Portability, Scalability, and Numerical Stability in Accelerated Kernels

John Stratton

Doctoral Candidate: University of Illinois at Urbana-Champaign

Senior Architect: MulticoreWare Inc

Outline

- Performance Portability
  - What CPU programmers need to learn from GPU computing
  - Corollary/Takeaway: Most developers should only write their code once
- Building a robust high-performance parallel codebase
  - Some things GPU programmers need to learn from the rich CPU library development history
  - Corollary/Takeaway: Performance isn't everything

High-performance GPU Software Needs…

Scalability

- Thousands of threads
- Algorithms we choose now have to be very parallel to last for years to come

Locality

- Randomly accessed global memory is slow

Regularity

- SIMD matters a lot

High-performance CPU Software Needs…

Scalability

- Number of cores continues to grow

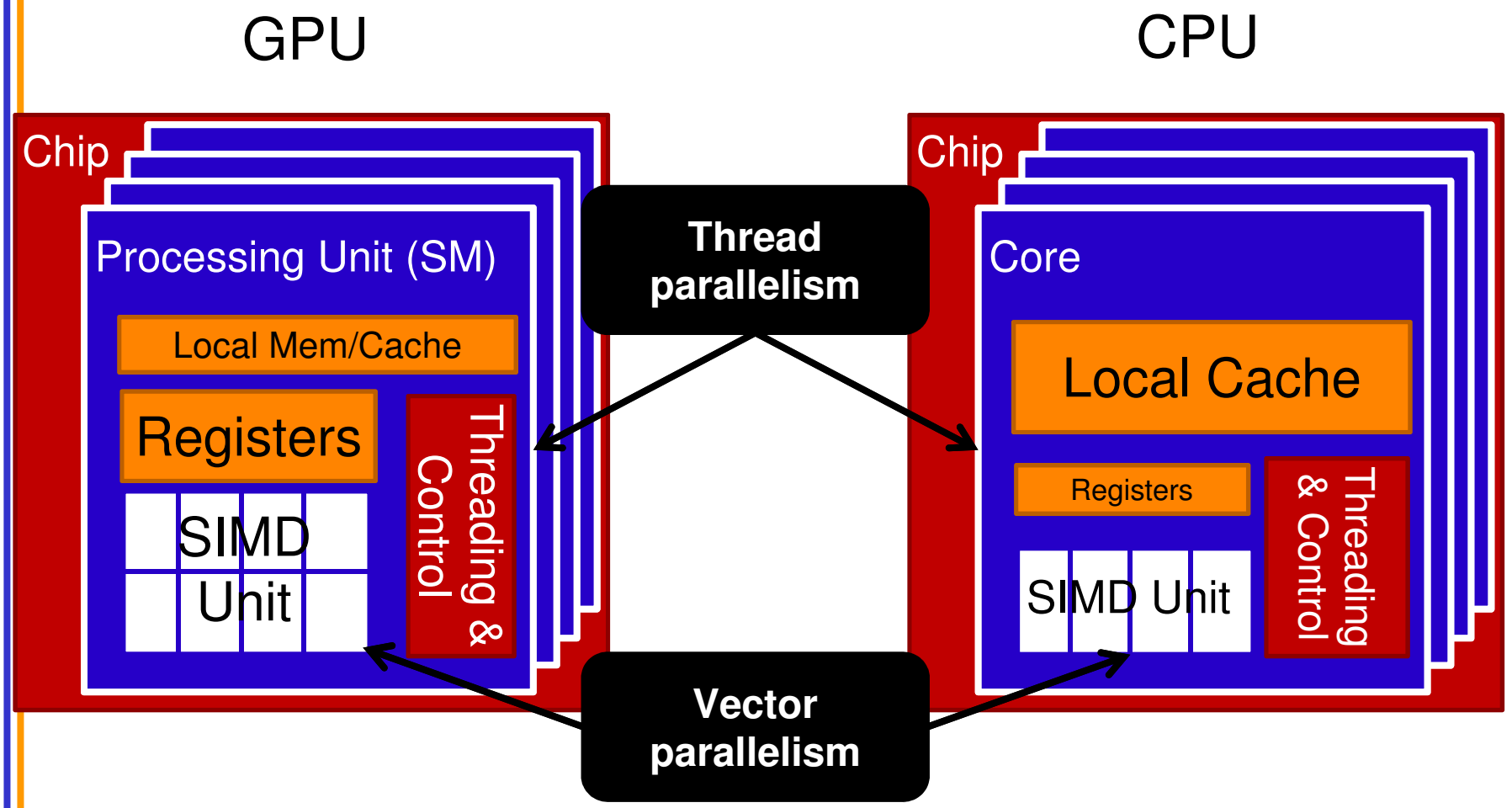
Locality

- Randomly accessed global memory is still slow

Regularity

- x86 SIMD widths trending up in particular

Simplistic Architecture Comparison



So, why write two versions of code?

You'll do a better job the second time.

A fast CPU version is easy to write so it doesn't really count.

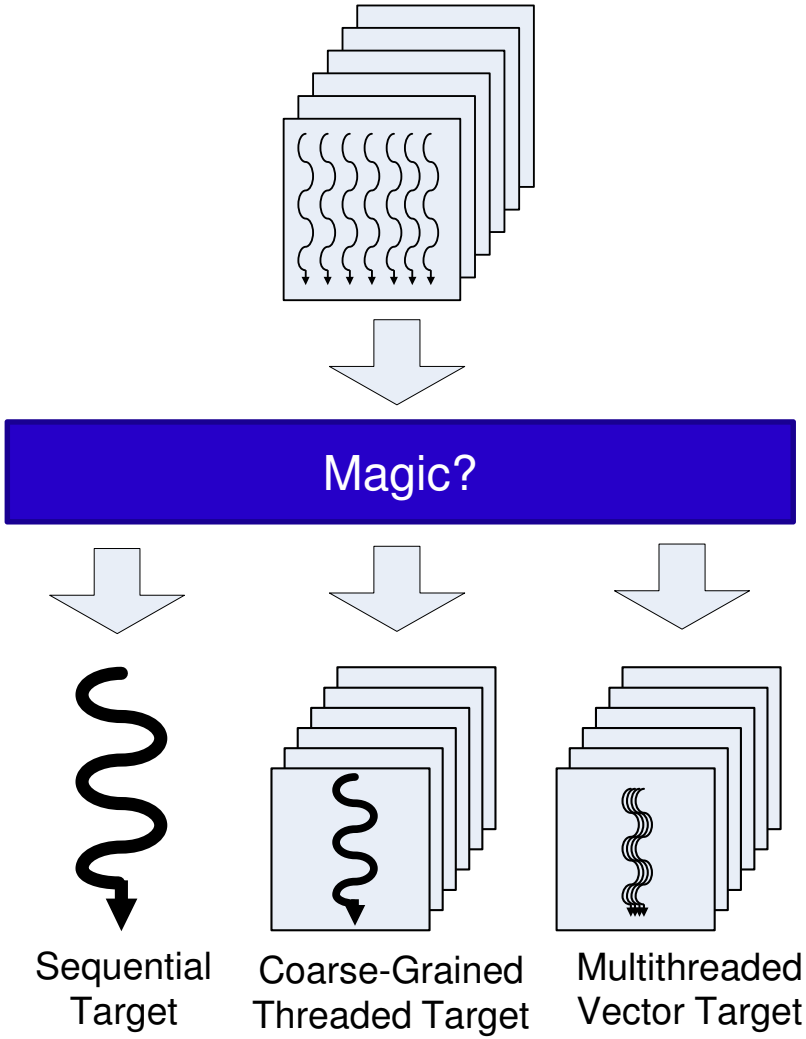
I get paid per line of code written, even if it just implements duplicate features.

</sarcasm>

GPUs and CPUs have incompatible ideas about "threads," and relationships between thread- and instruction-level parallelism.

What if there were tools to…

- Let the programmer ensure locality, SIMD-friendliness, etc.
- Adapt task granularity to target architecture and task scheduler
  - Biggest portability hurdle
- Generates multithreaded C or x86 code just like a normal compiler

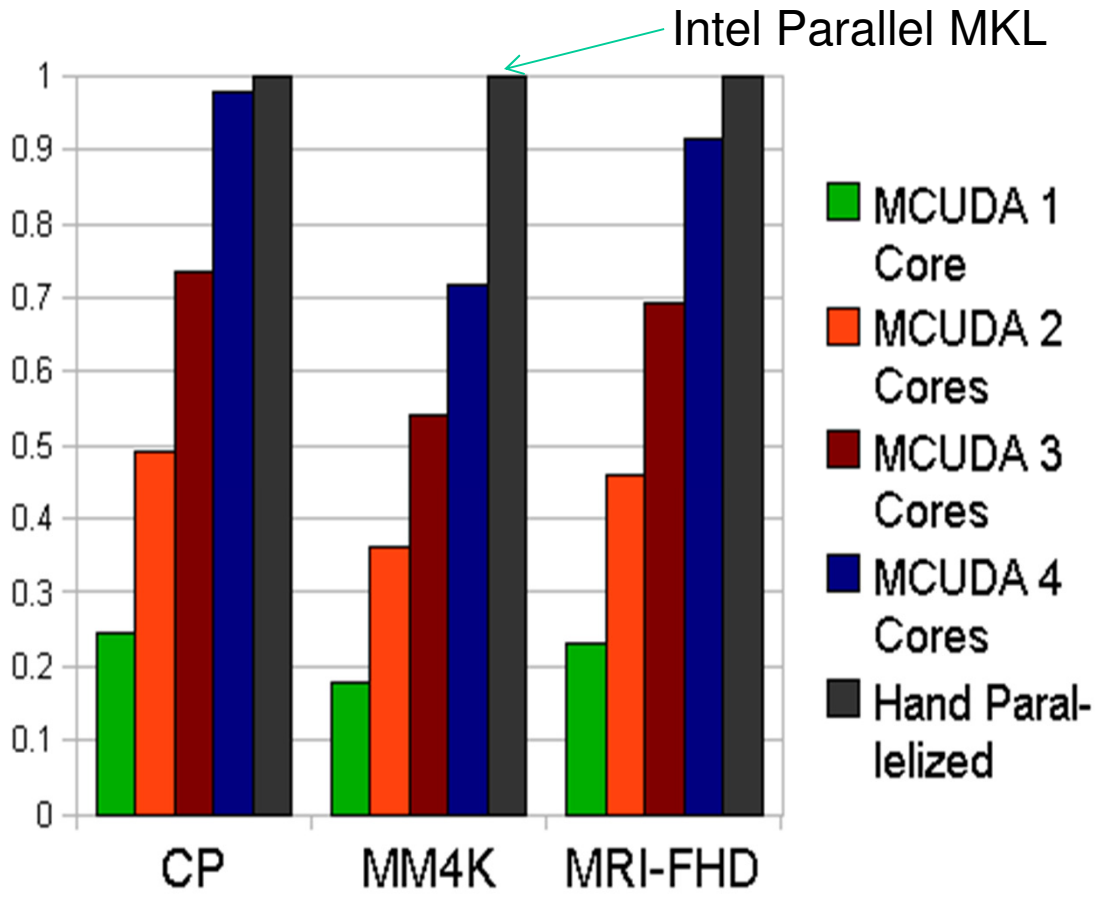


There are

- OpenCL implementations for x86 (Intel, AMD, MulticoreWare)
- PGI CUDA-x86 Compiler
- MCUDA

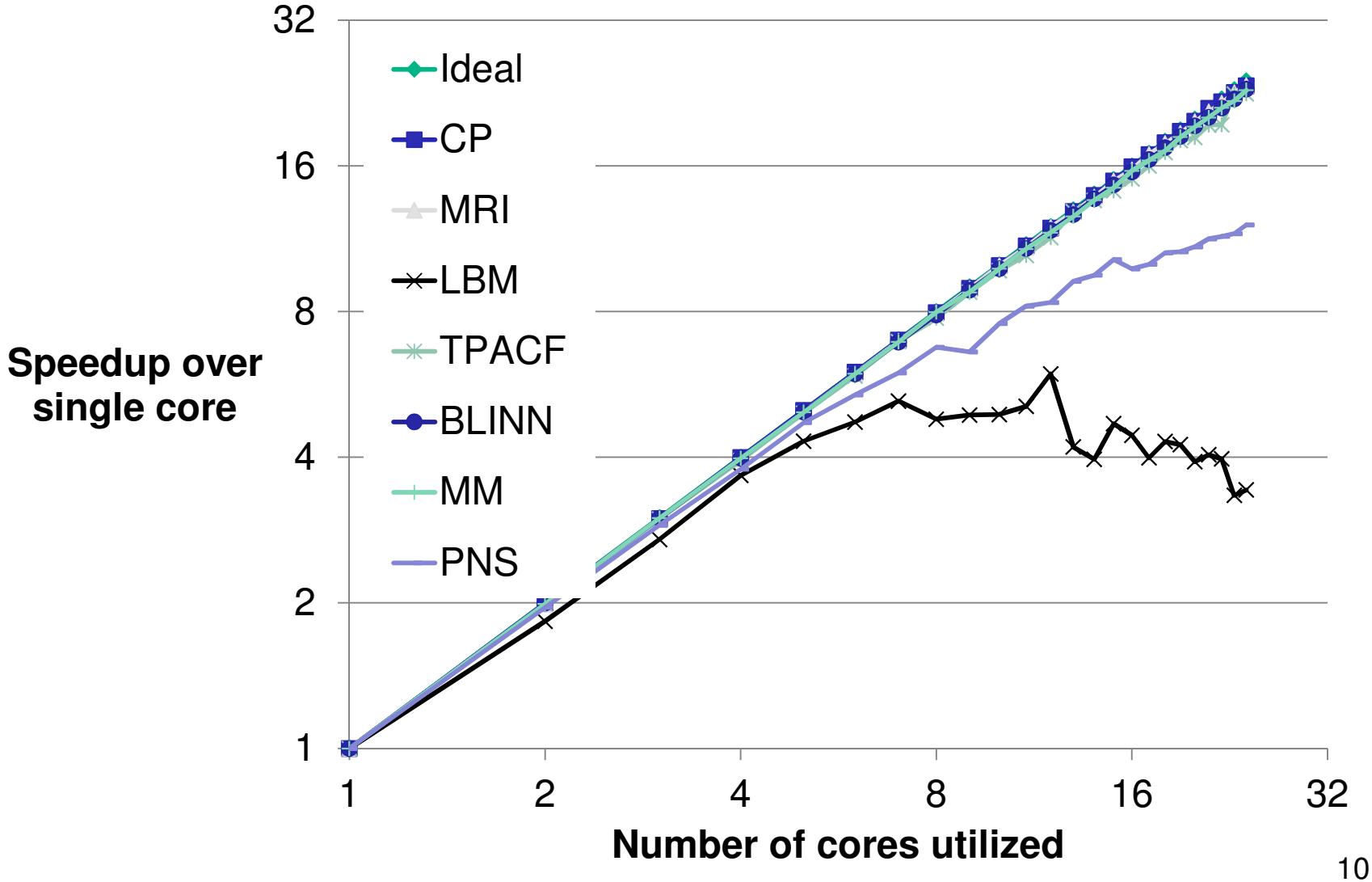
- Some are better than others: don't judge the principle based on one immature tool

Academic Proof of Concept



- Performance (reciprocal runtime) of MCUDA-translated app, normalized to hand-parallelized CPU code.

With scalable parallelism



Why this works well

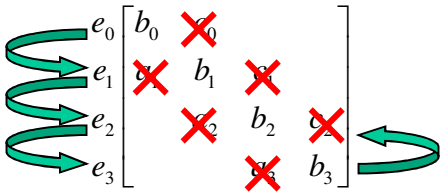
- High-performance programming is all about
  - Massive Parallelism
  - Locality and Hierarchy
  - Regularity
- Obviously true for GPUs
- Obviously true for clusters
- Becoming more true for CPUs each year
- Good CUDA optimizations are often good CPU optimizations
- Good CPU algorithms are often good GPU algorithms

*Thanks to Li-Wen Chang & Ray Sung for some of the content in this section

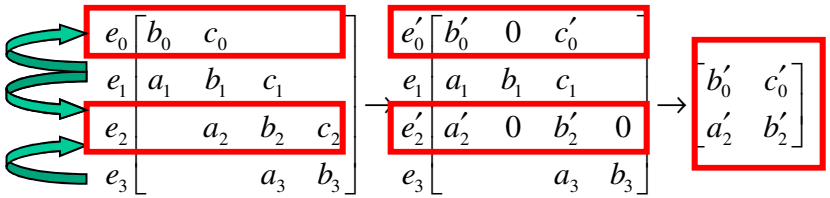
LEARNING TO WRITE LIBRARIES FROM THE EXPERTS

GPU Tridiagonal System Solver Case Study

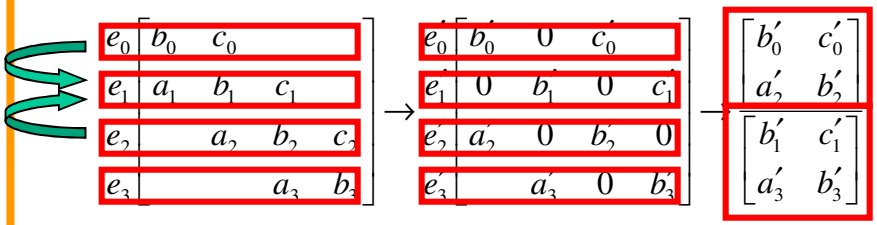
- Thomas (sequential)



- Cyclic Reduction (1 step)



- PCR (1 step)



- Hybrid Methods
  - PCR-Thomas (Kim 2011, Davidson 2011)
  - CR-PCR (CUSPARSE 2012)
  - Etc

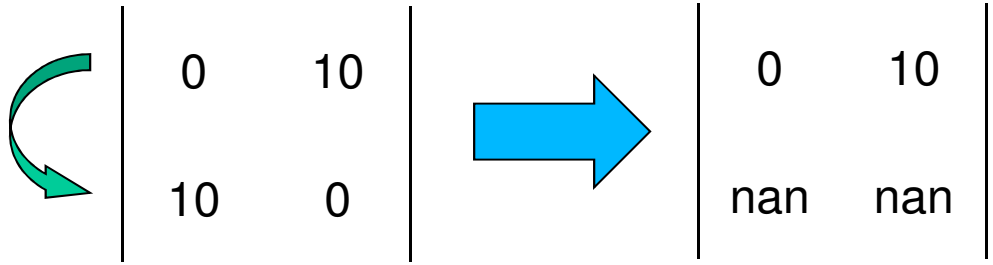
- CPU libraries use none of these: Numerically unstable

Numerical Stability

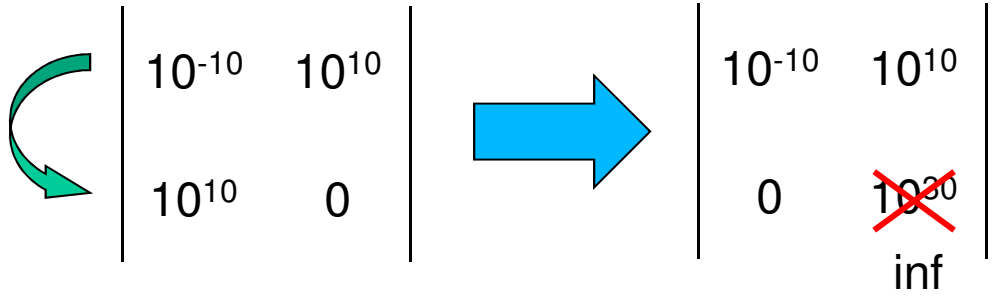
- An algorithm is *numerically stable* if it can always find an appropriate solution to the problem for any given input values, assuming one exists.
- Algorithms that fall short of this requirement are referred to as *numerically unstable*.

Examples of numerical instability

- Algorithms that don't check for divide by zero

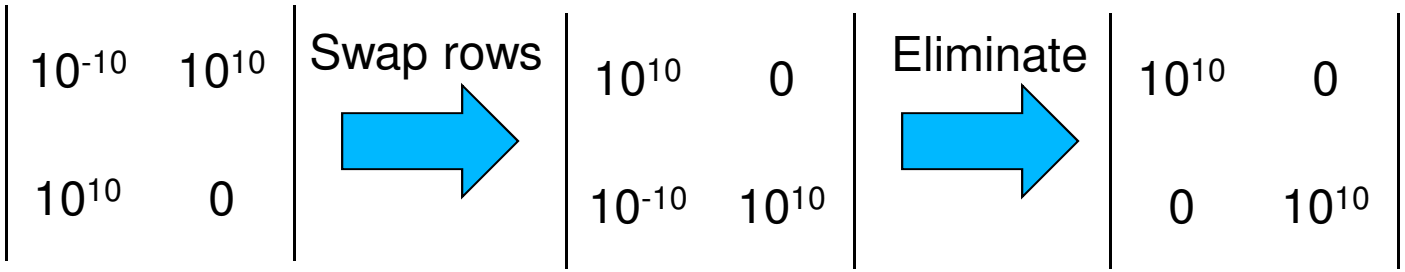


- Limited ability to represent precision and scale



Pivoting: a core stability technique

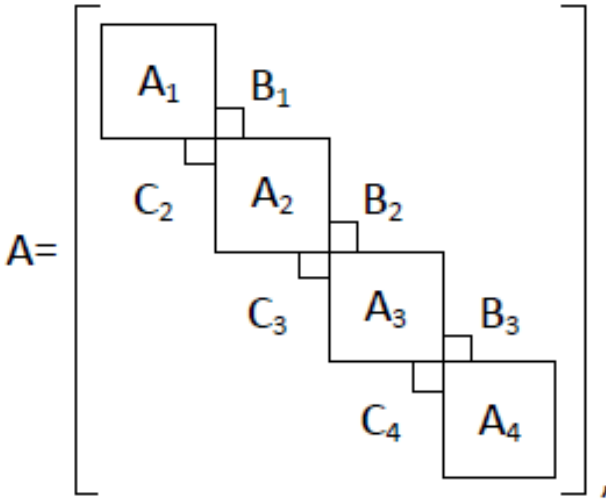
- Judiciously swap rows (or columns) to avoid bad cases

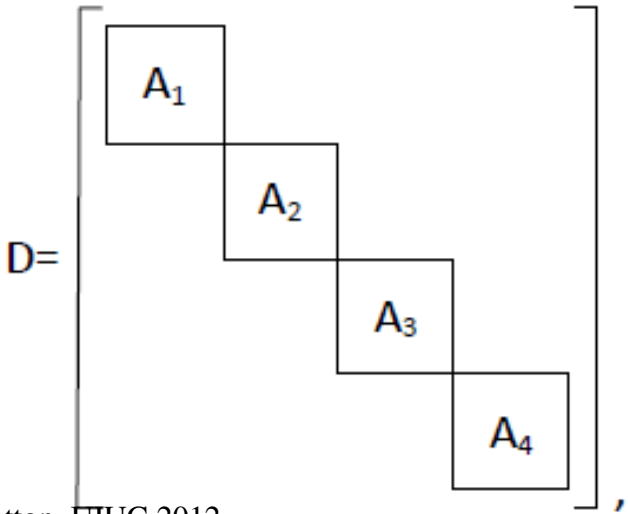


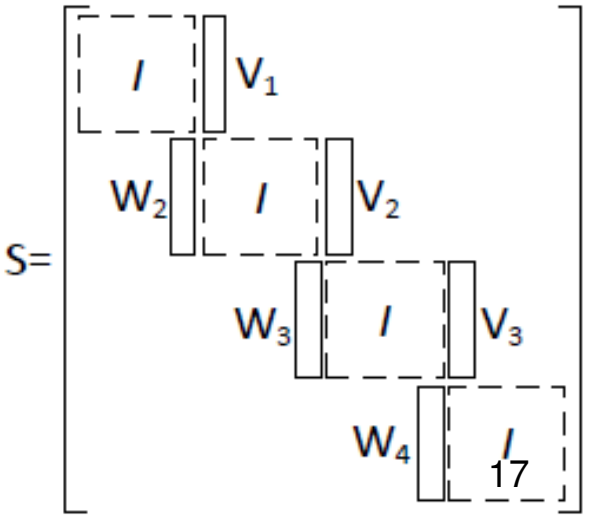
- Inherently sequential algorithm: we need more parallelism

SPIKE Partitioning Algorithm

- A numerically stable method for decomposing a banded system: A X = F
- Algebraically decompose A into D and S: D S X = F
- Compute D^{-1}
  - Solve by tiles
- Solve S X = F

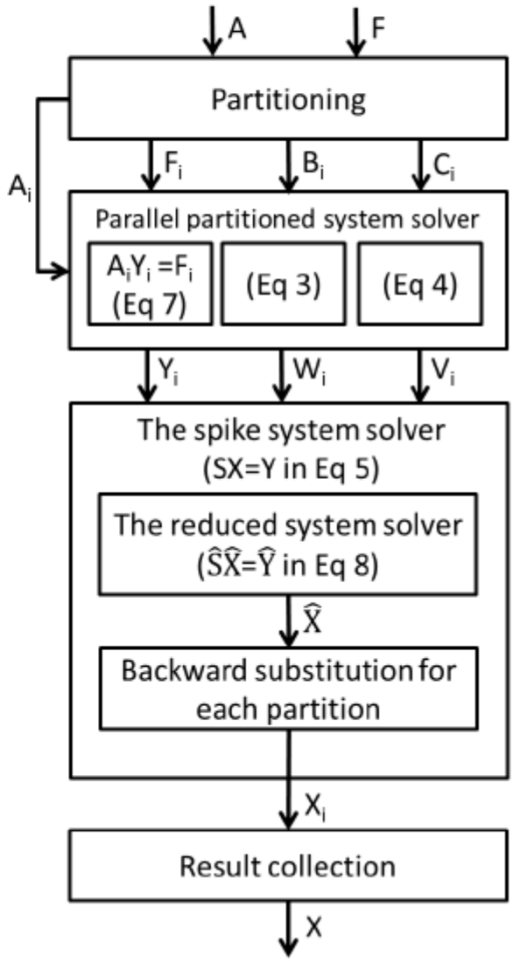






SPIKE Partitioning algorithm

- Creating S is just two more tiled inverse tridiagonal system problems (DV=F and DW=F)
  - Can be solved in parallel
- Solving S X = Y is a much easier problem because of the matrix structure
  - Takes at most 5% of total solution time



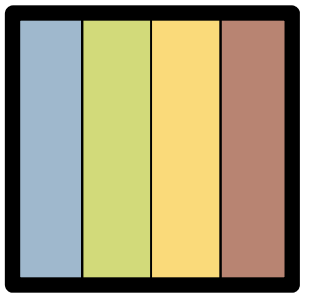
Put the stable sequential algorithm inside each GPU thread

- Each thread will process one tile by itself with a sequential, numerically stable pivoting algorithm

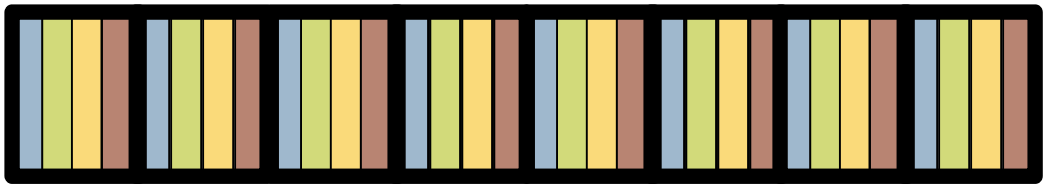
- Two problems
  - Each thread's first, second, etc. element are far away from the next thread's corresponding element of its own tile, resulting in large-strided accesses
  - Each thread consumes data from its tile at a different rate based on its pivoting decisions

Tiles Processed by Each Thread

- Each tile:



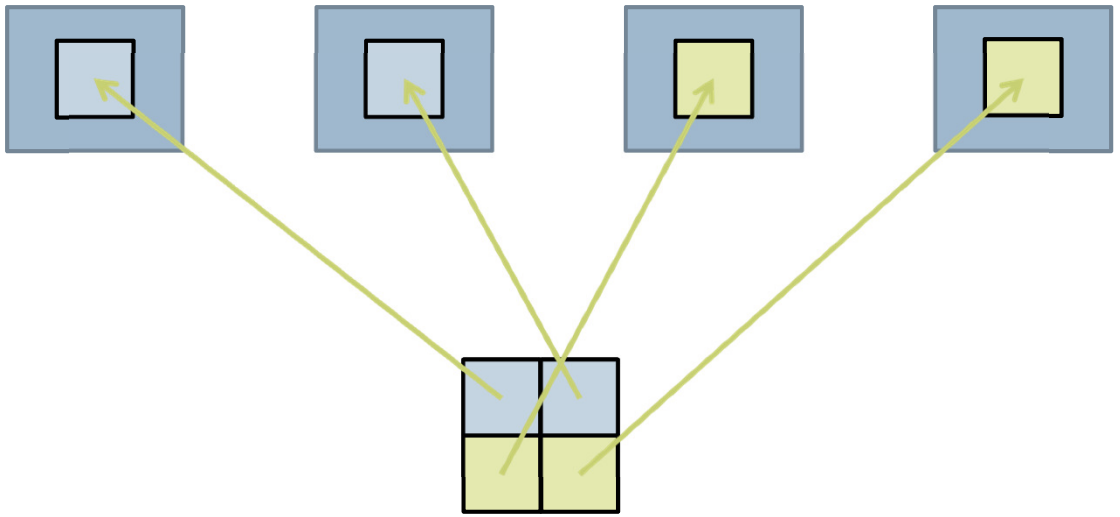
- Layout of all tiles:
  (similar to an array of structures layout)



- Let's do a transpose!
  - Out of place? 2X memory overhead.
  - In place? Genuinely difficult for arbitrary sizes.

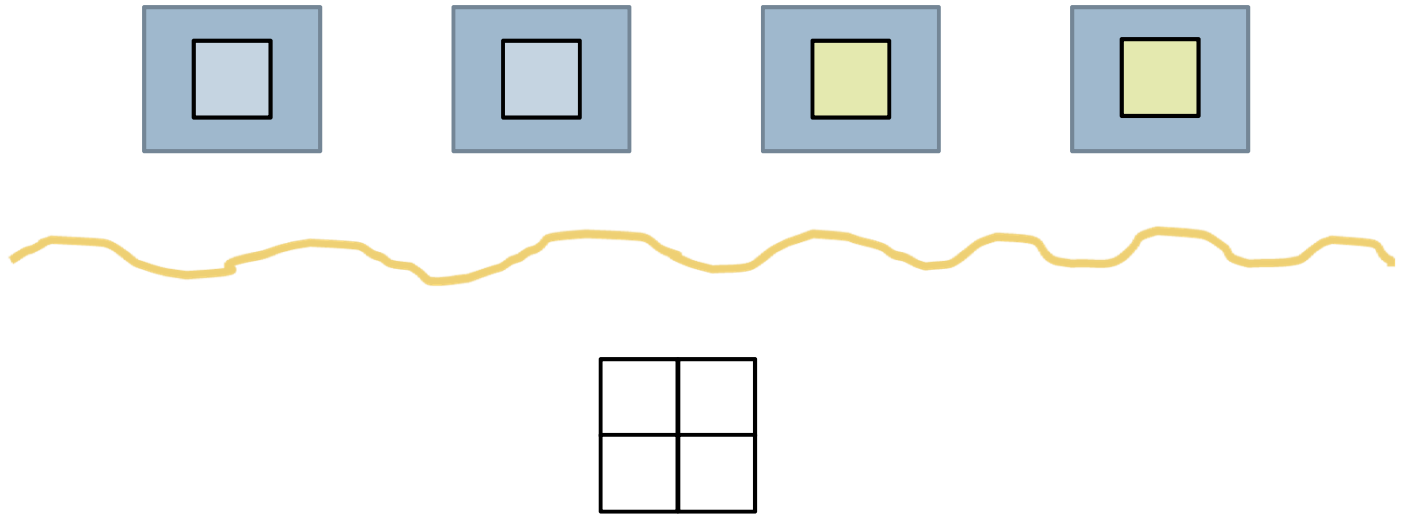
In-place Transpostion: simple case

```
// data[W][H]-->data[H][W]
parallel for (j<W)
  parallel for (i<H)
    float temp = data[j][i]; //offset = j*H + i
```



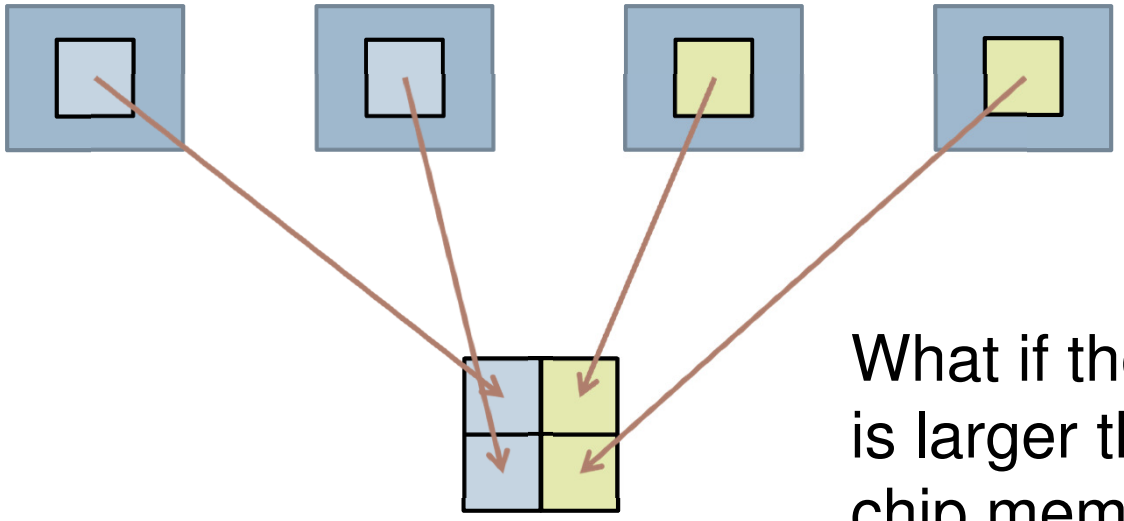
In-place Transpostion: First Attempt

```
// data[W][H]-->data[H][W]
parallel for (j<W)
  parallel for (i<H)
    float temp = data[j][i]; //offset = j*H + i
    barrier();
```



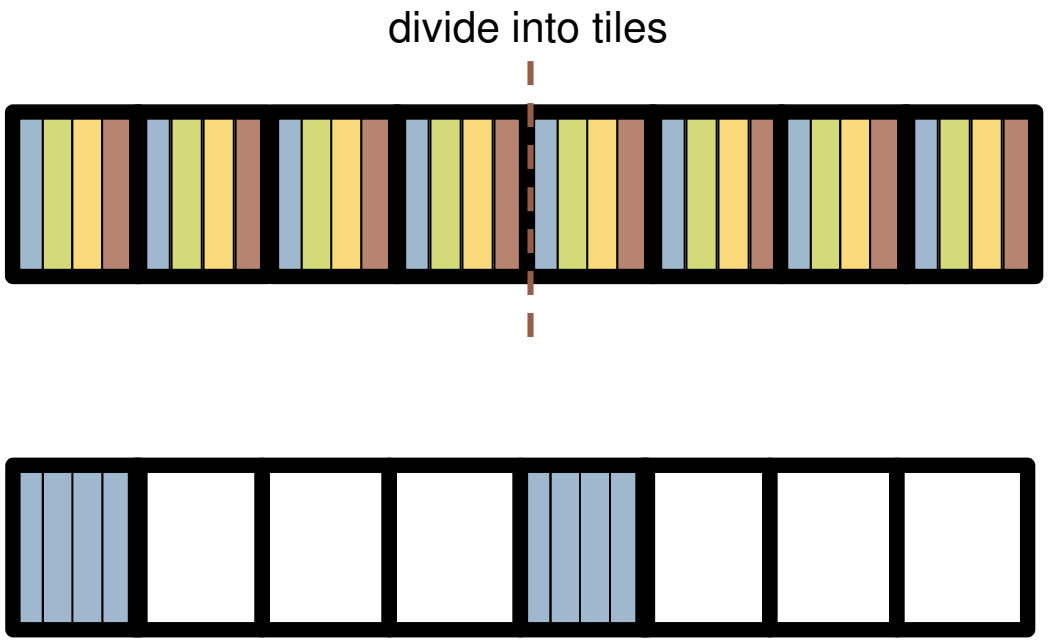
In-place Transpostion: First Attempt

```
// data[W][H]-->data[H][W]
parallel for (j<W)
  parallel for (i<H)
    float temp = data[j][i]; //offset = j*H + i
    barrier();
    data[i][j] = temp; //offset = i*W + j
```

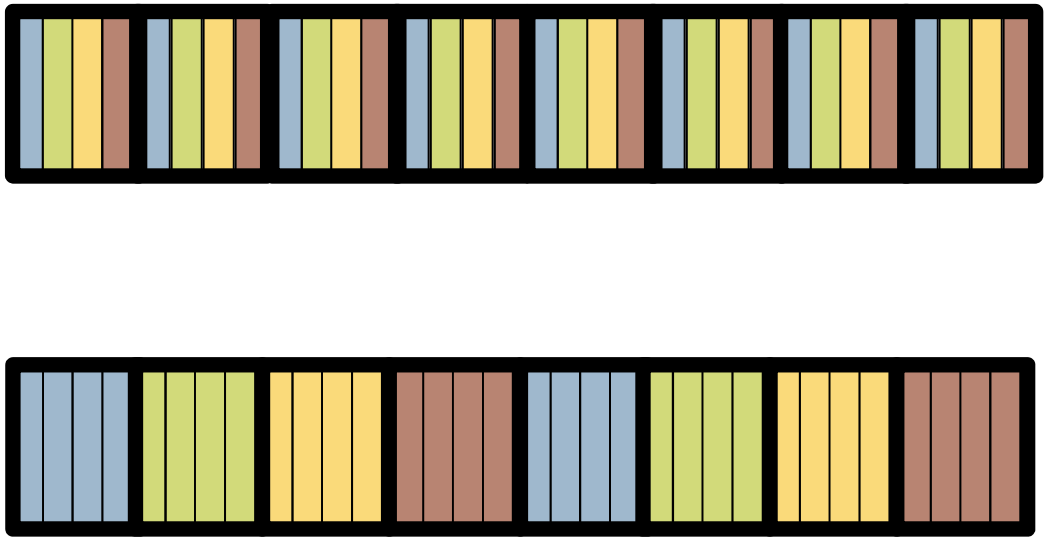


What if the dataset is larger than on-chip memory?

Another Data Layout Alternative



ASTA Data Layout

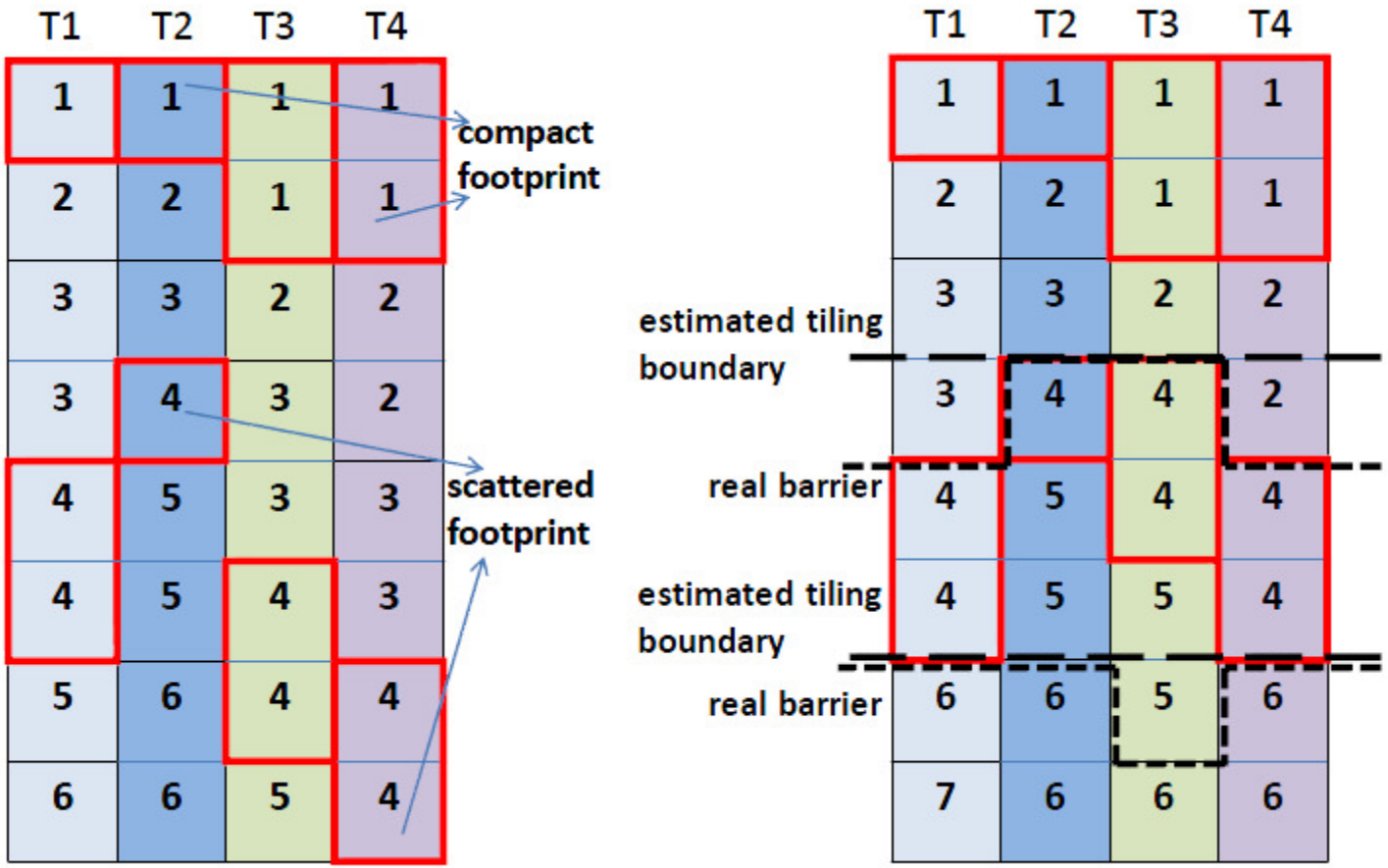


AoS to ASTA Transformation

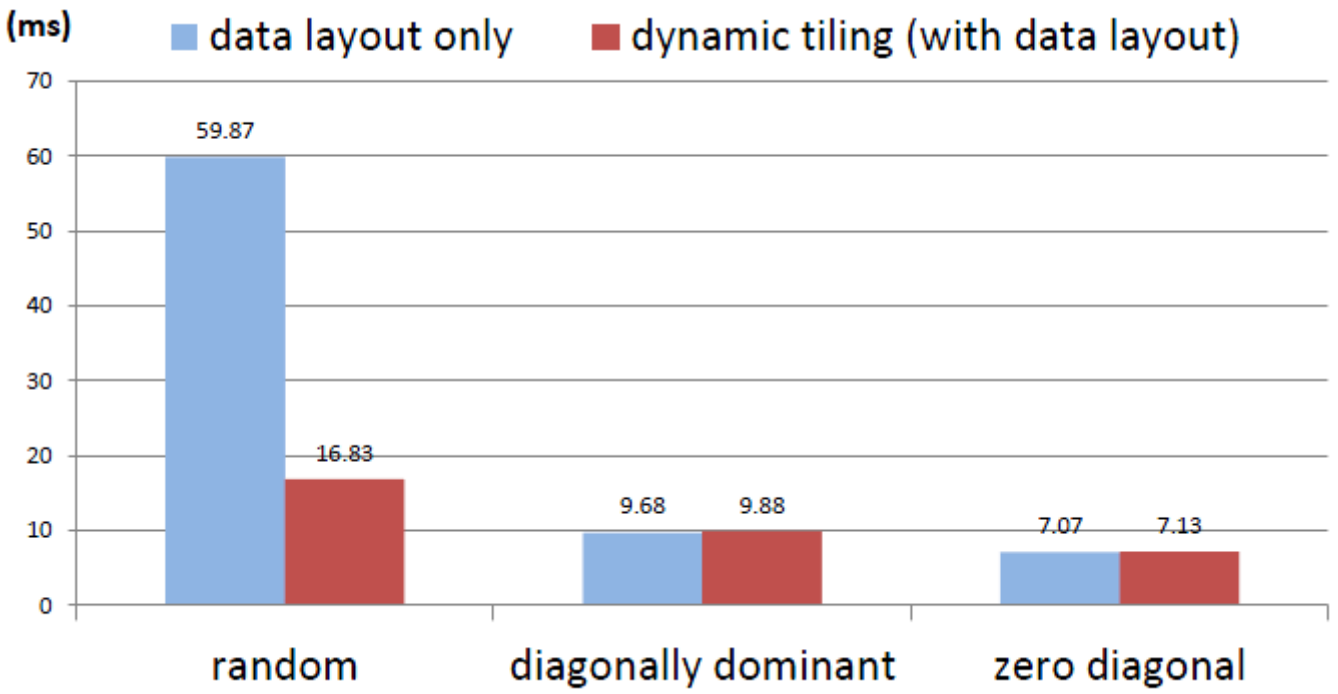
AoS to ASTA Marshaling Kernel	Global Memory Throughput (GB/s)	Fine Print
Out-of-Place	80	2x Space
In-Place Barrier Sync	95	Tile Size (tunable) < On-chip Memory

What if tile size > on-chip memory capacity ?
See Sung et al. “DL: A Data Layout Transformation System for Heterogeneous Computing,” InPar 2012

Dynamic tiling



Final performance results



Summary

- Learn good algorithms and good optimizations
  - State-of-the-art CPU algorithms are a great place to start for writing robust GPU libraries
  - State-of-the-art GPU optimizations are a great place to start for writing fast CPU code
- High-performance parallel computing major problems and techniques for solving them that are pretty common across architectures
  - Memory Locality -> Tiling & Layout
  - Execution Scalability -> Efficient, Parallel Algorithms
  - Limited Precision Computation -> Stable Algorithms

More information?

MCUDA: http://impact.crhc.illinois.edu/mcuda.aspx

Stratton et al. “Efficient Compilation of Fine-grained SPMD-threaded Programs for Multicore CPUs,” CGO '10
Not currently maintained.

MxPA: See MulticoreWare press release, contact info@multicorewareinc.com for more information

http://multicorewareinc.com/index.php?option=com_content&view=article&id=74&Itemid=86

Tridiagonal Solver Library: coming up in SC ‘12,
Chang et al. “A Scalable, Numerically Stable, High-performance Tridiagonal Solver using GPUs”