High-Level Programming Models for Heterogeneous Parallel Computing

Wen-mei Hwu

University of Illinois, Urbana-Champaign

Agenda

- Many-core Usage Pattern and Workflow
- Quick overview of current tools/environments
- Raising the Level of Kernel Development
- 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

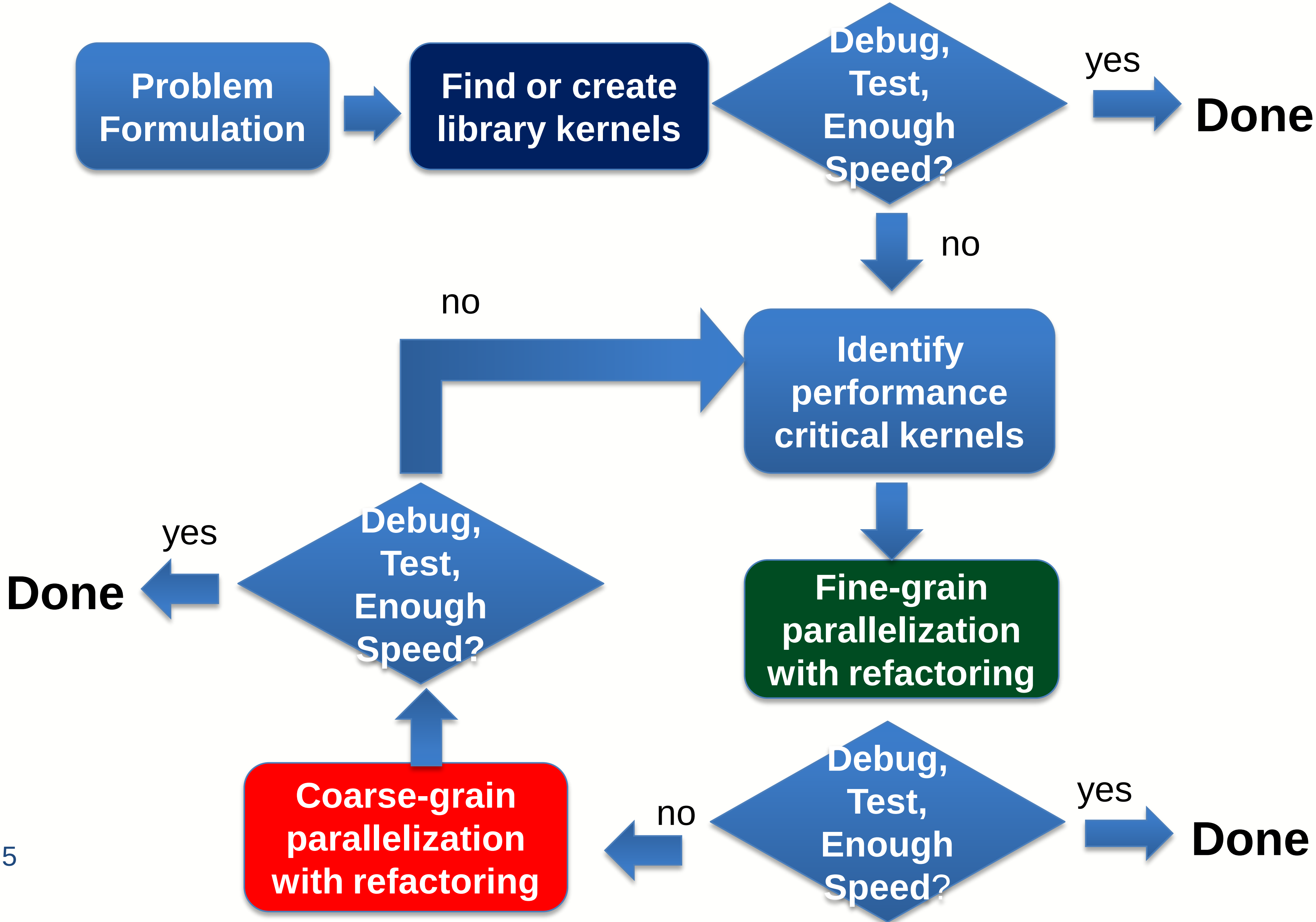
A Common GPU Usage Pattern

- A desirable approach considered impractical
  - Due to excessive computational requirement
  - But demonstrated to achieve domain benefit
  - Convolution filtering (e.g. bilateral Gaussian filters), De Novo gene assembly, etc.

- Use GPUs to accelerate the most time-consuming aspects of the approach
  - Kernels in CUDA or OpenCL
  - Refactor host code to better support kernels

- Rethink the domain problem

Practical Parallelization Flow



Problem Formulation

Find or create library kernels

Debug, Test, Enough Speed?

yes

Done

no

Identify performance critical kernels

Fine-grain parallelization with refactoring

Debug, Test, Enough Speed?

yes

Done

no

Coarse-grain parallelization with refactoring

Debug, Test, Enough Speed?

yes

Done

no

Need for Better Kernel Interface

- Tedious and error prone
  - Each object has two versions
  - Programmers try to avoid copying by tracking which object is needed by CPU and GPU at each program point
  - Double buffering and triple buffering in copying used to hide latency
- In real applications
  - Disk I/O and MPI messages are often done by different threads than compute to hide latency
  - Data copying involves tricky thread synchronization

Using Kernels in Higher-Level Environments

- Thrust – C++ STL for CUDA
  - Containers, iterators, common kernels
  - http://code.google.com/p/thrust/
- Jacket- use of CUDA functions in MatLab code
  - Auto. GPU memory management and data transfer
  - http://wiki.accelereyes.com
- PyCUDA- use of CUDA functions in Python code
  - Consistent with numpy usage
  - http://mathema.tician.de/software/pycuda
- GMAC – Global shared address space
  - AutoGPU memory management and data transfer.
  - http://adsm.googlecode.com/

Where are the pain points?

- Currently, these systems do not use advanced compiler technology or hardware support
  - Implemented mostly through templates/macros

- Unnecessary data transfers before and after kernel execution
  - Could be optimized away with information on side-effect of kernels and live range of data objects
  - Portability for fusion architectures

- Double and triple buffering done by hand
  - Could be automated with compiler transformations

AVAILABLE KERNELS

Library Kernels

- CUBLAS
  - Basic Linear Algebra
  - CUDA SDK
- CULA, Magma
  - Linear Algebra Solvers
  - www.culatools.com
  - http:/icl.cs.utk.edu/magma
- CUSP
  - Sparse data structures and algorithms
  - SpMV, CG, …

- 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

Four Challenges

- Computations with no known scalable parallel algorithms
  - Shortest path, Delaunay triangulation, …
- Data distributions that cause catastrophical 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

Kernel development for GPUs is heavy lifting.

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

Little code reuse due to kernel sensitivity to memory access patterns and work distribution.

KERNEL DEVELOPMENT

Many-core Kernel Development

- Many-core programming is about performance and scalability.
    - Scalability is also key to power efficiency.
    - Performance and scalability for many-cores requires largely the same techniques.
        - To regularize work and data for massively parallel execution.
        - To localize data for conserving memory bandwidth

- There is a gap between what the programmers need and what the tools provide today.

Key to Massive Parallelism - Regularity and Locality

Phillip Colella's "Seven dwarfs'

High-end simulation in the physical sciences = 7 numerical methods:

1. Structured Grids (including locally structured grids, e.g. Adaptive Mesh Refinement)
2. Unstructured Grids
3. Fast Fourier Transform
4. Dense Linear Algebra
5. Sparse Linear Algebra
6. Particles
7. Monte Carlo

- If add 4 for embedded, covers all 41 EEMBC benchmarks

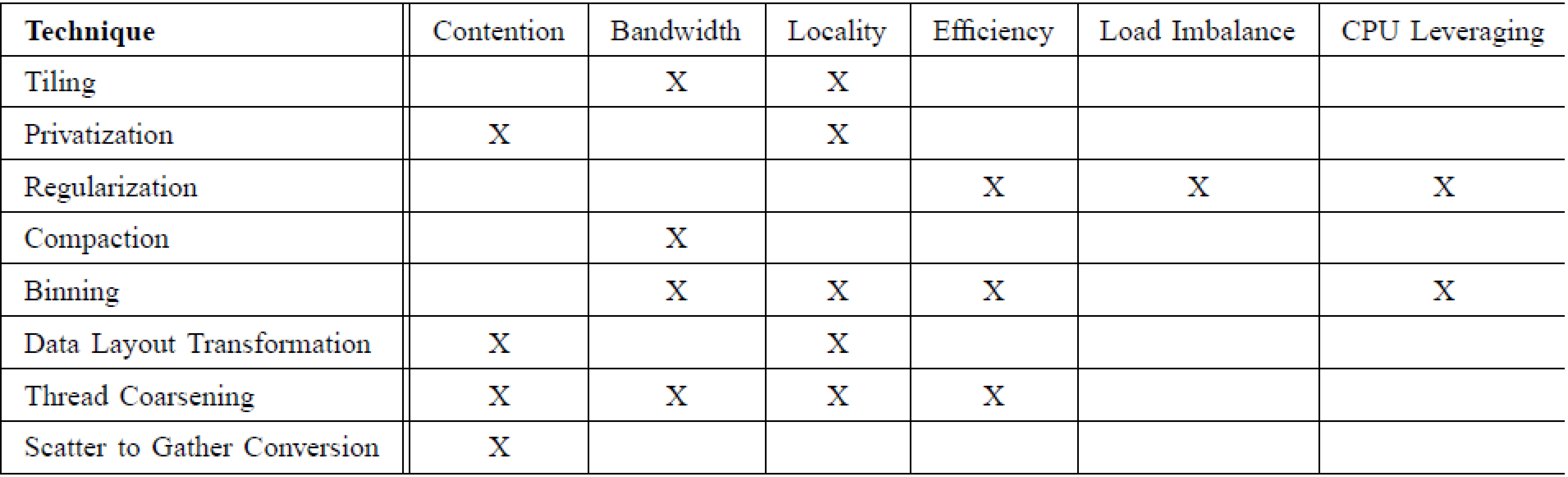
8. Search/Sort
9. Filter
10. Combinational logic
11. Finite State Machine

Well-defined targets from algorithmic, software, and architecture standpoint

Slide from "Defining Software Requirements for Scientific Computing", Phillip Colella, 2004

Courtesy: David Patterson

Eight Optimization Patterns for Algorithms (so far)

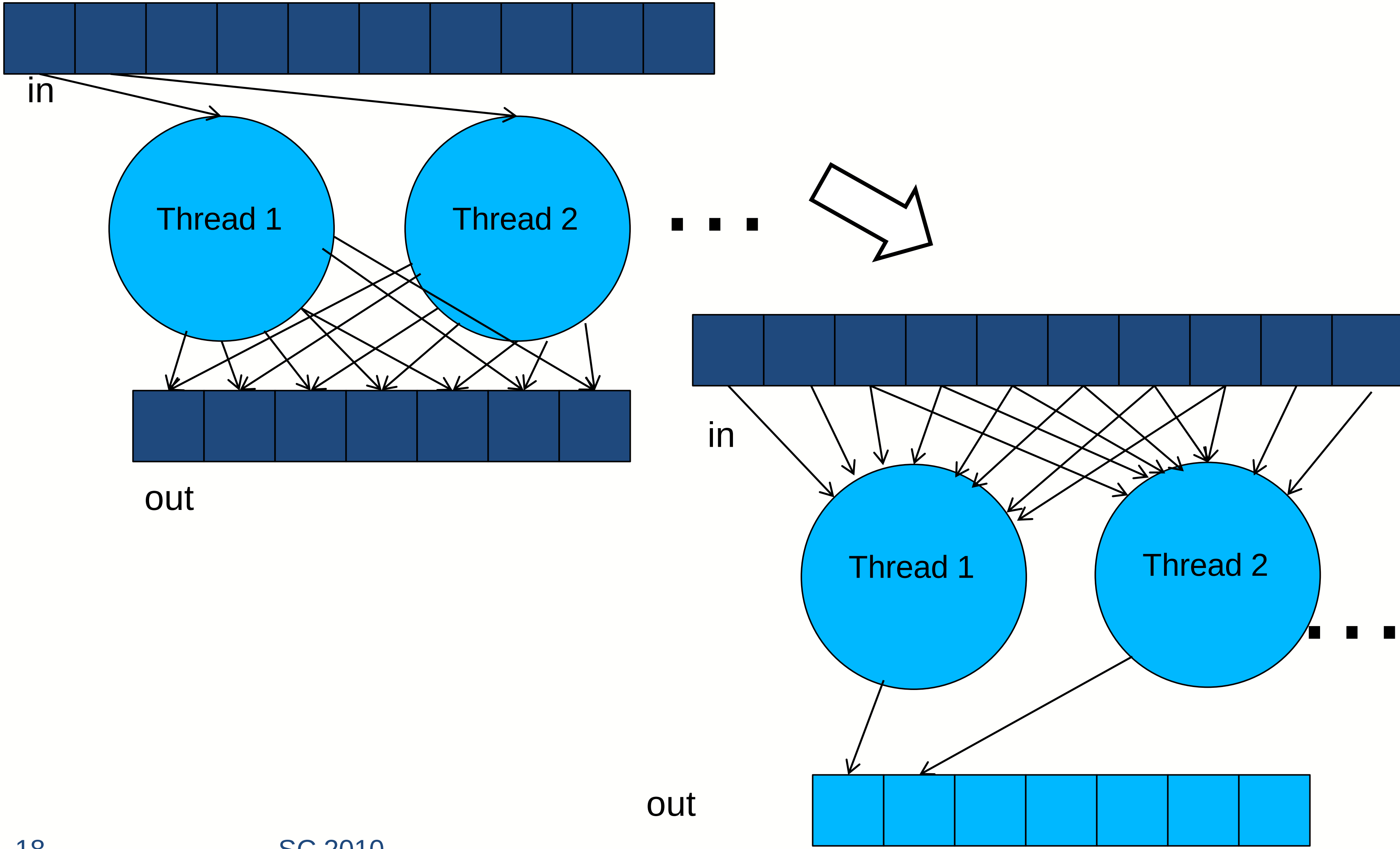


Technique	Contention	Bandwidth	Locality	Efficiency	Load Imbalance	CPU Leveraging
Tiling		X	X			
Privatization	X		X			
Regularization				X	X	X
Compaction		X				
Binning		X	X	X		X
Data Layout Transformation	X		X			
Thread Coarsening	X	X	X	X		
Scatter to Gather Conversion	X					

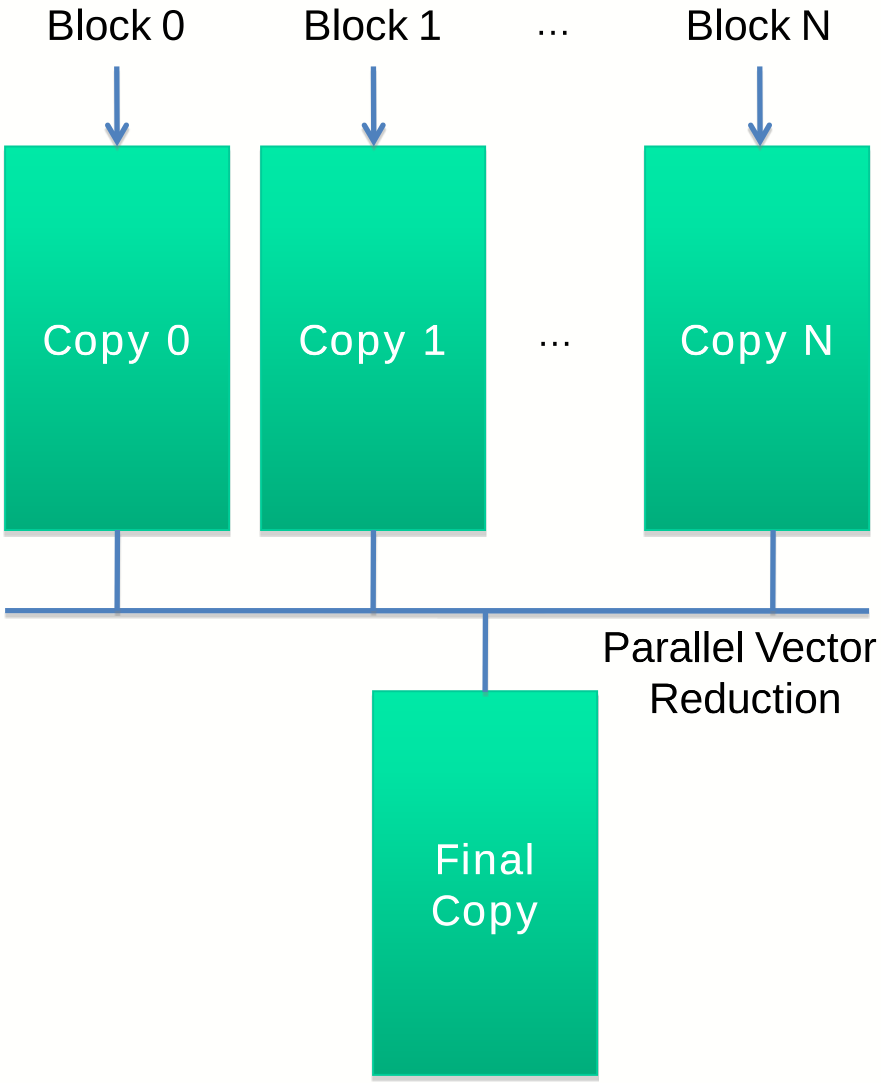
http://courses.engr.illinois.edu/ece598/hk/

GPU Computing Gems, Vol. 1 and 2

1: Scatter to Gather Transformation



2. Privatization

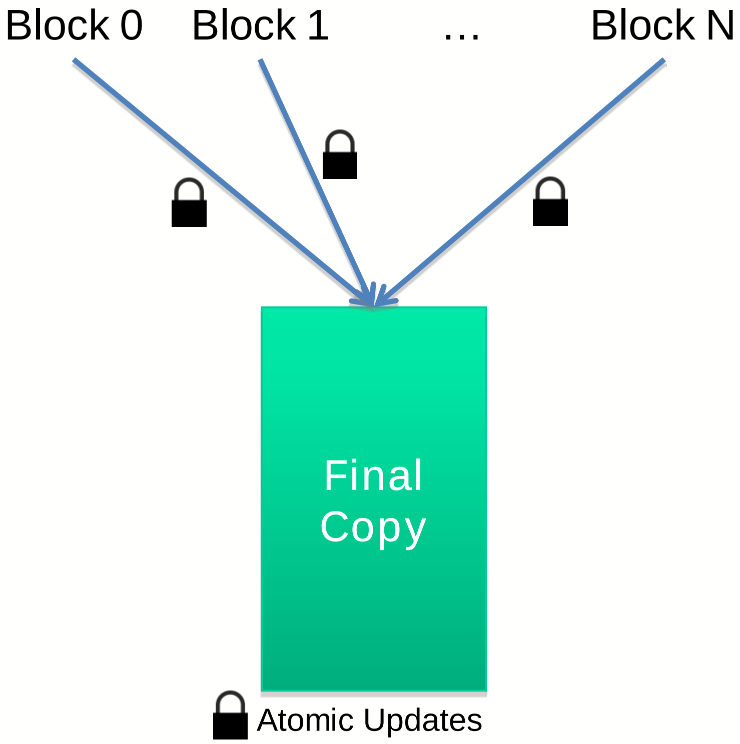


Block 0 Block 1 … Block N

Copy 0 Copy 1 … Copy N

Parallel Vector Reduction

Final Copy



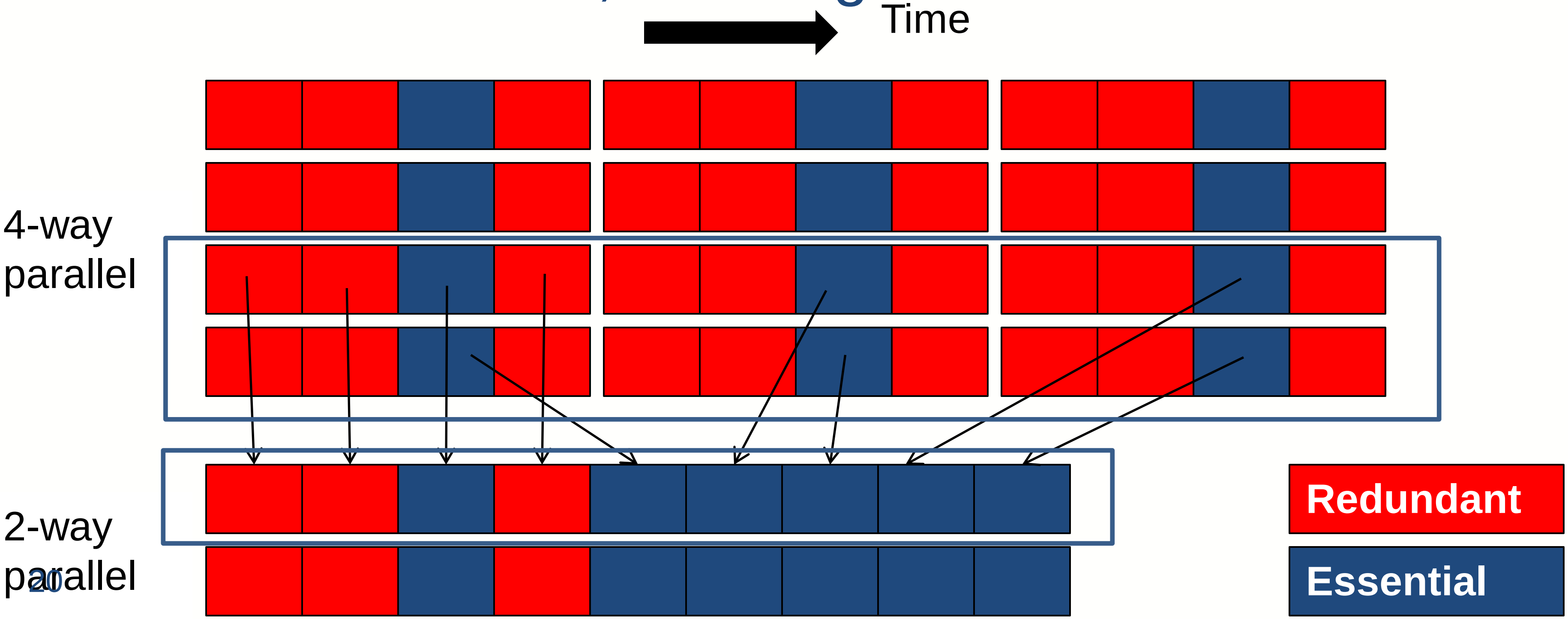
Block 0 Block 1 … Block N

Final Copy

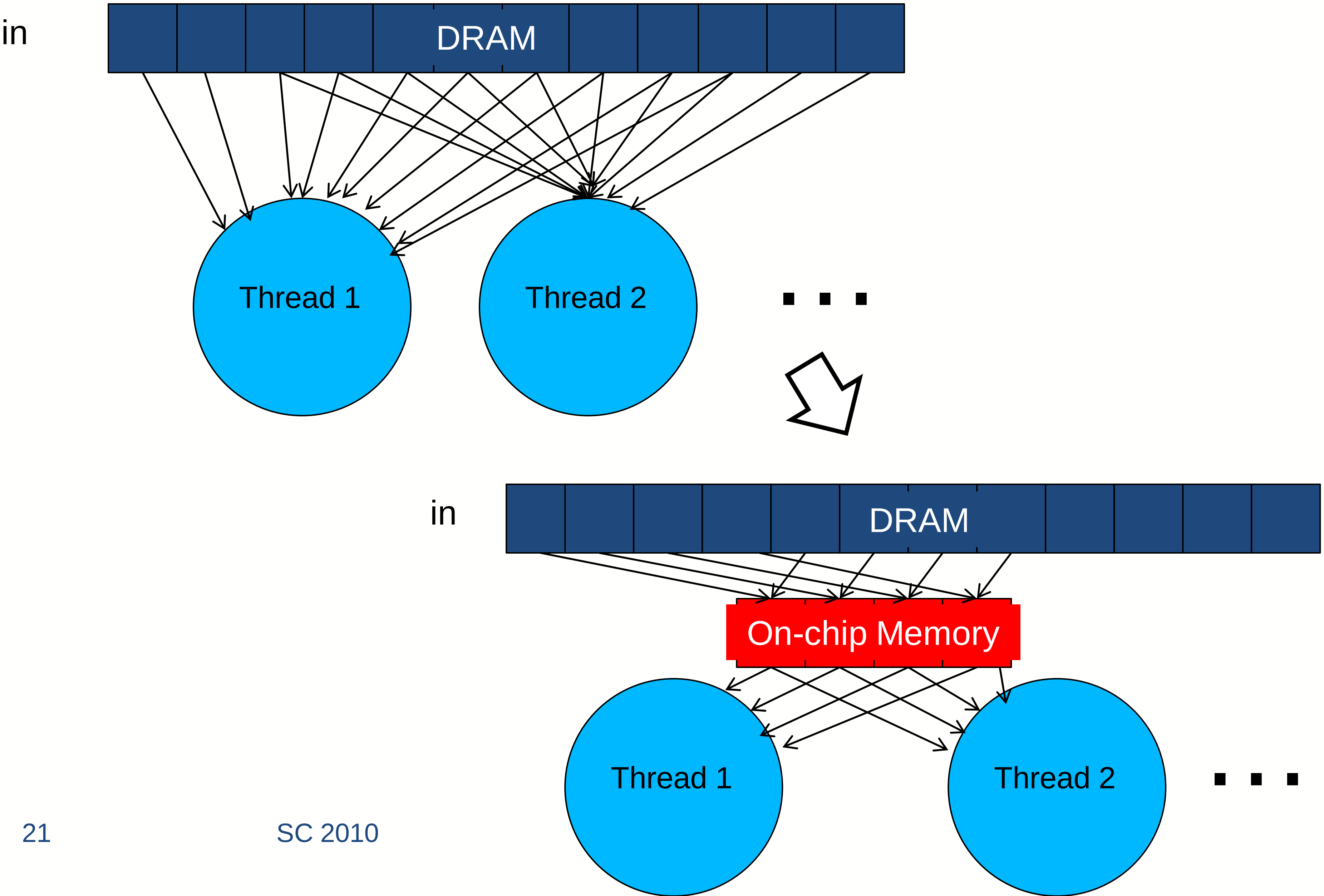
Atomic Updates

3. Granularity Coarsening and Register Data Reuse

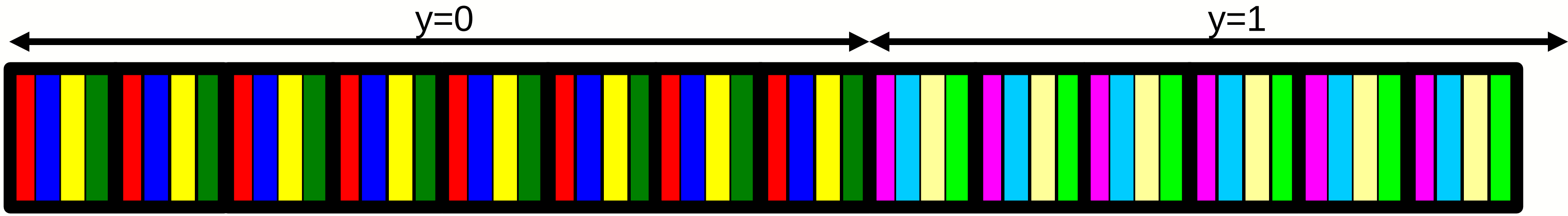
- Parallel execution often requires redundant and coordination work
  - Merging multiple threads into one allows re-use of result, avoiding redundant work



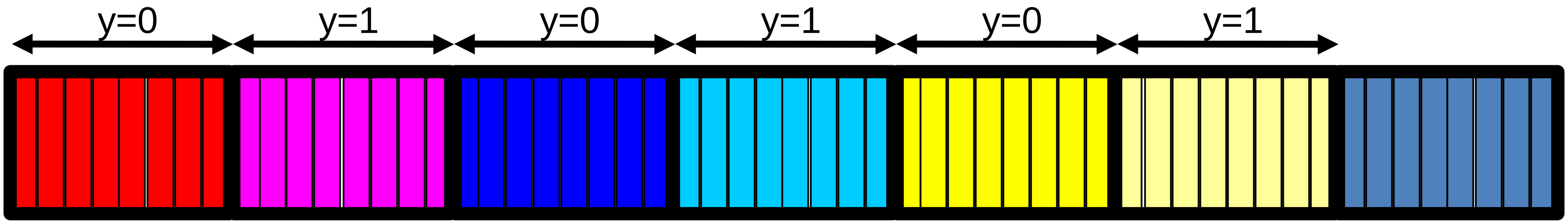
4: Data Access Tiling



5. Data Layout Transformation

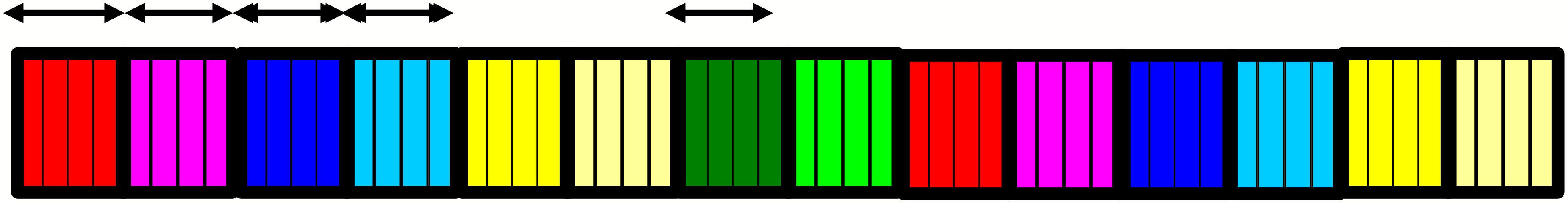


Array of Structure: [z][y][x][e]



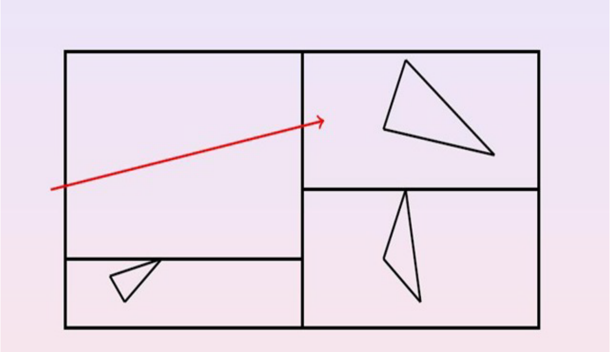
Structure of Array: [e][z][y][x]

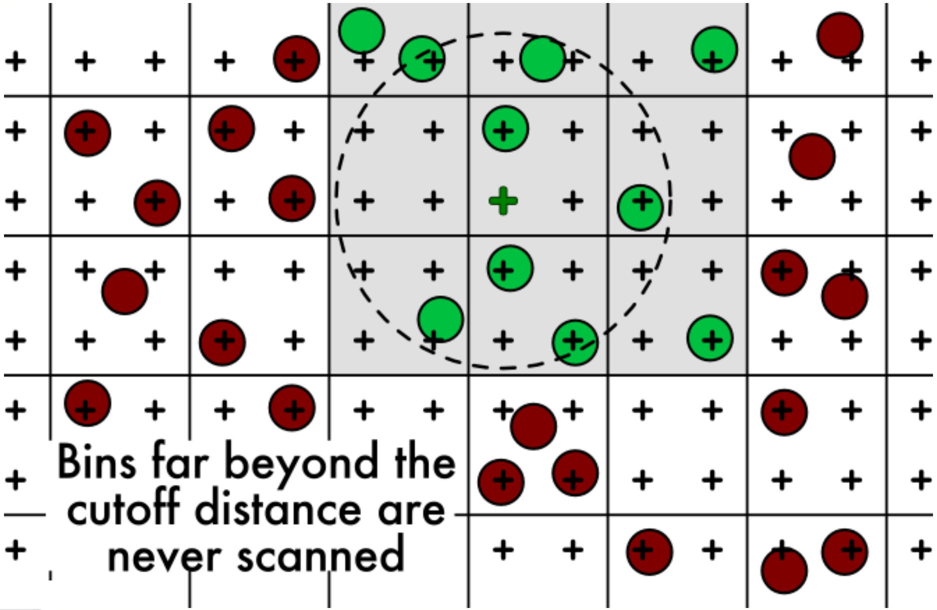
4X faster than AoS on GTX280



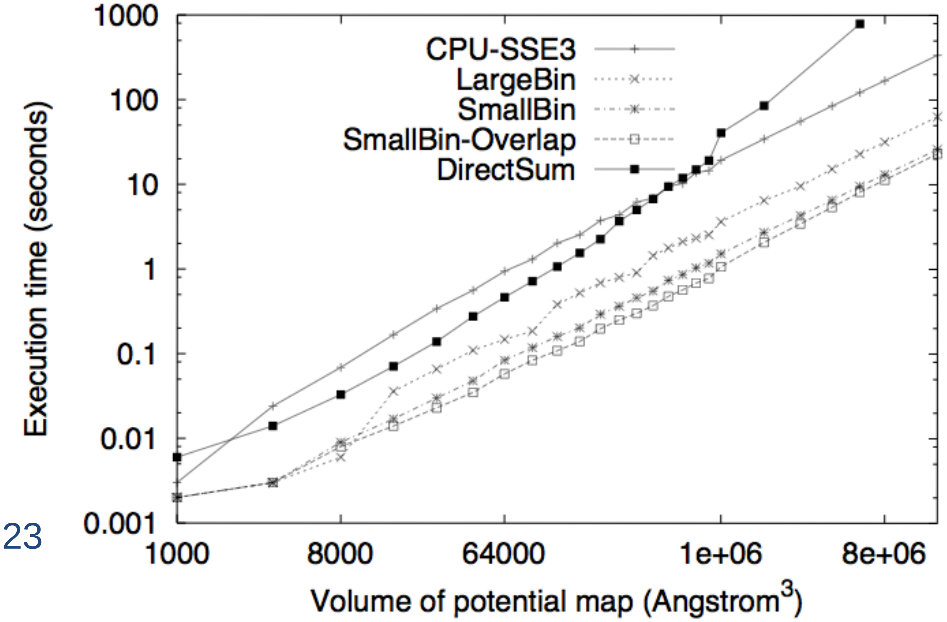
$[z][y_{31:4}][x_{31:4}][e]$**$[y_{3:0}][x_{3:0}]$**, 6.6 X faster than AoS

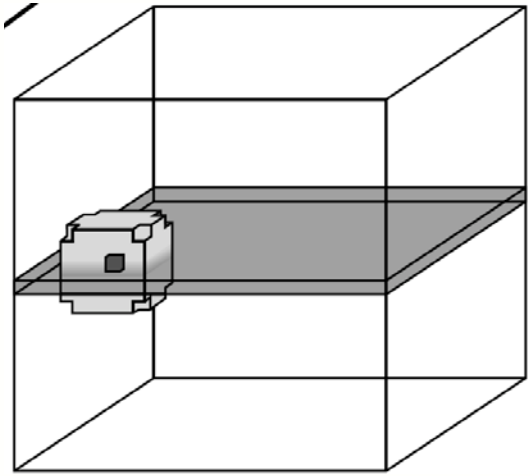
6: Input Data Binning





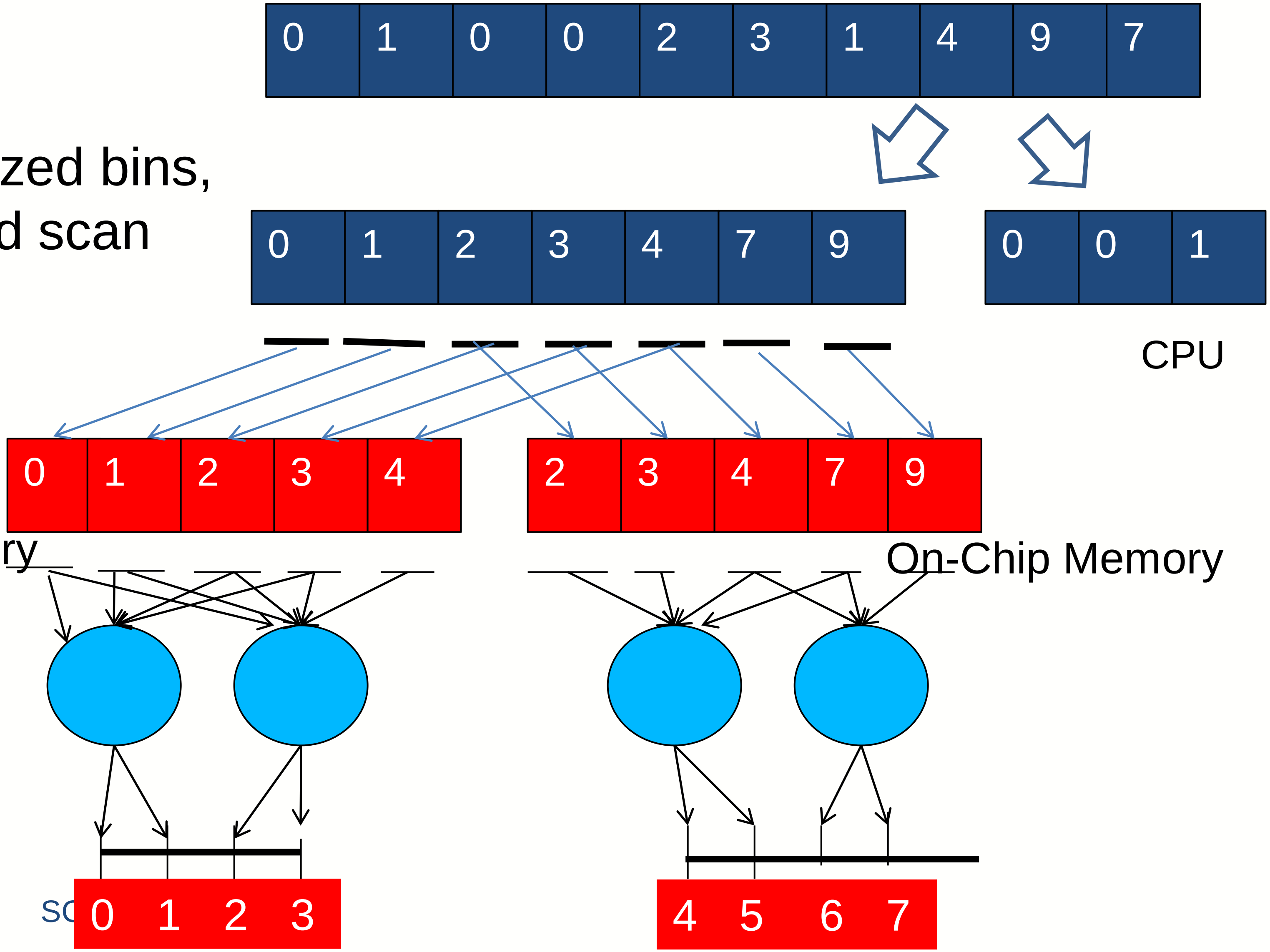
Bins far beyond the cutoff distance are never scanned



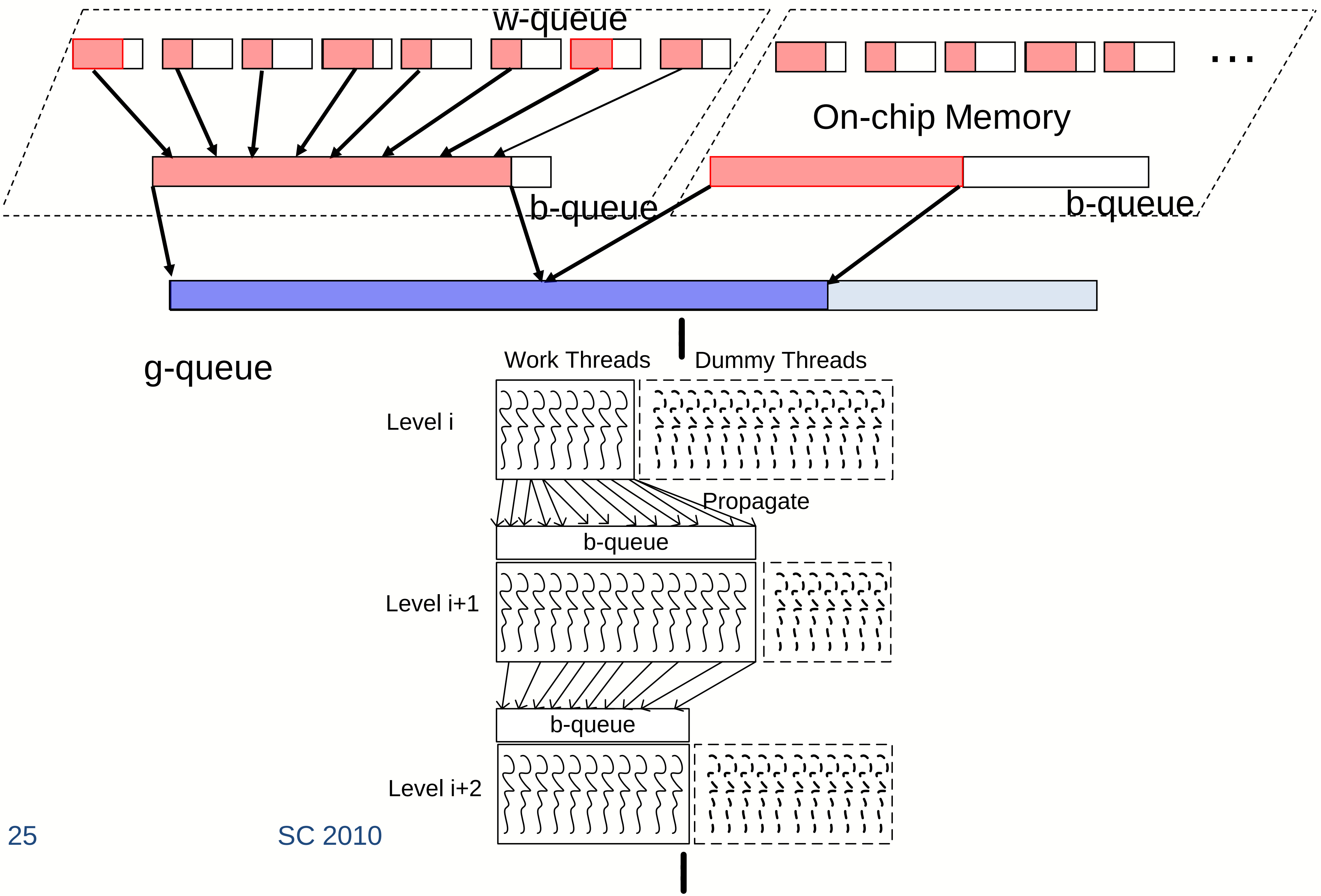


7. Compaction

Variable sized bins,
sort and scan



8. Regularization



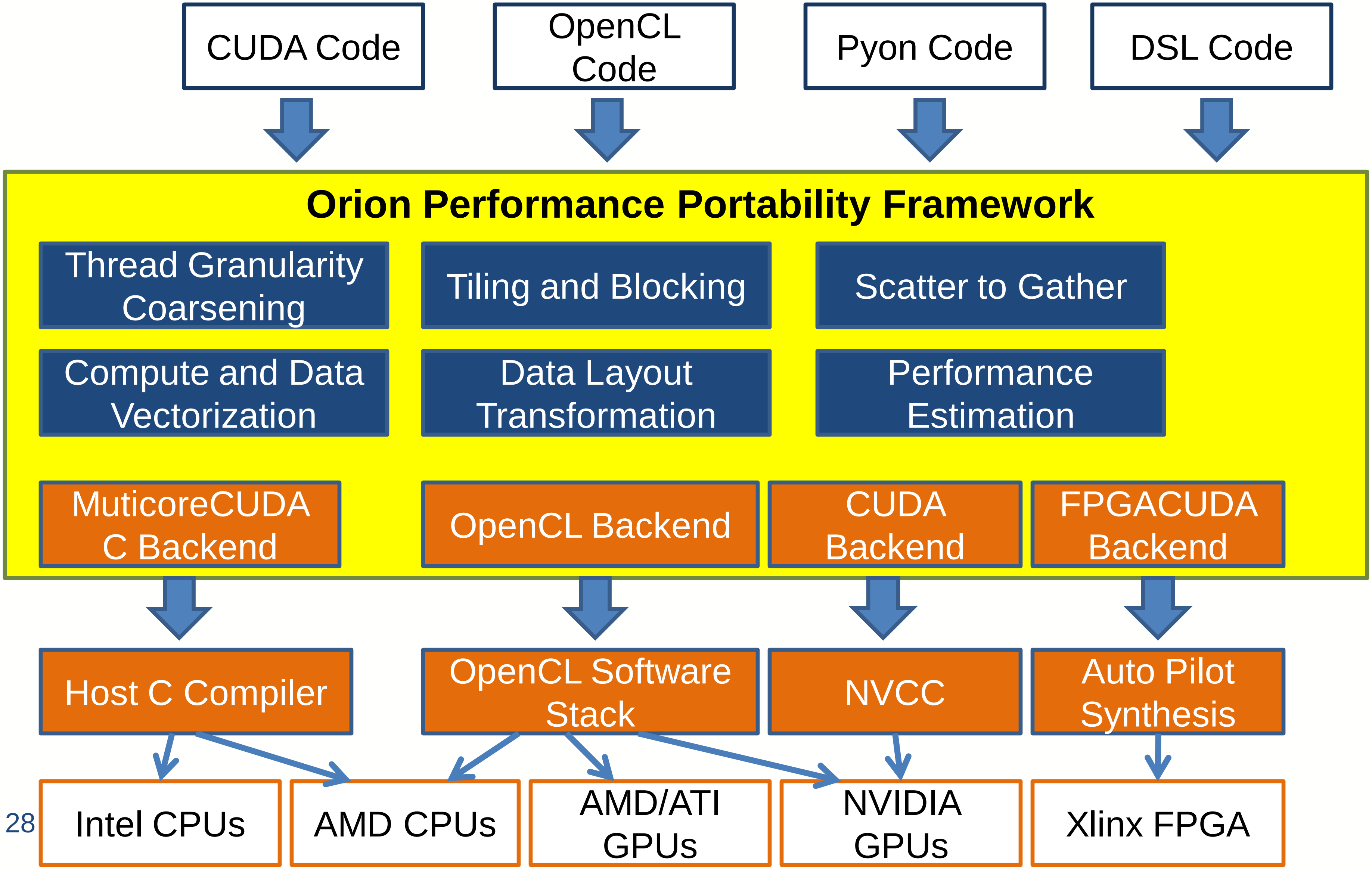
Tools go with techniques.

- 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, bin sorting, and hierarchical queues are much harder
    - Need to provide APIs understood by compilers/tools
    - Developer feedback critical to success

Some Current Efforts

- PGI FORTRAN (PGI)
  - FORTAN loop compilation into kernels
- Chapel (CRAY)
  - Chapel vector loop compilation into kernels
- Copperhead (Berkeley)
  - Annotated Python subset JITed into kernels
  - Modest performance within interpreter use model
- Pyon (Illinois)
  - Strongly typed Python subset compilation into kernels
  - Aggressive memory transformations

Orion: Reducing Performance Cost of Heterogeneous Parallelism



CUDA Code

OpenCL Code

Pyon Code

DSL Code

Orion Performance Portability Framework

Thread Granularity Coarsening

Tiling and Blocking

Scatter to Gather

Compute and Data Vectorization

Data Layout Transformation

Performance Estimation

MuticoreCUDA C Backend

OpenCL Backend

CUDA Backend

FPGACUDA Backend

Host C Compiler

OpenCL Software Stack

NVCC

Auto Pilot Synthesis

Intel CPUs

AMD CPUs

AMD/ATI GPUs

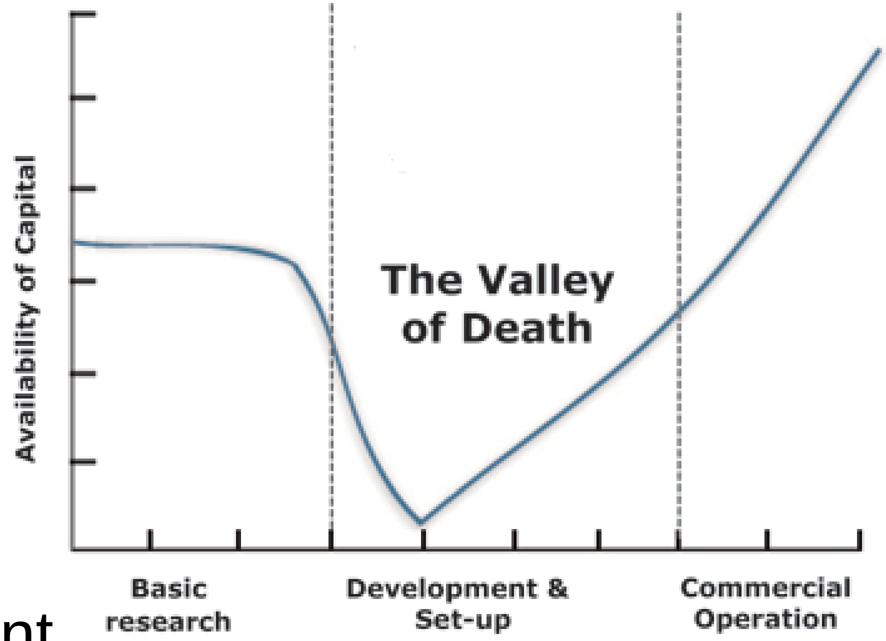
NVIDIA GPUs

Xlinx FPGA

Conclusion and Outlook

- Standard and domain library kernels are main use model of many-cores.
  - Kernel development is heavily lifting
  - Useful abstractions will require compilers to do some heavy lifting.
- However, compilers/tools are fragile.
  - Compilers transformations need to be part of the development, rather than afterthought
  - Developers must be able to reach into the abstraction whenever they want.

Crossing the Valley of Death



By giving developers what they want.

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. Gropp (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. Ryoo (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!