Tuesday 19th March, 2019
GPU Technology Conference 2019
San Jose, USA

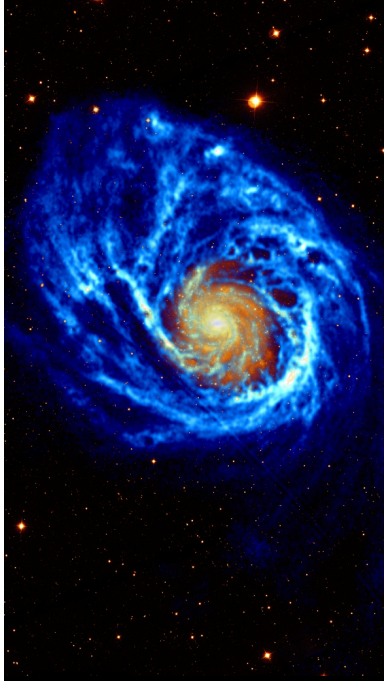
S9306

Extreme Signal-Processing Performance Using Tensor Cores Astronomical Imaging on GPUs

John Romein and Bram Veenboer

This talk consists of two parts. In the first part, we explain how we use Tensor Cores to obtain extreme signal-processing performance. In the second part of this talk, we explain how we solve the largest computational challenge in the imaging pipeline of modern radio telescopes.

ASTRON
Netherlands Institute for Radio Astronomy



Tensor Cores:
Signal Processing at Unprecedented Speeds

John Romein
ASTRON (Netherlands Institute for Radio Astronomy)

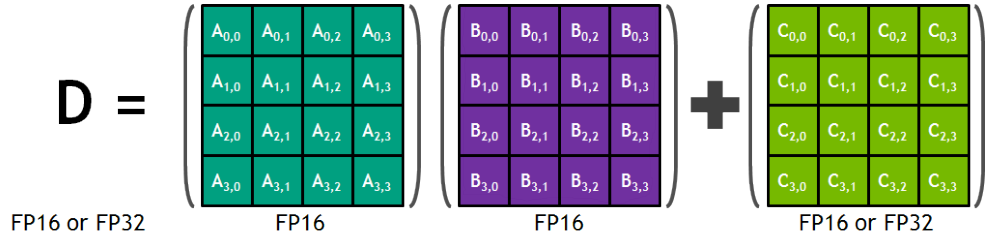
outline

- tensor cores
- complex numbers and matrix multiplications
- signal-processing algorithms
  - correlations
  - beam forming
  - ...

} analyze performance

tensor cores

- mixed-precision matrix multiplication hardware
  - Volta, Turing



- V100: peak 112 (!) TFLOPS
- designed for deep learning

how to use tensor cores

- libraries (cuBLAS, cutlass, ...) ✘
  - insufficient complex numbers support
- WMMA ✔
  - operates directly on 16x16 matrices (+ few more formats)
  - use in CUDA program

WMMA example

- warp performs 16x16 matrix multiplication

```
load_matrix_sync(a_frag, &a[…][…], K);
load_matrix_sync(b_frag, &b[…][…], N);
fill_fragment(c_frag, 0);
mma_sync(d_frag, a_frag, b_frag, c_frag);  // d=a*b+c
store_matrix_sync(&d[…][…], d_frag, …);
```

signal processing: complex numbers

- describes phase & amplitude of signal
- real and imaginary part (a_r, a_i)

- complex multiply-add = 4 real multiply-adds

c += ab

c_r += $a_r b_r$

c_r += $-a_i b_i$ –sign → no tensor core support

c_i += $a_r b_i$

c_i += $a_i b_r$

two complex array/matrix formats

1) split matrix

$r_{0,0}$	$r_{0,1}$	$r_{0,2}$	$r_{0,3}$
$r_{1,0}$	$r_{1,1}$	$r_{1,2}$	$r_{1,3}$
$r_{2,0}$	$r_{2,1}$	$r_{2,2}$	$r_{2,3}$
$r_{3,0}$	$r_{3,1}$	$r_{3,2}$	$r_{3,3}$

$i_{0,0}$	$i_{0,1}$	$i_{0,2}$	$i_{0,3}$
$r_{1,0}$	$i_{1,1}$	$i_{1,2}$	$i_{1,3}$
$i_{2,0}$	$i_{2,1}$	$i_{2,2}$	$i_{2,3}$
$i_{3,0}$	$i_{3,1}$	$i_{3,2}$	$i_{3,3}$

```
float real[4][4], imag[4][4];
```

2) interleaved

$r_{0,0}$	$i_{0,0}$	$r_{0,1}$	$i_{0,1}$	$r_{0,2}$	$i_{0,2}$	$r_{0,3}$	$i_{0,3}$
$r_{1,0}$	$r_{1,0}$	$r_{1,1}$	$i_{1,1}$	$r_{1,2}$	$i_{1,2}$	$r_{1,3}$	$i_{1,3}$
$r_{2,0}$	$i_{2,0}$	$r_{2,1}$	$i_{2,1}$	$r_{2,2}$	$i_{2,2}$	$r_{2,3}$	$i_{2,3}$
$r_{3,0}$	$i_{3,0}$	$r_{3,1}$	$i_{3,1}$	$r_{3,2}$	$i_{3,2}$	$r_{3,3}$	$i_{3,3}$

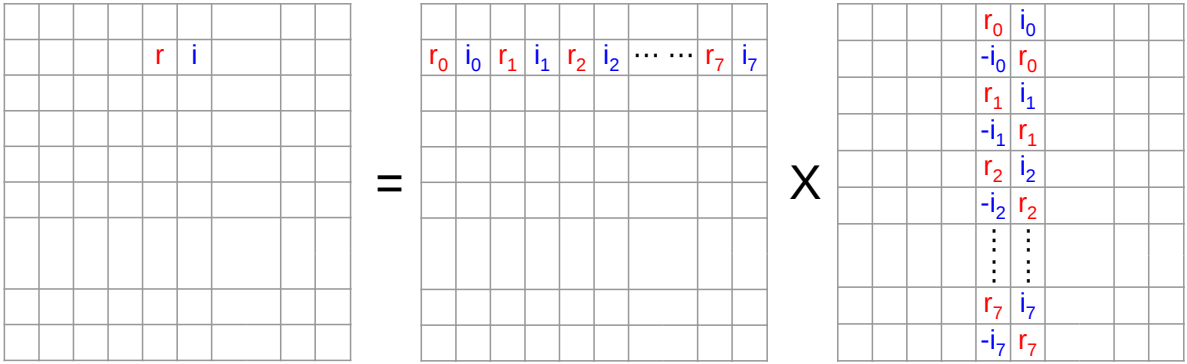
```
std::complex<float> matrix[4][4];
```

1) complex split matrices

$$[C]=[A][B] \rightarrow \begin{aligned} [C_r]&=[A_r][B_r]+[-A_i][B_i] \\ [C_i]&=[A_r][B_i]+[\ \ A_i][B_r] \end{aligned}$$

- maps well to tensor cores
  - negate A_i values

2) interleaved complex matrices



- reorder right matrix for tensor core use
  - duplicate/permute/negate entries

complex formats: split array vs. interleaved

- implemented both
- generally no big performance difference

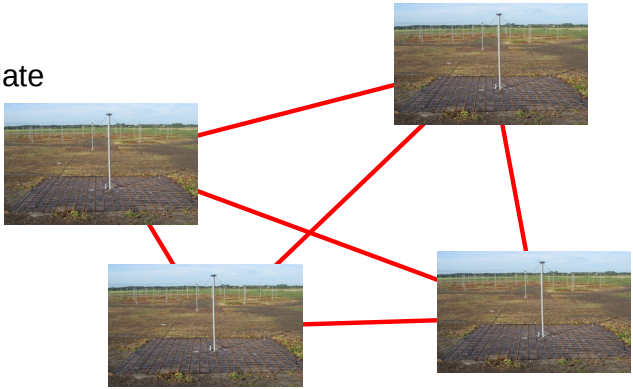
tensor cores for signal processing

- suitable if
  - input ≤ 16 bit ✔
  - algorithm translates to matrix-matrix multiplication ✔+✘

algorithm 1: correlations

correlations

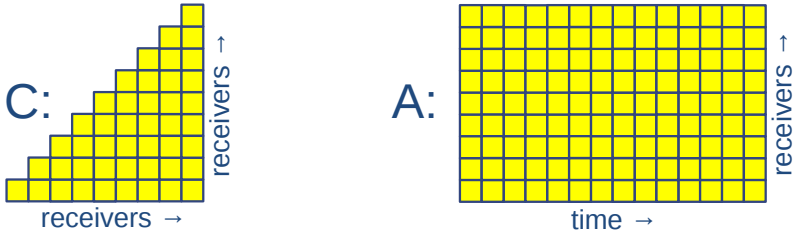
- combines telescope data
  - each pair: multiply & accumulate
  - *½r(r+1)* pairs



$$correlation_{recv_1,recv_2} = \sum_{time=0}^{integration\,time-1} sample_{recv_1,time} \times \overline{sample_{recv_2,time}}$$

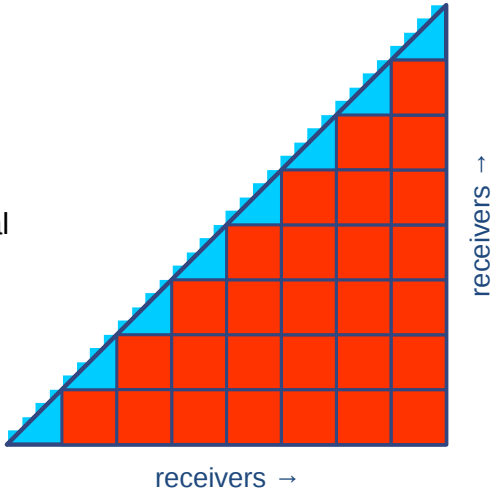
correlator computations

- $C \leftarrow A * A^H$
- $C = C^H \rightarrow$ compute & store triangle



work decomposition

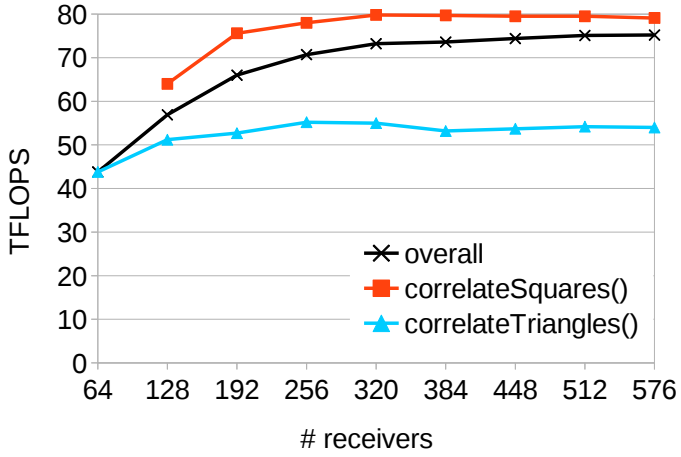
- computeSquares()
  - thread block: 64x64 receivers
  - warp: 32x16 receivers
- computeTriangles()
  - redundant computations above diagonal



correlator implementation

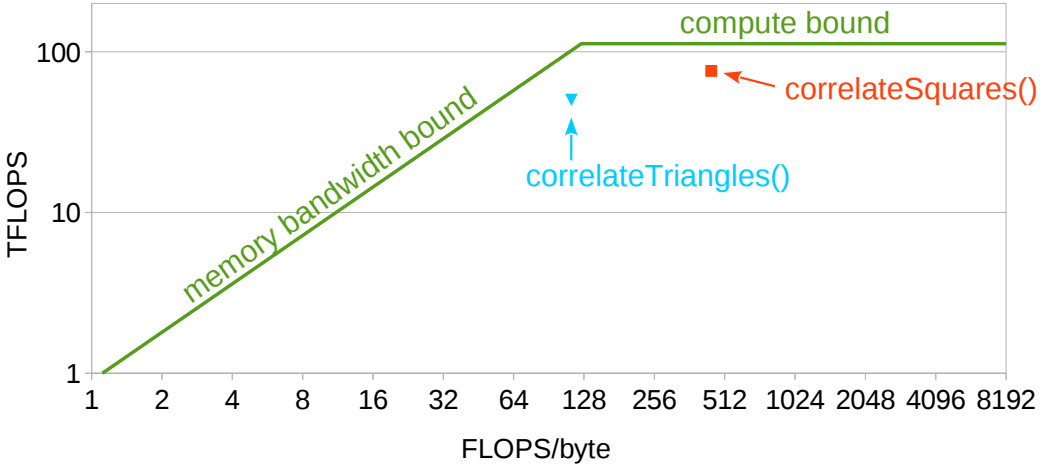
- cache input: L2 → shared mem → registers
  - fix –sign on the fly
- wmma::store_matrix_sync() cannot write to triangle
  - copy via shared mem, or
  - write accumulation registers directly (hack!)

correlator performance

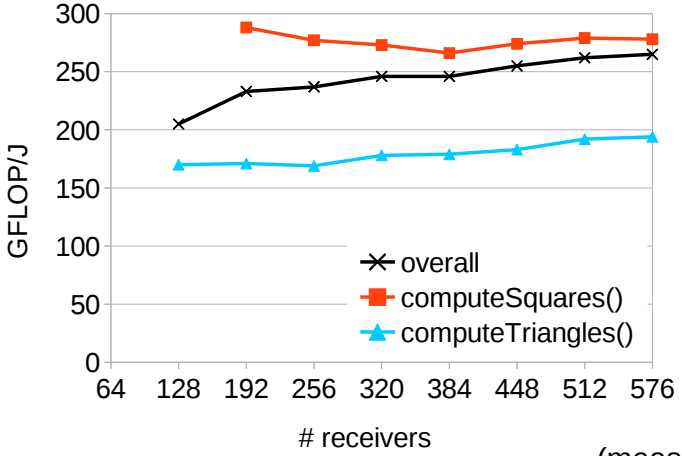


(measured on Tesla V100)

correlator roofline analysis

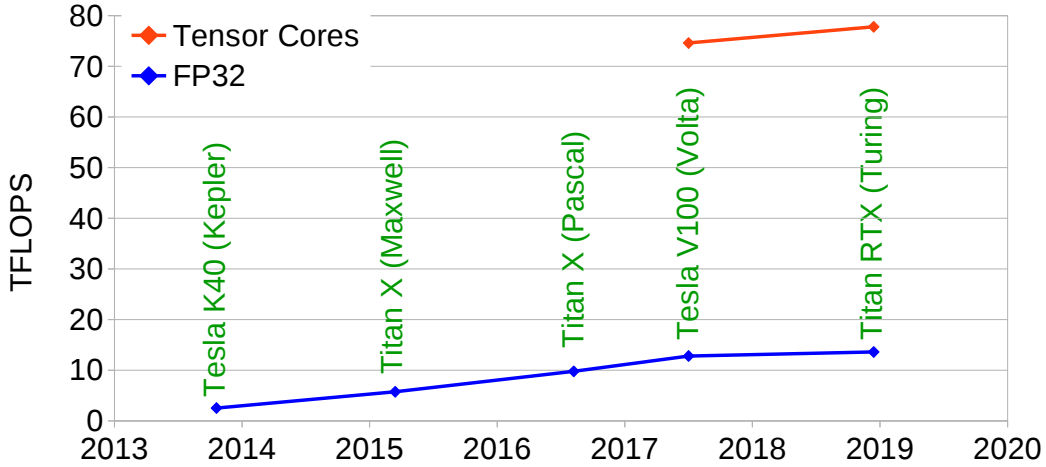


correlator energy efficiency



(measured on Titan RTX, not V100)

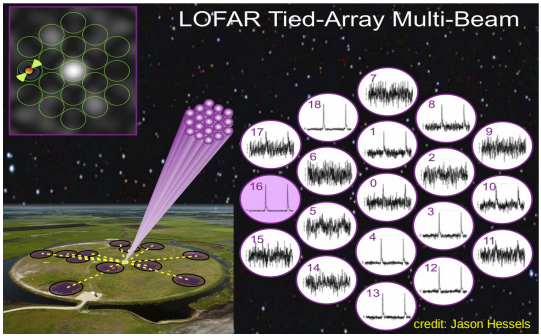
innovation beyond Moore's law



algorithm 2: beam forming

beam forming

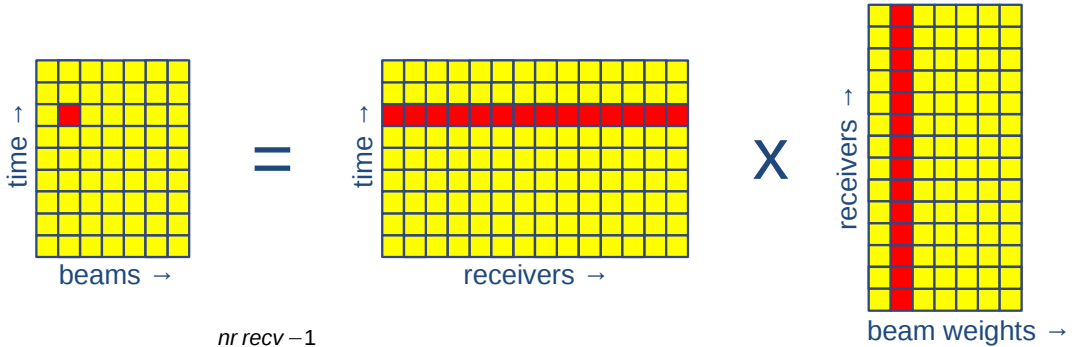
- increase sensitivity in particular direction
- (weighted) addition of signals



$$bfdata_{time,beam} = \sum_{recv=0}^{nr\,recv-1} samples_{time,recv}\, weights_{recv,beam}$$

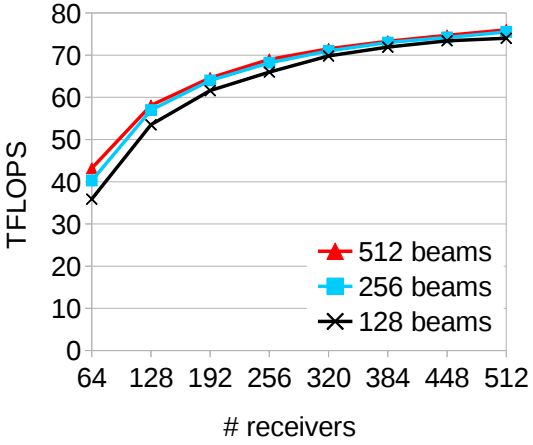
beam former implementation

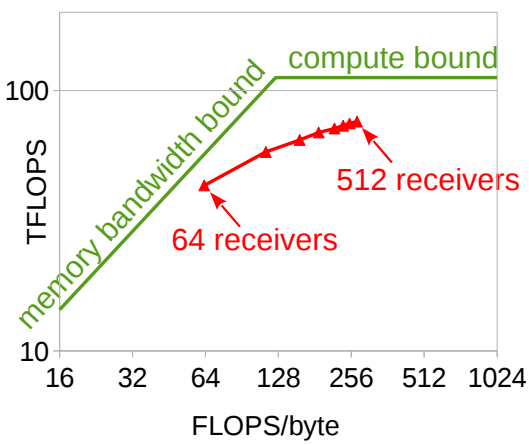
- multiple beams: complex matrix-matrix multiplication



$$bfdata_{time,beam} = \sum_{recv=0}^{nr\,recv-1} samples_{time,recv}\, weights_{recv,beam}$$

beam former performance and roofline analysis

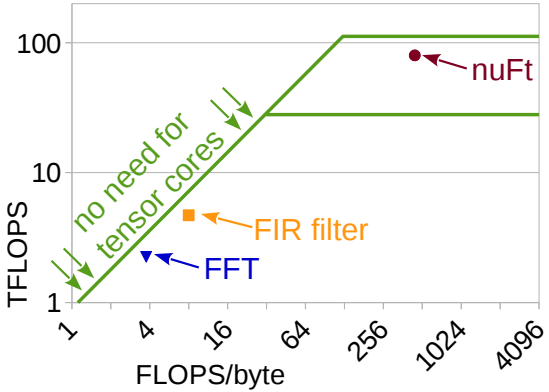




other algorithms

other signal-processing algorithms

- nonuniform Fourier transforms ✔
  - map well to complex matrix-matrix multiplication
  - ≤ 80 TFLOPS
- FIR filter ✘
  - matrix multiplication → many zeros
  - typically memory bandwidth bound
- FFT ✘
  - not a matrix multiplication
  - memory bandwidth bound



current / future work

- try further optimizations
  - correlator: near diagonal
  - beam forming: cublasLtMatmul() (CUDA 10.1)
- support *any* number of receivers/beams
- 8 bit, 4 bit

conclusions

- tensor cores for signal processing:
  - correlating
  - multi-beam forming
  - nonuniform Fourier transforms

  } matrix-matrix multiplication
- unprecedented performance (≤ ~75 TFLOPS, ≤ 6x)

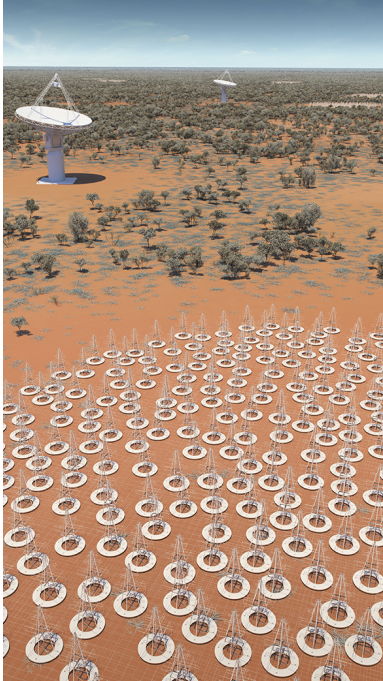
This work is funded by the European Union under grant no. H2020-FETHPC-754304 (DEEP-EST).

DEEP Projects

Tuesday 19th March, 2019
GPU Technology Conference 2019
San Jose, USA

Astronomical Imaging on GPUs

Bram Veenboer



ASTRON
Netherlands Institute for Radio Astronomy

Outline

- Introduction:
  - Interferometry
  - The **Image-Domain Gridding** algorithm
  - Performance analysis
- Analysis and optimization:
  - **Gridder kernel**
  - Imaging application
- Performance and energy-efficiency comparison
- Results in context of **Square Kilometre Array**
- Summary

Introduction to radio astronomy

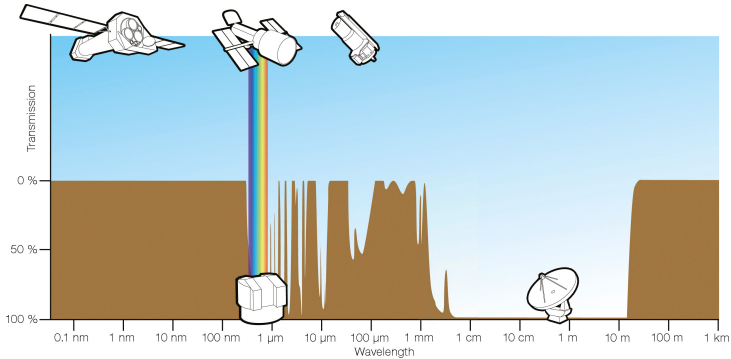


Image credits: NRAO

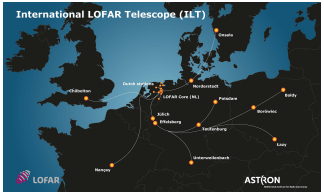
- Observe the sky at **radio wavelengths** $\longrightarrow$ **map of radio sources**
- Dish-based telescopes: VLA, ALMA, MeerKAT, SKA-1 Mid

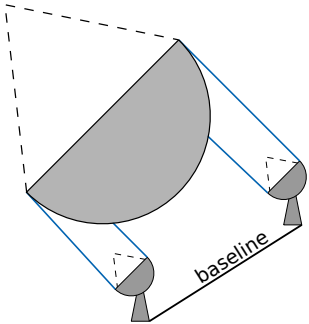
- Size of the telescope is proportional to the wavelength
- Use **array of antennas** for low frequencies: (LOFAR, MWA, SKA-1 Low)

ASTRON

Radio telescope: astronomical interferometer

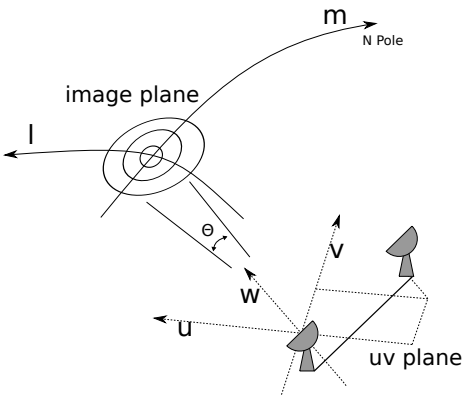
- Interferometer: array of seperate telescopes
- Interferometry: combine the signals from seperate radio telescopes
  - Resolution similar to one very large dish

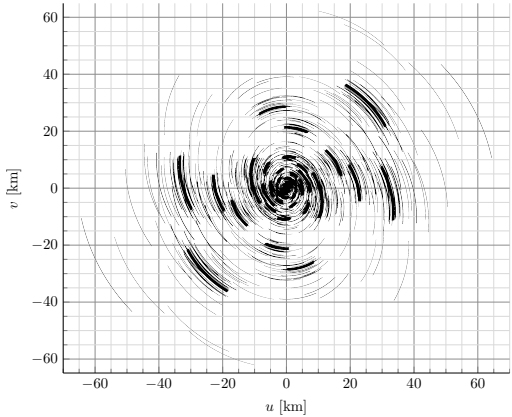




ASTRON

Interferometry theory

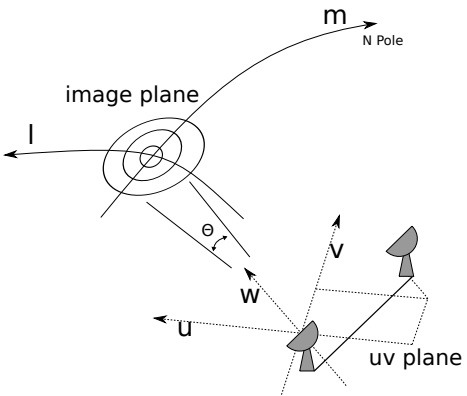


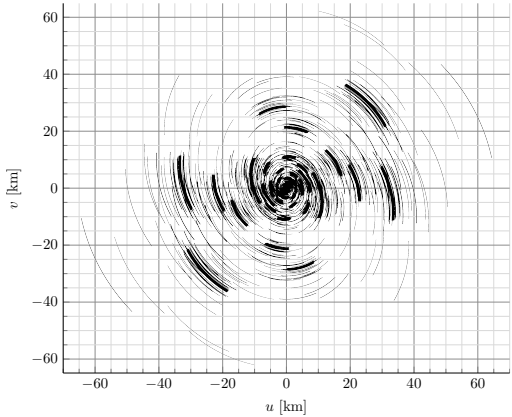


- Sampling of the ‘uv-plane’: ‘visibilities’
- Earth-rotation synthesis

ASTRON

Interferometry theory



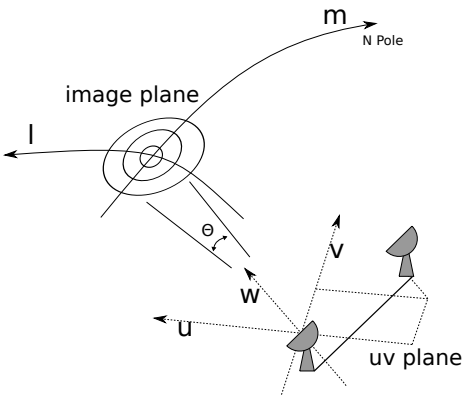


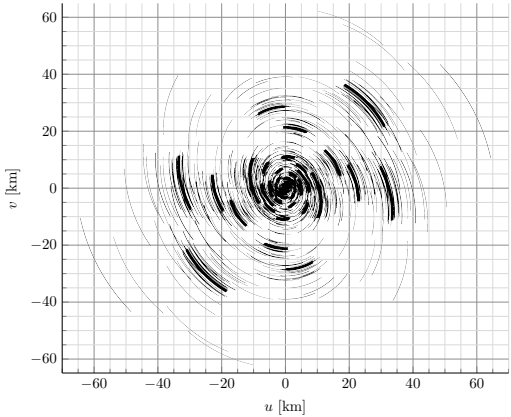
- Sampling of the ‘uv-plane’: ‘visibilities’
- Earth-rotation synthesis

- Visibilities $\xrightarrow{DFT}$ image

ASTRON

Interferometry theory





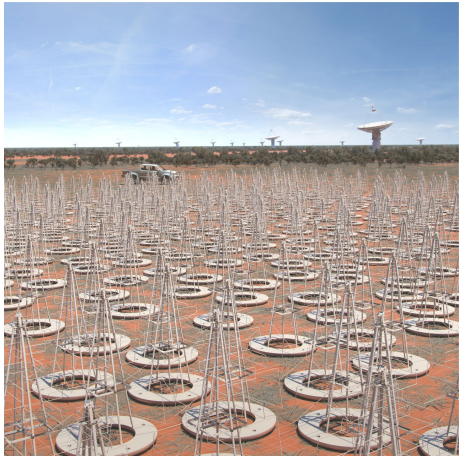
- Sampling of the ‘uv-plane’: ‘visibilities’
- Earth-rotation synthesis

- Visibilities $\xrightarrow{DFT}$ image
- Visibilities $\xrightarrow{gridding}$ grid $\xrightarrow{FFT}$ image

ASTRON

Square Kilometre Array

SKA1 Low, Australia

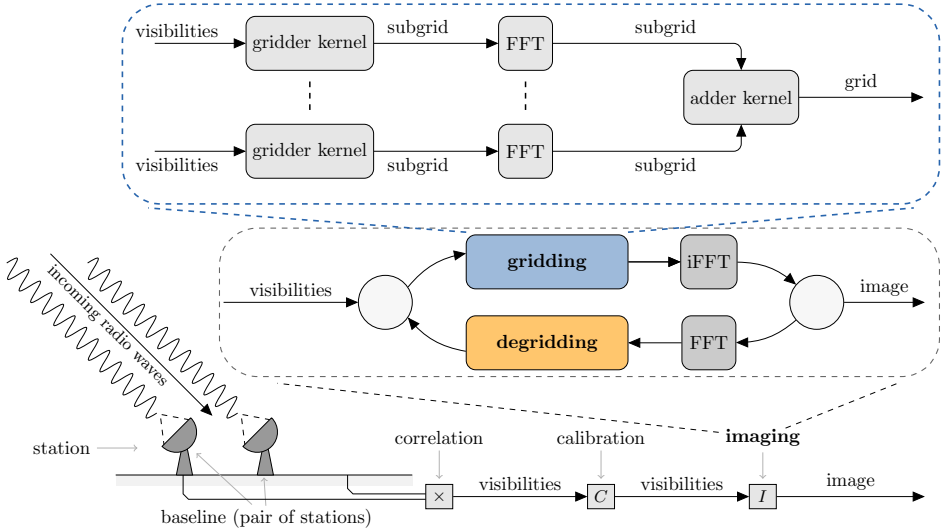


SKA1 Mid, Africa

- Data rates up to $\approx$ 10.000.000.000 visibilities/second

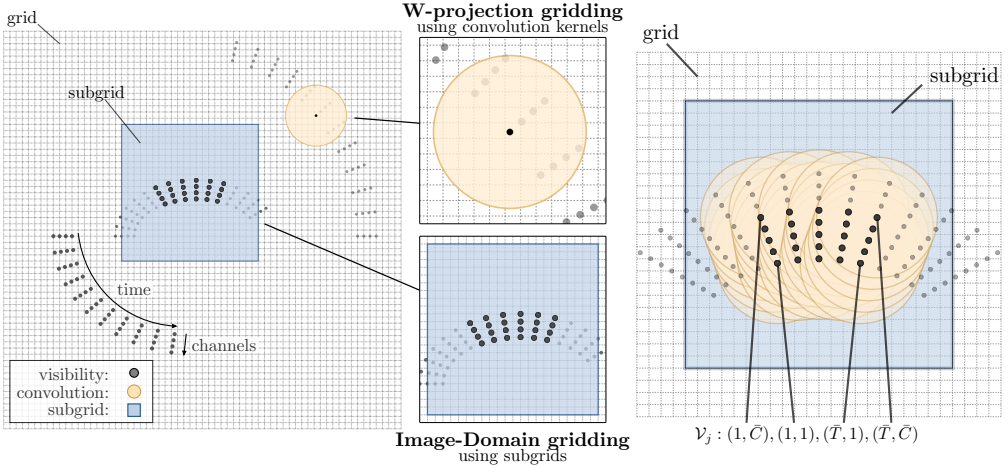
ASTRON

Sky-image creation using Image-Domain gridding



ASTRON

W-projection gridding and Image-Domain gridding



See "Image Domain Gridding: a fast method for convolutional resampling of visibilities, S. van der Tol et al., A&A 2018" and "Image-Domain Gridding on Graphics Processors, B. Veenboer et al., IPDPS 2017" for details.

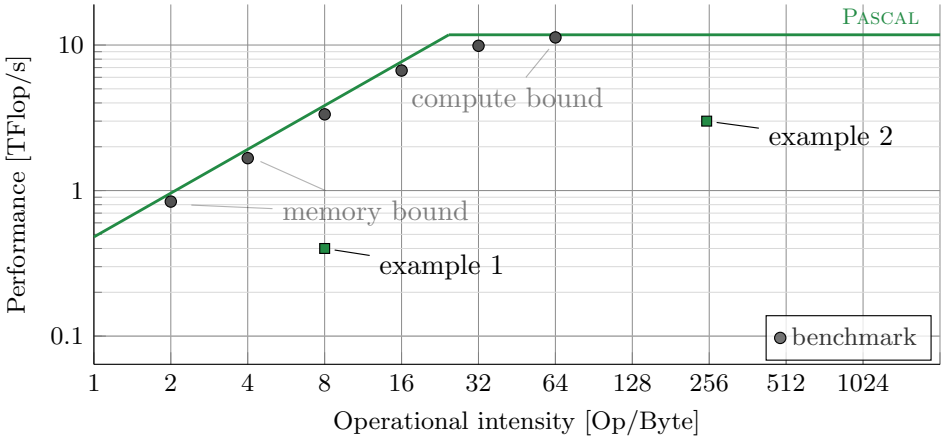
ASTRON

Image-Domain Gridding algorithm

```
1  for s = 1...S do                                   // subgrids mapped to thread blocks
2      complex<float> subgrid[P][N×N];
3      for i = 1...N×N do                                    // pixels mapped to threads
4          float offset = compute_offset(s, i);
5          for t = 1...T do
6              float index = compute_index(s, i, t);
7              for c = 1...C do
8                  float scale = scales[c];
9                  float phase = offset - (index × scale);                      // 1 fma
10                 complex<float> phasor = {cos(phase), sin(phase)};            // 1 cis
11                 for p = 1...P do                                  // 4 polarizations
12                     complex<float> visibility = visibilities[t][c][p];       // load 8 bytes
13                     subgrid[p][i] += cmul(phasor, visibility);               // 4 fma
14                 end
15             end
16         end
17     end
18     apply_corrections(subgrid);                              // aterm, taper, iFFT
19     store(subgrid);
20 end
```


ASTRON

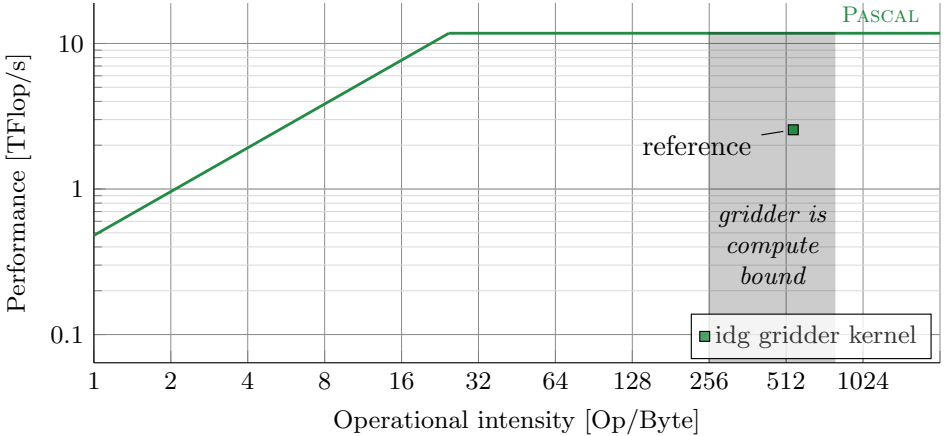
Roofline analysis



- Operational intensity gives upper bound on performance

ASTRON

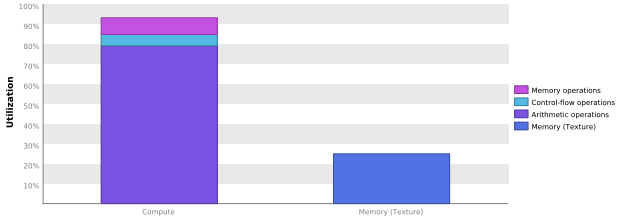
Kernel v1 (reference) 1/2



- Throughput: 70 Mvis/s (number of visibilities processed per second)

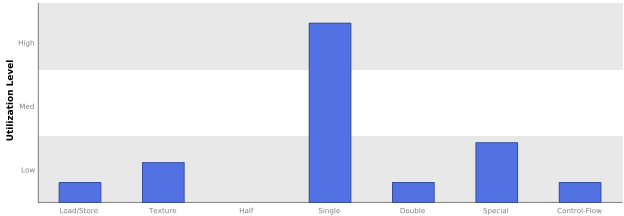
ASTRON

Kernel v1 (reference) 2/2



- High compute utilization
- Low memory utilization

$\longrightarrow$ compute bound



- High FPU utilization
- Low performance?
- What instructions are executed?

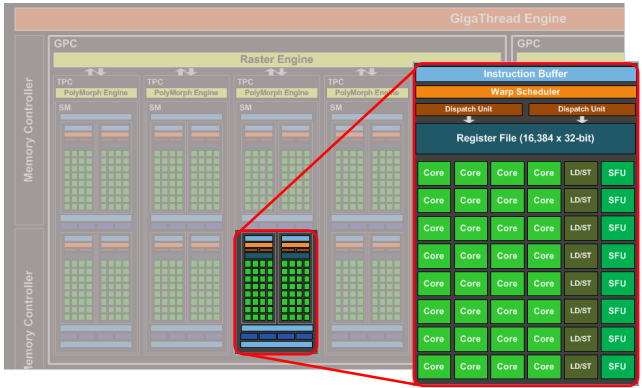
ASTRON

Image-Domain Gridding algorithm

```
1  for s = 1...S do                                   // subgrids mapped to thread blocks
2      complex<float> subgrid[P][N×N];
3      for i = 1...N×N do                                    // pixels mapped to threads
4          float offset = compute_offset(s, i);
5          for t = 1...T do
6              float index = compute_index(s, i, t);
7              for c = 1...C do
8                  float scale = scales[c];
9                  float phase = offset - (index × scale);                      // 1 fma
10                 complex<float> phasor = {cos(phase), sin(phase)};            // 1 cis
11                 for p = 1...P do                                  // 4 polarizations
12                     complex<float> visibility = visibilities[t][c][p];       // load 8 bytes
13                     subgrid[p][i] += cmul(phasor, visibility);               // 4 fma
14                 end
15             end
16         end
17     end
18     apply_corrections(subgrid);                              // aterm, taper, iFFT
19     store(subgrid);
20 end
```


ASTRON

Special Function Units (SFUs) to evaluate sine/cosine



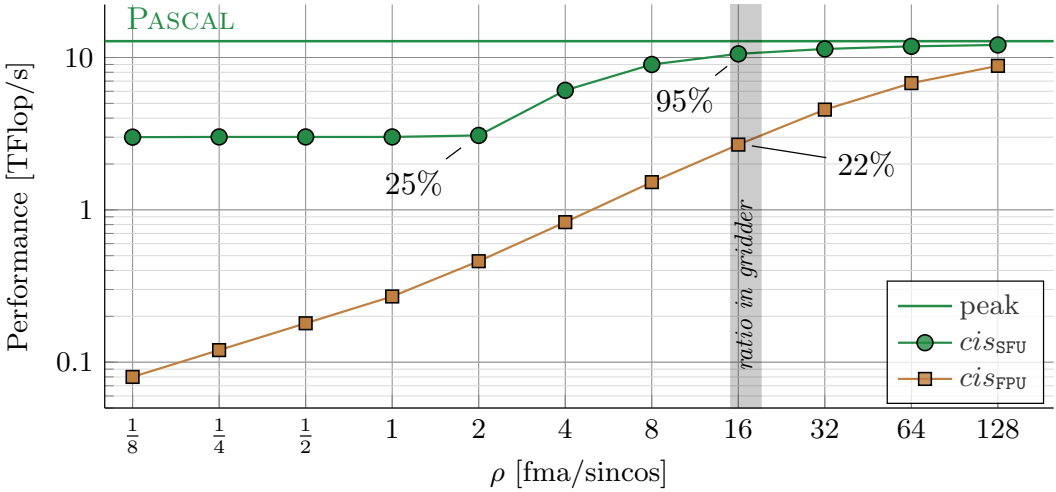
Compile using $-use\_fast\_math$, or specify explicitely:

b.x = cos(a.x) $\rightarrow$ asm(cos.approx.f32 %0, %1; : =f(b.x) : f(a.x));

b.y = sin(a.x) $\rightarrow$ asm(sin.approx.f32 %0, %1; : =f(b.y) : f(a.x));

ASTRON

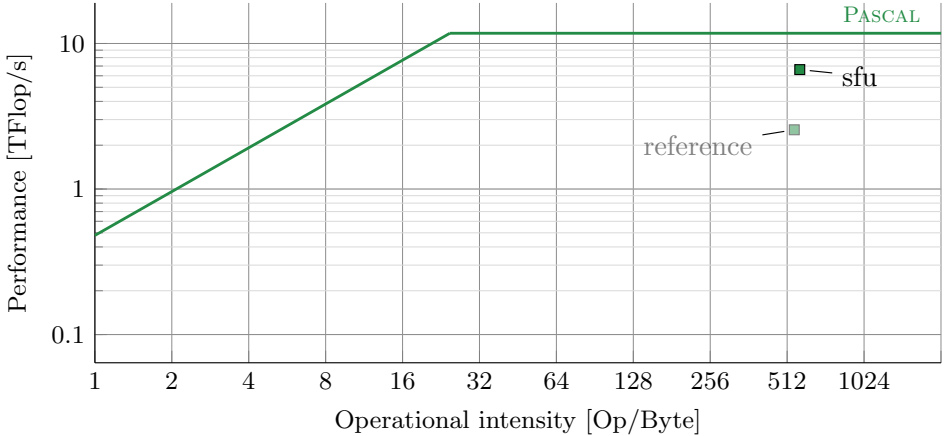
Peformance for instruction mix FMA + sine/cosine



- Using SFUs, FMAs and sine/cosine are computed simultaneously

ASTRON

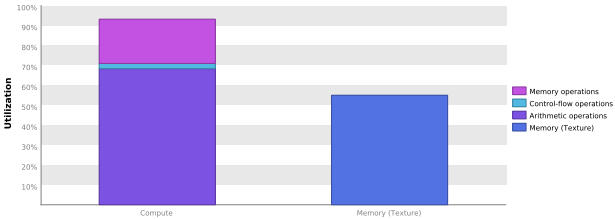
Kernel v2 (sfu) 1/2



- 2.5× performance increase, but still below the peak (180 MVis/s)

ASTRON

Kernel v2 (sfu) 2/2



- Many memory operations
- Memory utilization medium?

$\longrightarrow$ optimize memory acces pattern

Unified Cache

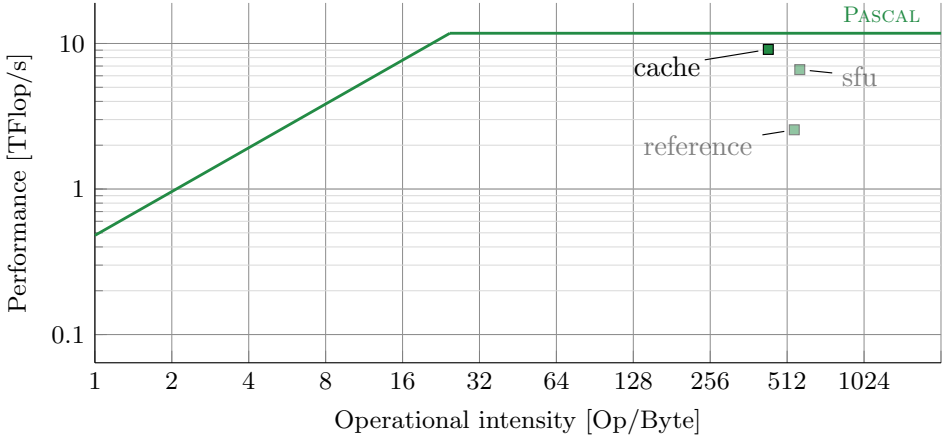
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	35040092162	3,176.086 GB/s	
Global Stores	28508160	5.66 GB/s	
Texture Reads	8760023040	1,739.156 GB/s	
Unified Total	43828623362	4,920.901 GB/s	Idle Low Medium High Max

- Many cache hits

$\longrightarrow$ cache input data explicitely using **shared memory**

ASTRON

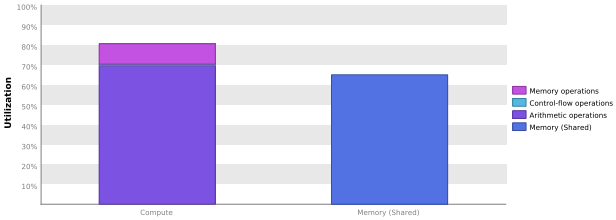
Kernel v3 (cache) 1/3



- Getting close to peak (70%, 250 MVis/s)

ASTRON

Kernel v3 (cache) 2/3



Shared Memory

Shared Loads	4789370880	4,457.865 GB/s	
Shared Stores	118514880	110.312 GB/s	
Shared Total	4907885760	4,568.176 GB/s	Idle Low Medium High Max

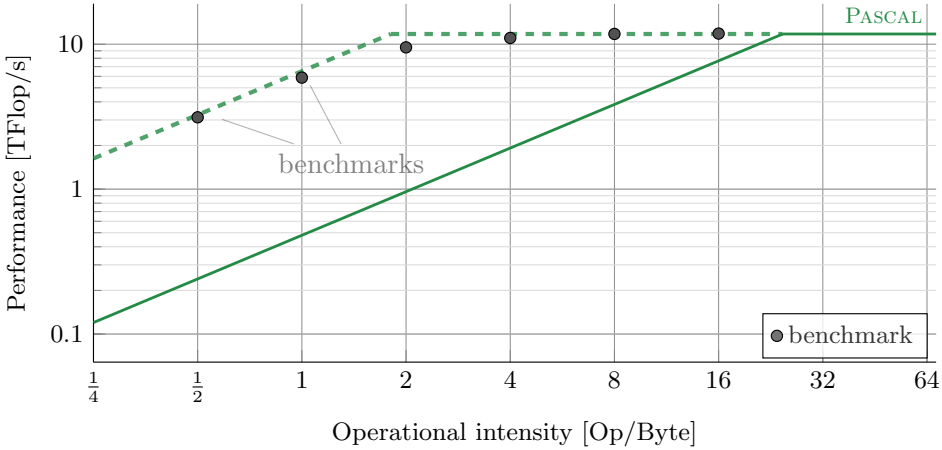
- Memory utilization texture → shared
- Memory operation overhead reduced from 20% to 10%

- High shared memory bandwidth

→ more analyis is needed...

ASTRON

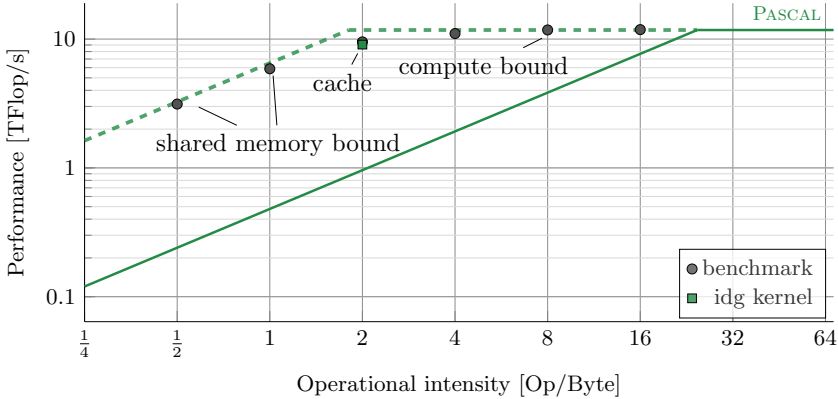
Roofline analysis



- Add additional roof for **shared memory**

ASTRON

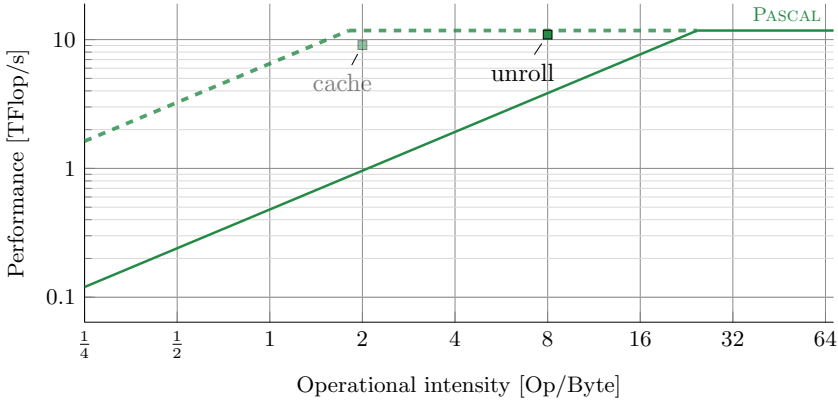
Kernel v3 (cache) 3/3



- Close to (measured) shared memory roof
- Increase operational intensity: every thread computes multiple pixels

ASTRON

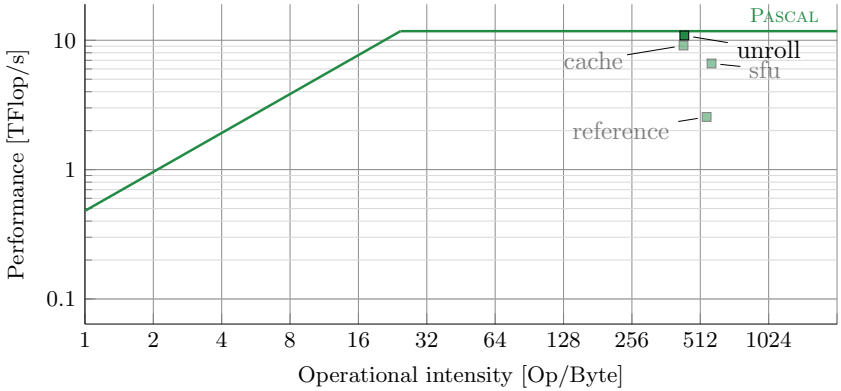
Kernel v4 (unroll) 1/3



- The new kernel is not bound by shared memory bandwidth

ASTRON

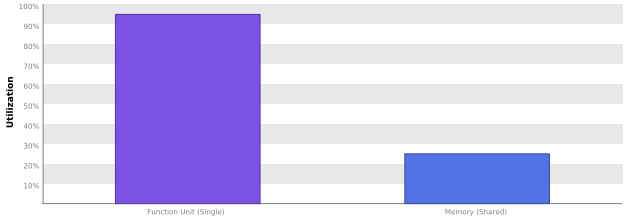
Kernel v4 (unroll) 2/3



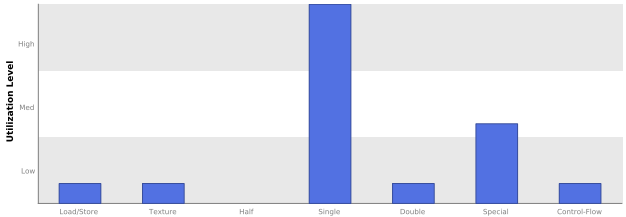
- The kernel is very close to the theoretical peak (90%, 300 Mvis/s)

ASTRON

Kernel v4 (unroll) 3/3



- Very high FPU utilization
- Low memory utilization



- FPU usage max

$\longrightarrow$ **gridder done!**

ASTRON

Kernel optimization summary

General optimization steps:

1. Measure (floating-point) performance and memory bandwidth
2. Apply roofline model to assess potential gains
3. Find optimization opportunities by profiling
4. Apply optimizations accordingly

Kernel optimization summary

General optimization steps:

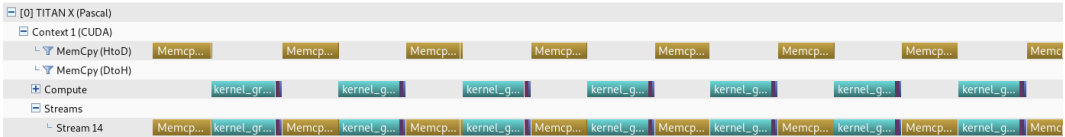
1. Measure (floating-point) performance and memory bandwidth
2. Apply roofline model to assess potential gains
3. Find optimization opportunities by profiling
4. Apply optimizations accordingly

In case of the gridder kernel:

- Reference: 70 MVis/s
- Use Special Function Units: 180 MVis/s (2.6×)
- Use shared memory cache: 250 MVis/s (1.4×)
- Increase operational intensity: 300 MVis/s (1.2×)

I/O optimization

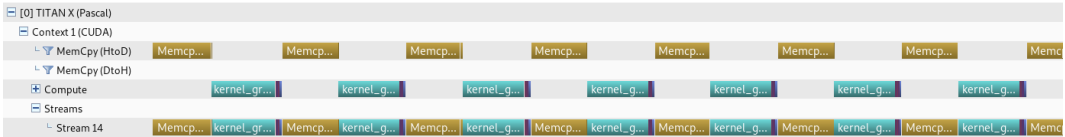
- Kernel throughput $\neq$ application throughput
- Data transfers can not be ignored:



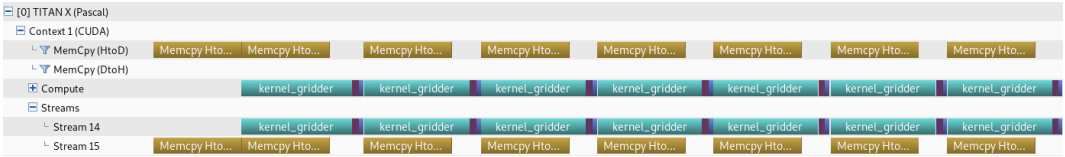
ASTRON

I/O optimization

- Kernel throughput $\neq$ application throughput
- Data transfers can not be ignored:



- Overlap PCIe transfers and compute:



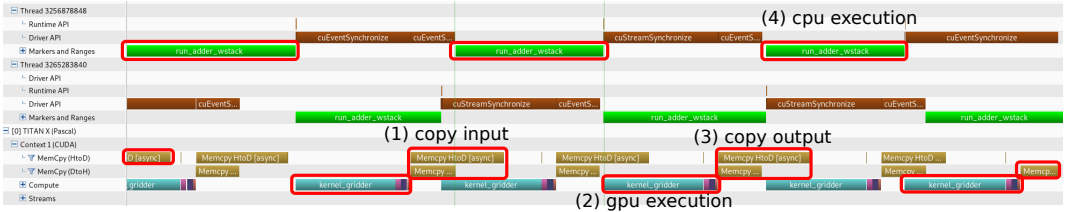
ASTRON

Creating large images: using CPU+GPU

- Subgrids are added onto a larger grid
- The grid might not fit in GPU memory (e.g. 40k×40k ≈ 48 GB)

Creating large images: using CPU+GPU

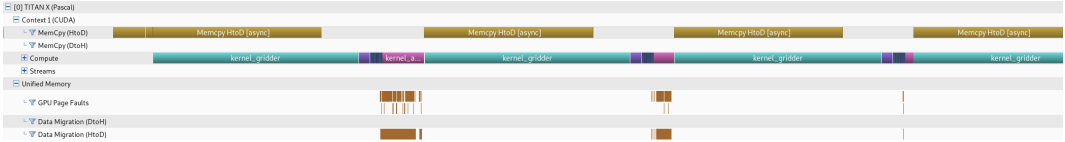
- Subgrids are added onto a larger grid
- The grid might not fit in GPU memory (e.g. 40k×40k ≈ 48 GB)
- Perform addition on the host:



ASTRON

Creating large images: using Unified Memory

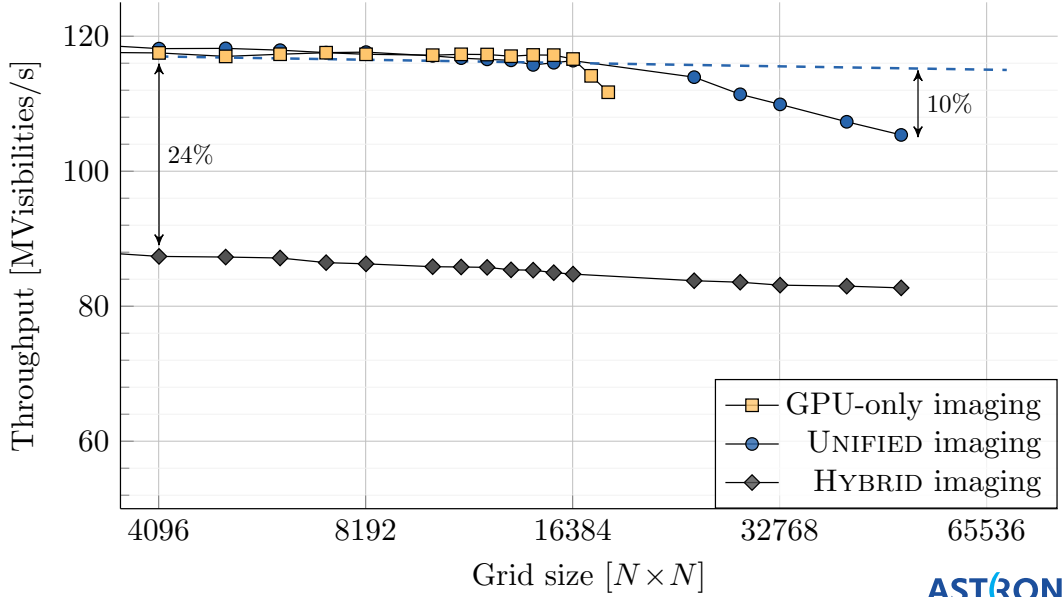
- Allocate memory:
  `malloc(size)` $\rightarrow$ `cuMemAllocManaged(ptr, size, CU_MEM_ATTACH_GLOBAL)`
- Apply tiling to keep pixels close in memory:
  `grid[`G_y`][`G_x`]` $\rightarrow$ `grid[`N_y`][`N_x`][`T_y`][`T_x`]` $(N_y \times T_y = G_y$ and $N_x \times T_x = G_x)$



GPU Page Fault groups		Data Migration (DtoH)		Data Migration (HtoD)	
Start	17.37 s (17,370,168,609 ns)	Start	53.121 s (53,121,339,457 ns)	Start	17.516 s (17,515,710,433 ns)
End	17.371 s (17,370,880,705 ns)	End	53.121 s (53,121,340,737 ns)	End	17.516 s (17,515,748,097 ns)
Duration	712.096 µs	Duration	1.28 µs	Duration	37.664 µs
Memory Acccess Type	Atomic	Size	4.096 kB	Size	442.368 kB
Virtual Address	0x2aba6fa00000	Migration Cause	Coherence	Migration Cause	Prefetch
GPU Page Faults	4	Throughput	3.2 GB/s	Throughput	11.745 GB/s
Process	13154	Virtual Address	0x2ac3d8000000	Virtual Address	0x2aba4fb14000
		Process	13154	Process	13154

ASTRON

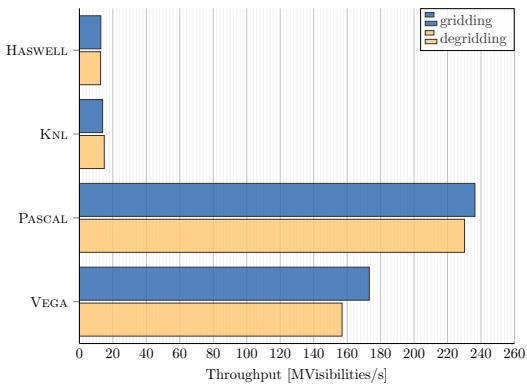
Creating large images: results

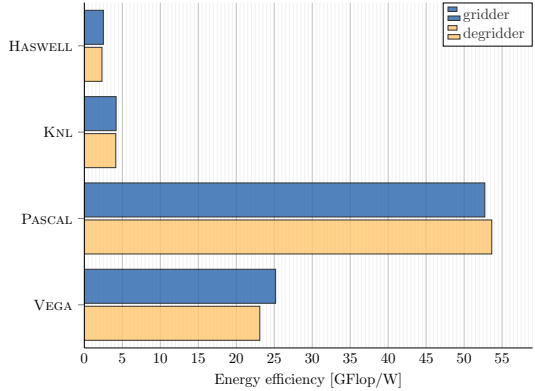


ASTRON

Performance and energy-efficiency comparison

- 2x Xeon E5-2697v3, Xeon Phi 7210X, NVIDIA GTX Titan X (Pascal) and AMD Vega FE:





- Pascal is the fastest and most energy-efficient device

ASTRON

Image-Domain Gridding for the Square Kilometre Array

- SKA-1 Low visibility rate: 9.5 GVis/s $\longrightarrow$ imaging data rate: **95 GVis/s**
- Compute: 50 PFlop/s (DP, total)
- Power cap: $\approx$ 5 MW (per site)

Image-Domain Gridding for the Square Kilometre Array

- SKA-1 Low visibility rate: 9.5 GVis/s $\longrightarrow$ imaging data rate: **95 GVis/s**
- Compute: 50 PFlop/s (DP, total)
- Power cap: $\approx$ 5 MW (per site)

- Imaging 'budget' $\approx$ 60%:
  - 15.3 PFlop/s (DP, per site)
  - 2887 Tesla P100 GPUs
  - 588.5 kW

- IDG on one Tesla P100: $\approx$ 0.10 GVis/s

Image-Domain Gridding for the Square Kilometre Array

- SKA-1 Low visibility rate: 9.5 GVis/s $\longrightarrow$ imaging data rate: **95 GVis/s**
- Compute: 50 PFlop/s (DP, total)
- Power cap: $\approx$ 5 MW (per site)

- Imaging 'budget' $\approx$ 60%:
  - 15.3 PFlop/s (DP, per site)
  - 2887 Tesla P100 GPUs
  - 588.5 kW

- IDG on one Tesla P100: $\approx$ 0.10 GVis/s
- IDG meets SKA-1 Low requirements!
  - 950 P100s required: **33%**
  - Power consumption: 171 kW, **29%**

ASTRON

Summary

- High performance astronomical imaging on GPUs with Image-Domain Gridding
  - Compute bound (unlike other imaging algorithms)
  - A-term correction at neglible cost
  - Very lage images (Unified Memory)

Summary

- High performance astronomical imaging on GPUs with Image-Domain Gridding
  - Compute bound (unlike other imaging algorithms)
  - A-term correction at neglible cost
  - Very lage images (Unified Memory)

- GPUs much faster and more (energy)-efficient than CPUs and Xeon Phi

Summary

- High performance astronomical imaging on GPUs with Image-Domain Gridding
  - Compute bound (unlike other imaging algorithms)
  - A-term correction at neglible cost
  - Very lage images (Unified Memory)
- GPUs much faster and more (energy)-efficient than CPUs and Xeon Phi
- **Most challenging sub-parts of imaging for SKA is solved!**

Summary

- High performance astronomical imaging on GPUs with Image-Domain Gridding
  - Compute bound (unlike other imaging algorithms)
  - A-term correction at neglible cost
  - Very lage images (Unified Memory)
- GPUs much faster and more (energy)-efficient than CPUs and Xeon Phi
- **Most challenging sub-parts of imaging for SKA is solved!**

More details:
"Image Domain Gridding: a fast method for convolutional resampling of visibilities, S. van der Tol et al., A&A 2018" and
Image-Domain Gridding on Graphics Processors, B. Veenboer et al., IPDPS 2017

Source available at:
`https://gitlab.com/astron-idg/idg`

ASTRON