Presented at the 36th Annual International Symposium on Computer Architecture June 22nd, 2009

Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator

John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri*, Steven S. Lumetta, Matthew I. Frank†, Sanjay J. Patel

*The author is now with NVIDIA.
† The author is now with Intel.

Rigel ILLINOIS UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Accelerated Computing: Today

Programmable accelerator: HW entity designed to provide advantages for a class of apps including: higher performance, lower power, or lower unit cost relative to a general-purpose CPU.

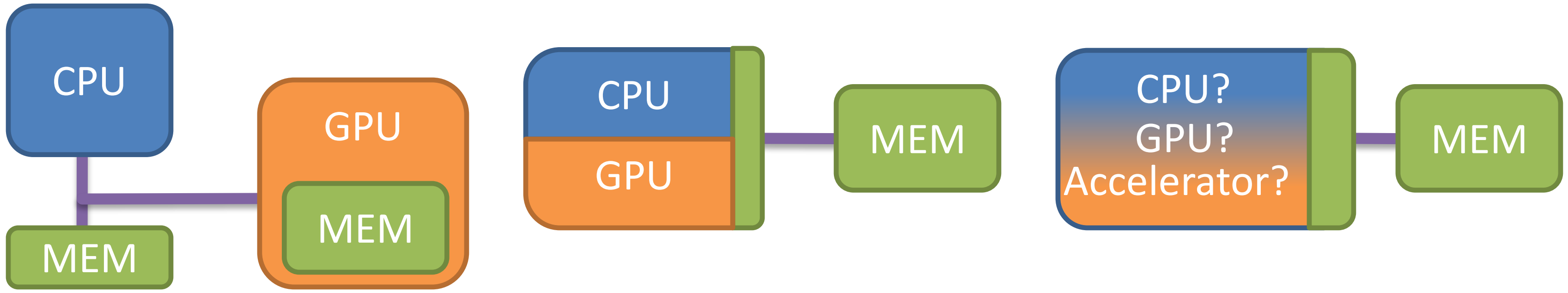
- Contemporary Accelerators: GPUs, Cell, Larrabee
- **Challenges**:
  1. Inflexible programming models
  2. Lack of conventional memory model
  3. Hard to scale irregular parallel apps

Effect on Development: Unattractive time to solution

Accelerated Computing: Tomorrow

- Why research accelerators?
  - Insight into future general-purpose CMPs
  - **Challenges**: Performance vs. programmer effort
- **Accelerator Trend**: Integration over time

Past… **…Present…** **…Future?**



Accelerated Computing: Metrics

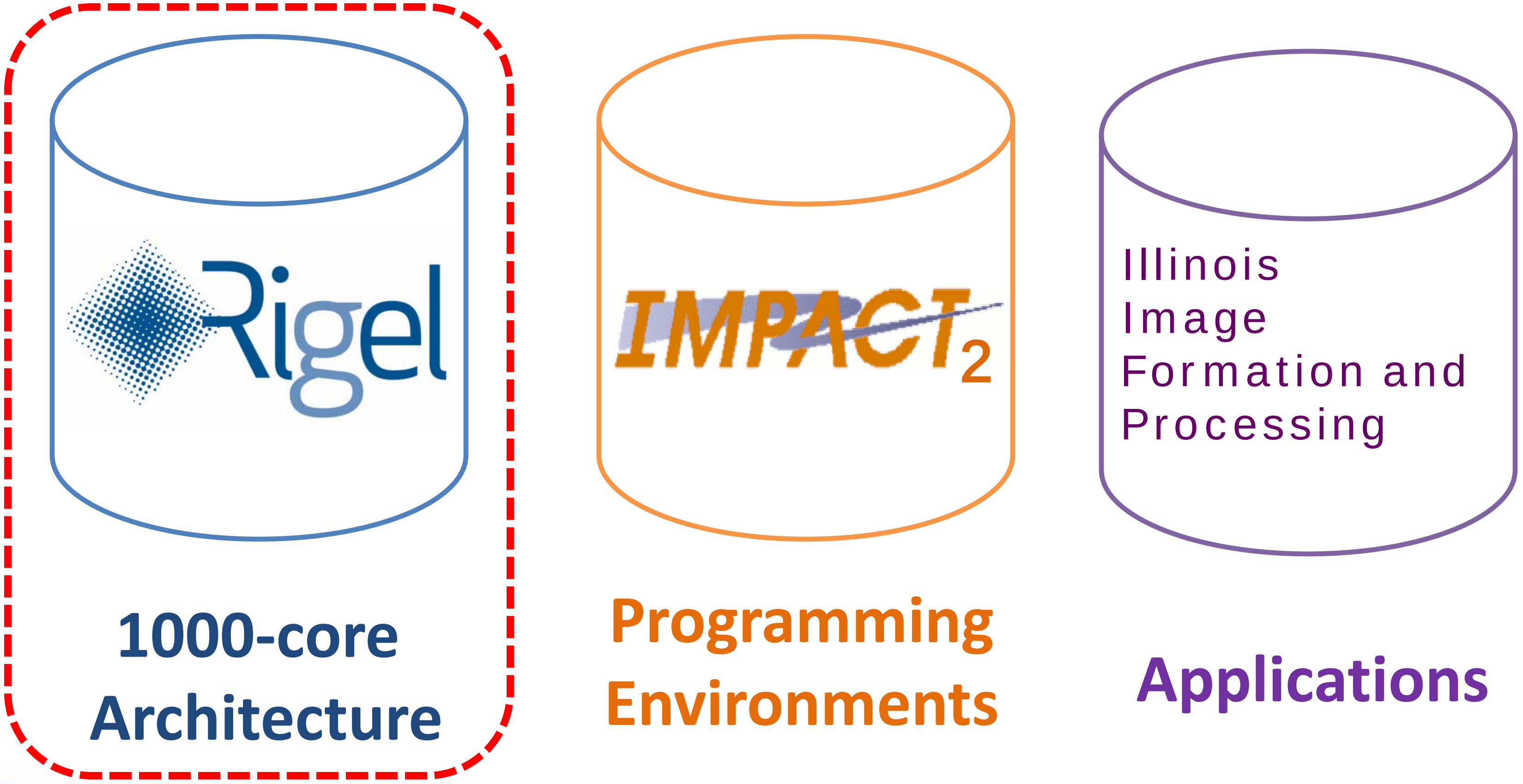
Challenges lead to:

- **FLOPS/$ (area)**
- **FLOPS/Watt (power)**
- **FLOPS/Programmer Effort**

- Enable new platforms
- Open new markets
- Enable new apps

Context: Project Orion

Applications, Programming Environments, and Architecture for 1000-core Parallelism



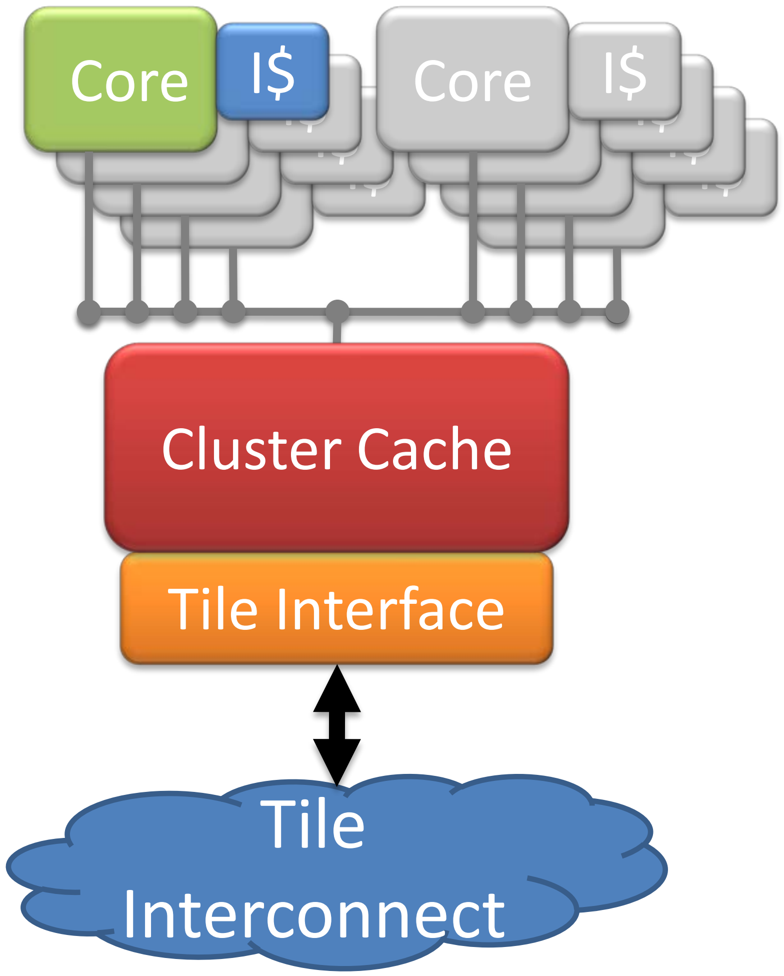
Rigel Design Goals

- **What**: Future programming models
  - Apps and models may not exist yet
  - We have ideas (**visual computing**), but who knows?
  - Flexible design → easier to retarget
- **How:** Focus on scalability, programmer effort
  - Room to play: Raised HW/SW interface
  - Focusing design effort: **Five Elements**

Outline

- Motivation
- Rigel architecture
- **Elements** in context of Rigel architecture
- Evaluation:
  - Area and power
  - Scalability
  - SW Task management
- Future work and conclusions

Rigel Architecture: Cluster View



- Basic building block
- Eight 32b RISC cores
- Per-core SP FPUs
- 64 kB shared cache
- Cache line buffer

Rigel Architecture: Full Chip View

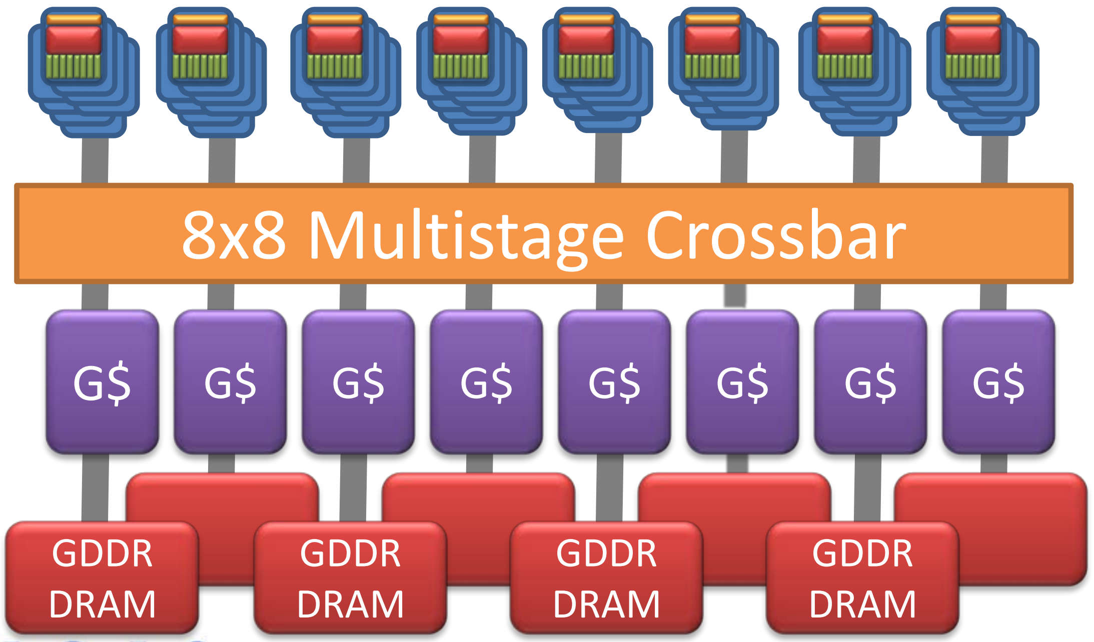
- Cluster caches not HW coherent (8 MB total)
- G$ fronts mem. controllers (4 MB total)
- Uniform cache access

8 Tiles Per Chip
16 Clusters per Tile

Global Interconnect

Global Cache Banks

8 GDDR Channels



Design Elements

- **Challenges** in accelerator computing
- **FLOPS/dev. effort** → Difficult to quantify
- Guiding our 1000-core architecture
- Room to Play: Raising the HW/SW interface

Design Elements

1. **Execution Model**: ISA, SIMD vs. MIMD, VLIW vs. OoOE, MT
2. **Memory Model**: Caches vs. scratchpad, ordering, coherence
3. **Work Distribution**: Scheduling, spectrum of SW/HW choices
4. **Synchronization**: Scalability, influence on prog. model
5. **Locality Management**
   – Moving data costs perf. and power
   – Balance: dev. effort, compiler, runtime, HW

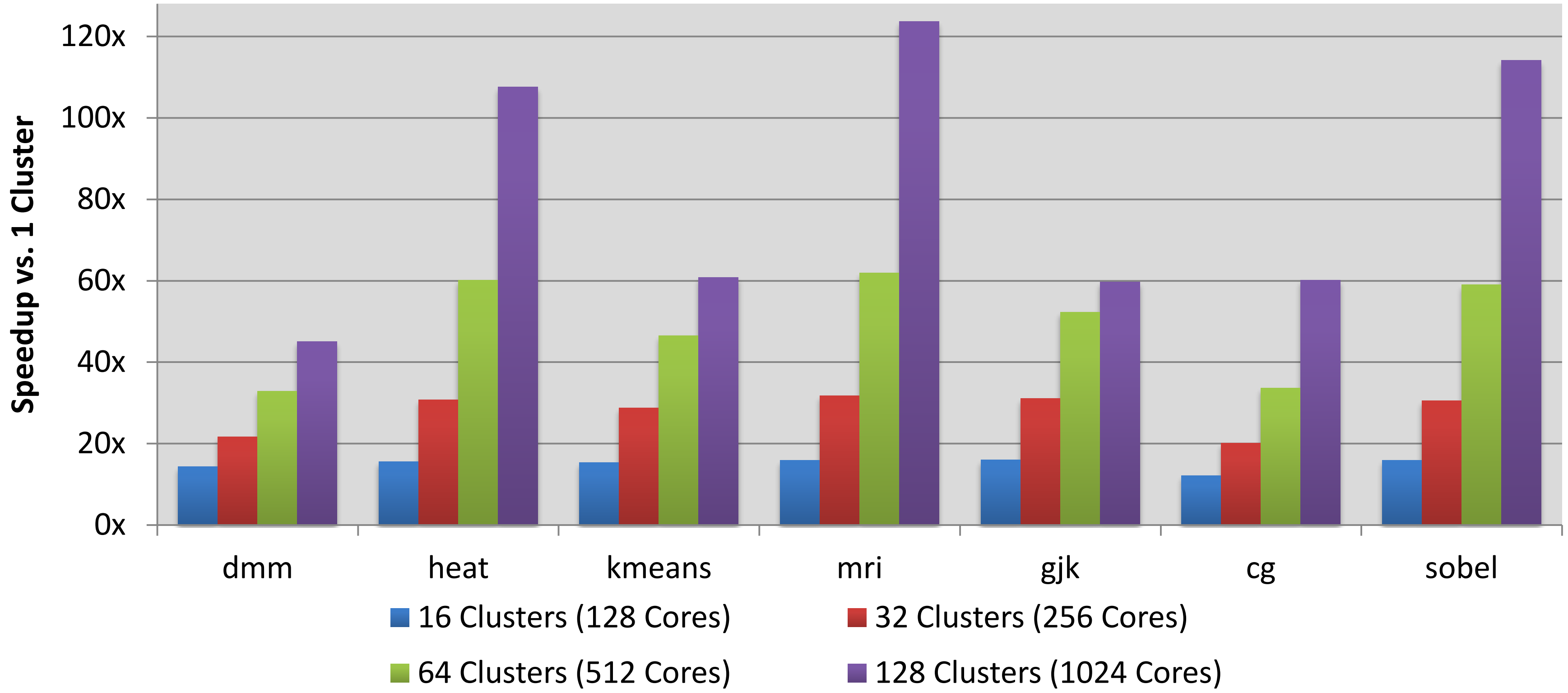
Element 1: Execution Model

- **Tradeoff 1**: **MIMD** vs. SIMD [Mahesri MICRO'08]
  - Irregular data parallelism
  - Task parallelism
- **Tradeoff 2:** Latency vs. **Throughput** [Azizi DasCMP'08]
  - Simple in-order cores
- **Tradeoff 3:** **Full RISC ISA** vs. Specialized Cores
  - Complete ISA → conventional code generation
  - Wide range of apps

Element 2: Memory Model

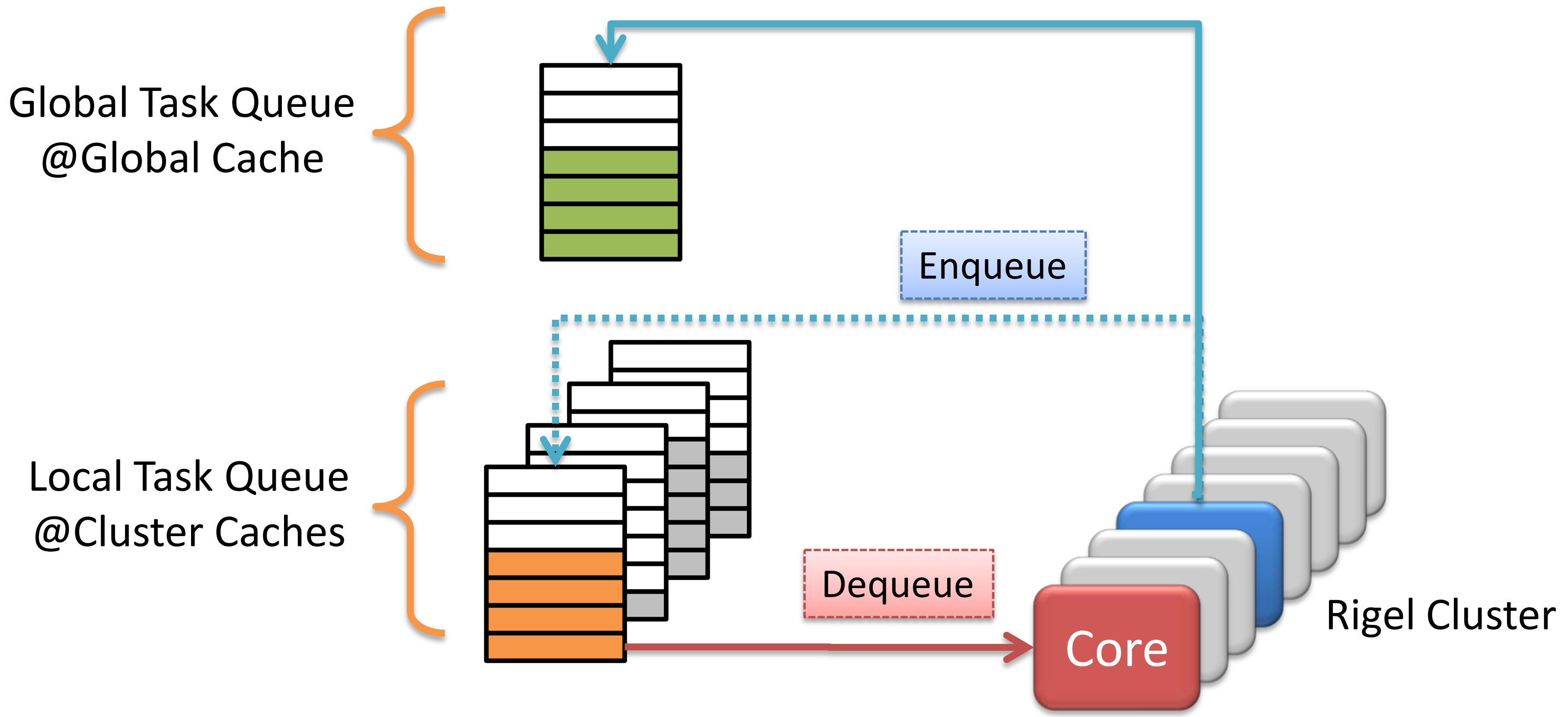
- **Tradeoff 1: Single** vs. multiple address space
- **Tradeoff 2: Hardware caches** vs. scratchpads
  - Hardware exploits locality
  - Software manages global sharing
- **Tradeoff 3: Hierarchical** vs. Distributed (NUCA)
  - Cluster cache/global cache hierarchy
  - ISA provides local/global mem. Operations
  - Non-uniformity → Programmer effort

Some Results: **Scalability**



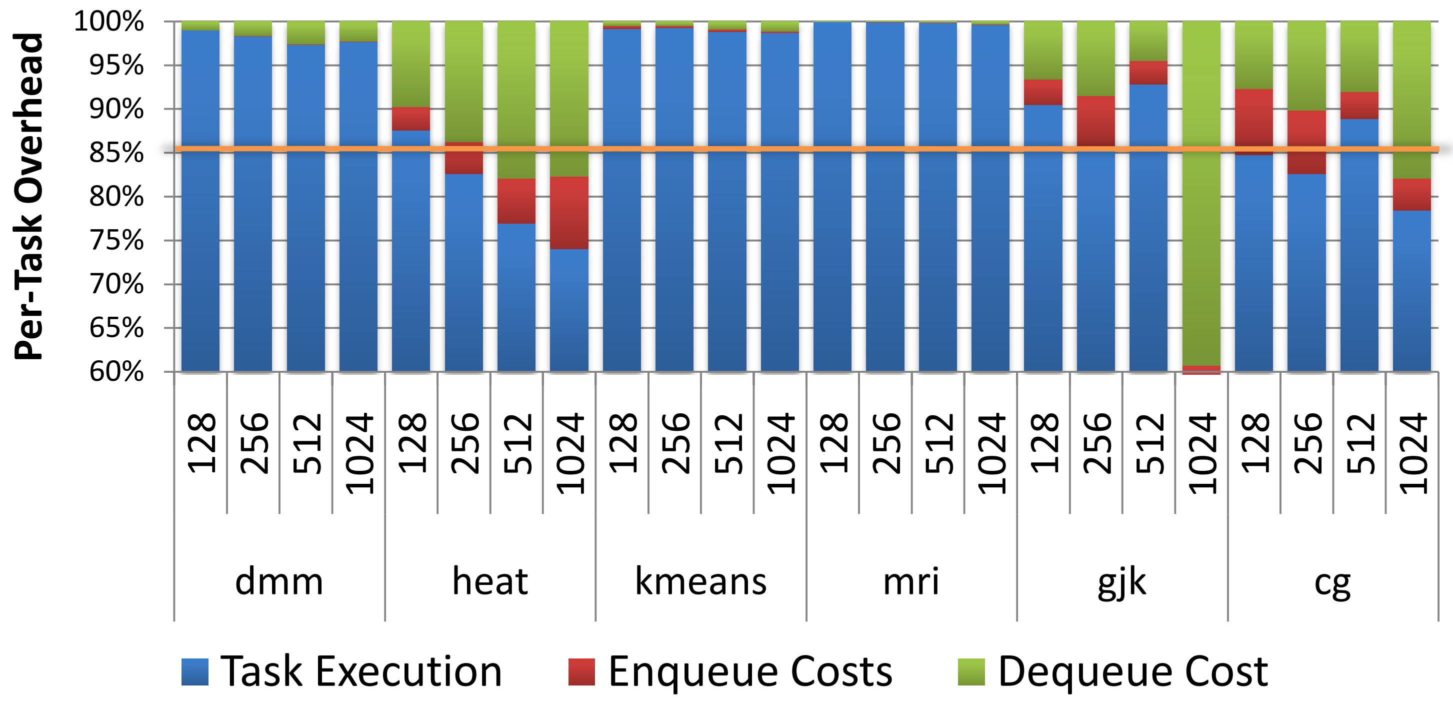
- Based on cycle-accurate, execution-driven simulation
- Library and run-time system code simulated
- Regular C code + parallel library, standard C compiler

Element 3: Work Distribution



- **Tradeoff (Spectrum):** HW vs. **SW Implementation**
  - SW task management: Hierarchical queues
  - Flexible policies + little specialized HW

Work Distribution: Rigel Task Model

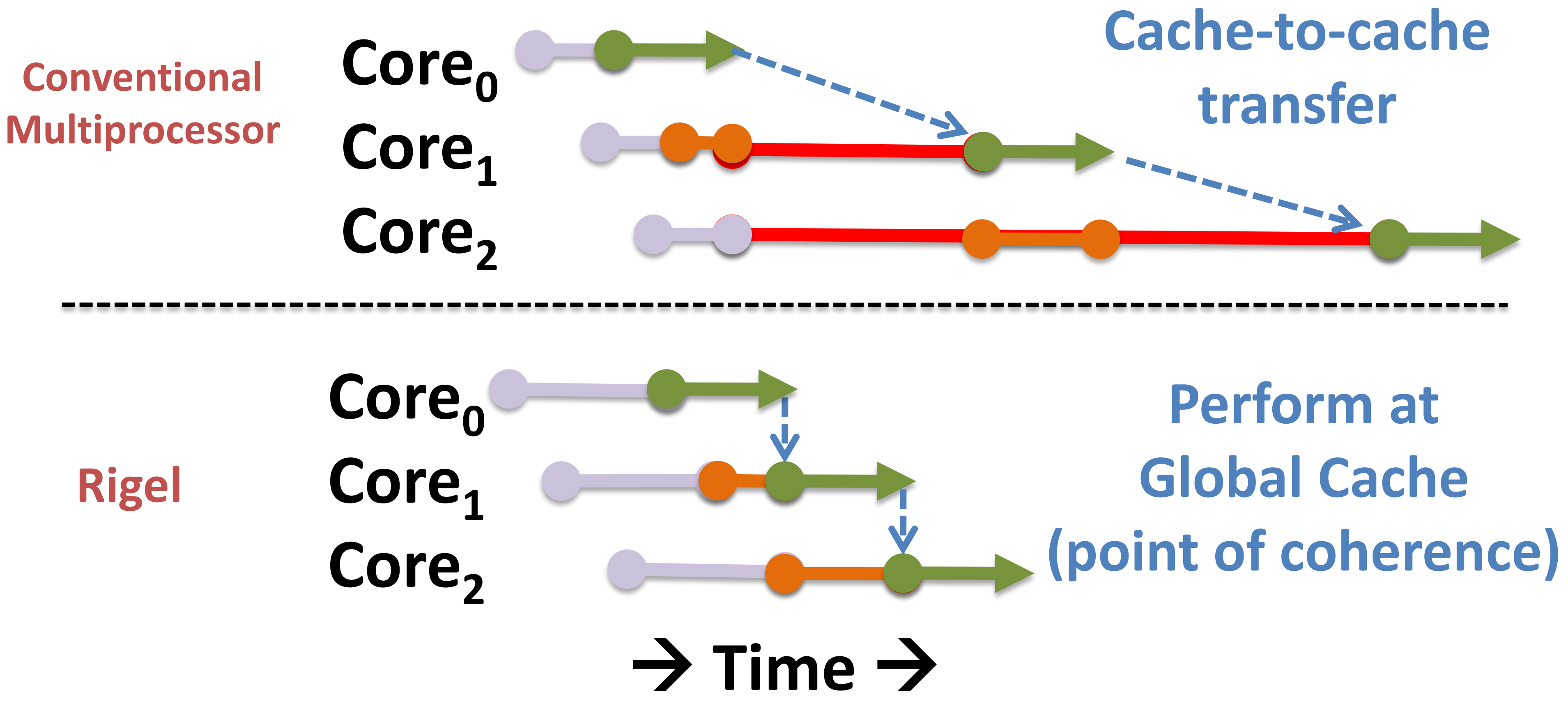


- < 5% overhead for most data-parallel workloads
- < 15% for most irregular data-parallel workloads
- Task lengths: 100’s-100k instructions

Element 4: Synchronization

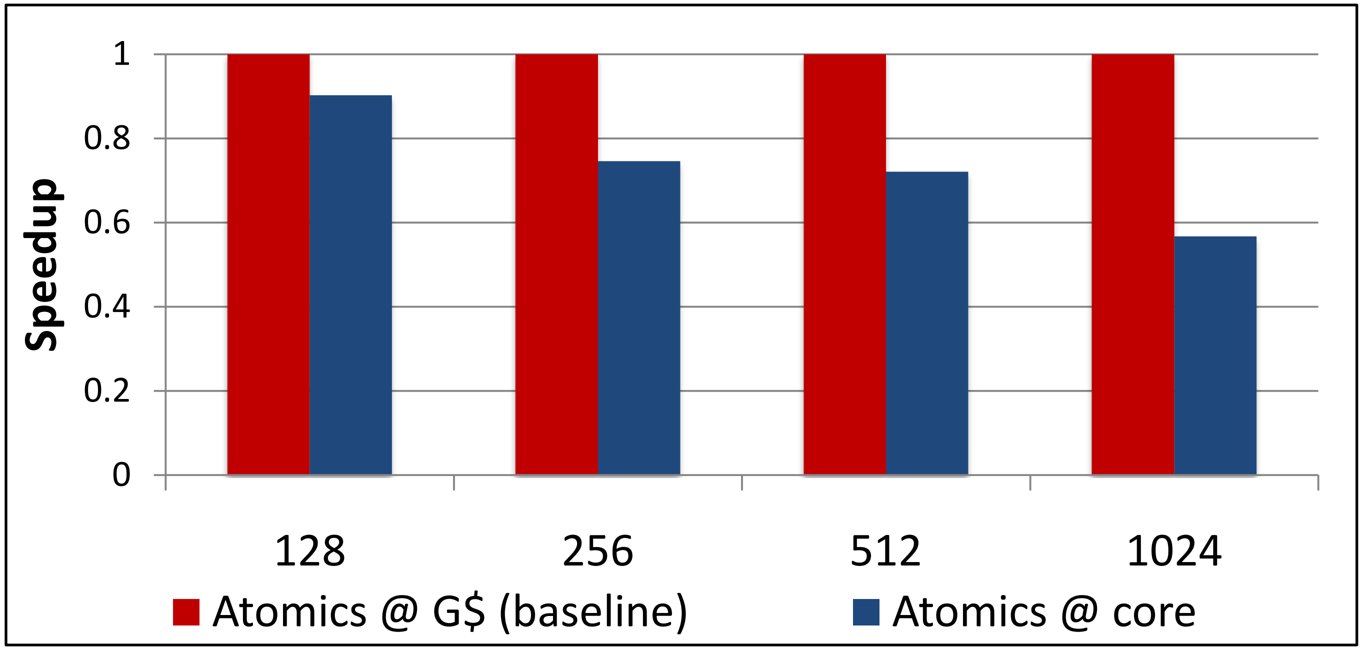
- Uses of coherence mechanisms:
  1. Control synchronization
  2. Data sharing
- Broadcast update
  – Use cases: flags and barriers
  – Reduce contention from polling
  – **Case Study**: 2x speedup for conjugate gradient (CG)
- Atomic primitives (example)

Element 4: Atomic Primitives



1. Network Latency
2. Overlapped Latency
3. Exposed Latency
4. Operation Execution (atom.inc)

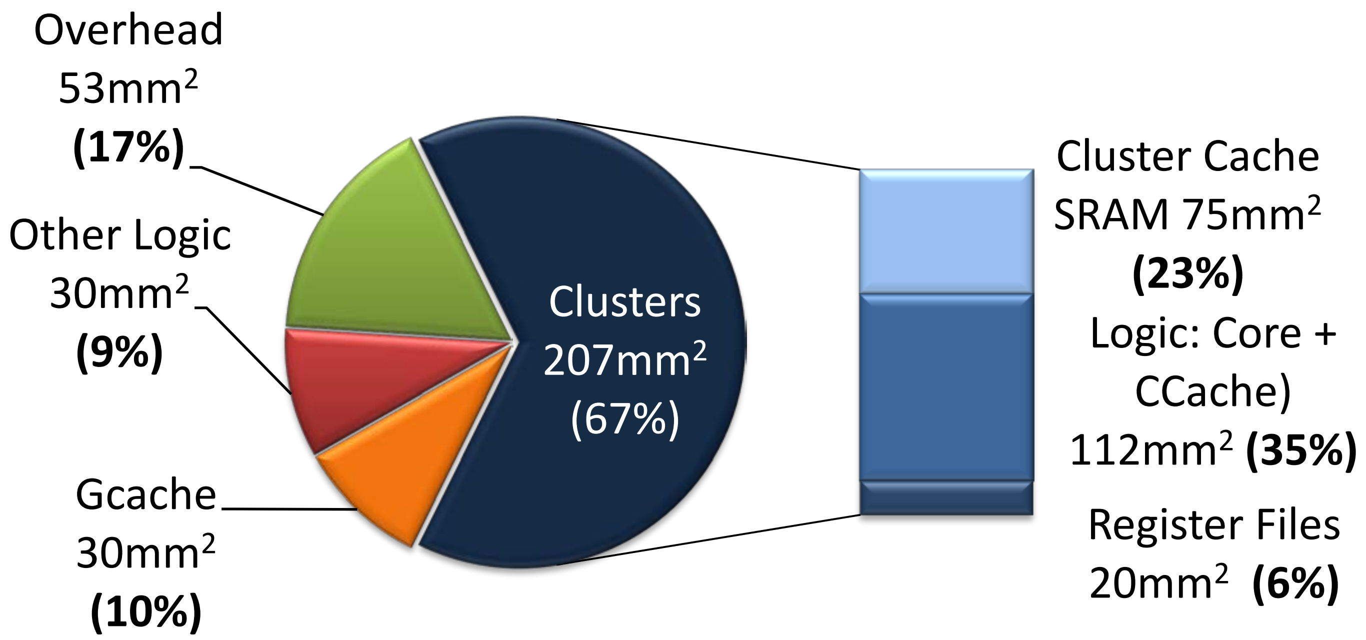
Evaluation: Atomic Operations



K-means Clustering

- Need global histogramming
- With G$ atomics → Pipelined in network
- Without atomics → Exposed transfer latency

So, Can We Build It?



- RTL synthesis results + memory compiler + datasheets
- Targeting 45nm process @ 1.2 GHz
- 320 mm^2 total die area, <100W average power
- Estimate FLOPS/W and FLOPS/mm^2 **match or exceed GPUs**

Current and Future Work

- RTL implementation
- Coherence and memory model [Kelm et al. PACT'09]
- Other programming models
- Multi-threading (1-4 threads)
- Element Five: Locality Management

Conclusions

- **FLOPS/Dev. Effort** →Elements can drive design
- Software coherence viable approach
- Task management requires little HW
- 1000-core accelerator is feasible
  - **Area/performance**: 8 GFLOPS/mm^2 @ ~100W
  - **Programmability**: Task API + MIMD execution