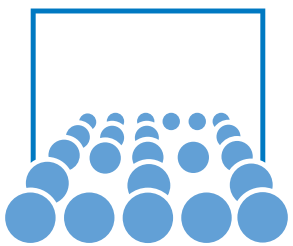
Memory-Bound Performance

CSCS-FoMICS-USI Summer School on Computer Simulations in Science and Engineering

Michael Bader

July 8–19, 2013



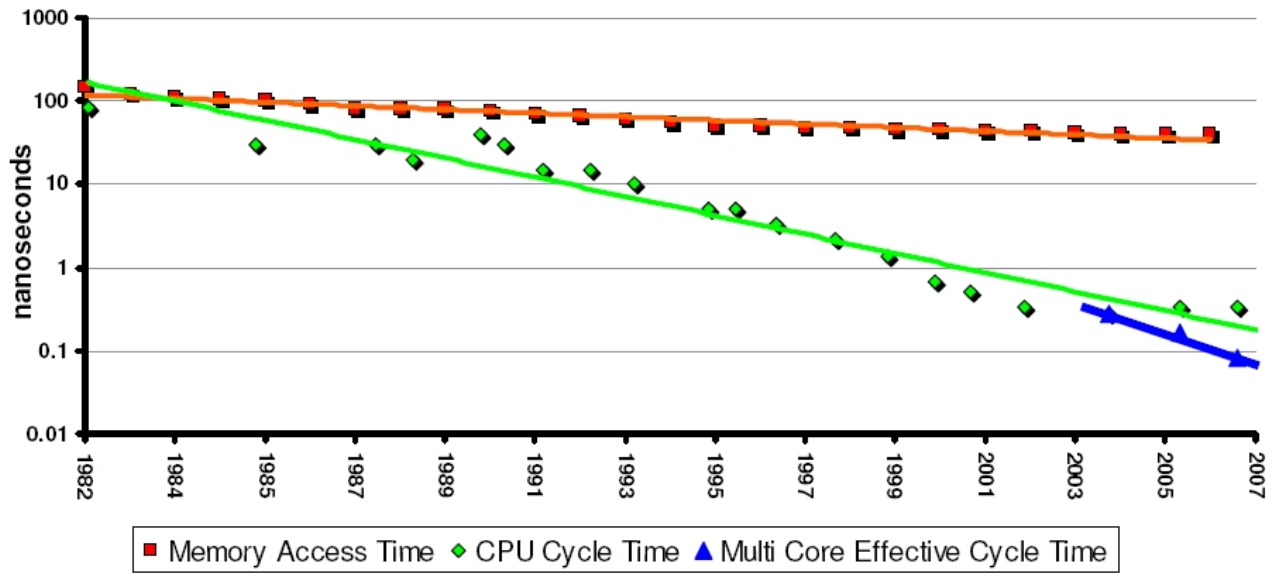
Part I

Parallel Architectures – A Memory Perspective

The Memory Gap

Graham et al., 2005:

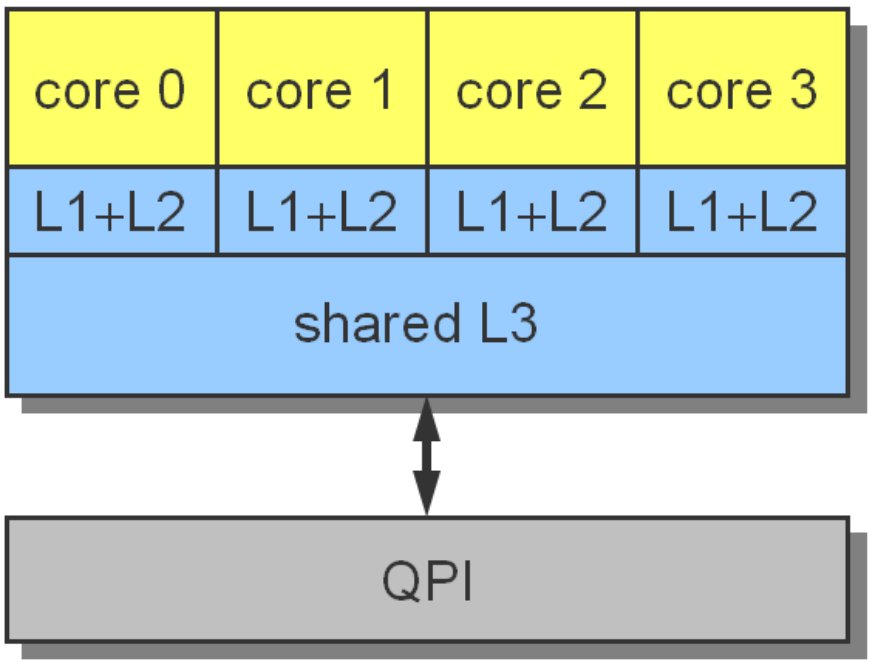
- annual increase of CPU performance (“Flop/s”): 59 %
- annual increase of memory bandwidth: ∼25 %
- annual improvement of memory latency: ∼5 %
- **memory is the bottleneck** (already, and getting worse);
- remedy: introduce cache memory!



Multicore CPUs – Intel's Nehalem Architecture

(from 2008/2009)

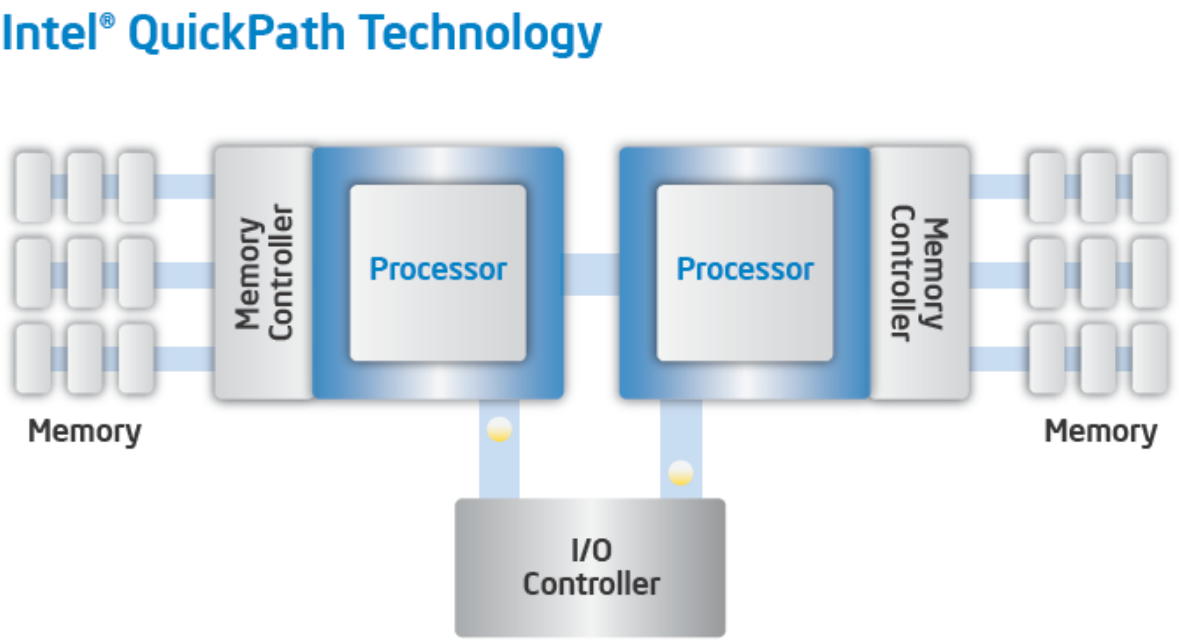
- quad-core CPU with shared and private caches
- simultaneous multithreading: 8 threads on 4 cores
- memory architecture: Quick Path Interconnect (replaced Front Side Bus)



Multicore CPUs – Intel's Nehalem Architecture (2)

(from 2008/2009)

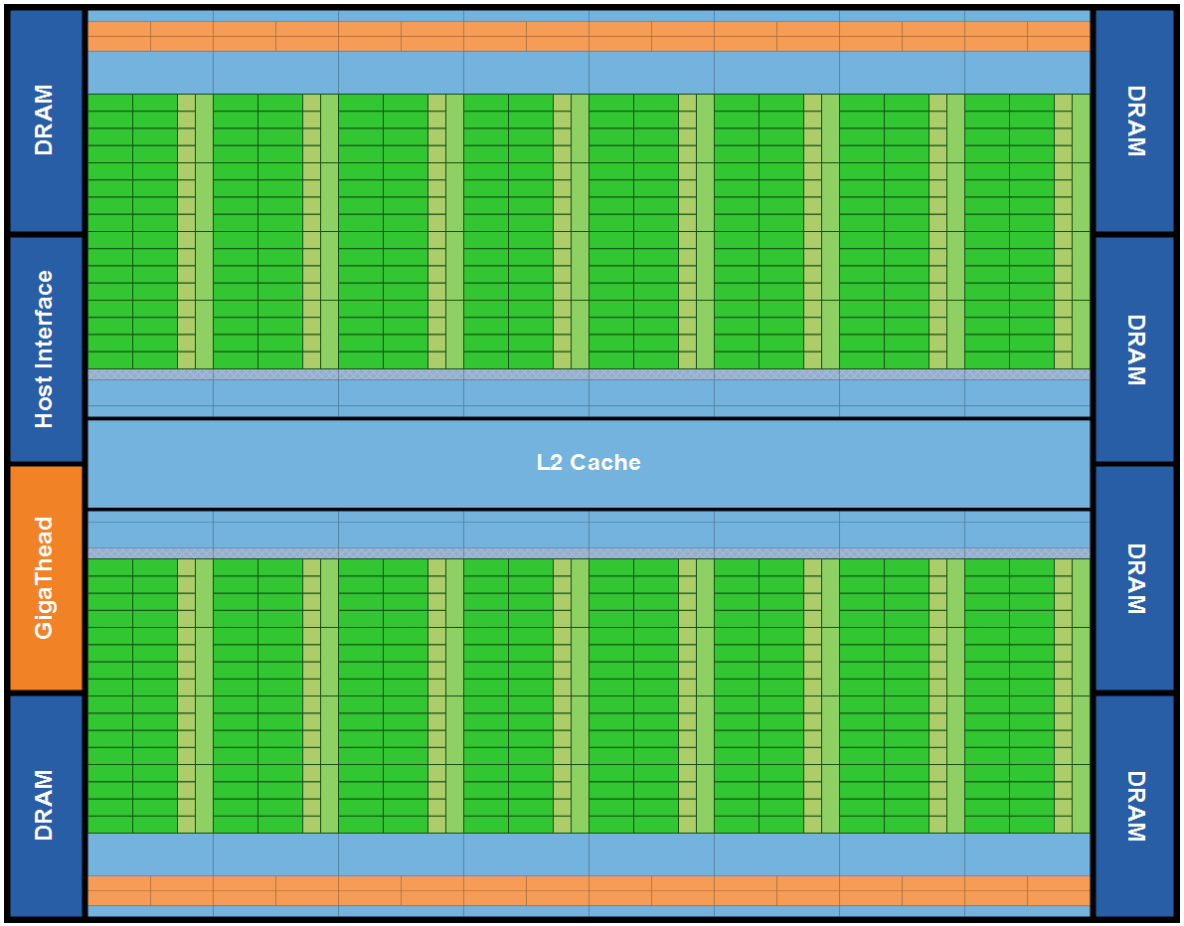
- NUMA (non-uniform memory access) architecture: CPUs have "private" main memory, but uniform access to "remote" memory
- max. 25 GB/s bandwidth



(source: Intel – Nehalem Whitepaper)

GPGPU – NVIDIA Fermi

(from 2009/2010)



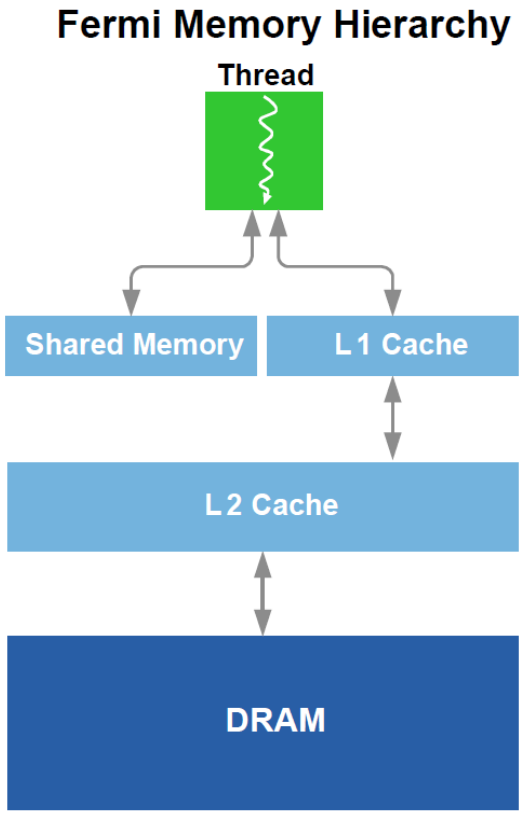
(source: NVIDIA – Fermi Whitepaper)

GPGPU – NVIDIA Fermi (3)

(from 2009/2010)

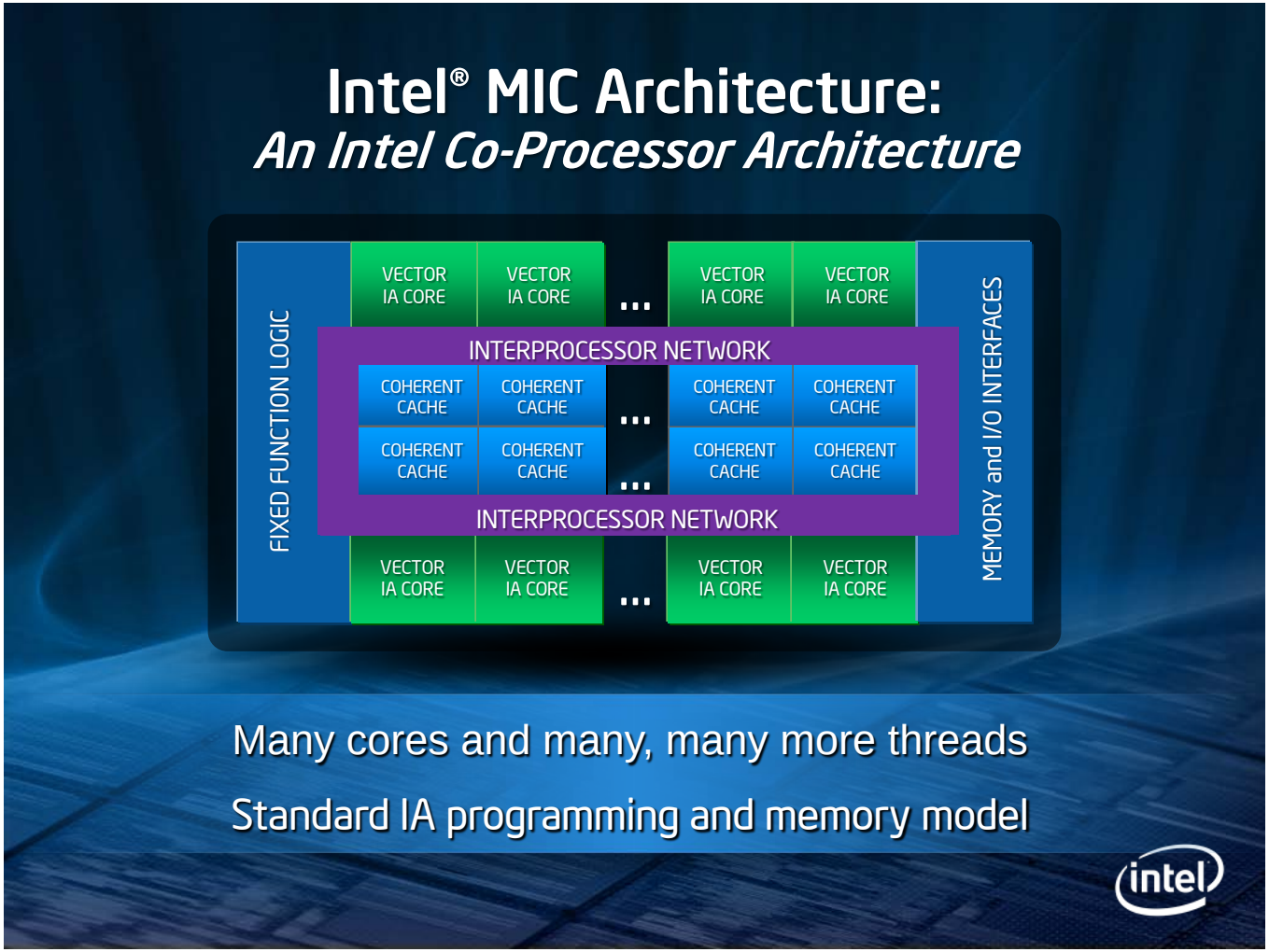
General Purpose Graphics Processing Unit:

- 512 CUDA cores
- improved double precision performance
- shared vs. global memory
- **new:** L1 und L2 cache (768 KB)
- trend from GPU towards CPU?



Manycore CPU – Intel Knights Ferry

(from 2011)



(source: Intel/K. Skaugen – SC'10 keynote presentation)

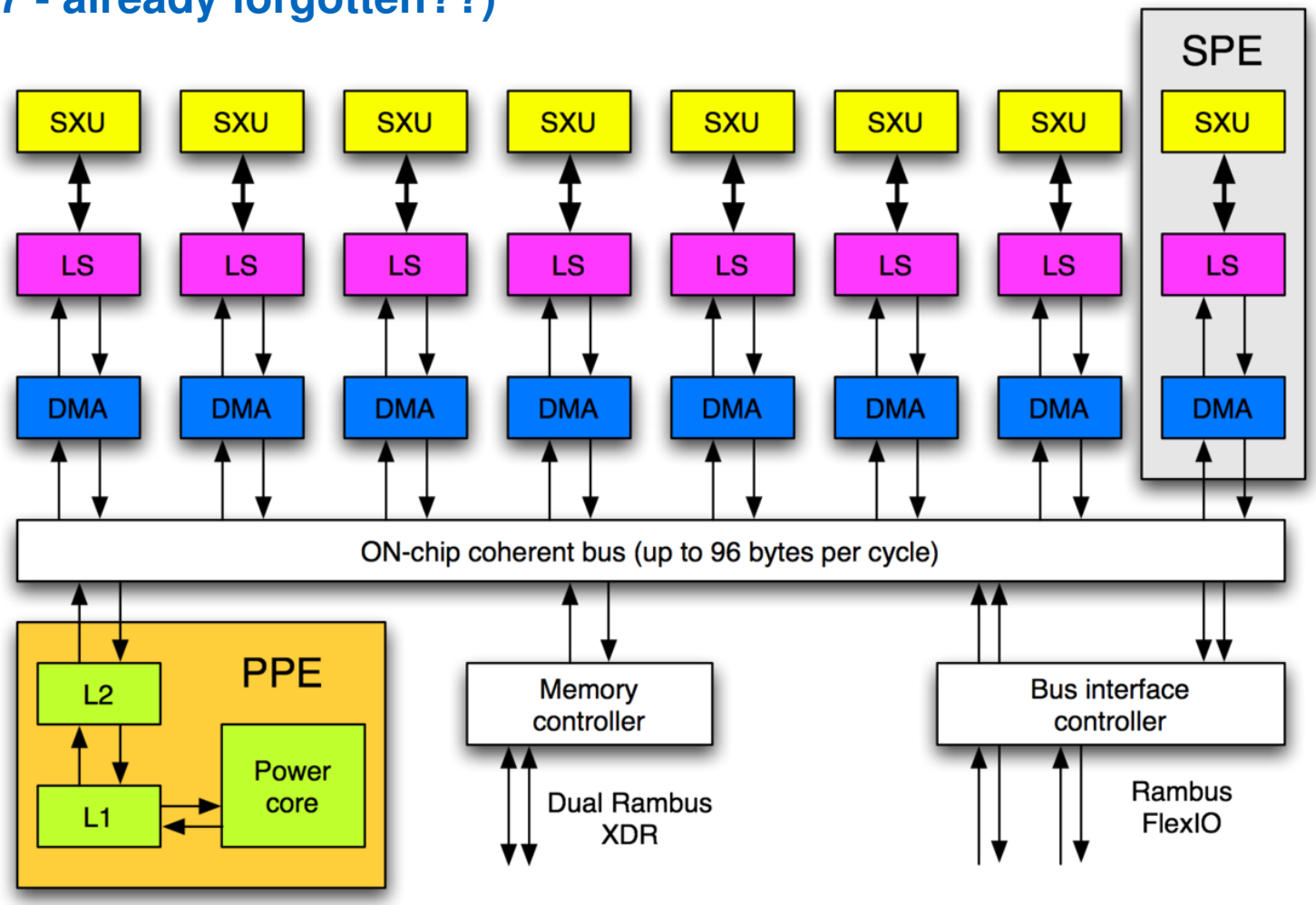
Manycore CPU – Intel Knights Ferry (2)

(from 2011)

(source: Intel/K. Skaugen – SC'10 keynote presentation)

Heterogeneous CPU – Cell BE

(from 2007 - already forgotten??)



(source: Wikipedia.de “Cell (Prozessor)”)

Cell BE and Roadrunner

Cell Broadband Engine (IBM, Sony, Toshiba)

- used for Playstation (and for supercomputing)
- PowerPC core plus 7–8 “Synergistic Processing Elements”
- small local store (256 KB) for each SPE
- explicit memory transfers required (no real caches)

Roadrunner (Los Alamos Nat. Lab.)

- # 1 of the Top 500, years 2008–2009
- first PetaFlop supercomputer: 1.026 PFlop/s (Linpack)
- costs of installation: $\sim$100 Mio$
- hybrid design: 6,562 dual-core Opteron, 12,240 Cell BE
- power consumption: 2.35 MW $\Rightarrow$ 437 MFlop/s per Watt(!)

`http://www.lanl.gov/roadrunner/rrtechnicalseminars2008.shtml`

Caches and The Memory Gap

Complicated Memory Hierarchies:

- multiple levels of caches, NUMA, main memory, ...
- sometimes explicit local store (Cell BE, GPU)
- regarding performance: **"moving memory is evil!"**

Caching Ideas and Strategies:

- **temporal locality**: if we access address x, we will probably access address x again
- replacement strategy: **least frequently/recently used**
- **spatial locality**: if we access address x, we will probably access addresses $x + \delta$ or $x - \delta$
- **cache lines** (with limited associativity)

Latency of Cache Misses

Example: Intel Sandy Bridge

Level	Latency (cycles)	Bandwidth (per core per cycle)
L1 Data L2 (Unified) Third Level (LLC)	4 12 26-31	2x16 bytes 1 x 32 bytes
L2 and L1 DCache in other cores	43–60	

Further Influences:

- prefetching due to stream detection
- eviction of cache lines bandwidth-limited

source: Intel® 64 and IA-32 Architectures Optimization Reference Manual

Types of Cache Misses – “Three C’s”

Capacity Miss:

- cache cannot hold the current memory set of the algorithm
- has to evict cache lines that are still needed $\rightsquigarrow$ cache miss

Conflict Miss:

- memory line can only be stored in certain cache lines (“cache associativity”)
- evicts cache line of data that is still required $\rightsquigarrow$ cache miss
- “false sharing”: two cores write-access the same cache line

Compulsory Miss:

- first access of a memory line: “cold miss”
- cache miss that cannot be avoided(?)
- each required piece of data is read at least once

Caches - Examples

Intel Core i7

- 32 KB L1-data-cache, 8-way associative
- 32 KB L1-instruction-cache
- 256 KB L2-Cache, 8-way associative
- 8 MB shared L3-Cache, 16-way associative
- 64 Byte cache-linesize

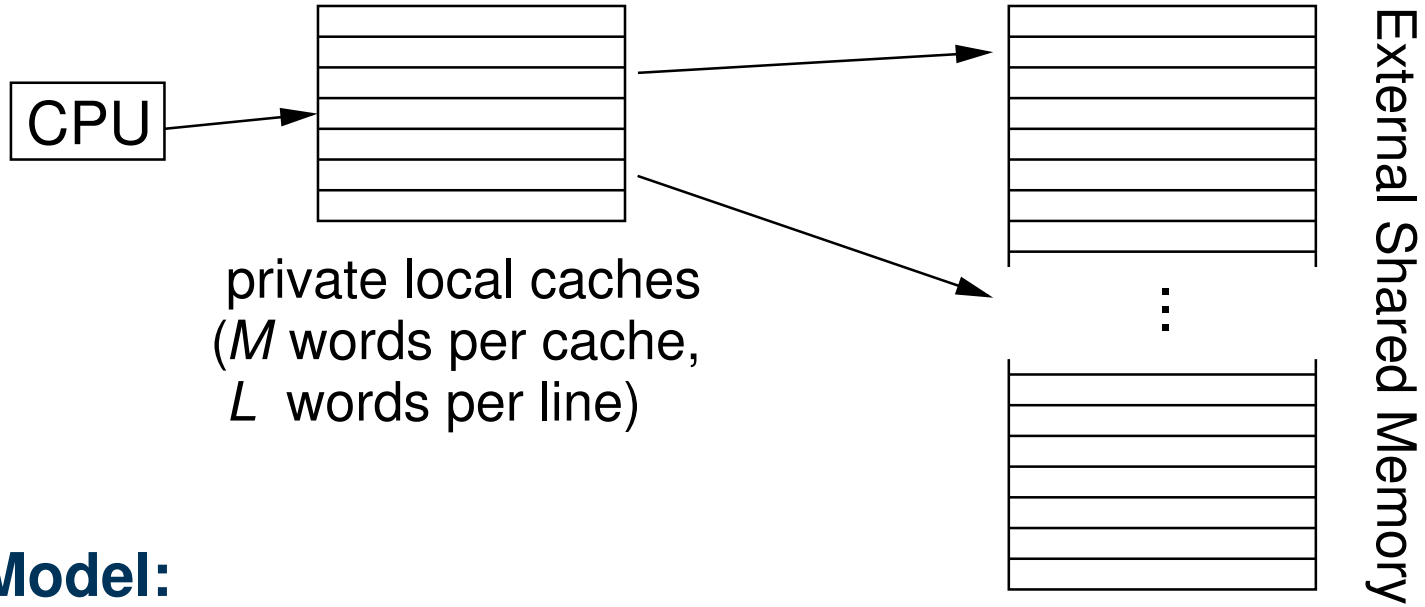
AMD Phenom II X4

- 64 KB L1-data-cache, 2-way associative
- 64 KB L1-instruction-cache
- 512 KB L2 Cache, 16-way associative
- 6 MB shared L3 Cache, 48-way associative
- 64 Byte cache-linesize

Part II

Towards Memory & Cache Efficient Algorithms
→ (Parallel) Memory Models

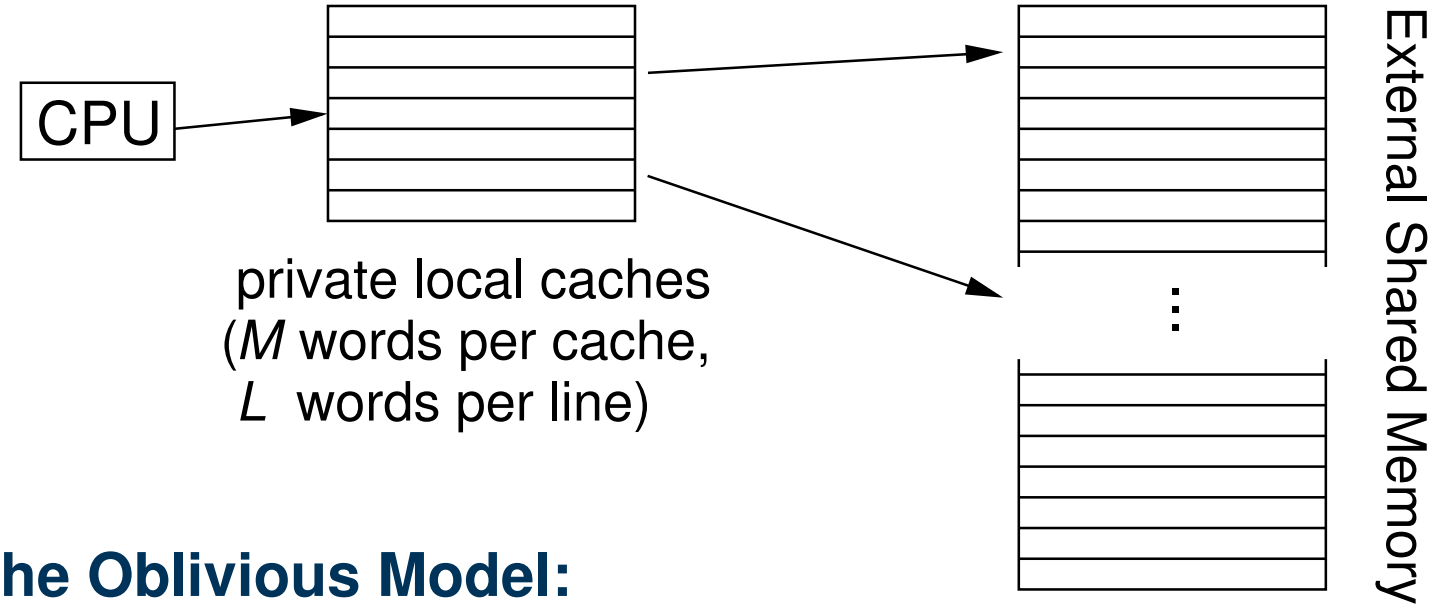
I/O Model and Cache Oblivious Model



I/O Model:

- main memory vs. hard disc; block-wise transfers
- algorithm contains explicit block transfers
- goal: algorithms with minimal number of transfers

I/O Model and Cache Oblivious Model



Cache Oblivious Model:

- cache lines vs. main memory
- assume perfect cache-replacement:
- goal: cache efficient algorithms that are “oblivious” of exact cache architecture

Example: Matrix Multiplication

Consider: Cache Oblivious Model

- number of memory transfers for the following program?

```
for i from 1 to n do
    for k from 1 to n do
        for j from 1 to n do
            C[i,k] = C[i,k] + A[i,j]*B[j,k]
```

- each j-loop reads $2n$ elements (from arrays A and B)
- thus $2n/L$ transfers per j-loop (cache line holds L words)
  only n/L transfers, if column B[:,k] stays in cache
- with n^2 iterations of the j-loop: $\mathcal{O}(n^3/L)$ transfers
- how to better exploit the local memory?

Block-Oriented Matrix Multiplication

Block-oriented formulation:

$$\begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix}$$

$\rightarrow$ each A_{ij}, B_{jk}, C_{ik} a square matrix block

Block operations:

- $C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31}$
  $C_{12} = A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32}$
  ...
- three blocks need to fit in local memory!

Block-Oriented Matrix Multiplication – Code

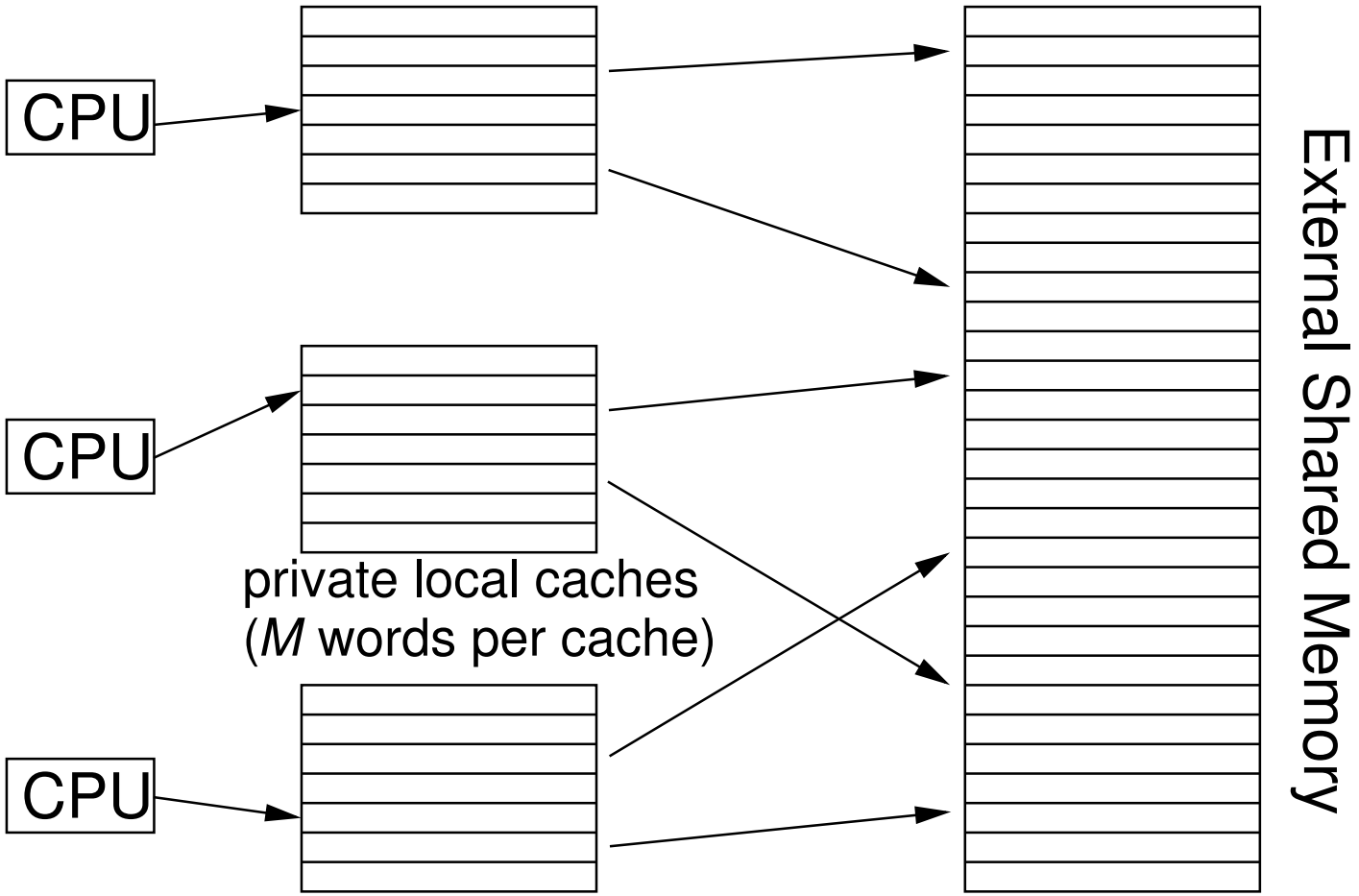
```
for i from 1 to n/b do
  for k from 1 to n/b do
    ! read/ initialise required block C
    for j from 1 to n/b do {
      ! read required blocks of A and B
      ! and perform block multiplication :
      for ii from 1 to b do
        for kk from 1 to b do
          for jj from 1 to b do
            C[i*b+ii,k*b+kk] = C[i*b+ii,k*b+kk]
                              + A[i*b+ii,j*b+jj]*B[j*b+jj,k*b+kk]
    }
```

Blocked MatrixMult – Memory Transfers

Number of Memory Transfers:

- local memory can hold M words;
  choose b such that $3b^2 < M$ (ideal: $3b^2 = M$)
- read/write local C block: $2b^2$ words per i- and k-loop
- read all n/b A blocks and n/b B blocks:
  $2n/b \cdot b^2$ words per i- and k-loop
- for all $(n/b)^2$ i-/k-loop iterations:
  $(n/b)^2 \cdot (2n/b + 2)b^2 = 2n^3/b + 2n^2$ words
- $b \in \Theta(\sqrt{M})$; move L words per transfer;
  therefore: $\Theta(n^3/(L\sqrt{M}))$ memory transfers

Parallel External Memory – Memory Scheme



[Arge, Goodrich, Nelson, Sitchinava, 2008]

Parallel External Memory – History

Extension of the classical I/O model:

- large, global memory (main memory, hard disk, etc.)
- CPU can only access smaller working memory (cache, main memory, etc.) of M words each
- both organised as cache lines of size B words
- algorithmic complexity determined by memory transfers

Extension of the PRAM:

- multiple CPUs access global shared memory (but locally distributed)
- EREW, CREW, CRCW classification (exclusive/concurrent read/write to external memory)
- similar programming model (sychronised execution, e.g.)

“Flops are For Free”

Consider a memory-bandwidth intensive algorithm:

- you can do a lot more flops than can be read from cache or memory
- **computational intensity** of a code:
  number of of flops per byte

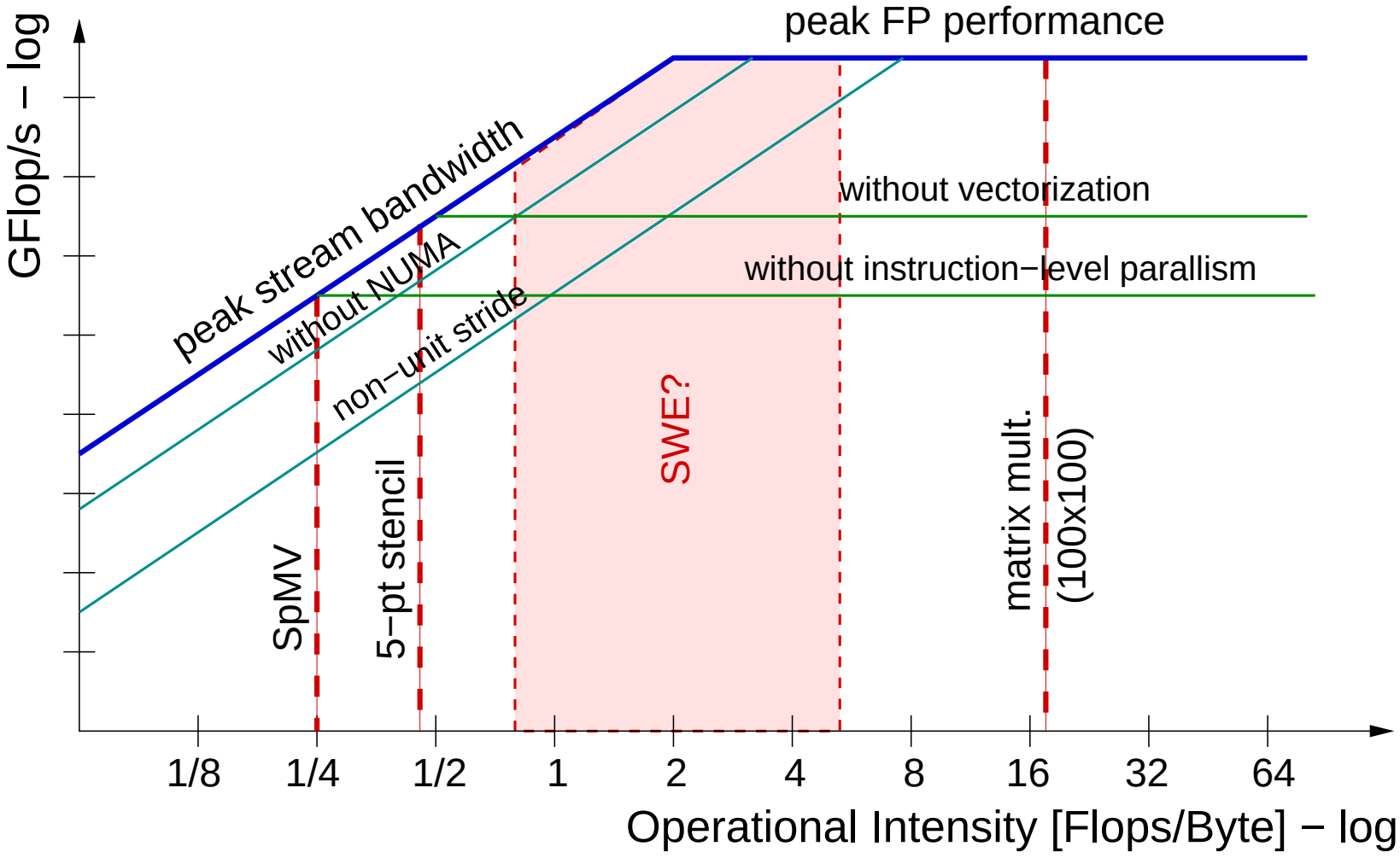
Memory-Bound Performance:

- computational intensity smaller than critical ratio
- you could execute additional flops “for free”
- speedup only possible by reducing memory accesses

Compute-Bound Performance:

- enough computational work to “hide” memory latency
- speedup only possible by reducing operations

The Roofline Model



[Williams, Waterman, Patterson, 2008]

Part III

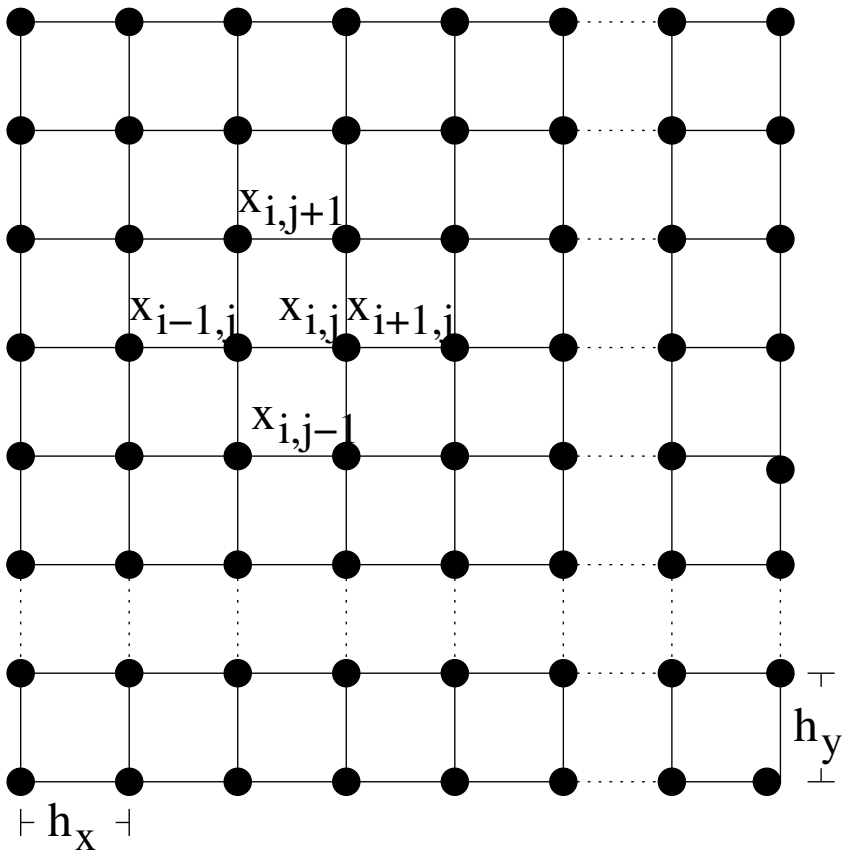
Structured Grids and Stencil Computations

Dwarf #5 – Structured Grids

1. dense linear algebra
2. sparse linear algebra
3. spectral methods
4. N-body methods
5. **structured grids**
6. unstructured grids
7. Monte Carlo

A Wiremesh Model for the Heat Equation

- consider rectangular plate as fine mesh of wires
- compute temperature x_{ij} at nodes of the mesh



Wiremesh Model Model for the Heat Equation (2)

- model assumption: temperatures in equilibrium at every mesh node
- for all temperatures x_{ij}:

$$x_{ij} = \frac{1}{4}\left(x_{i-1,j} + x_{i+1,j} + x_{i,j-1} + x_{i,j+1}\right)$$

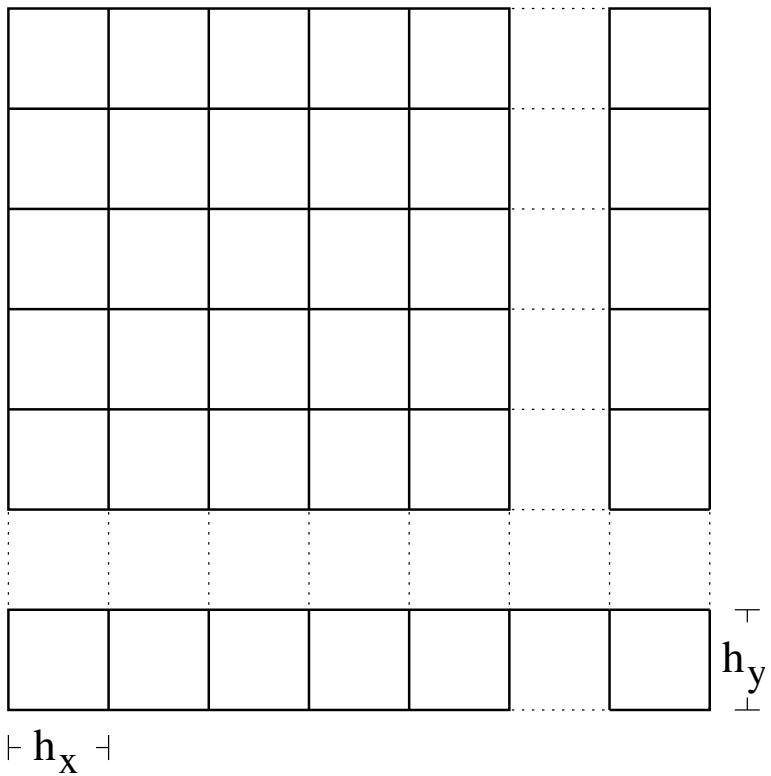
- temperature known at (part of) the boundary; for example:

$$x_{0,j} = T_j$$

- task: solve system of linear equations

A Finite Volume Model

- object: a rectangular metal plate (again)
- model as a collection of small connected rectangular cells



- examine the heat flow across the cell edges

Heat Flow Across the Cell Boundaries

- Heat flow across a given edge is proportional to
  - temperature difference $(T_1 - T_0)$ between the adjacent cells
  - length h of the edge
- heat flow across all edges $\rightsquigarrow$ change of heat energy:

$$\begin{aligned} q_{ij} &= k_x \left(T_{ij} - T_{i-1,j}\right) h_y + k_x \left(T_{ij} - T_{i+1,j}\right) h_y \\ &+ k_y \left(T_{ij} - T_{i,j-1}\right) h_x + k_y \left(T_{ij} - T_{i,j+1}\right) h_x \end{aligned}$$

- in equilibrium, with source term: $q_{ij} + F_{ij} = 0$

$$\begin{aligned} f_{ij} h_x h_y &= -k_x h_y \left(2T_{ij} - T_{i-1,j} - T_{i+1,j}\right) \\ &\quad -k_y h_x \left(2T_{ij} - T_{i,j-1} - T_{i,j+1}\right) \end{aligned}$$

- apply iterative solver

Towards a Time Dependent Model

- idea: set up ODE for each cell
- simplification: no external heat sources or drains, i.e. $f_{ij} = 0$
- change of temperature per time is proportional to heat flow into the cell (no longer 0):

$$\begin{aligned}\dot{T}_{ij}(t) &= \frac{\kappa_x}{h_x}\left(2T_{ij}(t) - T_{i-1,j}(t) - T_{i+1,j}(t)\right) \\ &+ \frac{\kappa_y}{h_y}\left(2T_{ij}(t) - T_{i,j-1}(t) - T_{i,j+1}(t)\right)\end{aligned}$$

- solve system of ODE
  $\rightarrow$ using Euler time stepping, e.g.:

$$\begin{aligned}T_{ij}^{(n+1)} = T_{ij}^{(n)} &+ \tau\frac{\kappa_x}{h_x}\left(2T_{ij}^{(n)}(t) - T_{i-1,j}^{(n)}(t) - T_{i+1,j}^{(n)}(t)\right) \\ &+ \tau\frac{\kappa_y}{h_y}\left(2T_{ij}^{(n)}(t) - T_{i,j-1}^{(n)}(t) - T_{i,j+1}^{(n)}(t)\right)\end{aligned}$$

General Pattern: Stencil Computation

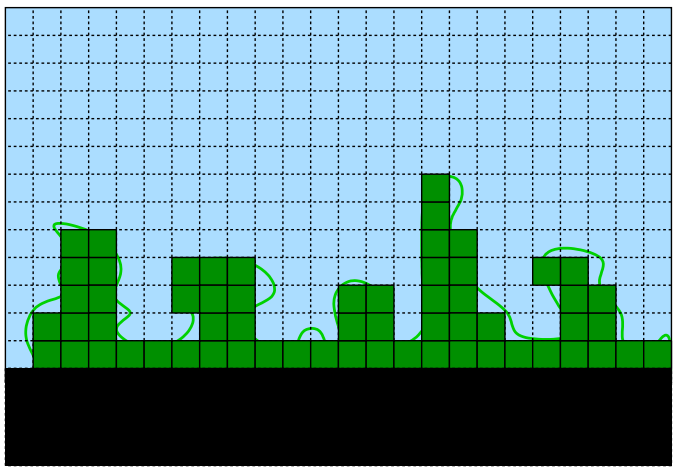
- update of unknowns, elements, etc., according to a fixed pattern
- pattern usually defined by neighbours in a structured grid/lattice
- task: “update all unknowns/elements” $\rightarrow$ *traversal*
- multiple traversals for iterative solvers (for systems of equations) or time stepping (for time-dependent problems)
- example: Poisson equation (h^2 factors ignored):

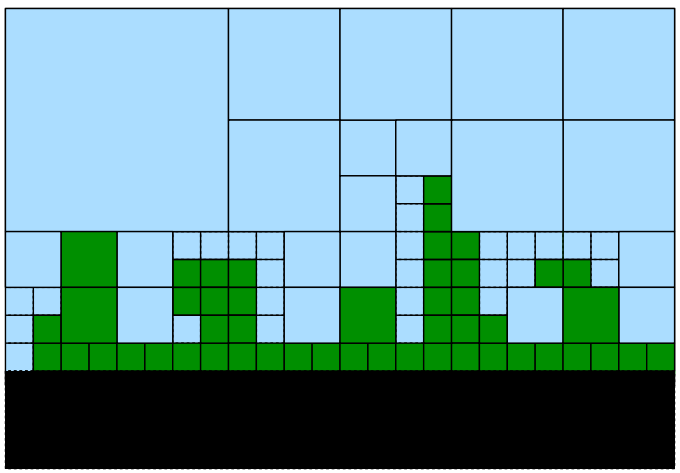
$$\text{1D:} \quad \begin{bmatrix} 1 & -2 & 1 \end{bmatrix} \qquad \text{2D:} \quad \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix}$$

- our example: shallow water equation on Cartesian grid (Finite Volume Model)

Structured Grids – Characterisation

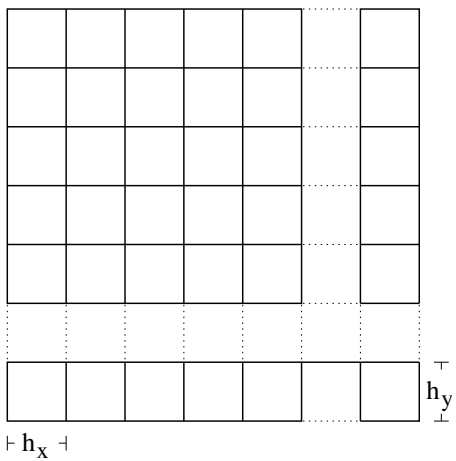
- construction of points or elements follows regular process
- *geometric* (coordinates) and *topological* information (neighbour relations) can be derived (i.e. are not stored)
- memory addresses can be easily computed

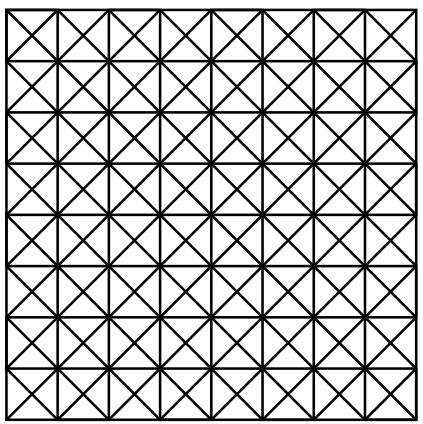


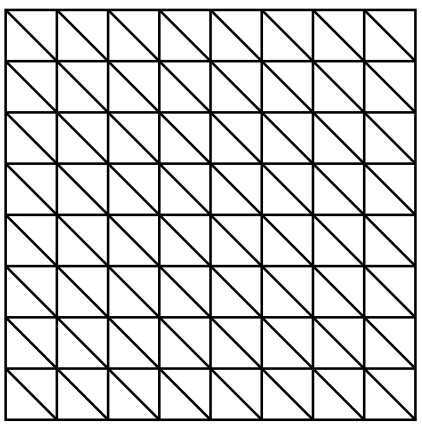


Regular Structured Grids

- rectangular/*cartesian* grids:
  rectangles (2D) or cuboids (3D)
- triangular meshes:
  triangles (2D) or tetrahedra (3D)
- often: row-major or column-major traversal and storage

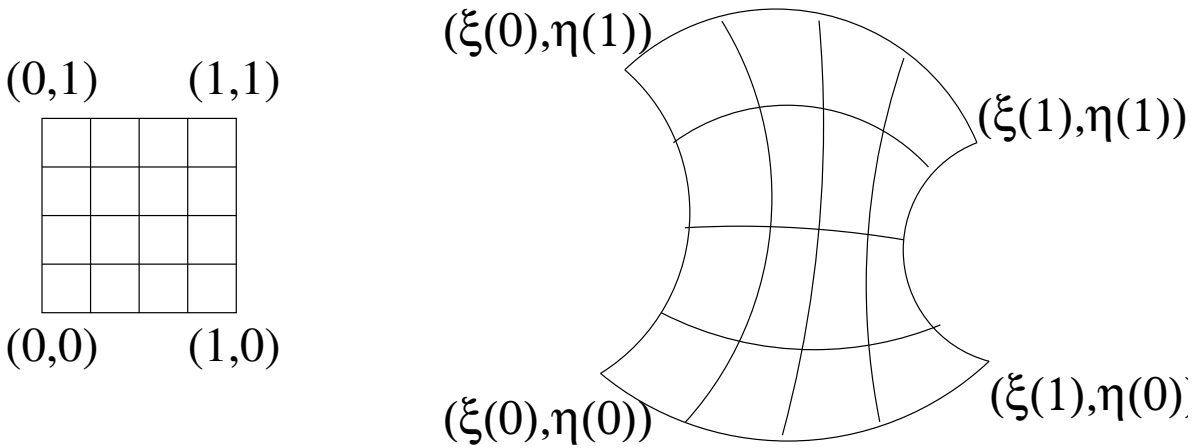






Transformed Structured Grids

- transformation of the unit square to the computational domain
- regular grid is transformed likewise

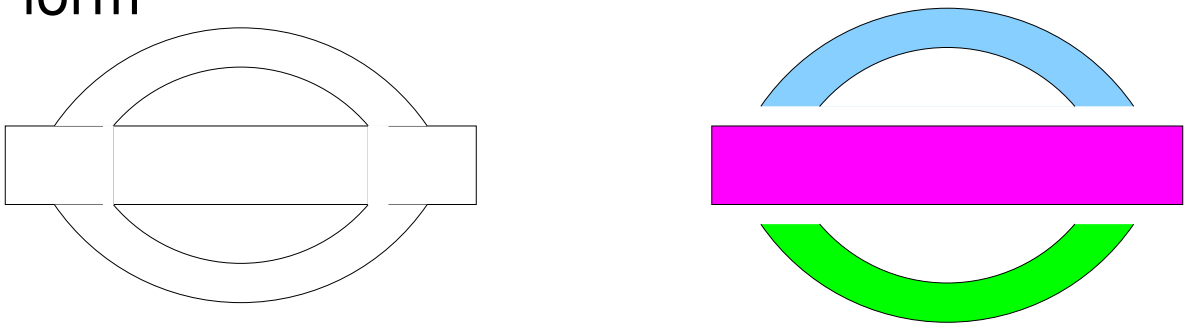


Variants:

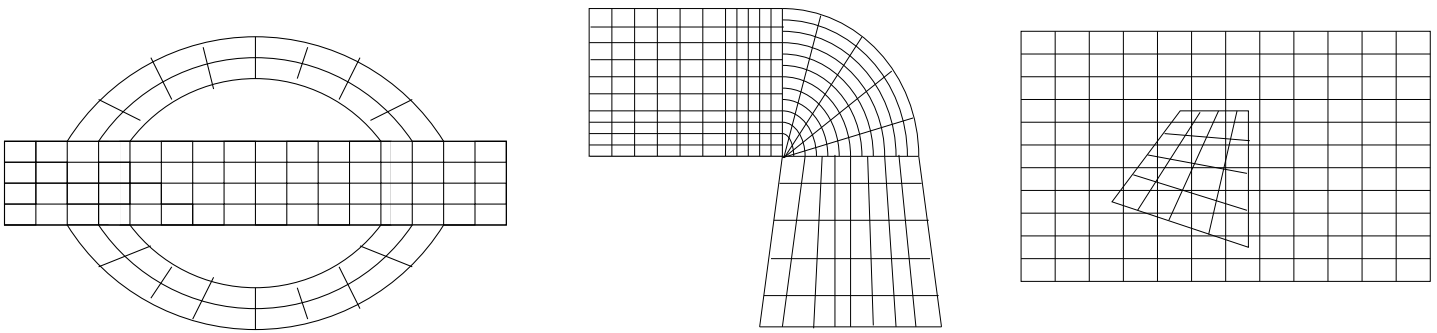
- *algebraic*: interpolation-based
- *PDE-based*: solve system of PDEs to obtain $\xi(x, y)$ and $\eta(x, y)$

Composite and Block Structured Grids

- subdivide (complicated) domain into subdomains of simpler form



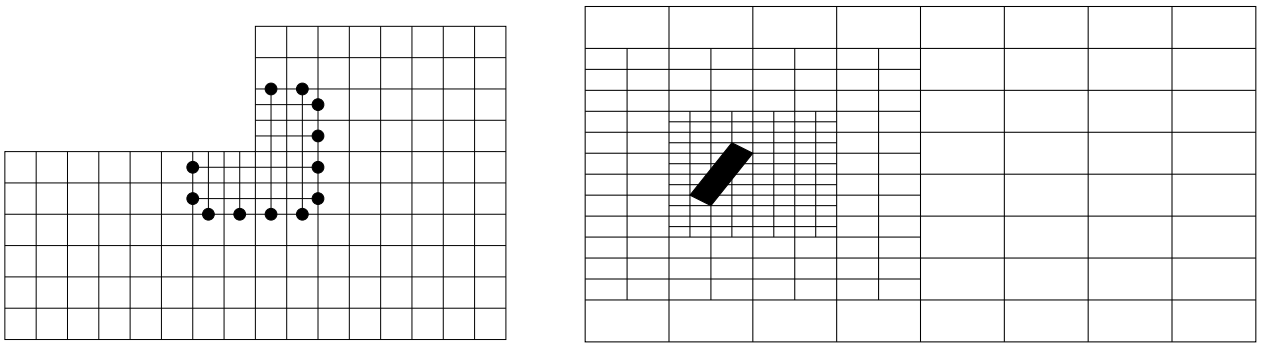
- and use regular meshs on each subdomain



- at interfaces:
  - conforming at interface (“glue” required?)
  - overlapping grids (*chimera* grids)

Block Adaptive Grids

- retain regular structure
- refinement of entire blocks
- similar to block structured grids



- efficient storage and processing
- but limited w.r.t. adaptivity

Part IV

(Cache-)Efficient (Parallel) Algorithms for Structured Grids

Analysis of Cache-Usage for 2D/3D Stencil Computation

We will assume:

- 2D or 3D Cartesian mesh with $N = n^d$ grid points
- stencil only accesses nearest neighbours
  $\rightarrow$ typically $c_M := 2d$ or $c_M := 3^d$ accesses per stencil
- c_F floating-point operations per stencil, $c_F \in \mathcal{O}(c_M)$

We will examine:

- number of memory transfers in the Parallel External Memory model (equiv. to cache misses)
- for different implementations and algorithms
- similar for ratio of communication to computation

Straight-Forward, Loop-Based Implementation

Example:

```
for i from 1 to n do
  for j from 1 to n do {
    x[i,j] = 0.25*(x[i-1,j]+x[i+1,j]+x[i,j-1]+x[i,j+1])
  }
```

Number of cache line transfers:

- x[i−1,j], x[i,j], and x[i+1,j] stored consecutive in memory $\rightsquigarrow$ loaded as one cache line (of size L)
- question: Cache size M large enough to hold n floats?
- if $n > M$: cache misses for x[i,j−1] and x[i,j+1]
- this: $3N/L = 3n^2/L$ transfers; no impact of cache size M

Loop-Based Implementation with Blocking

Example:

```
for ii from 1 to n by b do
  for jj from 1 to n by b do
    for i from ii to ii+b−1 do
      for j from jj to jj+b−1 do {
        x[i,j] = 0.25∗(x[i−1,j]+x[i+1,j]+x[i,j−1]+x[i,j+1])
      }
```

Number of cache line transfers:

- choose b such that the cache can hold 3 rows of x: $M > 3b$
- then: N/L transfers; still independent of cache size M (besides condition for b)

Extension to 3D stencils

Simple loops:

- if cache holds 3 planes of x, $M > 3n^2$, then N/L transfers
- if cache only holds 1 plane, $M > n^2$, then $3N/L$ transfers
- if cache holds lees than 1 row, $M < n$, then $5N/L$ transfers (if $c_M = 6$) or $9N/L$ transfers (if $c_M = 3^3 = 27$)

With blocking:

- cache needs to hold 3 planes of a b^3 block: $M > 3b^2$
- then: N/L transfers; again independent of cache size M (besides condition for b)

Further Increase of Cache Reuse

Requires multiple stencil evaluations:

```
for t from 1 to m do
  for i from 1 to n do
    for j from 1 to n do {
      x[i,j] = 0.25*(x[i-1,j]+x[i+1,j]+x[i,j-1]+x[i,j+1])
    }
```

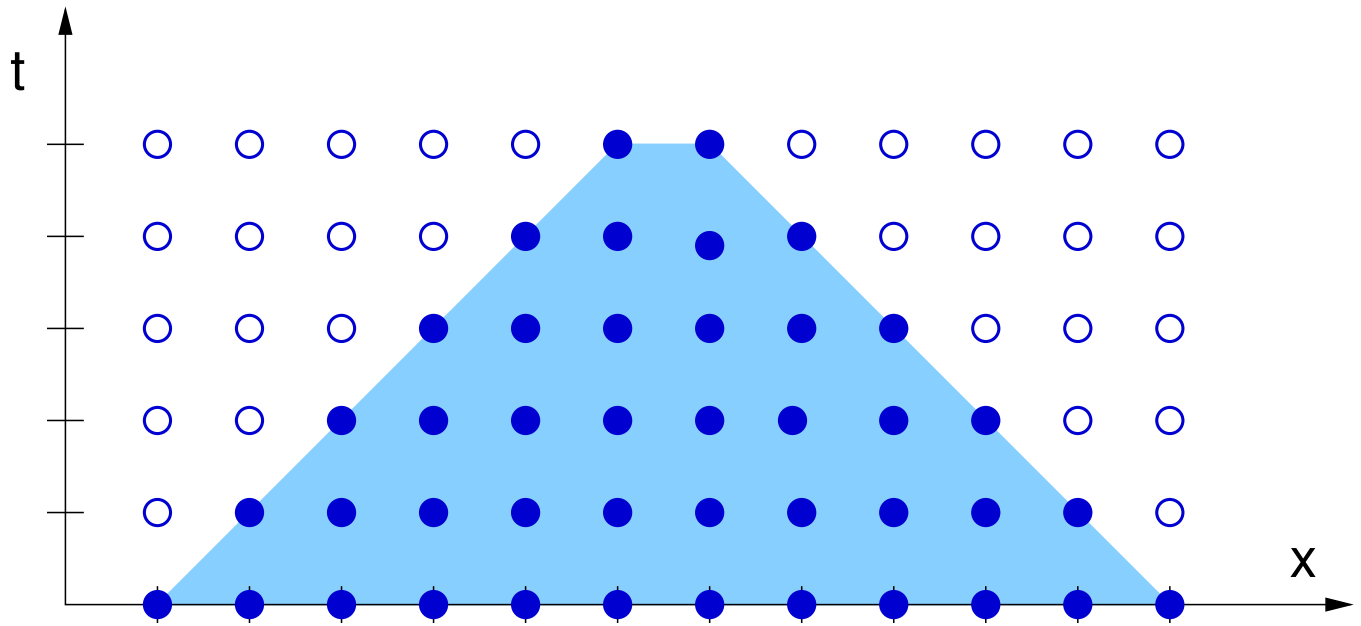
$\rightarrow$ for multiple iterations or time steps, e.g.

Possible approaches:

- blocking in space and time?
- what about precedence conditions of stencil updates?

Region of Influence for Stencil Updates

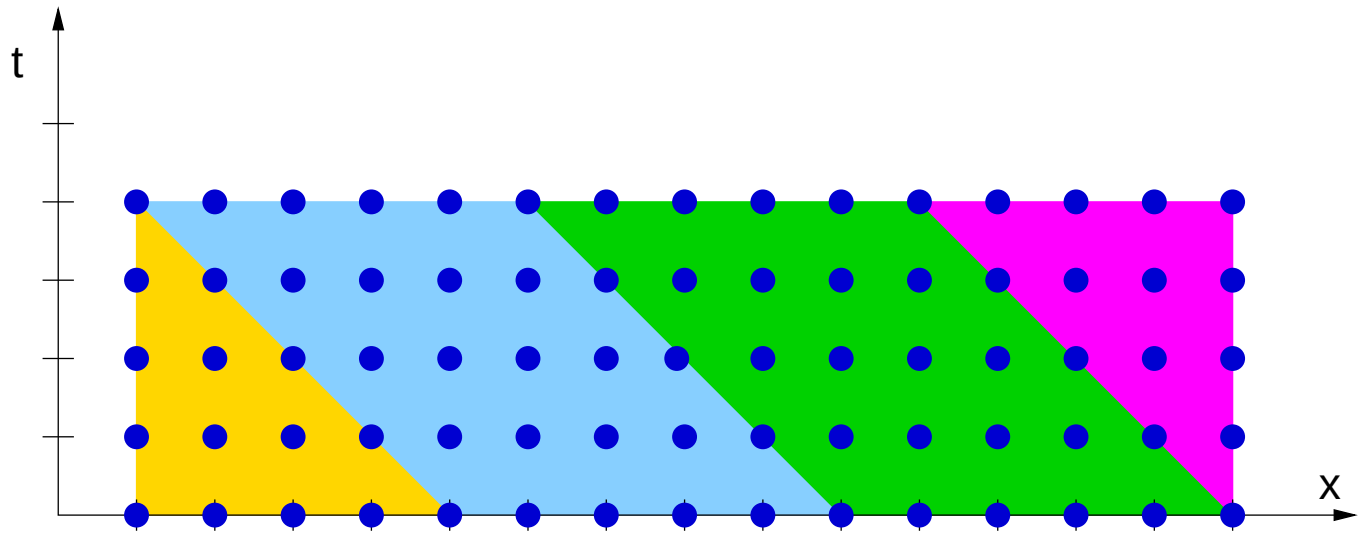
1D Example:



- area of “valid” points narrows by stencil size in each step
- leads to trapezoidal update regions
- similar, but more complicated, in 2D and 3D

Option: Time Skewing

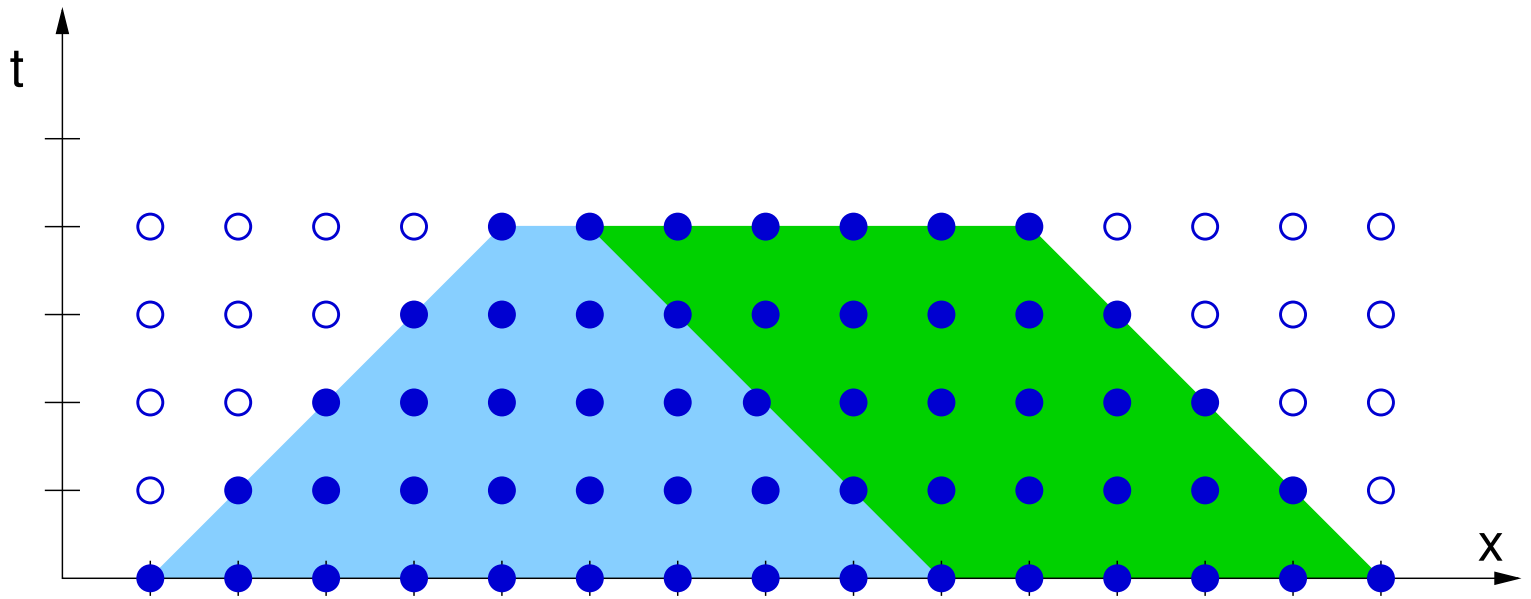
1D Example:



- sliding trapezoidal update “window”
- question: optimal size of window in x- and t-direction?
- can be extended to 2D and 3D

Divide & Conquer Algorithm: Space Split

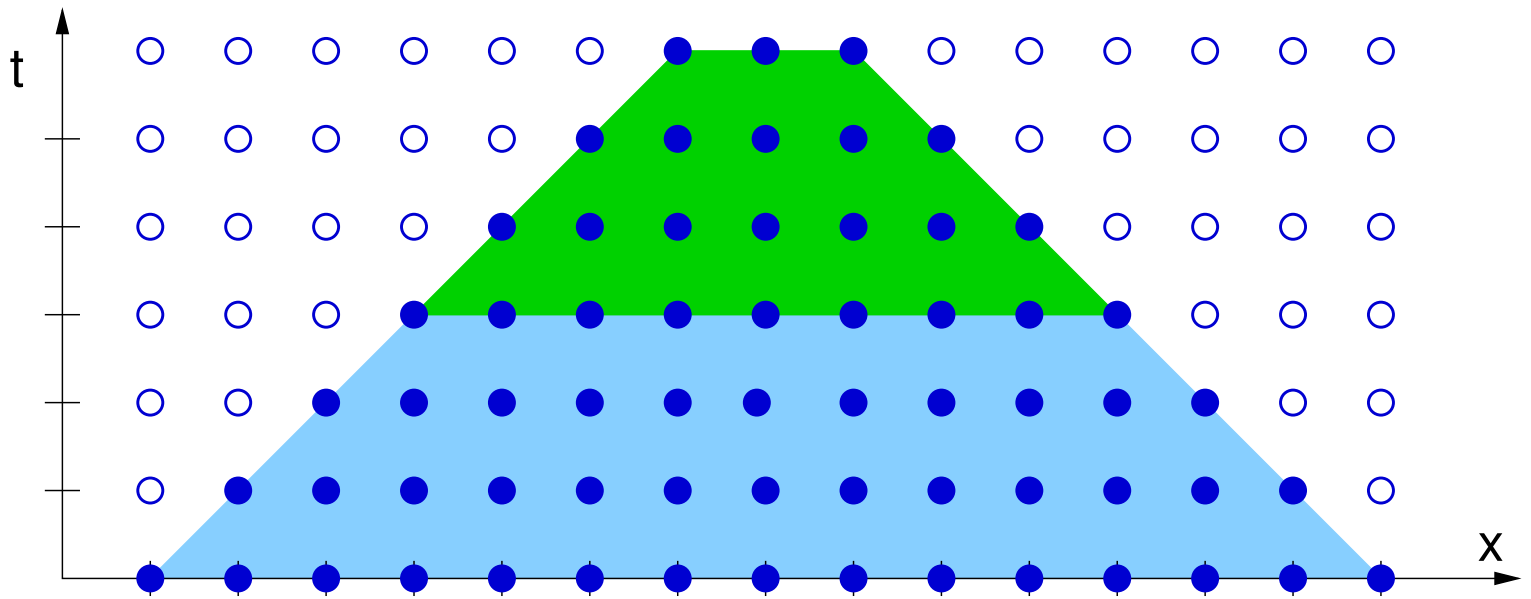
1D Example:



- applied, if spatial domain is at least “twice as large” as number of time steps
- note precedence condition for left vs. right subdomain

Divide & Conquer Algorithm: Time Split

1D Example:



- applied, if spatial domain is less than “twice as large” as number of time steps
- space split likely as the next split for the lower domain

Cache Oblivious Algorithms for Structured Grids

Algorithm by Frigo & Strumpen:

- divide & conquer approach using time and space splits
- $\mathcal{O}(N/\sqrt[d]{M})$ cache misses in "cache oblivious" model ("Parallel External Memory" with only 1 CPU and "ideal cache")

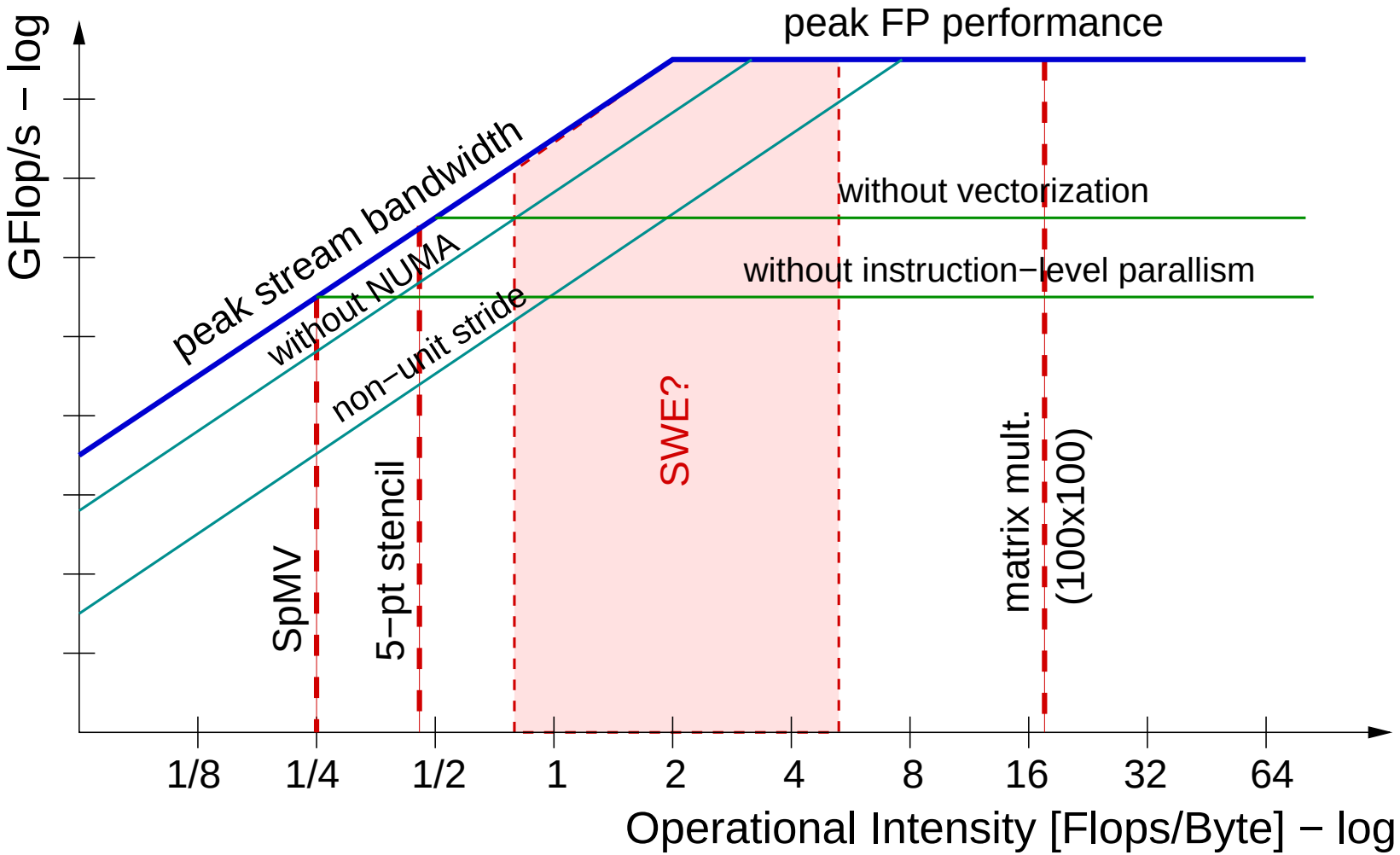
References/Literature:

- Matteo Frigo and Volker Strumpen: *Cache Oblivious Stencil Computations*, Int. Conf. on Supercomput., ACM, 2005.
- Matteo Frigo and Volker Strumpen: *The Memory Behavior of Cache Oblivious Stencil Computations*, J. Supercomput. 39 (2), 2007

Part V

Workshop – Memory Access in SWE

Towards a Roofline Model for SWE



Towards a Roofline Model for SWE

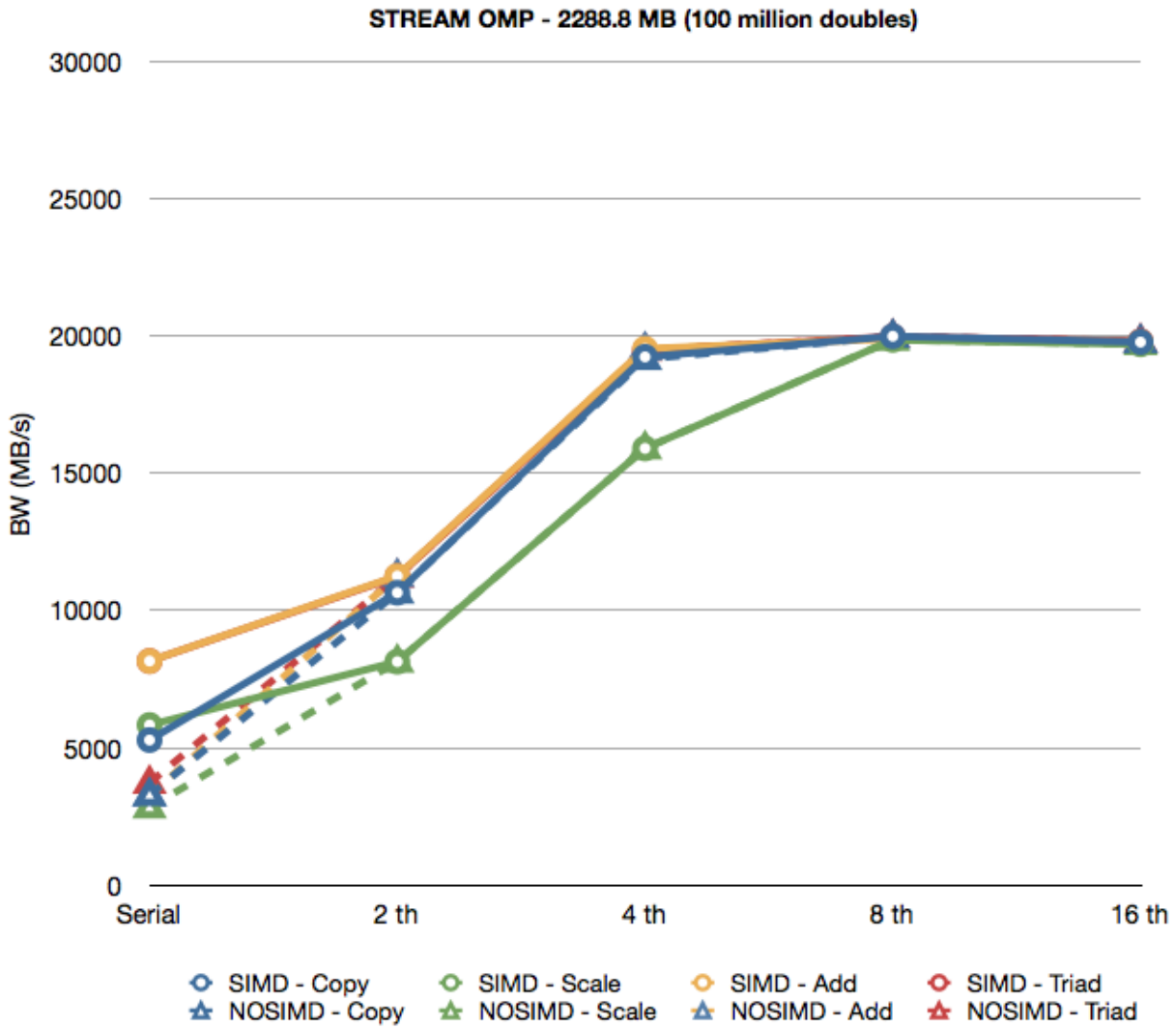
Components of the Roofline Model:

- peak floating point performance and peak bandwidth? (on a single core/socket/node)
- operational complexity of SWE?
- test performance for non-unit-stride memory access (exchange i-/j-loops, e.g.)
- test performance with and without vectorization
- test performance implications on serial *and* parallel code!

Possible improvements of SWE:

- examine memory access pattern of SWE $\rightarrow$ possibilities for improvement? (loop fusion? storage of net updates?)
- improve cache behaviour of SWE (loop skewing, e.g.??)

Roofline Ingredient #1: Stream Benchmark



An Improvement to Memory Usage of SWE

Consider time-stepping scheme:

$$Q_{i,j}^{n+1} = Q_{i,j}^{n} - \frac{\Delta t}{\Delta x}\left(\mathcal{A}^{+}\Delta Q_{i-1/2,j}^{n} + \mathcal{A}^{-}\Delta Q_{i+1/2,j}^{n}\right) - \frac{\Delta t}{\Delta y}\left(\mathcal{B}^{+}\Delta Q_{i,j-1/2}^{n} + \mathcal{B}^{-}\Delta Q_{i,j+1/2}^{n}\right).$$

Implementation w.r.t. memory:

1. compute and store all net updates $\mathcal{A}^{+}\Delta Q_{i-1/2,j}^{n}$, etc.
2. update quantities $Q_{i,j}^{n+1}$

Possible improvement:

1. accumulate $\frac{1}{\Delta x}\mathcal{A}^{+}\Delta Q_{i-1/2,j}^{n}$, etc., for each quantity
2. add these accumulated net updates to $Q_{i,j}^{n}$

References and Literatur

- L. Arge, M. T. Goodrich, M. Nelson, N. Sitchinava: *Fundamental parallel algorithms for private-cache chip multiprocessors*. Proc. 20th ACM Symp. Parallelism in Algorithms and Architectures (SPAA), 2008.
- K. Datta et al.: *Optimization and Performance Modeling of Stencil Computations on Modern Microprocessors*, SIAM Review 51 (1), 2009
- Graham, Snir, Patterson: *Getting Up to Speed: The Future of Supercomputing.* The National Academies Press, 2005
- S. W. Williams, A. Waterman, D. A. Patterson: *Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures*, UC Berkeley, Tech. Report No. UCB/EECS-2008-134