Hardware-aware Computing on Multicore Processors and Accelerators – Part II

Jan-Philipp Weiß
Outline I

- **Goal and Motivation**
  - Driving factor: applications
    - Grand challenges in numerical simulation
  - The multicore challenge
    - Status of multicore
    - How to face the dilemma

- **Application aspects**
  - Mathematics vs. hardware reality
  - Model problem of computational fluid dynamics
  - Algorithmic characteristics
    - Computational intensity
  - Simple performance model
Outline II

- The Multicore Era
  - Characteristics of current multicore hardware
  - Properties of accelerators
  - How to program?
  - How life has changed

- Hardware-aware computing
  - How to get the best performance?
    - Memory- and compute-bound algorithms
    - Optimization of computations
    - Optimization of data transfers
    - Optimization of data layout
  - Impact on algorithms and mathematical software
Outline III

- Case study: Implementation of stencil kernels
  - Multicore-CPUs
  - STI Cell
  - GPU stream computing (NVIDIA CUDA)

- Performance results
Goal and Motivation

- Multicore and coprocessor technologies offer a great potential of compute power (at least in theory) constantly growing performance rates

- General goal
  - Exploit this parallel potential for speeding up or enabling (large-scale) numerical simulation, image processing, real-time applications, ...

- Experience
  - It is a hard job
  - No free lunch – and we are getting hungrier and hungrier

- Will the bet on parallelism be successful?
  - Do normal users need unlimited performance?
    - Netbooks, social networks, cloud computing, ...
Driving factor: Applications

- Numerical simulation generates complex problems
  - Modeling
    - Problem description and mathematical / physical modeling
  - Numerical solution
    - Highly efficient and problem-adapted solvers
    - Algorithm design
  - Parallel implementation
    - Flexible and adaptive data layout
    - Optimal resource utilization (functional units, memory, bandwidth)
    - Scalability
  - Visualization and Validation

- Comprehensive hardware knowledge required for optimal throughput

Urgent demand for compute power, bandwidth, parallel concepts, tools, …

- Keywords
  - Performance, productivity, portability, and flexibility
Grand Challenges

- Climate prediction and weather forecast
- Medical engineering
  - United Airways project
- Computational fluid dynamics
- Life sciences and bioinformatics
- Automotive and aviation
- Astro- and plasma physics
- Material sciences

- Insatiable demand for compute power!
  - How to utilize compute power of multicore processors and accelerators most efficiently?
  - How to exploit special hardware capabilities (speed, reconfigurability, ...)?
Status of HW and SW

- Huge potential of parallel compute power
  - Theoretical performance proportional to #cores
  - Parallelism across several system levels

- Memory gap
  - Data cannot be delivered to the fast cores in time
  - Mitigation of gap by a hierarchy of nested levels in the memory subsystem

- Diverging approaches in technology
  - Different processing models
  - Different programming environments
  - Missing standards

- Incomplete tool chains
  - Hard development process
  - Bottlenecks and errors hard to find

- Major obstacles are impeding general success (and fun)
  - Well-known problem in HPC (since 19..)
How to face the dilemma?

Applied Mathematics

Problem characteristics

Math. Models
PDE, CFD, ...

Numerical Methods
FEM, LSE, ...

Applications

Parallel programming models + languages

Efficient implementation

Abstraction

Computer Science

Hardware characteristics

Multicore CPUs

Cell

GPUs

Hardware

Coprocessors

FPGAs

Future concepts

High Performance Computing

Electrical Engineering

Forschungszentrum Karlsruhe
in der Helmholtz-Gemeinschaft
Universität Karlsruhe (TH)
Forschungsuniversität • gegründet 1825
How to face the dilemma?

- Interaction between participating disciplines
  - Computer science, applied mathematics, high performance computing, hardware developers, engineering disciplines (physical problems)

- Interaction between hardware and application characteristics
  - Algorithms need to be adapted to
    - all available levels of parallelism
    - a nested, hierarchical and distributed memory sub-system

- Re-implementations strongly required (?!)
  - Question of time, money, patience and experience
  - Which is the right platform?
    - Still an open question...
The Multicore-Secret

- Power consumption: $P \sim f V^2 \sim V^3$
- Performance: $A \sim f$

- $f$ ... clock rate
- $V$ ... voltage

The basic idea
- Double number of cores: 1 core $\rightarrow$ 2 cores
- Reduce voltage and clock rate: $V \rightarrow 0.75V$, $f \rightarrow 0.75f$
- Results in less power consumption: $P \rightarrow 0.84 P$ (2x $0.75^3$)
- Results in more performance: $A \rightarrow 1.5 A$ (2x 0.75)
- Results in energy efficiency (Perf./Watt): $A/P \rightarrow 1.8 A/P$ (1.5/0.84)

- 50% more performance and 15% less power!
Amdahl’s and Gustafson’s law

\[ T_S \]
\[ T_1 = T_S + (T_1 - T_S) \]
\[ T_P = T_S + \frac{T_1 - T_S}{P} \]

... execution time of the sequential part
... total execution time on a single processor
... total execution time on \( P \) processors

\[
\text{Speedup } S = \frac{T_1}{T_P} = P \frac{T_1/T_S}{P + T_1/T_S - 1}
\]

Amdahl’s law: \( T_1/T_S = 1/f = \text{const} \)
\[
\lim_{P \to \infty} S = 1/f
\]

Gustafson’s law: \( T_1/T_S \to \infty \) for \( N \to \infty \) (problem size \( N \))
\[
\lim_{N \to \infty} S = P
\]
Moore’s cores

Exponentially increasing core counts expected

- General purpose CPUs:
  - 2, 4, 6, 8, ... cores
- Intel Larrabee, Polaris
  - 32 cores, 80 cores
- ClearSpeed
  - 192 cores
- NVIDIA GPUs
  - 240 cores
- AMD/ATI GPUs
  - 800 cores
- Tilera, Rapport
  - 64 cores, 256 cores
Application Aspects
Mathematics vs. Hardware

Reality

What mathematical cognition is telling

- Use locally adapted grids to resolve solution characteristics
  - Singularities, accumulated errors, huge domains, curvilinear boundaries...

- Optimize number of degrees of freedom by redistribution
  - Adaptive error control and goal oriented meshing

- Benefits in results outweigh grid management overhead (70%?)
  - Indirect addressing, pointer chasing, ...

- Implicit solution methods are more favorable than explicit methods
  - Stability and time step limitations

Good strategy for cache-based multi-core CPUs
with automatic data transfers
Mathematics vs. Hardware reality
What hardware reality is answering

- Explicit memory transfers require predictable and structured memory access patterns
- Bandwidth bottleneck can only be overcome by contiguous, coalesced and aligned memory access
  - No random or indirect access to memory
- Explicit methods allow temporal blocking techniques
  - Data reuse across several time steps

Algorithms need to be uniformly structured and simple for best throughput on multicore processors and accelerators
Major questions

- How to get the best performance from multicore-based and heterogeneous systems?

- How to adapt algorithms and implementations to the hardware and application characteristics?

- How to develop portable concepts and methodologies?
  - How to reuse existing code, libraries, ... ?
  - Are there any standards on the horizon (OpenCL, PGAS,...)?

- Which is the best platform?
  - Performance
  - Costs (Acquisition and operation)
  - Implementation effort and productivity
  - Long term persistence
  - Application requirements (reliability, precision,
Model problem of CFD

- Solution of 3D incompressible Navier-Stokes equations
  \[ \partial_t u - \nu \Delta u + (u \cdot \nabla)u + \nabla p = f \quad \text{in } \Omega \]
  \[ \nabla \cdot u = 0 \quad \text{in } \Omega \]

- Prototypical solution with Chorin-type projection methods

- Iterative scheme, \(k=0, 1, \ldots\)

Compute \(\tilde{u}^{k+1}\):
\[
\frac{\tilde{u}^{k+1} - u^k}{\Delta t} + (u^k \cdot \nabla)u^k - \nu \Delta u^k = f^k \quad \text{in } \Omega
\]

Compute \(p^{k+1}\):
\[
\Delta p^{k+1} = \frac{1}{\Delta t} \nabla \tilde{u}^{k+1} \quad \text{in } \Omega
\]

Compute \(u^{k+1}\):
\[
\frac{u^{k+1} - \tilde{u}^{k+1}}{\Delta t} = -\nabla p^{k+1} \quad \text{in } \Omega
\]
Model problem of CFD

- Compute-intensive part: solution of the pressure equation
- Solve a Laplacian equation, i.e. a huge linear system of equations
  - Use e.g. a conjugate gradient method (or multi-grid)

Algorithm characteristics
- Algorithm basically consists of four operations:
  (typical of solutions of partial differential equations)
  - Sparse matrix-vector multiplication or linear stencil operation
  - (Non)-linear stencil operations
  - DAXPY / SAXPY vector updates
  - Scalar products

- Encapsulated by iterative methods and time stepping schemes
Computational intensity

Computational intensity = 
# floating point operations per transferred word / byte

All basic operations have computational intensity of O(1)!

Consequences:
- Bandwidth limitations on many devices
- Only a small fraction of peak performance can be achieved
Elements of projection step

- LSE in projection step solved with conjugate gradient method (cg)
- Each cg-step consists of:

<table>
<thead>
<tr>
<th>Function</th>
<th>Occ.</th>
<th>w [#words]</th>
<th>f [#flop]</th>
<th>f / w Comp. int.</th>
</tr>
</thead>
<tbody>
<tr>
<td>7p-stencil operation</td>
<td>1</td>
<td>2N</td>
<td>8N</td>
<td>4.0</td>
</tr>
<tr>
<td>Vector norm</td>
<td>1</td>
<td>N+1</td>
<td>2N-1</td>
<td>2.0</td>
</tr>
<tr>
<td>Dot product</td>
<td>1</td>
<td>2N+1</td>
<td>2N-1</td>
<td>1.0</td>
</tr>
<tr>
<td>Normalization</td>
<td>4</td>
<td>2N</td>
<td>2N</td>
<td>1.0</td>
</tr>
<tr>
<td>DAXPY vector update</td>
<td>3</td>
<td>3N+1</td>
<td>2N</td>
<td>0.66</td>
</tr>
</tbody>
</table>
Computational intensity

Numerical simulation of PDEs and CFD

- Stencil
- Blas 1+2
- SpMV
- LBM

- FFT
- BLAS 3

- O(1)
- O(log(N))
- O(N)
Simple performance model

Run time: \( T_R \geq \max \{ T_C, T_T \} \)
- \( T_R \) ... total run time of an algorithm
- \( T_C \) ... compute time, \( T_C \geq f / P \)
- \( T_T \) ... transfer time, \( T_T \geq 8w / B \)

Algorithm characteristics
- \( f \) ... number of floating point operations
- \( w \) ... number of memory transfers (words)

For 7-point stencils we have \( f / w \leq 4 \) flop / word

Effective performance:
- \( P_{eff} \leq f / T_R \leq fB / (8w) \)
- For 7-point stencils we have \( P_{eff} \leq B / 2 \) flop / byte

Hardware characteristics
- \( P \) ... peak performance
- \( B \) ... memory bandwidth
The Multicore Era
## Multicore Hardware

### Examples of current technologies

<table>
<thead>
<tr>
<th></th>
<th># Cores</th>
<th>Peak Perform. P</th>
<th>Bandwidth B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPUs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD Opteron dual core</td>
<td>2</td>
<td>8.8 GFlop/s</td>
<td>10.6 GB/s</td>
</tr>
<tr>
<td>Intel Clovertown quad core</td>
<td>4</td>
<td>37.2 GFlop/s</td>
<td>10.6 GB/s</td>
</tr>
<tr>
<td>Intel Nehalem quad core</td>
<td>4</td>
<td>44.8 GFlop/s</td>
<td>32.0 GB/s</td>
</tr>
<tr>
<td>AMD Shanghai</td>
<td>4</td>
<td>63.0 GFlop/s</td>
<td>20.6 GB/s</td>
</tr>
<tr>
<td>Sun Niagara T2 octo core</td>
<td>8</td>
<td>11.2 GFlop/s</td>
<td>42.6 GB/s</td>
</tr>
<tr>
<td><strong>Accel.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cell (2nd gen.)</td>
<td>8+1</td>
<td>102.4 GFlop/s</td>
<td>25.6 GB/s</td>
</tr>
<tr>
<td>ClearSpeed CSX700</td>
<td>192</td>
<td>96.0 GFlop/s</td>
<td>8.0 GB/s</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 280</td>
<td>240</td>
<td>80.0 GFlop/s</td>
<td>140.2 GB/s</td>
</tr>
</tbody>
</table>

(Theoretical values)
Is there a perfect device?

- Different capabilities and different focus
  - HPC tries to utilize all of them
  - Different effort and different success
    - Of course depending on the application

- Accelerators are highly capable on its own
  - But is this sufficient for large scale-problems?
    - Small memory
    - Slow connection to the host
  - Only solutions of subtasks?

What about forward scalability and long-time persistence?
## Basic properties of accelerators

<table>
<thead>
<tr>
<th></th>
<th>Cell</th>
<th>GPU</th>
<th>ClearSpeed</th>
<th>FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specialized to</td>
<td>Graphics, SP</td>
<td>Graphics, SP</td>
<td>DP</td>
<td>Integer</td>
</tr>
<tr>
<td>Performance in DP</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>Expensive</td>
</tr>
<tr>
<td>Ext. Bandwidth</td>
<td>High / Low</td>
<td>Low</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Int. Bandwidth</td>
<td>High</td>
<td>Extremely high</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>IEEE 754</td>
<td>Yes, in DP</td>
<td>Partially</td>
<td>Yes</td>
<td>Expensive</td>
</tr>
<tr>
<td>ECC-Correction</td>
<td>Partially</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Programming</td>
<td>Medium</td>
<td>Easy</td>
<td>Easy</td>
<td>Hard</td>
</tr>
<tr>
<td>Power consumpt.</td>
<td>Medium</td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Price</td>
<td>High</td>
<td>Low / High</td>
<td>High</td>
<td>Medium</td>
</tr>
</tbody>
</table>

DP... Double precision  
SP... Single precision
How to program?

- Multicore CPUs
  - Classically: OpenMP (intranodal) and MPI (internodal), or hybrid
    - Is the story still the same?
  - PGAS languages

- Accelerators and heterogeneous multicore
  - Vendor-specific programming environments
    - CUDA, Cell SDK, C^n, Brook+, ...
  - Extra learning curves, no portability of solutions
  - Software needs to be re-implemented

- Cross-plattform solutions diminish additional effort
  - OpenCL, RapidMind, CAPS, ...

static void Jac(DenseMat A, Rhs f, void *fdata);
init_rhs(fdata);
for (int n=0; n < n_max; n++) { ... }
Increasing levels of parallelism and heterogeneity
- Applications need to exploit parallelism on all system levels

Finer granularities on each level
- Algorithms need to be adapted

Limited local memory space
- Dedicated blocking strategies required

Data transfers need to be managed and orchestrated manually
- You need to know what’s going on in the cores
- More efficient resource utilization

Slow connection between accelerators and host
Levels of Parallelism

Parallelism spreads across multiple system levels

- How to express and exploit in software?

- Multiple nodes in the system

- Node level: several processors, GPUs, accelerators, ...

- Processor level: several cores

- Core level: several threads

- Vector and SIMD units, large registers

- Functional unit level: several adders, multipliers, ALUs

- Instruction level parallelism, EPIC, VLIW, pipelining
Hardware-Aware Computing
How to achieve best performance?

Optimization of computation

- SIMDization (uniform operations on large arrays)
- Vectorization (uniform operations on short vectors)
- Use BLAS routines and optimized solvers wherever possible
- Use single instead of double precision (on vector units)
  - Mathematical quality permitting?
- Do compiler jobs
  - Array addressing, pointer arithmetics, loop unrolling, literals,…
- Loop transformations, different block sizes, precomputing of common sub-expressions, reduction of branches, FMA operations, no divisions,
How to achieve best performance? (cont.)

- Optimization of data transfers
  - Memory transfer reduction strategies
    - Stencils instead of matrices, kernels computed on the fly
    - Single instead of double precision (mathematical quality?)
  - Blocking techniques for data reuse
    - Spatial blocking: processor, core, thread, cache, register, TLB,...
    - Temporal blocking: across several iterations (e.g. time skewing)
  - Prevent cache misses
    - Cache bypassing, in-place calculation
  - Overlap computation and communication
    - Manual double buffering
  - Prefetch data
    - Hardware and software
- Increase computational intensity in the algorithm
  - E.g. use higher order finite elements
  - Dense problems (locally), prefer BLAS 3 routines
How to achieve best performance? (cont.)

- **Optimization of data layout**
  - Predictable memory access patterns required
  - Coalesce data transfers
    - Structure-of-arrays or array-of-structures
  - Work on continuous memory regions
  - No random memory access
  - Follow alignment constraints

- **Autotuning and manual optimization**
  - High dimensional search space becomes intractable
  - Determine optimal implementation parameters by automatic search and many test runs
  - Tuning of compiler options
Impact on algorithms

- Use block-structured grids with patches of tensor-product grids
  - Data structure known a priori
  - Grid management overhead reduced
  - Uniform and predictable memory access patterns
  - Local refinement by displacement and stretching
  - Apply stencils instead of sparse-matrix vector multiplication
Impact on algorithms (cont.)

- Use explicit solvers
  - Allow for temporal blocking across several iterations

\[ u^{k+1} = u^k - \Delta t \cdot \nabla p^k \] in \( \Omega \)

- Subdivide whole application into small, simple and independent kernels

- Consider also different algorithmic approaches

Proposed approaches may give better performance, but probably not the best mathematical methods are used
Case Study: Implementation of Stencil Codes
3D Laplace stencil

- Very important operation on block-structured grids
  - PDEs, fluid dynamics, heat diffusion, electromagnetics
  - Jacobi, Gauss-Seidel, CG, multigrid ...
  - Image processing

- Local updates using a weighted subset of nearest neighbors
  - Each grid point associated with a single value
  - Each data touched seven times

\[
v_{i,j,k} = 6 \ u_{i,j,k} - u_{i+1,j,k} - u_{i,j+1,k} - u_{i,j,k+1} - u_{i-1,j,k} - u_{i,j-1,k} - u_{i,j,k-1}
\]

- Global sweeps with high locality

Blocking strategies required for data reuse
Problem decomposition

How to organize halos?

- **Data organization**
  - Data stored linearly (row-wise) in memory

- **Performance penalties**
  - Left and right halos consist of fragmented data
  - Possibly non-continuous

Data stored linearly (row-wise) in memory
Stencils on Multicore-CPUs

- Data automatically loaded into caches
  - Improve cache locality by sophisticated sweeps over the grid

- Typically: one array for read and one array for write operations
  - Out-of-place implementation
  - Data dependencies due to Gauss-Seidel type
  - Compulsory cache misses due to write conflicts

Possible remedies:
- Cache-bypassing by SSE intrinsics
- In-place implementation with a single array for read and write
Stencil formulas in 1d

- **Out-of-place algorithm**
  \[ y[i] = x[i+1] - 2 \times x[i] + x[i-1] \]

- **In-place algorithm with shift**
  \[ x[i-1] = x[i+1] - 2 \times x[i] + x[i-1] \]

- **Final result shifted to the left by one**
Space-blocked 1d-stencil

Original domain

\[ \begin{array}{cccccccccc}
  a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & a_9 & a_{10} & a_{11} \\
\end{array} \]

Stencil

Steps

Processed and shifted domain

a ... original values
b ... processed values
### Space- and time-blocking in 1d

- **3 time steps blocked, result shifted by 3**

<table>
<thead>
<tr>
<th>a₀</th>
<th>a₁</th>
<th>a₂</th>
<th>a₃</th>
<th>a₄</th>
<th>a₅</th>
<th>a₆</th>
<th>a₇</th>
<th>a₈</th>
<th>a₉</th>
<th>a₁₀</th>
<th>a₁₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>b₁</td>
<td>b₂</td>
<td>b₃</td>
<td>b₄</td>
<td>a₅</td>
<td>a₆</td>
<td>a₇</td>
<td>a₈</td>
<td>a₉</td>
<td>a₁₀</td>
<td>a₁₁</td>
<td>a₀</td>
</tr>
<tr>
<td>c₂</td>
<td>c₃</td>
<td>c₄</td>
<td>c₅</td>
<td>c₆</td>
<td>a₇</td>
<td>a₈</td>
<td>a₉</td>
<td>a₁₀</td>
<td>a₁₁</td>
<td>a₀</td>
<td>a₁</td>
</tr>
<tr>
<td>d₃</td>
<td>d₄</td>
<td>d₅</td>
<td>d₆</td>
<td>d₇</td>
<td>d₈</td>
<td>d₉</td>
<td>d₁₀</td>
<td>c₁₀</td>
<td>c₁₁</td>
<td>b₁₀</td>
<td>b₁₁</td>
</tr>
<tr>
<td>b₀</td>
<td>a₀</td>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>a₅</td>
<td>a₆</td>
<td>a₇</td>
<td>a₈</td>
<td>a₉</td>
<td>a₁₀</td>
</tr>
<tr>
<td>b₂</td>
<td>b₃</td>
<td>b₄</td>
<td>b₅</td>
<td>b₆</td>
<td>b₇</td>
<td>b₈</td>
<td>b₉</td>
<td>b₁₀</td>
<td>b₁₁</td>
<td>b₀</td>
<td>a₀</td>
</tr>
<tr>
<td>c₂</td>
<td>c₃</td>
<td>c₄</td>
<td>c₅</td>
<td>c₆</td>
<td>c₇</td>
<td>c₈</td>
<td>c₉</td>
<td>c₁₀</td>
<td>c₁₁</td>
<td>b₁₀</td>
<td>b₁₁</td>
</tr>
<tr>
<td>d₃</td>
<td>d₄</td>
<td>d₅</td>
<td>d₆</td>
<td>d₇</td>
<td>d₈</td>
<td>d₉</td>
<td>d₁₀</td>
<td>c₁₀</td>
<td>c₁₁</td>
<td>b₁₀</td>
<td>b₁₁</td>
</tr>
<tr>
<td>b₀</td>
<td>a₀</td>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>a₅</td>
<td>a₆</td>
<td>a₇</td>
<td>a₈</td>
<td>a₉</td>
<td>a₁₀</td>
</tr>
<tr>
<td>b₂</td>
<td>b₃</td>
<td>b₄</td>
<td>b₅</td>
<td>b₆</td>
<td>b₇</td>
<td>b₈</td>
<td>b₉</td>
<td>b₁₀</td>
<td>b₁₁</td>
<td>b₀</td>
<td>a₀</td>
</tr>
<tr>
<td>c₂</td>
<td>c₃</td>
<td>c₄</td>
<td>c₅</td>
<td>c₆</td>
<td>c₇</td>
<td>c₈</td>
<td>c₉</td>
<td>c₁₀</td>
<td>c₁₁</td>
<td>b₁₀</td>
<td>b₁₁</td>
</tr>
<tr>
<td>d₃</td>
<td>d₄</td>
<td>d₅</td>
<td>d₆</td>
<td>d₇</td>
<td>d₈</td>
<td>d₉</td>
<td>d₁₀</td>
<td>c₁₀</td>
<td>c₁₁</td>
<td>b₁₀</td>
<td>b₁₁</td>
</tr>
<tr>
<td>b₀</td>
<td>a₀</td>
<td>a₁</td>
<td>a₂</td>
<td>a₃</td>
<td>a₄</td>
<td>a₅</td>
<td>a₆</td>
<td>a₇</td>
<td>a₈</td>
<td>a₉</td>
<td>a₁₀</td>
</tr>
</tbody>
</table>

*a ... original values, t=0  b,c,d ... proc. values, t=1,2,3*
Parallel space- and time-blocking in 1d2 processors / cores with replicated work at the interface

- 2 time steps blocked, result shifted by 2
Space-blocking in 2d

Schematic view on the grid
Space- and time-blocking in 3d

- Code example

```c
for( bz = 0; bz < zsize ; bz += bsz )
    for( by = 0; by < ysize; by += bsy )
        for( bx = 0; bx < xsize; bx += bsx )
            for( t = 0; t < tmax; t++ ) {
                tc = tmax - t - 1;
                xmin = max(0, bx - 2*t); xmax = min(xsize + 2*tc, bx+bsx - 2*t);
                ymin = max(0, by - 2*t); ymax = min(ysize + 2*tc, by+bsy - 2*t);
                zmin = max(0, bz - 2*t); zmax = min(zsize + 2*tc, bz+bsz - 2*t);
                for( z = zmin; z < zmax; z++ )
                    for( y = ymin; y < ymax; y++ )
                        for( x = xmin; x < xmax; x++ )
                            a[x, y, z] = stencil(a[x+1, y+1, z+1],
                                              a[x , y+1, z+1], a[x+2, y+1, z+1], a[x+1, y , z+1],
                                              a[x+1, y+2, z+1], a[x+1, y+1, z ], a[x+1, y+1, z+1]);
            }
```
Further optimization steps

- Parallel data allocation
  - Thread and memory pinning (sched_setaffinity)

- Cache-bypassing with SSE-instructions (mm_stream_pd, movntpd)

- Parallelization along three axes to reduce overlap and replicated work

- Manual SIMDization on SSE-units

- Register blocking

- Explicit software prefetch (compiler pragmas, SSE-intrinsics)

- Array padding
Stencil performance

- Intel Nehalem E5520 quad core, 2-way node, up to 2 threads per core
- 512 x 512 x 512 grid
Stencil performance

- Intel Clovertown X5355 quad core, 2-way node
- 512 x 512 x 512 grid
Stencil performance (cont.)

- AMD Shanghai 2384 quad core, 2-way node, HP DL165 G5
- 512 x 512 x 512 grid
Stencils on local store based platforms

- Block data to local memory
  - 256 KB local store on Cell’s SPEs
  - 16 KB shared memory on NVIDIA’s GTX 280
  - 6 KB local memory on PEs of ClearSpeed’s CSX600 / CSX700

- Strategies
  - Minimize overlap of sub-block interfaces, i.e. size of halos
  - Minimize surface-to-volume ratio
    - Minimize replicated work
Streaming in and out planes

- Best approach: work on 6 planes in parallel (on a single compute unit)
  - Hold 3 large input planes for stencil computation
  - Hold 1 output plane
  - Overlap communication and computation
  - Double buffer read and write operations by using 2 additional planes

- Data exchange between processes only in 2 (or 4) directions
- Requires intermediate data exchange (loading and storing planes)
  - Not possible in pure stream processing models
Managing stencil data

- On 256 KB local store on Cell’s SPEs
  - Hold 3 input planes, 1 output plane, 1 read plane, 1 write plane
  - Do not block in single-stride direction
    - Plane size: \((256+2) \times (16+2)\) for a \(256^3\) grid
    - Plane size: \((512+2) \times (6+2)\) for a \(512^3\) grid

- On 16 KB shared memory on NVIDIA’s GTX 280
  - Hold 1 plane in shared memory
    - Max. plane size 32 x 64
  - Hold thread-private data in registers
  - Stream input from GM to registers to shared memory to registers

Above / below the plane data is private to thread
Within the plane data is shared by 5 threads
Additional difficulties

- **Data alignment constraints for optimal bandwidth**
  - 128 Byte on Cell (cache line of PPE)
  - 128 Byte on NVIDIA GPUs (double values, size of half-warp is 16 threads)

- **Transfer size for optimal throughput**
  - Multiples of 128 Byte on Cell
  - Multiples of 128 Byte for CUDA (in double precision)

- **Shuffling of short vectors (2 doubles on Cell)**
  - Native load and store operations only for 128bit-vectors (2 double values)
  - Misaligned loads require two vectors and shuffle instructions
  - Linear stencil in unit stride direction requires shift by 1
Additional difficulties

- **Optimal block size**
  - Many parameter choices possible for CUDA grids and thread blocks

- **Boundary conditions**
  - Typically cyclic, Dirichlet or Neumann type
  - Require local modifications

- **NUMA systems**
  - Cell QS-blades are NUMA systems
    - Parallel data allocation, thread pinning and memory affinity

- Possibly no full use of FMA units

- **Shared resources and hardware contention**
  - Register pressure on GPUs
Remarks on Cell computing

- Architecture and data layout is more critical than on homogeneous processor types
- Local store is comparably large
  - Handling of 3D problems is feasible

Comparison of Cell to

- GPUs
  - GPUs are much faster in terms of SP performance and internal bandwidth, and easier to program
  - Cell is much more versatile, no restriction to stream processing
  - Cell programming much more tedious
- X86 CPUs
  - Nehalem is as fast as Cell (in terms of bandwidth), but much easier to handle
  - Waiting for the new Cell generation (?)
Cell programming

- Parallelization by manual data and workload decomposition and domain sub-structuring
  - Send SPEs parts of the global data for processing
  - Pthreads handle work on SPEs

- Message based communication
  - All data transfers user initiated by DMA transfers
    - Specific data structures for addresses, data and control
    - Communicate start addresses and data size
      - spe_mfcio_put, spe_mfcio_tag_status
      - spu_mfcdma64, spu_writech, spu_mfc_stat
Cell programming (cont.)

- Write two programs:
  - One for the PPE, one for the SPEs
  - Different compilers, different instruction sets

- Load, initiate and terminate SPE programs by PPE
  - `spe_context_create`, `spe_program_load`, `spe_image_open`, `pthread_create`
  - `spe_context_destroy`, `spe_image_close`, `pthread_join`

- Synchronize both programs via messages and mailboxes
  - `spe_out_mbox_status`, `spe_out_mbox_read`, `spe_in_mbox_write`
  - `spu_read_in_mbox`, `spu_write_out_mbox`

- Align data on PPE and SPEs to 128 Byte boundaries (cache line size)
  - Remark: typically no halos in unit stride direction, since local store is big enough
Cell programming (cont.)

- Transfer data as multiples of 128 Bytes (16 doubles)
  - Small workloads create bottlenecks
  - Optimal transfer size is 4 kB to 16 kB
  - Use manual double buffering

- Vector intrinsics on SPEs: 128-bit vectors (2 double values)
  - Stencil computation in unit stride direction (shifts)
  - Loads and stores only for vectors
  - spu_shuffle, spu_add, spu_sub, spu_msub

- BLAS routines
  - Stencil computation in non-unit stride direction
  - dcopy_spu, dscal_spu, daxpy_spu

- Register blocking
  - Ribbons instead of pencils
Stencil performance on Cell

- IBM BladeCenter QS22 with 2 PowerXCell 8i processors
- 32 GB memory
- 256 x 256 x 256 grid

Stencil Performance QS22

<table>
<thead>
<tr>
<th># SPEs</th>
<th>GFLOP/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>11</td>
</tr>
</tbody>
</table>
Remarks on GPU computing

- Hardware characteristics
  - Superior internal bandwidth
  - Impressive SP/DP performance
  - But: all advantages lost if you need to talk to the host CPU

- Stream processing model
  - High performance for uniform operations with high computational intensity on huge arrays
  - But limited cooperation between threads
  - Temporal locality of data cannot be exploited meaningfully
    - Data cannot be kept on the cores across kernel calls
    - Keep thread-local data in 64 KB registers
      - 64 KB shared by all threads in a block
CUDA performance

Performance on an ASUS ENGTX280 / NVIDIA G200, N=2^{25}

- Copy and scale bandwidth
  - 122 GB/s, 7.2 Gupdates/s for a 1D data layout
  - 112 GB/s, 7.0 Gupdates/s for a 3D data layout

- 3D: 3-point stencil in z-direction, no halos, registers/shared mem.
  - 108 GB/s, 6.7 Gupdates/s

- 3D: restrict to interior points of 32x16 thread blocks, no halos
  - 3-point stencil in z-direction, 4 flop per grid point
    - 99 GB/s, 5.5 Gupdates/s
  - 7-point Laplacian stencil, 8 flop per grid point
    - 58 GB/s, 3.2 Gupdates/s

- 3D: 7-point stencils with halos, misaligned data access
  - 52.5 GB/s, 3.3 Gupdates/s, 26.3 GFlop/s
### Stencil performance on hardware

- 7p-stencil is bandwidth-bound on most platforms
- No time-blocking
- 256x256x256 or 512x512x512 grid

<table>
<thead>
<tr>
<th></th>
<th>Bandwidth</th>
<th>Stencil max.</th>
<th>Experimental</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPUs</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMD Opteron dual core</td>
<td>10.6 GB/s</td>
<td>5.3 GFlop/s</td>
<td>2.8 GFlop/s</td>
</tr>
<tr>
<td>Intel Clovertown quad core</td>
<td>10.6 GB/s</td>
<td>5.3 GFlop/s</td>
<td>3.5 GFlop/s</td>
</tr>
<tr>
<td>Intel Nehalem quad core</td>
<td>32.0 GB/s</td>
<td>16.0 GFlop/s</td>
<td>8.5 GFlop/s</td>
</tr>
<tr>
<td>AMD Shanghai quad core</td>
<td>20.6 GB/s</td>
<td>10.3 GFlop/s</td>
<td>4.0 GFlop/s</td>
</tr>
<tr>
<td><strong>Acc.</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cell (2nd gen.)</td>
<td>25.6 GB/s</td>
<td>12.8 GFlop/s</td>
<td>7.4 GFlop/s</td>
</tr>
<tr>
<td>NVIDIA GeForce GTX 280</td>
<td>140.2 GB/s</td>
<td>70.1 GFlop/s</td>
<td>26.3 GFlop/s</td>
</tr>
</tbody>
</table>
# Memory limitations

## Main memory
- **GPUs**: 512 MB to 4 GB
- **Cell**: 256 MB to 16 GB
- **ClearSpeed**: 512 MB to 2 GB

## Local memory
- **Cell**: 256 KB (private)
- **NVIDIA GPUs**: 16 KB (shared)
- **ClearSpeed**: 6 KB (private)

3D simulation?  
Spatial blocking?
Memory bandwidth

Host to device
- **Cell:** 125 MB/s (PS3 / Ethernet)
  2 GB/s (QS22 / Dual Infiniband 4x)
  8 GB/s (GigaAccel / PCI 2.0 x16)
- **ClearSpeed:** 4 GB/s (PCIe 1.1 x16)
- **GPUs:** 8 GB/s (PCIe 2.0 x16)

Main memory to cores
- **Cell:** 25.6 GB/s
- **NVIDIA GPUs:** 140.0 GB/s
- **ClearSpeed:** 3.2 / 8.0 GB/s
Unmentioned topics

- **Further constraints**
  - ECC error correction
  - Conformity of floating point operations to IEEE 754 norm
    - Reproducibility of results across platforms

- **What about FPGAs and others?**
  - Hardware-centered programming not intuitive
  - Bandwidth and memory restrictions limit performance for data-intensive applications (e.g. CFD)
  - Double precision very expensive on FPGAs

- **How to get around?**
  - Application in numerical simulation have to be tailored and adapted to these devices
  - Exploit particular hardware capabilities
Summary

- Hardware-aware computing is the key factor for performance
  - Best performance can only be achieved by following a huge bunch of hardware-dependent optimization guidelines

- Bandwidth and its efficient utilization are far more important than pure GFlop/s performance numbers

- Critical issues
  - Data and memory layout
  - Coalesced and aligned memory access
  - Blocking techniques for data re-use

- Memory size and slow connection to host are limiting factors for accelerators in numerical simulation

- Always think of forward scalability!
Acknowledgements

The Shared Research Group at KIT is granted by Hewlett-Packard and the Concept for the Future of the Karlsruhe Institute of Technology (KIT) within the framework of the German Excellence Initiative.

Thanks to
- Rainer Buchty
- HP
- Vincent Heuveline
- Werner Augustin
- Dimitar Lukarski
- Michael Rückauer
Contact

Further information available at

- http://srg-multicore.rz.uni-karlsruhe.de
- jan-philipp.weiss@kit.edu

Thank you for your attention.