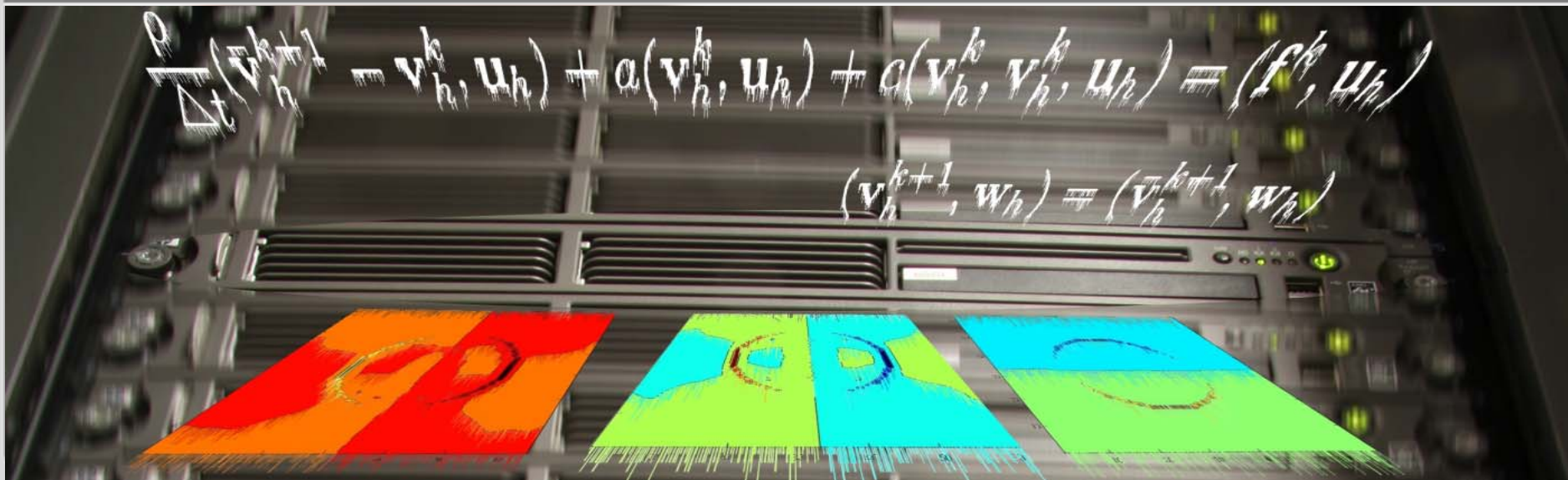


Aspects of Numerical Algorithms, Iterative Solvers and Preconditioners on GPUs

Tutorial Scientific Computing on GPUs – PPAM 2011, Sep 11, 2011

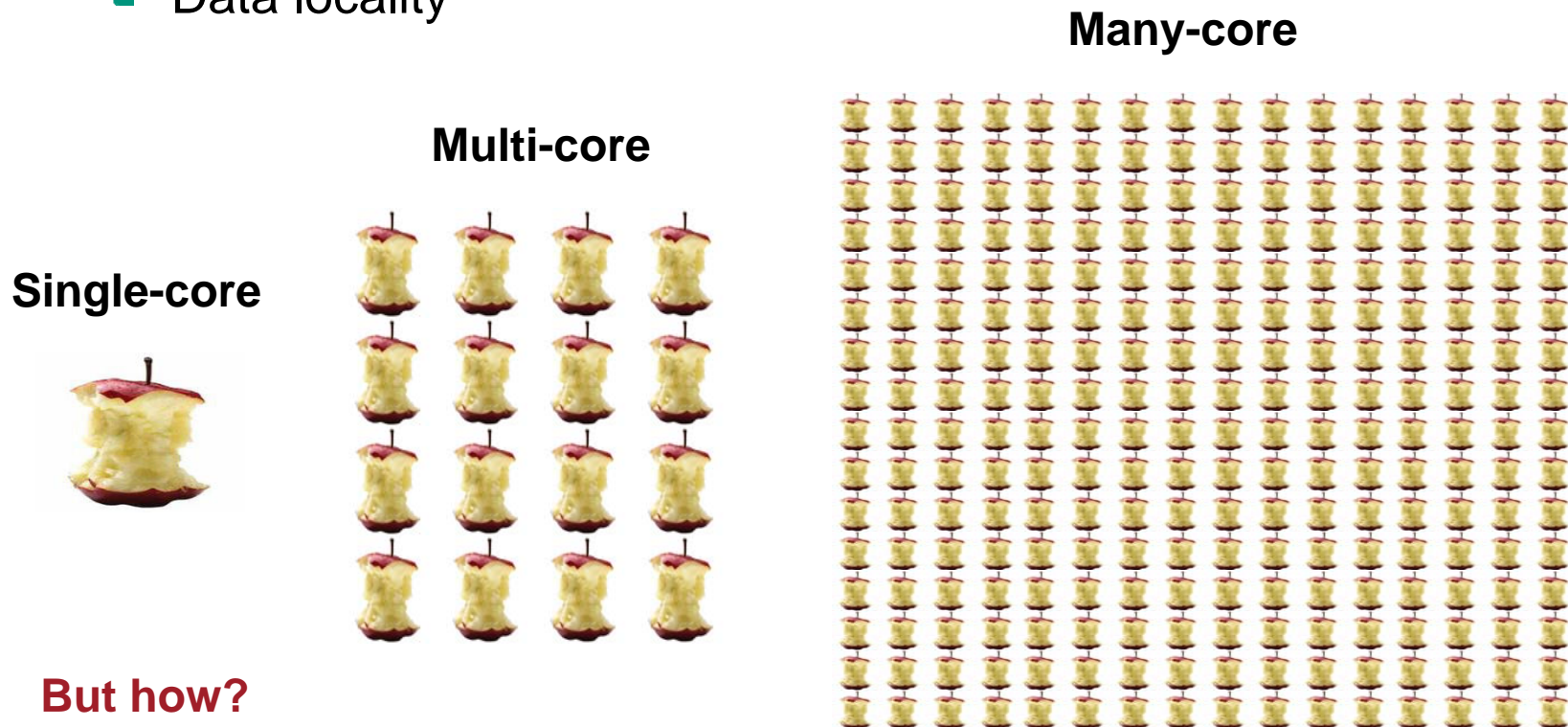
Jan-Philipp Weiss

Engineering Mathematics and Computing Lab (EMCL) / SRG New Frontiers in HPC



The Challenge of GPGPU Computing

- Make your algorithms ready for
 - Fine-grained parallelism
 - Scalability with respect to thousands of threads
 - Data locality



Simple example: saxpy vector update

■ Sequential kernel / CPU

```
void saxpy_serial(int n, float a, float *x, float *y){  
    for (int i = 0; i < n; i++)  
        y[i] = a*x[i] + y[i];  
}  
saxpy_serial(n, 2.0, x, y);
```

■ Many-threaded kernel / CUDA

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y){  
    int i = blockIdx.x*blockDim.x + threadIdx.x;  
    if (i < n) y[i] = a*x[i] + y[i];  
}  
int nblocks = (n + 255) / 256;  
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Characteristics of the saxpy-kernel

- Easily and optimally parallelizable
 - No data dependencies
 - No interaction / communication between threads
 - No synchronization / no thread divergence
 - Each thread processes its own vector component
- Can we reach peak performance on the GPU?
 - Yes – we have uniform operations on huge arrays
 - Yes – there are no branches
 - Yes – there is no synchronization / communication
 - **No !!! There is not enough work to be done on each data element!**

Which fraction of peak can we reach?

Kernel run time $T_R \geq \max \{ T_C, T_T \}$

- T_C ... Computing time
- T_T ... Data transfer time
- Ratio used for classifying compute-bound and memory-bound kernels

■ Characteristics of saxpy

- f ... # floating point operations = $2N$ für saxpy
- w ... # requested data words (read and write) = $3N+1$ for saxpy
- f/w ... computational intensity = $2/3$ für saxpy

■ Characteristics of hardware

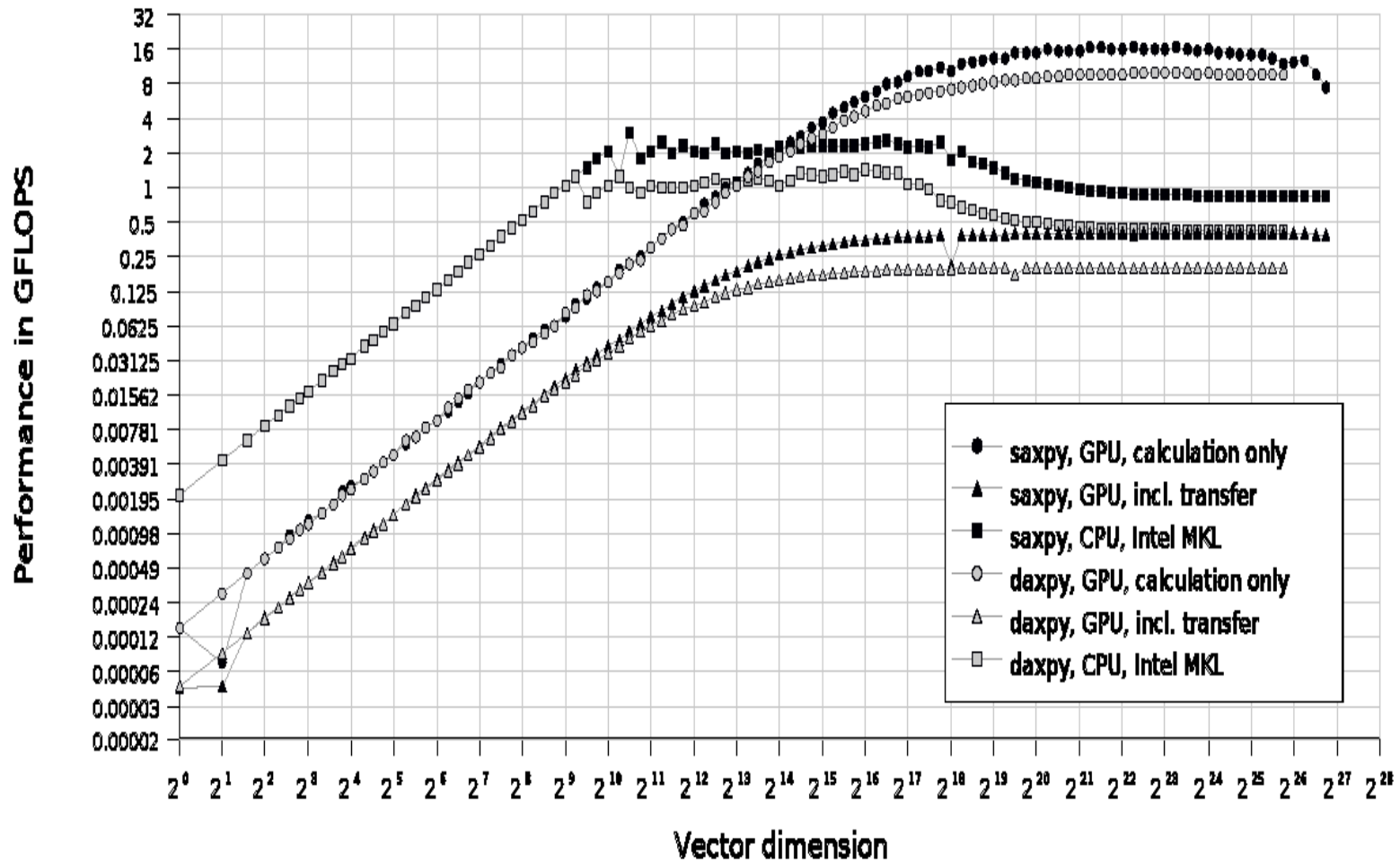
- P ... Peak Performance (in GFlop/s) = 933 GFlop/s (GTX 280)
- B ... Peak Bandwidth (in GByte/s) = 102 GB/s

Effective performance

- Lower bounds
 - $T_C \geq f / P$ and $T_T \geq 4w / B$ (for single precision)
- Effective performance
 - $P_{\text{eff}} = f / T_R \leq \min \{ P, f B / (4w) \}$
- We find: $P_{\text{eff}} \leq f B / (4w)$
 - Computational intensity defined by f / w
- For BLAS 3 routines (matrix-matrix multiplication)
 - f / w is of order $O(N)$! (for matrix size $N \times N$)
 - Hence: no bandwidth limitation for big problem size
- **But for saxpy**
 - $P_{\text{eff}} \leq B / 6 \text{ flop/byte} = 17 \text{ Gflop/s} = 1,8 \% \text{ peak !}$
 - **Practical experience shows ... !**

This is reality ...

■ SAXPY and DAXPY performance



Initial cognitions

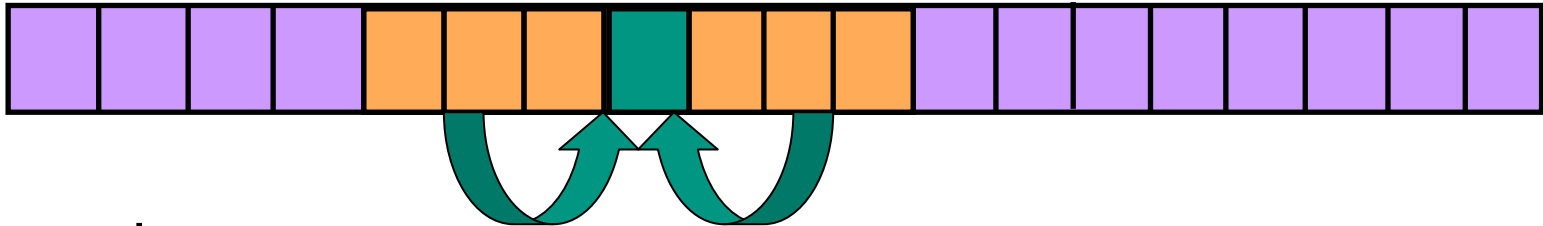
- For optimal performance rely on
 - Compute-intensive kernels
 - Uniform processing of huge arrays
 - Linear, coalesced and aligned memory access

- For better performance
 - Decrease the number of data transfers
 - Keep data close to the processing units
 - Make use of shared memory and huge registers
 - Increase the number of operations per data
 - Increase computational intensity

How can I increase data locality?

■ Example: 1d box filter

- Summing up neighboring entries (with radius 3) in a 1d-array, i. e. 7 entries
- Simple stencil operation



■ Procedure

- Each thread block computes blockDim.x sums
- Load $(\text{blockDim.x} + 2 * \text{radius})$ input elements into shared memory
 - Including halo cells at the left and right border
- Compute sums from data in shared memory
- Write output to device memory

Example program

```
__global__ void stencil(int* output, int* input) {
    __shared__ int s[blockDim.x+2*RADIUS];           // allocate shared mem
    int ix = blockIdx.x*blockDim.x + threadIdx.x;    // assign threads
    s[threadIdx.x + RADIUS] =input[ix];              // each threads reads single input
    if (threadIdx.x < RADIUS)
        s[threadIdx.x] =input[ix-RADIUS];           // load left halo
    if (blockDim.x-threadIdx.x < RADIUS)
        s[threadIdx.x + RADIUS] =input[ix+RADIUS]; // load right halo
    __syncthreads();                                 // synchronize loads
    int value=0;
    for (int i=0; i<=2*RADIUS, i++)
        value+= s[threadIdx.x+i];                  // compute sums
    output[ix] = value;                             // write output
}
```

Optimality of the implementation?

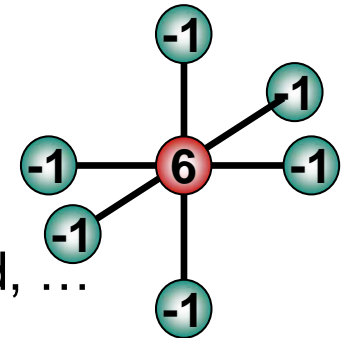
- Make use of shared memory!
 - Each data loaded a single time not seven times
 - SM = user managed cache
 - Performance boost: $4x \quad ((7l+1s) / (1l+1s))$

- What is not yet optimal?
 - Thread divergence
 - Thread block loads inner domain in one go
 - Small boundary zones loaded separately
 - Nearly all threads have to wait (2x)
 - Memory access
 - Data access of thread blocks not aligned
 - Address not a multiple of 16 (64 byte)
 - Bank conflicts?

Stencil computations

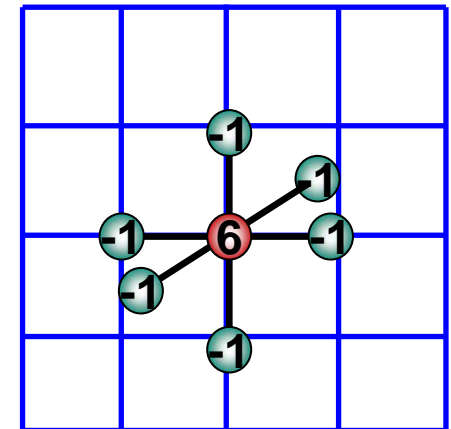
■ Laplacian stencils

- Important kernel for block-structured grids
- Poisson, Navier-Stokes, heat transfer, ...
- Jacobi, Gauss-Seidel, conjugated gradient, multigrid, ...



■ Local update by weighted sum of neighbor values

- Each grid point associated to a unknown
- Each unknown is accessed seven times



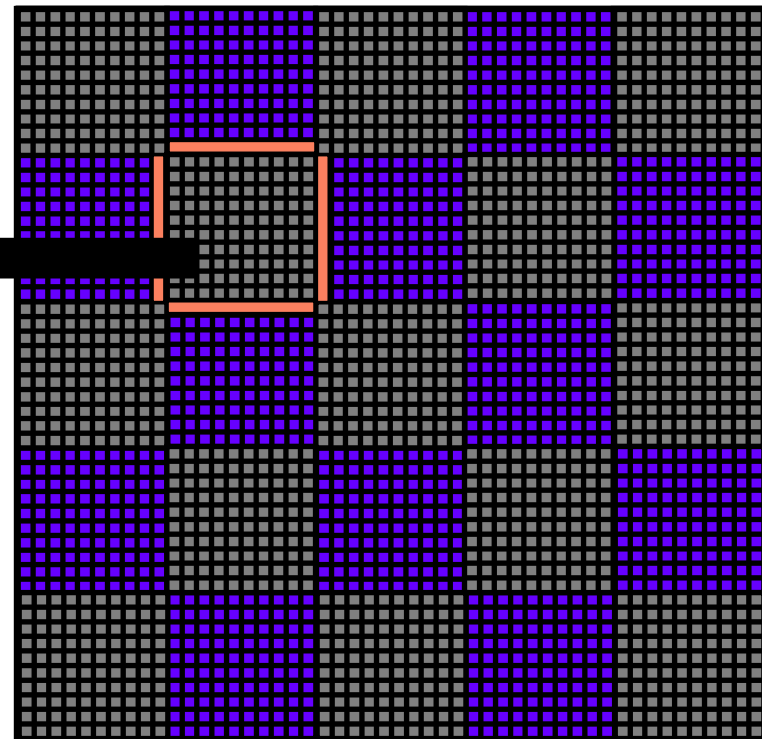
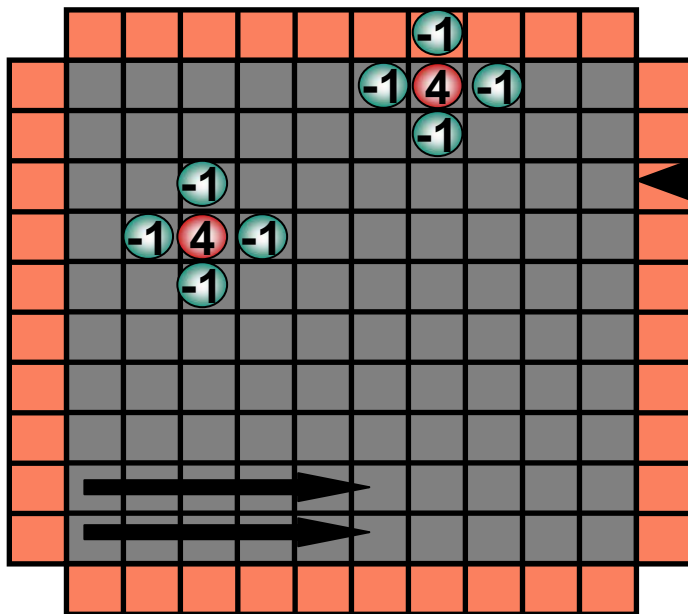
$$v_{i,j,k} = 6 u_{i,j,k} - u_{i+1,j,k} - u_{i,j+1,k} - u_{i,j,k+1}$$

Use blocking strategies for data reuse (local and temporal)

Problem decomposition

How to organize halos?

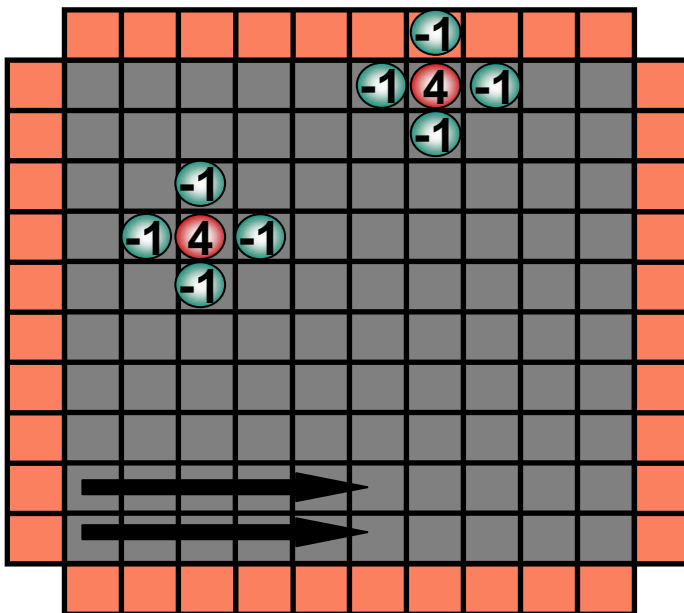
- Data organization
 - Data are linear (row-wise) in memory



Problem decomposition

How to organize halos?

- Data organization
 - Data are linear (row-wise) in memory



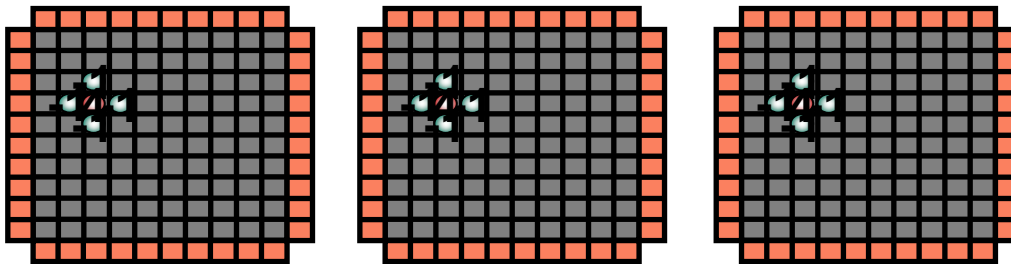
- Performance drops
 - Left and right halos are fragmented
 - Non-coalesced memory access
 - Inner domain or halos not aligned
 - Another thought: possible vectorization requires permutation in direction of fast index
 - Bad surface-to-volume ratio

Stencils on local-store platforms

- Decompose data into blocks for local store
 - 256 Kb local store on SPEs of Cell BE processor
 - 48 Kb shared memory on NVIDIA 20 series / Fermi
 - 16 Kb Shared Memory on NVIDIA 10 series / Tesla
 - 6 Kb local store on ClearSpeed CSX600 / CSX700

Strategies

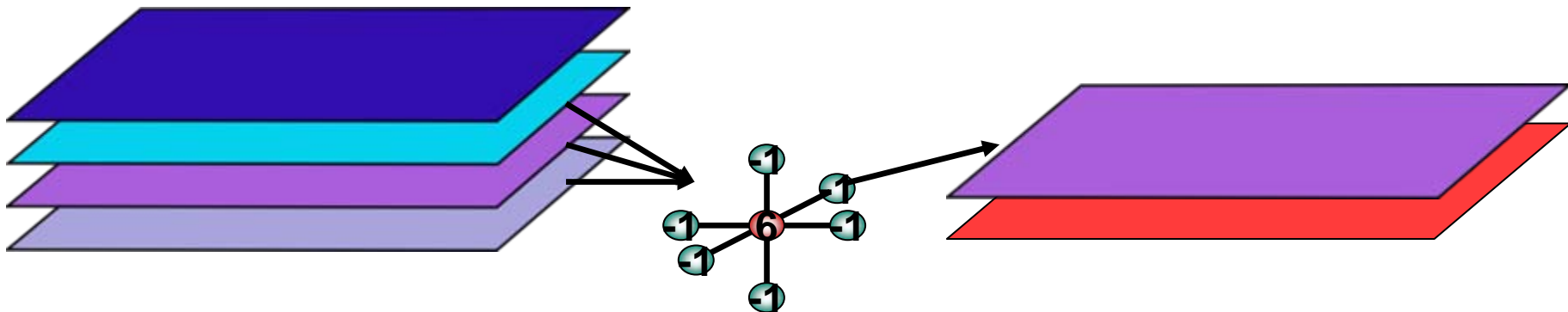
- Minimize overlap of sub-blocks (halos)
- Maximize volume-to-surface ratio



Streaming in and out planes

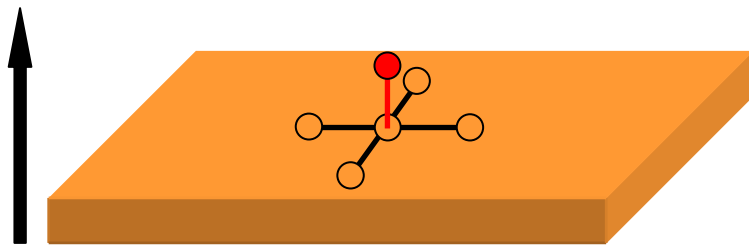
- Good approach: work on six planes in one go (in local memory)
 - 3 large planes for input
 - 1 plane for output
 - 2 additional planes for overlapping of communication and computation

- Problem: small planes due to memory limitation



Stencil computation on GPUs

- Only 16 Kb on shared memory of NVIDIA GTX 280
 - Keep 1 plane in shared memory
 - Maximal size of plane: 32 x 64
 - Data below and above plane only required by a single thread
 - Keep data in registers (data loaded into registers)
 - Stream data from registers into shared memory into registers when moving on to process the next plane
 - See papers/presentations by Paulius Micikevicius



Data above / below plane are thread-private
 In the plane: 5 threads share data

And now: How to solve my PDE problem?

■ Basic ingredients:

- Solver for linear systems of equations
- Time-stepping schemes
- Which solver should I choose?
 - Just any? Lowest complexity? Fastest speed?

■ How to discretize my problem?

- Finite differences on structured grids?
- Finite elements on unstructured grids?
- Explicit or implicit schemes?
 - This is a disruptive discussion!

Structured vs. unstructured grids

■ Structured grids

- Equidistant cartesian grids or tensor product grids with fixed topology / number of neighbors
- Simple data structures, regular data access patterns, low complexity of implementations
- But how to resolve local singularities and complex geometries / boundaries ?

■ Unstructured grids

- Maximal flexibility and problem-adapted grids
- Stencils / band matrices replaced by sparse matrices
- Irregular data access and complex implementation
- Lower number of unknowns for same error quality possible

Explicit or implicit schemes

■ Implicit schemes

- Solution of linear systems required in each time step
 - Additional costs
- Improved stability and accuracy
- Parallelization often on algorithmic level

■ Explicit schemes

- Only requires application of stencils or sparse matrix-vector multiplication
- Time step constraints usually lead to a large number of time steps
- Improved scalability due to less couplings
 - Example: lattice Boltzmann methods
- Parallelization typically on geometric level (easier)

Complexity of LSE-solvers for the Laplace problem

Asymptotical work depends on the chosen method

	dim = 2	dim = 3
Band-Gauss	$O(N^2)$	$O(N^{7/3})$
Jacobi, Gauss-Seidel	$O(N^2)$	$O(N^{5/3})$
Nested dissection	$O(N^{3/2})$	$O(N^2)$
Conjugated gradient (CG), SOR	$O(N^{3/2})$	$O(N^{4/3})$
SSOR-CG	$O(N^{5/4})$	$O(N^{7/6})$
FFT, cyclic reduction	$O(N \log N)$	$O(N \log N)$
Multigrid, cascadic iteration	$O(N)$	$O(N)$

N ... matrix dimension, $h = 1 / n$... mesh spacing, $N = n^{\text{dim}}$

Conjugated gradient method

random $x^0 \in \mathbb{R}^n$

$d^0 := r^0 := b - Ax^0$

FOR $k = 1, 2, \dots$

IF $d^k = 0$

break

ELSE

$$\alpha_k := \frac{\langle r^k, r^k \rangle}{\langle d^k, Ad^k \rangle}$$

$$x^{k+1} := x^k + \alpha_k d^k$$

$$r^{k+1} := r^k - \alpha_k Ad^k$$

$$\beta_k := \frac{\langle r^{k+1}, r^{k+1} \rangle}{\langle r^k, r^k \rangle}$$

$$d^{k+1} := r^{k+1} + \beta_k d^k$$

(New approximation)

(New residual)

(New search direction)

How can I solve this in parallel?

Building blocks of cg

- Dot product $\alpha = \vec{x} \cdot \vec{y}$ where $\alpha \in \mathbb{R}$, $\vec{x}, \vec{y} \in \mathbb{R}^n$
- Vector updates $\vec{x} = \vec{x} + \alpha \vec{y}$ where $\alpha \in \mathbb{R}$, $\vec{x}, \vec{y} \in \mathbb{R}^n$
- Sparse matrix-vector multiplication $\vec{v} = A\vec{u}$ where $A \in \mathbb{R}^{n \times n}$, $\vec{u}, \vec{v} \in \mathbb{R}^n$

And now: Use parallel version of these routines

- Nothing to do at all, use library implementations (CUBLAS, CUSPARSE)

The only problems left:

- Which compressed data format for sparse matrices?
- What about preconditioning?

What About the Algorithms?!

We have to rethink our methods and approaches!

Typical way of parallel thinking (?)

- Cluster-like problem decomposition with many fat nodes
 - Substructure problem into huge blocks (same size)
 - Process blocks in parallel
 - Minimize communication between blocks
 - But: sequential processing on the block-level

Alternative way of parallel thinking

- Decompose problem into blocks (of varying size)
 - There should only be a lower bound on the block size
- Process blocks in sequential order (or in parallel)
- But: use scalable parallelism on the block-level !

What About the Algorithms?!

We have to rethink our methods and approaches!

Typical way of parallel thinking (?)

- Cluster-like problem decomposition with many fat nodes
 - Substructure problem into huge blocks (same size)
 - Process blocks in parallel
 - Minimize communication between blocks
 - But: sequential processing on the block-level

Alternative way of parallel thinking

- Decompose problem into blocks (of varying size)
 - There should only be a lower bound on the block size
- Process blocks in sequential order (or in parallel)
- But: use scalable parallelism on the block-level !

The Need for Parallel Preconditioners

Krylov-type solvers like *Conjugate Gradient methods* (CG) heavily rely on preconditioners.

Parallel preconditioning is a challenging task!

The preconditioner should decrease the number of necessary iterations in the iterative solver loop

- In each step of the iterative solver we need to solve an additional linear system $Mz = r$
- Additional work should not outweigh benefits by preconditioning
- Classical preconditioners are based on triangular decompositions
 - Gauß-Seidel, SOR, ... (additive decompositions)
 - ILU(0), ILU(p) (multiplicative decomposition)
- Problem: Solution of triangular systems is a sequential task

Splitting-type Preconditioners

Idea:

Use the additive block-based decomposition

$$A = D + L + R$$

where L is lower-triangular, R is upper-triangular, and D only has blocks on the diagonal.

Then, the splitting-type preconditioners are:

- Jacobi with $M := D$
- Symmetric Gauß-Seidel (SGS) with $M := (D + L)D^{-1}(D + R)$
- SOR with $M := \frac{1}{\omega}(D + \omega L)$ or $M := \frac{1}{\omega}(D + \omega R)$
- SSOR with $M := \frac{1}{\omega(2-\omega)}(D + \omega L)D^{-1}(D + \omega R)$

How to solve the preconditioned system $Mz = r$ in parallel?

- Write the problem in block-form with diagonal blocks with sizes $b_i \times b_i$, where $i = 1, \dots, B$, and B is the number of blocks
- Obtain fine-grained parallelism based on matrix-vector multiplication routines and vector updates

SGS with $M := (D + L)D^{-1}(D + R)$ now reads

$$x_i := D_i^{-1} \left(r_i - \sum_{j=1}^{i-1} L_{i,j} x_j \right) \text{ for } i = 1, \dots, B$$

$$y_i := D_i x_i \text{ for } i = 1, \dots, B$$

$$z_i := D_i^{-1} \left(y_i - \sum_{j=1}^{B-i} R_{i,j} z_{i+j} \right) \text{ for } i = B, \dots, 1$$

D_1	R_{11}	R_{12}	R_{13}
L_{21}	D_2	R_{21}	R_{22}
L_{31}	L_{32}	D_3	R_{31}
L_{41}	L_{41}	L_{41}	D_4

Only problem left: How to compute D_i^{-1} in parallel?

Parallel Version: Use Multi-coloring

Transform A such that all D_i are diagonal matrices

Multi-coloring permutation

- Seek permutation π that re-orders the original matrix A into $A_\pi := \pi A \pi^{-1}$, where A_π has diagonal blocks with diagonal elements only
- Identify maximal independent sets of nodes (color classes), where neighbors in the adjacency graph have different colors
- Keep number of colors low

Multi-coloring algorithm

for $i = 1$ to N Set $\text{Color}(i)=0$; (where $N=\#\text{nodes}$)

for $i = 1$ to N Set $\text{Color}(i)=\min\{k > 0 : k \neq \text{Color}(j) \text{ for } j \in \text{Adj}(i)\}$

where $\text{Adj}(i) = \{j \neq i \mid a_{ij} \neq 0\}$ are the adjacents to node i .

Important Observation

Additive decomposition of splitting type methods maintains the re-arranged matrix pattern.

We have the inclusion

$$\mathcal{N}(L) \cup \mathcal{N}(D) \cup \mathcal{N}(R) \subseteq \mathcal{N}(A)$$

where \mathcal{N} denotes the non-zero matrix pattern

$$\mathcal{N}(A) := \{(i, j) \mid a_{ij} \neq 0, i, j = 1, \dots, N\}$$

for $A = (a_{ij})$.

Hence: Inversion of diagonal blocks D_i is an easy vector operation

Next idea: Extend the same approach to incomplete factorizations that often provide more efficient preconditioners.

Incomplete LU factorization

- Incomplete LU factorization

$$A = LU + R$$

where L is lower triangular, U is upper triangular, and R is the remainder matrix

- Do not allow fill-in elements (put them into R)
- Take preconditioner $M := LU$

Multi-coloring can be applied, since factorization maintains the non-zero pattern of A , i.e. we have $\mathcal{N}(L) \cup \mathcal{N}(U) \subseteq \mathcal{N}(A)$.

Parallel ILU(0) Preconditioners

Parallel ILU(0) without fill-in elements

- Perform multi-coloring permutation
- Write the problem in block-form with diagonal blocks with sizes $b_i \times b_i$ for $i = 1, \dots, B$, and B is the number of colors
- Obtain fine-grained parallelism based on matrix-vector multiplication routines

$$x_i := D_{Li}^{-1} \left(r_i - \sum_{j=1}^{i-1} L_{i,j} x_j \right) \text{ for } i = 1, \dots, B$$

$$z_i := D_{Ri}^{-1} \left(x_i - \sum_{j=1}^{B-i} R_{i,j} z_{i+j} \right) \text{ for } i = B, \dots, 1$$

D_{L1}	D_{R1}	R_{11}	R_{12}	R_{13}	
L_{21}	D_{L2}	D_{R2}	R_{21}	R_{22}	
L_{31}	L_{32}	D_{L3}	D_{R3}	R_{31}	
L_{41}	L_{41}	L_{41}	L_{41}	D_{L4}	D_{R4}

Inversion of D_{Li}^{-1} and D_{Ri}^{-1} is just a simple vector routine.

ILU(p) Preconditioners with Fill-in Elements

Extend approach to ILU(p) preconditioners with fill-in elements

- These are even more efficient preconditioners
- Maintain additional couplings by allowing fill-in elements controlled by up to p levels
- But: New sparsity pattern is unknown in advance
 - $\mathcal{N}(L) \cup \mathcal{N}(U) \supseteq \mathcal{N}(A)$
- Multi-coloring cannot be applied!

Alternative:

Level-scheduling (Y. Saad) may be applied

- Topological sorting of unknowns in order to resolve dependencies
- **But:** Large number of levels / too low degree of parallelism

Extend approach to ILU(p) preconditioners with fill-in elements

- These are even more efficient preconditioners
- Maintain additional couplings by allowing fill-in elements controlled by up to p levels
- But: New sparsity pattern is unknown in advance
 - $\mathcal{N}(L) \cup \mathcal{N}(U) \supseteq \mathcal{N}(A)$
- Multi-coloring cannot be applied!

Alternative:

Level-scheduling (Y. Saad) may be applied

- Topological sorting of unknowns in order to resolve dependencies
- **But:** Large number of levels / too low degree of parallelism

New Idea: Predetermine Pattern of $ILU(p)$

Observation

- We find that the non-zero pattern of $ILU(p)$ grows like $|A|^{p+1}$
- Idea: Apply multi-coloring to $|A|^{p+1}$, get permutation π
- Then we find $\mathcal{N}(\pi A \pi^{-1}) \subseteq \mathcal{N}(\pi |A|^{p+1} \pi^{-1})$
- Hence: if $\pi |A|^{p+1} \pi^{-1}$ has diagonal blocks on its diagonal, then so does $\pi A \pi^{-1}$

Power(q)-pattern enhanced multi-colored $ILU(p, q)$

$ILU(p)$ factorization with level- p fill-ins is performed over the pre-determined sparsity pattern of $|A|^q$ for $q = p + 1$.

- Use modified scheme for $ILU(p)$ decomposition

Then for L_p and U_p from $ILU(p, q)$, we have (by construction)

$$\mathcal{N}(L_p) \cup \mathcal{N}(U_p) \subseteq \mathcal{N}(|A|^{p+1})$$

New Idea: Predetermine Pattern of $ILU(p)$

Observation

- We find that the non-zero pattern of $ILU(p)$ grows like $|A|^{p+1}$
- Idea: Apply multi-coloring to $|A|^{p+1}$, get permutation π
- Then we find $\mathcal{N}(\pi A \pi^{-1}) \subseteq \mathcal{N}(\pi |A|^{p+1} \pi^{-1})$
- Hence: if $\pi |A|^{p+1} \pi^{-1}$ has diagonal blocks on its diagonal, then so does $\pi A \pi^{-1}$

Power(q)-pattern enhanced multi-colored $ILU(p, q)$

$ILU(p)$ factorization with level- p fill-ins is performed over the pre-determined sparsity pattern of $|A|^q$ for $q = p + 1$.

- Use modified scheme for $ILU(p)$ decomposition

Then for L_p and U_p from $ILU(p, q)$, we have (by construction)

$$\mathcal{N}(L_p) \cup \mathcal{N}(U_p) \subseteq \mathcal{N}(|A|^{p+1})$$

The Power(q)-Pattern Method

Power(q)-pattern enhanced multi-colored ILU(p, q)

- Perform multi-coloring analysis for $|A|^q$ with $q = p + 1$
 - Obtain corresponding permutation π
 - number of colors B and local block sizes b_i
- Permute $A_\pi := \pi A \pi^{-1}$
- Apply modified ILU(p) factorization to A_π , get L_p, U_p
- Solve $L_p U_p z = r$ by parallel forward/backward sweeps!!

Observations

- For $q = p + 1$ we obtain only diagonal elements in the diagonal blocks of L and U . No fill-ins here!!
- We can increase the degree of parallelism (with reduced number of colors) and decrease the quality of the preconditioner by drop-off techniques for $q < p + 1$

Modified ILU(p) scheme

Determine the sparsity pattern $\mathcal{N}(|A|^{p+1})$ of $|A|^{p+1}$

Set $\text{lev}(a_{ij}) = 0$ for all $a_{ij} \in \mathcal{N}(A)$, $\text{lev}(a_{ij}) = \infty$ otherwise

for $i = 2$ to N **do**

for $k = 1$ to $i - 1$ and $(i, k) \in \mathcal{N}(|A|^{p+1})$ with $\text{lev}(a_{ik}) \leq p$ **do**

$$a_{ik} = a_{ik} / a_{kk}$$

for $j = k + 1$ to N and $(i, j) \in \mathcal{N}(|A|^{p+1})$ **do**

$$a_{ij} = a_{ij} - a_{ik} a_{kj}$$

$$\text{lev}(a_{ij}) = \min(\text{lev}(a_{ij}), \text{lev}(a_{ik}) + \text{lev}(a_{kj}) + 1)$$

end for

end for

end for

- Multi-coloring can be performed in advance
- Short and compact loops, no dynamic data structures
- No fill-ins into diagonal blocks of A , i.e. parallel triang. sweeps

Test Configuration

- Solve $Ax = b$ with $b = 1$ and $x_0 = 0$
 - Krylov subspace solver: CG
 - Stopping criterion: 10^{-6} for relative residual

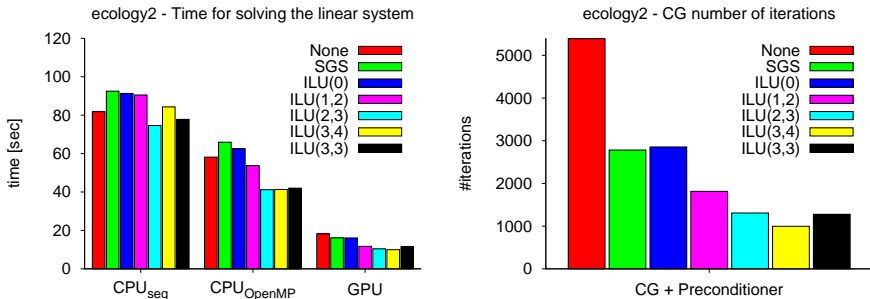
Name	Description of the problem	#rows	#non-zeros	Avg #nnz
s3dkq4m2	FEM - Cylindrical shells	90,449	4,820,891	53.3
G3 circuit	Circuit simulation	1,585,478	7,660,826	4.8
ecology2	Animal/gene movement	999,999	4,995,991	5.0

Table: Description and properties of three test matrices

- CPU: Intel i7-2600 Nehalem (4 cores + HT), 16GB memory
 - Sequential version
 - OpenMP parallel version
- GPU: NVIDIA GeForce GTX 580, 3GB memory
 - CUDA BLAS 1 (CuBLAS)
 - CUDA SpMV multiplication (with/without texture caching)

Eco: Solution Time and #Iterations

Solver times (left) and iteration counts (right)

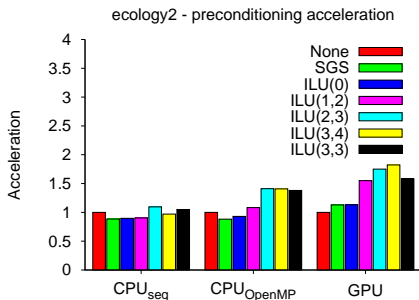
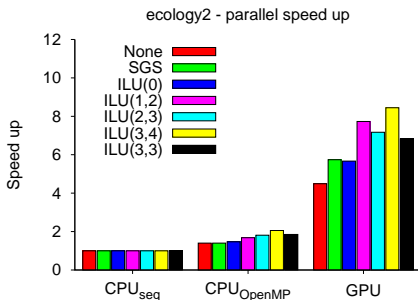


Name	None	SGS	ILU(0)	ILU(1,2)	ILU(2,3)	ILU(3,4)	ILU(3,3)
ecology2	5391	2783	2855	1815	1308	997	1277
acc. factor	1.0	1.93	1.88	2.90	4.12	5.40	4.22
#colors		2	2	7	8	19	8

Table: Number of iterations for solving the system with PCG

Eco: Speed-ups and Acceleration Factors

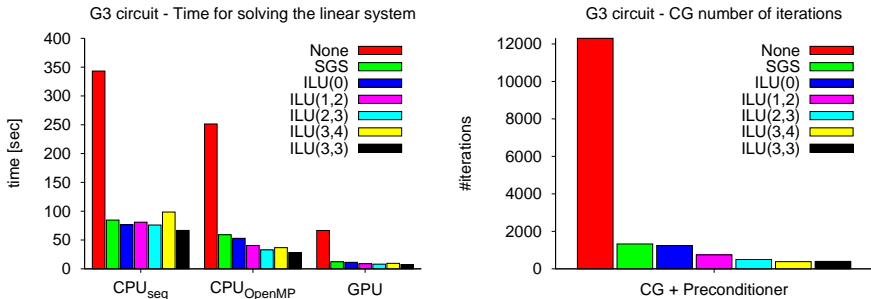
Parallel speed-up (left) and preconditioning acceleration (right)



- Good parallel scalability on the GPU
 - Saturation of the memory bandwidth on the CPU at 2 cores
- Only little preconditioner acceleration in this example
 - Up to 1.8x on the GPU

G3: Solution Time and #Iterations

Solver times (left) and iteration counts (right)

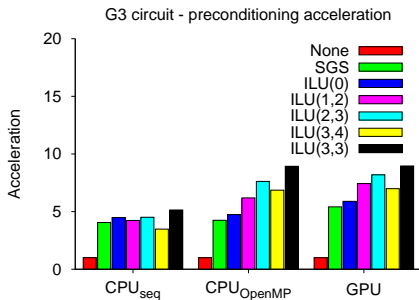
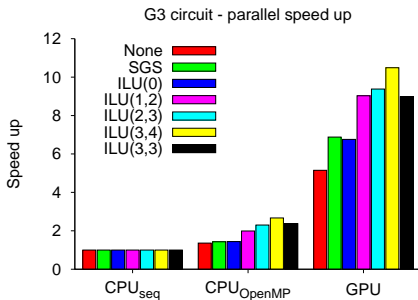


Name	None	SGS	ILU(0)	ILU(1,2)	ILU(2,3)	ILU(3,4)	ILU(3,3)
G3 circuit	12760	1328	1242	747	497	386	397
acc. factor	1.0	9.6	10.2	17.0	25.6	33.0	32.1
#colors		4	4	10	17	35	17

Table: Number of iterations for solving the system with PCG

G3: Speed-ups and Acceleration Factors

Parallel speed-up (left) and preconditioning acceleration (right)

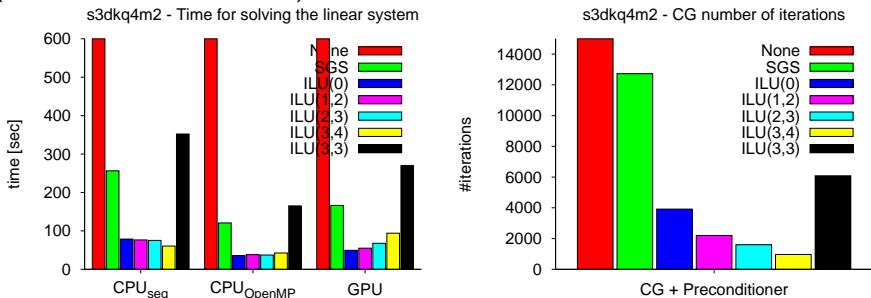


- Good scalability on CPU and GPU
- Parallel ILU(p,q) preconditioners give significant improvements on all systems (CPU and GPU)
 - Up to a factor of 8 on the CPU and GPU

S3: Solution Time and #Iterations

Solver times (left) and iteration counts (right)

(All red bars are cut off!)

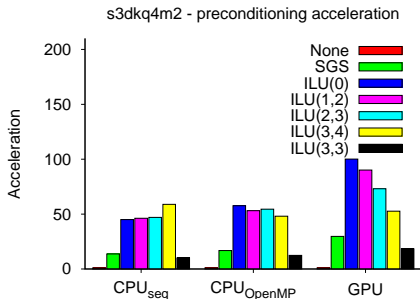
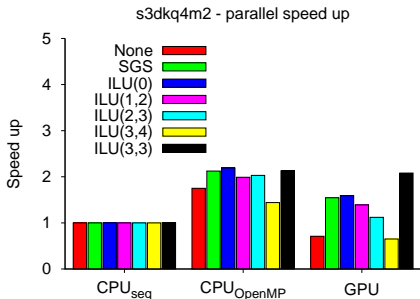


Name	None	SGS	ILU(0)	ILU(1,2)	ILU(2,3)	ILU(3,4)	ILU(3,3)
s3dkq4m2	535056	12728	3918	2203	1600	965	6086
acc. factor	1.0	42.0	136.5	242.8	334.4	554.4	87.9
#colors		24	24	56	96	150	96

Table: Number of iterations for solving the system with PCG

S3: Speed-ups and Acceleration Factors

Parallel speed-up (left) and preconditioning acceleration (right)

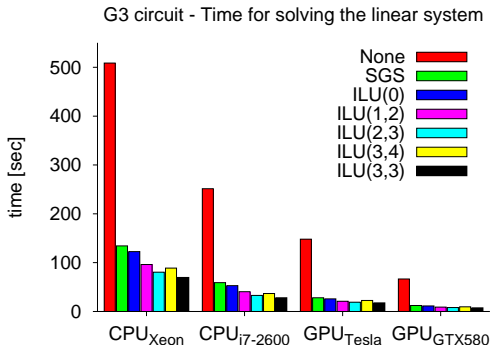


- Parallel speedup on GPU falls behind CPU
 - Small matrix (90449) with large number of colors (24-150)
- Significant improvement by the preconditioners (50x-100x)

Hardware comparison

Solver times for for G3 circuit matrix

- Dual socket x 4-core Intel Xeon E 5450
- Single socket x 4-core Intel i7-2600 Nehalem + HyperThreading
- NVIDIA Tesla S1070 (single GPU)
- NVIDIA Fermi GTX 580



Contact and Acknowledgements

Further Information

- jan-philipp.weiss@kit.edu
- <http://srg-multicore.math.kit.edu>
- <http://www.emcl.kit.edu>

Check our preprint at

- <http://www.emcl.kit.edu/preprints/emcl-preprint-2011-08.pdf>

Results in this preprint are on Tesla/Xeon not on Fermi/Nehalem

The work of the *Shared Research Group* is granted by Hewlett-Packard and the *Concept for the Future* of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative.

Thank you for your attention!

