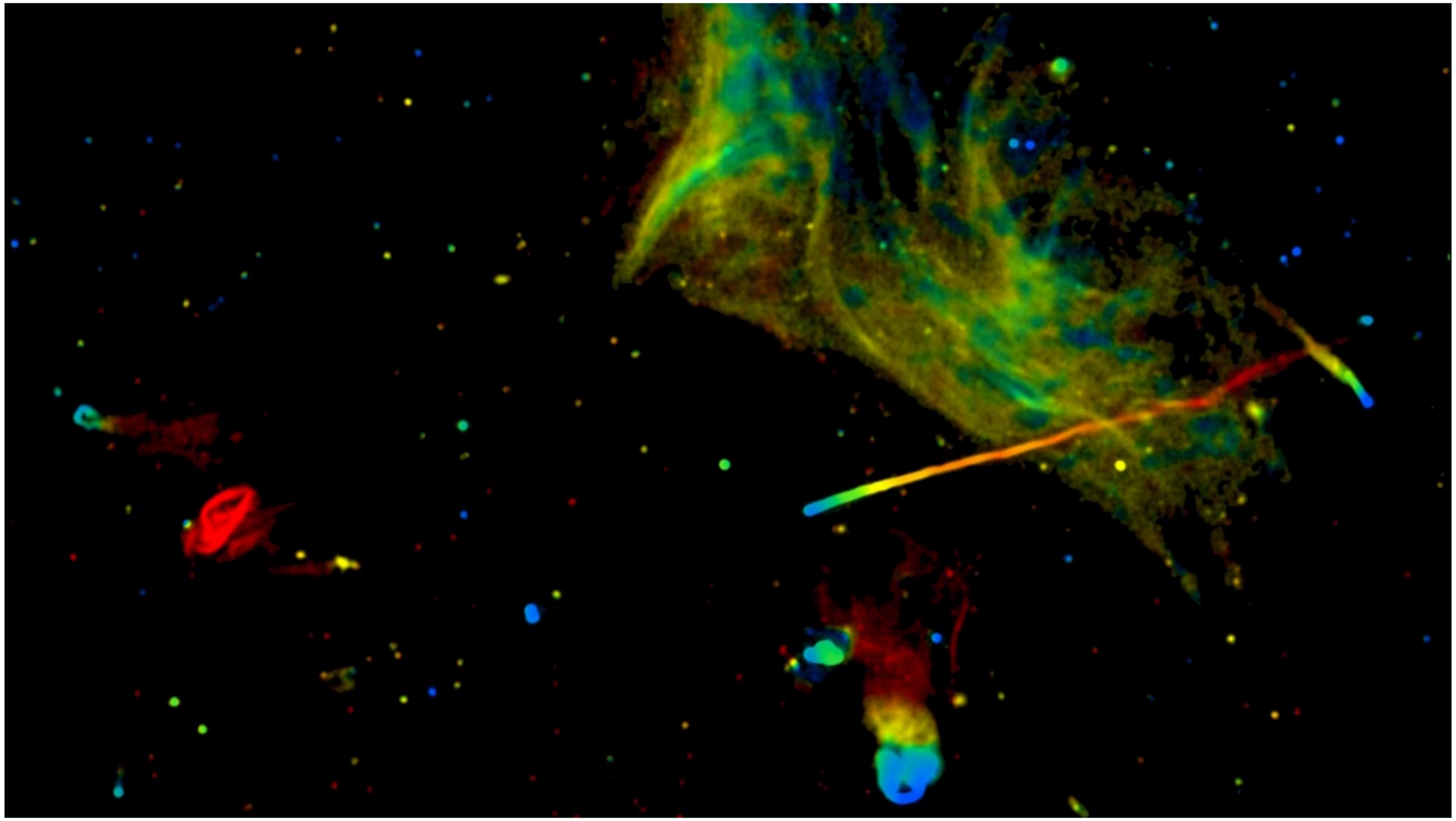
Exascale Computing for Radio Astronomy

How to Program?

Kees van Berkel

MPSoC 2017, July 2-7 — Annecy, France

1 Abell 2256



arXiv:1408.5931 (Image courtesy of NRAO/AUI)

2 Abell 2256

- a rich nearby galaxy cluster (> 500 galaxies),
- in the constellation Ursa Minor,
- measures 4M light years across,
- at a distance of about 800 million light-years.

Image (arXiv:1408.5931):

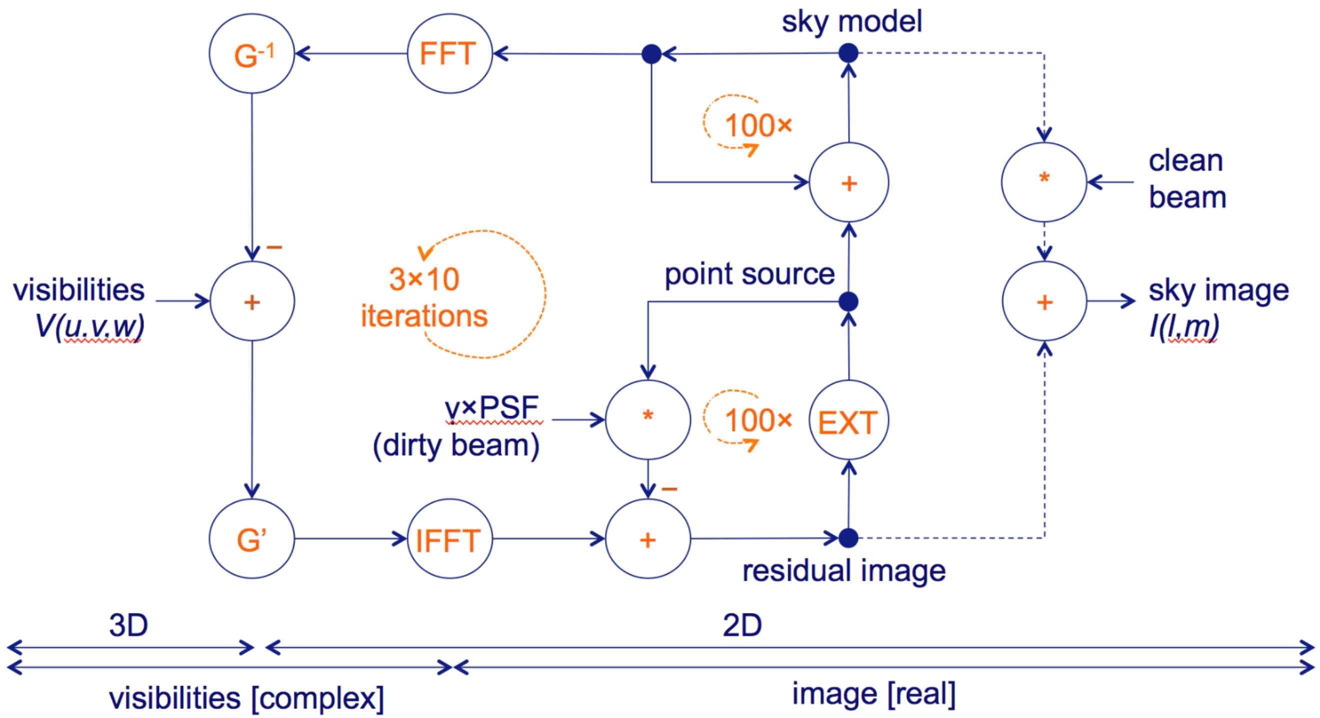
- VLA radio telescope, New Mexico, using 4 configurations,
- widefield image (almost the size of full moon),
- 47 hours of observation (2010-2012),
- spectrum: 1-8 GHz (color code = spectral index).

3 VLA radio telescope, New Mexico

- 27 independent antennae (dishes)
- each with a diameter of 25m

- 50-node compute cluster ≈ 1 *teraflops/sec* (10^{12}):
  50 × 2 × 8 × Intel E5 @ 2.6GHz;
- total compute load ≈ *petaflops/sec* (10^{15}), FPGA + hardware.
  Modern radio astronomy is increasingly *software defined*.

4 MPSoC 2015: Astronomical Workloads



Imaging algorithm [Tho01]: CLEAN [Hög74] + W-snapshot [Cor08].
Workload SKA1-mid telescope: $10^{17} - 10^{18}$ FLOPs/sec, "exascale".

5 MPSoC 2016: GPU or FPGA?

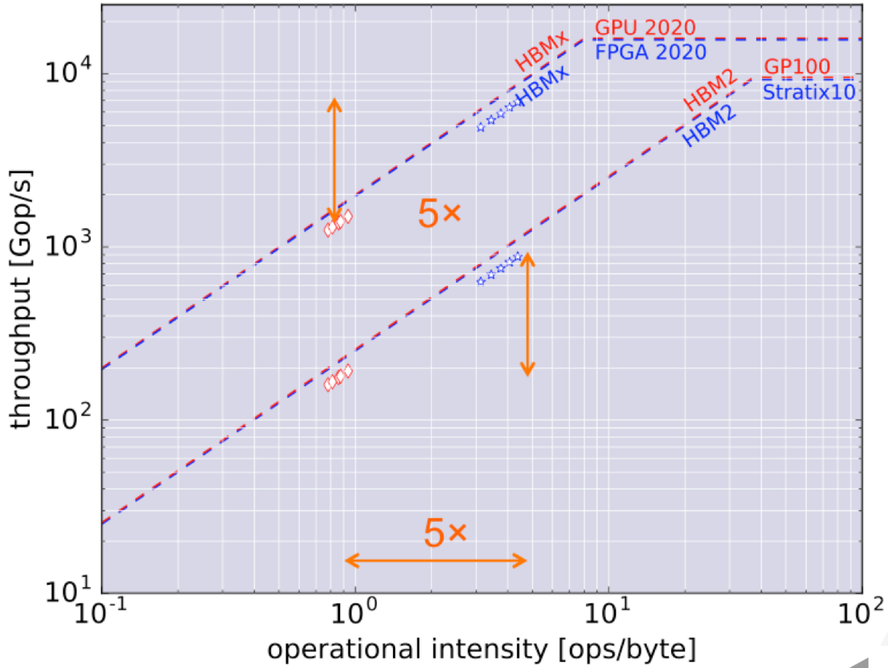
Exascale Computing for Radio Astronomy: GPU or FPGA?

Key imaging algorithm: 2D-FFT (16k × 16k)

FPGAs relative to GPUs:

- 5× less DRAM bandwidth,
- 5× more throughput,
- 10× less energy /2D-FFT

intrinsically.



6 State-of-the-art GPU and FPGAs

		Nvidia	Intel/Altera	Xilinx
		GP100	Stratix 10	VU13P
cmos	nm	16	14	16
clock frequency	MHz	1328	800	800
scalar/dsp processors		3584	11520	11,904
peak throughput	GFLOP/s	9519	9216	7619
data type	[32b]	float	float	fixed
DRAM bandwidth (HBM2)	GB/s	256	256	256
power consumption	W	300	126	
GFLOP/W		32	73	

7 Programming model for exascale computing

Programming model needs to support reasoning about parallelism :

- schedules, throughputs, resource utilization,
  and parameterization, program transformations, scaling, ..

Approach (Edward Lee, "*The Problem with Threads*" [Lee06]):

- "start with a deterministic mechanism", and "introduce judiciously and carefully nondeterminism where needed".

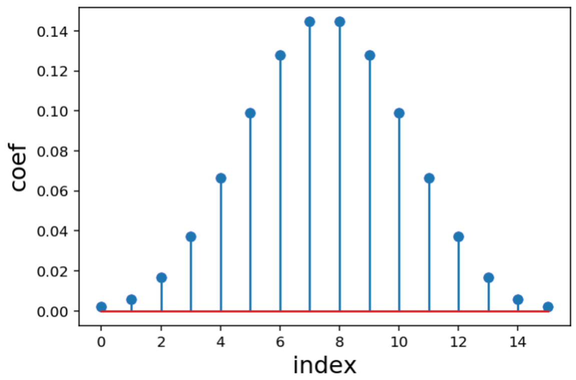
and (Lee & Messerschmitt, "*Synchronous Data Flow*" [Lee87]):

- "SDF explicitly displays concurrency, .. permits automatic scheduling onto parallel processors, ... is hierarchical, ...".

8 Audio filter: coefficients (Hamming)

In [3]:

```
from scipy import signal
coef = signal.firwin(numtaps=16, nyq=22500, cutoff=2500)
plt.xlabel('index', fontsize=16); plt.ylabel('coef', fontsize=16)
plt.plot(coef, 'bo');               plt.stem(coef);

# derive coefficients for sub-filters [odd, even+odd, even]
h   = np.array([coef[1::2], coef[::2]+coef[1::2], coef[::2]])
```



9 Audio filter: dataflow graph

In [4]:

```
class FIR(node):
    def __init__(self, I=[], fir=[]):
        super(FIR, self).__init__(I=I)
        self.N=len(fir)
        self.fo = [lambda x: int(sum([fir[i]*x[self.N-1-i] for i in range
    def _init(self):                          # runs post build(), i.e. when e
        self.I[0].init(x=[0 for i in range(self.N)])
```

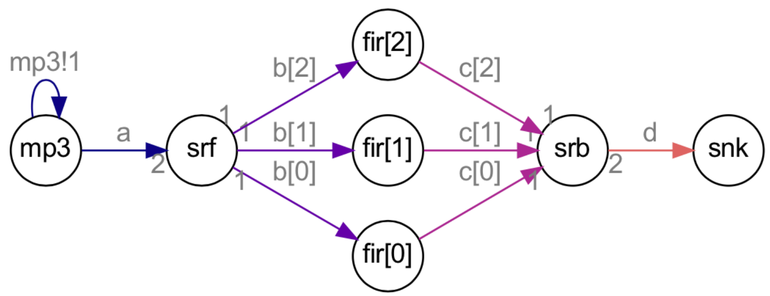
10 Audio filter: dataflow graph

In [5]:

```
G0    = graph(name="GFIRsr", rate=50000) # Strength-reduced filter [Par9:
G0.mp3= MP3(mp3='sultans.mp3', rate=44100, tmin=0.5)
G0.srf= LM(M=2, L=1, fo=[lambda x: x[1]-x[2], lambda x: x[2], lambda x: :
G0.fir= [FIR(fir=h[i]) for i in range(3)]
G0.srb= LM(M=1, L=2, fo=[lambda x, r: x[0]+x[1] if r==0 else x[2]+x[1]])
G0.snk= node ()
G0.a  = edge (G0.mp3, G0.srf)
G0.b  = [edge(G0.srf.O[i], G0.fir[i]) for i in range(3)]
G0.c  = [edge(G0.fir[i], G0.srb.I[i]) for i in range(3)]
G0.d  = edge (G0.srb, G0.snk)
G0.a.init (x=[0,0], S=2)
G0.build()#colormap='hot')
G0.plot_graph()
```

```
mp3 (MP3)      :       : sultans.mp3: mono/16b, len=59.5s*44100Hz=2623950
G              :       : no errors
```

Out[5]:



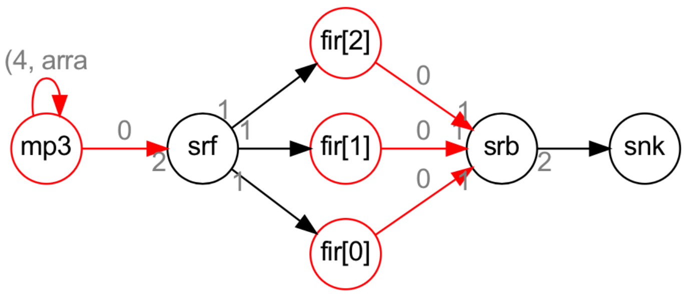
11 Audio filter: simulation

A *strength-reduced* FIR: $\frac{3}{2}N$ taps delivers $2\times$ throughput vs N taps.

In [26]:
```
G0.view(sim=True)
```

node G s view data n next cyc : 1 run

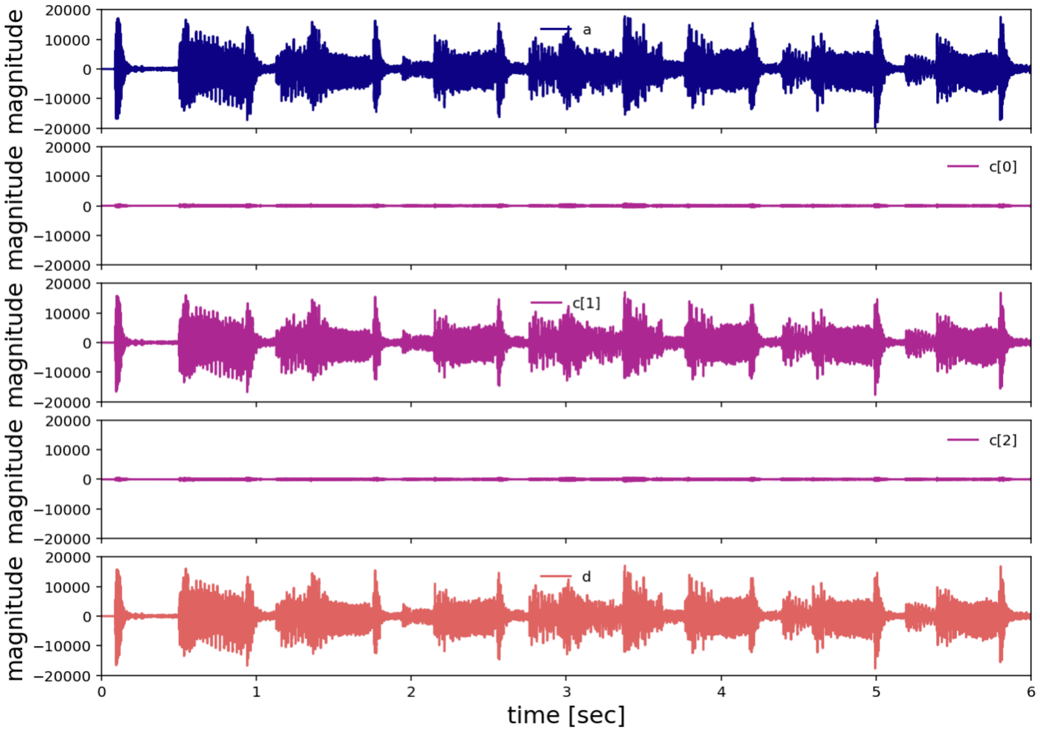
```
rate:50.0kHz   cyc:4          sec:80.000us   cpu:0.1s       pause (0ke/cs)
```



12 Audio filter: time domain

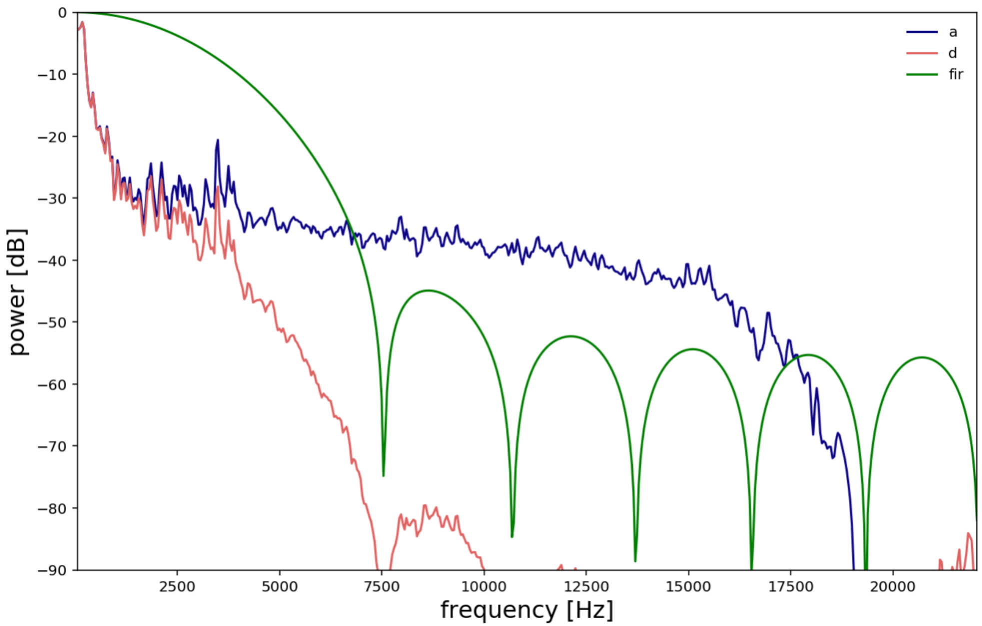
In [6]:
```
G0.sim(T=6, mute=True)
G0.plot_data(Edges=[G0.a, G0.c, G0.d], stacked=True, fix=20000);
```

```
rate:50.0kHz  cyc:300000    sec:6.000s    cpu:102.4s    pause (16ke/cs)
```



13 Audio filter: frequency domain

In [7]:
```
G0.plot_spectra(Edges=[G0.a, G0.d], fir=coef);
```



In [8]:
```
G0.d.play_data()
```

```
d : rate= 44.1kHz
```

Out[8]:

14 Sobel filter: dataflow graph

In [14]:

```
(width, height)= Image.open('valve.png').size   # image width, height
wh, W = width*height, 8    # image size,  burst size for RAM accesses
fx= lambda x: np.int16(x[0][0]+2*x[1][0]+x[2][0]-x[0][2]-2*x[1][2]-x[2][
fy= lambda x: np.int16(x[1][2]+2*x[1][1]+x[1][0]-x[0][2]-2*x[0][1]-x[0][
fm= lambda x: np.int16(min(255, 0.5*np.sqrt(np.int32(x[0])*x[0] +
                                          np.int32(x[1])*x[1])))
G1     = graph (name="Gsobel", rate=10**8)
G1.sr = SRC (f=lambda x: W*x, fh=lambda x: x>wh/W)
G1.sw = SRC (f=lambda x: wh+ W*x)
G1.ram= RAM (I=[G1.sr, G1.sw], size=1024**2, W=W)
G1.unp= LM  (I=[G1.ram], L=W, M=1, O_p=[3], fo= lambda x, r: np.int16(x[
G1.dl0= node(I=[G1.unp.O[0]], O_p=[2])
G1.dl1= node(I=[G1.dl0.O[0]], O_p=[2])
G1.sbx= node(I=[G1.unp.O[1], G1.dl0.O[1], G1.dl1.O[0]], fo=fx)
G1.sby= node(I=[G1.unp.O[2], G1.dl1.O[1]], fo=fy)
G1.mag= node(I=[G1.sbx, G1.sby], fo=fm)
G1.pck= LM  (I=[G1.mag], L=1, M=W, fo=lambda x,r: np.array(x[0:W],dtype=
G1.e  = edge(G1.pck, G1.ram.I[2])   # to close the loop
G1.dl0.I[0].init(D=1, S=0, x=[np.int16(0) for n in range(width)])
G1.dl1.I[0].init(D=1, S=0, x=[np.int16(0) for n in range(width)])
G1.build()
for e in G1.sbx.I+ G1.sby.I:
    e.init(x=[0,0])
```

```
G               :       : no errors
```

15 Sobel filter: dataflow graph

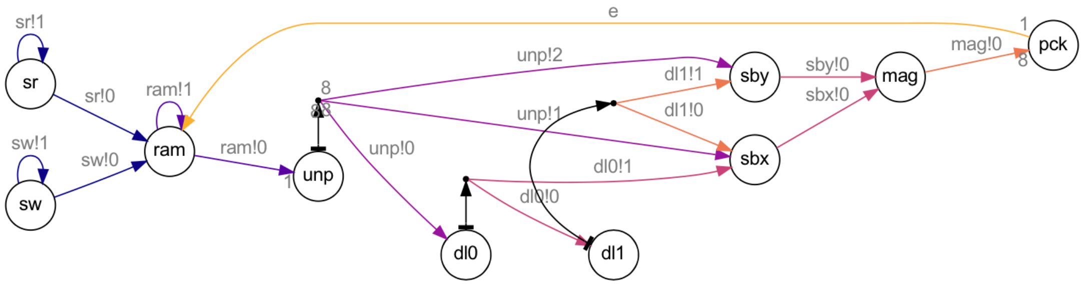
A Sobel filter (image processing) emphasizes edges in images.

In [15]:

```
G1.ram.load_image(file='valve.png')
G1.ram.new_image ('sobel', start=wh, width=width, height=height)
G1.sim(T=1, mute=True)
G1.plot_graph()
```

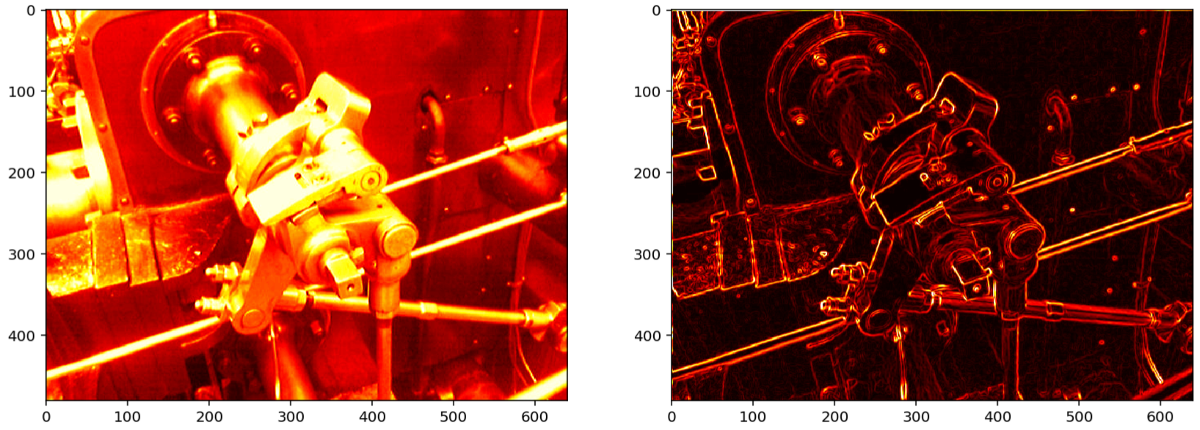
```
ram (RAM)      :       : valve.png (image size= (640, 480))
rate:100.0MHz cyc:307195    sec:3.072ms   cpu:154.3s    sr: halted;
rate:100.0MHz cyc:307223    sec:3.072ms   cpu:154.3s    quiescence
                                                        (22ke/cs)
```

Out[15]:



16 Sobel filter: image domain

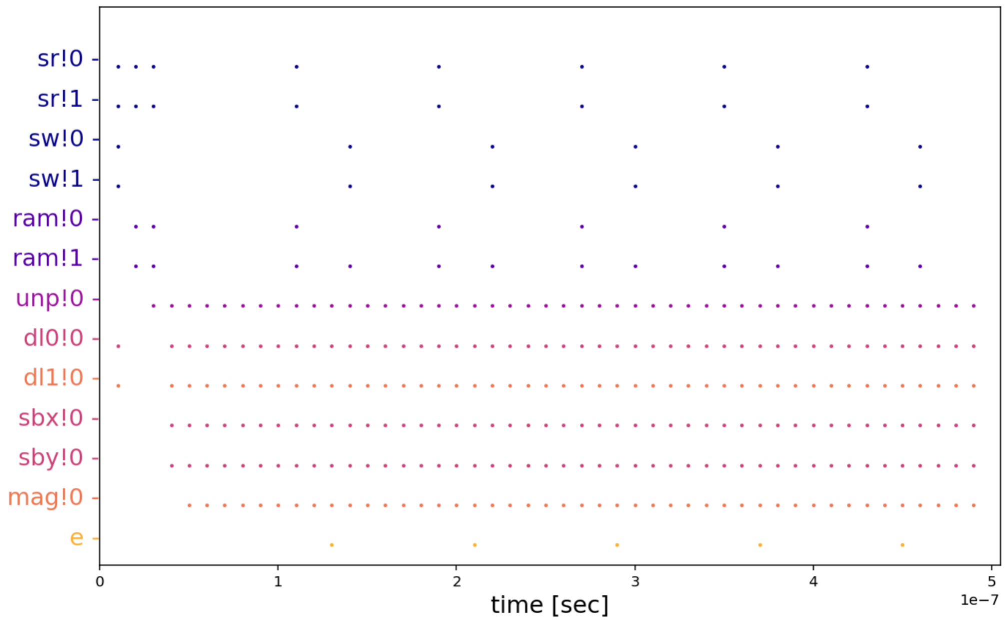
In [16]:
```
G1.ram.view_images(color='hot')
```



17 Sobel filter: flow domain

- RAM read and write once per W=8 cycles;
- words of W=8 pixels are unpacked and packed.

In [17]:
```
G1.plot_flow(tmax=0.0000005);
```



18 Lorentz attractor: dataflow graph

Numeric integration (Euler) of

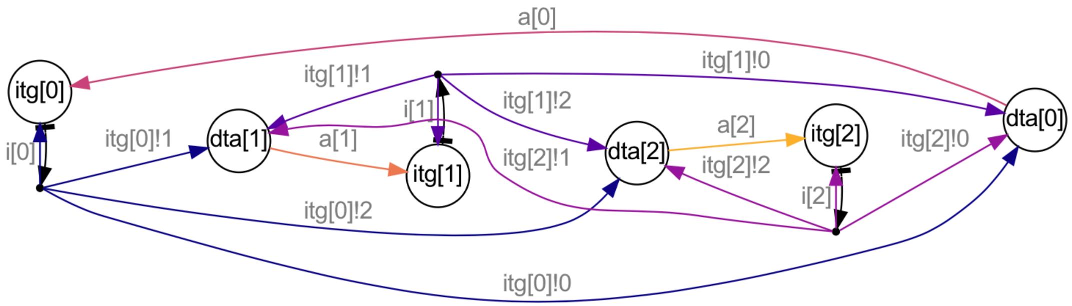
$$\begin{aligned} \mathrm{d}x/\mathrm{d}t &= \sigma(y - x), \\ \mathrm{d}y/\mathrm{d}t &= x(\rho - z) - y, \\ \mathrm{d}z/\mathrm{d}t &= xy - \beta z. \end{aligned}$$

In [19]:

```
G2.sim(T=10000)
G2.plot_graph()
```

```
rate:1.0Hz     cyc:10000      sec:10000.0s   cpu:2.6s       pause (29ke/cs)
```

Out[19]:



19 Lorentz attractor: dataflow graph

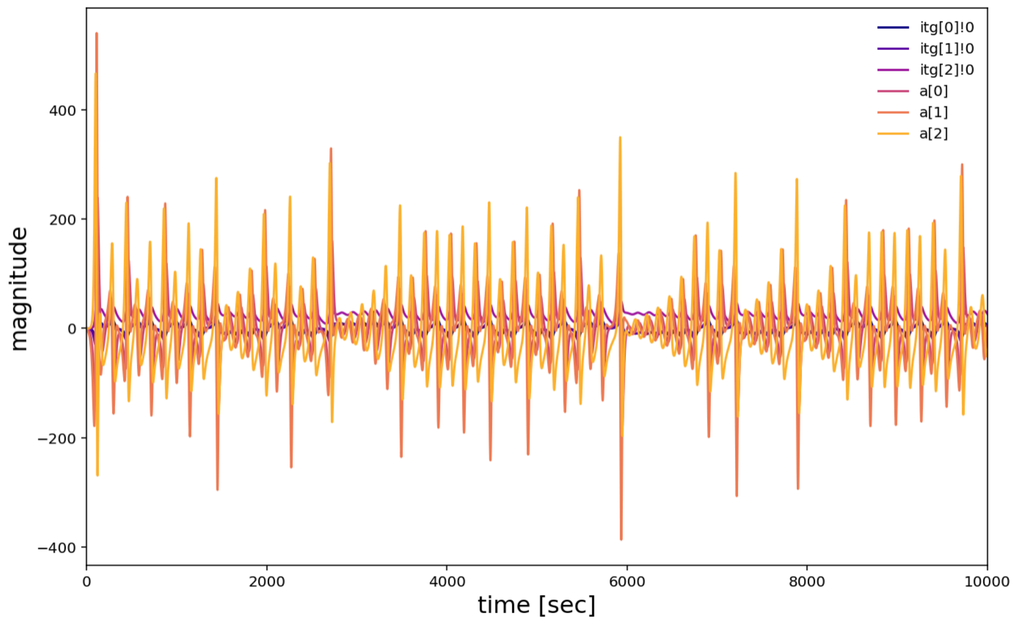
In [18]:

```
(sigma, rho, beta) = (10.0, 28.0, 8.0/3.0)
initial_state      = [-0.15, -0.2, 0.2]

G2     = graph (name="Glorenz")                          # Euler integration
G2.itg= [node (O_p=[4], fo= lambda x: x[1]+0.01*x[0]) for i in range(3)]
G2.i   = [edge (G2.itg[i].O[3], G2.itg[i].I[1])        for i in range(3)]
G2.dta= [node (I=[G2.itg[0].O[i], G2.itg[1].O[i], G2.itg[2].O[i]])
                                                          for i in range(3)]
G2.a   = [edge (G2.dta[i], G2.itg[i])                     for i in range(3)]

G2.dta[0].init(fo= lambda x: sigma*(x[1] - x[0]))
G2.dta[1].init(fo= lambda x: (rho-x[2])*x[0] - x[1])
G2.dta[2].init(fo= lambda x: x[0]*x[1] - beta*x[2])
for i, e in enumerate(G2.i):
    e.init(D=1, S=0, x=initial_state[i])
for i, e in enumerate(G2.a):
    e.init(D=1, S=0, x=0)
G2.build (N_color=True)
```

```
G               :       : no errors
```

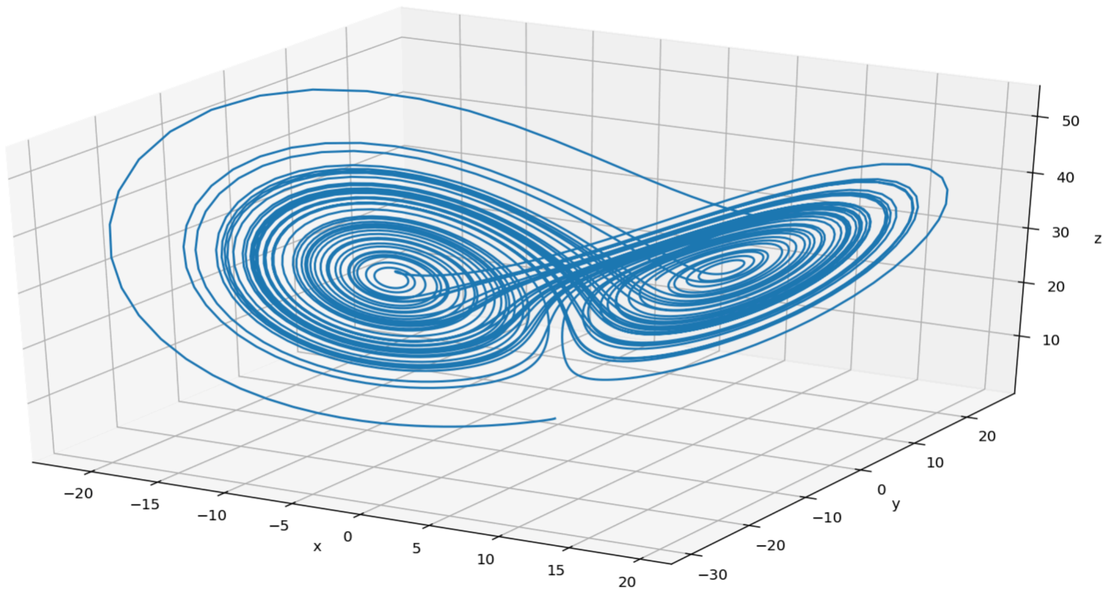
20 Lorentz attractor: time domain

In [20]:
```
G2.plot_data();
```



21 Lorentz attractor: trajectory

In [21]:
```
from mpl_toolkits.mplot3d import Axes3D
[x, y, z] = [n.O[0]._V for n in G2.itg]
fig = plt.figure(figsize=(16,8))
ax = fig.gca(projection='3d'); ax.plot(x, y, z)
ax.set_xlabel('x'); ax.set_ylabel('y'); ax.set_zlabel('z')
plt.show();
```



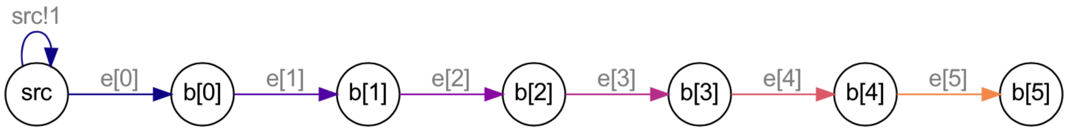
22 Pipeline: dataflow graph

In [22]:

```
N       = 6
G3      = graph(name="Gpipe")
G3.src  = SRC(rom=[i for i in range(100)])
G3.b    = [node() for i in range(N)]
G3.e    = path(G3.src, G3.b)
G3.build()
G3.plot_graph()
```

```
G              :       : no errors
```

Out[22]:



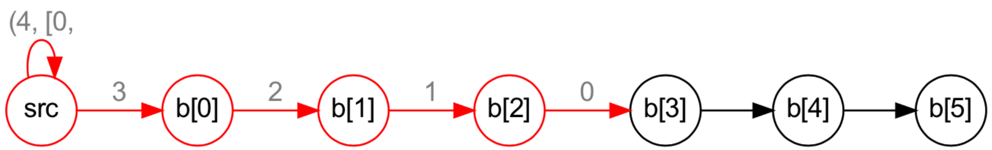
23 Pipeline: simulation

In [23]:

```
G3.view(sim=True)
```

node G s view data n next cyc : 1 run

rate:1.0Hz cyc:4 sec:4.000s cpu:0.0s pause



24 Pipeline: varying source and sink rates

In [24]:

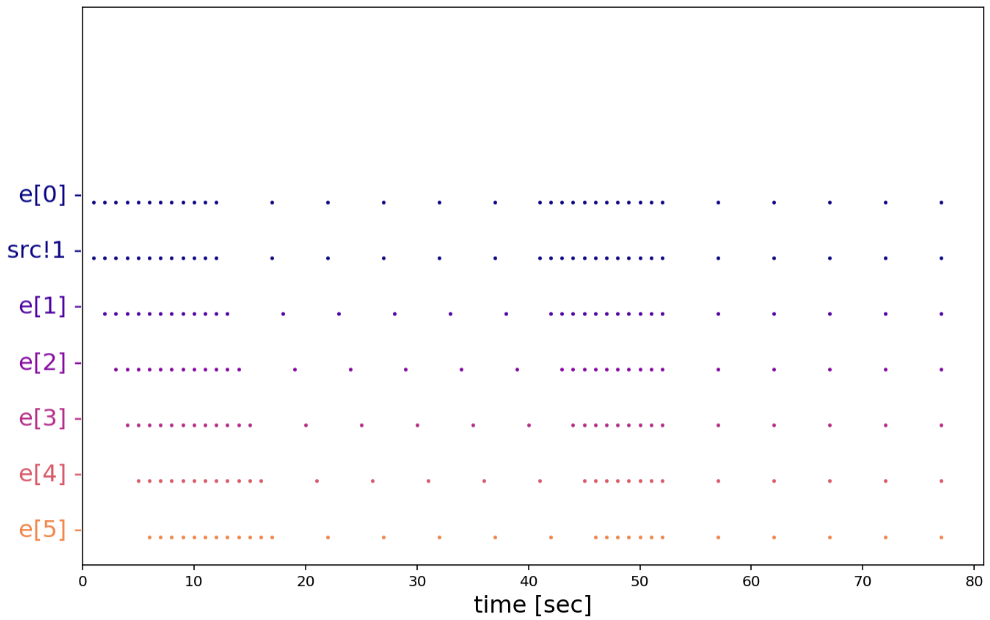
```
G3.src.set_rate(rate=1)

G3.reset()
G3.sim(T=10)
G3.src.set_rate(rate=0.2)
G3.sim(T=40)
G3.src.set_rate(rate=1)
G3.sim(T=50)
G3.b[N-1].set_rate(rate=0.2)
G3.sim(T=80)
```

```
rate:1.0Hz     cyc:10         sec:10.0s      cpu:0.0s       pause
rate:1.0Hz     cyc:40         sec:40.0s      cpu:0.0s       pause
rate:1.0Hz     cyc:50         sec:50.0s      cpu:0.0s       pause
rate:1.0Hz     cyc:80         sec:80.0s      cpu:0.0s       pause
```

25 Pipeline: flow domain

- tokens ripple 1 stage per clock cycle;
- filled pipeline can pause and restart *instantaneously*.

In [25]:
```
G3.plot_flow();
```



26 "StaccatoLab" execution model

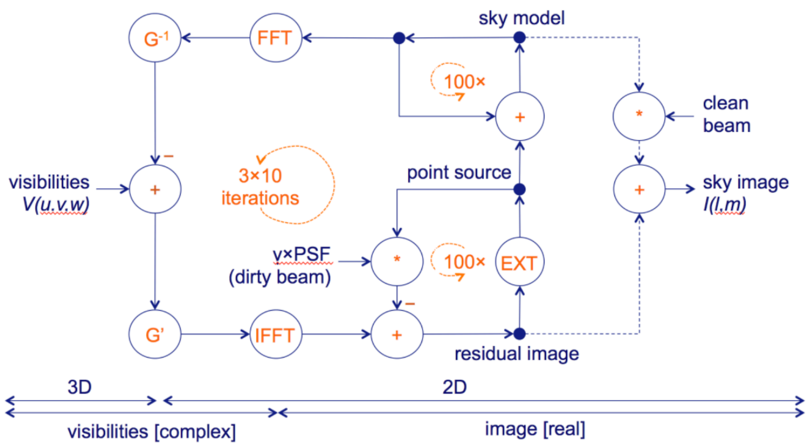
Feature	brings
Self-**T**imed	max parallelism, max throughput ("data driven")
Clocked	match with synchronous hardware (FPGA); SRDF= one firing per clock
Throttled	real-timing of average throughput; throttle-up = unfolding
One token per firing	best trade-off between hardware resources and throughput (no loss in expressiveness)
Look-**A**head **B**ack pressured	instantaneous pipeline pause and restart, at full speed, at lowest costs

27 StaccatoLab programming model

- program = dataflow graph;
- {single-rate, multi-rate, cyclo-static, boolean} dataflow, non-deterministic merge;
- medium to coarse grained;
- RAM node for array tokens (image/video processing);
- Python as host language:
  concise graph descriptions (hierarchical, parameterized),
  dynamic typing, rich libraries (vizualization, domain-specific);
- interactive simulation+debug (Jupyter notebook);
- work in progress: graph transformations, verilog backend.

28 StaccatoLab: scaling up

... from teraflops to exaflops (?)



- *hierarchy*: node = subgraph;
- *repetitive* graph structures, parameterized;
- program *transformations*, incl. unfolding;
- *abstraction*, e.g. 16k×16k image = 1 *array token*; edges with array tokens to be mapped onto (multiple) DRAMs;
- *fault tolerance*, dataflow based (?)

29 References

- [Bil96] G. Bilsen et al. Cyclo-static dataflow. IEEE Transactions on Signal Processing, 44(2): 397–408, 1996.
- [Cor08] T.J. Cornwell et al, Wide field imaging for the Square Kilometre Array, arXiv:1207.5861
- [Hög74] J. Högbom, Aperture Synthesis with a Non-Regular Distribution of Interferometer Baselines, Astronomy and Astrophysics Supplement, 19974Vol. 15, pp. 417-426.
- [Lee87] E. A. Lee and Messerschmitt, Synchronous Data Flow, Proc of the IEEE, Vol. 75, No. 9, Sep. 87.
- [Lee06] E. A. Lee, "*The Problem with Threads*", Computer, vol. 39, no. , pp. 33-42, May 2006.
- [Par99] K. K. Parhi, VLSI Digital Signal Processing Systems: Design and Implementation, Wiley,1999.
- [Tho01] A. Thompson et al, 2001, Interferometry and synthesis in radio astronomy, Wiley, N.Y