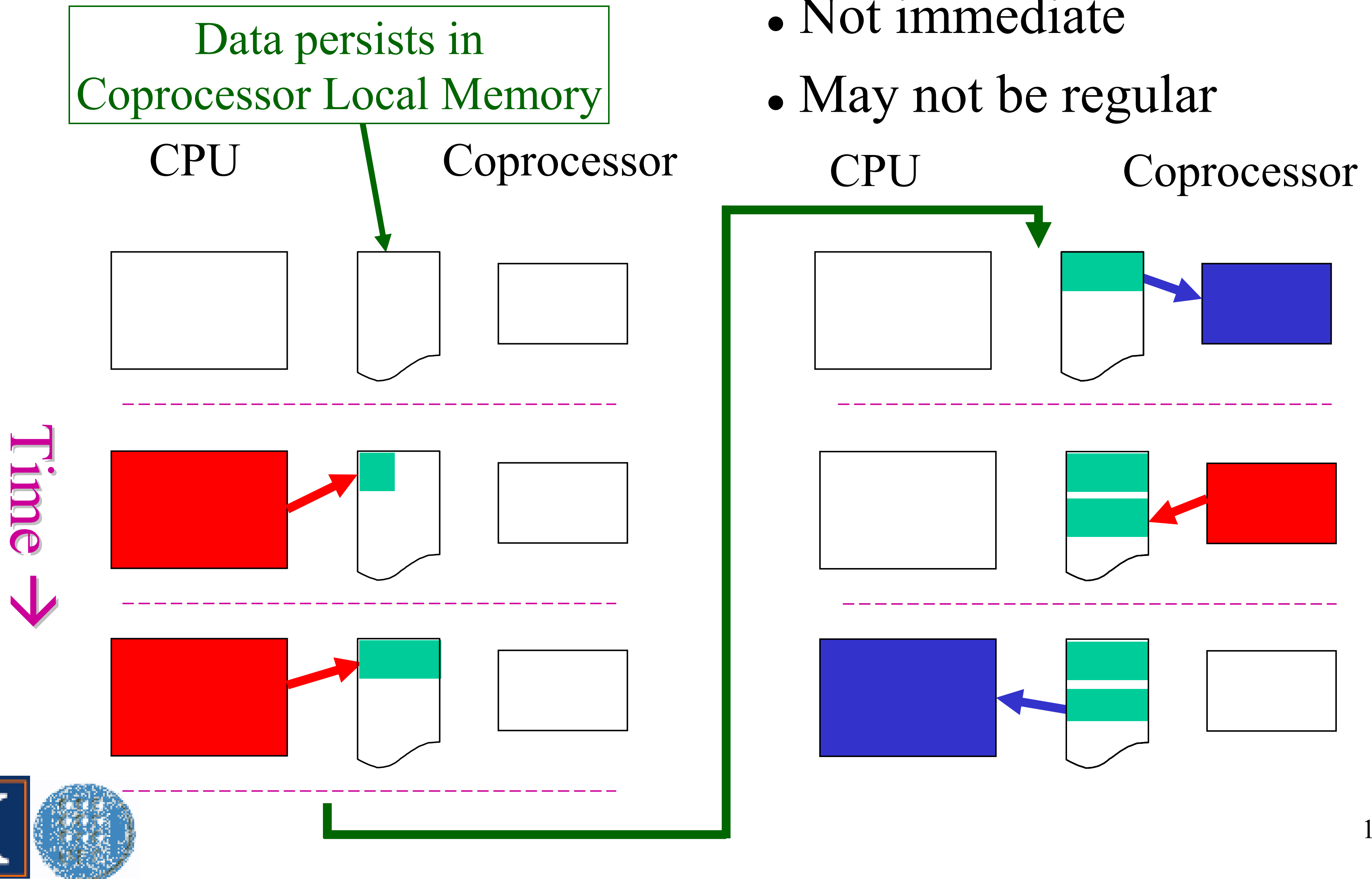
- Pass-by-value (**marshalling**)
  - Data copied (explicitly): Application → Coprocessor
  - Coprocessor has private copy
  - Data not persistent
- Pass-by-reference (**no marshalling**)
  - Coprocessor has reference to data (offset, pointer, etc.) with updates in-place
  - Data accessible by both CPU and coprocessor
  - Data built piecemeal, persists
- **Extend** coprocessor models: **Add** pass-by-reference



Data Persistence

Producer → Consumer:

- Not immediate
- May not be regular

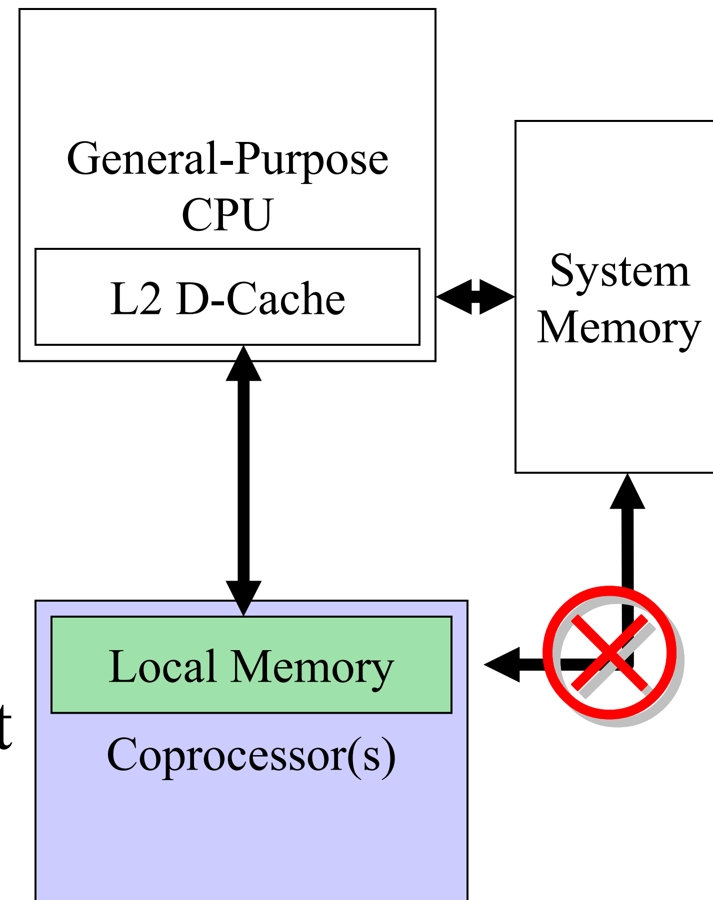


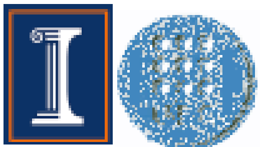
CPU/Coprocessor Architecture

Techniques based on mapping apps to CUBA CPU/coprocessor architecture

Attributes include:

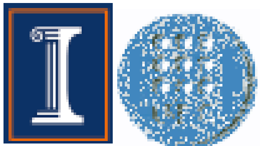
- –Coprocessor local memory ***hosts*** data
- –Coprocessor local memory ***cacheable*** by CPU (unlike MMIO)
- –***Low-latency*** for ***both*** CPU and coprocessor
- –Coprocessor memory not kept coherent (software managed coherence)



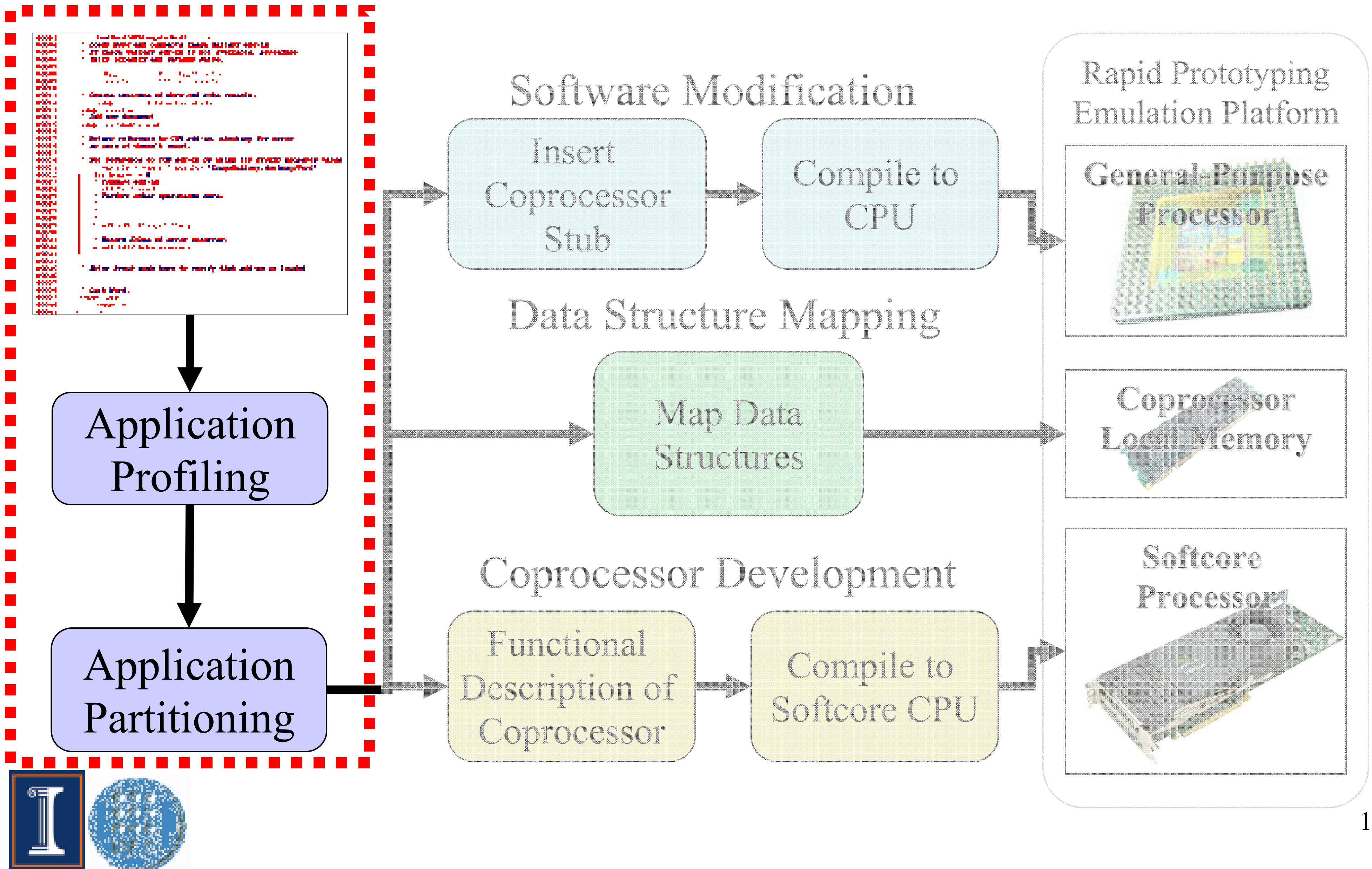


Where Does CIGAR Fit In?

- Sequential code → CUBA
- Aid developers partitioning sequential apps using visualization techniques
- Discover persistent state + appropriate code regions; map to data-parallel coprocessor with hosting
- Provide platform for prototyping partitioned designs
- **Bottom line**: More ***easily*** create ***correct*** mappings using ***fast*** emulation



CIGAR Methodology: **Analysis**



CIGAR Methodology: **Partitioning**

Software Modification

Insert Coprocessor Stub → Compile to CPU

Data Structure Mapping

Map Data Structures

Application Profiling

Application Partitioning

Coprocessor Development

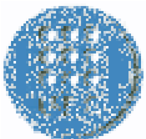
Functional Description of Coprocessor → Compile to Softcore CPU

Rapid Prototyping Emulation Platform

General-Purpose Processor

Coprocessor Local Memory

Softcore Processor



CIGAR Methodology: **Design/Debug**

Software Modification

Insert Coprocessor Stub → Compile to CPU

Data Structure Mapping

Map Data Structures

Application Profiling

Application Partitioning

Coprocessor Development

Functional Description of Coprocessor → Compile to Softcore CPU

Rapid Prototyping Emulation Platform

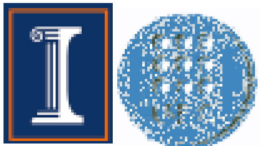
General-Purpose Processor

Coprocessor Local Memory

Softcore Processor

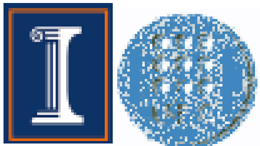
Pre-processing and Profiling

- Instrument source to provide hooks into CIGAR
- Profile apps to find comp. intense regions of code
- Filter out subroutines with little compute time, accelerates subsequent steps
- **Result**: C*andidate routines* to investigate



Data Parallelism Discovery

- **Simple Metric**: Analyze loops for data parallelism
- **Method**: Divide total number of instructions in trace by calculated height of DFG
- **Result:** Regions of code that may be accelerated with data-parallel coprocessors
- **Drawback:** Dynamic → Input dependent



Access Intensity

- Correlation between data structure access and candidate functions
- Visualization aids developer in making appropriate mapping
- **Demonstrates**: Need low-latency for CPU and coprocessor

CPU Access

Accelerator Access

Accelerator Access

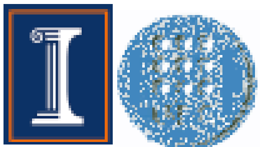
Function Execution Intervals

Candidate Functions

Hosted Data Structures

1 memory access per 9 instructions

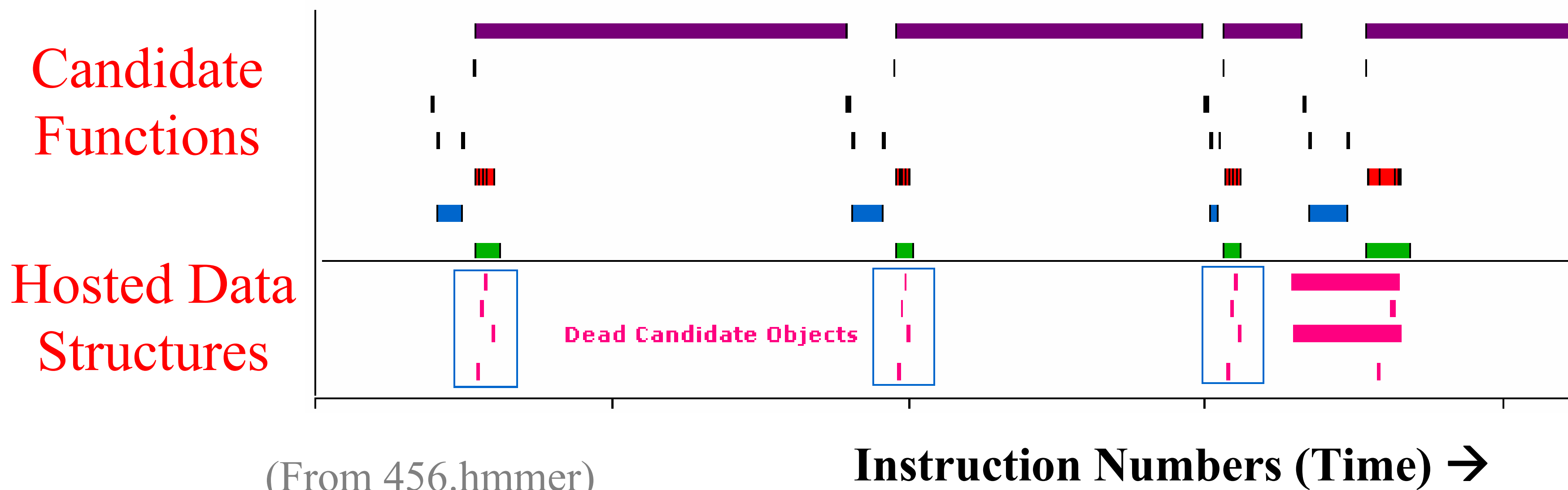
Instruction Numbers (Time) →



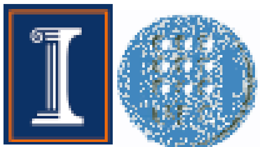
(From 462.libquantum)

Liveness Analysis

- With no backing-store, CLM state must be saved between remap or reallocation (**expensive operation**)
- Find intervals where hosted data structures dead
- **Drawback**: Need to hand verify correctness

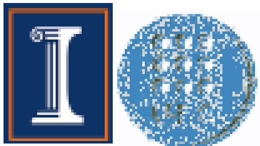


(From 456.hmmer)



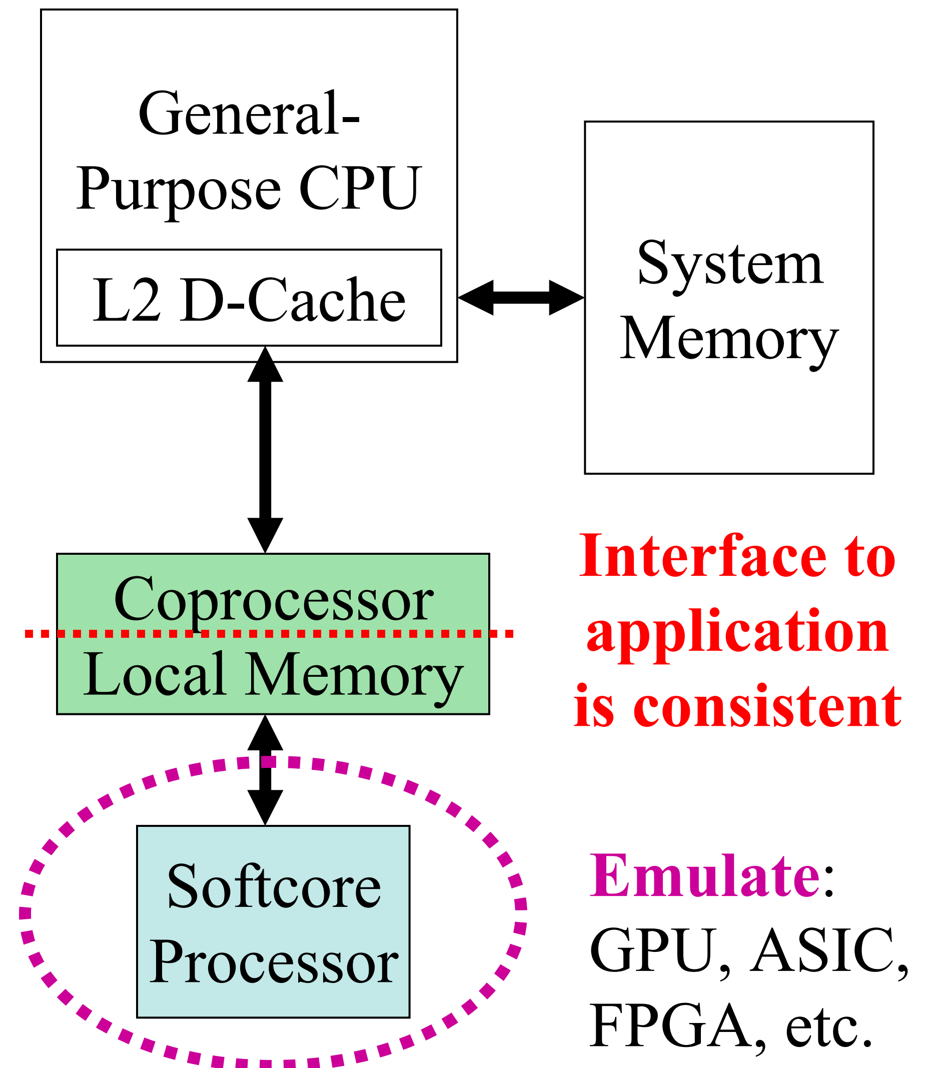
Emulation Platform

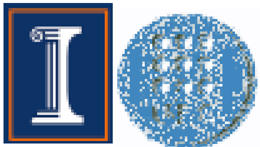
- Softcore processor for emulating coprocessor
- Local memory of coprocessor exposed
  - **Same interface** exported by actual coprocessor
  - Work out interface and ensure proper remapping
- Separates coprocessor function from implementation
- Iterate through designs quickly by **avoiding**:
  - High-level synthesis
  - Writing RTL
  - Place-and-route for FPGA designs
  - Waiting for silicon before software integration



Emulation Platform Implementation

- Supports standard compilers (`gcc`), debuggers (`gdb`), and performance monitors (`gprof`)
- Data structures mapped in stages
- Debug using standard practices
- Evaluate many *functionally* different coprocessors quickly

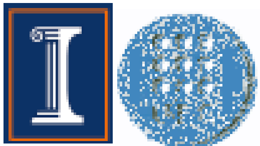




Simulation vs. Emulation

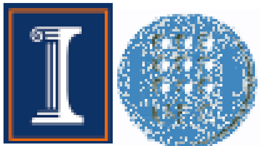
	462.libquantum	456.hmmer	464.h264ref
Native	1 x (0.30 s)	1 x (0.10 s)	1 x (1m13s)
Emulation	56 x	73 x	30 x
Simulation	2437 x	1180 x	3151 x

- Simulation
  - ⊕ High visibility
  - ⊗ Long runtime
  - ⊗ Left to debug the simulator
- Emulation
  - ⊕ Orders of magnitude better than simulation
  - ⊕ Stable platform: CPU and coprocessor local memory fixed
  - ⊕ More visibility vs. native
  - ⊗ Cannot evaluate performance directly



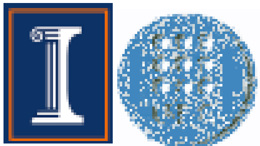
Summary

- Prevalence of data-parallel coprocessors
- Extend design techniques
- Extend architecture to avoid data marshaling + reduce overhead
- **CIGAR**: Techniques for isolating and mapping hosted data structures into CUBA
- Rapid prototyping platform



Conclusions

- Consistent view of resources (1. Extend)
- Visualizations + simple metrics (2. Methodology)
- Reduce difficulty prototyping/debug (3. Prototyping)
- Future work:
  - Virtualization of coprocessor local memory
  - More efficient memory usage/caching
- **Takeaway**: Map software to CPU/coprocessor by extending tools and techniques software developers understand



The End

Detailed View of CUBA

L1

CPU

L2

Memory Controller

Chip Area

Memory Controller

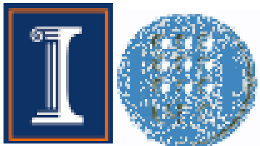
CLSC

Main Memory

Coprocessor

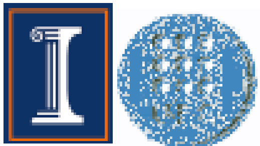
Coprocessor Local Storage

PCI Card



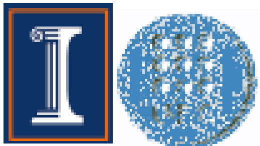
Execution Modes

- **Baseline**
  - Block on coprocessor access
  - Immediate polling
- **Independent Execution Mode**
  - Concurrent CPU and coprocessor execution
  - Defer polling
- **Exception Handling Mode**
  - Begin executing on coprocessor
  - On error, revert to CPU execution
  - Simplify coprocessor design: Eliminate infrequent execution paths



Mapping Steps

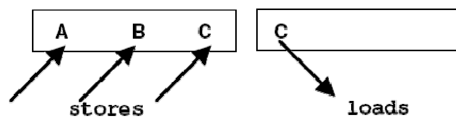
1. Software-only Profile
   - Collect information
   - Determine initial partitioning
2. Software Memory Debug
   - Move data structures to coprocessor local memory
   - Software structure remains unchanged
3. Coprocessor Debug
   - Add coprocessor to design
   - Debug *functional* correctness
4. Coprocessor Profiling
   - Evaluate quality of partitioning



Data Synchronization Granularity

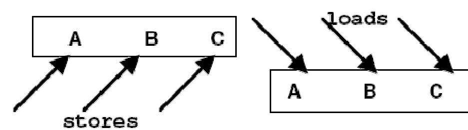
Sequential Access Pattern | Concurrent Access Pattern

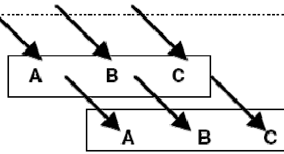
Stacked



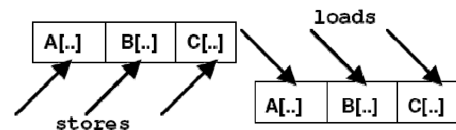
X

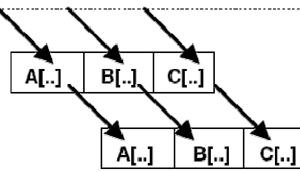
Full Streaming



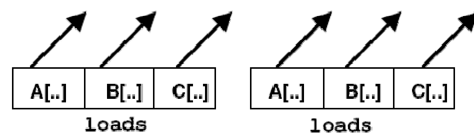


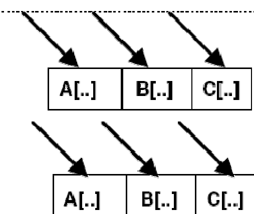
Block Streaming





Read–Only Concurrent





Flow of Execution

- Patterns found in benchmarks evaluated
- Impact on CPU/ Coprocessor architecture
- Use coprocessor data structure hosting to increase concurrency