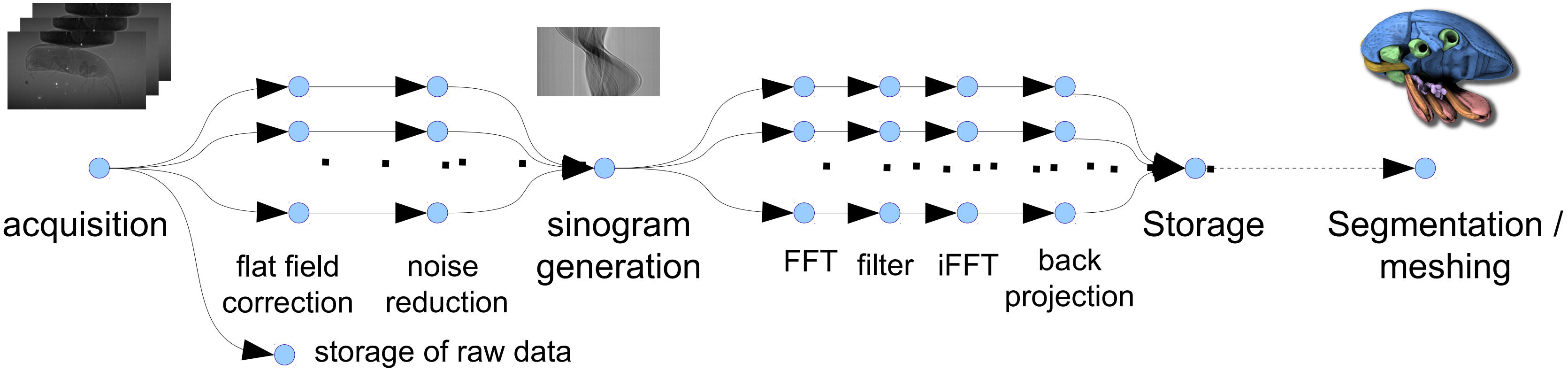
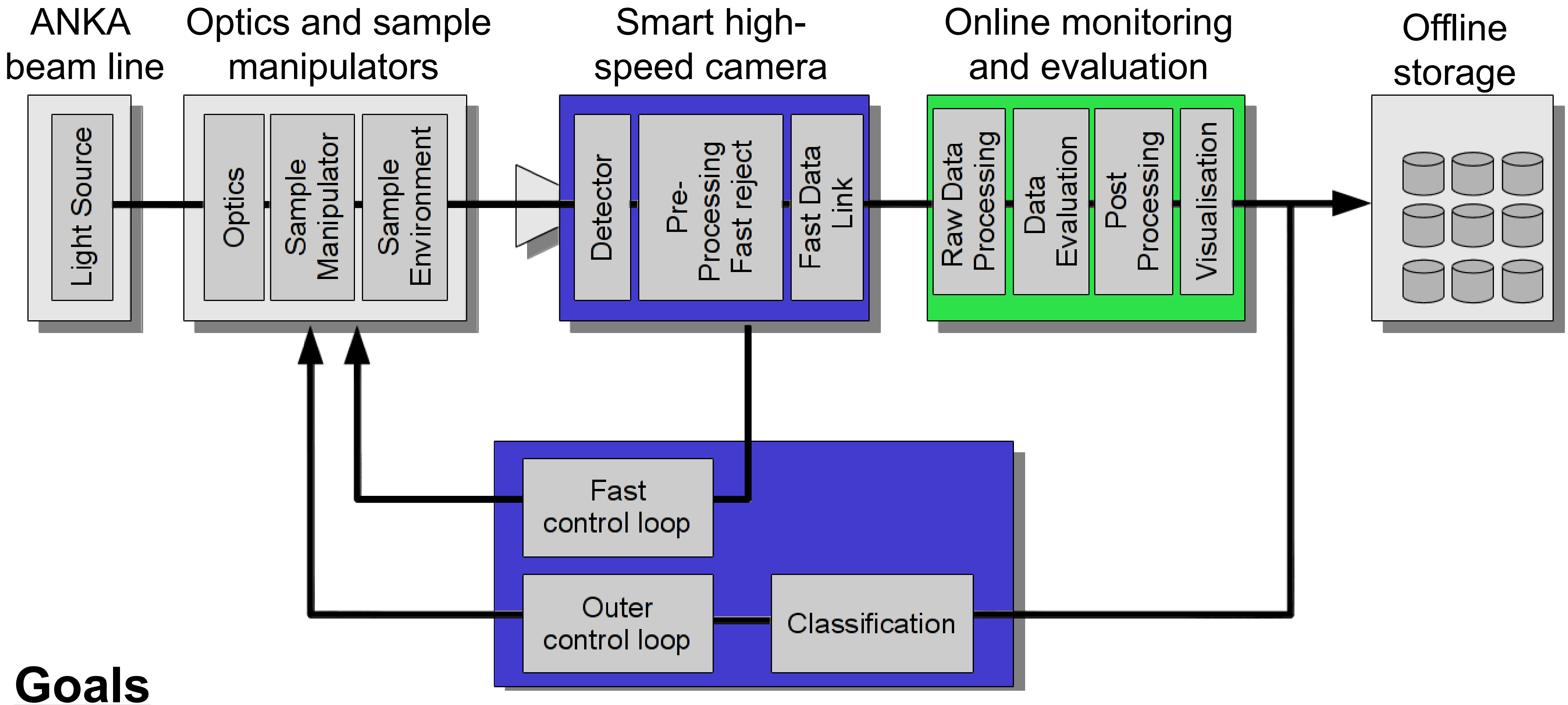
Karlsruhe Institute of Technology

Adapting Filtered Back Projection algorithm for various parallel architectures

S. Chilingaryan


acquisition
flat field correction
noise reduction
sinogram generation
FFT
filter
iFFT
back projection
Storage
Segmentation / meshing
storage of raw data

UFO *Ultra Fast X-ray Imaging of Scientific Processes with On-Line Assessment and Data-Driven Process Control*



ANKA beam line
- Light Source

Optics and sample manipulators
- Optics
- Sample Manipulator
- Sample Environment

Smart high-speed camera
- Detector
- Pre-Processing Fast reject
- Fast Data Link

Online monitoring and evaluation
- Raw Data Processing
- Data Evaluation
- Post Processing
- Visualisation

Offline storage

Fast control loop

Outer control loop

Classification

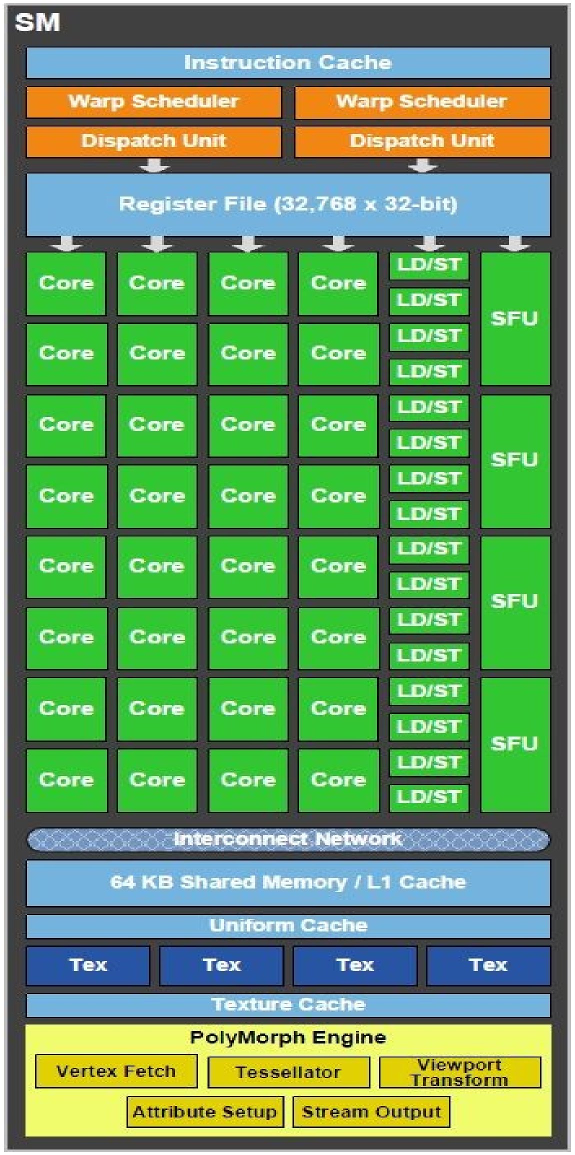
Goals

- High speed tomography
- Increase sample throughput
- Tomography of temporal processes
- Allow interactive quality assessment
- Enable data driven control
  - Auto-tunning optical system
  - Tracking dynamic processes
  - Finding area of interest

Optimizing for parallel architectures

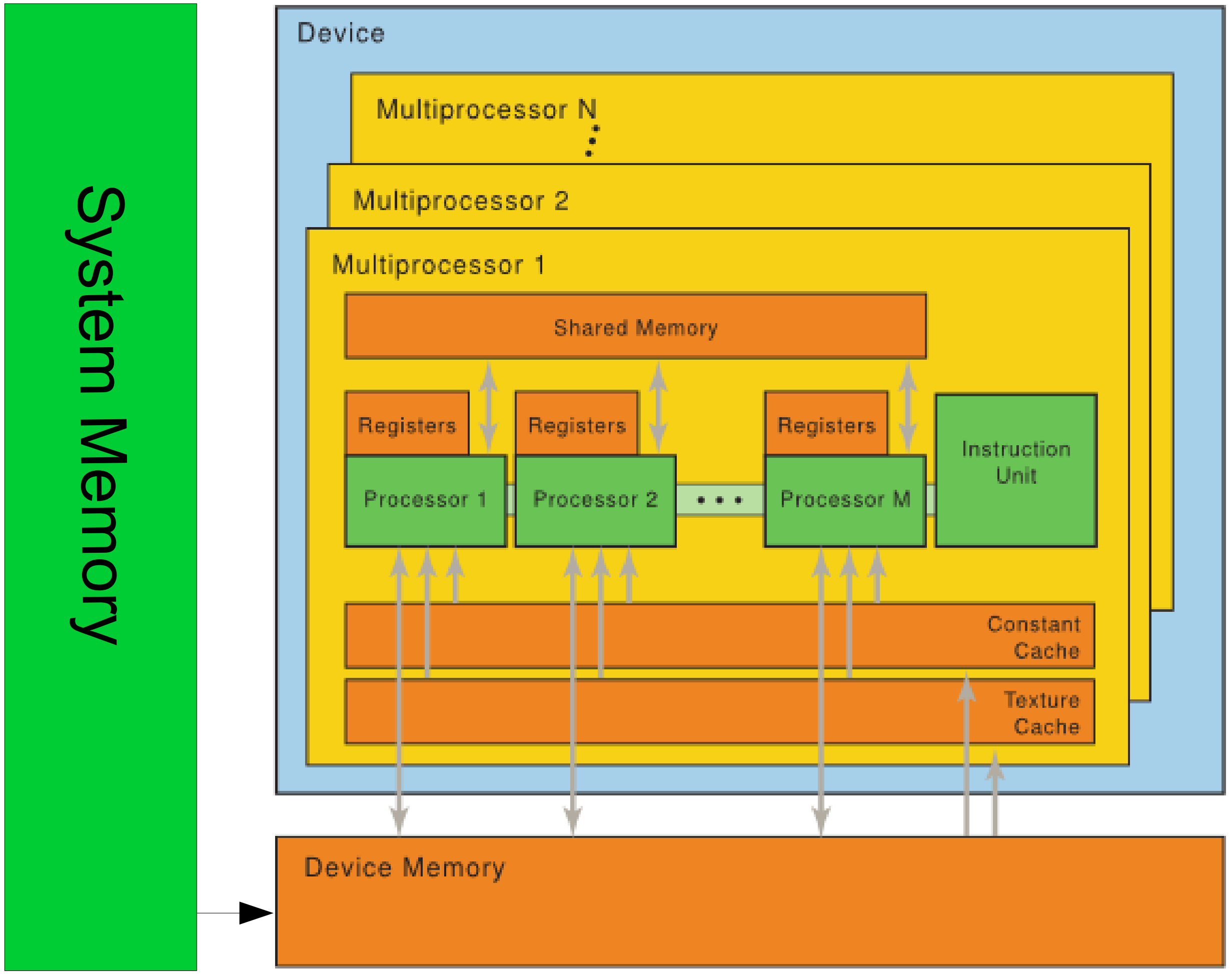
- **Consists of SIMD-type Compute Units (CU)**
  - One instruction is executed on many data items
  - Each CU able to execute several operation types
  - But only FP additions/multiplications are fast
- **Posses complex memory hierarchy**
  - Low Bandwidth-per-flop ratio and small caches
  - Up to four different types of memory
  - Optimal access pattern have to be followed
- **Architectures vary drastically**
  - Sizes, speed, and structure of memories / caches
  - Types and amount of provided processing units
  - Balance of operation throughput

Codes and algorithms have to be carefully optimized for the specific parallel architecture



Compute Unit on Fermi

Memory model



- **Host Memory**
  - 6 GB/s (PCIe x16 gen2) to 12 GB/s (PCIe x16 gen3)
- **Global Memory**
  - 100 – 300 GB/s with latencies up to 1000 clocks
- **Local Memory**
  - 1 – 2 TB/s (total) with latencies below 100 clocks
- **Registers**
  - private to threads
- **Caches**
  - L1/L2 cache
  - Texture cache
  - Constant memory

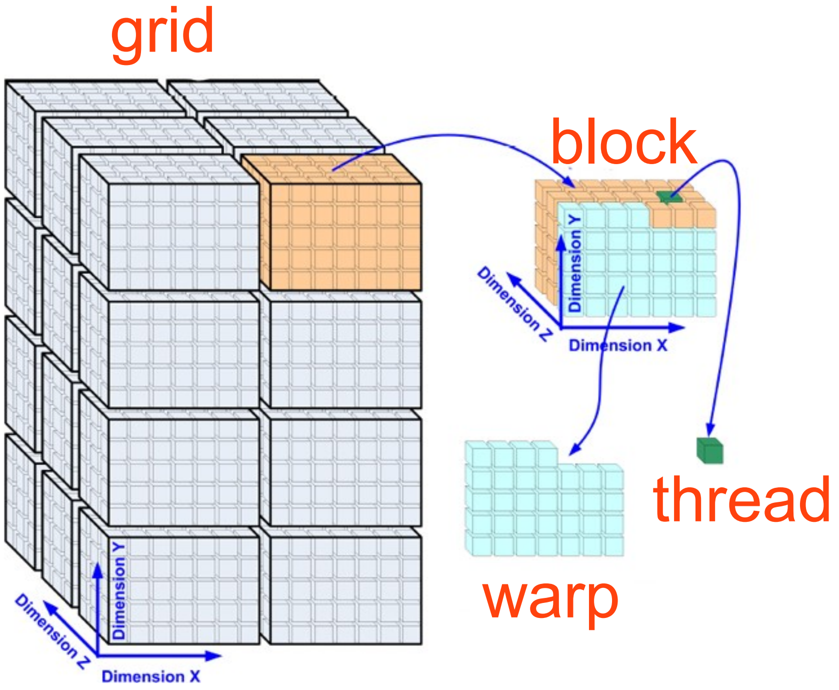
Complex memory hierarchy consisting of 4 levels and with each level one order of magnitude faster when previous!

Programming Model

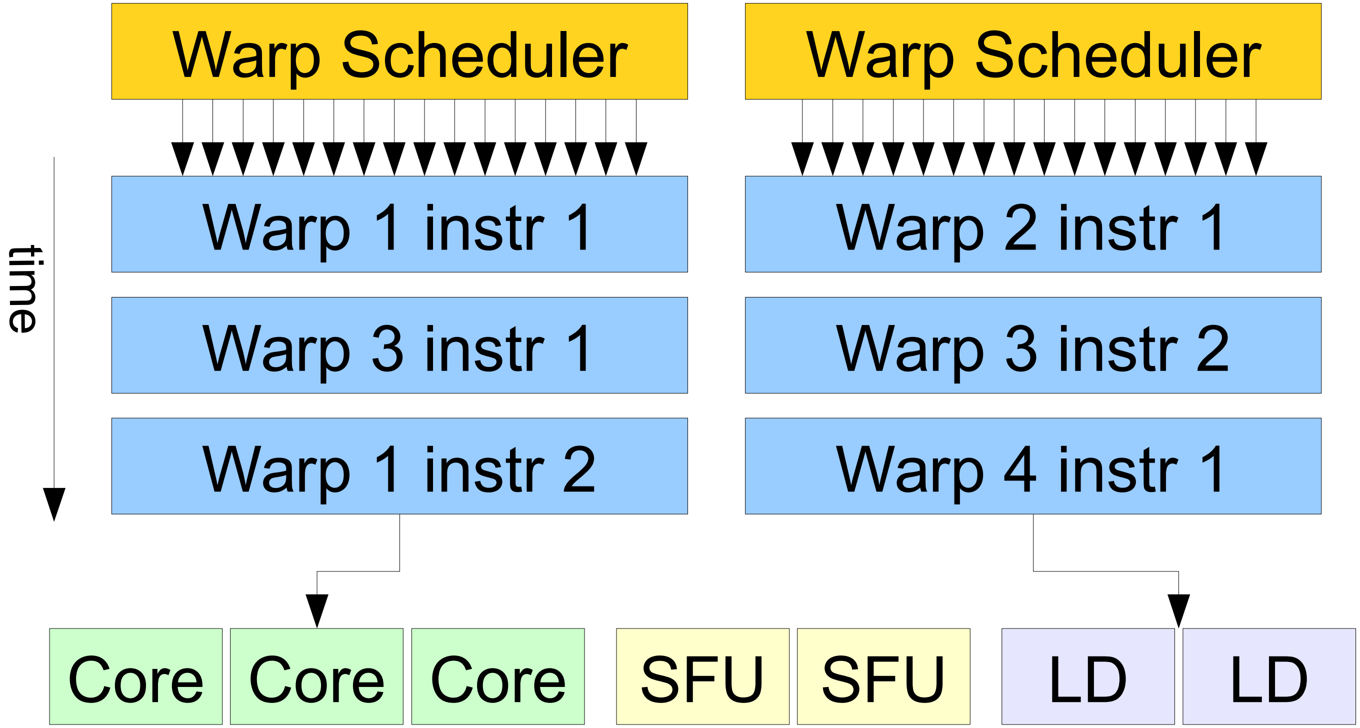
Thread abstraction is used to split the problem space into the independent GPU tasks

- All threads execute the same code (**kernel**)
- Task is defined by the linear or volumetric index of the thread
- GPU schedules threads in groups of fixed size (**warp**)
- A user-defined **block** of threads is assigned to a specific CU
- Threads of the block may exchange data using CU shared memory

e.g. resulting image is mapped to a 1-, 2-, or 3D grid of GPU threads and each pixel is computed by a thread with the index equal to pixel coordinates



Scheduling



time

Warp Scheduler

Warp Scheduler

Warp 1 instr 1

Warp 2 instr 1

Warp 3 instr 1

Warp 3 instr 2

Warp 1 instr 2

Warp 4 instr 1

Core

Core

Core

SFU

SFU

LD

LD

Multiple warps on CU executed in parallel

Independent instructions executed in parallel

Warp 4 will be blocked for a long time, but other warps on CU will execute and hide the latency

- Warps from several blocks are executed by CU in parallel
- The number of currently resident warps is called **occupancy**
- Occupancy is limited by available registers and shared memory
- Suboptimal occupancy limits the instruction bandwidth

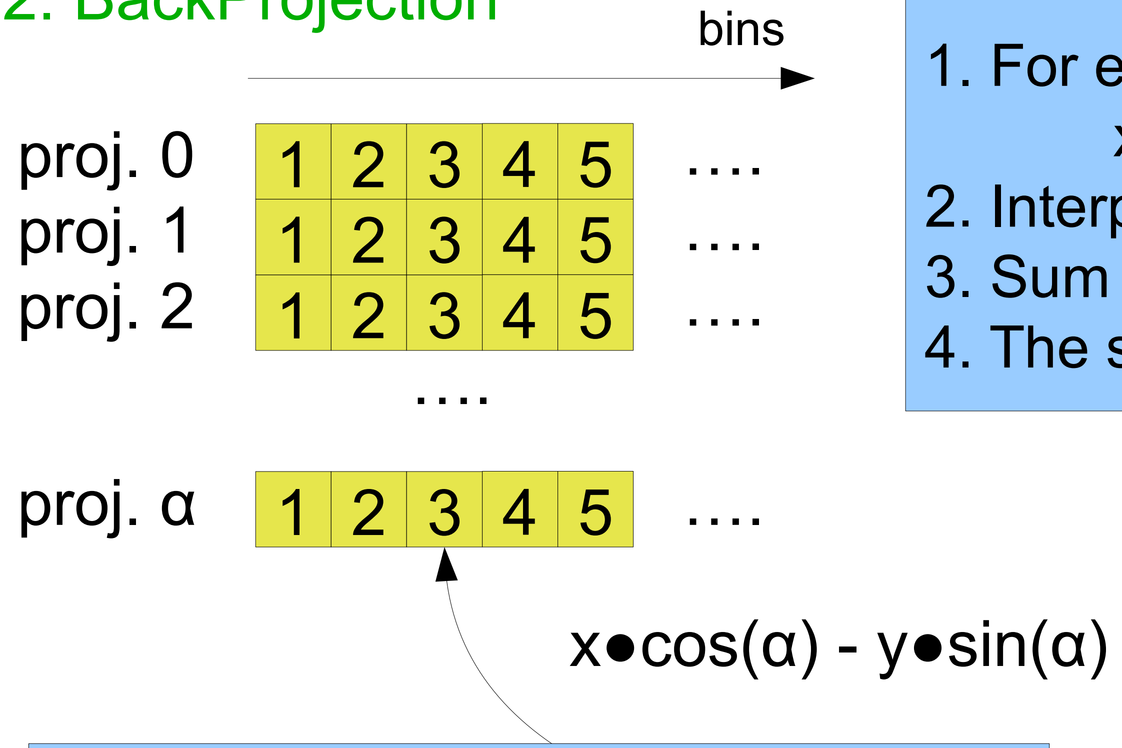
For optimal performance we have to increase occupancy and number of independent instructions

FBP Reconstruction

1. Filtering

Multiplication with the configured filter in the Fourier space

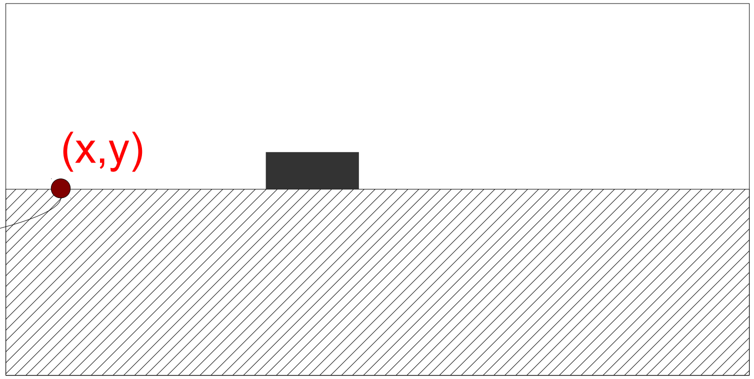
2. BackProjection



x•cos(α) - y•sin(α)

1. For each position we compute:
   x•cos(α) - y•sin(α)
2. Interpolate between neighboring bins
3. Sum over all projection
4. The sum is the value of (x,y)

For each texel of output volume and for each projection we perform a single linear interpolation



S. Chilingaryan et. all

Texture Engine

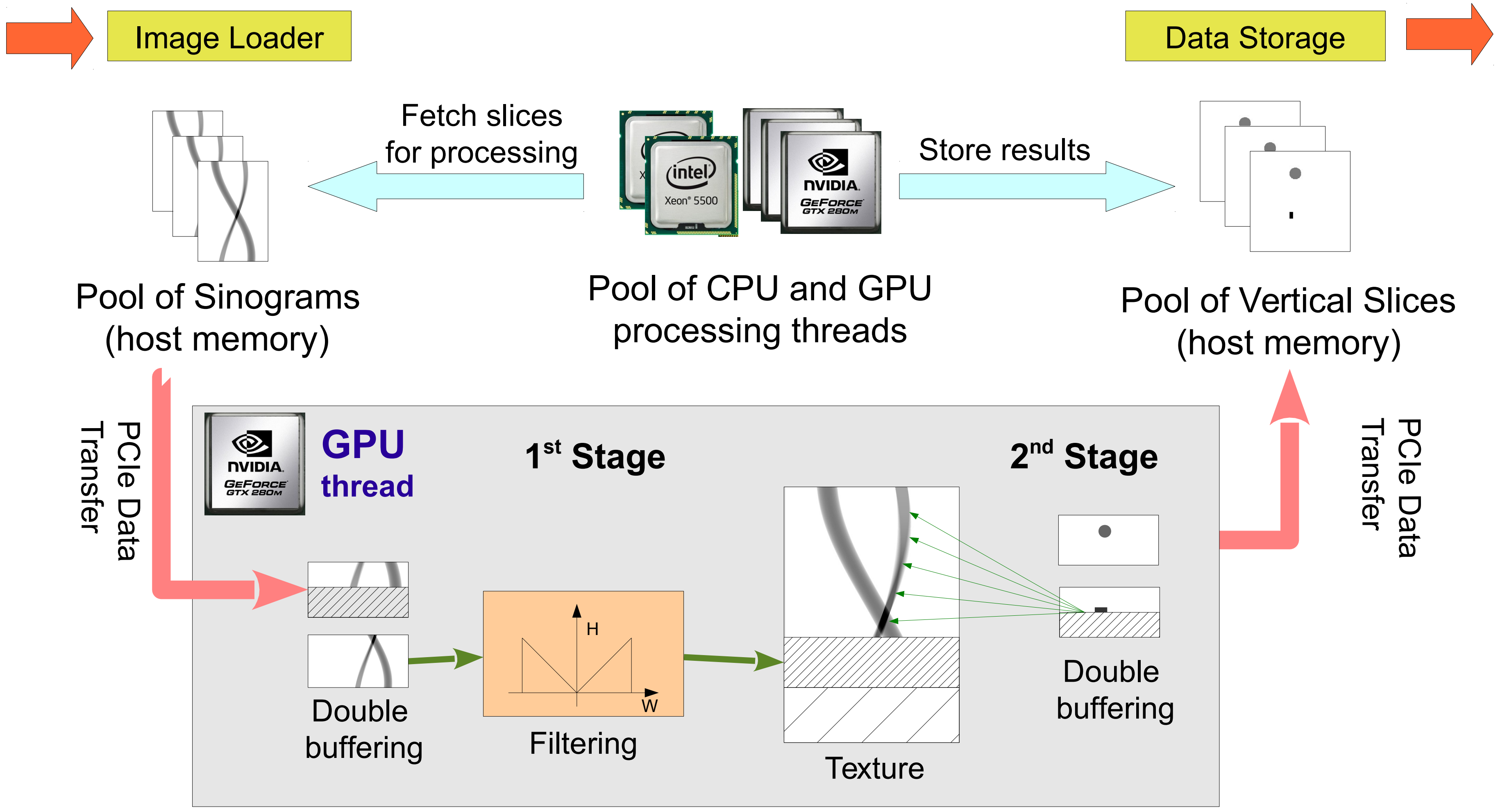
Features:

- Spatial-aware cache
- Bi/tri-linear interpolation
- Normalized coordinates
- Different clamping modes

Applications:

- Linear interpolation, i.e. image scaling
- Optimize random access to multidimensional arrays

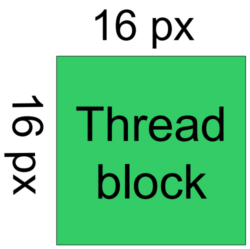
Filtered Back Projection



Performance of Texture Engine

	GT280	GTX580
Core Throughput	930 GF	1580 GF
Texture Fill Rate	48 GT/s	49 GT/s
Ratio	**19.3**	**31.6**

Optimizing FBP for Fermi

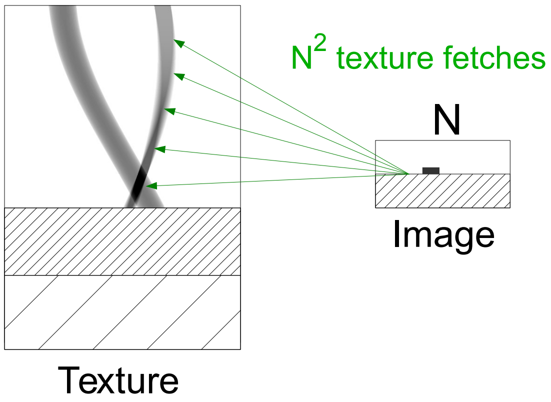


$$v = x \bullet \cos(\alpha) - y \bullet \sin(\alpha)$$

$$\max_{x,y<N}(v) - \min_{x,y<N}(v) < N\sqrt{2}$$

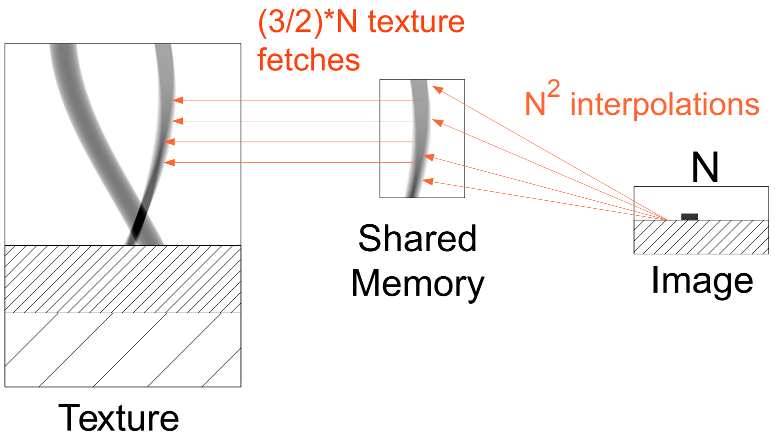
$$N\sqrt{2} < 1.5\ N$$

Each block of threads accesses actually only 3 • N / 2 bins per projection



Standard Version

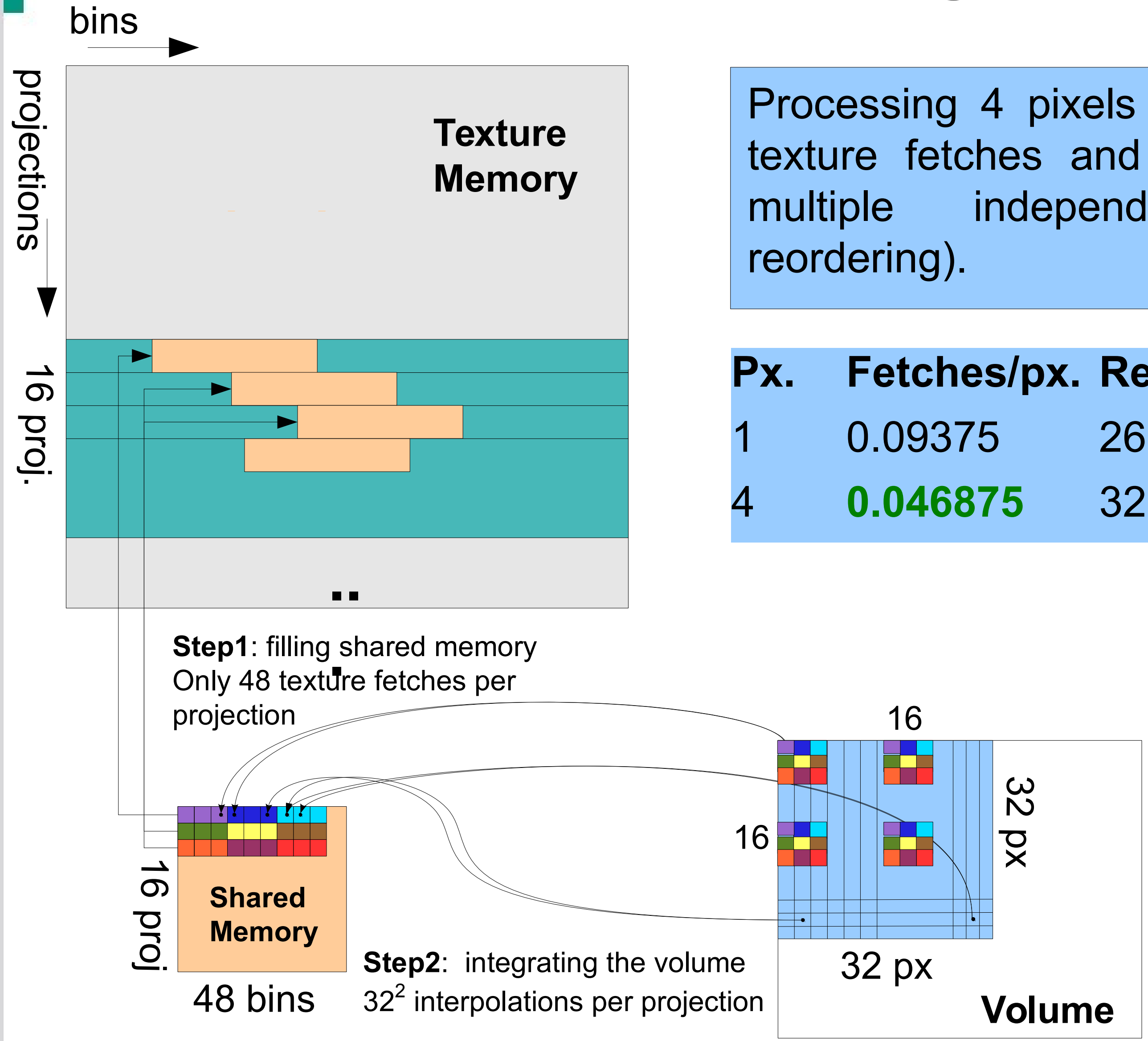
Texture engine is heavily loaded



Fermi-optimized Version

Both texture & computations engines are used

Pixel to thread mapping



Step1: filling shared memory
Only 48 texture fetches per projection

Step2: integrating the volume
32^2 interpolations per projection

Processing 4 pixels per thread reducing amount of texture fetches and hides operation latencies with multiple independent operations (instruction reordering).

Px.	Fetches/px.	Regs	ShMem	Occup.	ILP
1	0.09375	26	1536	66%	1
4	**0.046875**	32	3072	66%	**4**

Legend

🟦 Processed by a single thread block (16x16)

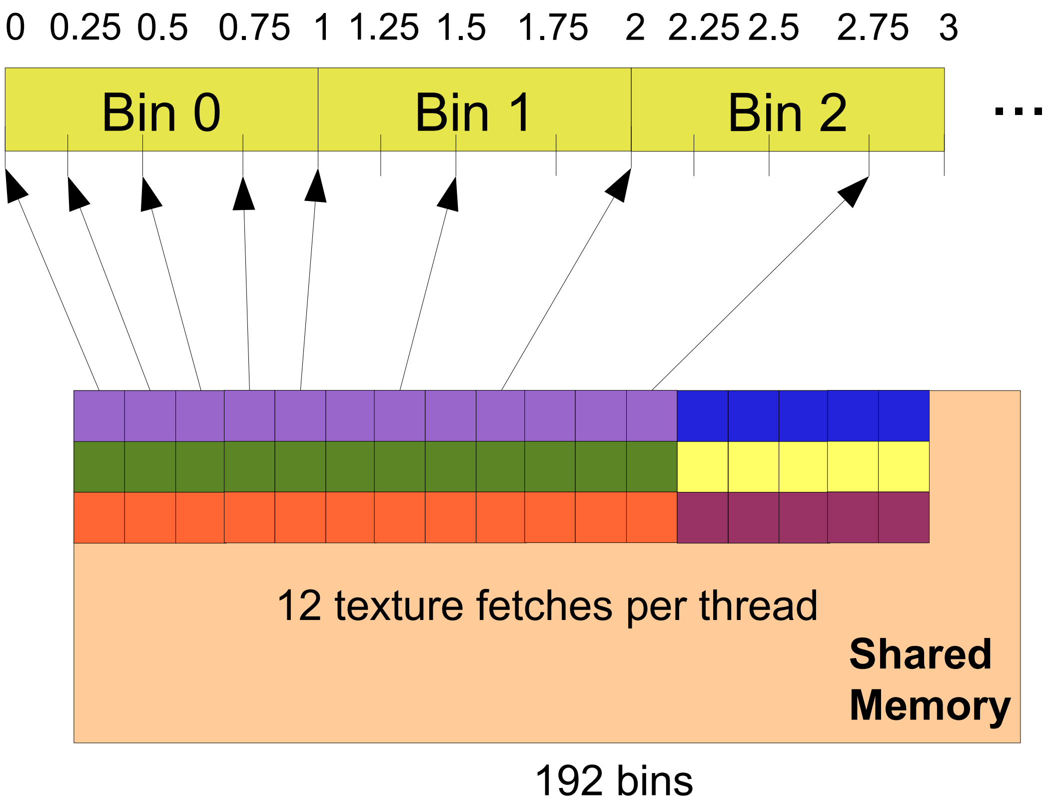
🟧 48 bins of a projection required for current block

🟩 16 of the projections processed in a single pass

🟪 thr (1,1)
🟦 thr (1,2)
🟦 thr (1,3)
🟩 thr (2,1)
🟨 thr (2,2)
🟫 thr (2,3)
🟧 thr (3,1)
🟪 thr (3,2)
🟥 thr (3,3)

Processing in multiple passes,
16 projections each

Oversampling



Linear interpolation is slow, and nearest neighbor is not precise enough

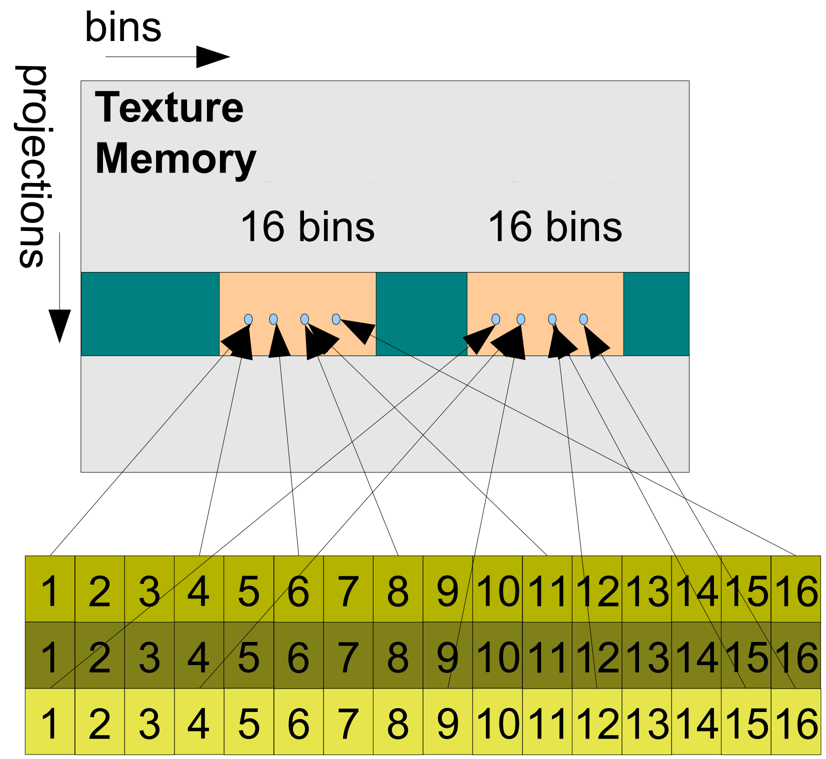
With oversampling the texture engine is used to interpolate 4 positions for each projection bin and near-neighbor interpolation is used then.

Method	Fetches/px	Regs	ShMem	Occup.	Reads/px	Flops/px
Linear	0.046875	32	3072	66%	2	7
Oversample	0.1875	42	12288	50%	1	4

Kepler: Fast Texture Engine is Back

	GT580	GTX680	Change
Texture Engine	49.4 GT/s	128.8 GT/s	**2.6 x**
Floating-point operations	16 x 32 x 1.55 GHz	8 x 192 x 1.006 GHz	**1.94 x**
Integer multiplication, bit operations, type conversions	16 x 16 x 1.55 GHz	8 x 32 x 1.006 GHz	**0.65 x**
Shared Memory	48 KB	48 KB	**1**
Blocks per SM	8	16	**2**
Registers	32K per SM, 63 per thr.	64K per SM, 63 per thr.	**1**

Default approach

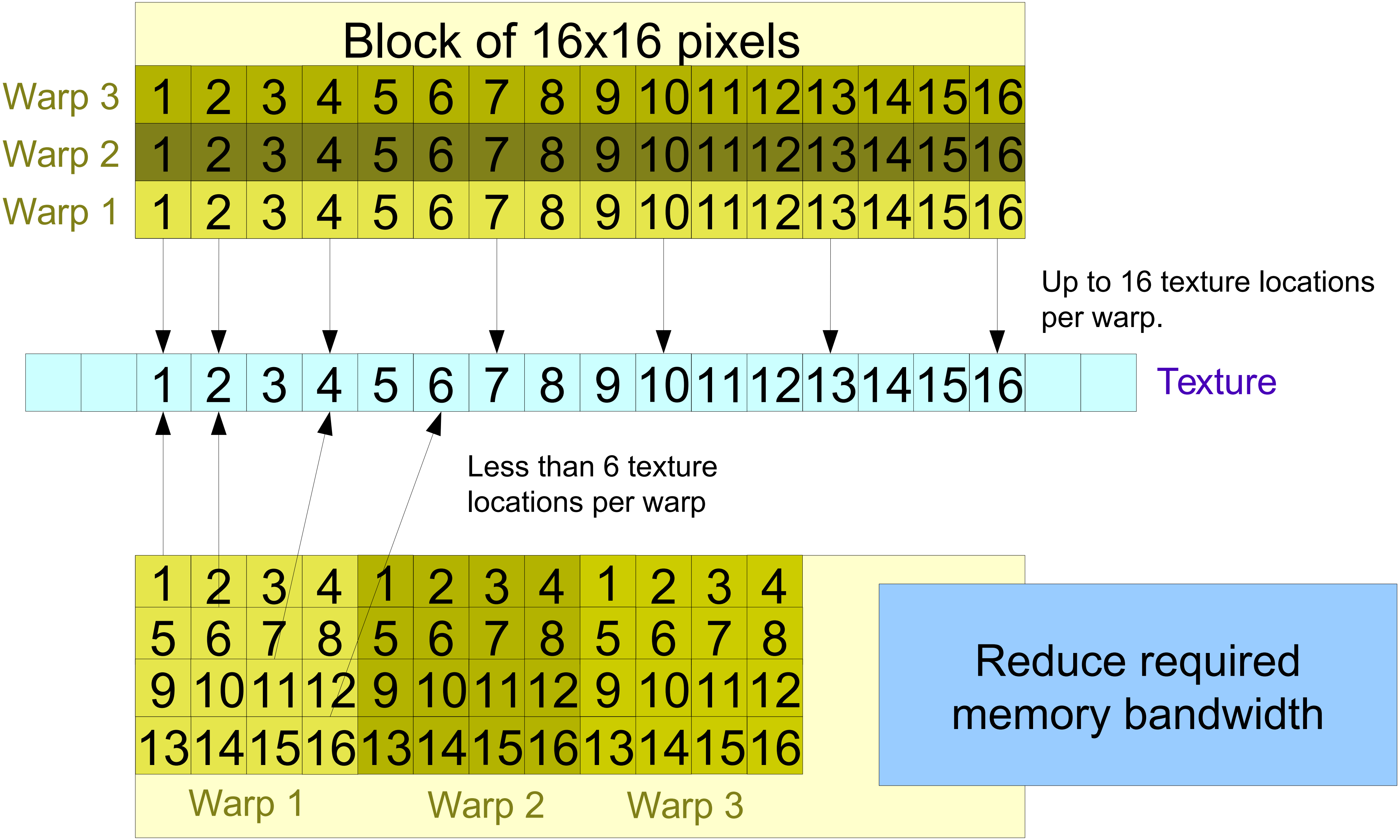


Block of 16x16 pixels

Texture Cache Hit Rate	89 %
Texture Throughput	79.3 GT/s
Theoretical Throughput	128.8 GT/s

1. Up to 16 bins are accessed per warp
2. All threads are accessing a single texture row

Optimizing the thread mapping



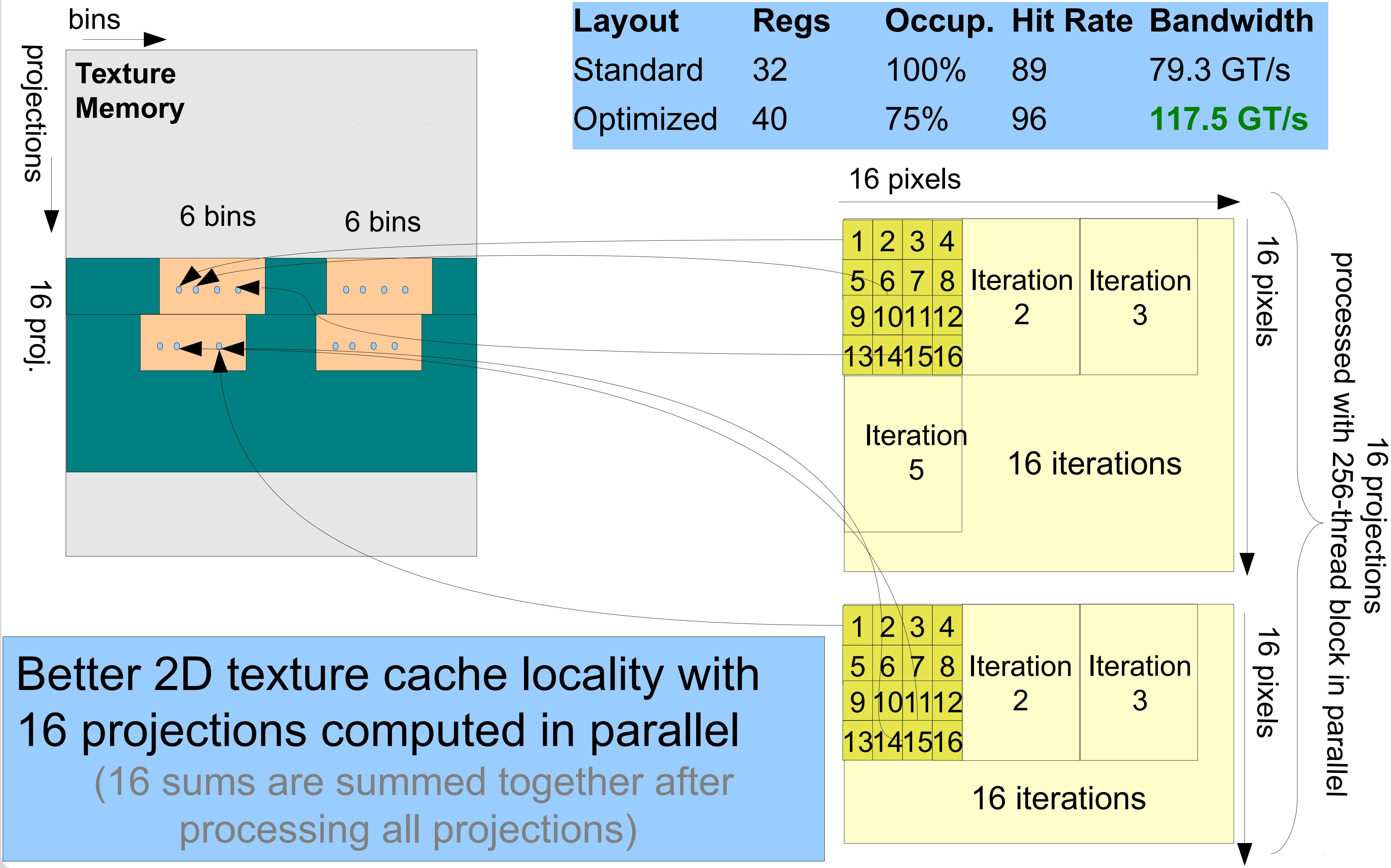
Up to 16 texture locations per warp.

Less than 6 texture locations per warp

Reduce required memory bandwidth

Using spatial locality

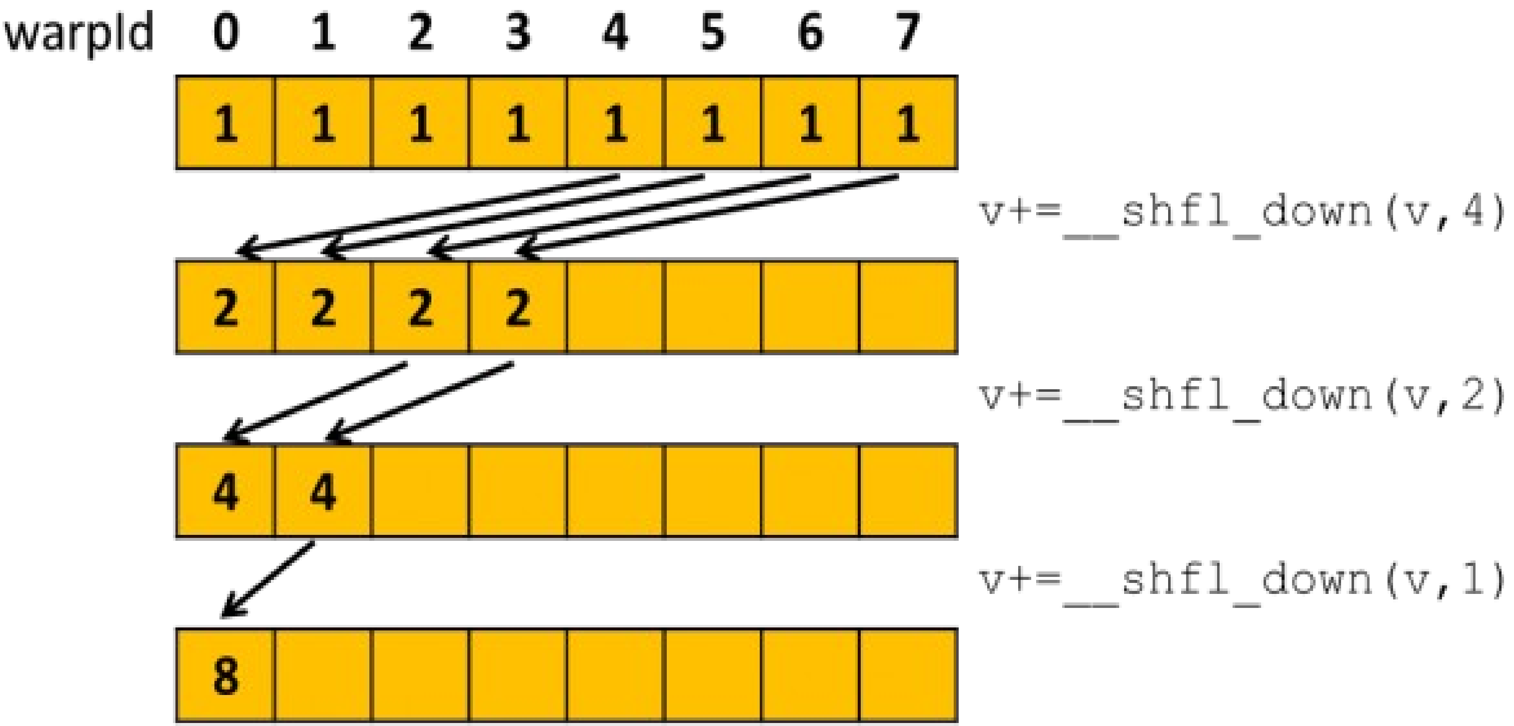
Layout	Regs	Occup.	Hit Rate	Bandwidth
Standard	32	100%	89	79.3 GT/s
Optimized	40	75%	96	**117.5 GT/s**



Better 2D texture cache locality with 16 projections computed in parallel

(16 sums are summed together after processing all projections)

Faster reduction with shuffle instruction

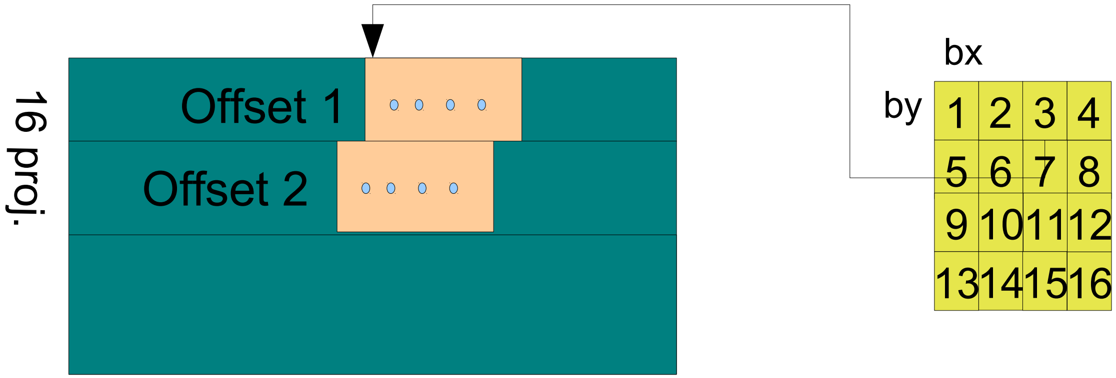


```
v+=__shfl_down(v,4)
v+=__shfl_down(v,2)
v+=__shfl_down(v,1)
```

Shuffle instruction introduced by Kepler architecture allows fast exchange of information between threads of the warp.

Oversampling approach on Kepler

Slow performance of integer and rounding operations makes Fermi oversampling algorithm slow.



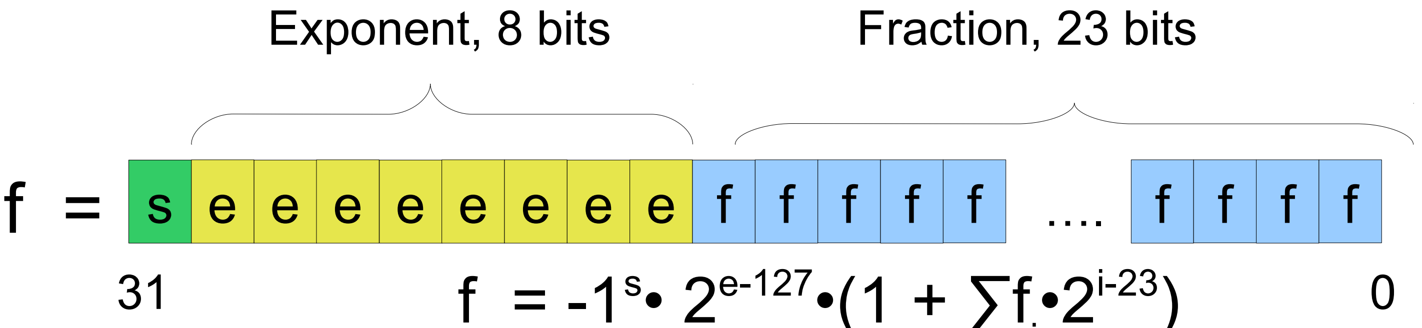
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

$$proj\_offset = \lfloor bx \bullet \cos(\alpha) - by \bullet \sin(\alpha) + correction(\alpha) \rfloor$$

On Fermi, for each block and projection we compute smallest-bin offset on the fly by each thread. On Kepler instead we can:

- Optimize rounding routine
- Pre-calculate and cache offsets

Looking for faster rounding on Kepler



Exponent, 8 bits

Fraction, 23 bits

f = | s | e | e | e | e | e | e | e | e | f | f | f | f | f | …. | f | f | f | f

31

0

$$f = -1^{s} \cdot 2^{e-127} \cdot (1 + \sum_i f_i \cdot 2^{i-23})$$

IEEE 754
single-precision
floating point number

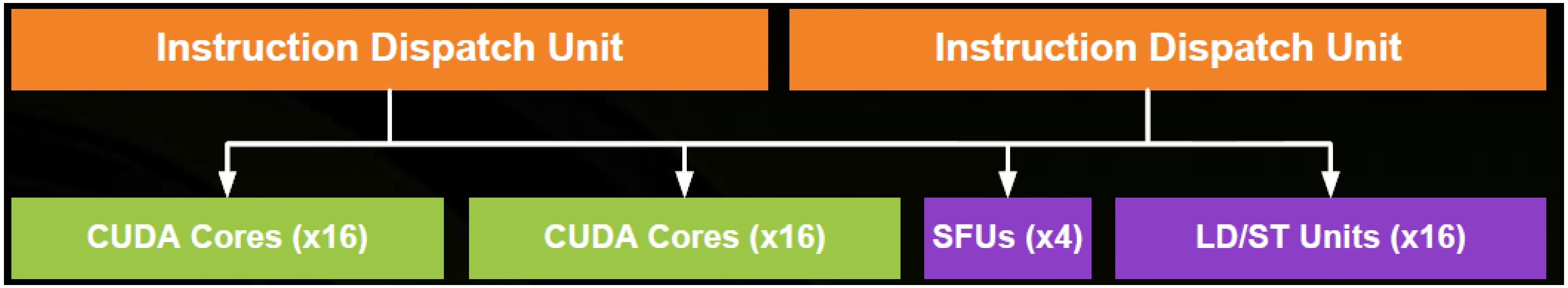
Only 23 significant positions, for small positive numbers:

$$f + 2^{23} = 2^{23} \cdot (1 + \sum_i f_i \cdot 2^{i-23})$$

i.e. no fractional part

$$\mathrm{round}(f) = f + 2^{23} - 2^{23}$$

$$(\mathrm{int})f = f + 2^{23} - \mathrm{0x4B000000}$$



Instruction Dispatch Unit

Instruction Dispatch Unit

CUDA Cores (x16)

CUDA Cores (x16)

SFUs (x4)

LD/ST Units (x16)

fp math

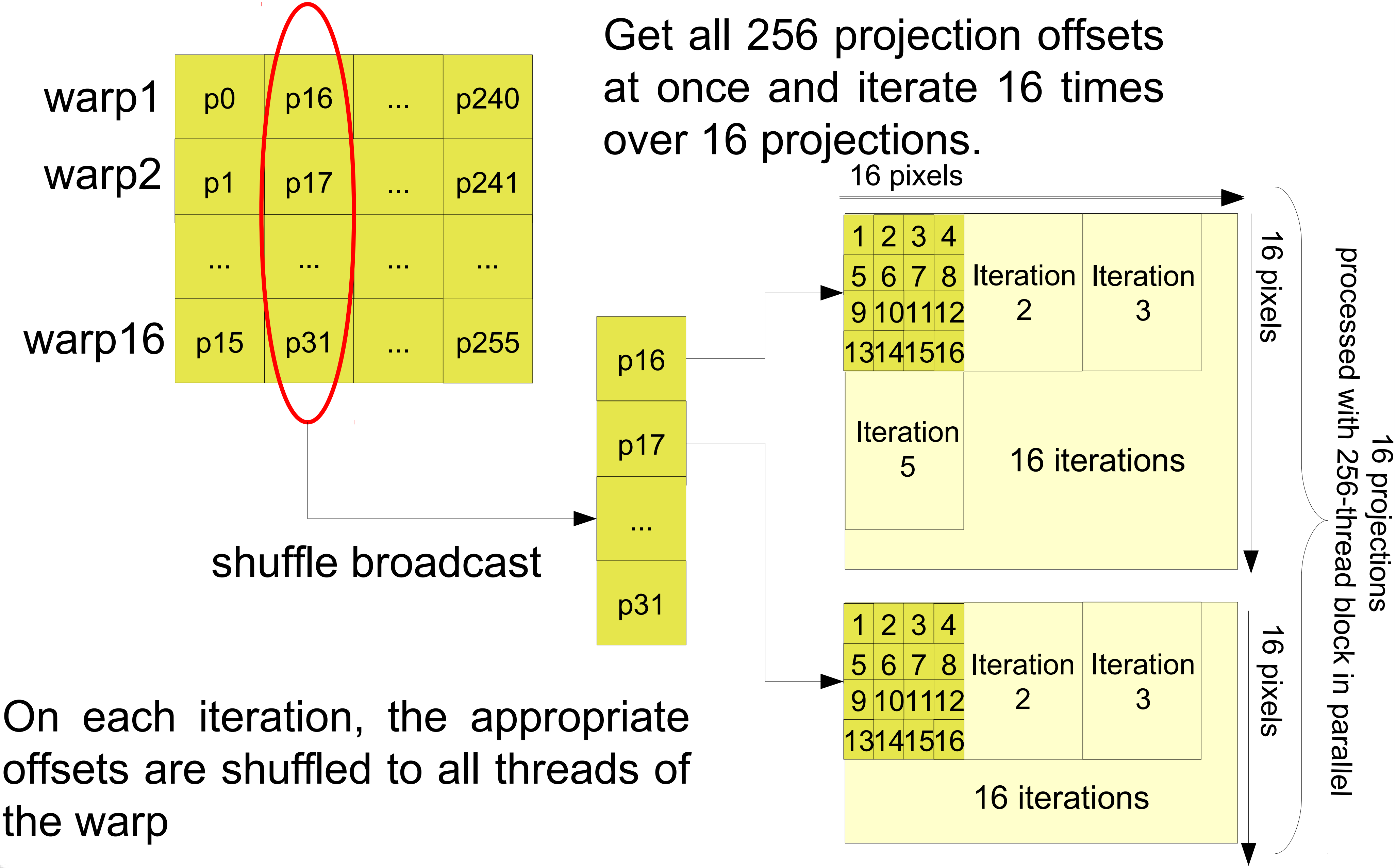
rounding

texture

We get faster rounding, but SFUs left unused and we got no speed up...

Reducing number of rounding operations

Get all 256 projection offsets at once and iterate 16 times over 16 projections.



On each iteration, the appropriate offsets are shuffled to all threads of the warp

Summary: 3 stages of oversampling

Work-group of 256 threads used to backproject area of 32x32 pixels from 256 projections

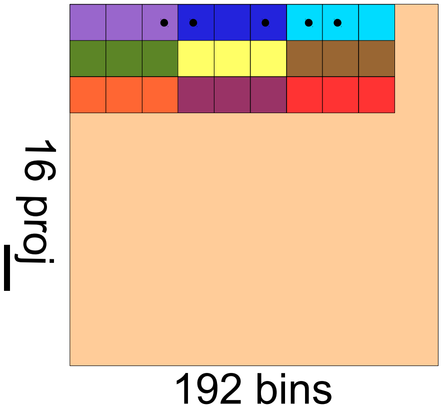
1

p0	p16	...	p240
p1	p17	...	p241
...	...	...	...
p15	p31	...	p255

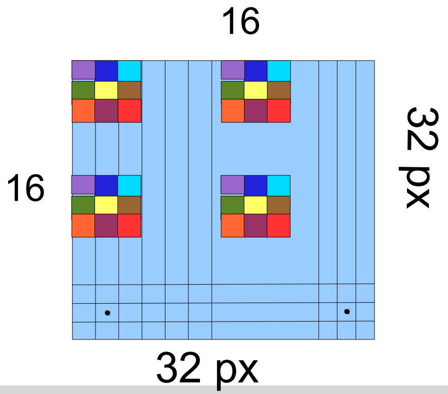
compute all offsets
work-items are mapped linearly to all projections.

2 *16 iterations (only 16 projections at once)*

cache data in shmem
warps are mapped to projections and individual work-items to its bins.



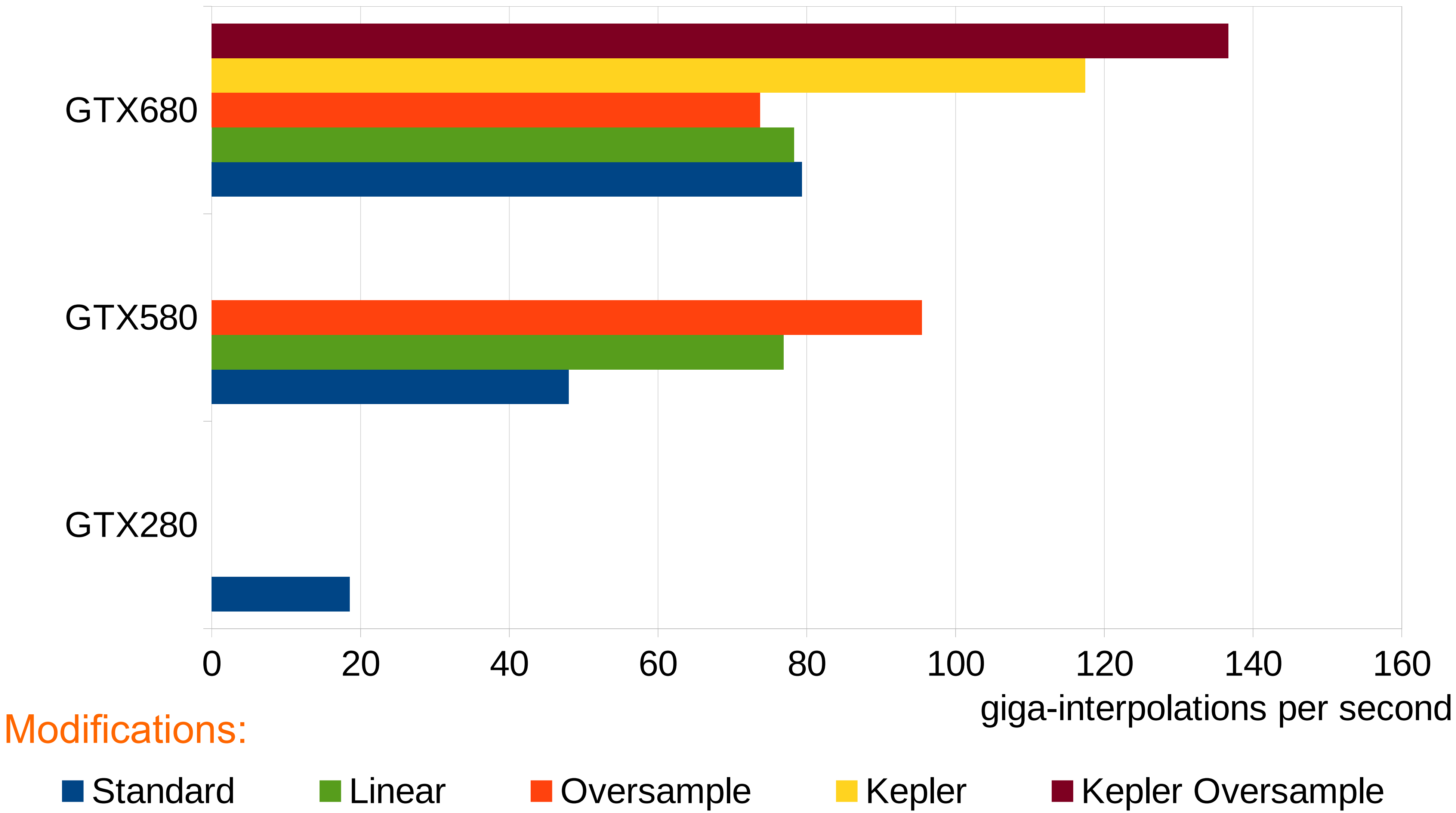
3 256 iterations each processing a single projection



interpolate pixels
work-items are mapped to area 16x16 pixels and proess 4 pixels at once

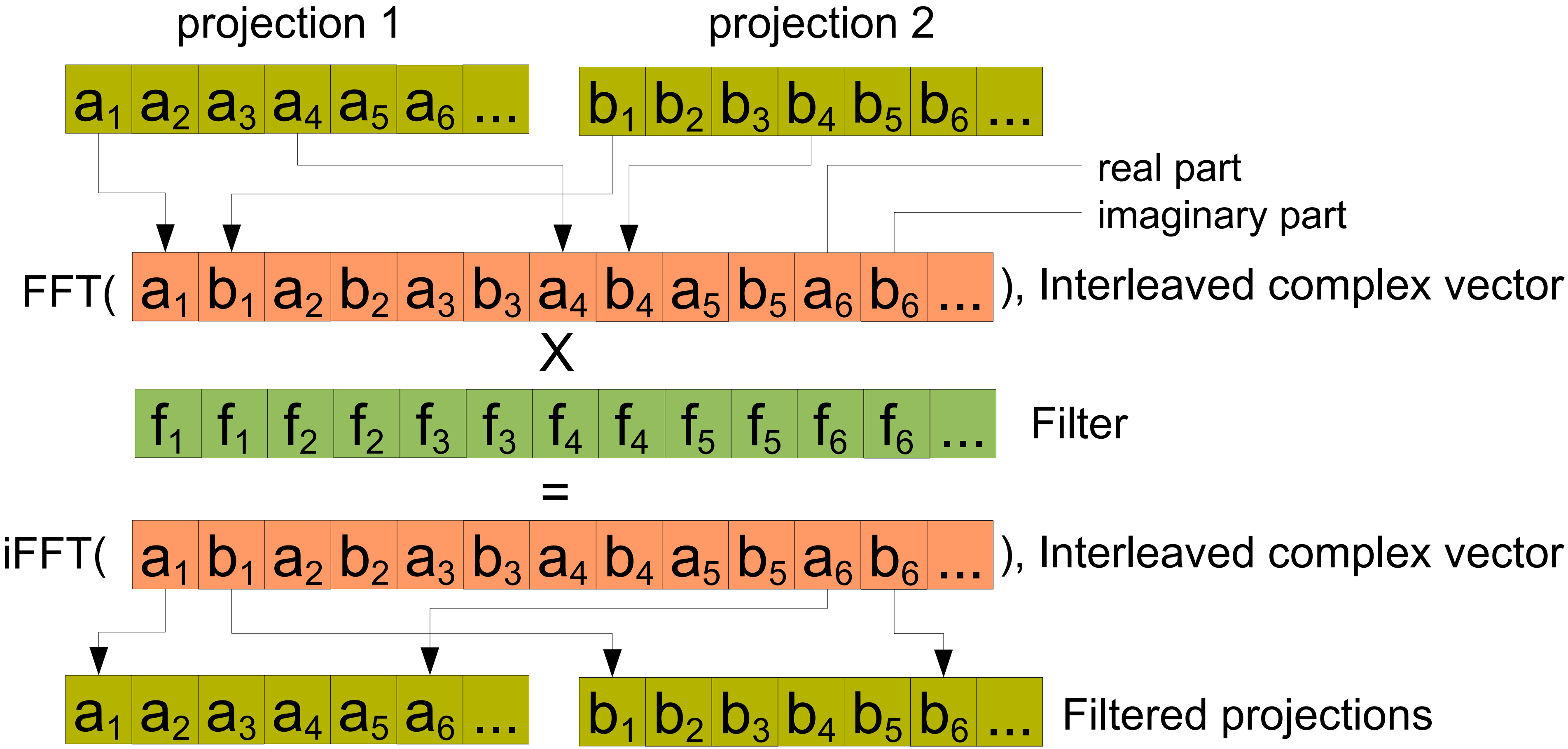
3 different mappings for optimal performance

Performance of Back Projection



Optimizing Filtering Step

FFT library is optimized for complex-to-complex transforms while we are dealing with real numbers.



projection 1

a_1 a_2 a_3 a_4 a_5 a_6 ...

projection 2

b_1 b_2 b_3 b_4 b_5 b_6 ...

real part

imaginary part

FFT(a_1 b_1 a_2 b_2 a_3 b_3 a_4 b_4 a_5 b_5 a_6 b_6 ...), Interleaved complex vector

X

f_1 f_1 f_2 f_2 f_3 f_3 f_4 f_4 f_5 f_5 f_6 f_6 ... Filter

=

iFFT(a_1 b_1 a_2 b_2 a_3 b_3 a_4 b_4 a_5 b_5 a_6 b_6 ...), Interleaved complex vector

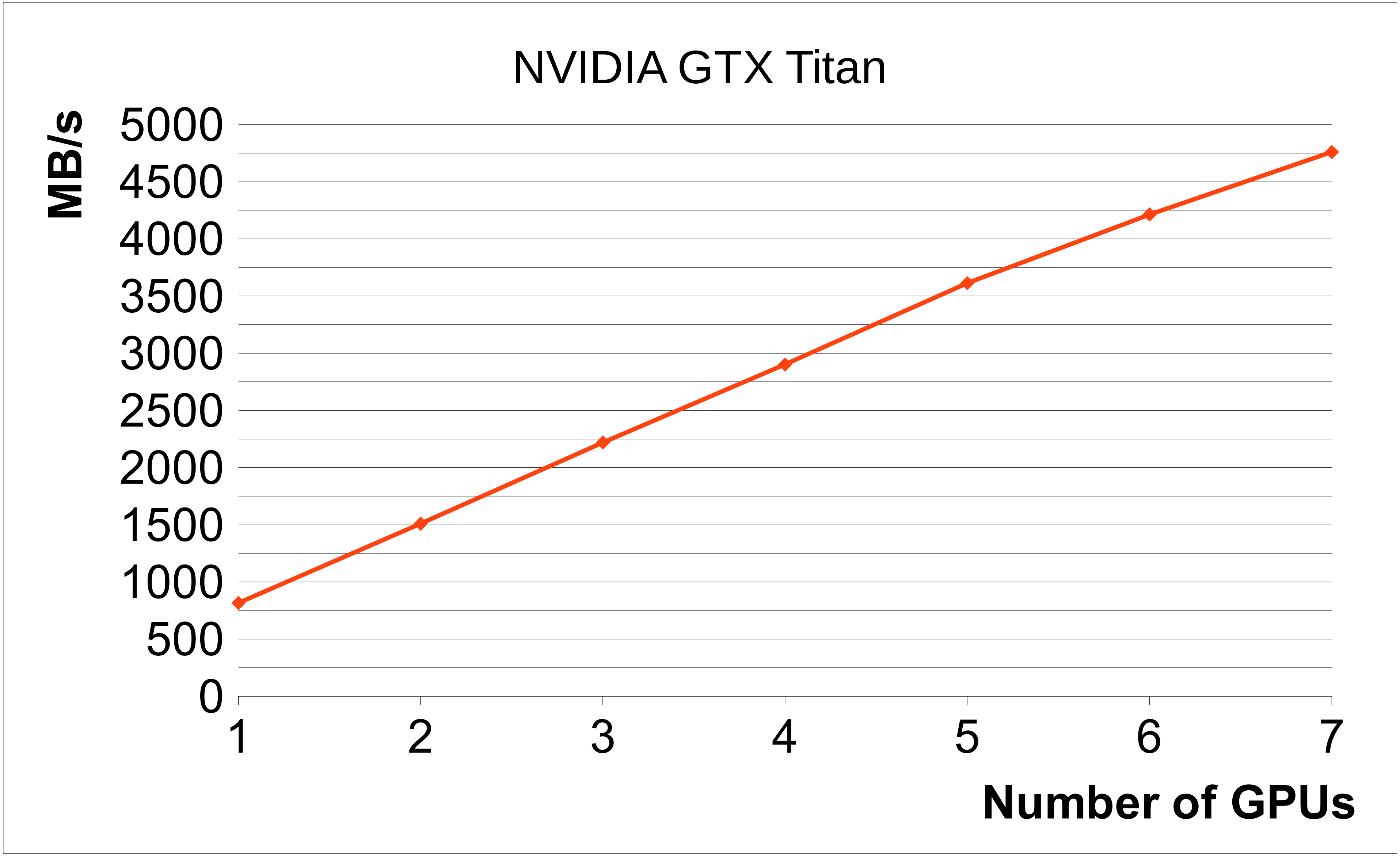
a_1 a_2 a_3 a_4 a_5 a_6 ...

b_1 b_2 b_3 b_4 b_5 b_6 ...

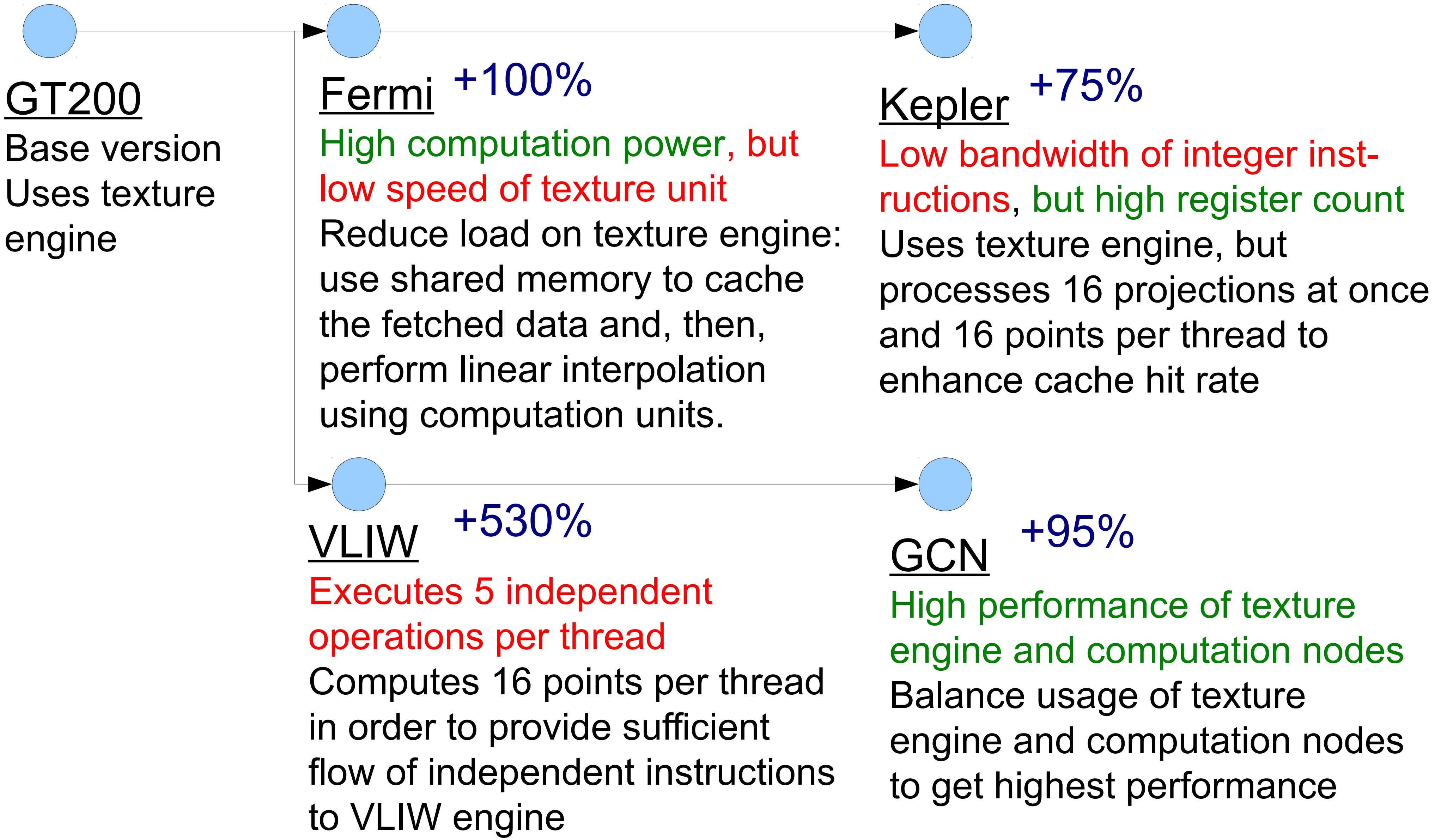
Filtered projections

- **Also**
  - Pad data to a size equal to the closest power of 2
  - Batched processing

Overall performance and scalability



Summary



GT200

Base version
Uses texture engine

Fermi +100%

High computation power, but low speed of texture unit

Reduce load on texture engine: use shared memory to cache the fetched data and, then, perform linear interpolation using computation units.

Kepler +75%

Low bandwidth of integer instructions, but high register count

Uses texture engine, but processes 16 projections at once and 16 points per thread to enhance cache hit rate

VLIW +530%

Executes 5 independent operations per thread

Computes 16 points per thread in order to provide sufficient flow of independent instructions to VLIW engine

GCN +95%

High performance of texture engine and computation nodes

Balance usage of texture engine and computation nodes to get highest performance