

# Meeting the Multicore Parallel Programming Scalability Challenge

**Wen-mei Hwu**

University of Illinois, Urbana-Champaign



# Agenda

- Need for Scalable Kernels
- Systematic Techniques and Tools
- Conclusion and Outlook

# GPU computing is catching on.

Financial  
Analysis

Scientific  
Simulation

Engineering  
Simulation

Data  
Intensive  
Analytics

Medical  
Imaging

Digital  
Audio  
Processing

Digital  
Video  
Processing

Computer  
Vision

Biomedical  
Informatics

Electronic  
Design  
Automation

Statistical  
Modeling

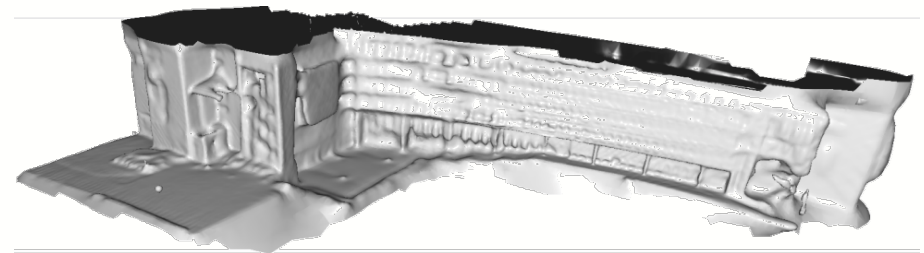
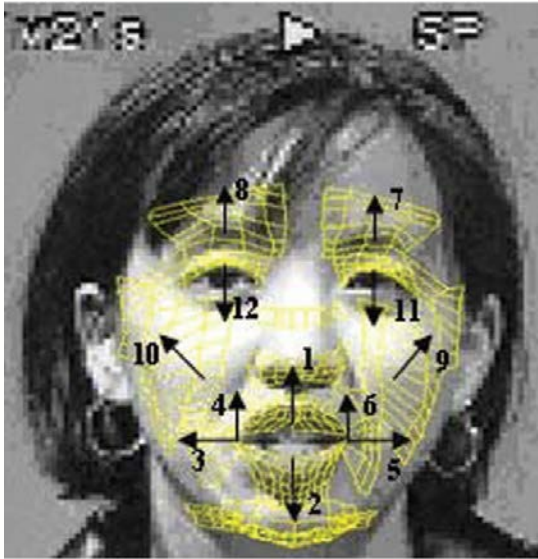
Ray  
Tracing  
Rendering

Interactive  
Physics

Numerical  
Methods

- 280 submissions to GPU Computing Gems
  - 110 articles included in two volumes

# GPU computing examples in consumer applications



Real-time surface reconstruction

Real-time emotion analysis



Frontal for Digit 2

# Library Kernels

- CUBLAS
  - Basic Linear Algebra
  - CUDA SDK
- CULA, Magma
  - Linear Algebra Solvers
  - Missing some solvers
- CUSP
  - Sparse data and algorithms
  - SpMV, CG, ...
  - Missing direct and iterative solvers, pre-conditioners, eigen analysis
- Graph algorithms
  - BFS kernels exist
  - Need graph partitioning kernels
- Unstructured grid algorithms
  - 3D surface mesh generation/refinement
  - Need 3D volume mesh generation (e.g. CGAL)/refinement
- Add your favorite library here

# Scalable kernel development for GPUs is heavy lifting.

Each kernel is typically a 3-month job but very few developers benefit from systematic techniques and advanced compiler technology today.



# However

- Software lasts through many hardware generations

Scalable algorithms and libraries can be the best legacy we can leave behind from this era.

# SCALABLE KERNEL DEVELOPMENT

# Four Challenges

- Computations with no known scalable parallel algorithms
  - Shortest path, Delaunay triangulation, ...
- Data distributions that cause catastrophic load imbalance in parallel algorithms
  - Free-form graphs, MRI spiral scan
- Computations that do not have data reuse
  - Matrix vector multiplication, ...
- Algorithm optimizations that are hard and labor intensive
  - Locality and regularization transformations

# Developing a Scalable Kernel Requires

- Massive parallelism in application algorithms
  - Data parallelism
- Regular computation and data accesses
  - Similar, balanced work for parallel threads
- Avoidance of conflicts in critical resources
  - Off-chip DRAM (Global Memory) bandwidth
  - Conflicting parallel updates to memory locations

# Massive Parallelism - Regularity



# Global Memory Bandwidth

Ideal



Reality



# Global Memory Bandwidth

- Many-core processors have limited off-chip memory access bandwidth compared to peak compute throughput
- Fermi
  - 1 TFLOPS SPFP peak throughput
  - 0.5 TFLOPS DPFP peak throughput
  - 144 GB/s peak off-chip memory access bandwidth
    - 36 G SPFP operands per second
    - 18 G DPFP operands per second
  - To achieve peak throughput, a program must perform  $1,000/36 = \sim 28$  SPFP (28 DPFP) arithmetic operations for each operand value fetched from off-chip memory

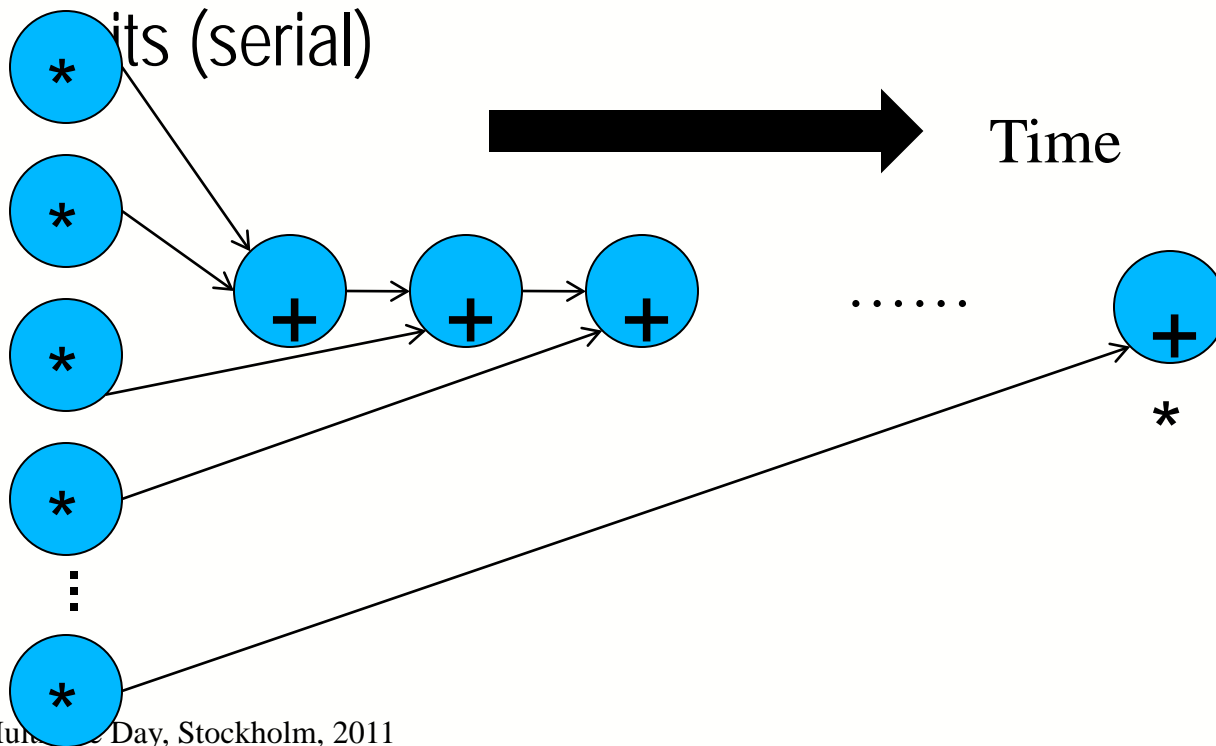
# Conflicting Data Accesses Cause Serialization and Delays

- Massively parallel execution cannot afford serialization
- Contentions in accessing critical data causes serialization



# A Simple Example

- A naïve inner product algorithm of two vectors of one million elements each
  - All multiplications can be done in time unit (parallel)
  - Additions to a single accumulator in one million time

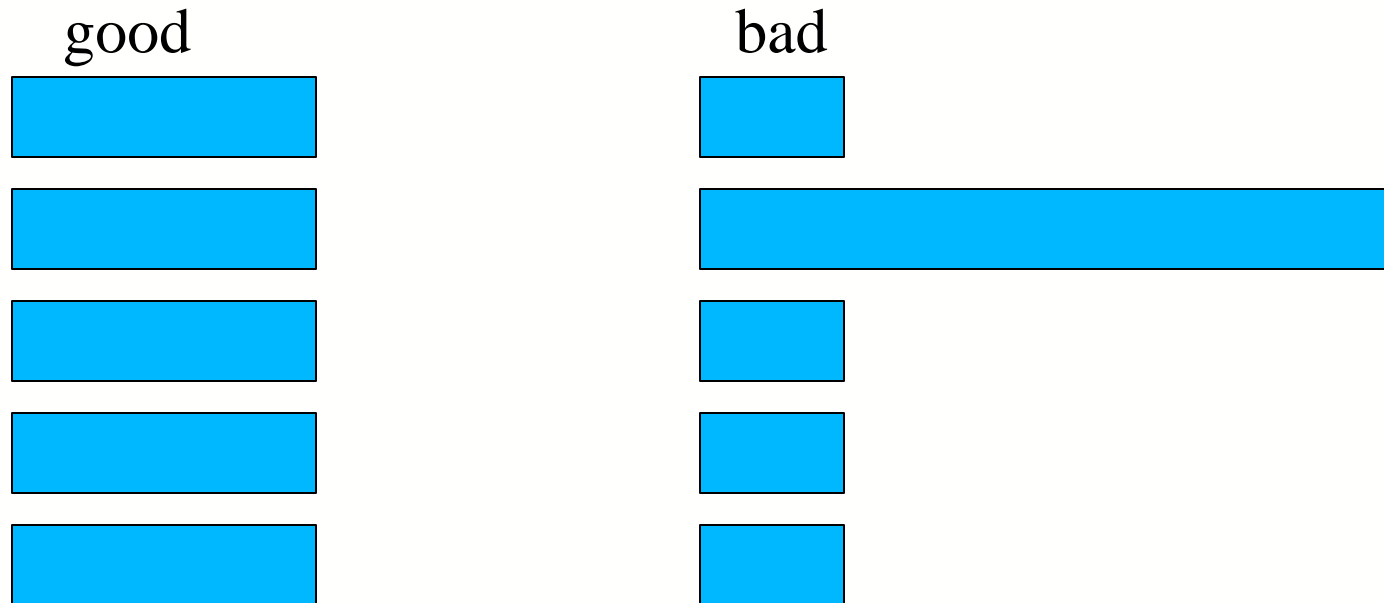


# How much can conflicts hurt?

- Amdahl's Law
  - If fraction  $X$  of a computation is serialized, the speedup can not be more than  $1/(1-X)$
- In the previous example,  $X = 50\%$ 
  - Half the calculations are serialized
  - No more than  $2X$  speedup, no matter how many computing cores are used

# Load Balance

- The total amount of time to complete a parallel job is limited by the thread that takes the longest to finish



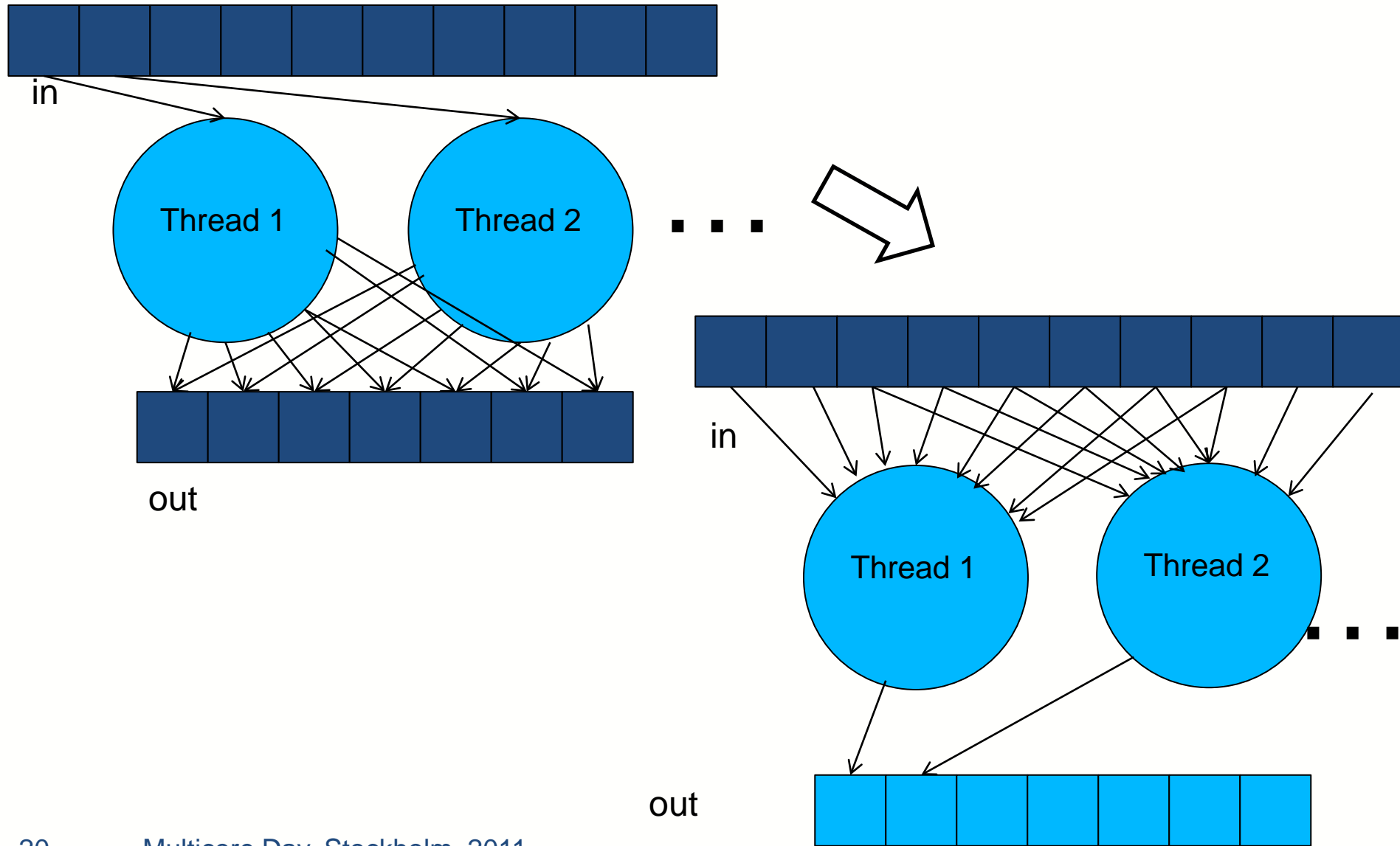
# How bad can it be?

- Assume that a job takes 100 units of time for one person to finish
  - If we break up the job into 10 parts of 10 units each and have 10 people to do it in parallel, we can get a 10X speedup
  - If we break up the job into 50, 10, 5, 5, 5, 5, 5, 5, 5, 5 units, the same 10 people will take 50 units to finish, with 9 of them idling for most of the time. We will get no more than 2X speedup.

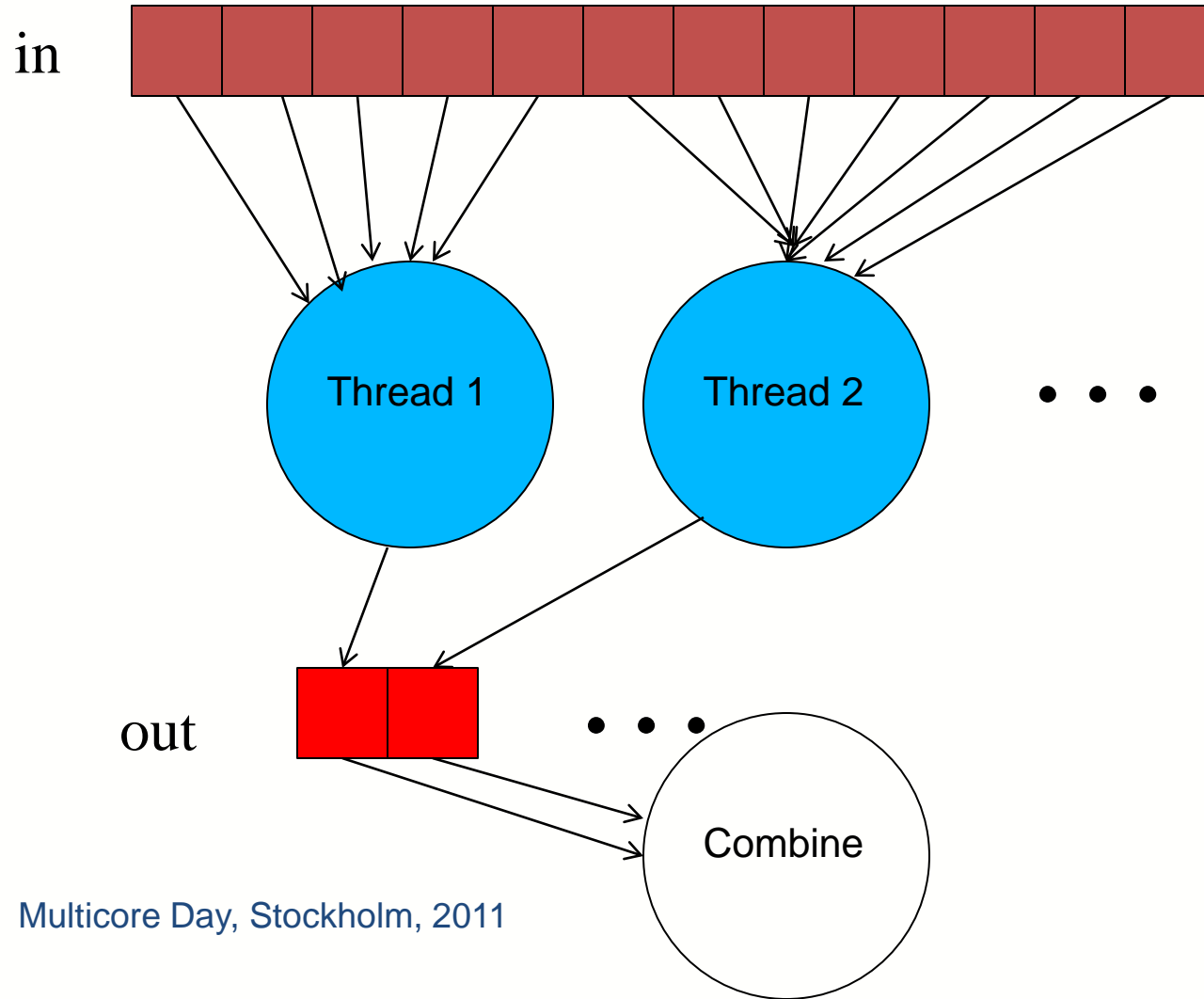
# Eight Techniques for Scalable Kernels (so far)

Techniques	Memory Bandwidth	Update Contention	Load Balance	Regularity	Efficiency
Scatter to Gather		X			
Privatization		X			
Tiling	X				X
Coarsening	X	X			X
Data Layout	X	X			X
Input Binning	X				X
Regularization			X	X	X
Compaction	X		X	X	X

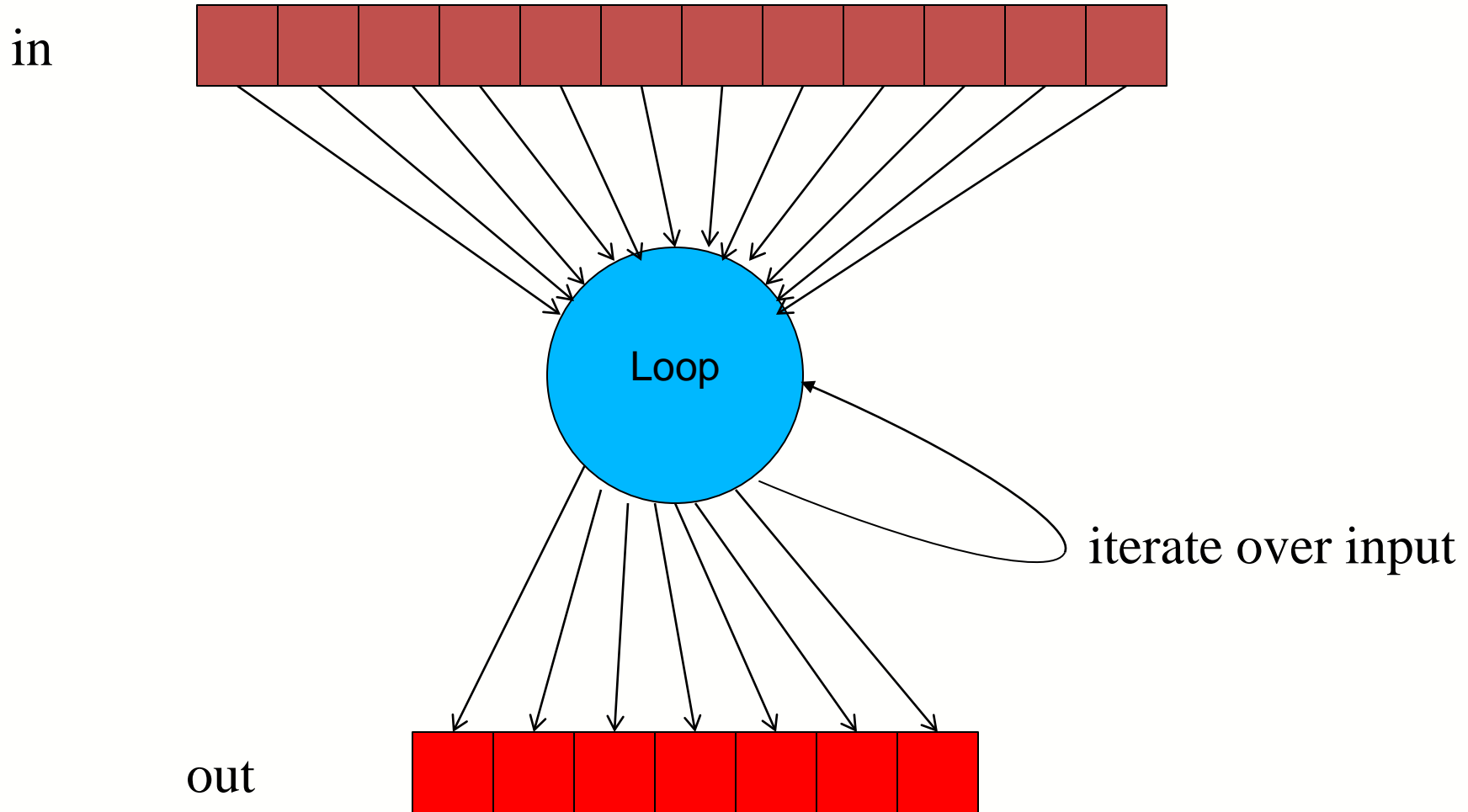
# 1: Scatter to Gather Transformation



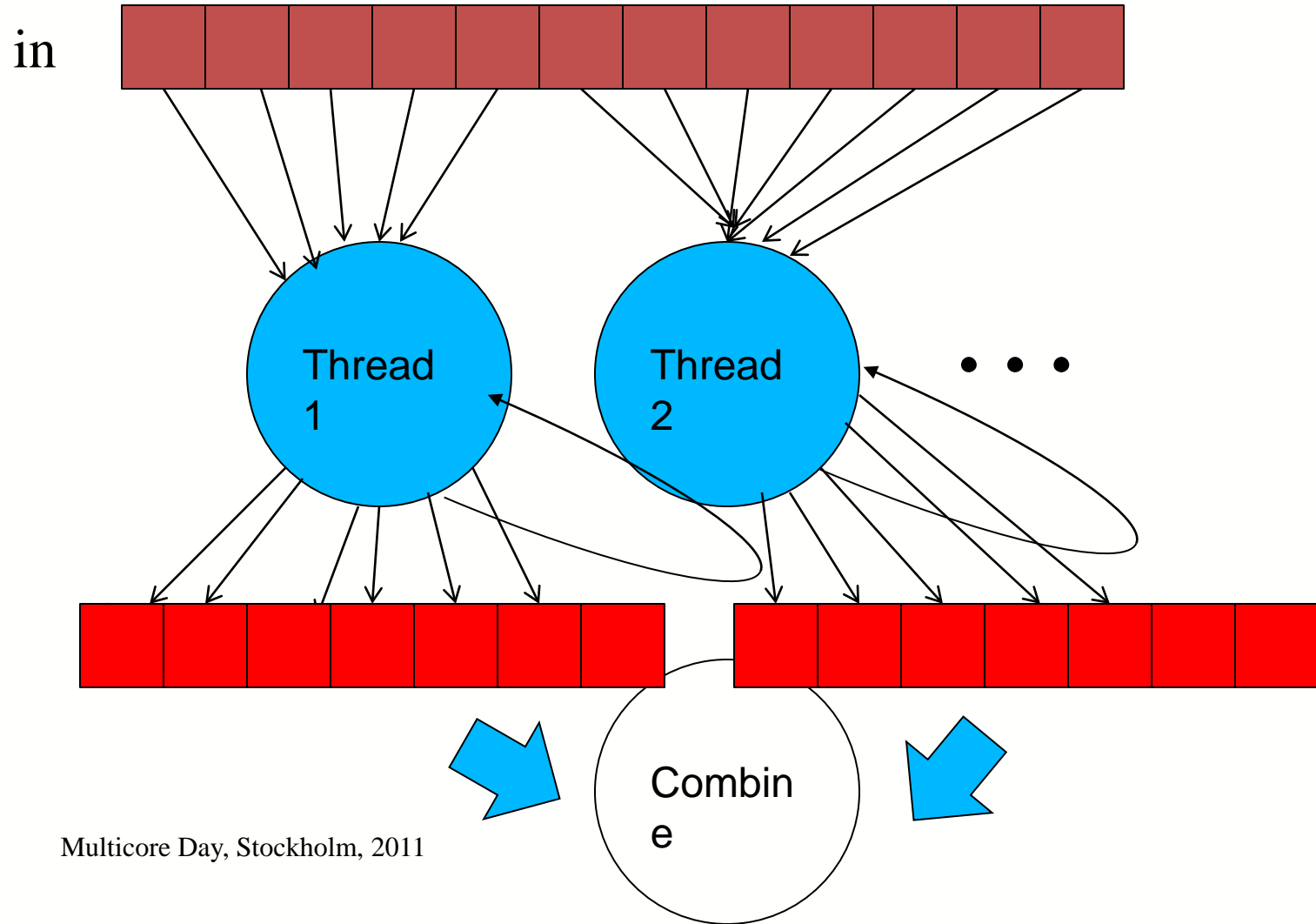
# 2. Privatization



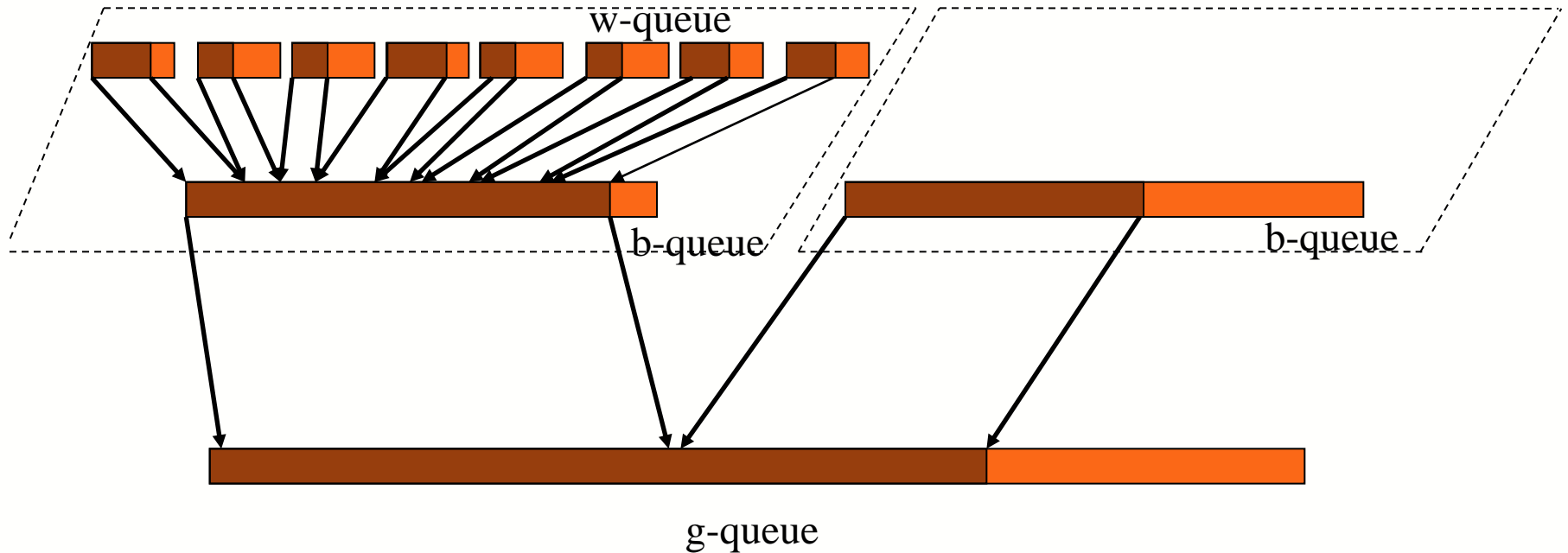
# Histogram is a scatter operation whose output is data dependent



# Histogram Privatization

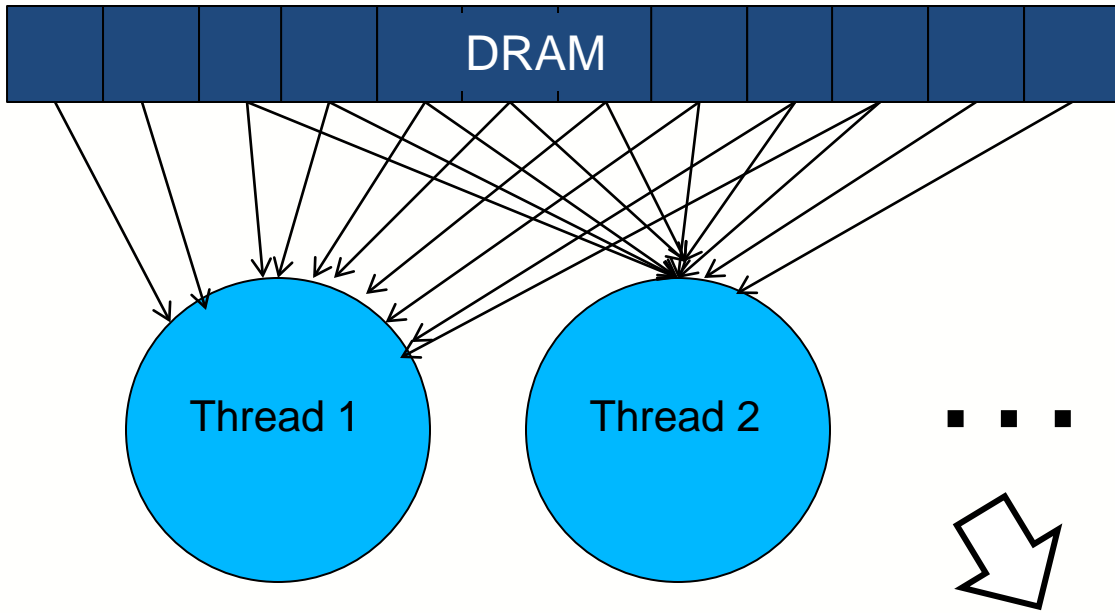


# Privatized Queue Hierarchy for Graph Algorithms

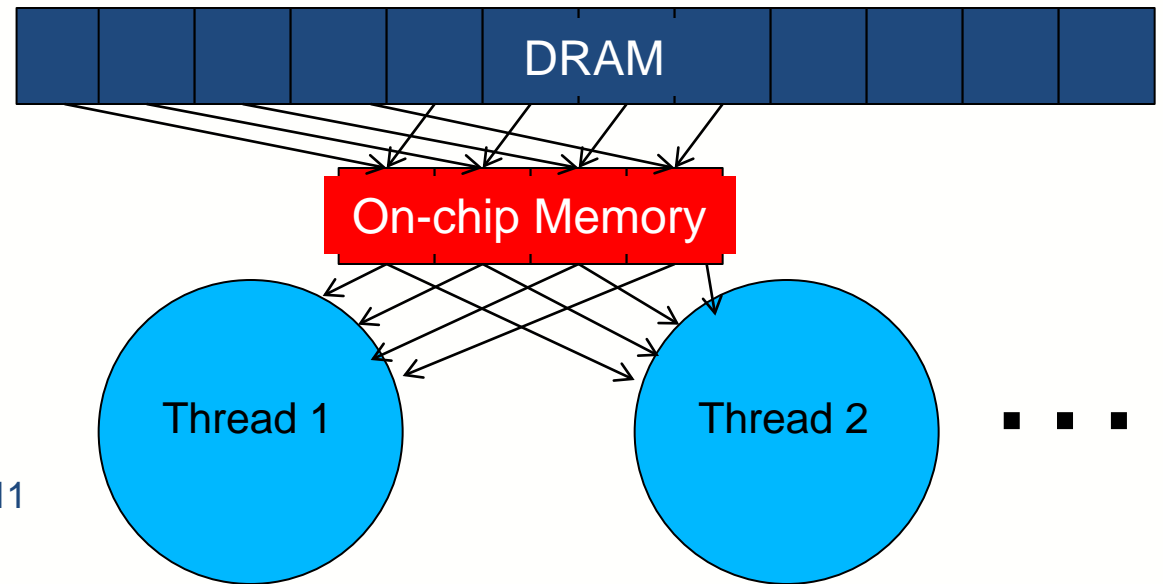


# 3: Data Access Tiling

in



in



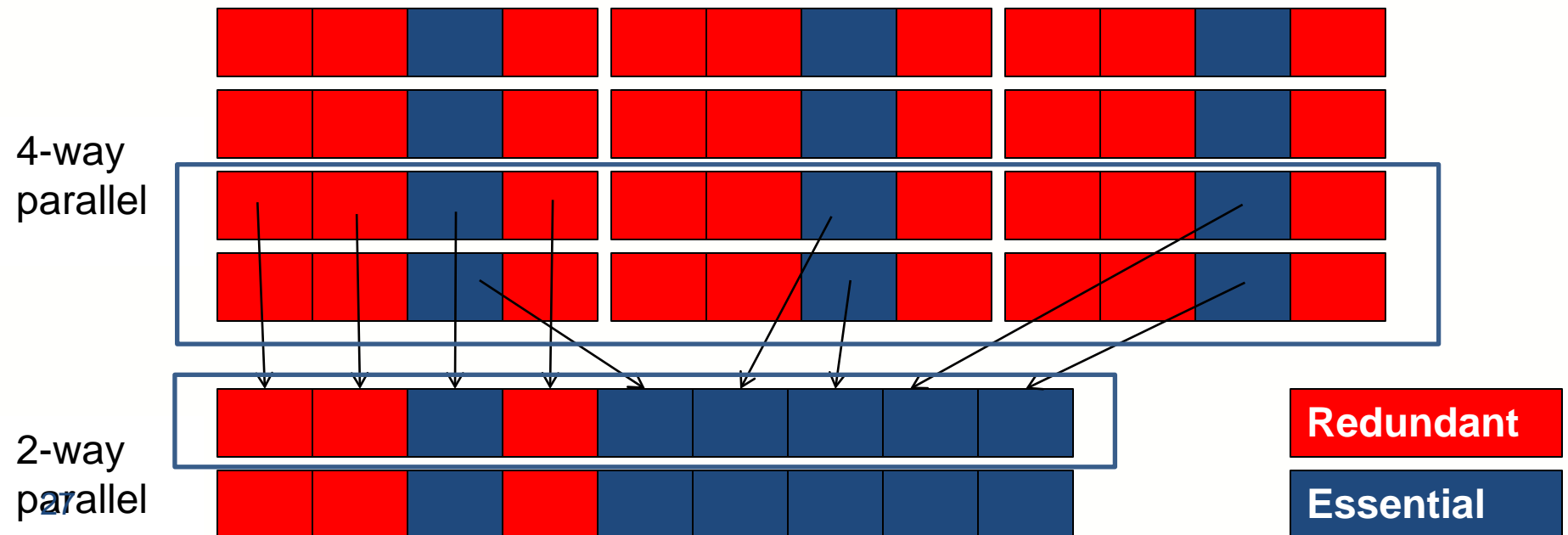
# Tiling can be tricky in practice.

- Halo cells in PDE solvers and Tri-diagonal solvers can drastically reduce the effectiveness
  - Sliding window approach at ICPP 2011 for Tri-diagonal solver makes tiling data scalable
  - Caching in Fermi makes PDE solvers much more scalable
- Irregular inputs cannot be tiled easily
  - Input binning (later) allows irregular inputs to be effectively tiled

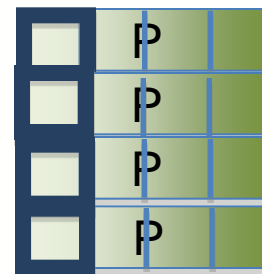
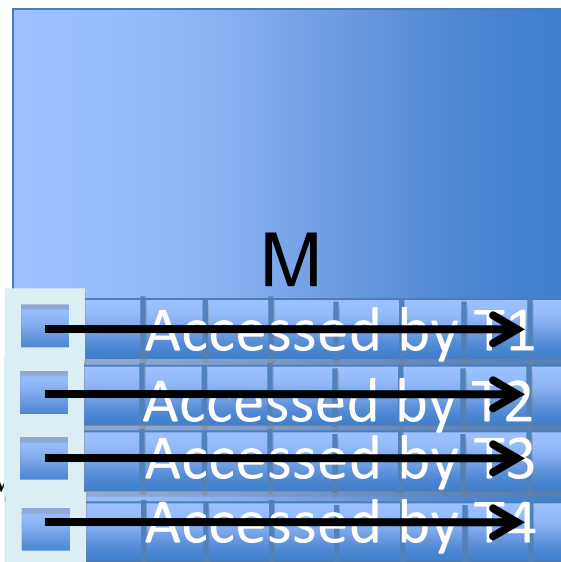
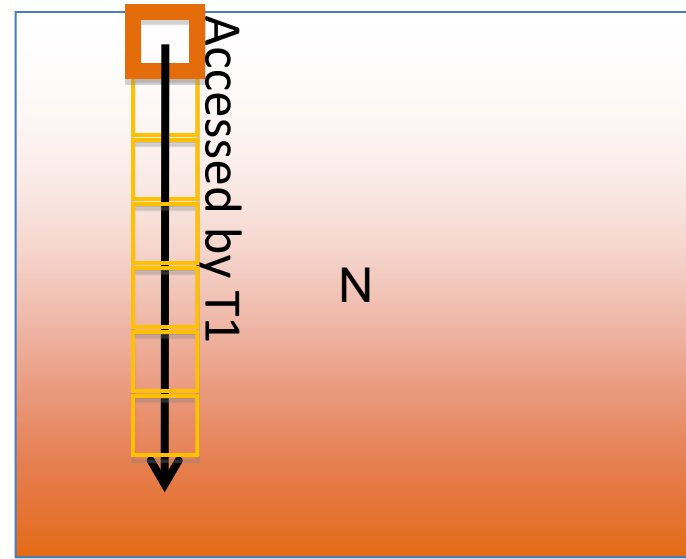
# 4. Granularity Coarsening and Register Data Reuse

- Parallel execution often requires redundant and coordination work
  - Merging multiple threads into one allows reuse of result, avoiding redundant work

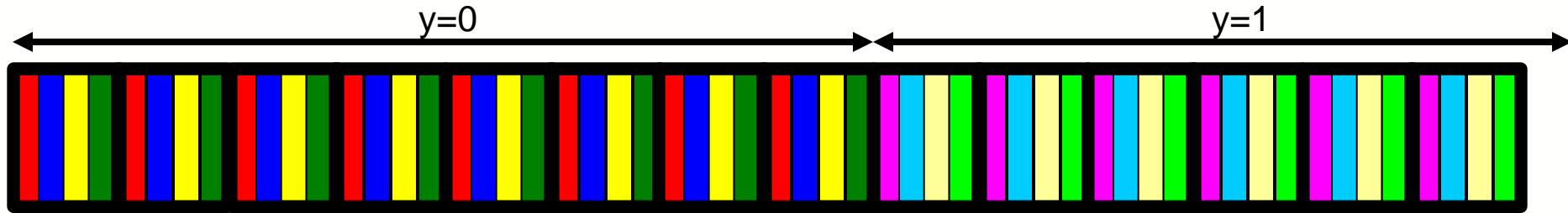
Time →



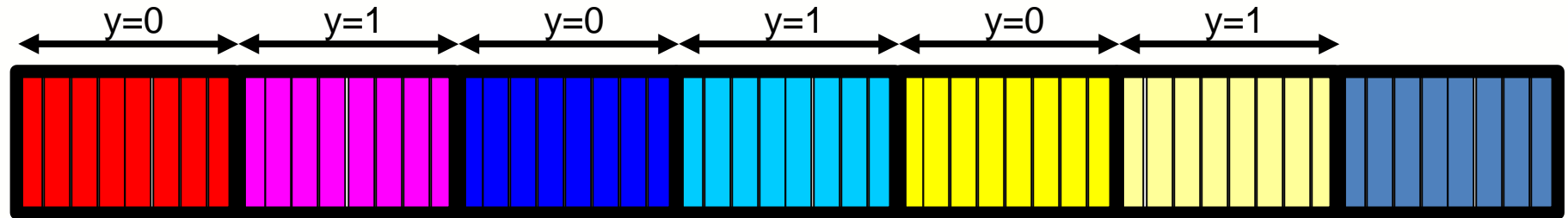
# Data Reuse in Matrix-Matrix Multiplication Revisited



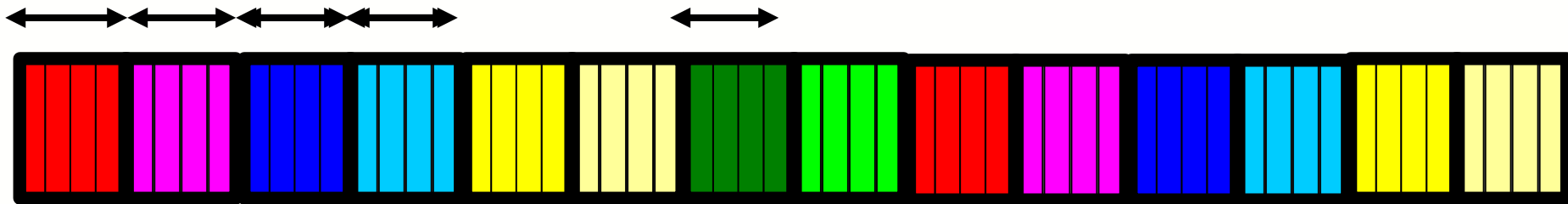
# 5. Data Layout Transformation



Array of Structure:  $[z][y][x][e]$

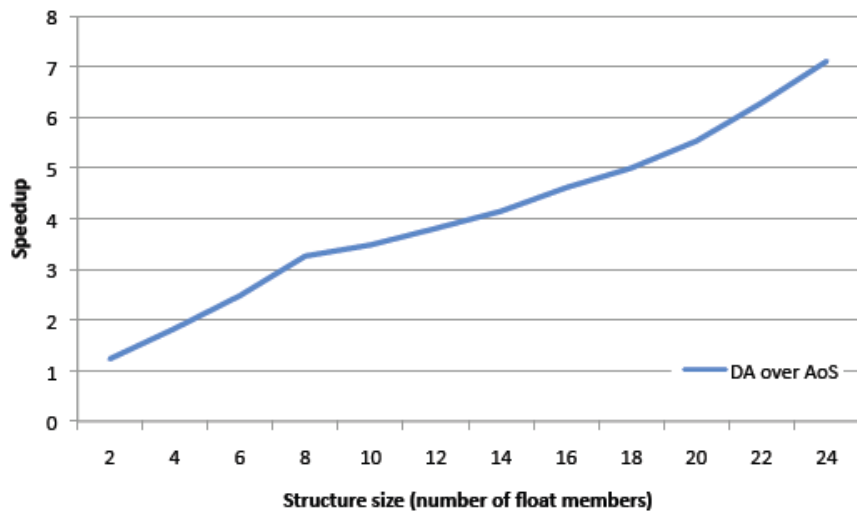


Structure of Array:  $[e][z][y][x]$

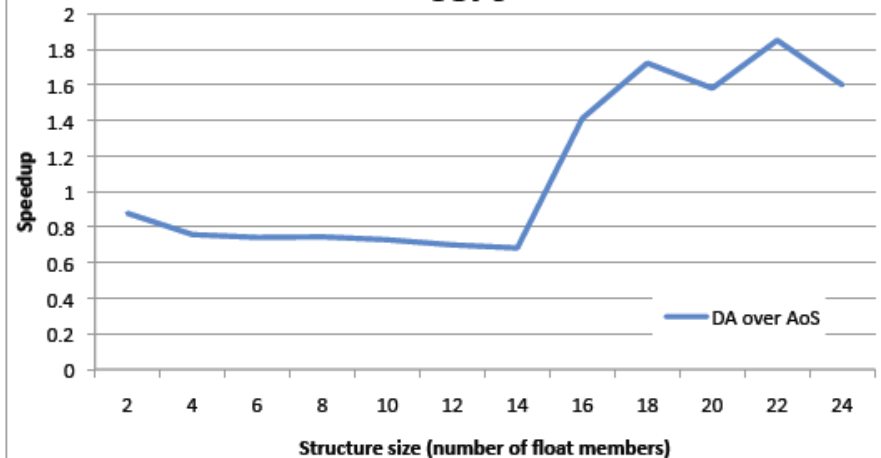


# SoA is not always better

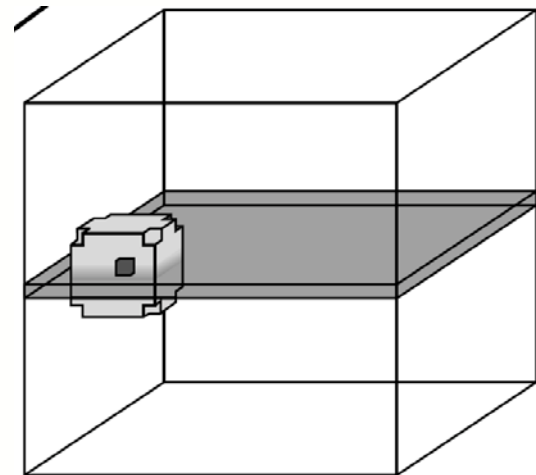
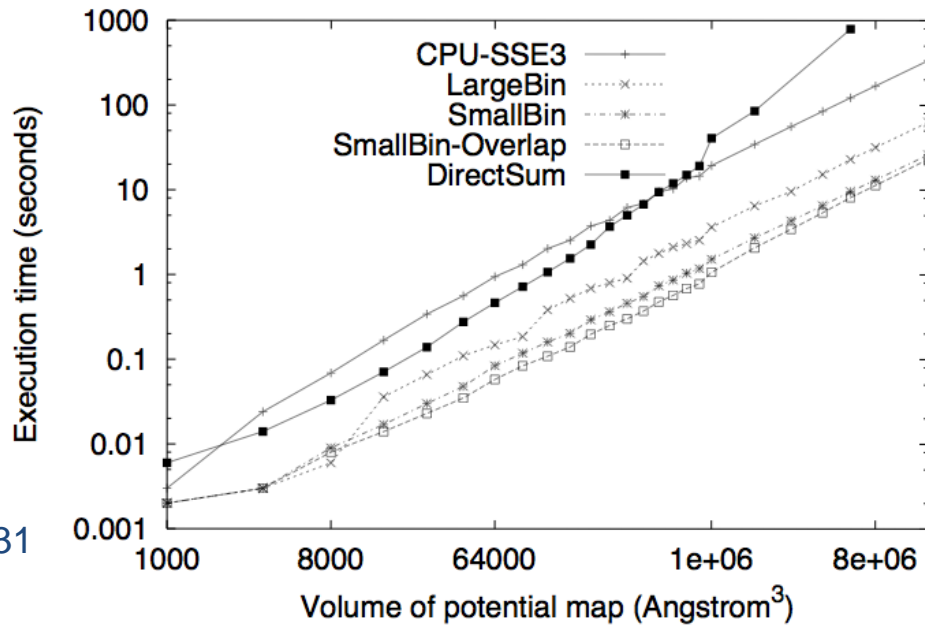
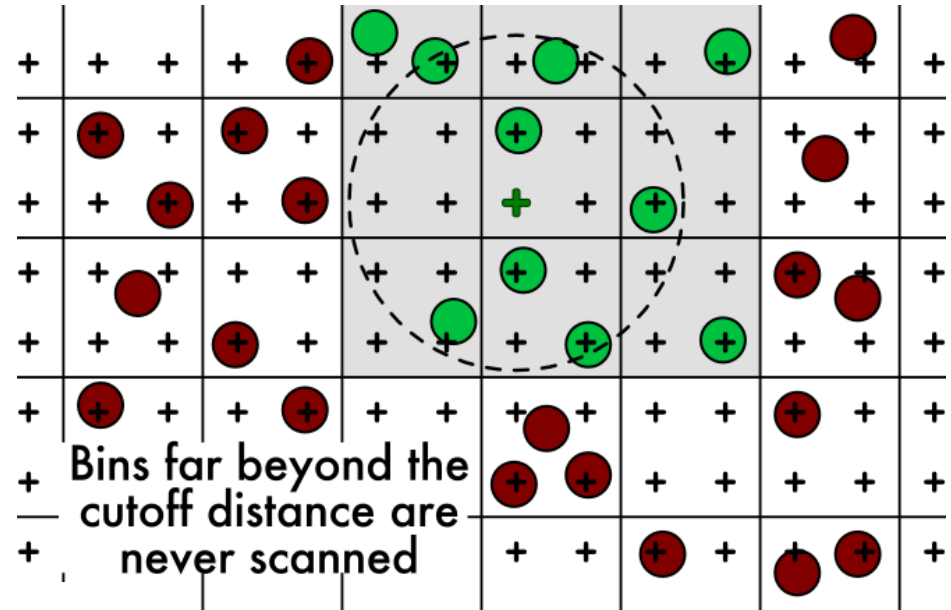
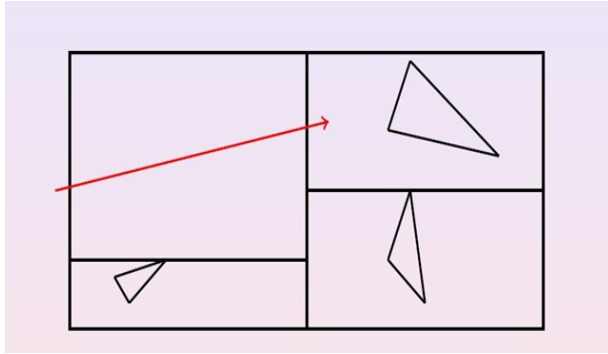
Speedup of DA over AoS, NVIDIA GTX480



Speedup of DA over AoS, ATI Radeon HD 5870

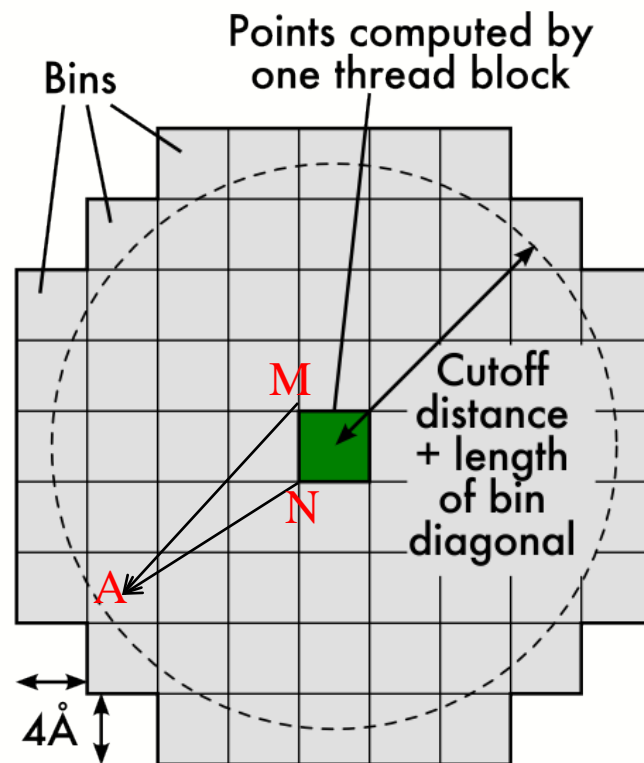


# 6: Input Data Binning



# Binned Kernels – Work Efficiency and Data Tiling

- Thread block examines atom bins up to the cutoff distance
  - All threads in a block scan the same bins and atoms
    - No hardware penalty for multiple simultaneous reads of the same address
    - Simplifies fetching of data
  - The sphere has to be big enough to cover all grid point at corners
  - There will be a small level of divergence
    - Not all grid points processed by a thread block relate to all atoms in a bin the same way
    - (A within cut-off distance of N but outside cut-off of M)

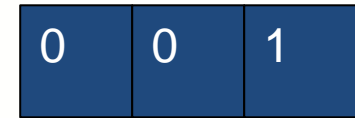


# 7. Regularization

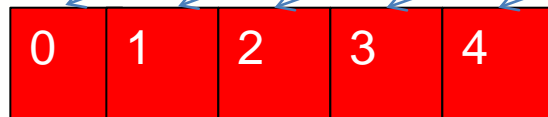
# 8. Compaction



Variable sized bins,  
sort and scan

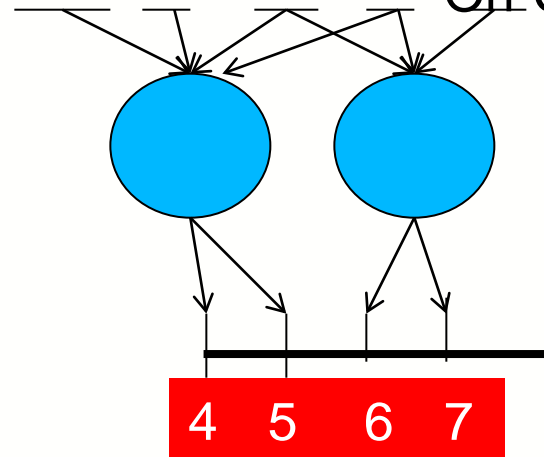
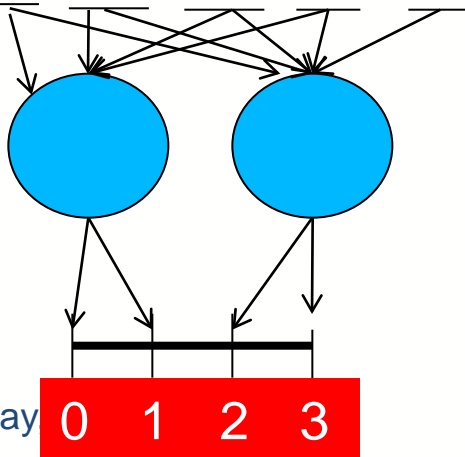


CPU



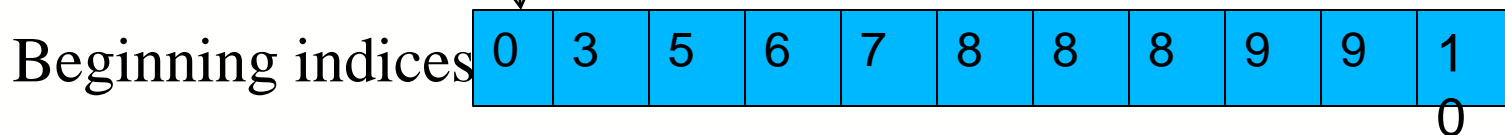
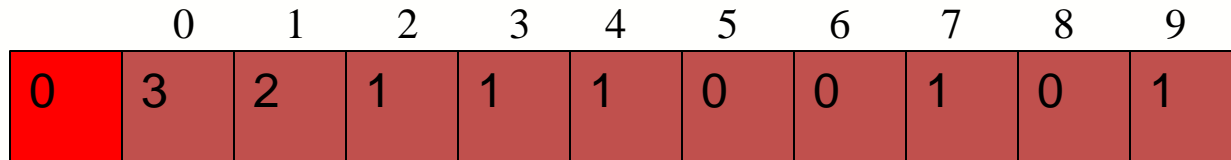
On-chip Memory

On-Chip Memory



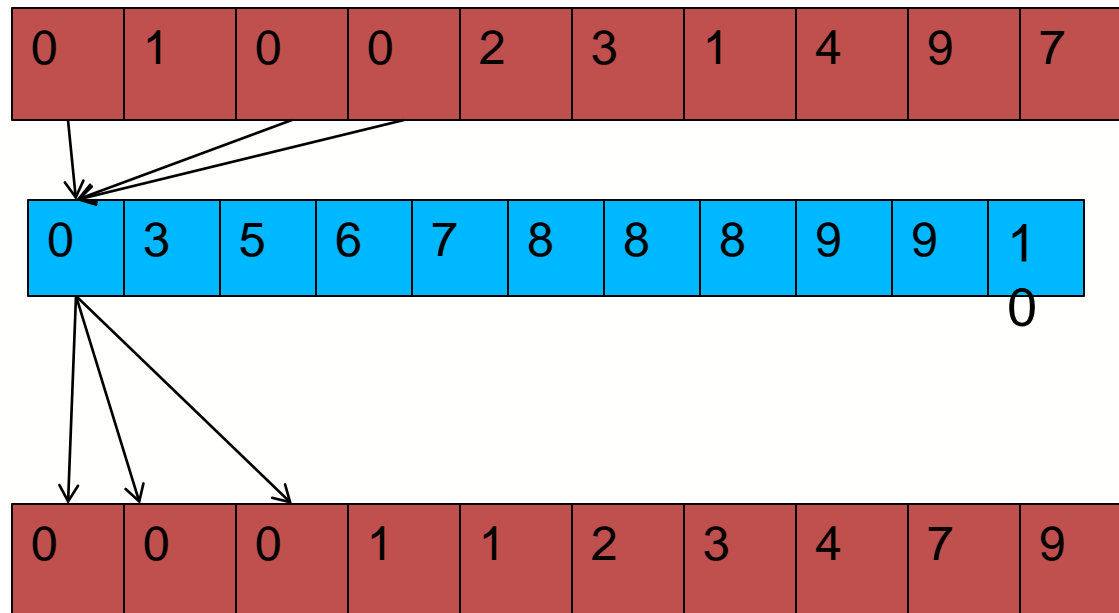
# Determine Start and End of Bins

- Use parallel pre-fix scan operations on the bin capacity array to generate an array of starting points of all bins (CUDPP)



# Actual Binning

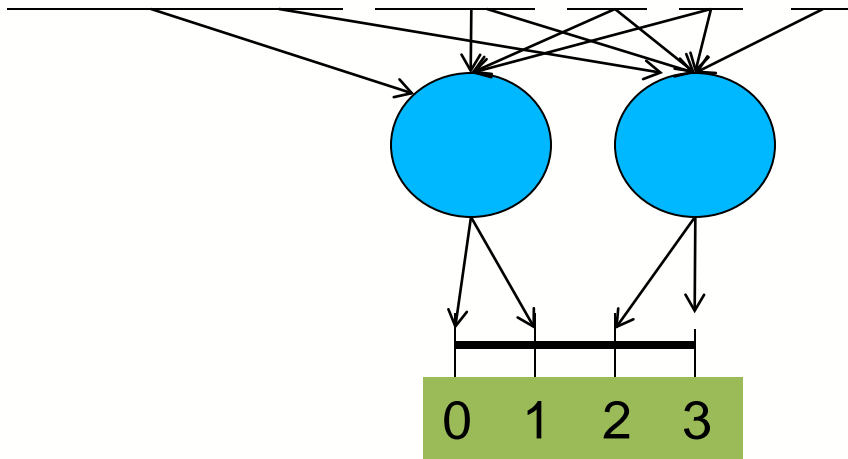
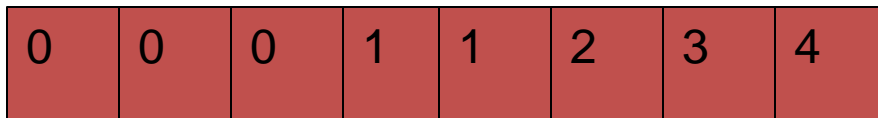
- All inputs can now be placed into their bins in parallel, using atomic operations



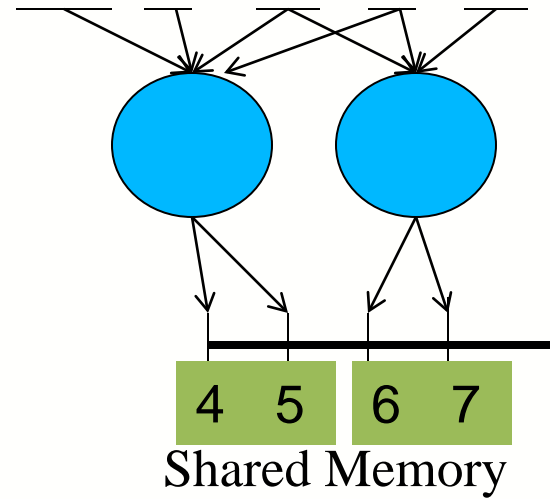
# A Tiled Gather Implementation

k

Shared Memory



Shared Memory



# See Parboil for practical embodiment of the techniques.

- [Impact.crhc.illinois.edu/Parboil](http://Impact.crhc.illinois.edu/Parboil)
- Kernels from real applications
  - sparse matrix-vector multiplication, dense matrix-matrix multiplication, Jacobi stencil, electrical static potential calculation,
- Timing and experimental framework to quantify the benefit of techniques

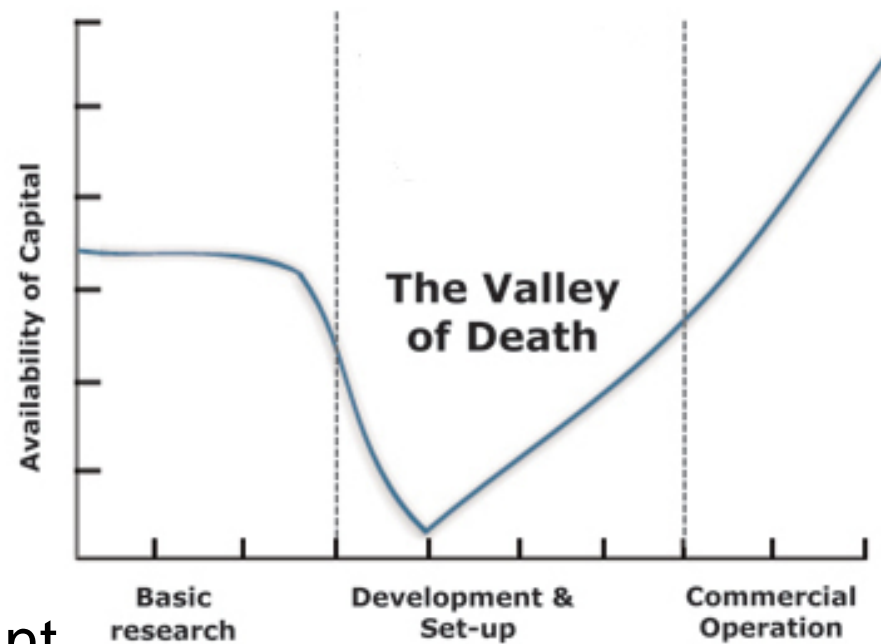
# Use of Optimizations in Parboil

Benchmark	Unoptimized Implementation Bottleneck	Optimizations Applied	Optimized Implementation Bottleneck	Primary Limit of Efficiency
cutcp	Contention, Locality	Scatter-to-Gather, Binning, Regularization, Coarsening	Instruction Throughput	Reads/Checks of Irrelevant Bin Data
mri-q	Poor Locality	Data Layout Transformation, Tiling, Coarsening	Instruction Throughput	N/A (true bottleneck)
gridding	Contention, Load Imbalance	Scatter-to-Gather, Binning, Compaction, Regularization, Coarsening	Instruction Throughput	Reads/Checks of Irrelevant Bin Data
sad	Locality	Tiling, Coarsening	Memory Bandwidth/Latency	Register Capacity
stencil	Locality	Coarsening, Tiling	Bandwidth	Local Memory, Register Capacity
tpacf	Locality, Contention	Tiling, Privatization, Coarsening	Instruction Throughput	N/A (true bottleneck)
lbn	Bandwidth	Data Layout Transformation	Bandwidth	N/A (true bottleneck)
dmm	Bandwidth	Coarsening, Tiling	Instruction Throughput	N/A (true bottleneck)
spm	Bandwidth	Data Layout Transformation	Bandwidth	N/A (true bottleneck)
bfs	Contention, Load Imbalance	Privatization, Compaction, Regularization	Bandwidth	Whole-Device Local Memory Capacity
histogram	Contention, Bandwidth	Privatization, Scatter-to-Gather	Bandwidth	Reads of Irrelevant Input (alleviated by cache)

# Tools go with techniques.

- Tools should facilitate key techniques
  - Programmers should write code “for others to understand instead of for computers to execute”
    - Dijkstra
- Techniques vary in their potential for automation
  - Scatter-to-gather, granularity coarsening, data access tiling, and memory layout quite amenable
    - Need clear performance guidance
  - Input binning, regularization, and compaction are much harder
    - Need to provide APIs understood by compilers/tools
    - Developer feedback critical to success

# Crossing the Valley of Death



By giving developers what they want.

# Conclusion and Outlook

- Standard and domain library kernels are main use model of many-cores.
  - Kernel development is heavily lifting
  - Systematic techniques and tools help
- However, compilers/tools are fragile.
  - Compilers transformations need to be part of the development flow, rather than afterthought

# Acknowledgements

- D. August (Princeton), S. Baghsorkhi (Illinois), N. Bell (NVIDIA), D. Callahan (Microsoft), J. Cohen (NVIDIA), B. Dally (Stanford), J. Demmel (Berkeley), P. Dubey (Intel), M. Frank (Intel), M. Garland (NVIDIA), M. Gschwind (IBM), R. Hank (Google), J. Hennessy (Stanford), P. Hanrahan (Stanford), M. Houston (AMD), T. Huang (Illinois), D. Kaeli (NEU), K. Keutzer (Berkeley), I. Gelago (UPC), B. Groppe (Illinois), D. Kirk (NVIDIA), D. Kuck (Intel), S. Mahlke (Michigan), T. Mattson (Intel), N. Navarro (UPC), J. Owens (Davis), D. Padua (Illinois), S. Patel (Illinois), Y. Patt (Texas), D. Patterson (Berkeley), C. Rodrigues, S. Ryou (ZeroSoft), C. Rodrigues (Illinois) K. Schulten (Illinois), B. Smith (Microsoft), M. Snir (Illinois), I. Sung (Illinois), P. Stenstrom (Chalmers), J. Stone (Illinois), S. Stone (Harvard) J. Stratton (Illinois), M. Valero (UPC)
- And many others!

**THANK YOU!**

# OpenCurrent – PDE Solvers

- Structured Grids **Cohen**
  - 1D, 2D, 3D - Single and double precision
  - Linear combinations, host-device transfers, interpolation at non-grid points, array-wide reduction
- Solvers
  - Calculate terms from discretization of PDEs
  - Finite-difference advection and diffusion schemes
  - Multi-grid solver for Poisson Equations
- Equations
  - Solve time-dependent PDEs
  - Incompressible Navier-Stokes solver, ...
  - Need many more types of PDEs