High Performance Medical Image Reconstruction. What could go wrong?

Christopher Kumar Anand
Andrew Curtis
Michal Dobrogost
Maryam Moghadas
Jessica Pavlin
Yuriy Toporovskyy
Wolfram Kahl
What can go wrong?

- algorithm selection and parameterization
  - non-convex objective
  - non-convergence (ill-conditioned)
  - insufficient signal versus noise
- implementation
  - transcription errors
  - non-deterministic parallelization
  - model does not capture physics (including units)
COCONUT
COde CONstructing User Tool

• **DSLs** embedded in Haskell
  • **strong** types (detect errors in model and code)

• **high** level
  • start with objective function
  • differentiate symbolically
  • simplify expressions
  • recognize recurrence relations
  • generate code

• verification
  • SIMD & distributed parallelization (private and shared-memory)
  • *linear-time* verification via **AVOp model**

• algorithm selection and experiment design
  • homotope models to maintain convexity
  • optimize SNR and bound maximum noise (**SSV metric**)
  • scale regularization to ensure required convergence
Models

define variables

```haskell
> let x = var3d (16,16,16) "x"
> let y = var3d (16,16,16) "y"
> ft (x :+: y)
(FT((x(16,16,16)+:y(16,16,16))))
```

define an objective function to minimize

$$\| ft(x + iy) \|_2$$

and take it’s derivative

```haskell
>> diff (mp ["X"] ) (norm2 (ft (x :+: y)))
(((Re(FT((d(x[16] [16] [16])+0.0[16,16,16])))))(Re(FT((x[16] [16] [16]+:y[16] [16] [16]) ))))
+ (((Re(FT((d(x[16] [16] [16])+0.0[16,16,16])))))(Re(FT((x[16] [16] [16]+:y[16] [16] [16]) )))))
+ (((Im(FT((d(x[16] [16] [16])+0.0[16,16,16])))))(Im(FT((x[16] [16] [16]+:y[16] [16] [16]) )))))
+ (((Im(FT((d(x[16] [16] [16])+0.0[16,16,16])))))(Im(FT((x[16] [16] [16]+:y[16] [16] [16]) )))))
+ (((Im(FT((d(x[16] [16] [16])+0.0[16,16,16])))))(Im(FT((x[16] [16] [16]+:y[16] [16] [16]) )))))
```
more complicated models

• e.g., conservation of mass in material transport models is a function

\[
\text{massConservation (ThreeD } \text{vx, ThreeD } \text{vy, ThreeD } \text{vz)} \\
= \text{norm2 ( conv3Zip1ZM com vx vy vz )}
\]

where

\[
\text{com (vX, vY, vZ) = (vX[1,0,0] - vX[-1,0,0])} \\
+ (vY[0,1,0] - vY[0,-1,0]) \\
+ (vZ[0,0,1] - vZ[0,0,-1])
\]
what was that $d(X)$, and where's $\nabla f$?

- $d(X)$ is a vector of differential forms
- comes from *implicit* derivative
- we can extract $\nabla f$ by simplifying
simplify rules

- commute $d(x)$ to LHS of dot
- move linear operators to RHS via adjoint

\[
\begin{align*}
\text{Left} & \uparrow \\
\Re & \\
\mathfrak{f}t & \\
(+i0) & \\
\text{dx} & \\
\hline
\text{Right} & \downarrow \\
\Re & \\
\mathfrak{f}t & \\
(+i0) & \\
x & \\
\hline
\text{Left} & \uparrow \rightarrow \\
\mathfrak{f}t^{-1} & \\
(+i0) & \\
\Re & \\
\mathfrak{f}t & \\
(+i0) & \\
x & \\
\hline
\text{Right} & \downarrow \rightarrow \\
\Re & \\
\mathfrak{f}t^{-1} & \\
(+i0) & \\
\Re & \\
\mathfrak{f}t & \\
(+i0) & \\
x & \\
\end{align*}
\]
simplification DSL

\[ x * (y * z) \Rightarrow (x * y) * z \]
\[ 0 * x \Rightarrow 0 \]
\[ 0 * x \Rightarrow 0 \]
\[ \text{ft} \ (\text{invFt} \ z) \Rightarrow z \]

etc.

• natural algebraic simplification rules
• distinguishes scaling and multiplication
Type Checking (|| work)

• with stronger typing, compilers find more mistakes
• in C, all dynamic arrays (float*) look alike
• others check size and dimension
• we can do better
Arrays of Samples
Arrays of Samples

• number of samples
• dimension
• frame of reference
• resolution (including units)
• units of measurement
Formalization

- in Haskell types define
  - physical units
  - array size and dimension
  - frame of reference

```haskell
canalSample2 :: Discretization1D (F "CanalFrame")
(NAT 12)
[float | 0.02 | ]
Meter
[Double]
```
array sizes match, so try to add them …
Type Inference Across Linear Operations

- e.g., Fourier Transforms easy to break
- Nyquist Sampling Theorem, etc.
- frame of reference
- resolution (including units)
- units of measurement
Classy Proofs

• Properties are Classes

class FT a b | a → b, b → a where
  ft :: a → b
  invFt :: b → a

class MultD3 f2 f1 f0 e2 e1 e0 g2 g1 g0 |
  f2 f1 f0 e2 e1 e0 → g2,
  f2 f1 f0 e2 e1 e0 → g1,
  f2 f1 f0 e2 e1 e0 → g0 where
Proofs are Instances

instance (  
    AssertDualFrames frame1 frame2, Frame frame1, Frame frame2,  
    IsFloat stepSize2, IsFloat stepSize1, ToFloat numSamp ~ numSampF,  
)

and encode the Nyquist criterion:

MultNZ stepSize1 numSampF t0,  
MultNZ t0 stepSize2 t1,  
t1 ~ FLOAT Pos 1 (E 0)

⇒  
FT (Discretization1D frame1 numSamp stepSize1 rangeU [Complex Double])  
(Discretization1D frame2 numSamp stepSize2 rangeU [Complex Double])

where

ft (Discretization1D x) = (Discretization1D $ FFT fft x)  
invFt (Discretization1D x) = (Discretization1D $ FFT ifft x)

instance (  
    Times f0 e0 p00h p00l, Times f1 e0 p10h p10l,  
    Times f2 e0 D0 p20l, Times f0 e1 p01h p01l,  
    Times f1 e1 D0 p11l, Times f2 e1 D0 D0,  
    Times f0 e2 D0 p02l, Times f1 e2 D0 D0,  
    Times f2 e2 D0 D0,  
    Add3 p00h p10l p01l c1h c1l,  
    Add5 p20l p01h p11l p02l c1h D0 c2l)  
⇒ MultD3 f2 f1 f0 e2 e1 e0 c2l c1l p00l where
Readable Errors

- type inference fails on modelling errors
- ghc *loves* to throw up thousand-line errors
- we tamed it

No `instance` for `(DualUnits
  (SIUnit ('M 1) ('S 0) ('Kg 0) ('A 0) ('mol 0) ('K 0))
  (SIUnit ('M 0) ('S 0) ('Kg 0) ('A 0) ('mol 0) ('K 0)))`
## Multi-Core = ILP Reinvented

<table>
<thead>
<tr>
<th>Instruction Level Parallelism</th>
<th>Multi-Core Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Chip</td>
</tr>
<tr>
<td>Execution Unit</td>
<td>Core</td>
</tr>
<tr>
<td>Load/Store Instruction</td>
<td>DMA</td>
</tr>
<tr>
<td>Arithmetic Instruction</td>
<td>Computational Kernel</td>
</tr>
<tr>
<td>Register</td>
<td>Buffer / Signal</td>
</tr>
</tbody>
</table>
The Catch: Soundness

- on CPUs hardware maintains OOE
  - instructions execute out of order
  - hardware hides this from software
    - ensures order independence
- in our Multi-Core virtual CPU
  - compiler inserts synchronization
    - soundness up to software
    - uses asynchronous communication
Asynchronous

- no locks
- locking is a multi-way operation
  - a lock is only local to one core
    - incurs long, unpredictable delays
- use asynchronous messages
  - matches efficient hardware
Async Signals

No writes to buffer until DMA completion is confirmed

No reads or writes to buffer until past barrier WaitData

SendSignal

WaitData

WaitDMA

WaitSignal

SendData
# Multi-Core Language

**AVOps**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computation</strong></td>
<td>do a computation with local data</td>
</tr>
<tr>
<td><strong>SendData</strong></td>
<td>start DMA to send local data off core</td>
</tr>
<tr>
<td><strong>WaitData</strong></td>
<td>wait for arrival of DMAed data</td>
</tr>
<tr>
<td><strong>WaitDMA</strong></td>
<td>wait for locally controlled DMA to complete</td>
</tr>
<tr>
<td><strong>SendSignal</strong></td>
<td>send a signal to distant core</td>
</tr>
<tr>
<td><strong>WaitSignal</strong></td>
<td>wait for signal to arrive</td>
</tr>
<tr>
<td><strong>Loop</strong></td>
<td>body; $\pi$(body); $\pi(\pi$(body))...</td>
</tr>
</tbody>
</table>

- **Computation** *operation* bufferList
- **SendData** *localBuffer* remoteBuffer tags
- **WaitData** *localBuffer* tag
- **WaitDMA** *tag*
- **SendSignal** *core* signal
- **WaitSignal** signal
**locally Sequential Program**

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>SendSignal $s \to c2$</td>
<td>WaitSignal $s$ computation</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>SendSignal $s \to c2$</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
</tbody>
</table>

- total order for instructions
- easier to think in order
- send precedes wait(s)
**NOT sequential**

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SendSignal $s \rightarrow c2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>SendSignal $s \rightarrow c2$</td>
</tr>
<tr>
<td></td>
<td><em>second signal overlaps the first, only one registered</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>computation</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
</tbody>
</table>

- can execute out of order
does NOT imply order independent

<table>
<thead>
<tr>
<th>index</th>
<th>core 1</th>
<th>core 2</th>
<th>core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>long computation</td>
<td>SendSignal $s \rightarrow c2$</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SendSignal $s \rightarrow c2$</td>
<td>computation using wrong assumptions</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>WaitSignal $s$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Linear-Time Verification

• must show
• results are independent of execution order
• no deadlocks
• need to keep track of all possible states
• linear in time = one-pass verifier
• constant space
  • = max possible states at each instruction
Impact

- no parallel debugging !!
- every optimization trick used for ILP can be adapted
But...

- assumes you have signals
- what about shared memory?
- still lock-free synchronization?
- let’s try it on x86
Single-Reader, Single-Writer AVOp Ring Buffers (SRSWARB)

AVOp Stream

... Core 4: c_{63} = a_{63} * b_{63} Core 3: c_{47} = a_{47} * b_{47} Core 2: c_{31} = a_{31} * b_{31} Core 1: c_{15} = a_{15} * b_{15} Core 0: c_{0} = a_{0} * b_{0} Core 4: z_{63} = x_{63} + y_{63} Core 3: z_{47} = x_{47} + y_{47} Core 2: z_{31} = x_{31} + y_{31} Core 1: z_{15} = x_{15} + y_{15} Core 0: z_{0} = x_{0} + y_{0}
Limit Hazards

- AVOps can only conflict if they can fit in the Ring Buffers at the same time
- on each core, AVOps are sequential, therefore safe
- reads on different cores are safe
- while write AVOp is on a core, check that no other core reads or writes
Nest Step

• signals across nodes

• SRSWARB system for multicore

• new system for GPU
Details ...


Jessica L M Pavlin and Christopher Kumar Anand, Symbolic Generation of Parallel Solvers for Inverse Imaging Problems, CAS-14-05-CA.

Maryam Moghadas, Yuriyy Toporovskyy, Christopher Kumar Anand, Type-Safety for Inverse Imaging Problems, CAS-14-04-CA.
It Works!

• used to generate, from the objective function, a multi-core (shared memory) image reconstruction software for parallel Magnetic Resonance Imaging for AllTech Medical Systems America
MRI

- Image values & tissue contrast depends on how the experiment is conducted

- Qualitative and quantitative
Quantum Mechanical Foundation of MRI

slide contents tunnelled away
Signal and Contrast

- Sample magnetization $M$ - vector
  \[ M \]
  
  $B_0$
  
  Evolves in time $\rightarrow$ Bloch Equation
  \[
  \frac{dM}{dt} = M \times B_0
  \]

- Controllable by application of RF and magnetic fields:
  \[
  \frac{d}{dt} \begin{pmatrix} M_x \\ M_y \\ M_z \end{pmatrix} = \begin{pmatrix} -R_2 & -\Omega^0 & bi \\ \Omega^0 & -R_2 & -br \\ -bi & br & -R_1 \end{pmatrix} \begin{pmatrix} M_x \\ M_y \\ M_z \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ R_1 \times M_e \end{pmatrix}
  \]
Controls and Data

Controls

- big uniform magnet
- medium linear electro-magnets
- radio-frequency coils

rf in

rf out

data

computer talks to nuclei
MRI Experiment
MRI Experiment

Create Signal → Generate Contrast → Acquire Data

Reconstruct Images → Postprocess
Data Acquisition

• Unlike x-ray/CT, PET, or Microscopy, we don’t capture projections or reflections of the incident waves.
K-Space

- Via some Nobel prize winning work:
- Modulate phase of magnetization in space & measure DFT. — frequency domain
Each sample point gives spatial information.
K-space properties

- Nyquist, my friend. Sample rate $\sim$ kspace spacing

$\Delta k = \frac{1}{\text{FOV}}$  \hspace{1cm}  $\Delta w = \frac{1}{k_{\text{FOV}}}$
Receiver Elements

- In addition to phase modulation for k-space sampling, data is acquired via multiple receivers
  - spatial coverage
  - improved signal
  - spatial encoding
Parallel Imaging By Example

Original formulation: SENSE
Parallel Imaging by Example

• How do we recover the underlying magnetization?

• Recall: multiple receivers

• Consider two overlapping voxels and receivers a & b

• Signal from a and b ~

\[ s_a = c_a \cdot m_1 + c_a \cdot m_2 \]
\[ s_b = c_b \cdot m_1 + c_b \cdot m_2 \]

• Only works for nice aliasing
Forward Problem

• Extending the previous example, more general image acquisition process:

  • measurement =
    k-encoding * FT * receive sensitivity * magnetization

  \[ M_i = k \cdot FT \cdot S_i \cdot \rho_0, \quad i = 1 \ldots nC \]
Inverse Problem?

- Reduction factor of 4: 
  Vastly different practical performance thanks to receiver profiles.

- How do we measure which will be better before running the MRI? Expected reconstruction error!
Geometry-factor

- Gold standard metric for assessing effect of under sampling patterns on the resultant image

- Estimates difficulty of un-aliasing **pairs**/sets of spatial overlapping voxels

- n values. What is important? mean? max? 95%le?

Preussman, MRM 2001
Dense aliasing

• For non-standard grids the aliasing patterns and G-factor are complicated, because signal can alias anywhere

• No simple DFT relation
Problem with Gold

- takes too long to mine (hours or days)
- bad for environment
New Approach

• Estimate largest and smallest magnitude singular values of $M$
  \[ M_i = k \cdot FT \cdot S_i \cdot \rho_0, \quad i = 1 \ldots nC \]

• Physical interpretations

• Smallest singular value: **SSV Metric**
  — worst case noise amplification from reconstruction

• Largest singular value:
  — maximum of 1 with proper scaling.

• Large/small ~ condition #
SSV Metric Method

• M is rectangular!
  Form normal system $M^HM$ and find min/max eigenvalues $\lambda_{min} = \sigma_{min}^2$

$$M^HM = \sum_{i}^{N_{coils}} C_i^H \cdot iFT \cdot k^H \cdot k \cdot FT \cdot C_i$$

• $k^Hk$ is really just zeroing of non-sampled locations

• Might affect convergence if SSV is small already

• Can include regularization: $(M^HM + R)\nu = \lambda\nu$

• ARPACK + FFTW
Assessing SSV

• Compute slow G-factor metrics, L2 error, and SSV for a family of well know sampling grids
Good as Gold

- Compute slow G-factor metrics, L2 error, and SSV for a family of well know sampling grids

- Get same ranking in a few seconds

<table>
<thead>
<tr>
<th></th>
<th>( \delta^{*} )</th>
<th>( \delta^{*} )</th>
<th>( \delta^{*} )</th>
<th>( \delta^{*} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gamma_1 )</td>
<td>0.94 ***</td>
<td>0.97 ***</td>
<td>0.93 ***</td>
<td>-0.032</td>
</tr>
<tr>
<td>( \gamma_2 )</td>
<td>0.95 ***</td>
<td>0.96 ***</td>
<td>-0.032</td>
<td></td>
</tr>
<tr>
<td>( \gamma_3 )</td>
<td>0.97 ***</td>
<td>-0.014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \gamma_4 )</td>
<td>( g_{\text{mean}} )</td>
<td>-0.074</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>( \delta^{*} )</th>
<th>( \delta^{*} )</th>
<th>( \delta^{*} )</th>
<th>( \delta^{*} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \gamma_1 )</td>
<td>0.84 ***</td>
<td>0.87 ***</td>
<td>0.86 ***</td>
<td>-0.4</td>
</tr>
<tr>
<td>( \gamma_2 )</td>
<td>0.96 ***</td>
<td>0.96 ***</td>
<td>1 ***</td>
<td>-0.46</td>
</tr>
<tr>
<td>( \gamma_3 )</td>
<td>( g_{\text{mean}} )</td>
<td>1 ***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \gamma_4 )</td>
<td>( g_{\text{mean}} )</td>
<td>-0.41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SSV Metric Results

- **SSV** ranks candidate sampling patterns very similarly to gold standard metrics.

- Orders of magnitude faster ~ seconds-minutes.

- Doesn’t require image values, just **receivers** and **pattern**.

- Trade off: loss of spatial information that G-factor and simulation L2 error provides.

- Opens door for optimization!
App 1: How Random?

- Combinatorial nightmare

- Random sampling $\rightarrow$ incoherent aliasing “noise”
  $\rightarrow$ CS techniques

![Uniform Random](image1)

![Poisson Disk](image2)

Uniform Random $\quad$ R=9 $\quad$ Poisson Disk
App 2: Why low k-space?

- folklore: always sample centre
- more “energy” at centre of k-space
- low res is easy—> compact coil support, so sample more at high k

- Test: Generate patterns with varying density at centre
Results:

- Centre sampling predominantly affects Largest singular value
SSV

• **SSV** is a *fast* metric for experiment designs
  
  • demonstrated on:
    1) Assessing Random patterns
    2) Assessing effects of densely sampling k0

• fast enough to use to optimize experiment design
Coconut Project

- common sources of error fixable
  - transcription of model
  - wrong frame of reference
  - Heisenbugs
  - lots of little bugs left
- can effectively estimate SNR
  - use this to ensure statistical validity
Thanks

Stephen Adams
Curtis d’Alves
Kevin Browne
Shiqi Cao
Nathan Cumpson
Saeed Jahed
Damith Karunaratne
Clayton Goes

Gabriel Grant
William Hua
Fletcher Johnson
Wei Li
Nick Mansfield
Maryam Moghadas
Mehrdad Mozafari
Adam Schulz

Anuroop Sharma
Sanvesh Srivastava
Wolfgang Thaller
Gordon Uszkay
Christopher Venantius
Paul Vrbik
Fei Zhao

Robert Enenkel

IBM Centre for Advanced Studies, CFI, OIT, MITACS, NSERC, OCA Inc. and Apple Canada for research support.
Bonus: 3D sampling

- We said lines of k-space are “free”
- In 2D: we choose which lines to sample
- In 3D, we have a 2D transverse plane of possible locations