Block AIR Methods

For Multicore and GPU

Per Christian Hansen
Hans Henrik B. Sørensen
Technical University of Denmark

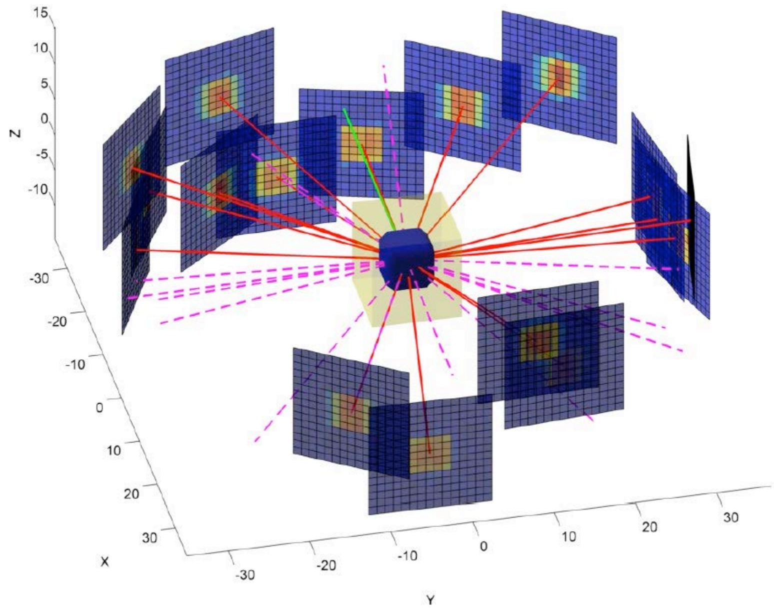
HDtomo
HIGH - DEFINITION TOMOGRAPHY

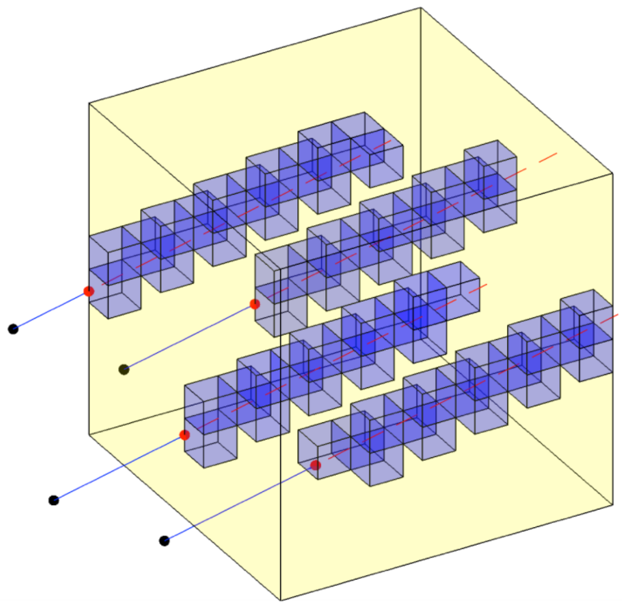
GPUlab
DTU Informatics

DTU Informatics
Department of Informatics and Mathematical Modeling

Model Problem and Notation

Parallel-beam 3D tomography





$$A\,x \simeq b, \qquad b = \bar{b} + \delta b, \qquad A \in \mathbb{R}^{m\times n}.$$

$\bar{x}$: exact solution

$\bar{b} = A\,\bar{x}$: exact data

δb : noise

$\|\bar{x} - x^k\|_2 / \|\bar{x}\|_2$: relative error

ART (Algebraic Reconstruction Technique)

Algorithm: ART (Classical Kaczmarz)

Initialization: choose an arbitrary $x^0 \in \mathbb{R}^n$

Iteration: for $k = 0, 1, 2, \ldots, \texttt{maxiter}$ or until convergence:

$$x^{k,0} = x^{k-1}$$

$$x^{k,i} = P_C\left(x^{k,i-1} + \lambda \, \frac{b_i - a_i^T \, x^{k,i-1}}{\|a_i\|_2^2} \, a_i\right), \qquad i = 1, \ldots, m$$

$$x^k = x^{k-1,m}$$

Characteristics

– Relaxation parameter $\lambda \in [0, 2]$

– Projection P_C

– Fast initial convergence.

– Parallelism at the level of an inner product

SIRT (Simultaneous Iter. Reconstr. Tech.)

Algorithm: SIRT

Initialization: choose an arbitrary $x^0 \in \mathbb{R}^n$, and two SPD matrices $M \in \mathbb{R}^{m \times m}$ and $T \in \mathbb{R}^{n \times n}$.

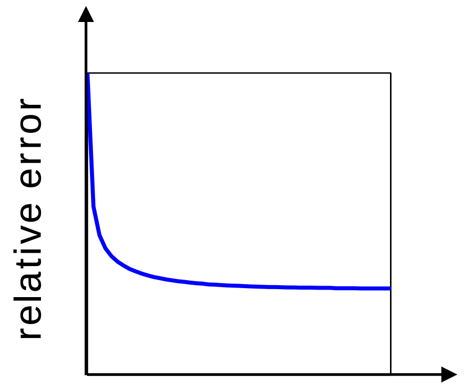
Iteration: for $k = 0, 1, 2, \ldots,$ `maxiter` or until convergence:

$$x^k = P_C\big(\, x^{k-1} + \lambda\, T A^T M\, (b - A\, x^{k-1})\,\big)$$

Characteristics

- Relaxation parameter $\lambda \in [\,0\,,\,2/\|A^T A\|_2\,]$
- Projection P_C
- Convergence + relaxation depends on T and M
- Slow initial convergence.
- Parallelism at the level of a matrix-vector product

Performance



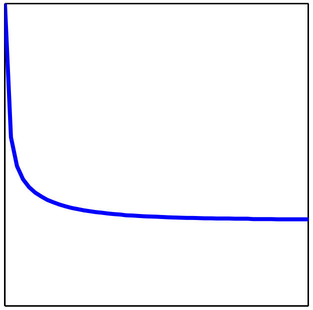
13 projections

$m \times n$	t/iter
$13 \cdot 128^2 \times 64^3$	0.08 s
$13 \cdot 256^2 \times 128^3$	0.93 s
$13 \cdot 512^2 \times 256^3$	10.8 s

Test Problem:

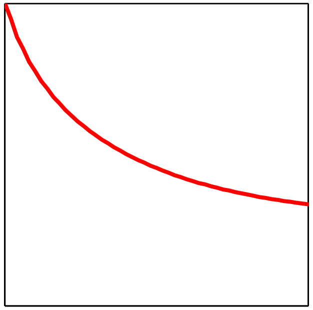
- parallel-beam tomography,
- 3D Shepp-Logan phantom, Schabel (2006).

Performance



$m \times n$	t/iter	t/iter
$13 \cdot 128^2 \times 64^3$	0.08 s	0.08 s
$13 \cdot 256^2 \times 128^3$	0.93 s	1.02 s
$13 \cdot 512^2 \times 256^3$	10.8 s	14.7 s

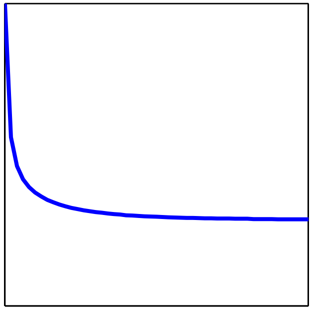
Intel Xeon E5620 2.40 GHz (1 core)



Same number of flops!

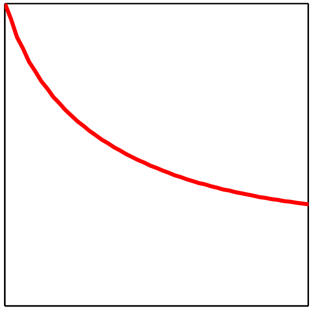
The difference is due to the cache: ART reuses row a_i immediately.

Performance



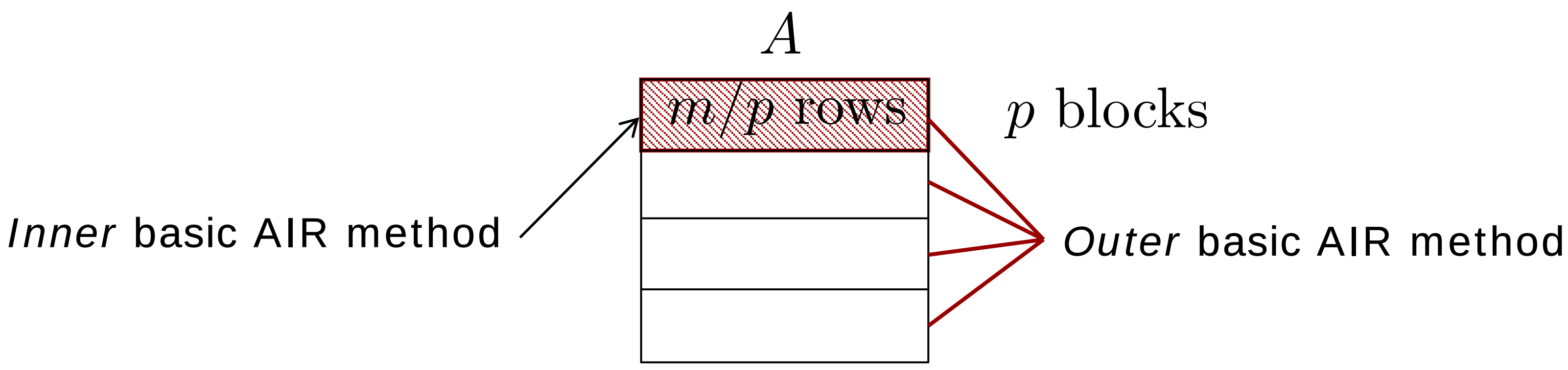
$m \times n$	t/iter	t/iter
$13 \cdot 128^2 \times 64^3$	0.08 s	0.04 s
$13 \cdot 256^2 \times 128^3$	0.93 s	0.41 s
$13 \cdot 512^2 \times 256^3$	10.8 s	4.12 s

Intel Xeon E5620 2.40 GHz (4 cores)



Block Methods

$$A = \begin{pmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \end{pmatrix}, \qquad A_\ell \in \mathbb{R}^{m_\ell \times n}, \quad \ell = 1, \ldots, p,$$



Parallelism given by the tradeoff: m/p rows vs. p blocks

Block-Sequential Method

Inner method = SIRT / outer method = ART

Algorithm: Block-Sequential

Initialization: choose an arbitrary $x^0 \in \mathbb{R}^n$

Iteration: for $k = 0, 1, 2, \ldots,$ `maxiter` or until convergence:

$$x^{k,0} = x^{k-1}$$

$$x^{k,\ell} = P_C\big(x^{k,\ell-1} + \lambda T\, A_\ell^T\, M_\ell\, (b_\ell - A_\ell\, x^{k,\ell-1}) \big), \quad \ell = 1, 2, \ldots, p$$

$$x^k = x^{k-1,p}$$

Eggermont, Herman & Lent (1981)

Characteristics

- Semi-convergence depends on *p*:
  - If $p = 1$, we recover SIRT
  - If $p = m$, we recover ART
- Parallelism at the level of a mat-vec product of size m/p

Block-Parallel method

Inner method = ART / outer method = SIRT

Algorithm: Block-Parallel

Initialization: choose an arbitrary $x^0 \in \mathbb{R}^n$

Iteration: for $k = 0, 1, 2, \ldots,$ `maxiter` or until convergence

for $\ell = 1, \ldots, p$ execute in parallel

$$y^{0,\ell} = x^{k-1}$$

$$y^{i,\ell} = y^{i-1,\ell} + \lambda^{\ell} \frac{b_i - a_i^T y^{i-1,l}}{\|a_i\|_2^2} a_i, \qquad i = 1, \ldots, m_\ell$$

$$x^{k,\ell} = y^{m_\ell,\ell}$$

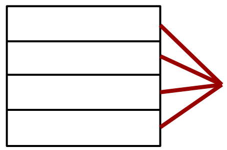
$$x^{k+1} = \sum_{\ell=1}^{p} D^{\ell} x^{k,\ell}$$

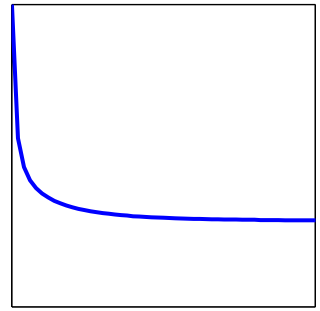
Characteristics

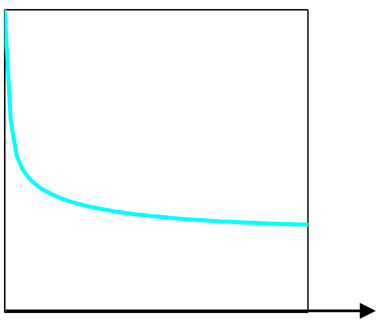
Gordon & Gordon (2005): CARP

- Semi-convergence depends on *p*:
  - ➢ If *p* = 1, we recover ART
  - ➢ If *p* = *m*, we recover SIRT
- Parallelism is coarse-grained: *p* blocks

Block Sequential

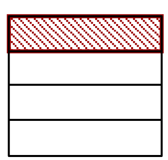


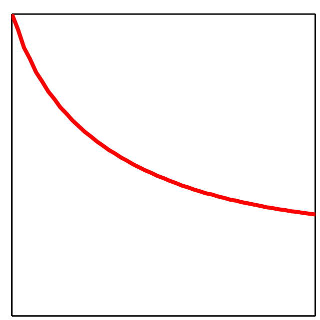




$m \times n$	t/iter	t/iter	t/iter
$13 \cdot 128^2 \times 64^3$	0.08 s	0.04 s	0.05 s
$13 \cdot 256^2 \times 128^3$	0.93 s	0.41 s	0.48 s
$13 \cdot 512^2 \times 256^3$	10.8 s	4.12 s	4.36 s

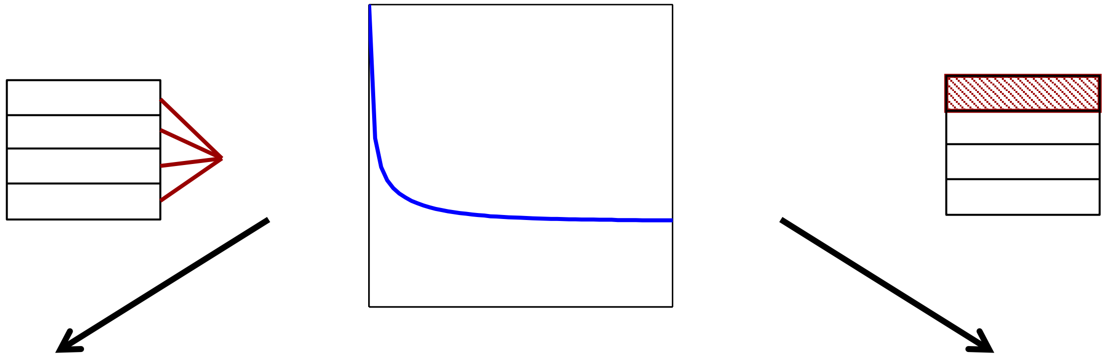
4 blocks

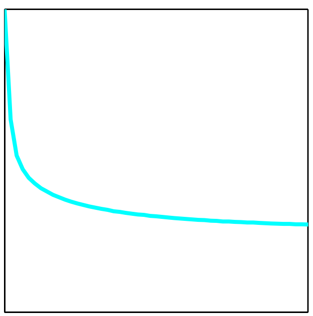




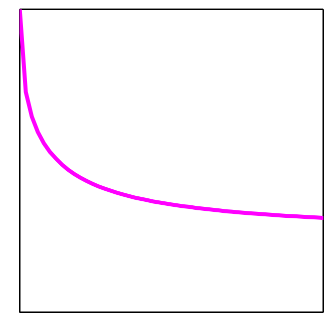
The convergence is close to that of ART, in spite of the fact that the computational "building blocks" are SIRT iterations (suited for multicore).

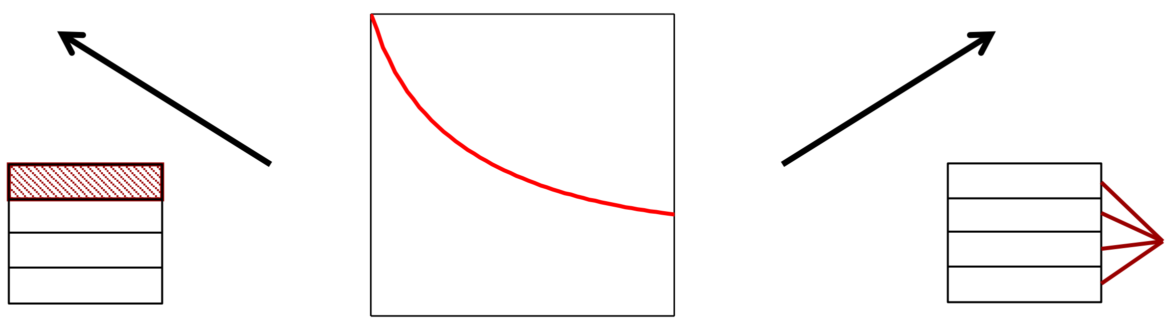
Block Parallel





$m \times n$	t/iter	t/iter	t/iter	t/iter
$13 \cdot 128^2 \times 64^3$	0.08 s	0.04 s	0.05 s	0.10 s
$13 \cdot 256^2 \times 128^3$	0.93 s	0.41 s	0.48 s	0.37 s
$13 \cdot 512^2 \times 256^3$	10.8 s	4.12 s	4.36 s	5.41 s





Fair Comparison of the Methods …

It is quite easy to make an unfair comparison between the different methods: choose a bad λ for the method you don't like.

To make a *fair* comparison between the methods, we must choose the value of λ that is (near) optimal for each method!

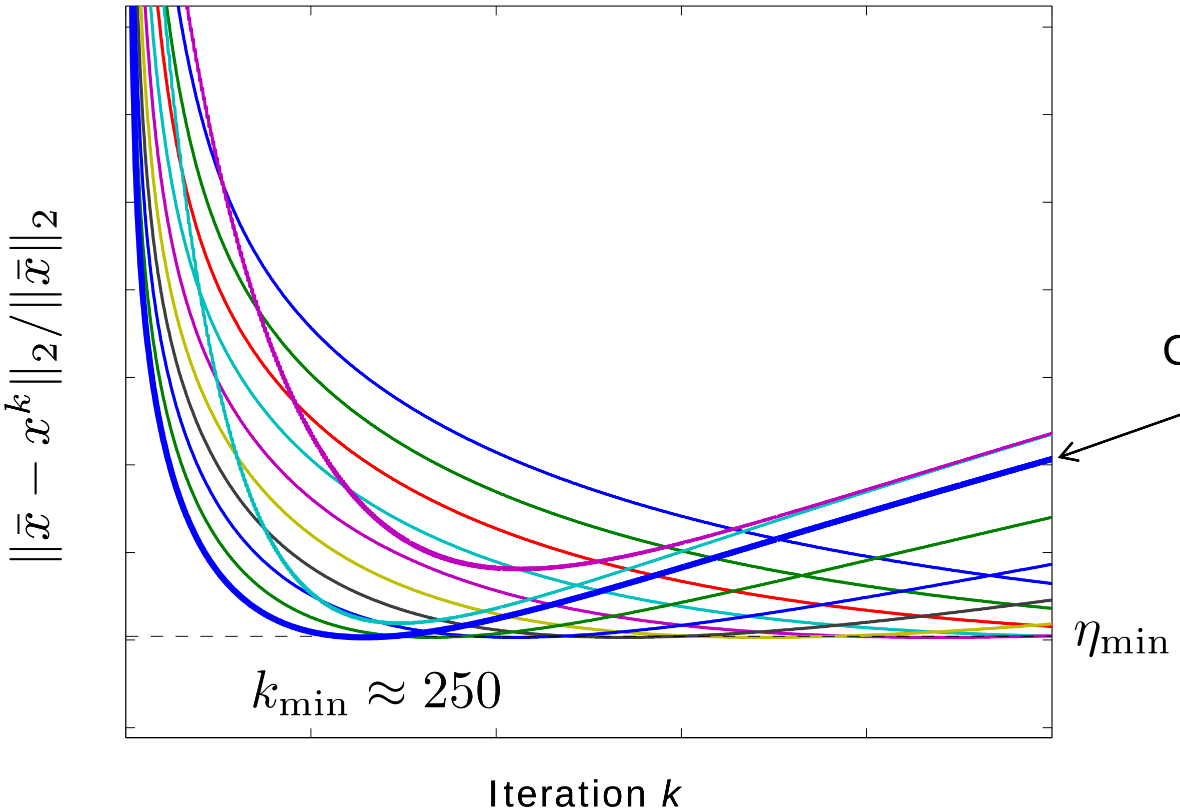
What do we mean by "(near) optimal"?

Use **training** (implemented in AIR Tools):

- Choose a test problem with a known solution, and which resembles the class of problems you need to solve.
- Find the parameter λ that gives fastes semi-convergence.

Training for Optimal λ

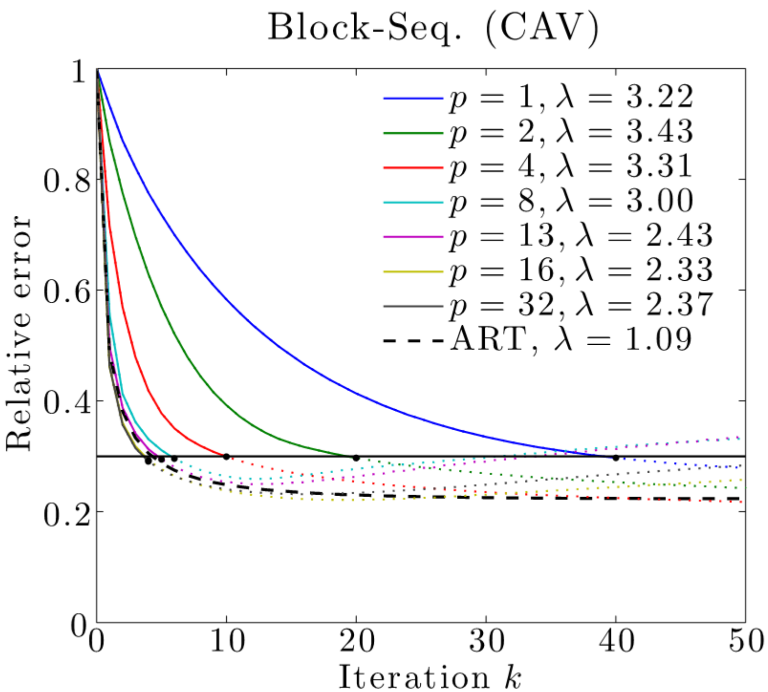
Semi-convergence and relaxation parameter λ

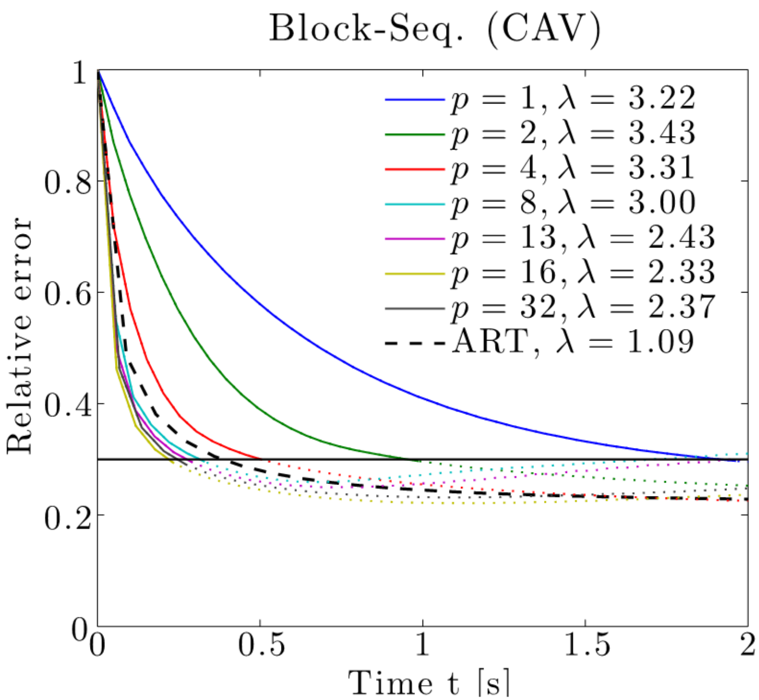


Optimal λ reaches min. error $\eta_{\min}$ in fewest iterations $k_{\min}$

Preliminary Results

$$m = 13 \times 128^2,\ n = 64^3$$





p	1	2	4	8	13	**16**	32	ART
k_{min}	40	20	10	6	5	**4**	4	5
t_{min}	1.96	0.99	0.51	0.33	0.29	**0.23**	0.27	0.45

The advantage of "block sequential" over standard ART is due to the improved use of the multicore architecture.

Typical GPU Hardware

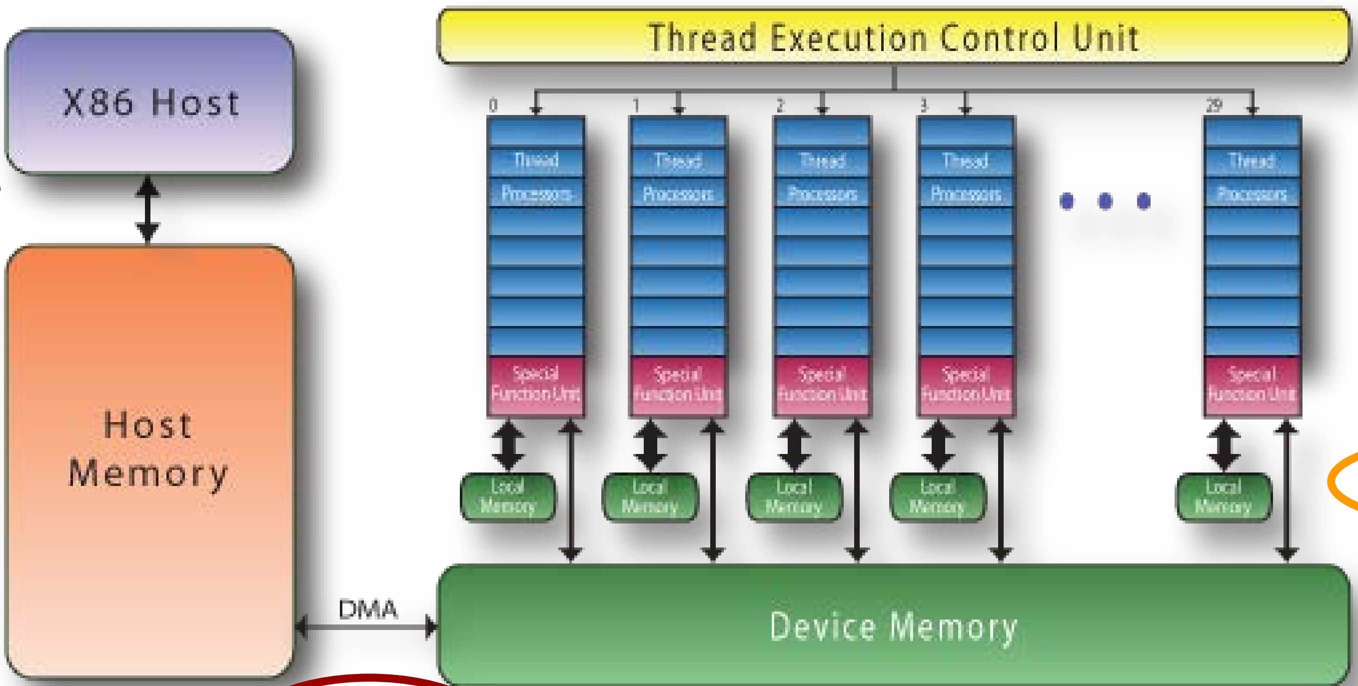
Host

Intel Xeon
4 cores
2.4 GHz
38 Gflop/s (DP)

Accelerator (GPU)

Nvidia C2050 “Fermi”
448 cores
1.15 GHz
515 Gflop/s (DP)

Front Side
6.4 Gb/s



Theoretical bandwidth
144 Gb/s

PCI Express
5.3 Gb/s

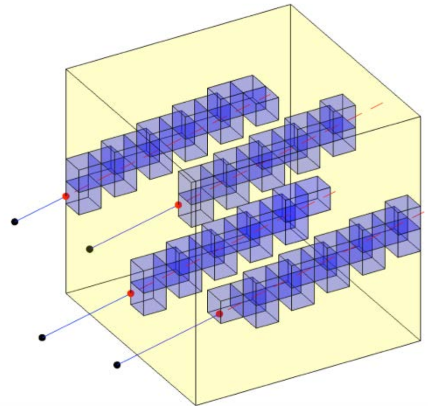
Towards a GPU Algorithm

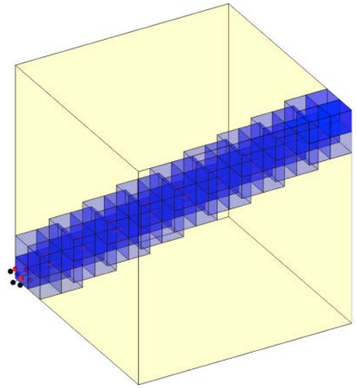
The best way to utilize the GPU is to give it tasks with very fine-grained parallelism.

Think of ”SIMD” – single instruction-stream multiple data-stream.

In *tomography*, it is easy to find sets of rows that are orthogonal due to the structure of zeros/nonzeros.

Thus, a re-ordering of the rows can produce blocks with mutually orthogonal rows.





Fine-Grained Parallelism

Consider a block A_ℓ whose rows are all *structurally orthogonal*, i.e., their nonzeros are located such that $a_i^T a_j$ for all $i \neq j$.

Now consider the sequential updates, for $i \neq j$:

$$\begin{aligned}\hat{x} &= x + \lambda \, \frac{b_i - a_i^T x}{\|a_i\|_2^2} \, a_i \\ \hat{\hat{x}} &= \hat{x} + \lambda \, \frac{b_j - a_j^T \hat{x}}{\|a_j\|_2^2} \, a_j\end{aligned}$$

Since there is no overlap between the locations of the nonzeros in a_i and a_j, we can compute the updates *in parallel*. If $\mathcal{I}$ and $\mathcal{J}$ denote the indices of the nonzeros in a_i and a_j, with $\mathcal{I} \cap \mathcal{J} = \emptyset$, we have:

$$\begin{aligned}\hat{x}(\mathcal{I}) &= x(\mathcal{I}) + \lambda \, \frac{b_i - a_i^T x}{\|a_i\|_2^2} \, a_i(\mathcal{I}) \\ \hat{x}(\mathcal{J}) &= x(\mathcal{J}) + \lambda \, \frac{b_j - a_j^T x}{\|a_j\|_2^2} \, a_j(\mathcal{J}).\end{aligned}$$

GPU-Block-Sequential Method

Inner method = ART-Orthogonal / outer method = ART

Algorithm: GPU-Block-Sequential

Initialization: choose an arbitrary $x^0 \in \mathbb{R}^n$

Iteration: for $k = 0, 1, 2, \ldots, \texttt{maxiter}$ or until convergence:

$$x^{k,0} = x^{k-1}$$

for $l = 1, \ldots, p$ execute sequentially

for $i = 1, \ldots, m_l$ execute in parallel

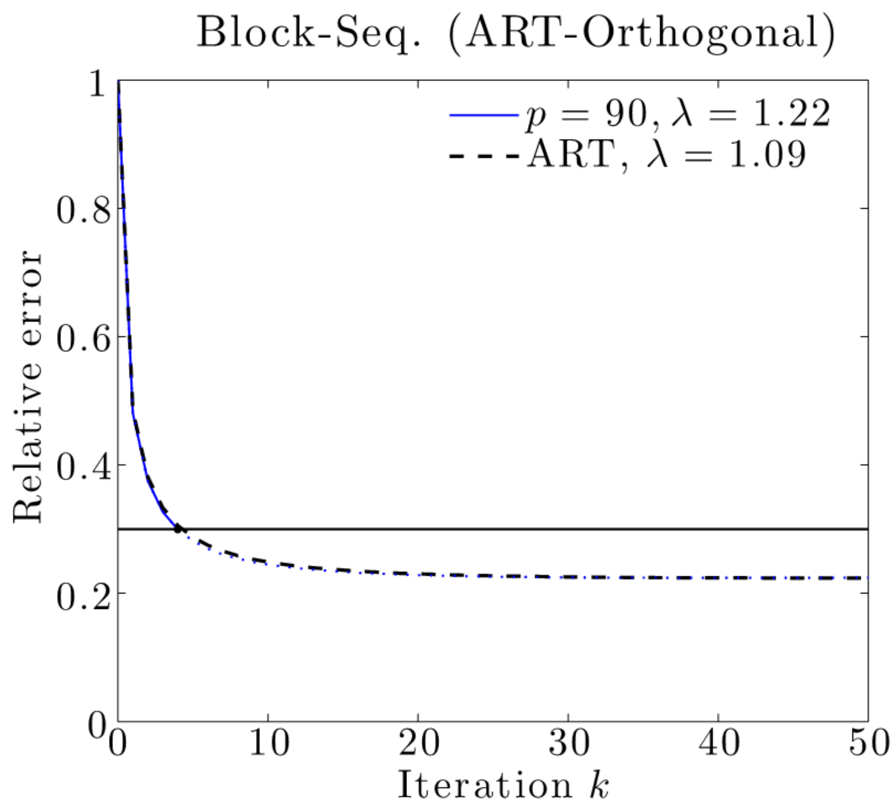
$$x^{k,l} = P_C\left(x^{k,l-1} + \lambda \frac{(b_l)_i - (A_l)_i^T x^{k,l-1}}{\|(A_l)_i\|_2^2} (A_l)_i\right)$$

$$x^k = x^{k-1,p}$$

Characteristics

- Convergence identical to ART.
- Here *p* is the number of blocks required for each block to have mutually orthogonal rows.
- Parallelism is fine-grained ≈ *m*/*p*.

Preliminary GPU Results



threads: the CPU has 4 cores, but hyper-threading is allowed

threads	1	2	4	8	16	**GPU**	ART
t/iter	0.0961	0.0629	0.0475	0.0429	0.0517	**0.0484**	0.0850

The limiting factor is the CPU-GPU bandwidth, because blocks of A are moved to the GPU in each iteration.

Conclusions

Multicore

- Block-sequential methods are able to achieve convergence similar to that of ART (error reduction per iteration),
- and with *smaller computing time* because we can utilize the multicore architecture.

GPU

- With a suitable row ordering and choice of blocks, we can utilize the fine-grained parallelism of GPUs.
- Next step: generate the matrix A on the GPU (don't move it).