Efficient Graph Matching and Coloring on the GPU

Jonathan Cohen

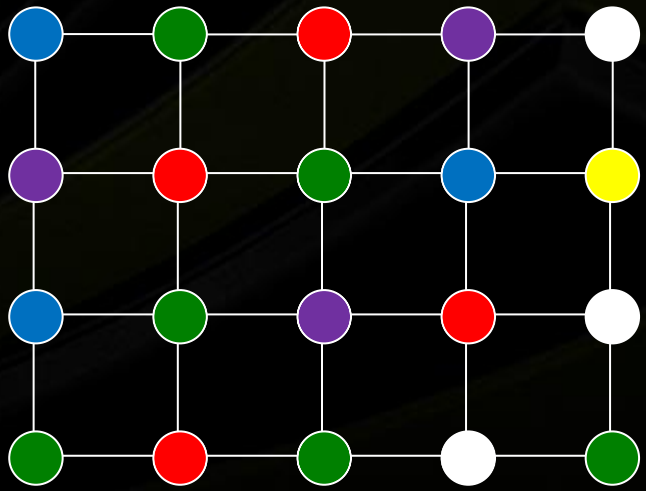
Patrice Castonguay

Key Idea

- Cost model: count global syncs
- Reasoning:
  - One kernel invocation = One global sync
    1) Read graph data
    2) Compute something,
    3) Write results
    4) Wait for all threads to finish (sync)
  - Assume "read graph data" and "wait" (sync) dominate
- Model is too crude today, but leads to algorithms that scale to future trends (and bigger machines)
- Reducing kernel launches generally improves perf
- Conclusion: want coloring and matching algorithms requiring fewest number of kernel launches

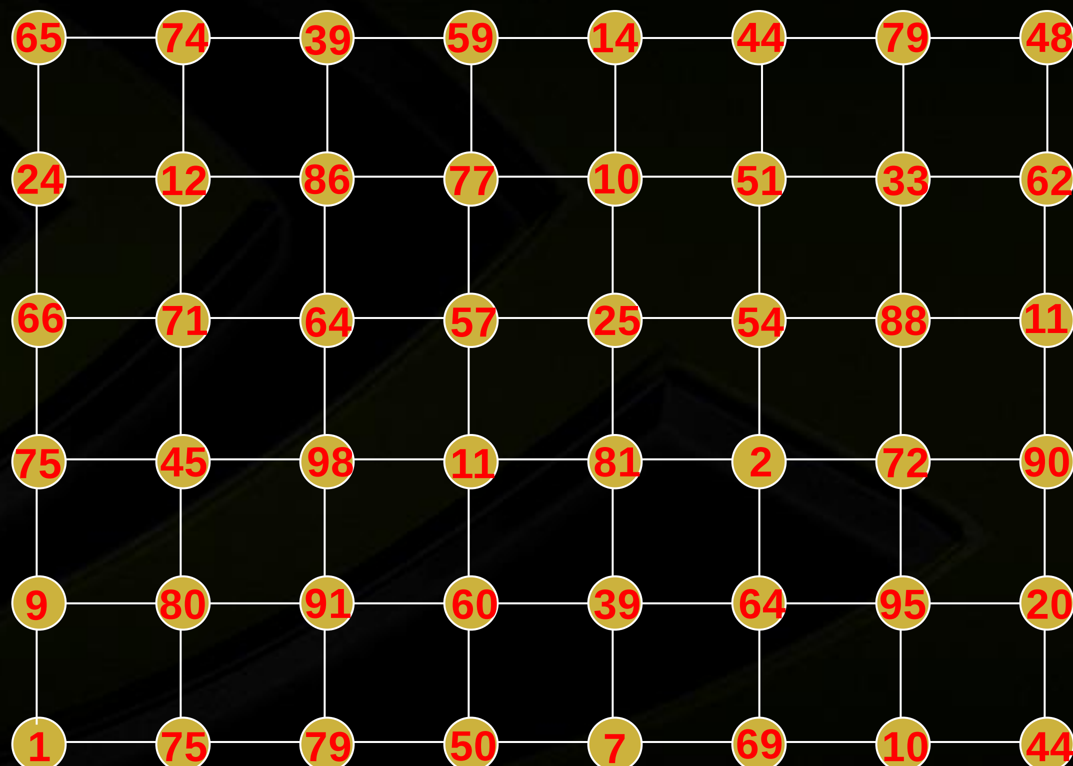
Graph Coloring

- Assignment of “color” (integer) to vertices, with no two adjacent vertices the same color
- Each color forms independent set (conflict-free)
  - reveals parallelism inherent in graph topology
- “inexact” coloring is often ok
- Our focus: fast, cheap, non-optimal colorings



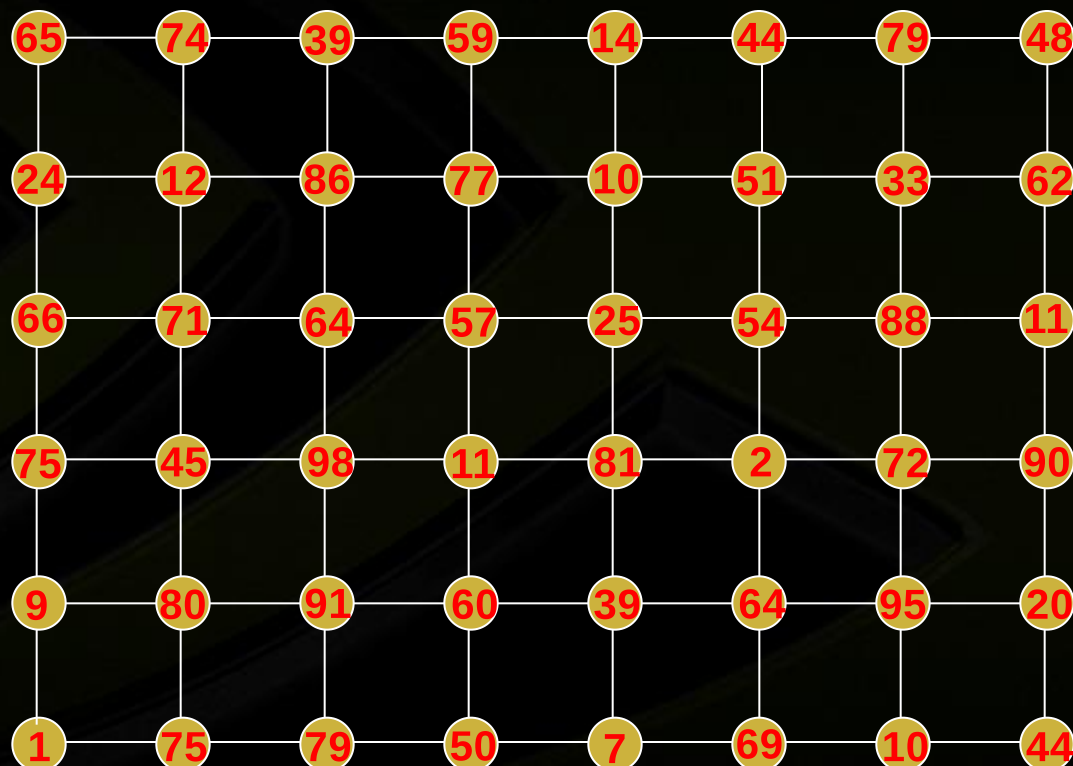
Parallel Graph Coloring – Luby-Jones

- **Parallel graph coloring algorithm of Luby / Jones-Plassman**



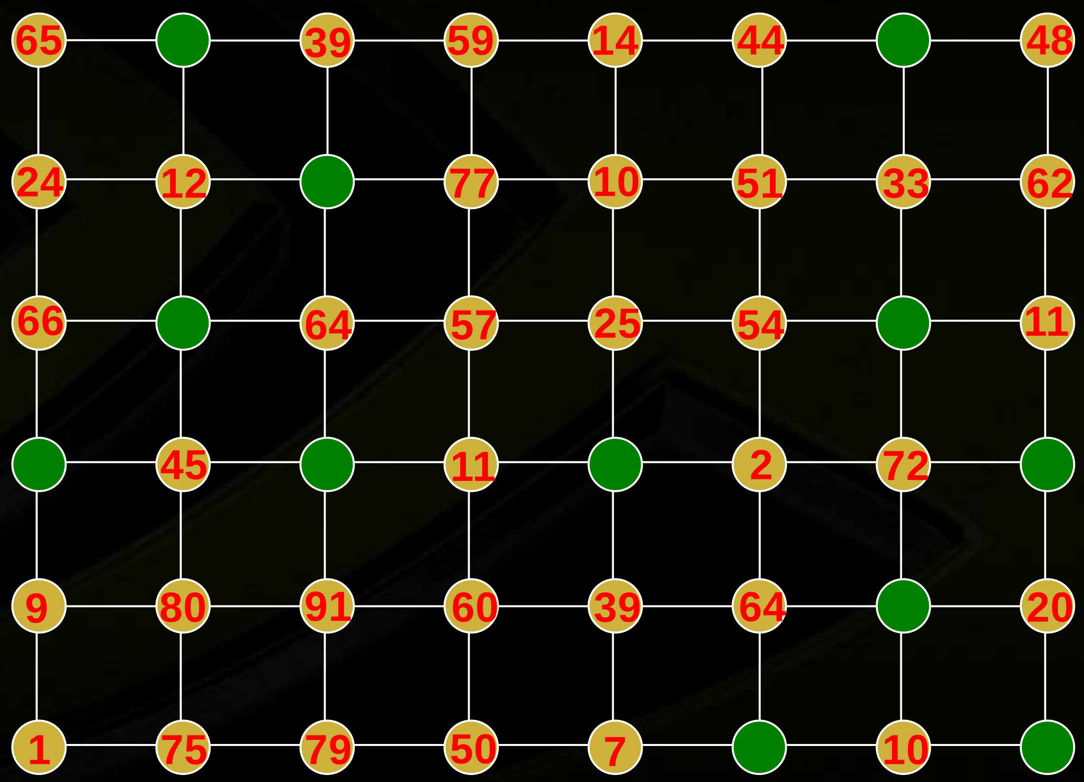
Parallel Graph Coloring – Luby-Jones

- Classic approach: compute array of random numbers
- First optimization: compute a hash function of vertex index on the fly
- Vertex can compute hash number of its neighbors' indices
- Trades bandwidth for compute, skip kernel to assign random numbers



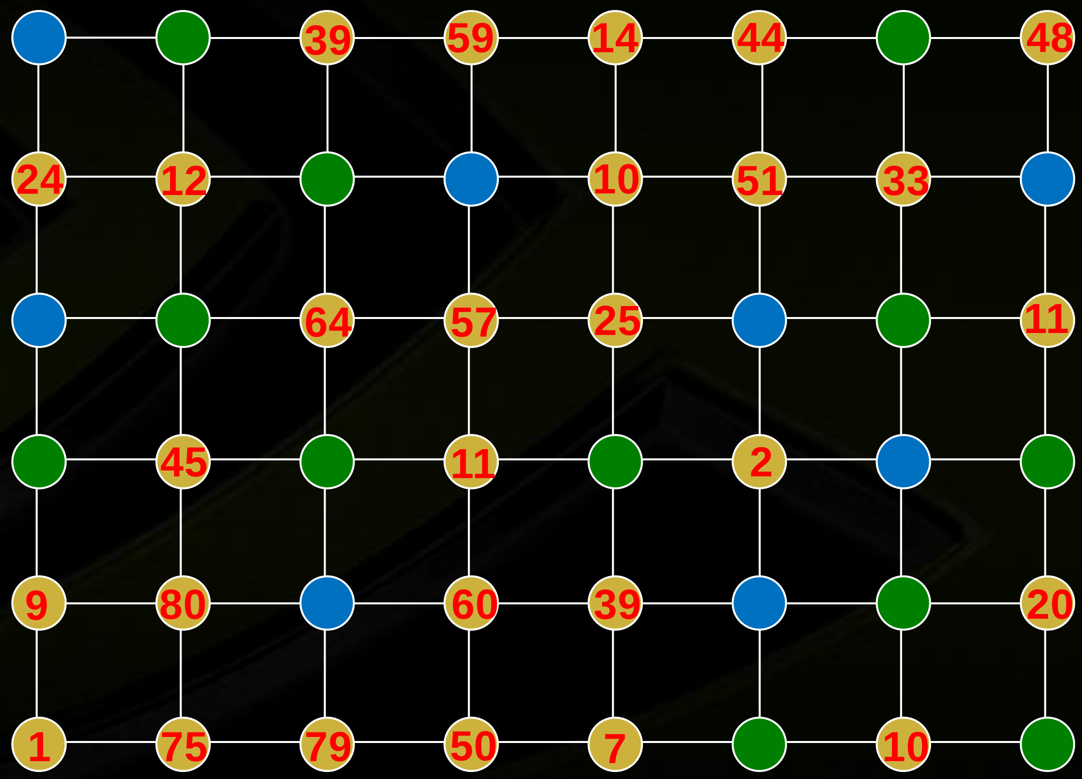
Parallel Graph Coloring – Luby-Jones

- **Round 1: Each vertex checks if local maximum**
- **=> Adjacent vertices can't both be local maxima**
- **If max, color=green.**



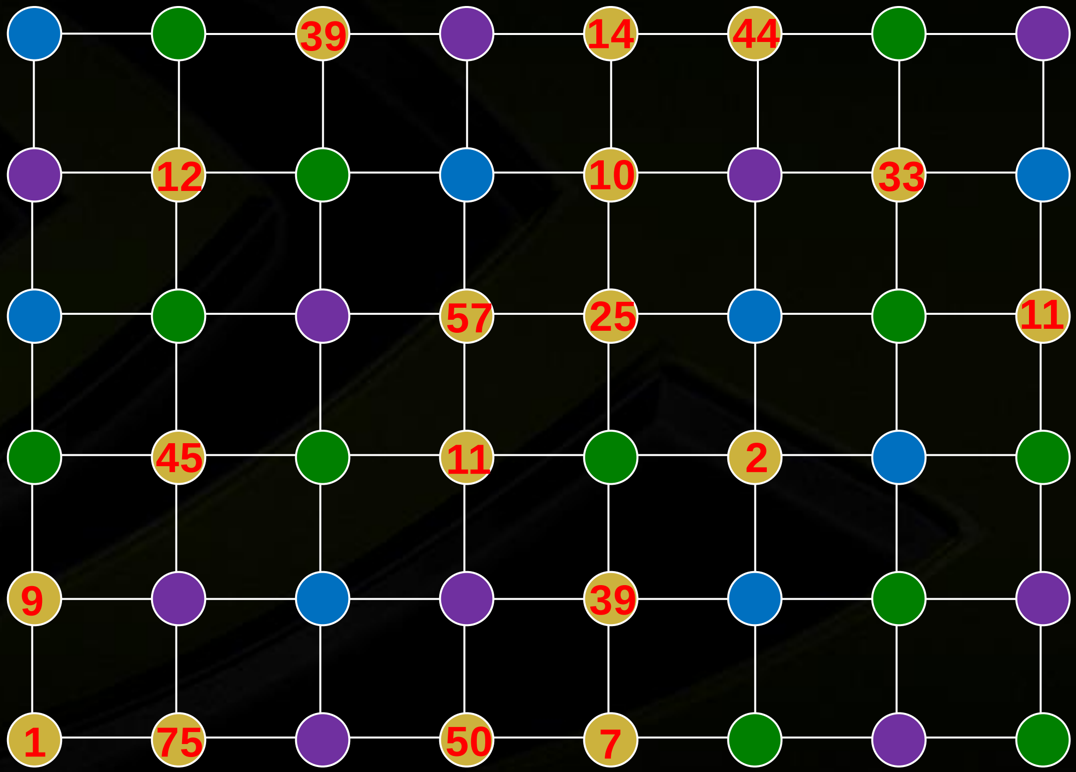
Parallel Graph Coloring – Luby-Jones

- **Round 2: Each vertex checks if local maximum, ignoring green**
- **If max, color=blue**



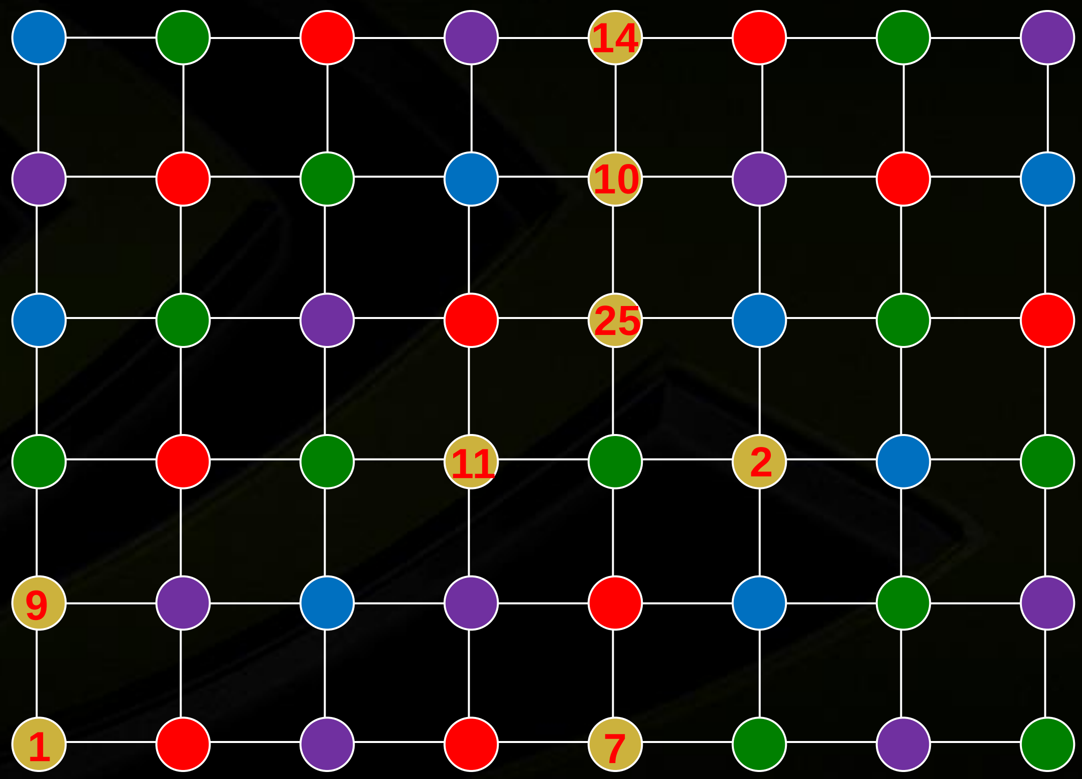
Parallel Graph Coloring – Luby-Jones

- **Round 3: Each vertex checks if local maximum, ignoring colored nbrs**
- **If max, color=purple**



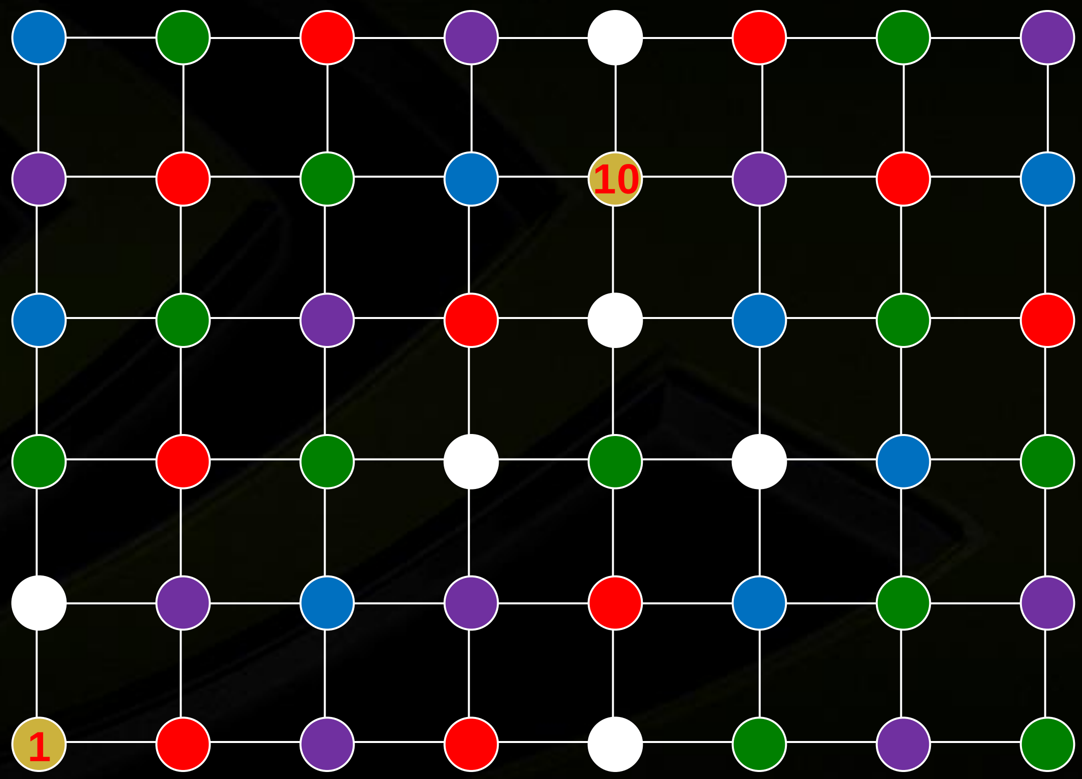
Parallel Graph Coloring – Luby-Jones

- **Round 4: Each vertex checks if local maximum, ignoring colored nbrs**
- **If max, color=red**



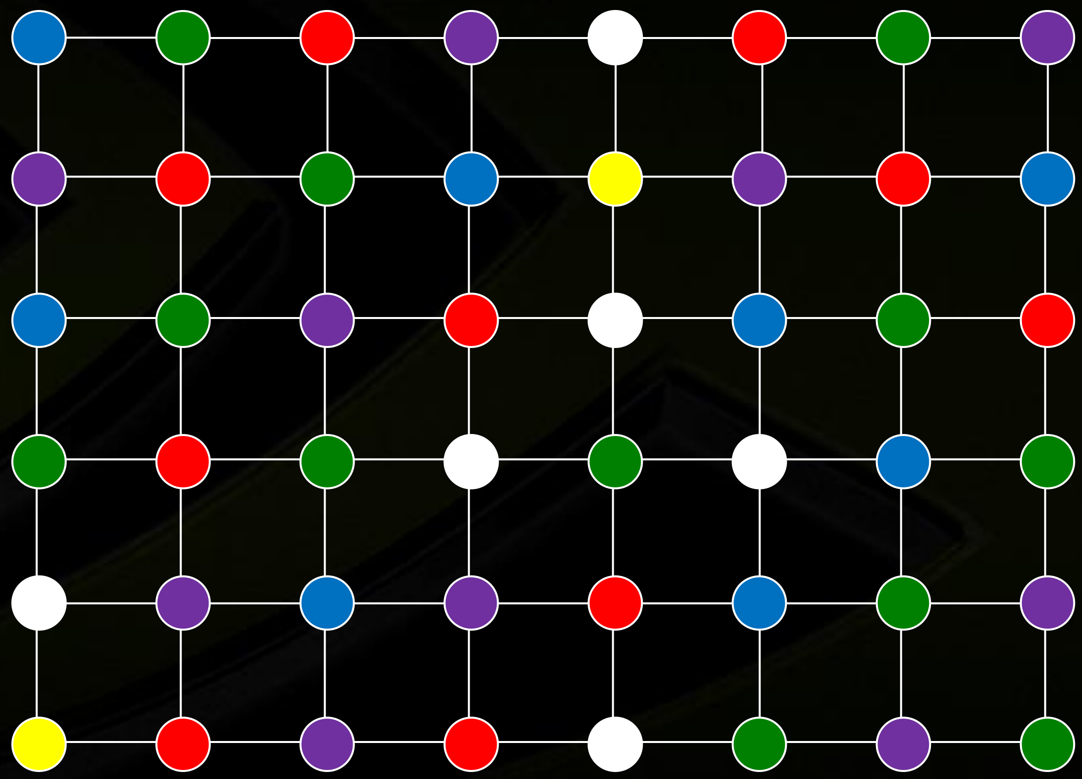
Parallel Graph Coloring – Luby-Jones

- **Round 5: Each vertex checks if local maximum, ignoring colored nbrs**
- **If max, color=white**



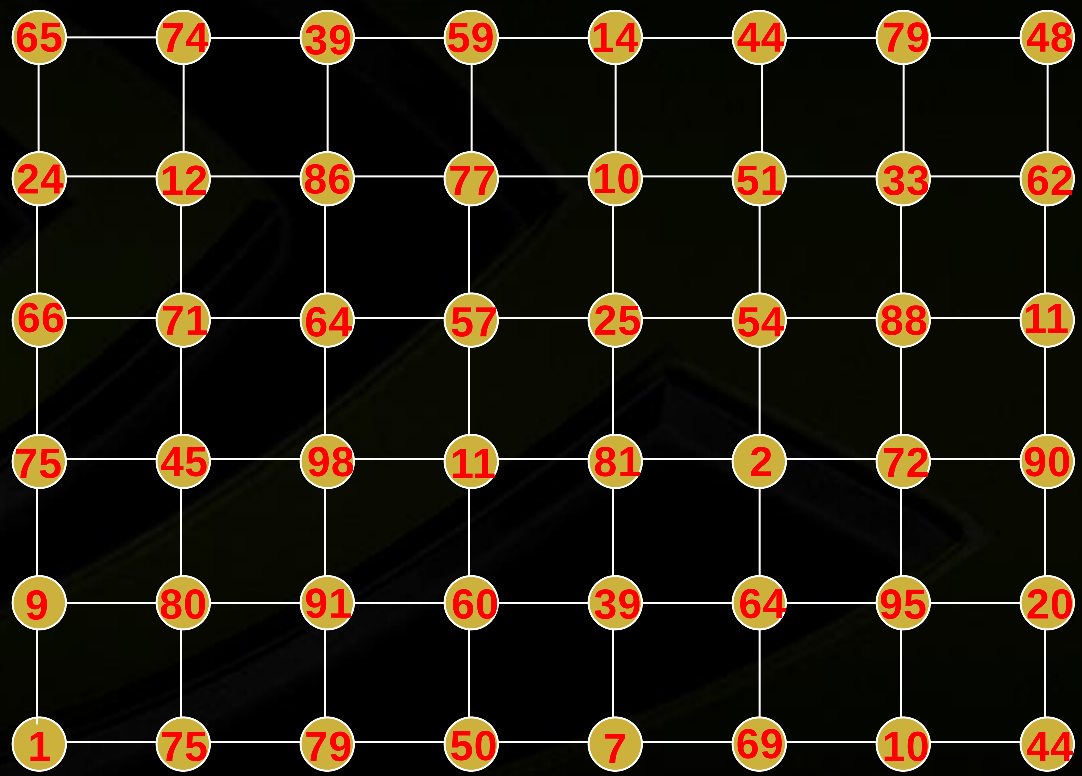
Parallel Graph Coloring – Luby-Jones

- Round 6: Each vertex checks if local maximum, ignoring colored nbrs
- If max, color=yellow
- Completes in 6 rounds



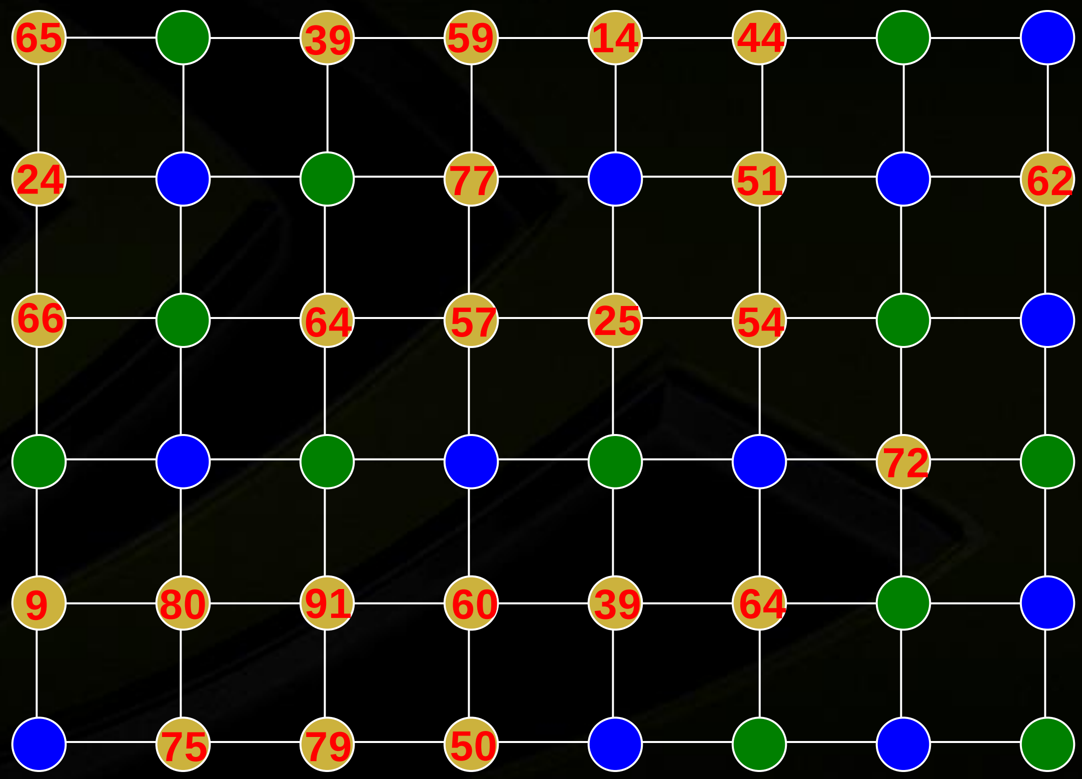
Parallel Graph Coloring – Min-Max

- **Realization: Local min and local max are both independent sets**
- **They are disjoint => can produce 2 colors per iteration**



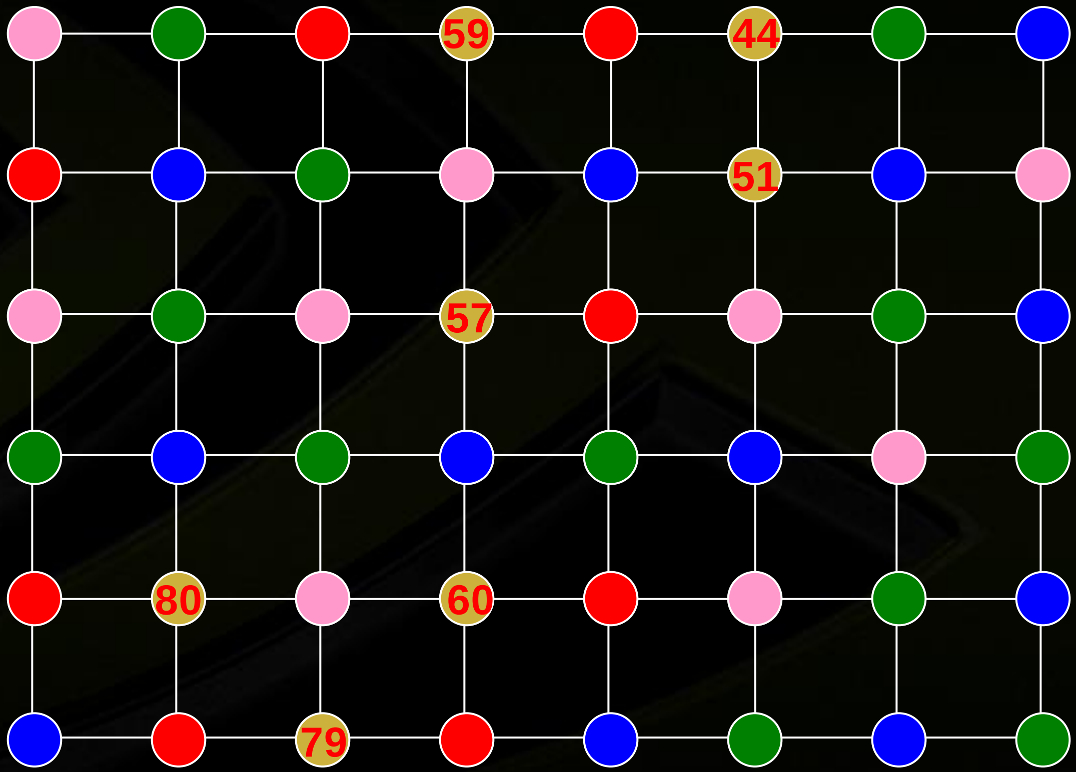
Parallel Graph Coloring – Min/Max

- **Round 1: Each vertex checks if it's a local maximum or minimum.**
- **If max, color=blue. If min, color=green**



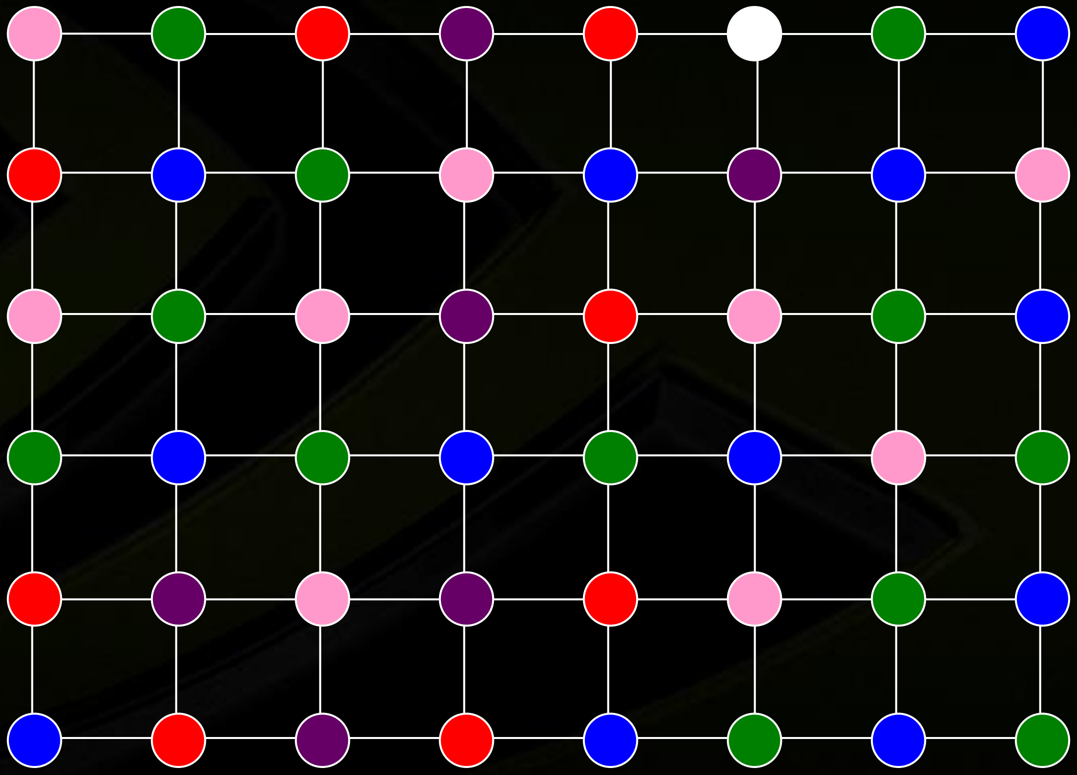
Parallel Graph Coloring – Min/Max

- **Round 2: Each vertex checks if it's a local maximum or minimum.**
- **If max, color=pink. If min, color=red**



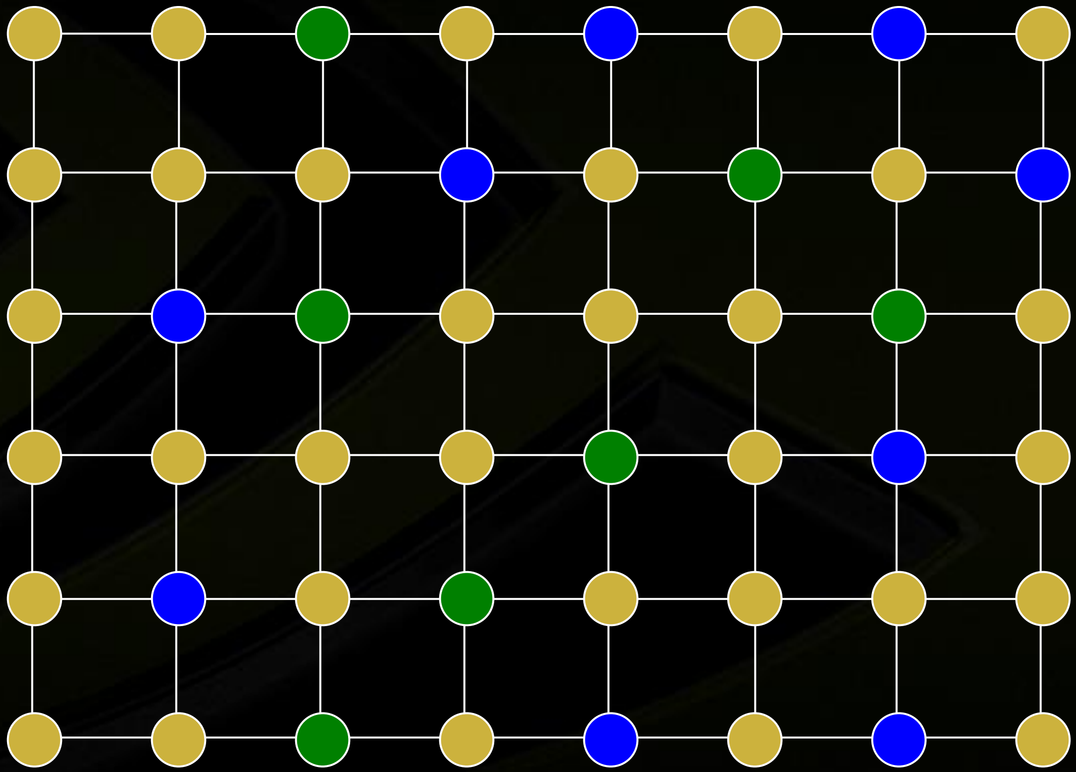
Parallel Graph Coloring – Min/Max

- Round 3: Each vertex checks if it's a local maximum or minimum.
- If max, color=purple. If min, color=white
- Improvement: 3 rounds versus 6



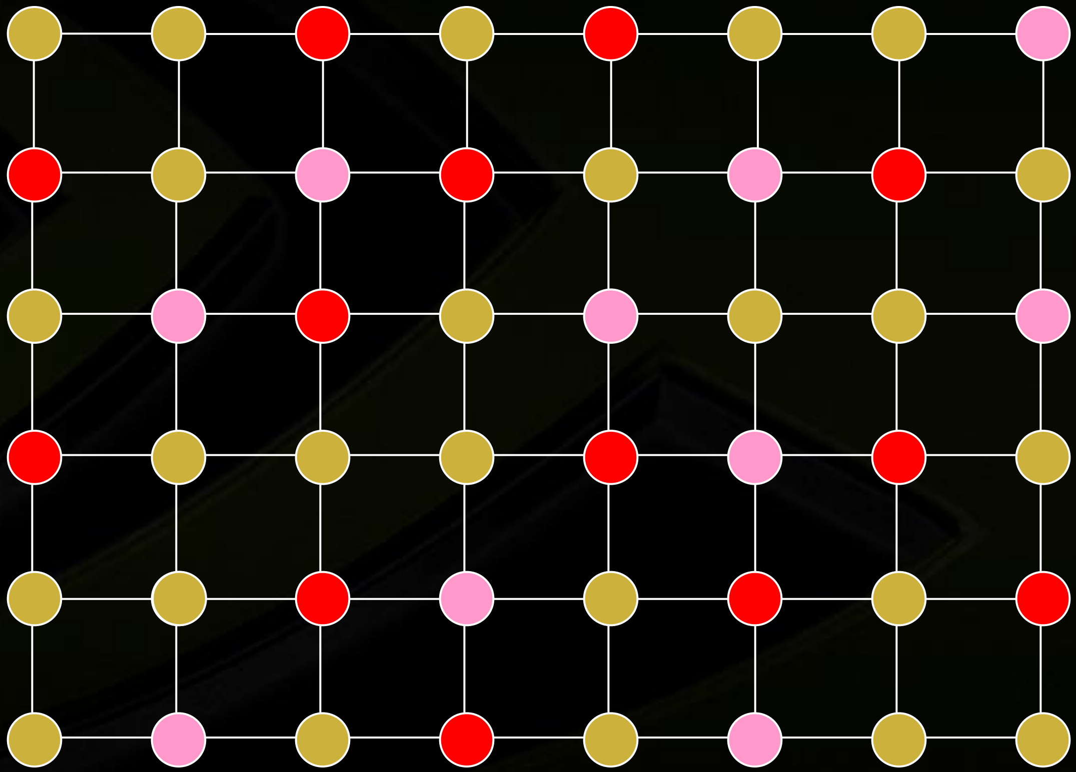
Parallel Graph Coloring – Multi-Hash

- Use multiple hash functions to obtain multiple 2-coloring of the graph
- Hash function 1:



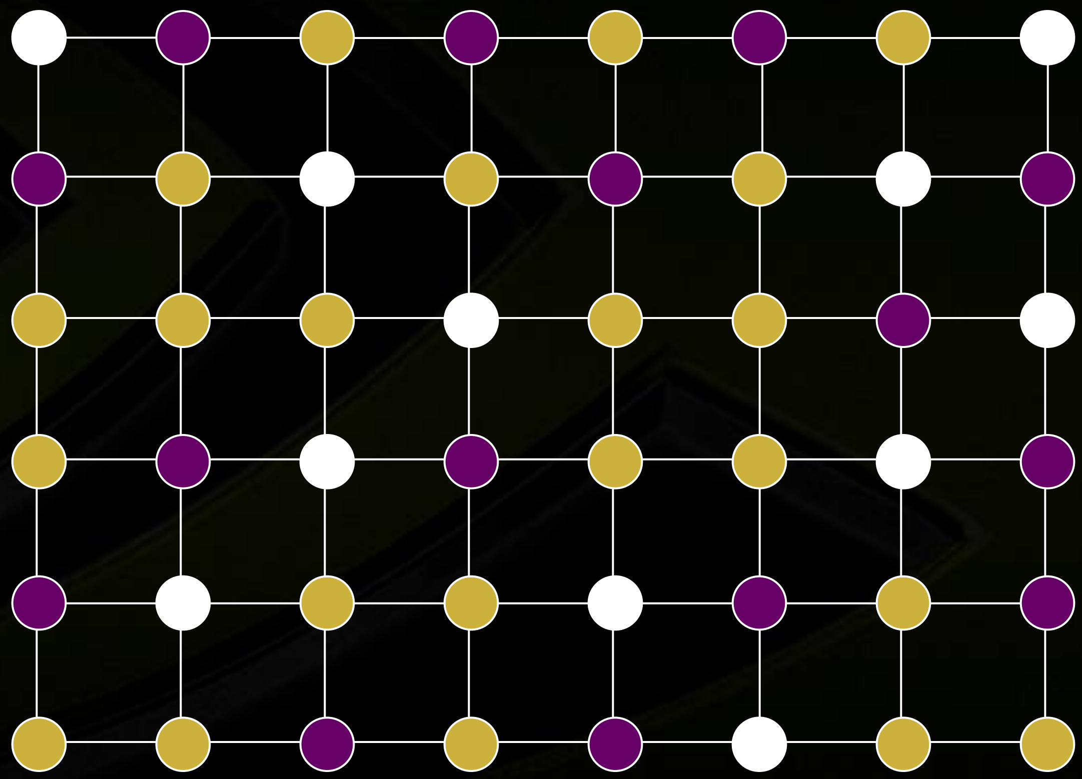
Parallel Graph Coloring – Multi-Hash

- Use multiple hash functions to obtain multiple 2-coloring of the graph
- Hash function 2:



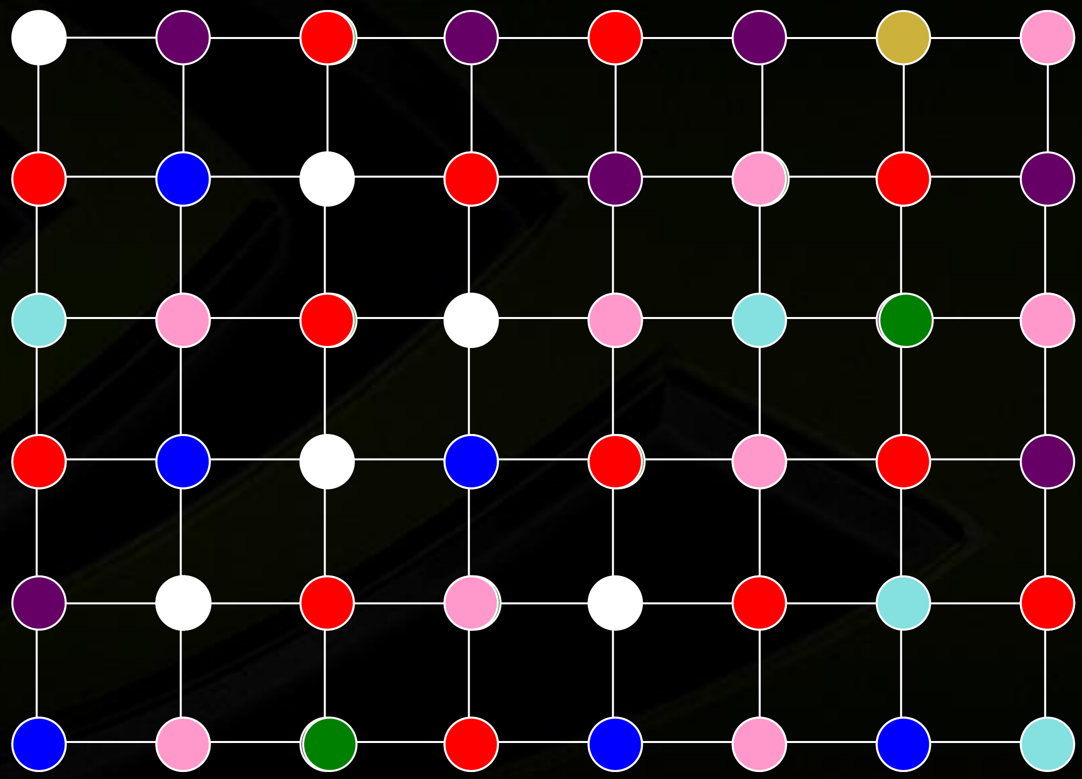
Parallel Graph Coloring – Multi-Hash

- **Use multiple hash functions to obtain multiple 2-coloring of the graph**
- **Hash function 3:**

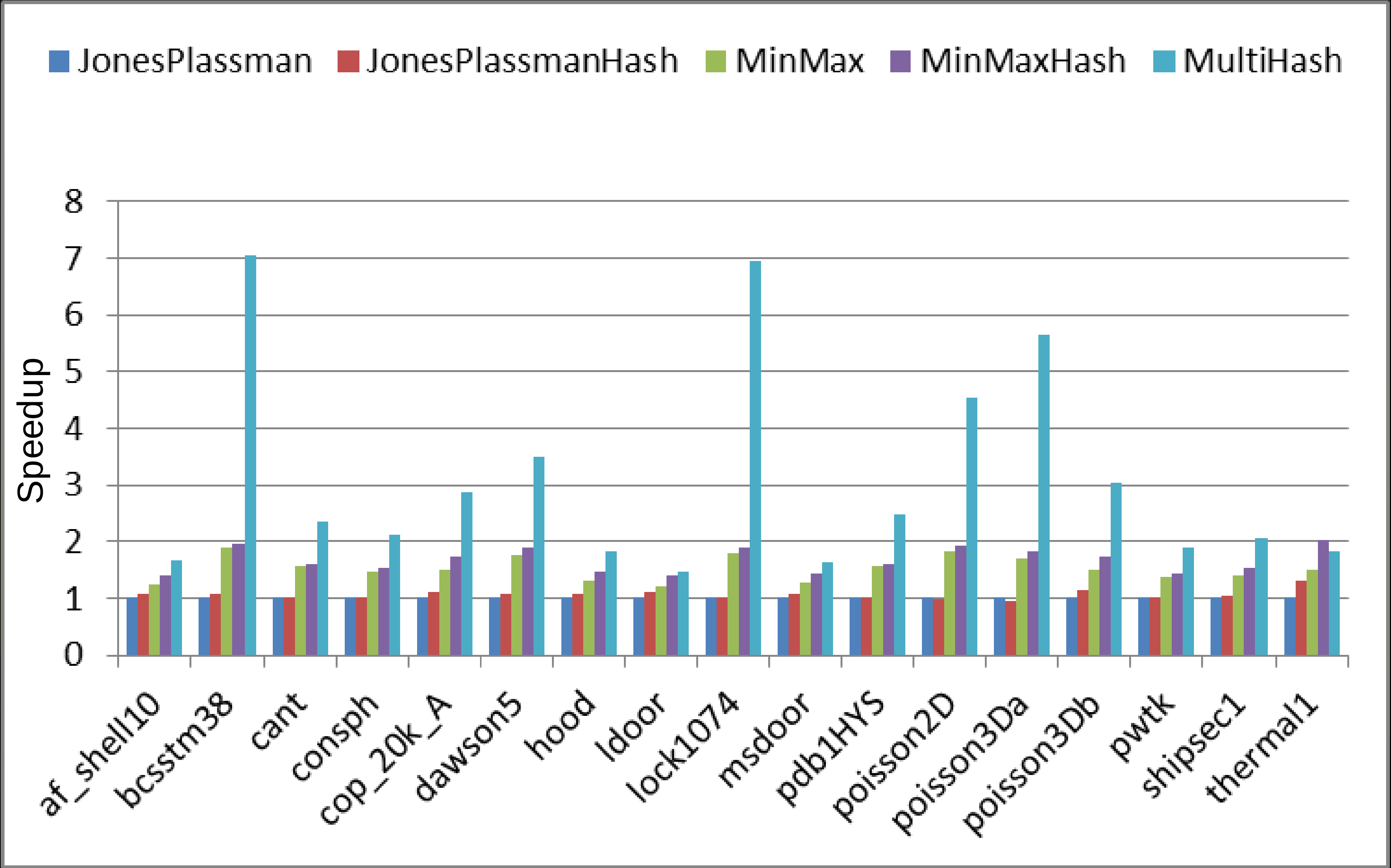


Parallel Graph Coloring – Multi-Hash

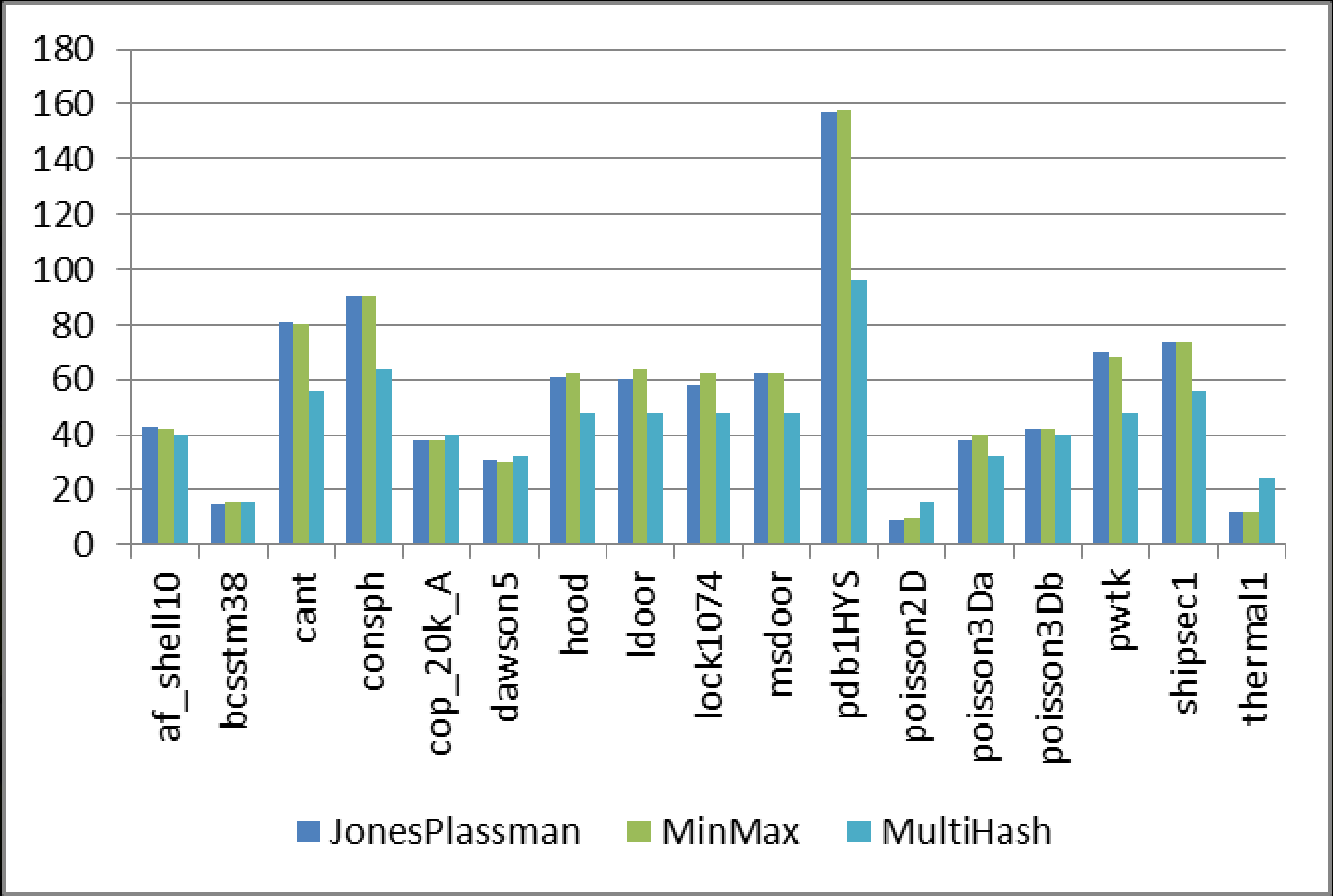
- Combine all 2-colorings – completes in 1 round!
- Creates well-balanced graph colorings
- Empirically: produces better colorings than Luby-Jones – not sure why



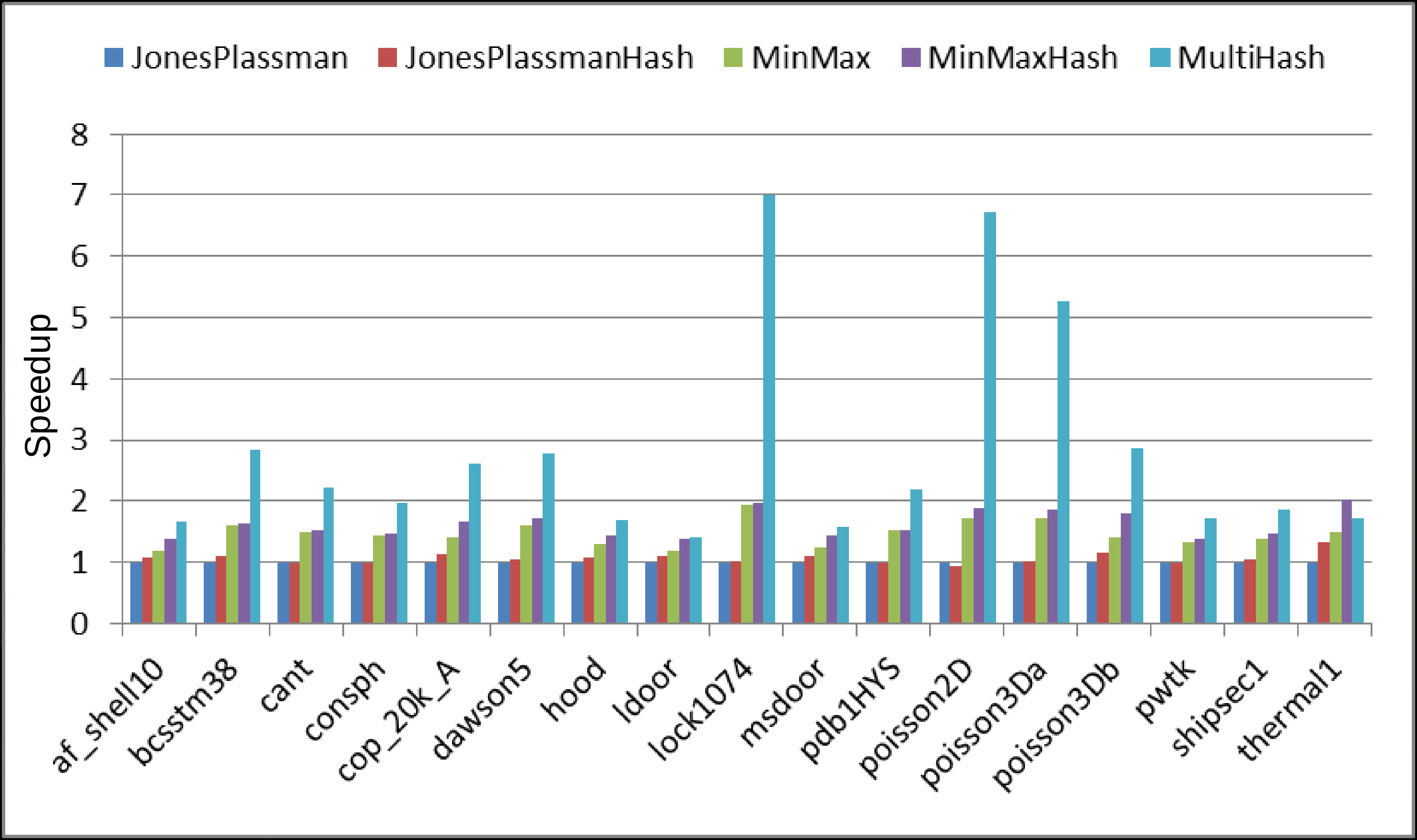
100% Coloring Results



Number of Colors (100% Coloring)



95% Coloring Results



"Root-to-Tip" Parallelism for Linear Solvers on Unstructured Problems

Jonathan Cohen, NVIDIA

The Power Wall - Physics Conquers All

In the past we had constant-field scaling

$L' = L/2$	(feature length)
$V' = V/2$	(voltage)
$E' = \frac{1}{2}CV^2 = E/8$	(capacitance ~ L)
$f' = 2f$	(frequency ~ 1/L)
$A' = L^2 = A/4$	(area)
$P' = P$	(power/area = Ef/A)
$f'/A' = 8\ f/A$	(ops/s/area)

Halve L and get 8x op rate for the same power in fixed area

The Power Wall - Physics Conquers All

Now voltage is held nearly constant

$L' = L/2$	(feature length)
$V' = V$	(voltage)
$E' = \frac{1}{2}CV^2 = E/2$	(capacitance ~ L)
$f' = 2f$	(frequency ~ 1/L)
$A' = L^2 = A/4$	(area)
$P' = 4P$	(power/area = Ef/A)
$f'/A' = 8\ f/A$	(ops/s/area)

Halve L and get 8x op rate for 4x the power in fixed area

Consequences

Power / Area ~ Frequency_Ratio * (Voltage^2 / Length^2)

=> Can control Power / Area by underclocking

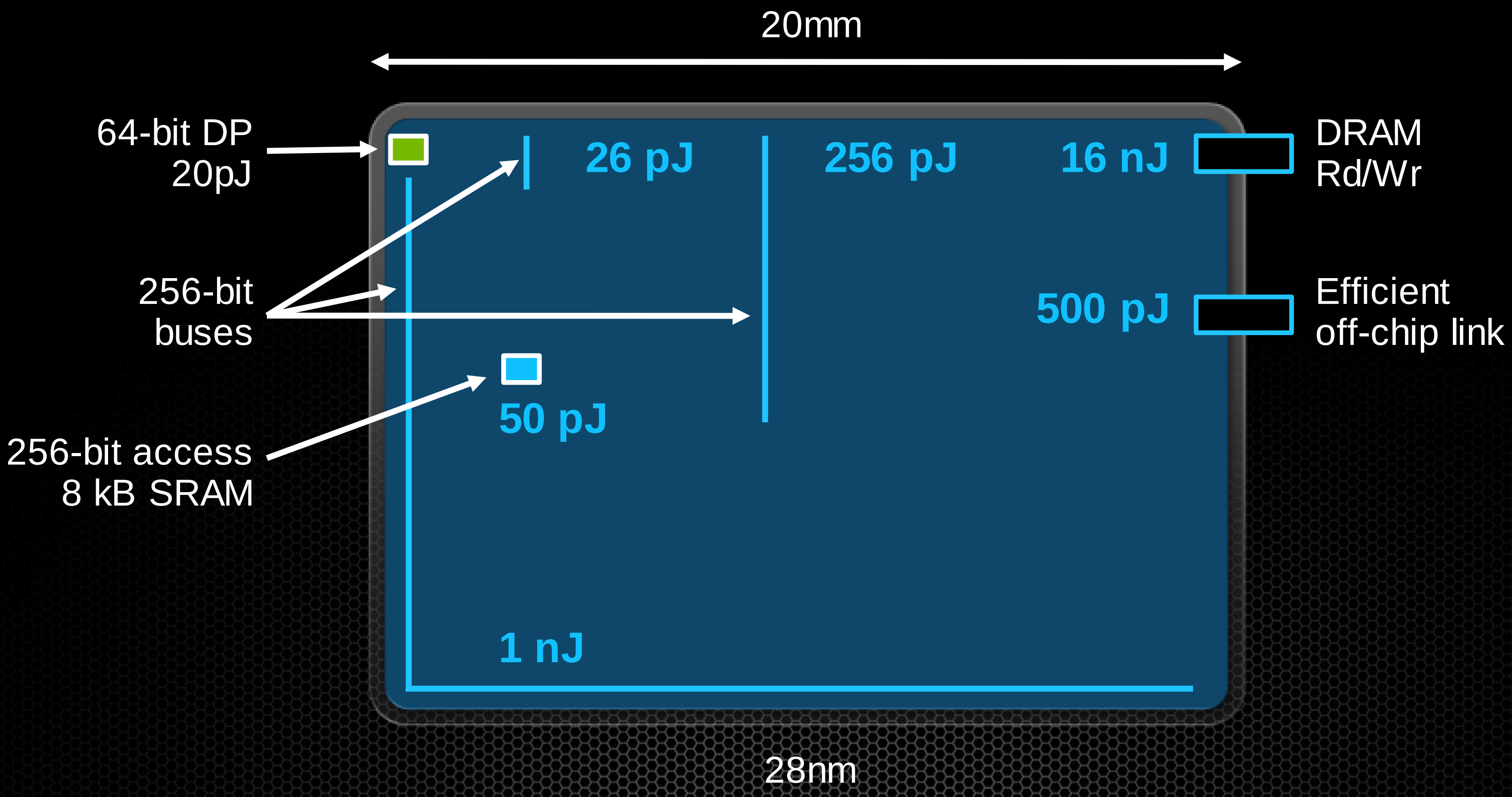
=> Power considerations prevent frequency from scaling as 1/L

=> Amount of available parallelism scales with area = Length^2

=> Exponentially increasing parallelism (throughput)
Slowly increasing frequency (latency)

The High Cost of Data Movement

Fetching operands costs more than computing on them



Thread Count in the Exascale

	2010: 4640 GPUs	~2018: 90K GPUs
Threads/SM	1.5 K	$O(10^3)$
Threads/GPU	21 K	$O(10^5)$
Threads/Cabinet	672 K	$O(10^7)$
Threads/Machine	97 M	$O(10^9)$ - $O(10^{10})$

Billion-fold parallel fine-grained threads for Exascale

Note that “threads / SM” doesn’t change much

Laws of Physics Apply to Everyone…

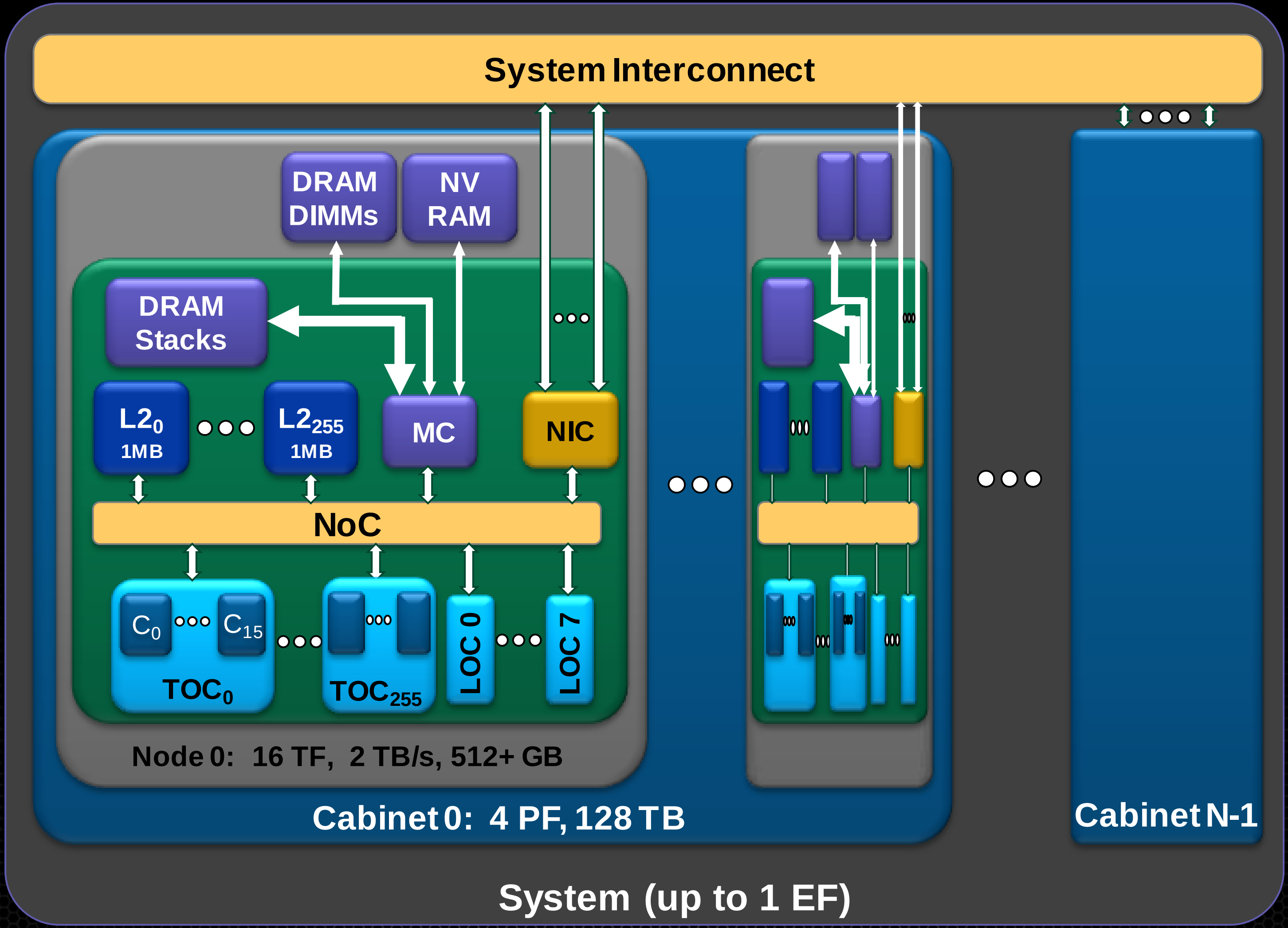
- CPUs & GPUs both need significant parallelism
  - Whether memory- or arithmetic-bound
  - CPUs need 100s ways of parallelism (~10x more than 6 years ago)
  - GPUs/Phi need 1000s ways of parallelism

  Intel Sandybridge: 8 cores x 2 threads x 8 SIMD lanes x a few FP pipes

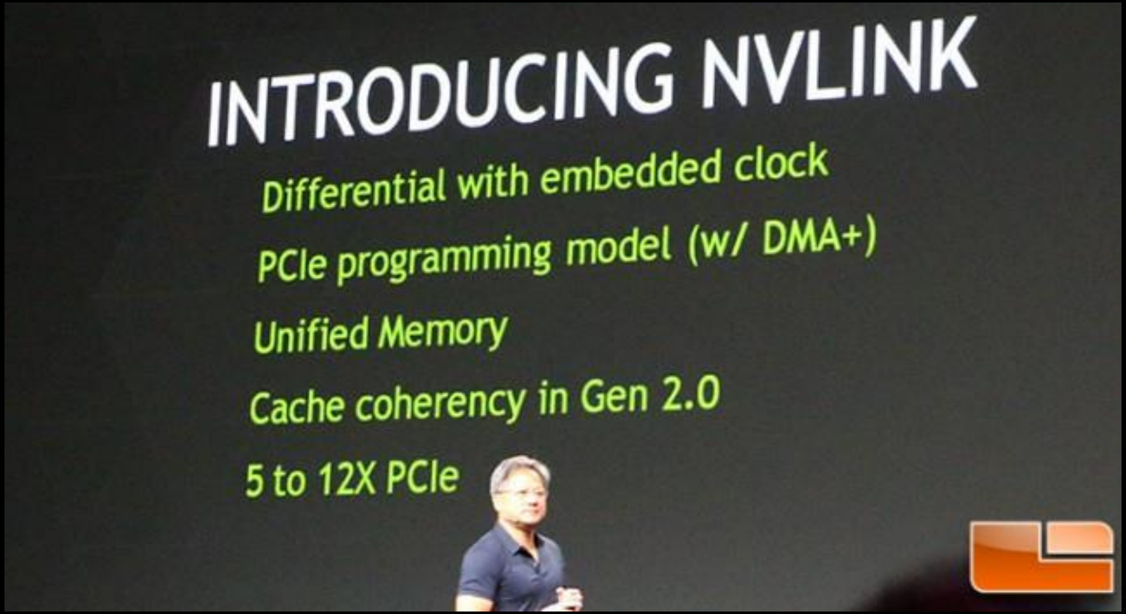
  Intel Xeon Phi: 60 cores x 4 threads x 16 SIMD lanes

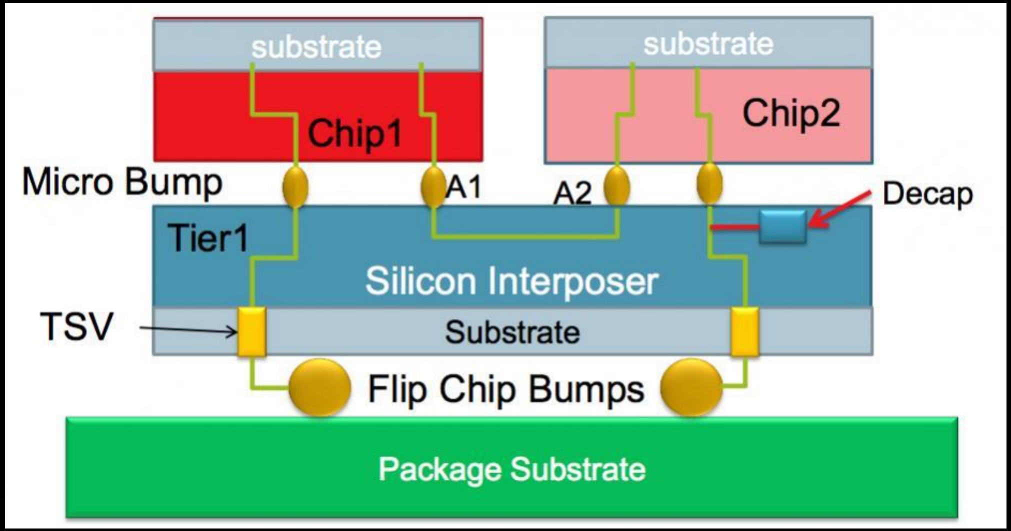
  NVIDIA Kepler: 15 SMs x 64 warps x 32 SIMT lanes x 6-way issue

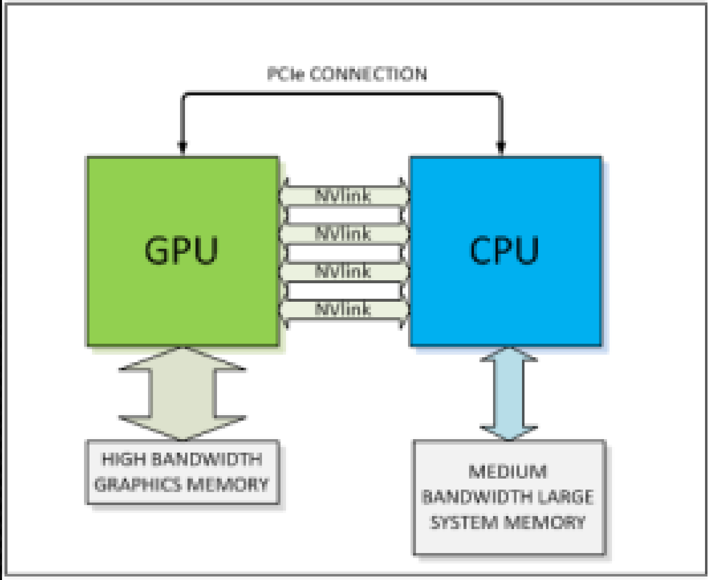
- Legacy CPU codes underutilize CPUs - need rewriting
  - Single-thread, non-vectorized, naïve access patterns

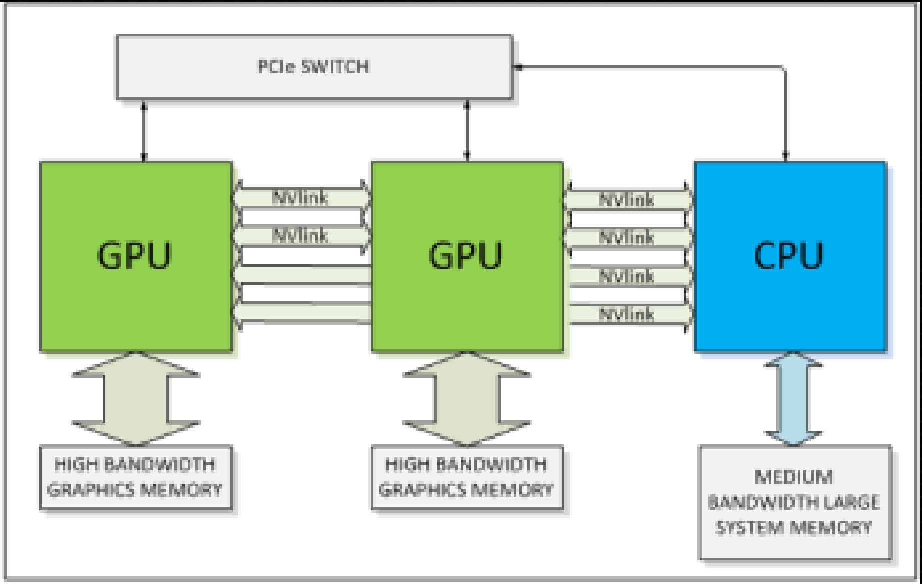


System Interconnect
DRAM DIMMs
NV RAM
DRAM Stacks
L2$_0$ 1MB
L2$_{255}$ 1MB
MC
NIC
NoC
C$_0$
C$_{15}$
TOC$_0$
TOC$_{255}$
LOC 0
LOC 7
Node 0: 16 TF, 2 TB/s, 512+ GB
Cabinet 0: 4 PF, 128 TB
Cabinet N-1
System (up to 1 EF)


INTRODUCING NVLINK
Differential with embedded clock
PCIe programming model (w/ DMA+)
Unified Memory
Cache coherency in Gen 2.0
5 to 12X PCIe


substrate
Chip1
substrate
Chip2
Micro Bump
A1
A2
Decap
Tier1
Silicon Interposer
TSV
Substrate
Flip Chip Bumps
Package Substrate


PCIe CONNECTION
GPU
NVlink
NVlink
NVlink
NVlink
CPU
HIGH BANDWIDTH GRAPHICS MEMORY
MEDIUM BANDWIDTH LARGE SYSTEM MEMORY


PCIe SWITCH
GPU
NVlink
NVlink
GPU
NVlink
NVlink
NVlink
NVlink
CPU
HIGH BANDWIDTH GRAPHICS MEMORY
HIGH BANDWIDTH GRAPHICS MEMORY
MEDIUM BANDWIDTH LARGE SYSTEM MEMORY

Implication for Linear Solvers

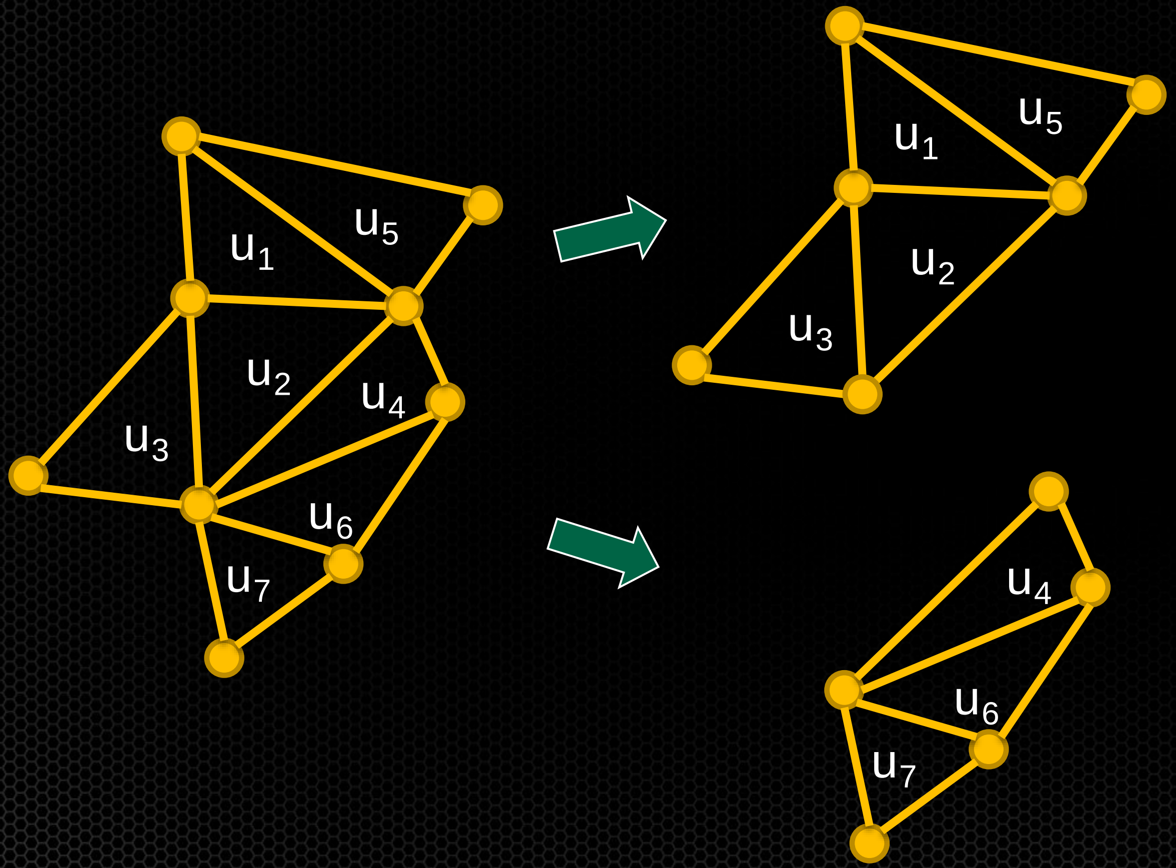
Domain decomposition-based parallelism isn't enough

Will method scale to the limit as each domain => single unknown?

Multiplicative parallelism

- Coarse grain: decompose into domains
- Fine grain: parallelize *everything* within each domain

Root: Domain Decomposition



$$\begin{pmatrix} A_{11} & A_{12} & & A_{15} \\ A_{21} & A_{22} & A_{23} & \\ & A_{32} & A_{33} & \\ A_{51} & & & A_{55} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ u_5 \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \\ R_3 \\ R_5 \end{pmatrix}$$

$$\begin{pmatrix} A_{44} & A_{46} & \\ A_{64} & A_{66} & A_{67} \\ & A_{76} & A_{77} \end{pmatrix} \begin{pmatrix} u_4 \\ u_6 \\ u_7 \end{pmatrix} = \begin{pmatrix} R_4 \\ R_6 \\ R_7 \end{pmatrix}$$

Tip: Smoother Within Each Domain

Example: Highly parallel Incomplete LU

Approximate $A \approx LU$, L lower triangular, U upper triangular

$Ax = b \rightarrow LUx = b \rightarrow$ solve for x with 2 triangular solves

Use it as a preconditioner

Use it as a smoother

Common sense tells us: triangular solve is serial

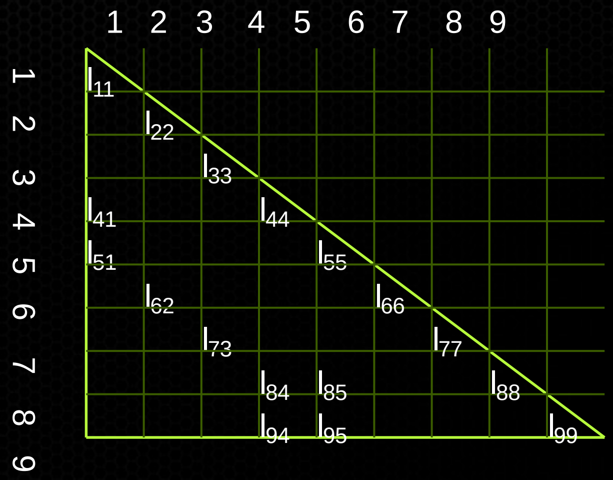
Parallel Incomplete-LU

- ➢ Level scheduling ⇔ implicit reordering
  - ✓ Solve the same linear system $A\,x = f$,
    but reorder A so that the rows in the same level are adjacent
  - ✓ ILU preconditioner computed for the original A
  - ✓ Can improve the memory access pattern
  - ✓ Does not affect convergence

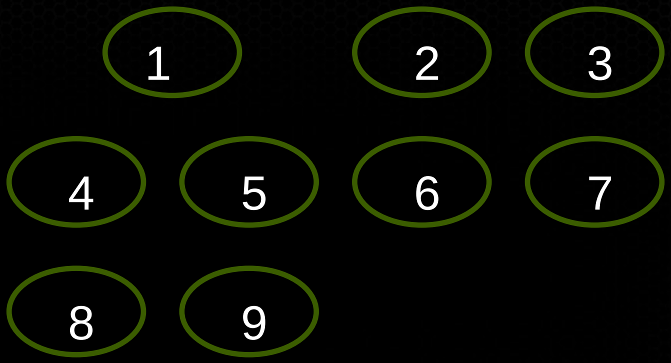
- ➢ Graph coloring ⇔ explicit reordering
  - ✓ Solve $(P^T A\, Q)\,(Q^T x) = P^T f$,
    where P and Q are permutation matrices
  - ✓ ILU preconditioner computed on the permuted $P^T A\, Q$
  - ✓ Can significantly increase parallelism
  - ✓ Can adversely affect convergence

Level Scheduling: Example

matrix sparsity pattern



directed acyclic graph (DAG)



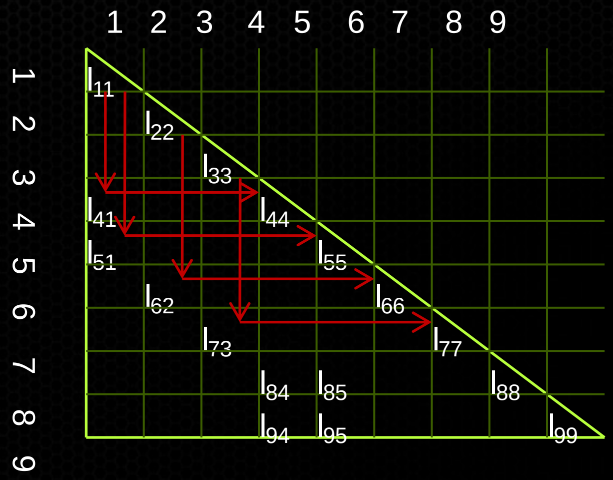
Level Ptr 1

Level Index 1 2 3

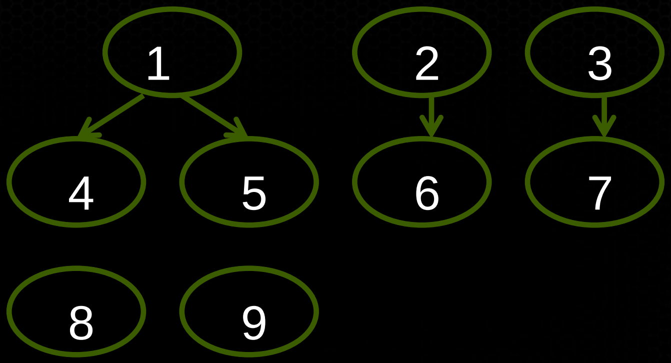
Level/Depth 1

Level Scheduling: Example

matrix sparsity pattern



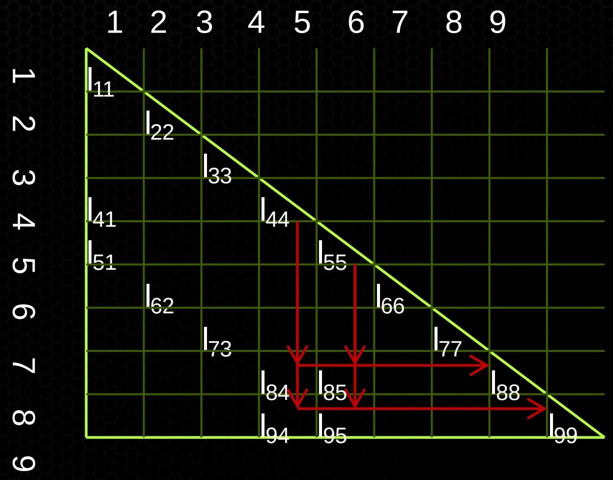
directed acyclic graph (DAG)



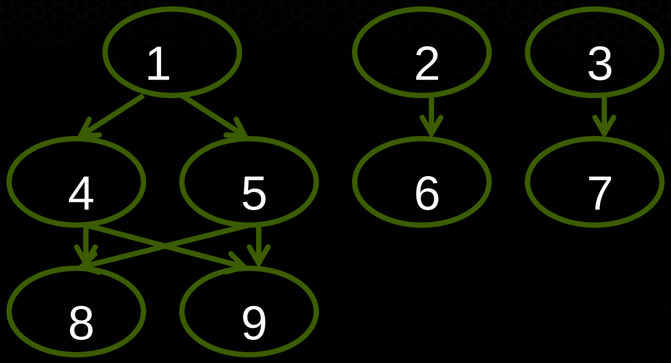
Level Ptr	1	4
Level Index	1 2 3	4 5 6 7
Level/Depth	1	2

Level Scheduling: Example

matrix sparsity pattern



directed acyclic graph (DAG)



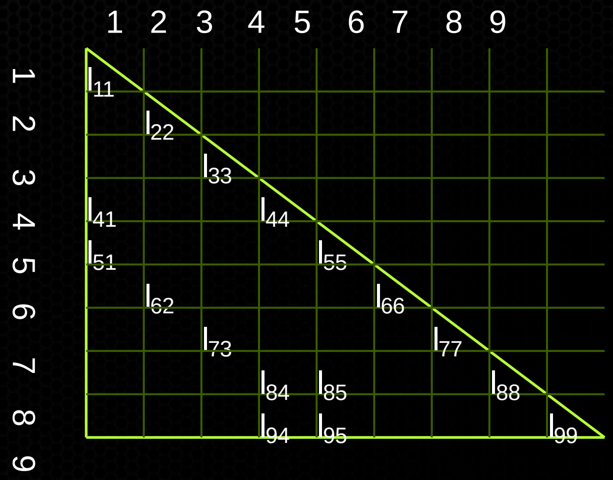
Level Ptr 1 4 8

Level Index 1 2 3 | 4 5 6 7 | 8 9

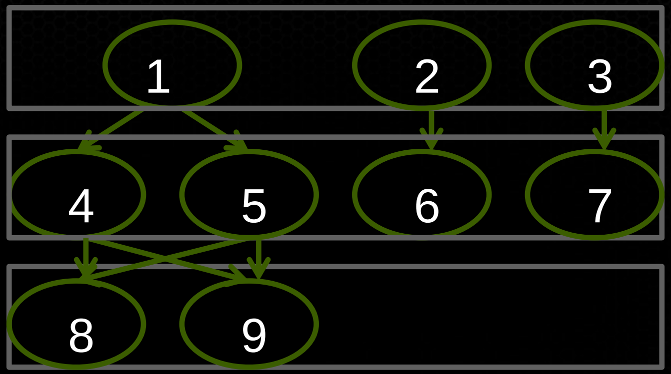
Level/Depth 1 2 3

Level Scheduling: Example

matrix sparsity pattern



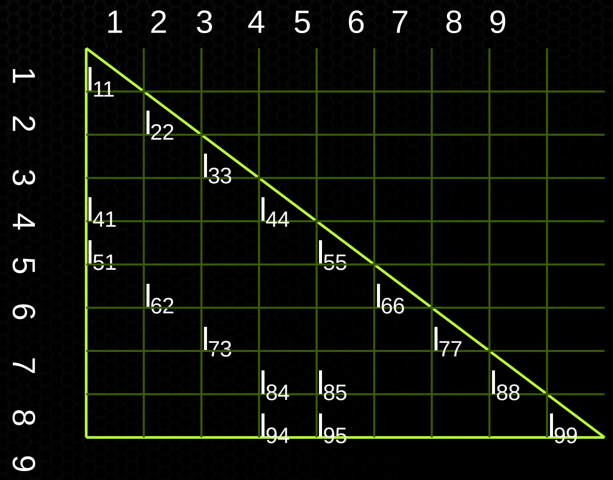
directed acyclic graph (DAG)



Level Ptr	1	4	8	10
Level Index	1 2 3	4 5 6 7	8 9	
Level/Depth	1	2	3	

Graph Coloring: Example

matrix sparsity pattern

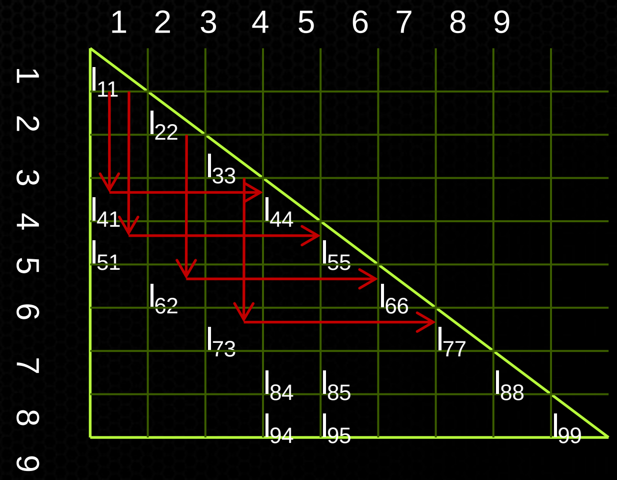


Graph Coloring

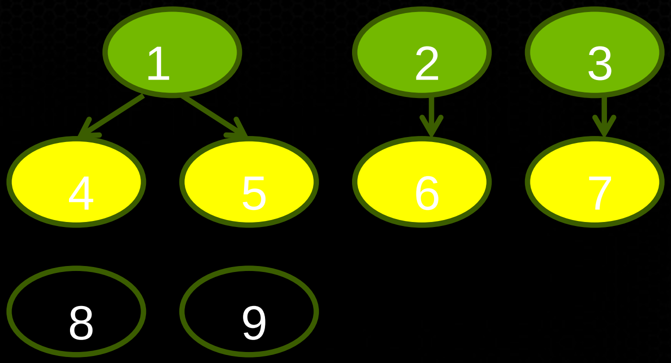
Node/Color 1 2 3 | 4 5 6 7 | 8 9

Graph Coloring: Example

matrix sparsity pattern



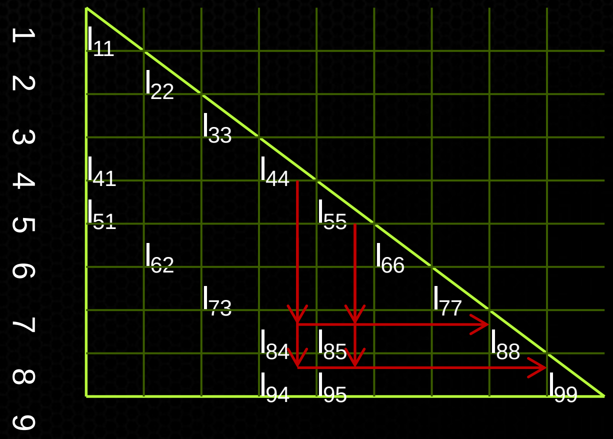
Graph Coloring



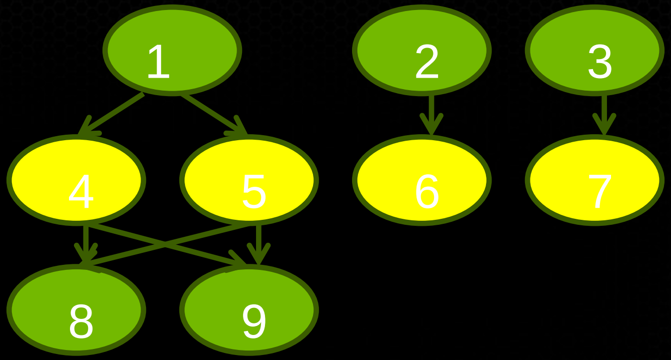
Node/Color 1 2 3 | 4 5 6 7 | 8 9

Graph Coloring: Example

matrix sparsity pattern



Graph Coloring

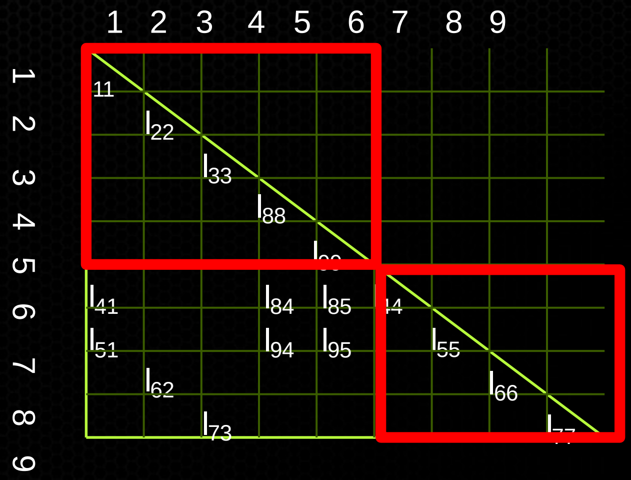


Node/Color 1 2 3 | 4 5 6 7 | 8 9

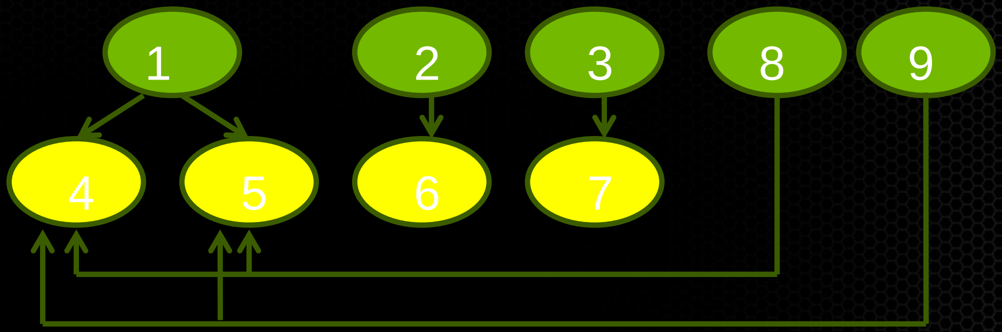
Permutation 1 2 3 8 9 4 5 6 7

Graph Coloring: Example

matrix sparsity pattern



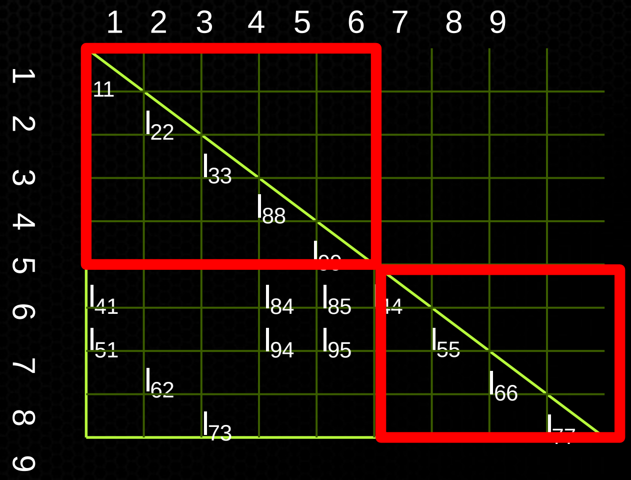
Graph Coloring



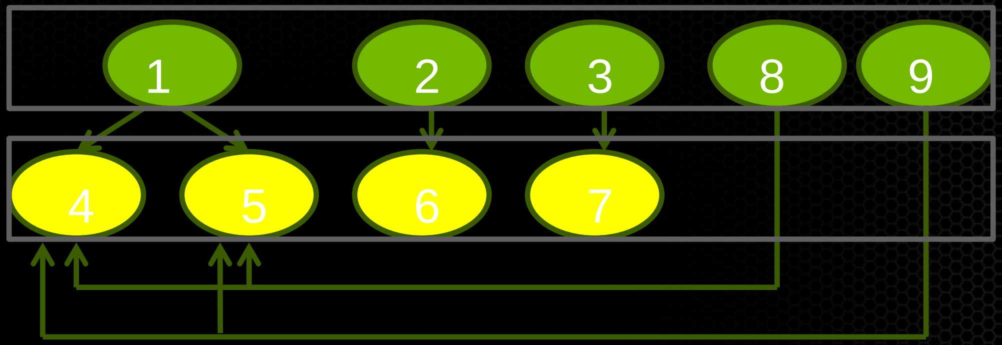
Node/Color	1 2 3 | 4 5 6 7 | 8 9
Permutation	1 2 3 8 9 4 5 6 7

Graph Coloring: Example

matrix sparsity pattern



Graph Coloring



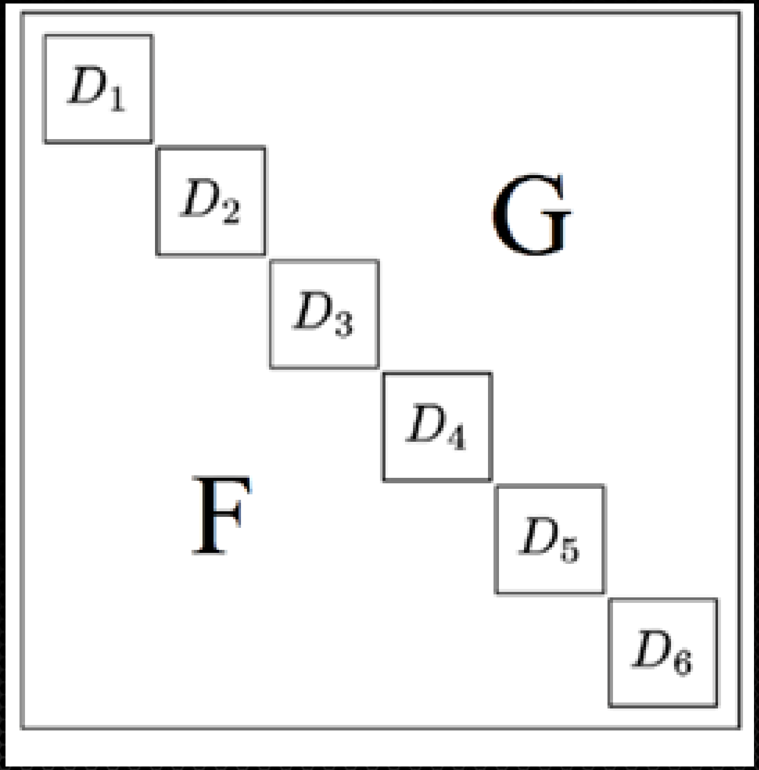
Level Ptr 1 6 10

Level Index 1 2 3 8 9 4 5 6 7

Level/Depth 1 2

Parallelizing the Factorization Step?

- Compute Graph Coloring – yes, many fine-grain parallel algorithms
- Factorization – yes!
- Simplified form: D(iagonal) ILU



Parallel DILU Factorization

- DILU preconditioner has the form

$$M_{DILU} = (E + L)E^{-1}(E + U)$$

- *E* is such that

$$\mathrm{diag}(M_{DILU}) = \mathrm{diag}(A)$$
$$\mathrm{diag}(M_{DILU}) = \mathrm{diag}(E + LE^{-1}U) = \mathrm{diag}(A)$$

- Equivalent to ILU(0) preconditioner for certain matrices
- Only requires one extra diagonal of storage
- Cheap, strong, low-storage

DILU Smoother

- Setup is sequential

$$
\begin{aligned}
E_{11} &= A_{11} \\
E_{22} &= A_{22} - L_{21} E_{11}^{-1} U_{12} \\
E_{33} &= A_{33} - L_{31} E_{11}^{-1} U_{13} - L_{32} E_{22}^{-1} U_{23} \\
E_{44} &= A_{44} - L_{41} E_{11}^{-1} U_{14} - L_{42} E_{22}^{-1} U_{24} - L_{43} E_{33}^{-1} U_{34}
\end{aligned}
$$

- Solve is also sequential (two triangular solve)

$$\Delta = M^{-1}(b - Ax)$$

$$M_{DILU} = (E + L)E^{-1}(E + U)$$

Multi-color DILU Smoother

- Use coloring to extract parallelism
- Setup:

$$
\begin{aligned}
E_{11} &= A_{11} \\
E_{22} &= A_{22} \\
E_{33} &= A_{33} - L_{31}E_{11}^{-1}U_{13} - L_{32}E_{22}^{-1}U_{23} \\
E_{44} &= A_{44} - L_{41}E_{11}^{-1}U_{14} - L_{42}E_{22}^{-1}U_{24}
\end{aligned}
$$

- Forward solve: include contributions from neighbors whose color is less than yours

- Backward solve: include contributions from neighbors whose colors is greater than yours

Graph Coloring

Aggregation AMG - It's All Parallel

SETUP

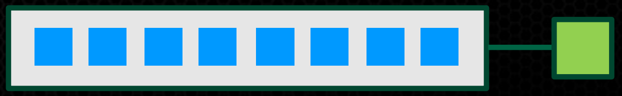
1. Choose aggregates based on A^f	Graph matching, parallel partitioning
2. Construct coarsening operator ($R = P^T$)	tranpose (sort)
3. Construct coarse matrix ($A^c = R\,A^f\,P$)	SpMM
4. Initialize smoother (if needed)	Graph coloring, ILU factorization, etc.

SOLVE

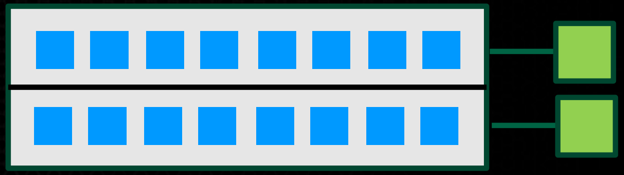
1. Smooth	SpMV / triangular solve
2. Compute residual ($r^f = b^f - A^f\,x^f$)	SpMV
3. Restrict residual ($R\,r^f = r^c$)	SpMV
4. Recurse on coarse problem	
5. Prolongate correction ($x^f = x^f + Pe^c$)	SpMV
6. Smooth	SpMV / triangular solve
7. If not converged, goto 1	reduction

Desired Node Configurations

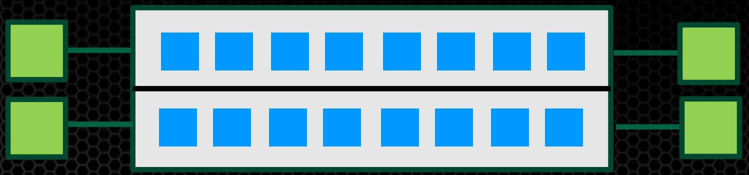
1 CPU socket <=> 1 GPU



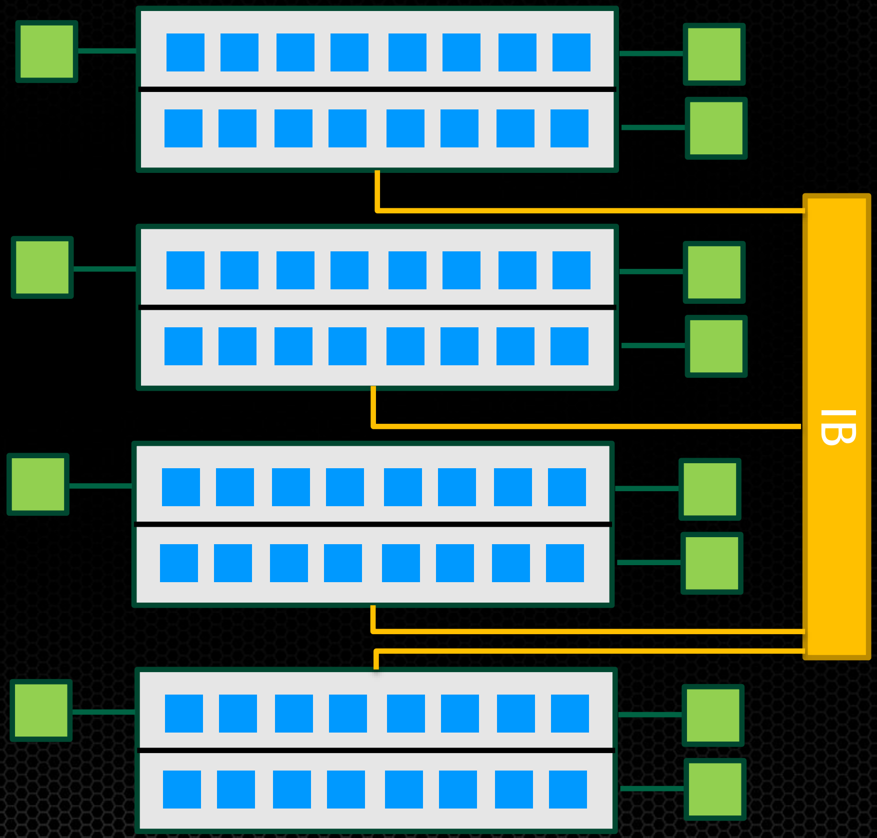
Dual socket CPU <=> 2 GPUs

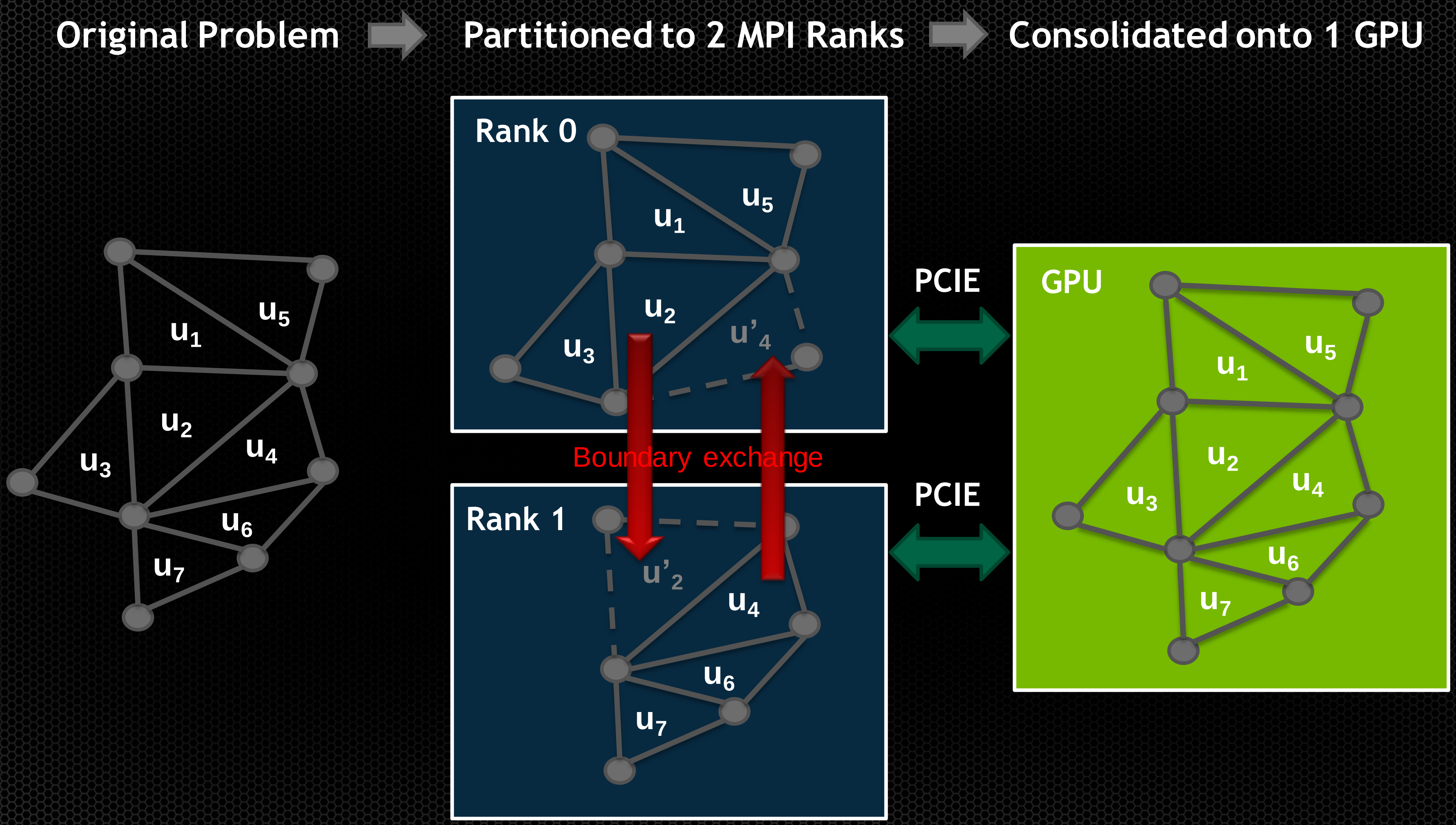


Dual socket CPU <=> 4 GPUs



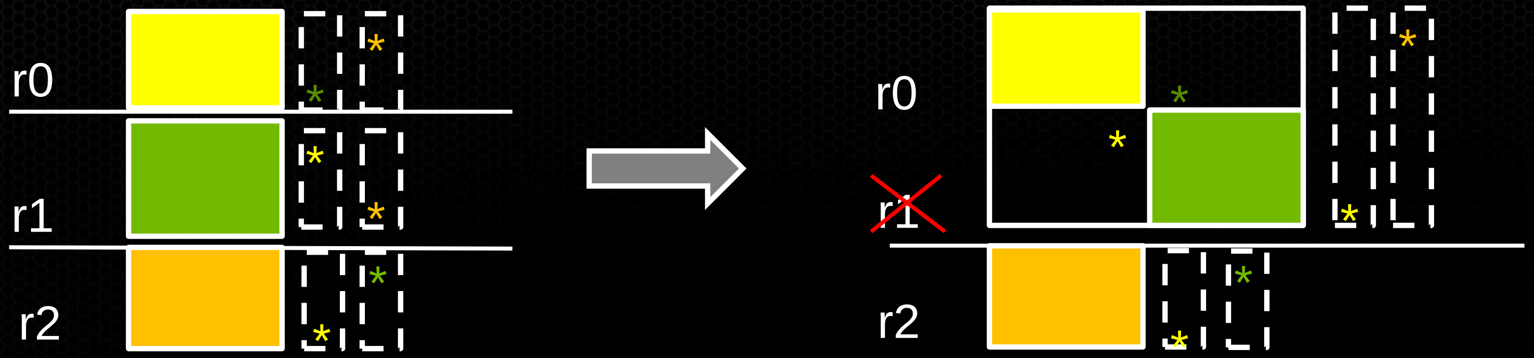
Arbitrary Cluster:
4 nodes x [2 CPUs + 3 GPUs]





Original Problem
Partitioned to 2 MPI Ranks
Consolidated onto 1 GPU
u_1
u_2
u_3
u_4
u_5
u_6
u_7
Rank 0
u_1
u_5
u_2
u_3
u'_4
Boundary exchange
Rank 1
u'_2
u_4
u_6
u_7
PCIE
PCIE
GPU
u_1
u_5
u_2
u_3
u_4
u_6
u_7

Consolidation



- Consolidation
  - ✓ Coarse level: little work to do (most time spent in communication)
  - ✓ Fine level: used to allow multiple ranks on a single GPU

Some Results

AmgX

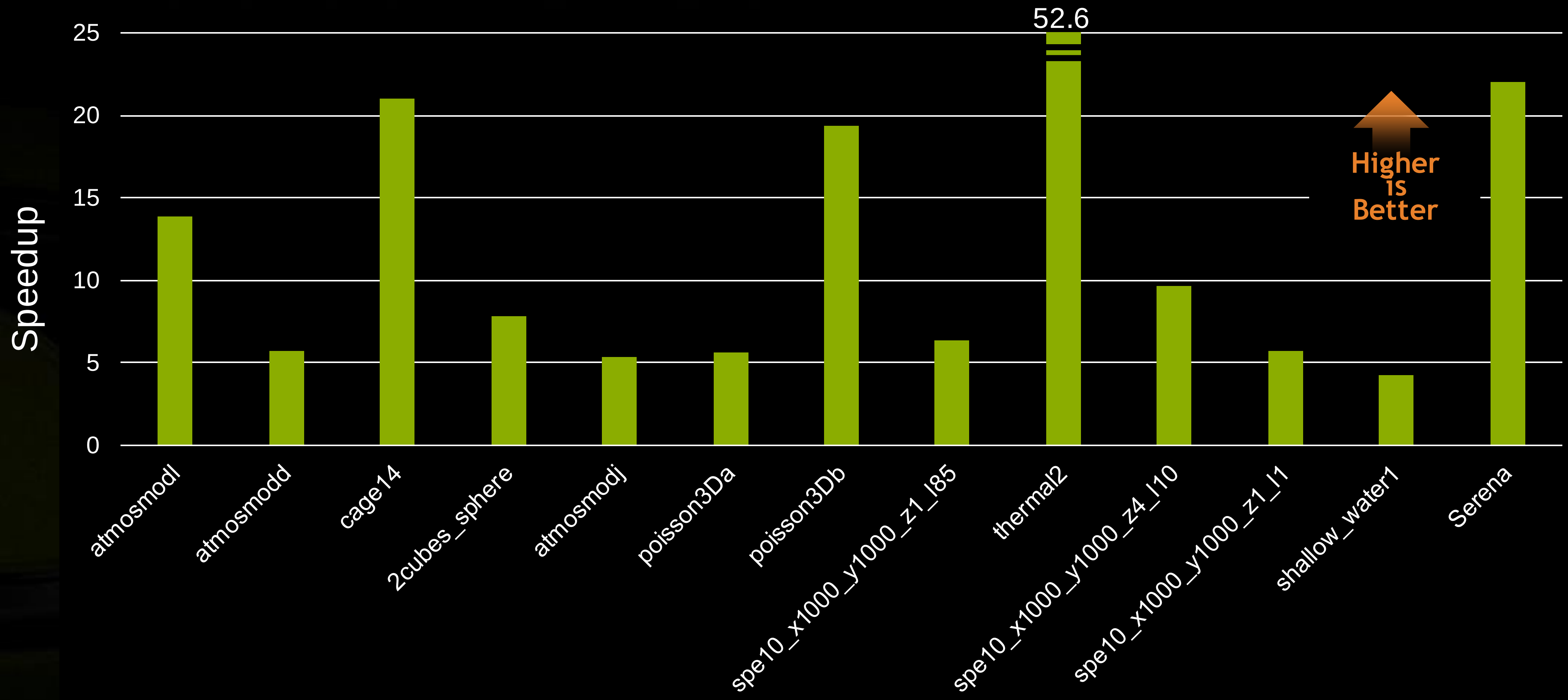
Fast, scalable linear solvers, emphasis on iterative methods

Flexible toolkit provides GPU accelerated $Ax = b$ solver

Simple API makes it easy to solve your problems faster

Florida Sparse Matrix Collection

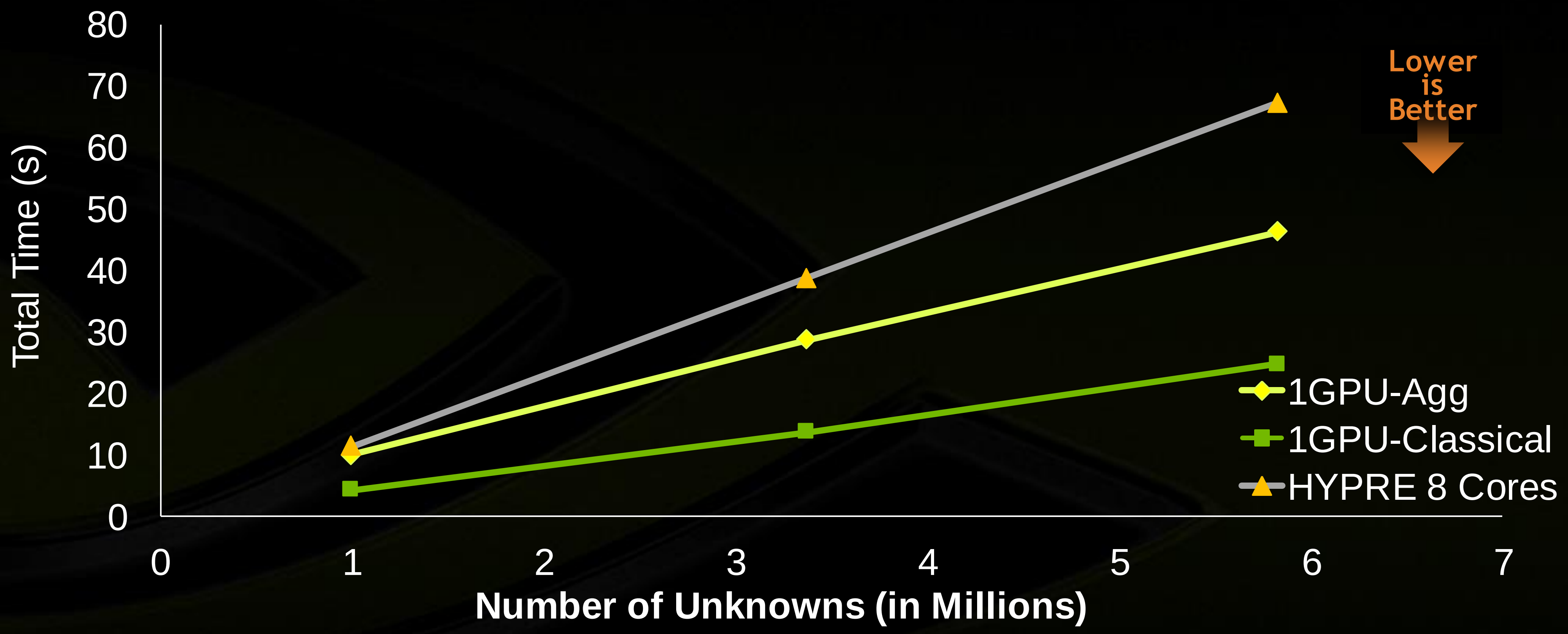
AmgX Classical vs. HYPRE



GPU: NVIDIA K40
CPU: Intel i7-3930K CPU @ 3.20GHz

miniFE* vs HYPRE

Single Node: 1 CPU Socket & 1 GPU

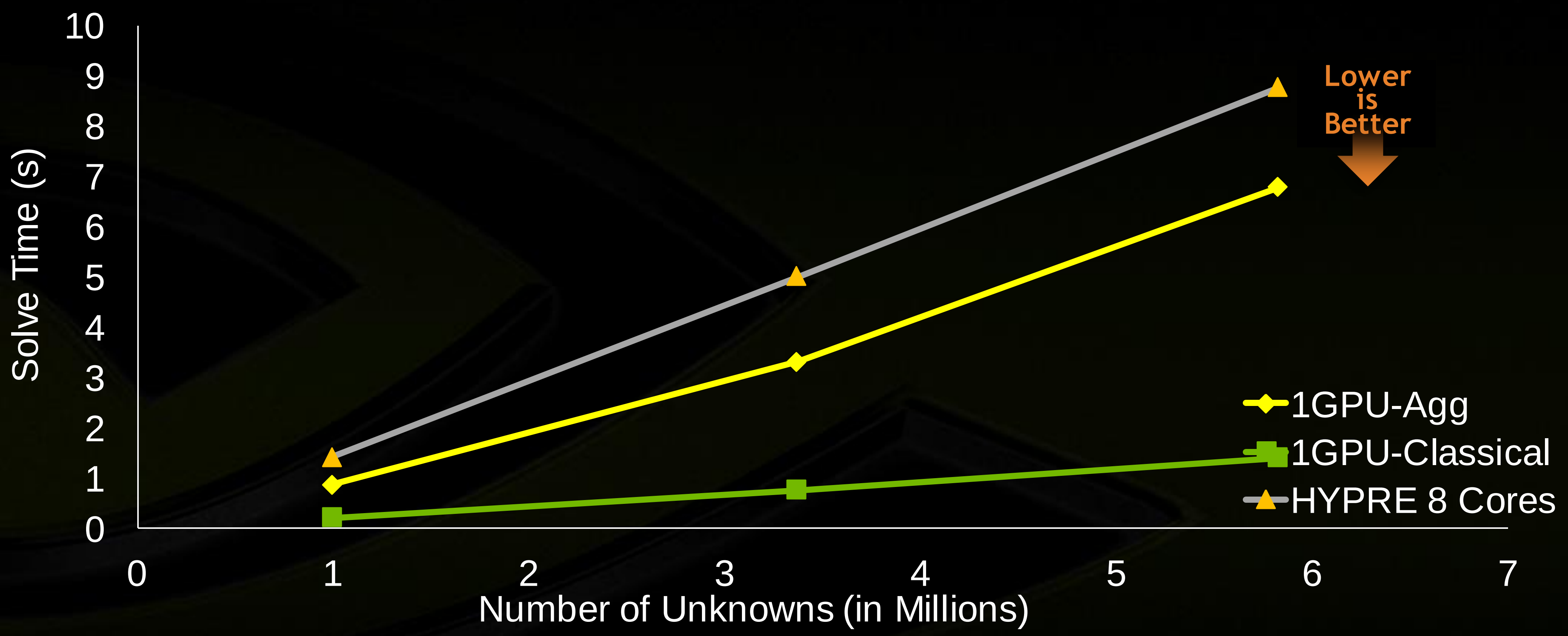


*: "mini app" from Sandia that performs assembly and solution of typical DOE Finite Element mesh

GPU: NVIDIA K40
CPU: Intel Xeon E5-2670 @ 2.60GHz

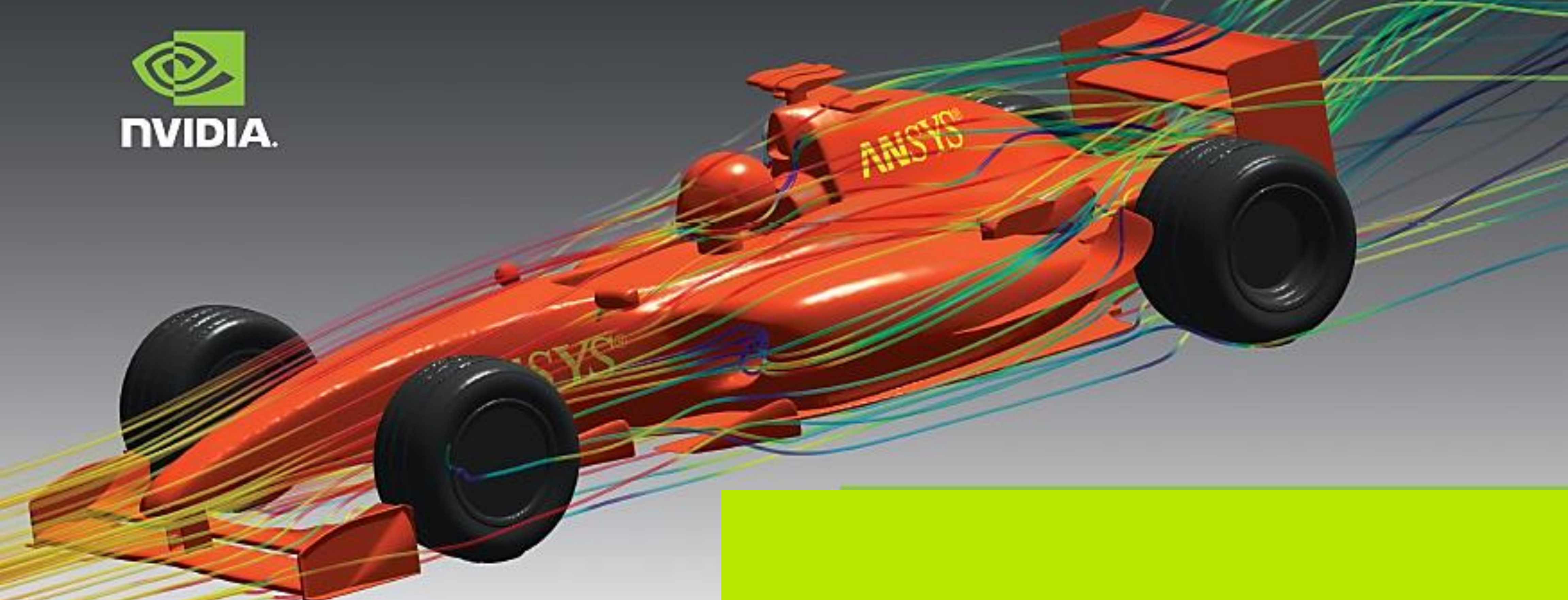
miniFE Benchmark vs HYPRE

Single Node: 1 CPU Socket & 1 GPU



*: "mini app" from Sandia that performs assembly and solution of typical DOE Finite Element mesh

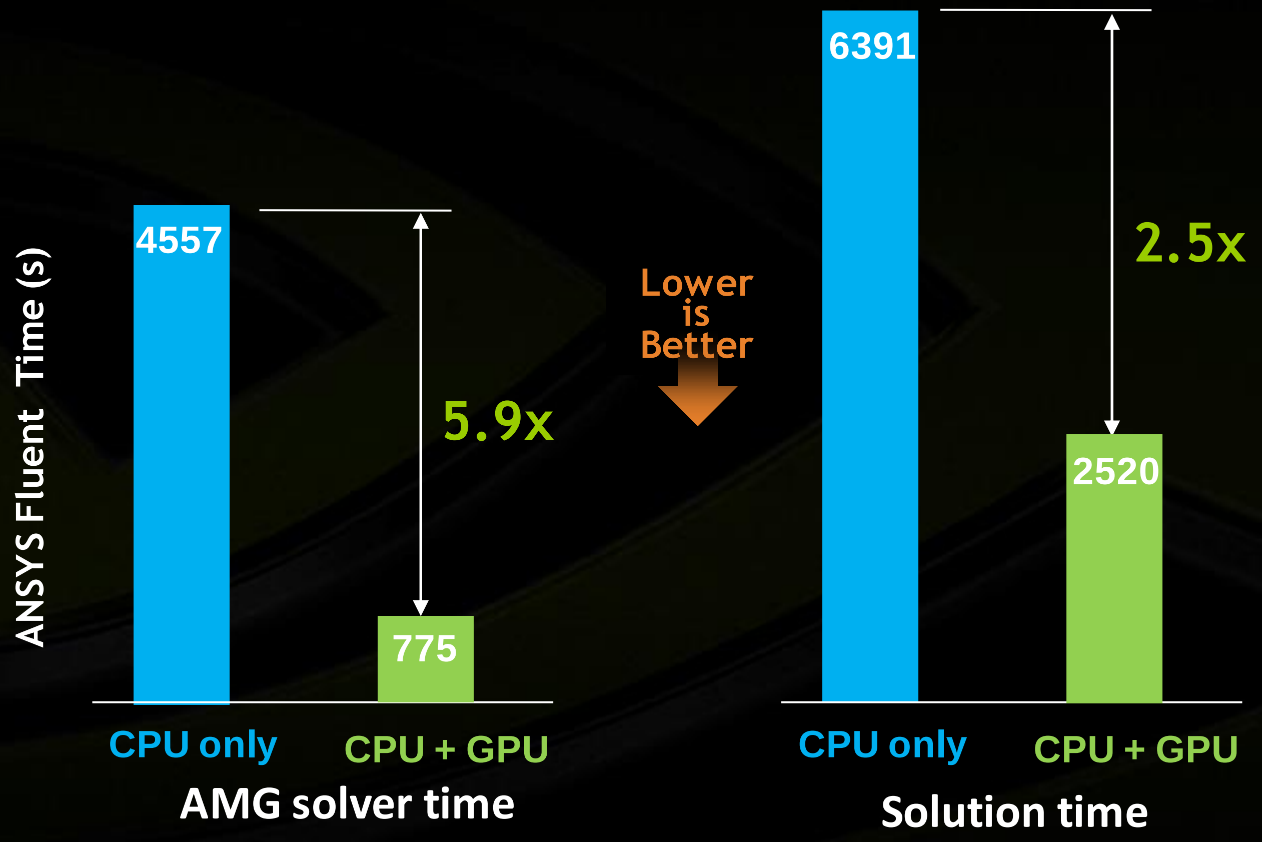
GPU: NVIDIA K40
CPU: Intel Xeon E5-2670 @ 2.60GHz



ANSYS® Fluent 15.0

GPU Acceleration of Water Jacket Analysis

ANSYS Fluent 15.0 performance on pressure-based coupled Solver



Water jacket model

- Unsteady RANS model
- Fluid: water
- Internal flow
- CPU: Intel Xeon E5-2680
- GPU: 2 X Tesla K40

NOTE: Times for 20 time steps

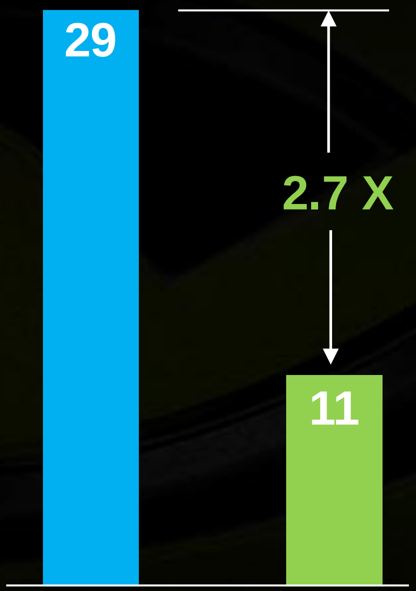
GPU Scaling on 111M Aerodynamic Problem

Better performance on problems with relatively high %AMG solver time

144 CPU cores – Amg

48 GPUs – AmgX

80% AMG solver time

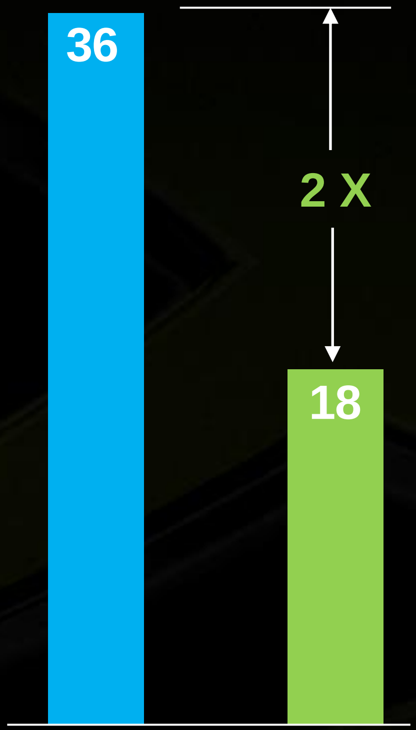


AMG solver time per iteration (s)

Lower is Better

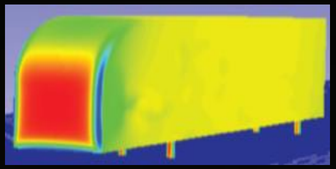
144 CPU cores

144 CPU cores + 48 GPUs



Fluent solution time per iteration (s)

Truck Body Model



- 111M mixed cells
- External aerodynamics
- Steady, k-ε turbulence
- Double-precision solver
- CPU: Intel Xeon E5-2667; 12 cores per node
- GPU: Tesla K40, 4 per node

NOTE: **AmgX is a GPU solver developed by NVIDIA and is implemented by ANSYS in Fluent for accelerating CFD**

Thanks

- AmgX Team: Maxim Naumov, Marat Arsaev, Patrice Castonguay, Jonathan Cohen, Julien Demouth, Joe Eaton, Simon Layton, Nikolay Markovskiy, Istvan Reguly, Nikolai Sakharnykh, Robert Strzodka, Zhenhai Zhu

- We're always looking for great students!
  - Interns
  - Openings on FFT and Sparse/AmgX teams
  - Email me: jocohen@nvidia.com