Architecture for Energy Efficient Execution of Graph Analytics Applications

SERIF YESIL[†], M. MUSTAFA OZDAL[†], OZCAN OZTURK[†]

[†]BILKENT UNIVERSITY, ANKARA, TURKEY

TAEMIN KIM[*], ANDREY AYUPOV[*], STEVEN M. BURNS[*]

[*]STRATEGIC CAD LABS, INTEL

Dark Silicon Era

- End of Dennard scaling & dark silicon

  Power is the main limiting factor

- Hardware specialization & heterogeneity

  Execute each workload on the most efficient hardware

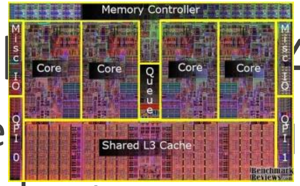
- What type of architectures needed for different application domains?

  Many core? SIMD? Latency tolerant? …

Limitations of Existing Architectures

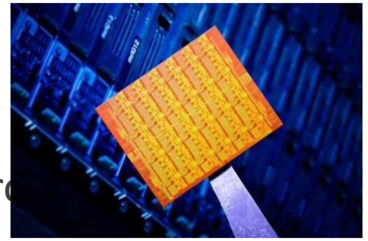
- CPUs: [1,2,3]
  - Low IPC for state-of-the-art systems
  - Single OOO core number of outstandi equests are limited
  - Synchronization overhead
- Thr rc 4,5
  - Separa exe uire nchr B
  - Control divergence due to asymmetric convergence
  - Memory divergence un-coalesced memory accesses are limited due to irregular memory accesses

GPUs



Multicore

Clusters



Mini Clouds

amazon web services™

Clouds

Cloud Computing & Data Centers

- Increasingly more computation done in the cloud
- NRDC report:

  If worldwide data centers were a country, it would be the 12^{th} largest consumer of electricity.

- Specialize data centers for energy efficiency
- What type of architectures needed for different application domains?

Source: NRDC

Energy Crisis in Data Center

Year	End-use Energy (B kWh)	Elec. Bills (US, $B)	Power plants (500 MW)	CO2 (US) (million MT)
2013	91	$9.0	34	97
2020	139	$13.7	51	147
2013-2020 increase	47	$4.7	17	50

This chart shows the estimated power usage (in billions of kilowatt-hours), and the cost of power used, by U.S. data centers in 2013 and 2020, and the number of power plants needed to support the demand. The last column shows carbon dioxide (CO_2) emissions in millions of metric tons. (Source: NRDC)

[Source:NRDC]

Motivation for Accelerators

- Many datacenters execute the same tasks
  - Hardware can be customized for specific workloads
  - Especially for big data problems like Graph applications
  - Power and performance efficiency is needed

24 Million
Wikipedia Pages

750 Million
Facebook Users

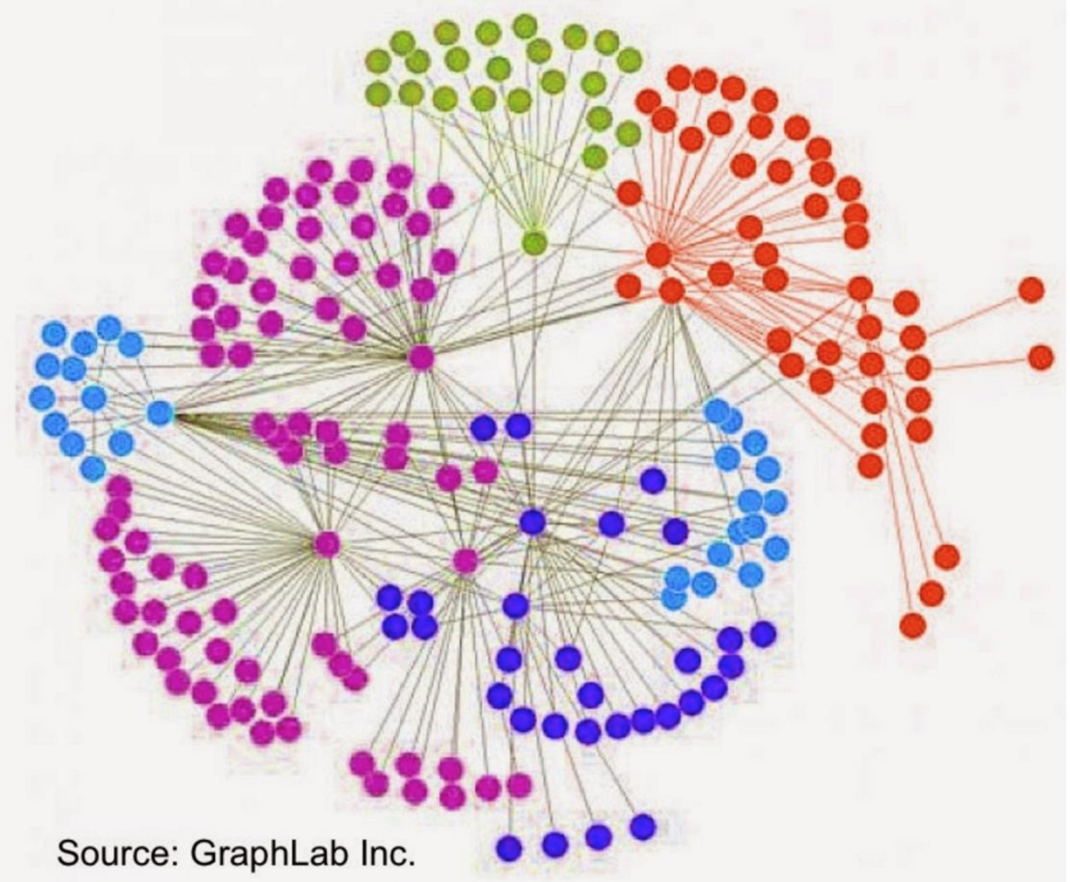
6 Billion
Flickr Photos

48 Hours a Minute
YouTube

[From GraphLab]

Graph Analytics

- Model relationships between individual entities
- Knowledge discovery and data mining
  Extract actionable information from data.
- Many application areas:
  Web, social networks, biological pathways, …
- Example applications: PageRank, Collaborative Filtering, Loopy Belief Propagation, Betweenness Centrality, …



Objectives

- ❑ Identify the archtiectural requirements of energy-efficient execution of irregular graph applications.

- ❑ Why focus on graph applications?
  - ❑ Increasing importance in emerging applications
  - ❑ Different than traditional grid-based HPC

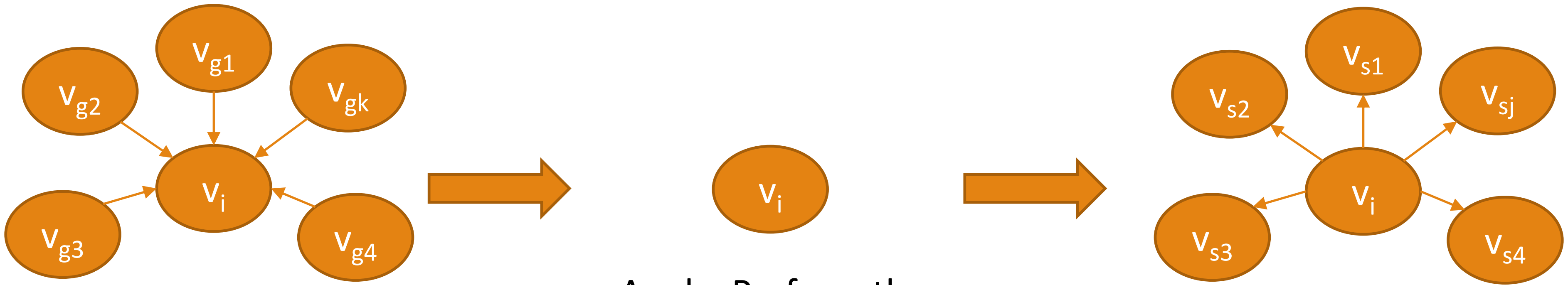
❑ Irregular data access & communication
❑ Low data locality
❑ Low computation-to-communication ratio
❑ Dynamic/hard-to-predict work assignment

J. Feo, "Graph Analytics in Big Data", Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC) 2012.

Outline

- ❑ Application Characteristics
- ❑ Implementation Challenges and Customization Opportunities
- ❑ Experimental Results

Gather-Apply-Scatter (GAS) Model



Gather:
Collect and accumulate data from the neighboring vertices and edges

Apply: Perform the main computation for the input vertex using the Gather results.

Scatter: Distribute the vertex data computed in Apply to neighbors. Determine whether to schedule the neighboring vertices for future execution

Example Application: PageRank

Vertex program executed for each vertex v

sum = 0

for each vertex u for which (u → v) exists

$$sum = sum + \frac{rank(u)}{degree\ (u)}$$

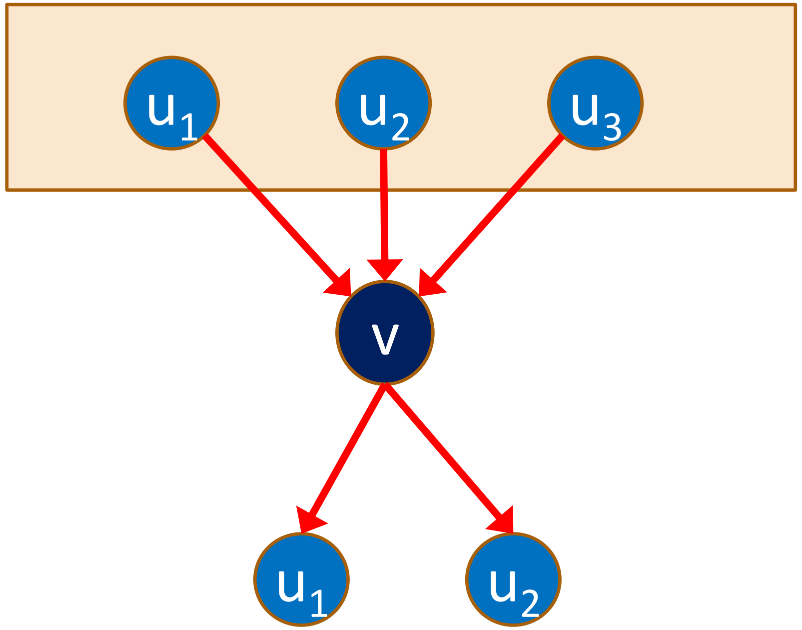
$$rank^{new}(v) = \frac{1-\alpha}{|V|} + \alpha\ sum$$

if $|\, rank^{new}(v) - rank(v)\,| > \epsilon$ then

for each vertex w for which (v → w) exists

schedule w for future execution

$$rank(v) = rank^{new}(v)$$



Example Application: PageRank

Vertex program executed for each vertex v

sum = 0

for each vertex u for which (u → v) exists

$$sum = sum + \frac{rank(u)}{degree\,(u)}$$

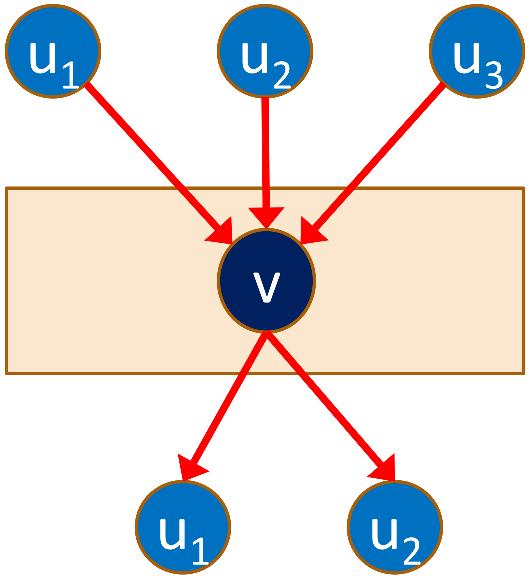
$$\boxed{rank^{new}(v) = \frac{1-\alpha}{|V|} + \alpha\ sum}$$

if $|\,rank^{new}(v) - rank(v)\,| > \epsilon$ then

for each vertex w for which (v → w) exists

schedule w for future execution

$$\boxed{rank(v) = rank^{new}(v)}$$



Example Application: PageRank

Vertex program executed for each vertex v

sum = 0

for each vertex u for which (u → v) exists

$$sum = sum + \frac{rank(u)}{degree\ (u)}$$

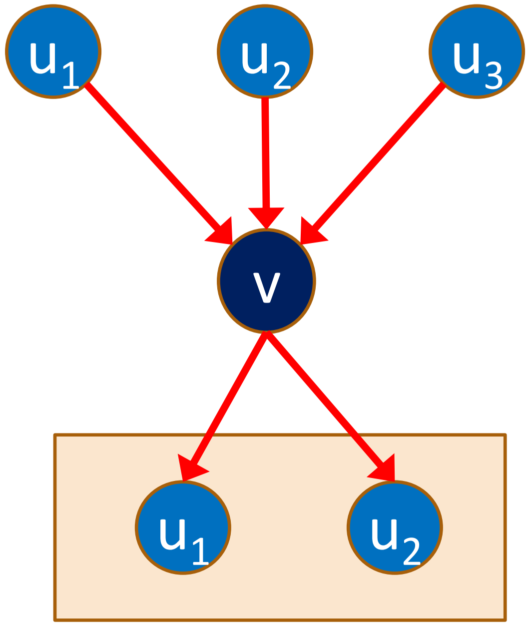
$$rank^{new}(v) = \frac{1-\alpha}{|V|} + \alpha\ sum$$

if $|\,rank^{new}(v) - rank(v)\,| > \epsilon$ then

for each vertex w for which (v→ w) exists

schedule w for future execution

$$rank(v) = rank^{new}(v)$$



Asymmetric Convergence

- Process all vertices in every iteration?
  - Easier to implement
  - Typically higher throughput due to more regularity
  - Unnecesary computation

    e.g. Only 0.3% of the vertices need all 77 iterations

- Process only “active” vertices?
  - Overhead to keep track of the active list
  - Harder to parallelize
  - Control divergence issues for SIMD
  - More work-efficient

Synchronous vs. Asynchronous Execution

Jacobi iteration formula for PageRank:

$$r^{k+1}(v) = (1-\alpha) + \alpha \sum_{(u \to v)} \frac{r^k(u)}{degree(u)}$$

Synchronous: All vertices are updated simultaneously.

Gauss-Seidel iteration formula for PageRank:

$$r^{k+1}(v) = (1-\alpha) + \alpha \sum_{\substack{u<v \\ (u \to v)}} \frac{r^{k+1}(u)}{degree(u)} + \alpha \sum_{\substack{u>v \\ (u \to v)}} \frac{r^k(u)}{degree(u)}$$

Asynchronous: Updates to a vertex are visible to others in the same iteration.

Synchronous vs. Asynchronous Execution

- PageRank: Gauss-Seidel can converge ~2x faster than Gauss-Jordan
- Concept generalized by GraphLab:
  - **Synchronous**: v's data visible to neighbors in the next iteration
  - **Asynchronous**: v's data visible to neighbors immediately in the same iteration

- *Sequential consistency: The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [Lamport, 1979].*
  - In short: Parallel execution corresponds to some sequential execution.

Convergence Behavior

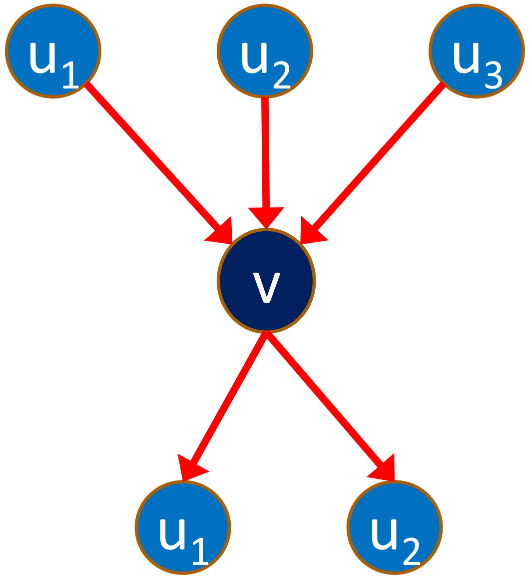
- Example: *Simple graph coloring*
  - <u>*Vertex program*</u>: Read the colors of all neighbors.
    Choose a color different from all neighbors.
  - Sequential version is guaranteed to terminate.
  - Parallel execution without sequential consistency may not terminate

- Better convergence with sequential consistency for some iterative algorithms
  - Examples: Alternating Least Squares, Gibbs Sampling [Low, 2012]

Memory Access Bottlenecks

- Typically, small amount of computation per vertex or edge.

- Unstructured graphs: Poor spatial and temporal locality
  - Access to the neighboring vertex/edge data likely to be a cache miss



Vertex Degrees

- Power law distribution for vertex degrees
  - A small percent of vertices are connected to most of the edges.

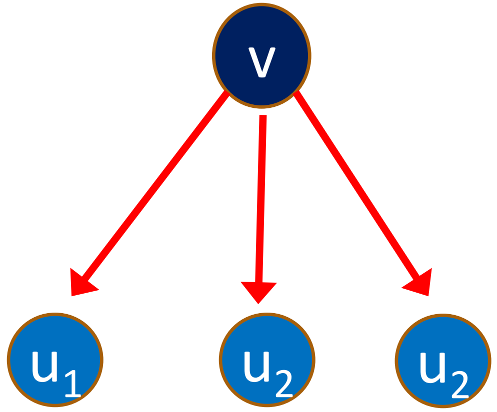
- Vertex-based partitioning likely to lead to load imbalances

Outline

- Application Characteristics
- Implementation Challenges and Customization Opportunities
- Experimental Results

Active Vertex Set Requirements

- ❑ Storage in long-latency main memory with efficient caching
- ❑ Latency tolerance mechanisms
- ❑ High-throughput access mechanisms
- ❑ Race-free simultaneous accesses without explicit locks
- ❑ Asynchronous execution support



Efficient hardware mechanisms needed for neighbor activation

Asynchronous Execution Support

- *Synchronous*: All vertices logically executed simultaneously in an iteration
  - Read from last iteration's data, write to next iteration's data
- *Asynchronous*: Vertices executed in a (logically) sequential order
  - Read from and write to the same data

- Asynchronous execution with or without sequential consistency
- Sequential consistency expensive to enforce in software

Hardware support for low-overhead race-free execution of many vertices simultaneously

Latency Tolerance Support

- General-purpose OOO logic not necessary for graph-parallel execution
- Basic idea:
  - Maintain a partial state for each vertex or edge processed
  - Non-blocking access to memory
  - Special mechanisms to guarantee race-free execution and sequential consistency.

Hardware support for efficient latency tolerance for graph parallelism.

Dynamic Load Balancing

- Many vertices/edges processed simultaneously
- Power-law distribution for vertex degrees
- Hardware resources should be utilized efficiently in different cases:
  - Many vertices with small degrees
  - Few vertices with very large degrees
- Control divergence issues for SIMD-style execution of vertices

  e.g. When vertices assigned statically to GPU threads

Hardware support for dynamic scheduling of vertices and edges

Memory Subsystem Customization

- Different access patterns per data structure
- Examples:
  - Good spatial locality for adjacency list
  - Poor temporal/spatial locality for edge data
  - Special data structure for active list

Custom cache/buffer types and microarchitecture parameter set for each data type

Proposed Architecture

- Tens of vertices and hundreds of edges are processed simultaneously
- Dynamic load balancing, via keeping partial states for vertices
- A distributed synchronization unit to ensure sequential consistency
- Keeps an active list for not-yet-converged vertices
- An optimized memory subsystem for irregular memory accesses

Outline

- Application Characteristics
- Implementation Challenges and Customization Opportunities
- Experimental Results

Benchmarks

Applications

- PageRank (PR)
- Single Source Shortest Path (SSSP)
- Stochastic Gradient Descent (SGD)
- Loopy Belief Propagation (LBP)

Datasets

- *PR & SSSP*: 6 datasets from Snap and generated with Graph500 (**up to 1B edges**)
- *LBP*: 3 images generated with GraphLab's synthetic image generator (**up to 18M edges**)
- *SGD:* 2 movie datasets from MovieLens (**up to 10M edges**)

Experimental Setup

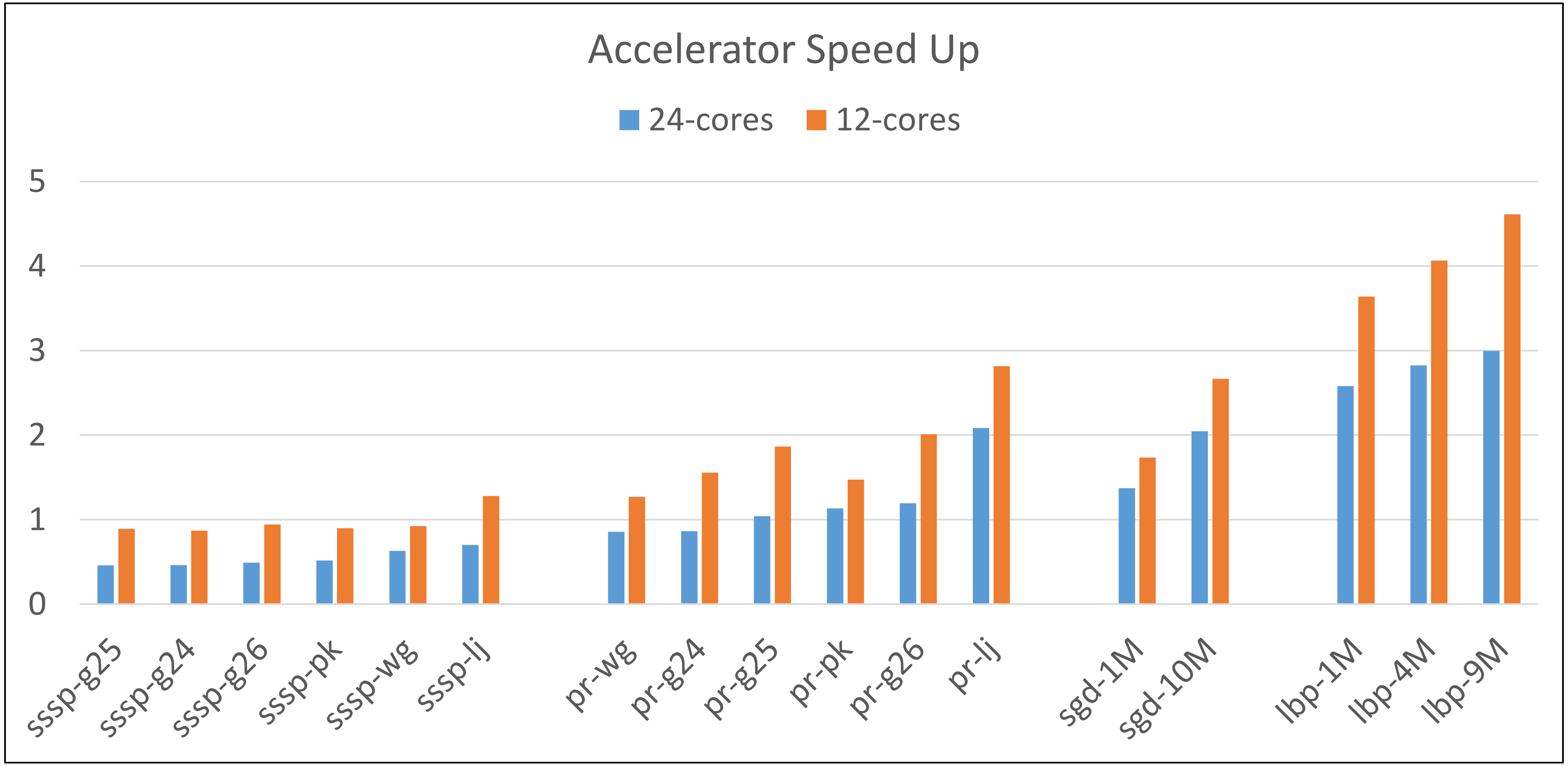
Baseline CPU

- 2-socket 24-core IvyBridge Xeon with 30MB LLC and 132GB of main memory
- Optimized software implementations in OpenMP/C++
- Running Average Power Limit (RAPL) to estimate energy
- Projected DDR3 power (measured) to DDR4 power (in-house DDR4 model)

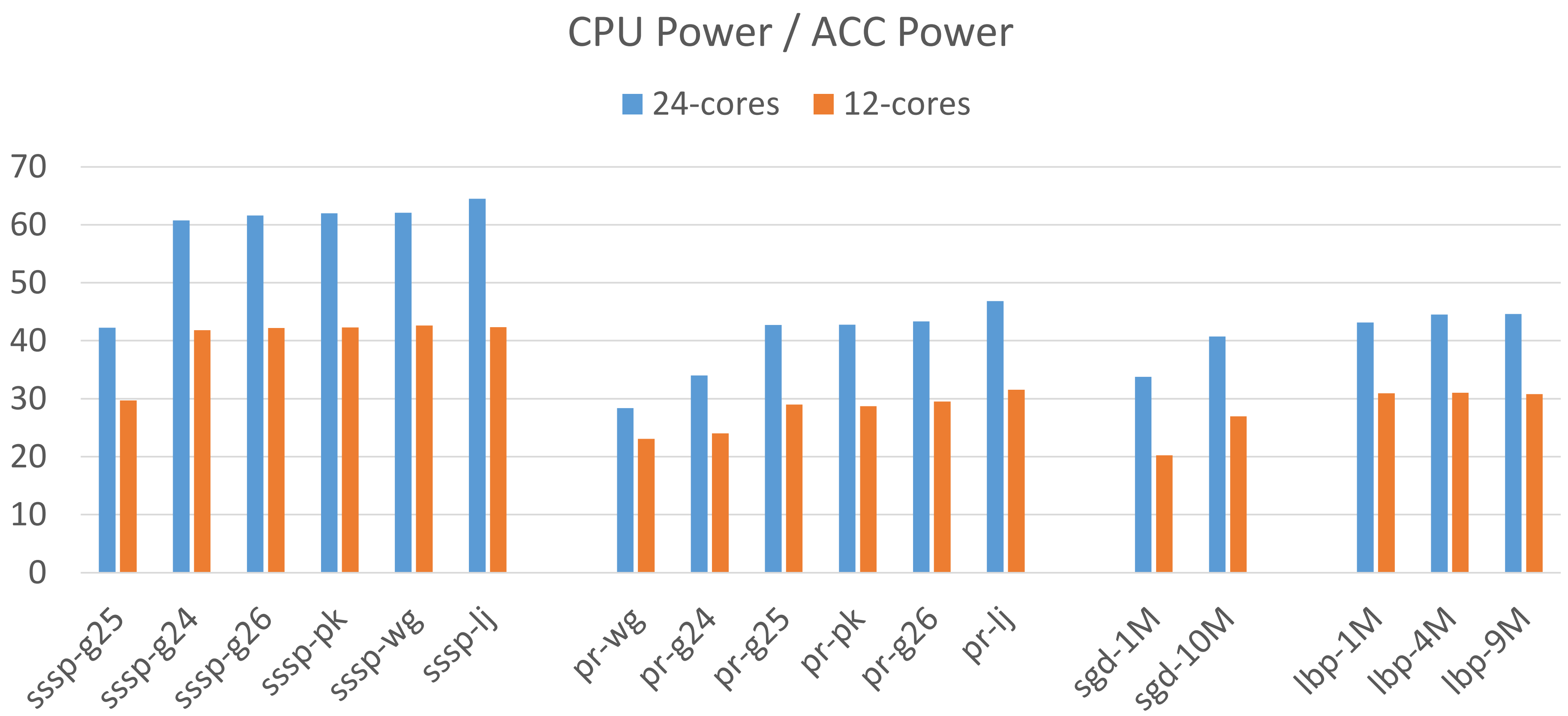
Proposed Accelerator

- *Performance*: Cycle accurate SystemC model + DRAMSim2
- *Accelerator power and area:* HLS + physical-aware logic synthesis with a 22nm industrial library
- *Cache power and area:* CACTI models
- *DRAM power:* in-house DDR4 model

Performance Comparison



Power Comparison



Accelerator power is dominated by DRAM power. Improvements would be ~10x higher without DRAM power

Current & Future Work

- Exploration of benefits of a template design compared to direct application specific implementations of aforementioned applications
  - Template approach proposed in this work outperforms direct HLS in terms of execution time
  - However, direct HLS approach can be more area efficient
- A heuristic for design space exploration
  - A two step optimization algorithm
  - First optimizes $Tput/Area$
  - Then, maximizes $\Delta Tput - \alpha\Delta Area$

Conclusions

- A template architecture for graph-analytics is proposed
  - Latency tolerance for irregular accesses
  - Graph-parallel execution with sequential consistency
  - Asynchronous execution and active vertex set support

- Synthesizable and cycle-accurate SystemC models
  - Different accelerators generated by plugging in app-specific functions
  - Template code size : 39K lines, user code size 43 lines for PageRank

- Experiments with 22nm industrial libraries:
  - Performance comparable with a 24-core Xeon system (except SSSP)
  - Up to 65x less power

Thank you