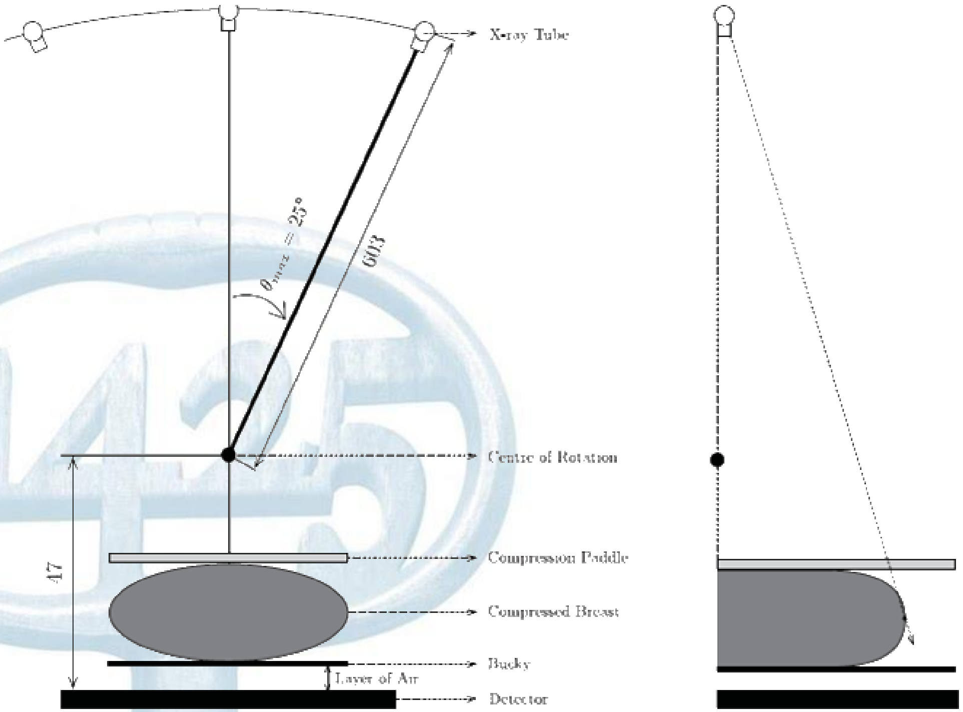
KATHOLIEKE UNIVERSITEIT
LEUVEN

Parallel data processing on GPU and CPU using OpenCL

Koen Michielsen
Dept. of Imaging and Pathology &
Medical Imaging Research Center,
KU Leuven, Belgium

Digital Breast Tomosynthesis

- Limited angle tomography: (depending on the vendor)
  - 11 to 25 exposures
  - Angular range: 15 to 50 degrees



Digital Breast Tomosynthesis

- Data (sinogram) size:
  3584 x 2816 x 25 angles x 16 bit ≈ 500 MB

- Typical image (reconstruction) size:
  3584 x 2816 x 45 x 32 bit ≈ 1.7 GB

- Example study: 12 patients, ± lesion, ± scatter
  → 48 reconstruction (10 iterations of MLTR)

  → Originally: Intel Xeon E5440 @ 2.8 GHz, 1 thread:

  24h / reconstruction

Projection / Backprojection

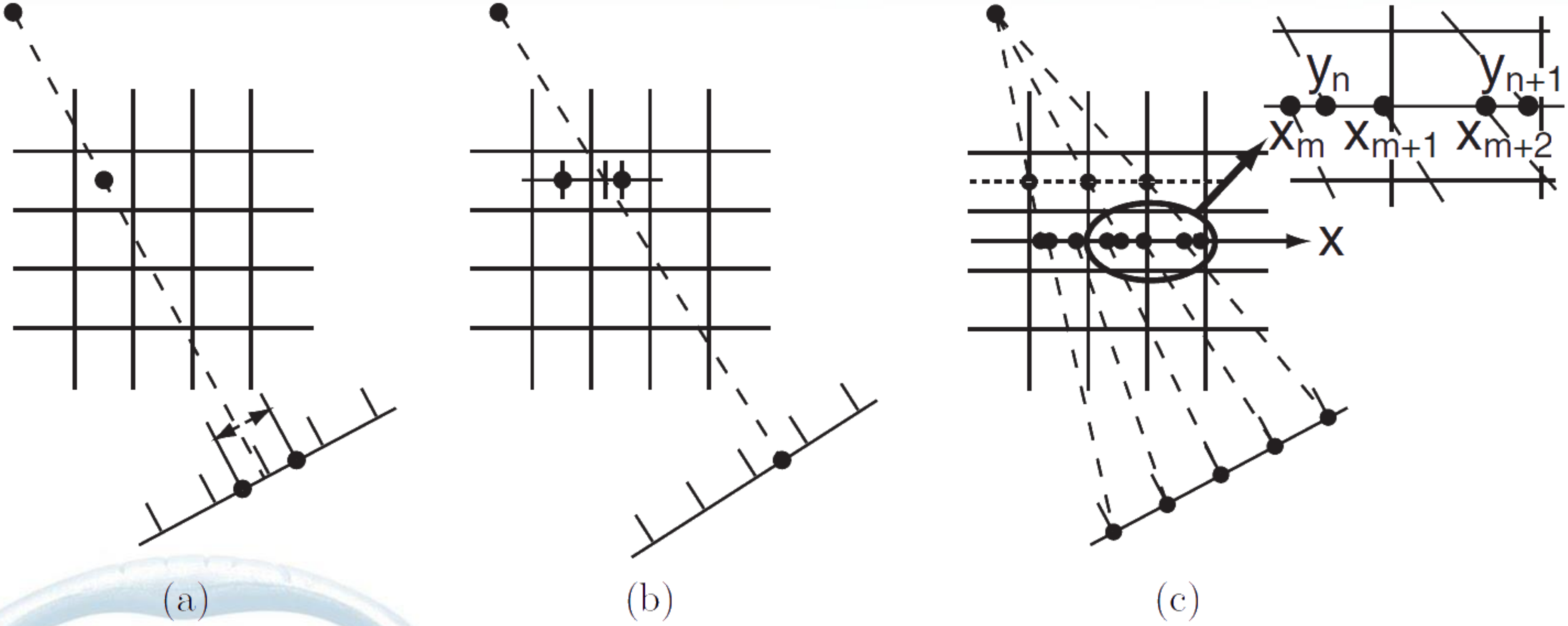
- MLTR update step:

$$\Delta\mu_j = \frac{\sum_i l_{ij}(\hat{y}_i - y_i)}{\sum_i l_{ij}(\sum_k l_{ik})\hat{y}_i}$$

- Main computational bottleneck: l_{ij}

• Sinogram elements	N = 2.5 * 10^8
• Reconstruction elements	M = 4.5 * 10^8
• Projection matrix elements	M*N = 1.1 * 10^{17} (~ $10^{11} \neq 0$)

Projection / Backprojection



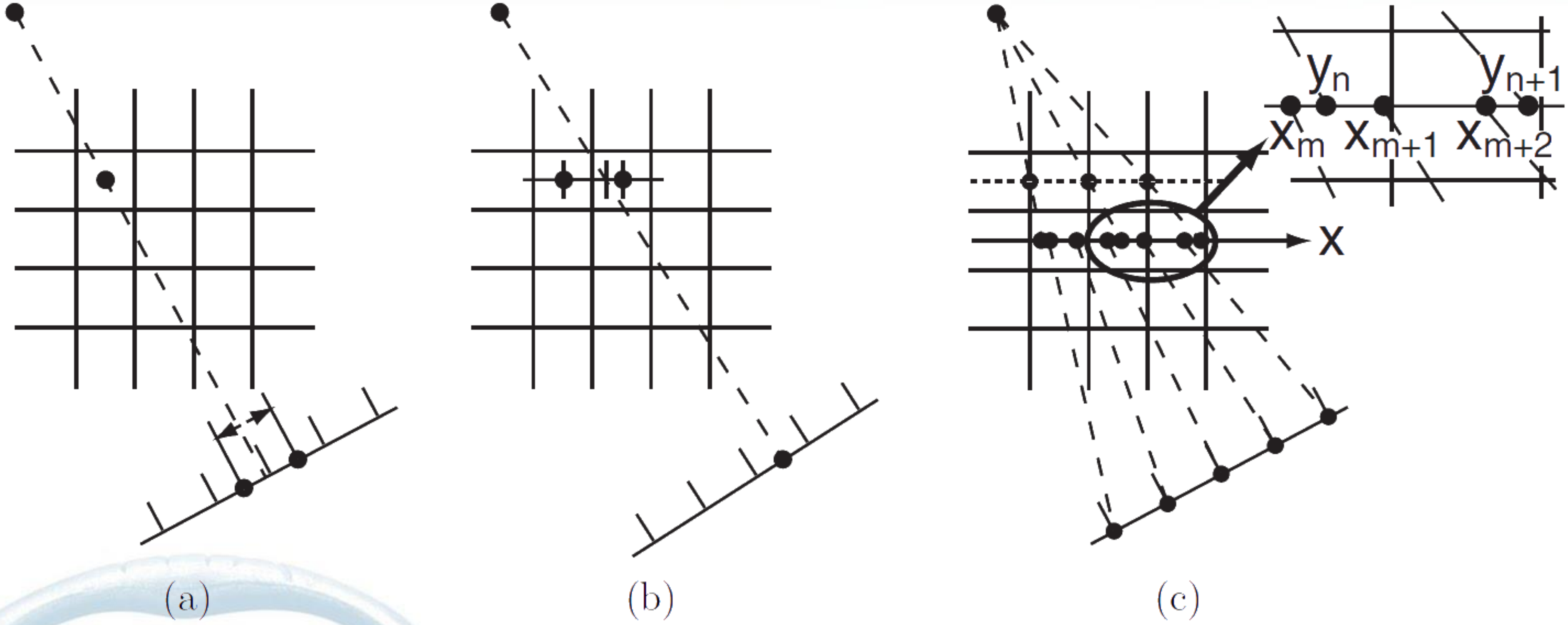
(a) Pixel driven

(b) Ray driven

(c) Distance driven

Figure: B. De Man and S. Basu, “Distance-driven projection and backprojection in three dimensions,” Physics in Medicine and Biology, vol. 49, no. 11, pp. 2463–2475, Jun. 2004.

Projection / Backprojection

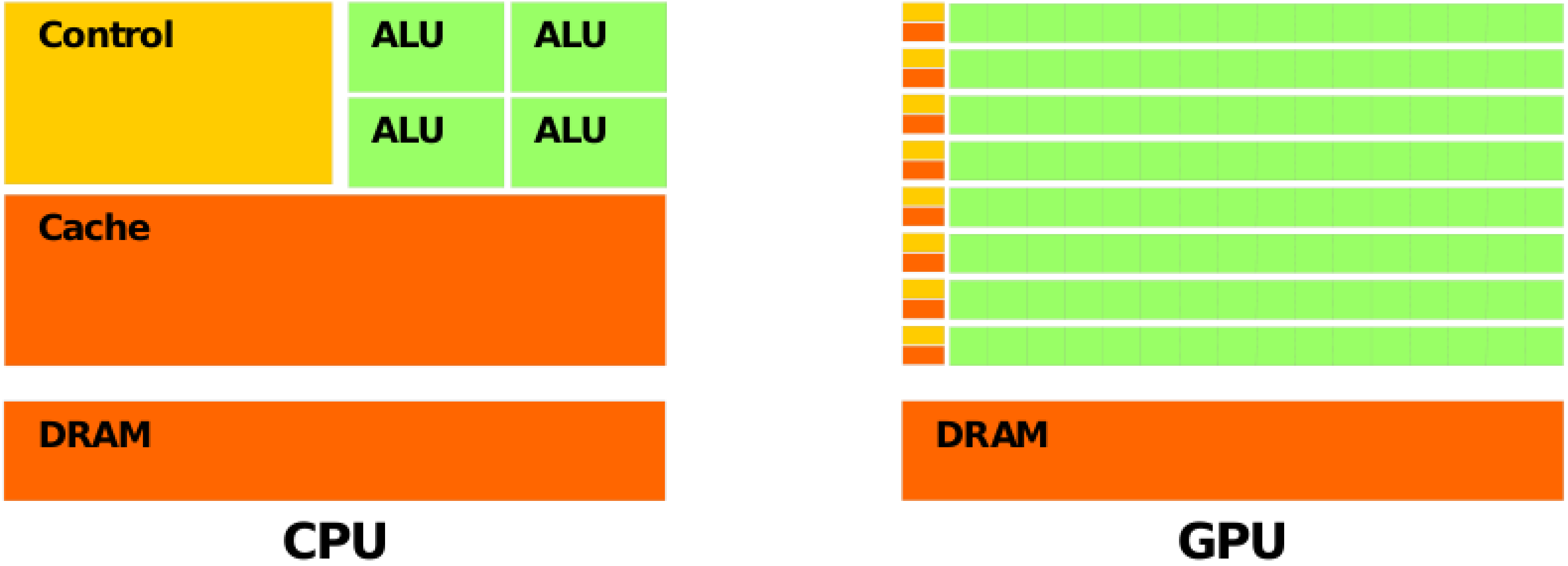


(a) Pixel driven
(b) Ray driven
(c) Distance driven

Embarrassingly parallel

Figure: B. De Man and S. Basu, “Distance-driven projection and backprojection in three dimensions,” Physics in Medicine and Biology, vol. 49, no. 11, pp. 2463–2475, Jun. 2004.

GPU Structure



- CPU: few large cores optimized for serial processing
- GPU: many small cores optimized for parallel performance

GPU Programming

- GPU’s are programmable from 1994
  using graphics languages (Cg / HLSL / GLSL)
  on graphics objects (vertices, textures)
  → Problems need to be translated

- General purpose computing
  - CUDA (C++) released 2007
  - OpenCL (C) released 2008

- Choosing OpenCL or CUDA?

GPU Programming

- GPU’s are programmable from 1994
  using graphics languages (Cg / HLSL / GLSL)
  on graphics objects (vertices, textures)
  → Problems need to be translated

- General purpose computing
  - CUDA (C++) released 2007
  - OpenCL (C) released 2008

- Choosing OpenCL ~~or CUDA?~~
  → Vendor agnostic: allows processing on GPU / CPU / any hardware with drivers
  → Similar performance if optimized

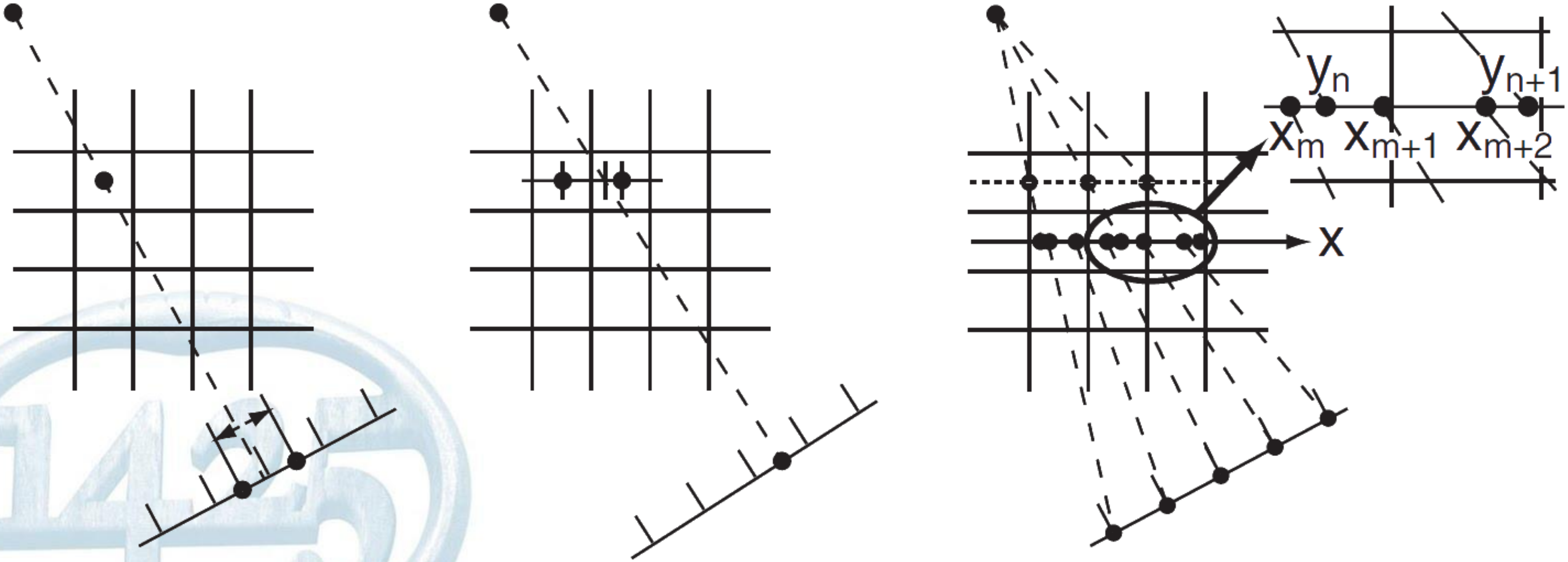
OpenCL basics

- Host code
  - Interface with main software
  - Device management
  - Memory management
  - Just in time compilation
  - …

- Compute device (GPU/CPU) code
  - Performs actual parallel workload
  - C99 subset, including:
    - Vector types
    - Image manipulations
    - Math functions
    - …

Projection / Backprojection

- Distance driven implementation



(a) (b) (c)

Projection / Backprojection

- Distance driven implementation

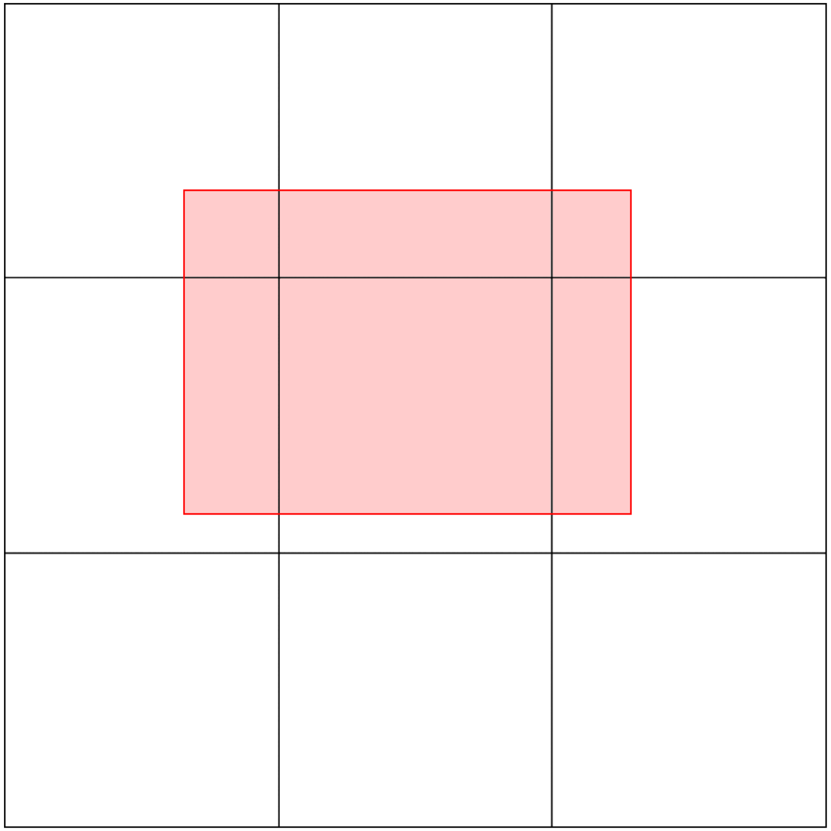
- Input / output data:

  - image
  - sinogram
  - detector elements (corner coordinates)
  - source coordinates
  - image size
  - sinogram size
  - image offset
  - voxel size

Projection / Backprojection

- Distance driven implementation:
  - Memory access: gather versus scatter
    coalesced access

Projection
→ parallel over sinogram

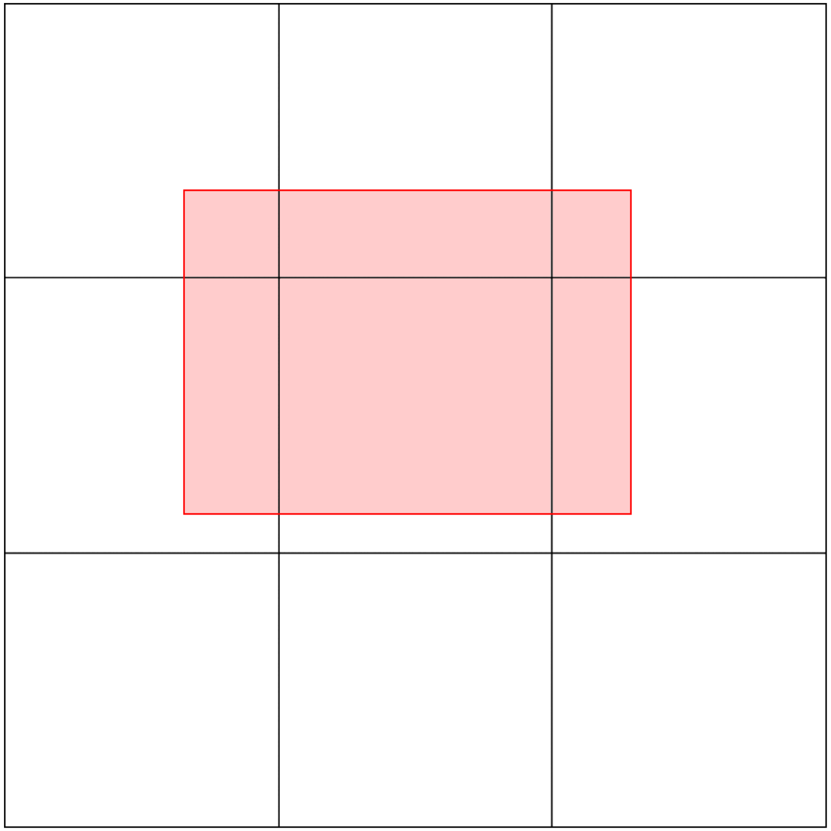


- Check ray direction
- For each plane:
  - Determine intersections
  - Calculate weights
  - Loop over elements
    (read values from volume)
- Write value to sinogram

Projection / Backprojection

- Distance driven implementation:
  - Memory access: gather versus scatter
    coalesced access

Backprojection
→ parallel over sinogram



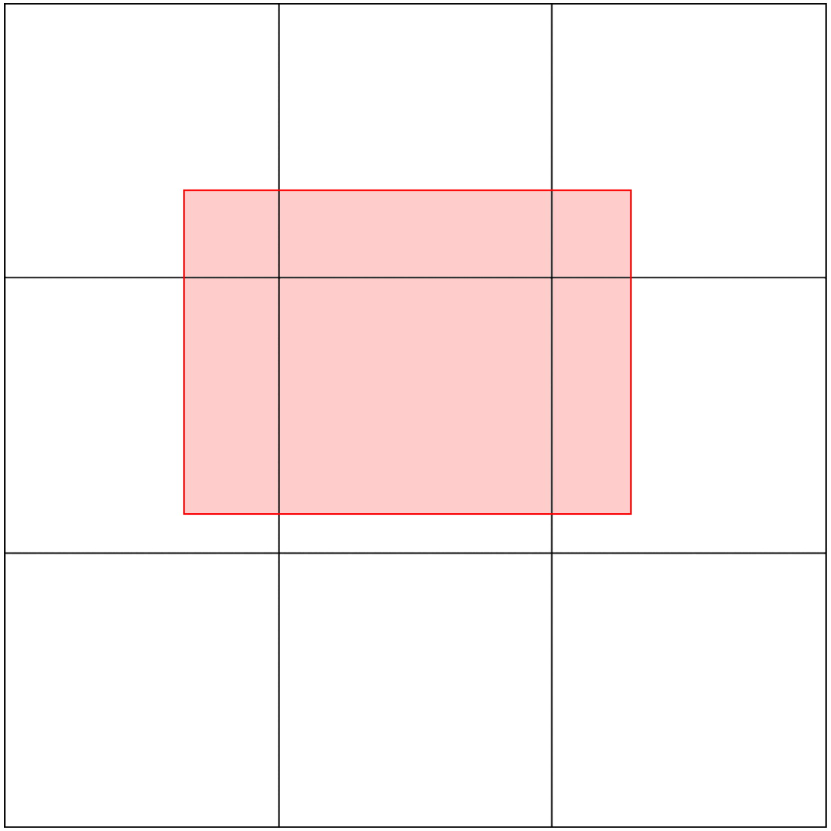
- Check ray direction
- Read value from sinogram
- For each plane:
  - Determine intersections
  - Calculate weights
  - Loop over elements
    (write value to volume)

Projection / Backprojection

- Distance driven implementation:
  - Memory access: gather versus scatter
    coalesced access

Backprojection
→ parallel over volume

- For each angle:
  - Check ray direction
  - Determine intersections
  - Calculate weights
  - Loop over elements
    (read value from volume)
- Write value to volume



Projection / Backprojection

- Distance driven implementation
  - Memory access: gather versus scatter
    coalesced access

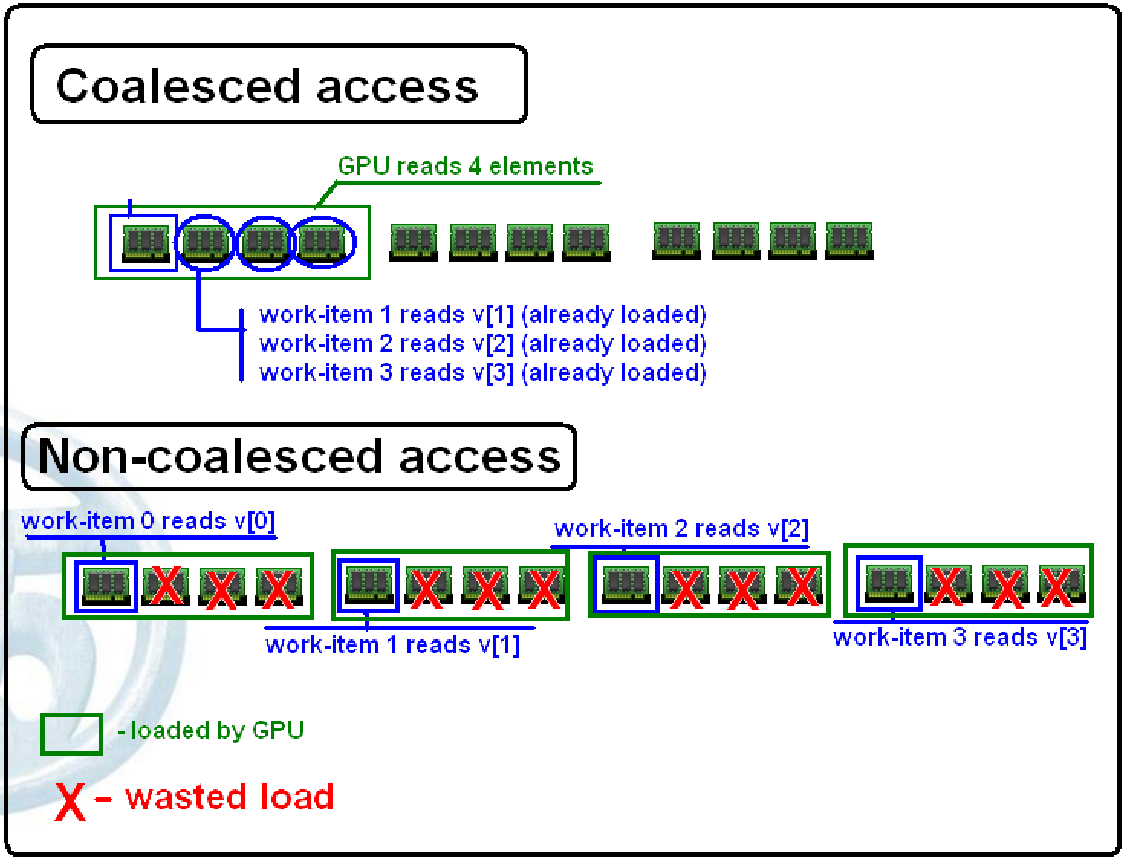


Figure: ‘OpenCL Optimization Strategies’, www.cmsoft.com.br

Projection / Backprojection

- Distance driven implementation
  - Memory access: gather versus scatter
    coalesced access

- Inefficient (atomic) implementation for backwards compatibility

Projection of 45 planes to 25 angles:

- Parallel over sinogram: 8.2s

Backprojection from 25 angles to 45 planes:

- Parallel over sinogram: 38.4 s
- Parallel over image: 18.3 s

Projection / Backprojection

- Distance driven implementation
  - Tomosynthesis versus general detector geometry:
    - flat detector, parallel to volume
    - one ray direction

```
#ifdef TOMOSYNTHESIS
    temp_float4 = ((corner1 + corner4) * 0.5f) - source;
#else
    temp_float4 = ((corner1 + corner2 + corner3 + corner4) * 0.25f) - source;
#endif
...
corner1 -= source;
#ifndef TOMOSYNTHESIS
    corner2 -= source;
    corner3 -= source;
#endif
corner4 -= source;
```

Projection / Backprojection

- Distance driven implementation
  - Tomosynthesis versus general detector geometry:
    - flat detector, parallel to volume
    - one ray direction

- Kernel timings (no overhead):

  Projection of 45 planes to 25 angles:

  - General detector geometry

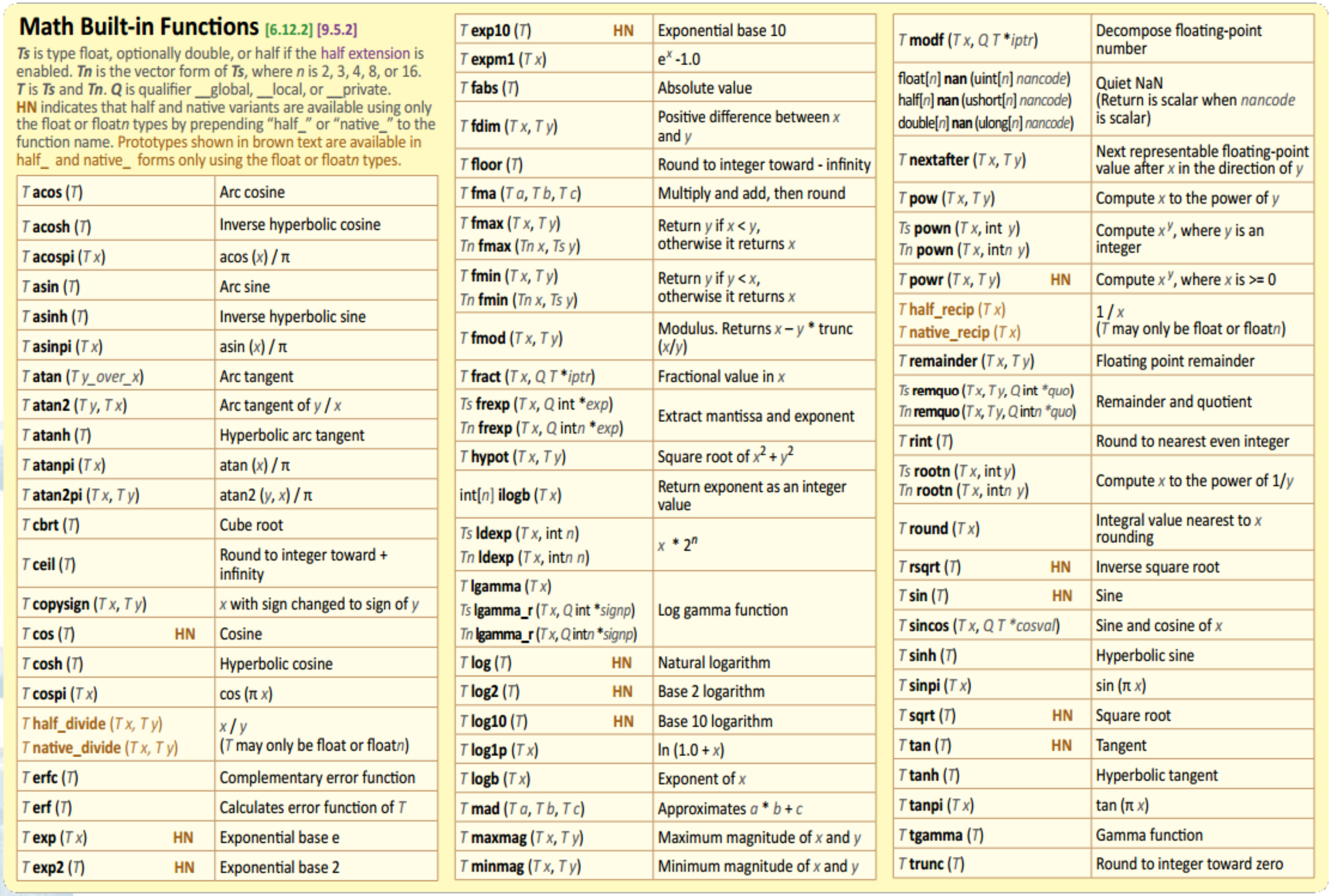
    10 s

  - Flat detector geometry

    7.6 s

Projection / Backprojection

- Distance driven implementation
  - Fast mathematical functions: fused, native, half-precision



Math Built-in Functions [6.12.2] [9.5.2]

Ts is type float, optionally double, or half if the half extension is enabled. *Tn* is the vector form of *Ts*, where *n* is 2, 3, 4, 8, or 16. *T* is *Ts* and *Tn*. *Q* is qualifier __global, __local, or __private. HN indicates that half and native variants are available using only the float or float*n* types by prepending "half_" or "native_" to the function name. Prototypes shown in brown text are available in half_ and native_ forms only using the float or float*n* types.

T **acos** (T)		Arc cosine
T **acosh** (T)		Inverse hyperbolic cosine
T **acospi** (T x)		acos (x) / π
T **asin** (T)		Arc sine
T **asinh** (T)		Inverse hyperbolic sine
T **asinpi** (T x)		asin (x) / π
T **atan** (T y_over_x)		Arc tangent
T **atan2** (T y, T x)		Arc tangent of y / x
T **atanh** (T)		Hyperbolic arc tangent
T **atanpi** (T x)		atan (x) / π
T **atan2pi** (T x, T y)		atan2 (y, x) / π
T **cbrt** (T)		Cube root
T **ceil** (T)		Round to integer toward + infinity
T **copysign** (T x, T y)		x with sign changed to sign of y
T **cos** (T)	HN	Cosine
T **cosh** (T)		Hyperbolic cosine
T **cospi** (T x)		cos (π x)
T **half_divide** (T x, T y) T **native_divide** (T x, T y)		x / y (T may only be float or floatn)
T **erfc** (T)		Complementary error function
T **erf** (T)		Calculates error function of T
T **exp** (T x)	HN	Exponential base e
T **exp2** (T)	HN	Exponential base 2
T **exp10** (T)	HN	Exponential base 10
T **expm1** (T x)		e^x -1.0
T **fabs** (T)		Absolute value
T **fdim** (T x, T y)		Positive difference between x and y
T **floor** (T)		Round to integer toward - infinity
T **fma** (T a, T b, T c)		Multiply and add, then round
T **fmax** (T x, T y) Tn **fmax** (Tn x, Ts y)		Return y if x < y, otherwise it returns x
T **fmin** (T x, T y) Tn **fmin** (Tn x, Ts y)		Return y if y < x, otherwise it returns x
T **fmod** (T x, T y)		Modulus. Returns x – y * trunc (x/y)
T **fract** (T x, Q T *iptr)		Fractional value in x
Ts **frexp** (T x, Q int *exp) Tn **frexp** (T x, Q intn *exp)		Extract mantissa and exponent
T **hypot** (T x, T y)		Square root of $x^2 + y^2$
int[n] **ilogb** (T x)		Return exponent as an integer value
Ts **ldexp** (T x, int n) Tn **ldexp** (T x, intn n)		$x * 2^n$
T **lgamma** (T x) Ts **lgamma_r** (T x, Q int *signp) Tn **lgamma_r** (T x, Q intn *signp)		Log gamma function
T **log** (T)	HN	Natural logarithm
T **log2** (T)	HN	Base 2 logarithm
T **log10** (T)	HN	Base 10 logarithm
T **log1p** (T x)		ln (1.0 + x)
T **logb** (T x)		Exponent of x
T **mad** (T a, T b, T c)		Approximates a * b + c
T **maxmag** (T x, T y)		Maximum magnitude of x and y
T **minmag** (T x, T y)		Minimum magnitude of x and y
T **modf** (T x, Q T *iptr)		Decompose floating-point number
float[n] **nan** (uint[n] nancode) half[n] **nan** (ushort[n] nancode) double[n] **nan** (ulong[n] nancode)		Quiet NaN (Return is scalar when nancode is scalar)
T **nextafter** (T x, T y)		Next representable floating-point value after x in the direction of y
T **pow** (T x, T y)		Compute x to the power of y
Ts **pown** (T x, int y) Tn **pown** (T x, intn y)		Compute x^y, where y is an integer
T **powr** (T x, T y)	HN	Compute x^y, where x is >= 0
T **half_recip** (T x) T **native_recip** (T x)		1 / x (T may only be float or floatn)
T **remainder** (T x, T y)		Floating point remainder
Ts **remquo** (T x, T y, Q int *quo) Tn **remquo** (T x, T y, Q intn *quo)		Remainder and quotient
T **rint** (T)		Round to nearest even integer
Ts **rootn** (T x, int y) Tn **rootn** (T x, intn y)		Compute x to the power of 1/y
T **round** (T x)		Integral value nearest to x rounding
T **rsqrt** (T)	HN	Inverse square root
T **sin** (T)	HN	Sine
T **sincos** (T x, Q T *cosval)		Sine and cosine of x
T **sinh** (T)		Hyperbolic sine
T **sinpi** (T x)		sin (π x)
T **sqrt** (T)	HN	Square root
T **tan** (T)	HN	Tangent
T **tanh** (T)		Hyperbolic tangent
T **tanpi** (T x)		tan (π x)
T **tgamma** (T)		Gamma function
T **trunc** (T)		Round to integer toward zero

Projection / Backprojection

- Distance driven implementation
  - Fast mathematical functions: fused, native, half-precision
    - Fused operations:

      mad(a,b,c) = a * b + c
    - Native operations:

      uses hardware optimized instructions
    - Half-precision operations:

      uses 16 bit precision instead of 32 bit

→ Not necessarily IEEE 754 compliant !

Projection / Backprojection

- Distance driven implementation
  - Fast mathematical functions: fused, native, half-precision

```
#ifdef FAST_MATH
   plane_p.z = mad(this_vox_z + 0.5f, vox_size.z, img_offset.z);
#else
   plane_p.z = (this_vox_z + 0.5f) * vox_size.z + img_offset.z;
#endif
...
#ifdef FAST_MATH
   temp_float4 = native_recip(temp_float4);
#else
   temp_float4 = 1.0f / temp_float4;
#endif
```

Projection / Backprojection

- Distance driven implementation
  - Fast mathematical functions: fused, native, half-precision

- Kernel timings (no overhead):

  Projection of 45 planes to 25 angles:

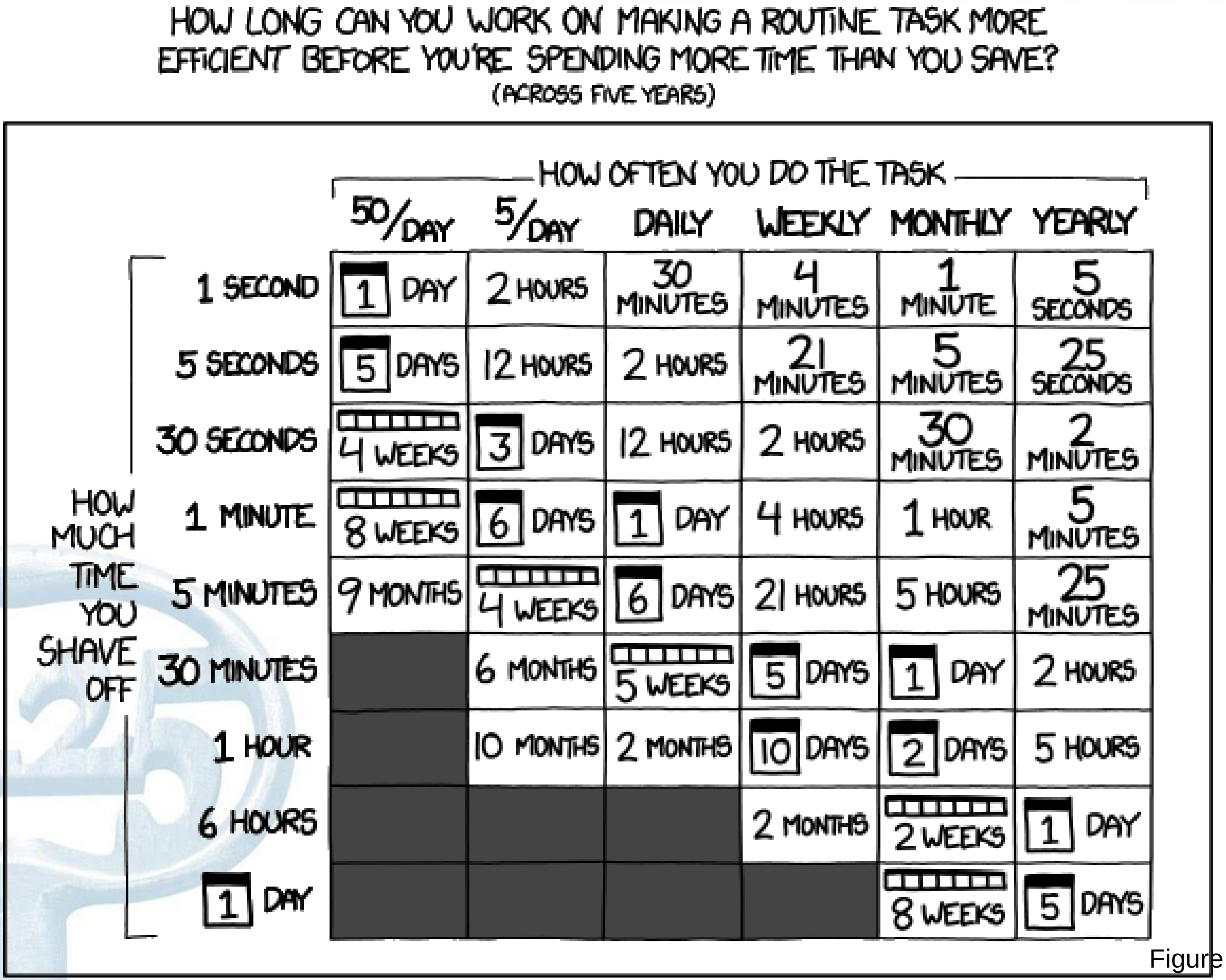
  - Without ‘fast math’

    7.6 s

  - With ‘fast math’

    6.4 s

More Optimizations?



HOW LONG CAN YOU WORK ON MAKING A ROUTINE TASK MORE EFFICIENT BEFORE YOU'RE SPENDING MORE TIME THAN YOU SAVE?
(ACROSS FIVE YEARS)

	HOW OFTEN YOU DO THE TASK					
HOW MUCH TIME YOU SHAVE OFF	50/DAY	5/DAY	DAILY	WEEKLY	MONTHLY	YEARLY
1 SECOND	1 DAY	2 HOURS	30 MINUTES	4 MINUTES	1 MINUTE	5 SECONDS
5 SECONDS	5 DAYS	12 HOURS	2 HOURS	21 MINUTES	5 MINUTES	25 SECONDS
30 SECONDS	4 WEEKS	3 DAYS	12 HOURS	2 HOURS	30 MINUTES	2 MINUTES
1 MINUTE	8 WEEKS	6 DAYS	1 DAY	4 HOURS	1 HOUR	5 MINUTES
5 MINUTES	9 MONTHS	4 WEEKS	6 DAYS	21 HOURS	5 HOURS	25 MINUTES
30 MINUTES	■	6 MONTHS	5 WEEKS	5 DAYS	1 DAY	2 HOURS
1 HOUR	■	10 MONTHS	2 MONTHS	10 DAYS	2 DAYS	5 HOURS
6 HOURS	■	■	■	2 MONTHS	2 WEEKS	1 DAY
1 DAY	■	■	■	■	8 WEEKS	5 DAYS

Figure: XKCD 1205

More Optimizations?

- Using constant memory
  - pre-cached on GPU
- Using vector types
  - 1 float4 operation instead of 4 float operations (superfluous when using a smart compiler)
- Unrolling loops / fixed length loops
  - Branching is expensive on GPU
- Use #define macros to pass variables at compilation
  - Literal values are more efficient than variables
- Proper local memory use
  - Local memory is much faster than global memory
- …

More Optimizations?

- Using constant memory
- Using vector types
- Unrolling loops / fixed length loops
- Use #define macros to pass variables at compilation
- Proper local memory use
- …

However: balance optimizations against flexibility

- Tomosynthesis only **OR** all projection geometries
- GPU and / or CPU optimization

Conclusions

- Example study: 12 patients, ± lesion, ± scatter
  → 48 reconstruction (10 iterations of MLTR)

  → Originally (Intel Xeon E5440 @ 2.8 GHz, 1 thread):

  24h / reconstruction

  → Currently (without ‘fast math’):

  - AMD Opteron 6166 HE @ 1.8 GHz, 32 threads:

  50m / reconstruction

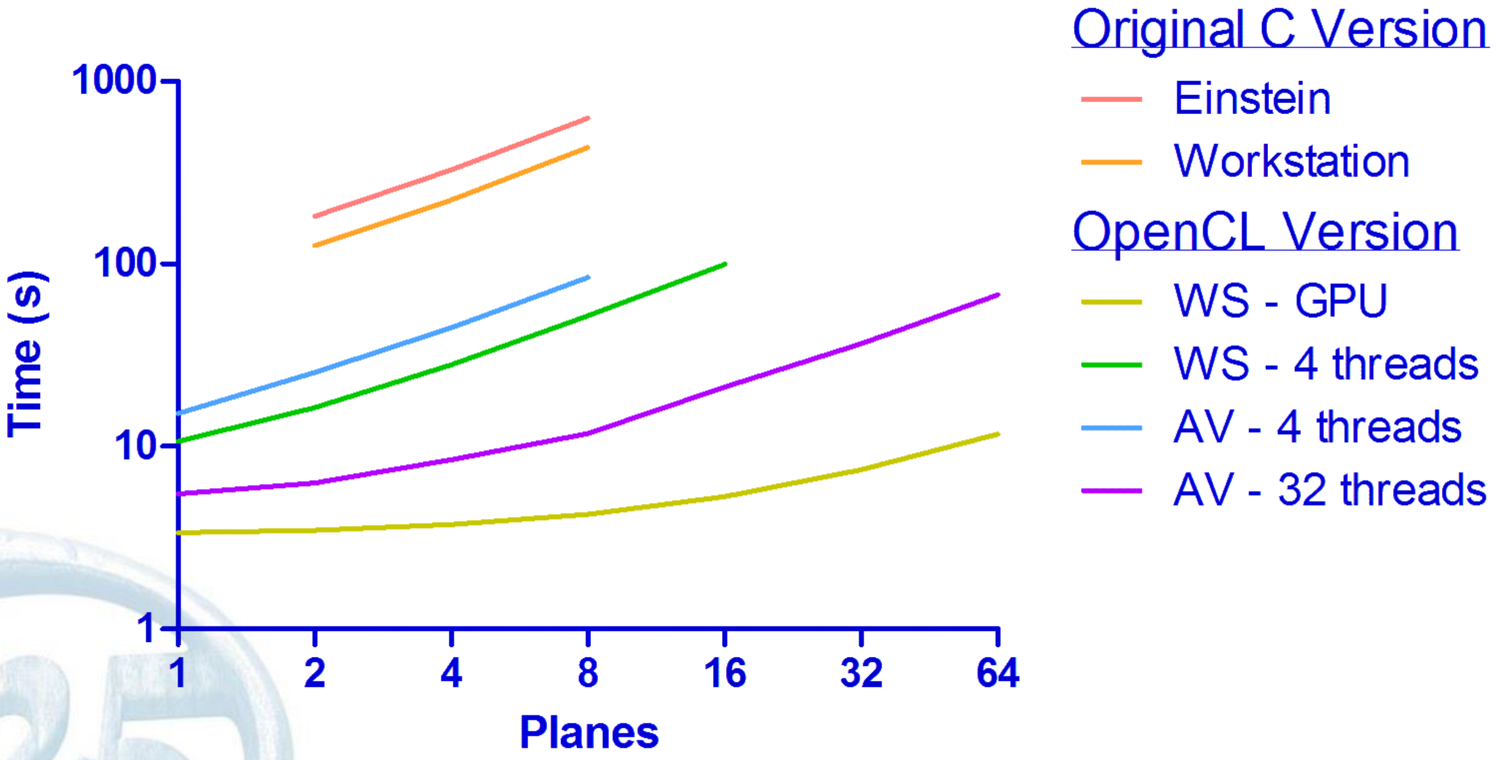
  - nVidia Tesla C2075 @ 1.15 GHz, 448 threads

  8m / reconstruction

Thanks!

OpenCL Implementation

- Some benchmarks:



• Einstein	CPU	Intel Xeon E5440 @ 2.83 GHz
• Workstation	CPU	Intel Xeon E5606 @ 2.13 GHz
	GPU	Nvidia Tesla C2075
• Avalok	CPU	AMD Opteron 6128HE @ 2.0 GHz