ASTRON

Netherlands Institute for Radio Astronomy

COBALT: Creating a High-Throughput, Real-Time Production System Using CUDA, MPI and OpenMP

GTC 2014

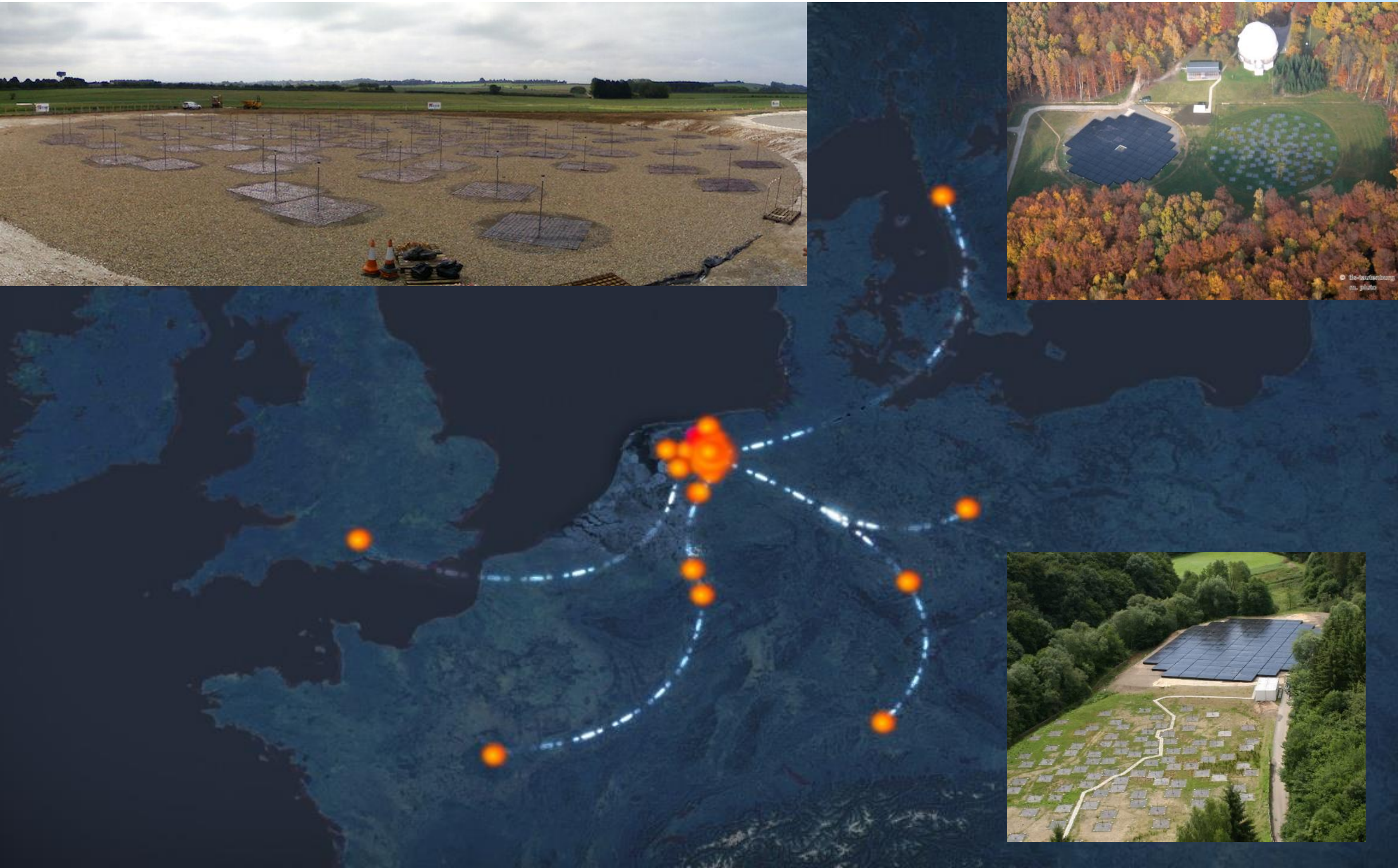
Jan David Mol
Wouter Klijn

Outline

- Introduction to LOFAR and Radio astronomy
- COBALT system
    - Hardware software co-design
    - Performance
- Time line
- Software
    - Refactoring / OpenCL vs CUDA / JIT compilation
    - Abstraction layer / Libraries

- Conclusions

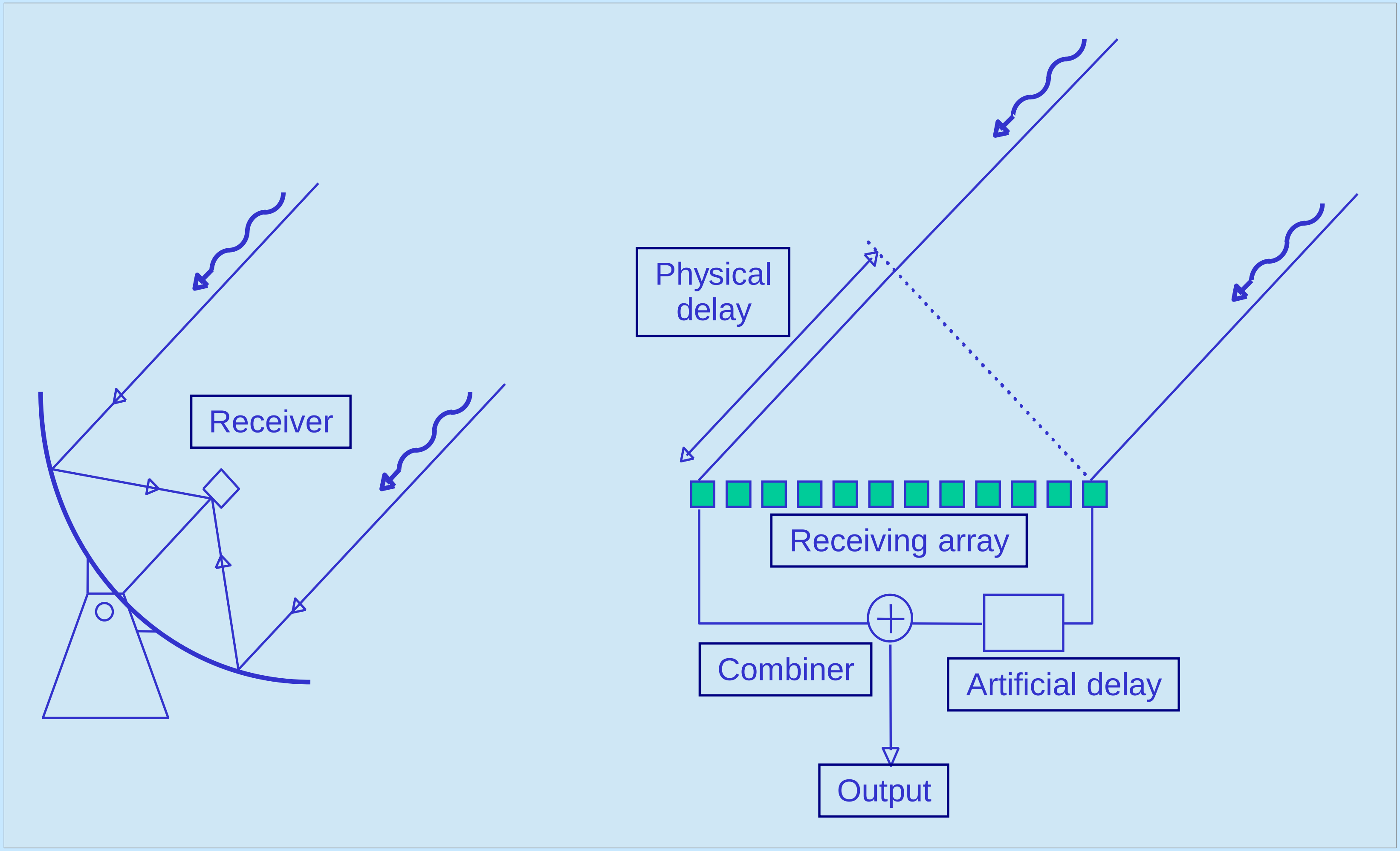
Introducing LOFAR



Central antenna fields

Dutch antenna field,
Inset: low band antenna

Phased Arrays



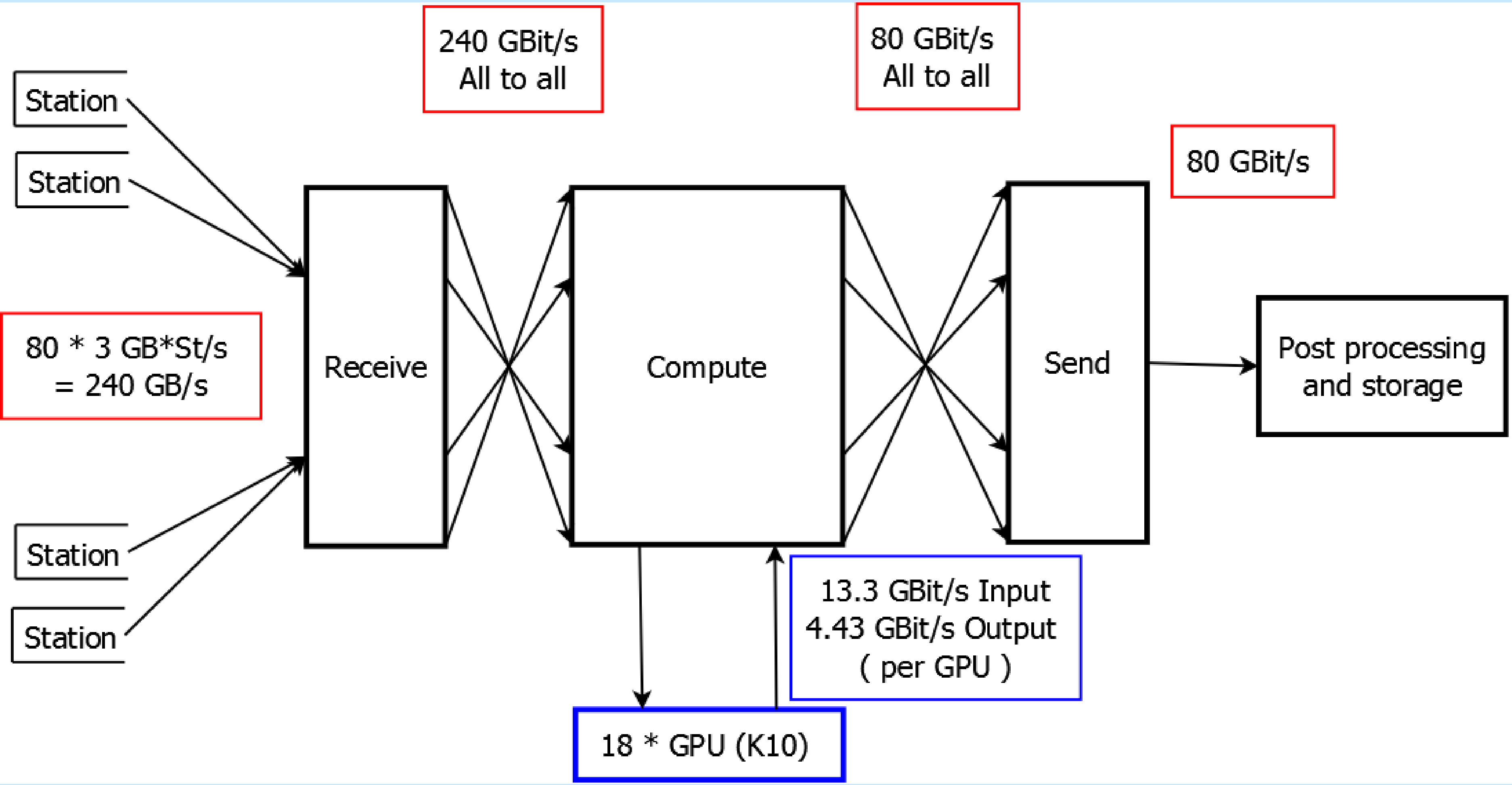
Current compute cluster: IBM Blue Gene/P

Critical Design Review
Feb 2013

- Recommendations:
    - Consolidate choices fast:
        - OpenCL vs CUDA
        - AMD vs Nvidia (vendor lock in?)
    - Get hardware ASAP
    - Contain external dependencies (infrastructure and system administration)
    - Exchange man power for hardware if possible

- Limited available experience with GPU programming!

Central processing
Abstract workflow



Hardware prototypes
(Mar 2013)

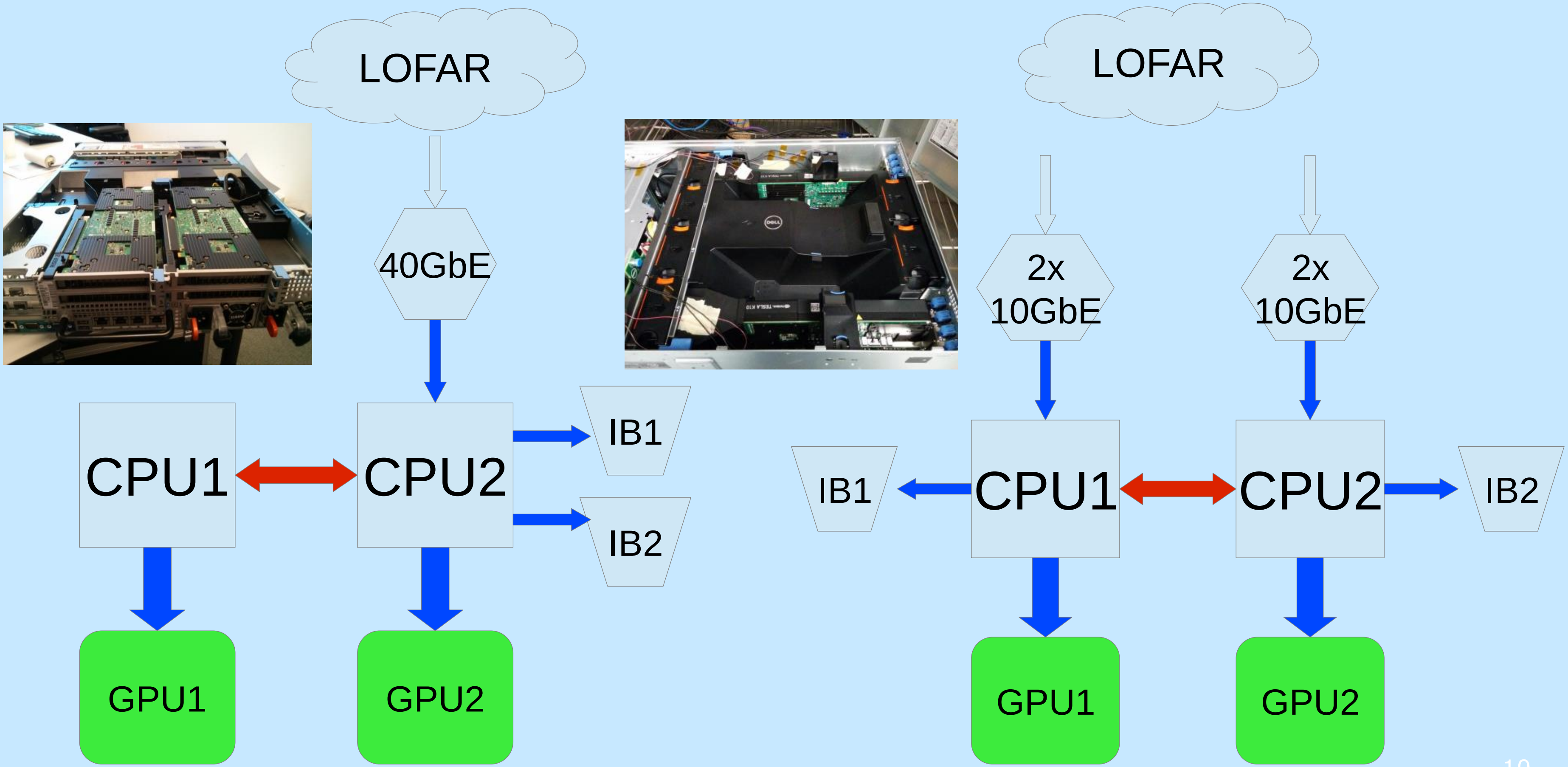
First Design

Dell R720

40GbE
PCIe balance
load on QPI

Second Design

Dell T620



LOFAR

40GbE

CPU1

CPU2

IB1

IB2

GPU1

GPU2

LOFAR

2x
10GbE

2x
10GbE

IB1

CPU1

CPU2

IB2

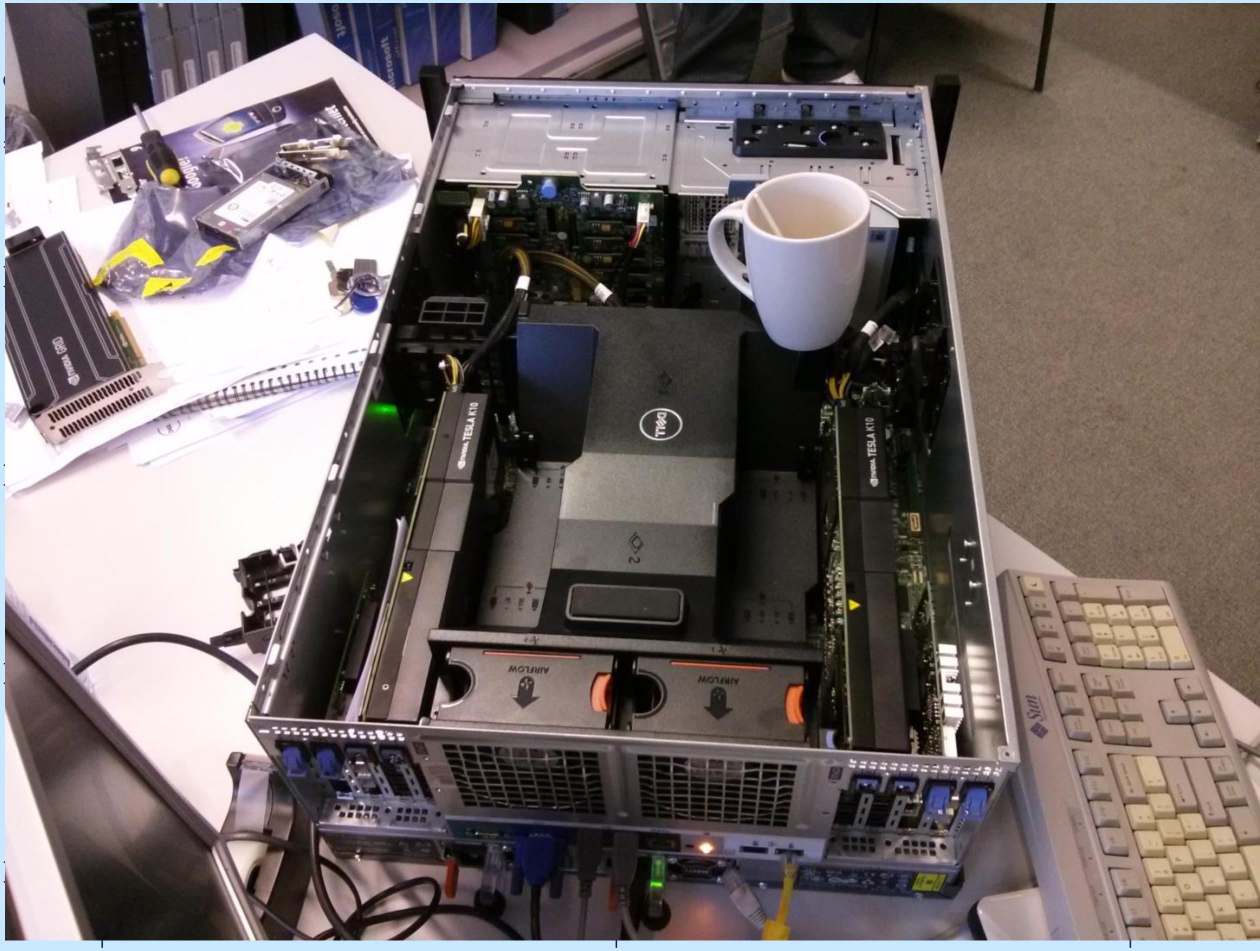
GPU1

GPU2

Hardware Prototype

- GPU idle temperatures:

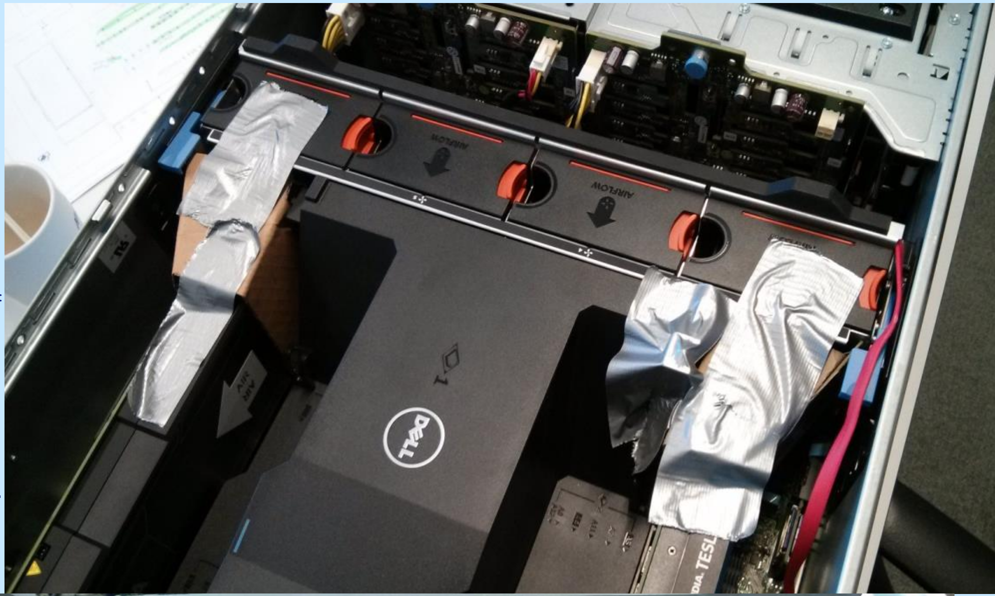
```
| NVIDIA-SMI 5.319.12   Driv
|=============================
|   0  Tesla K10.G2.8GB    O
| N/A   75C     P0    43W / E
+-----------------------------
|   1  Tesla K10.G2.8GB    O
| N/A   76C     P0    42W / E
+-----------------------------
|   2  Tesla K10.G2.8GB    O
| N/A   62C     P0    42W / E
+-----------------------------
|   3  Tesla K10.G2.8GB    O
| N/A   46C     P0    36W / E
+-----------------------------
```

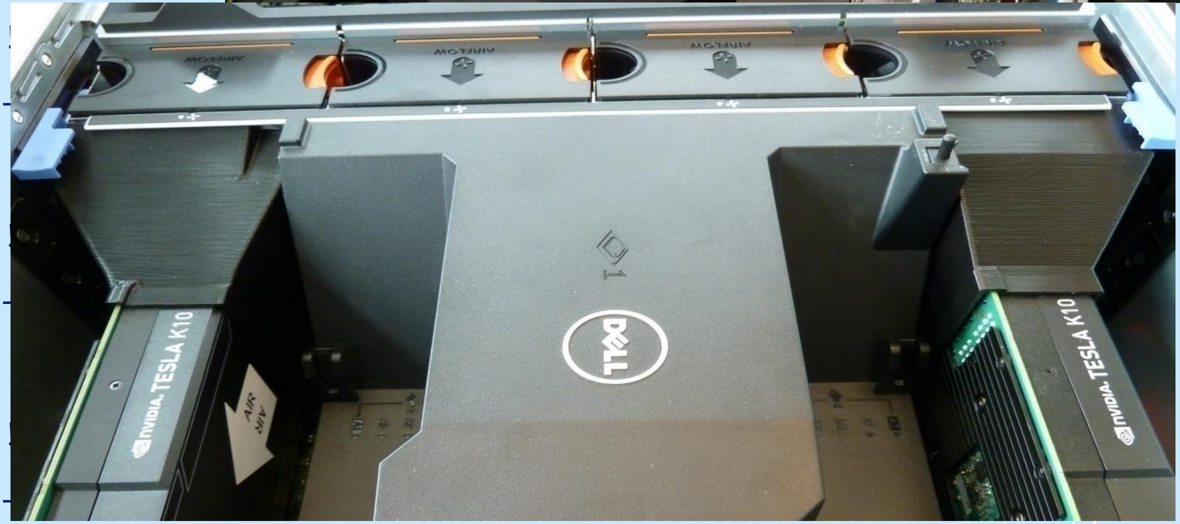


Duct-taped vs. 3D-printed air-flow guides (Apr 2013)

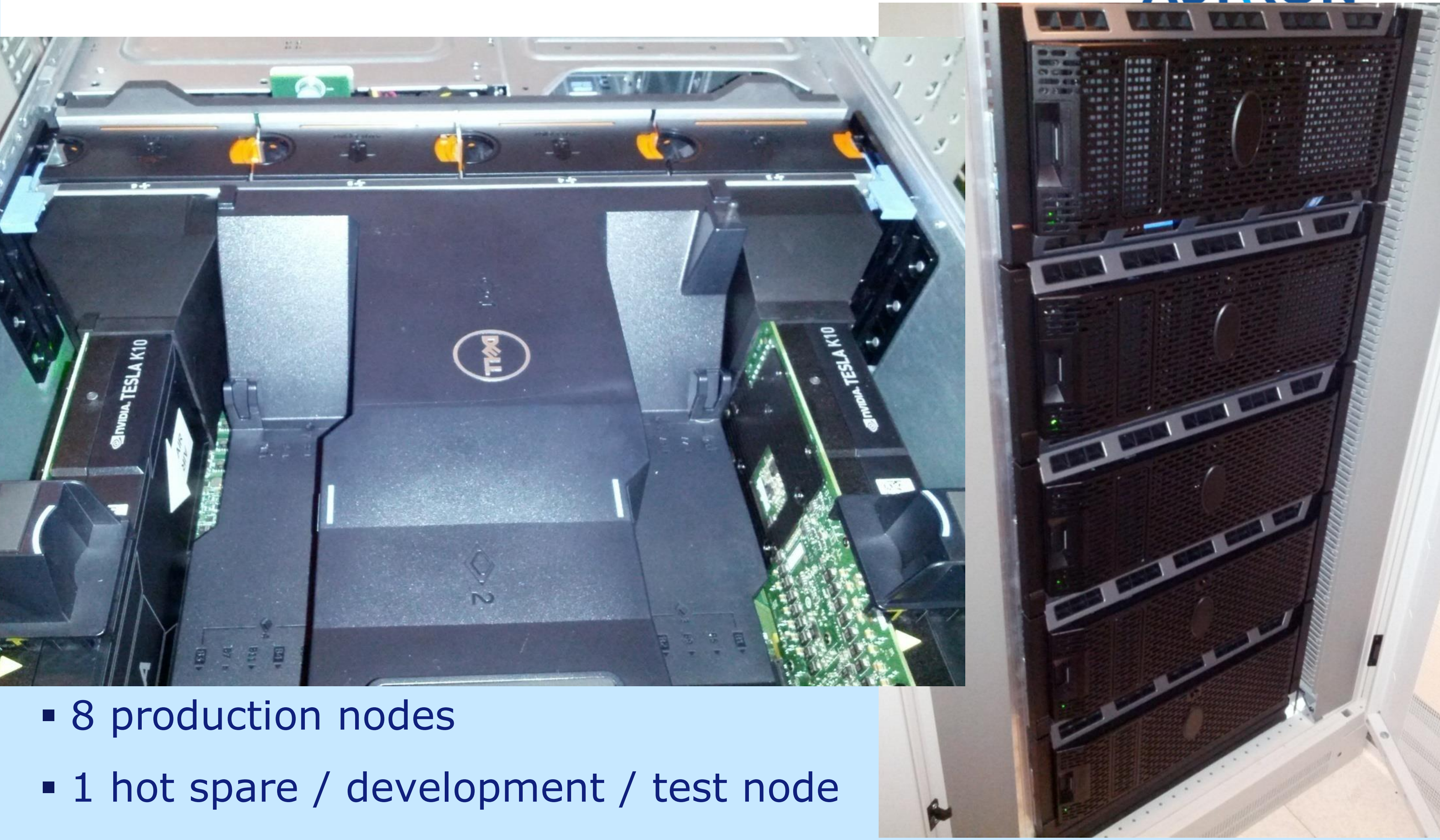
- GPU full load temperature
- Validated by DELL this week

```
| NVIDIA-SMI 5.319.12   Driver Version:
|===============================+=======
|   0  Tesla K10.G2.8GB     Off  | 0000:0
| N/A   48C    P0    92W / ERR! |    2%
+-------------------------------+-------
|   1  Tesla K10.G2.8GB     Off  | 0000:0
| N/A   52C    P0    91W / ERR!
+------------------------------
|   2  Tesla K10.G2.8GB     Off
| N/A   51C    P0    92W / ERR!
+------------------------------
|   3  Tesla K10.G2.8GB     Off
| N/A   49C    P0    95W / ERR!
+------------------------------
```



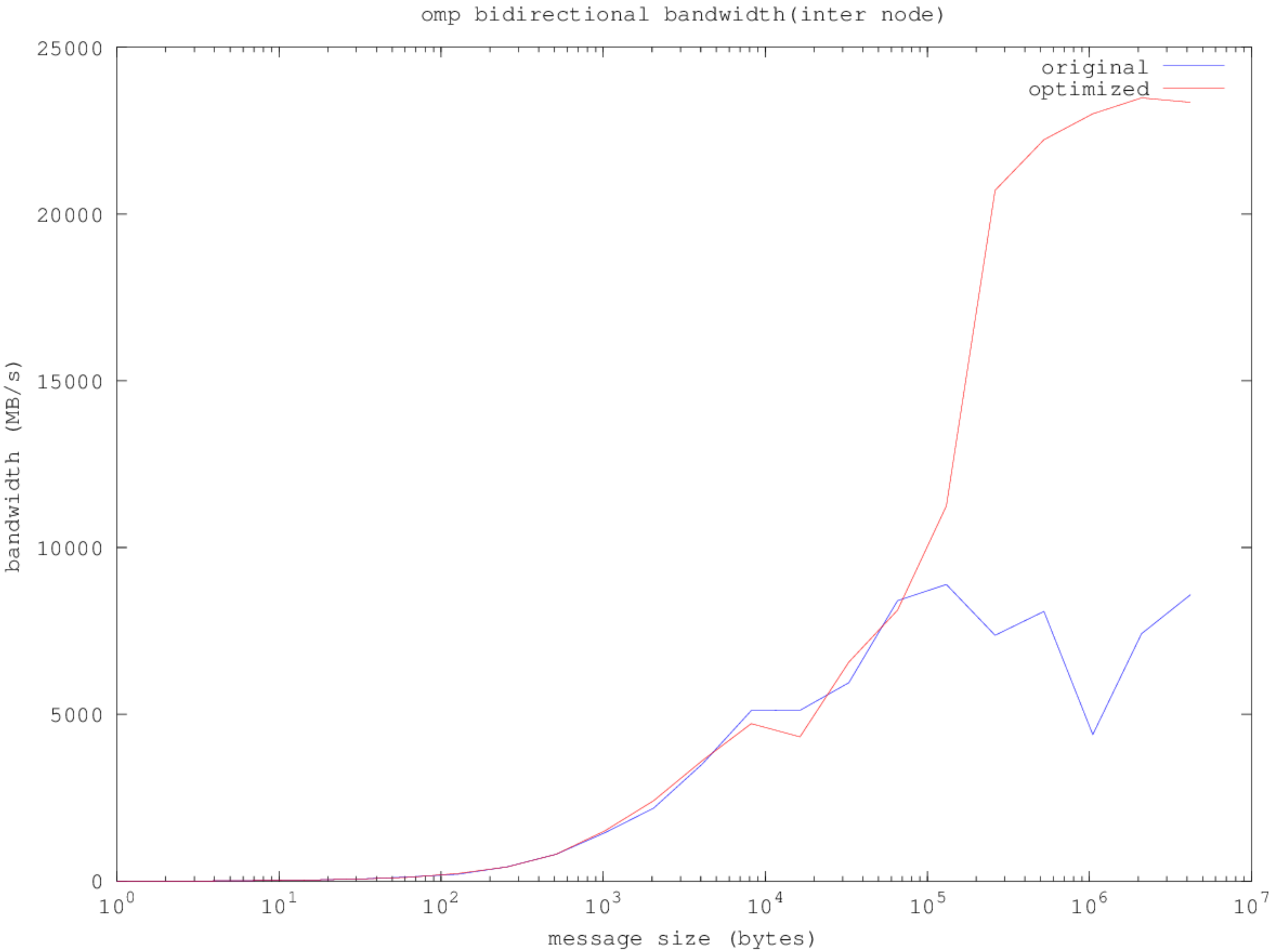


Final Cobalt Hardware (Jun 2013)
System Ready (Sep 2013)



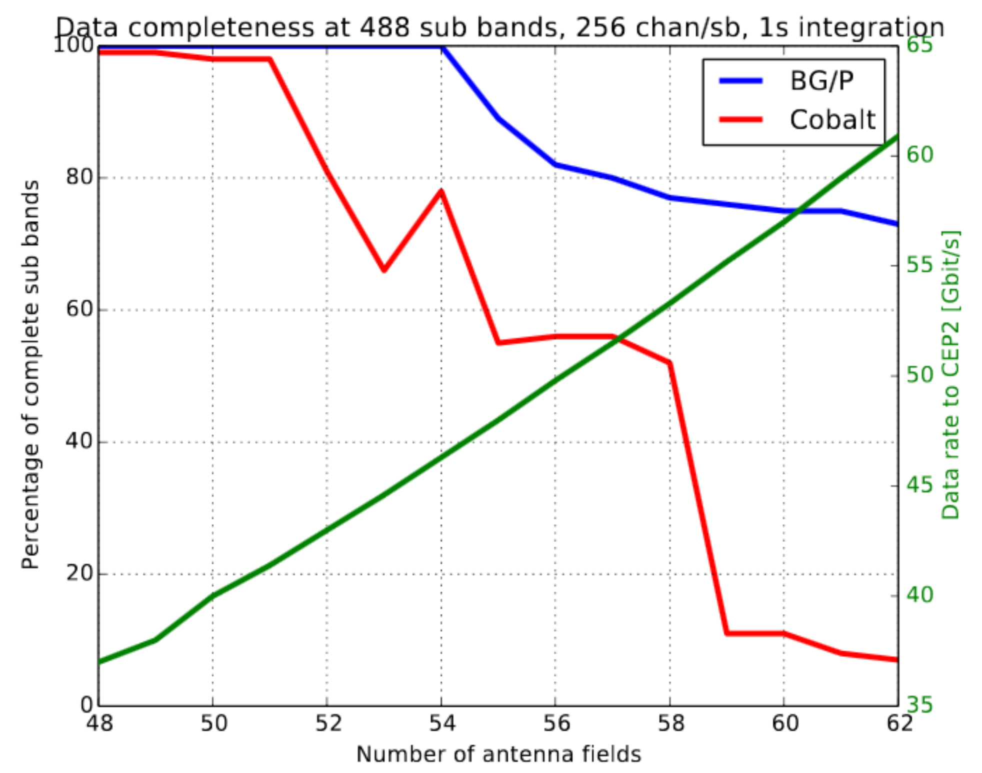
- 8 production nodes
- 1 hot spare / development / test node
- All infrastructure ready (Oct 2013)

MPI Tuning


omp bidirectional bandwidth(inter node)
bandwidth (MB/s)
25000
20000
15000
10000
5000
0
10^0
10^1
10^2
10^3
10^4
10^5
10^6
10^7
message size (bytes)
original
optimized

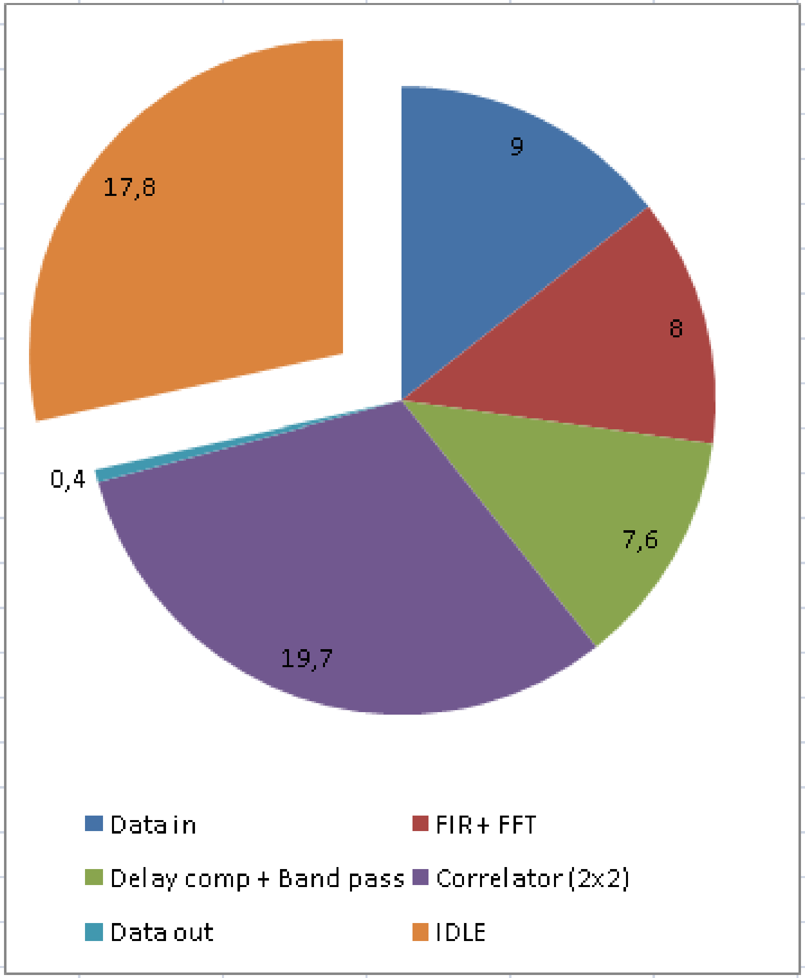
Cobalt Performance

- Lower input losses than BG/P
- Output losses at >30 Gbit/s to storage

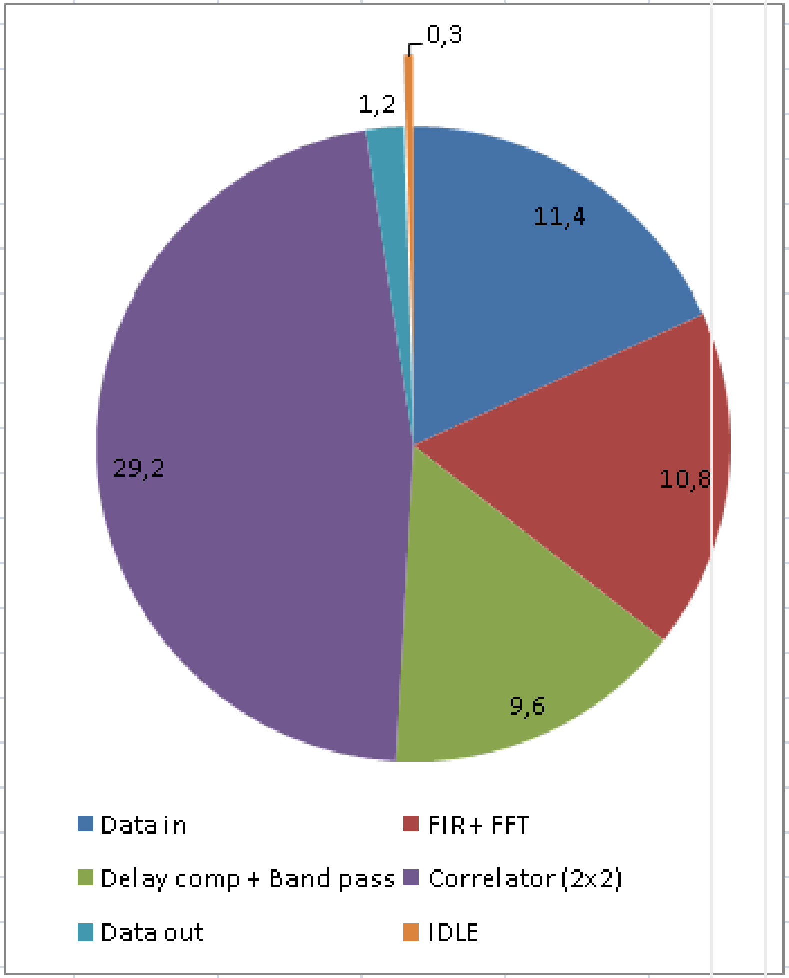


GPU Correlator Load

64 stations (192 Gbit/s)



80 stations (240 Gbit/s)

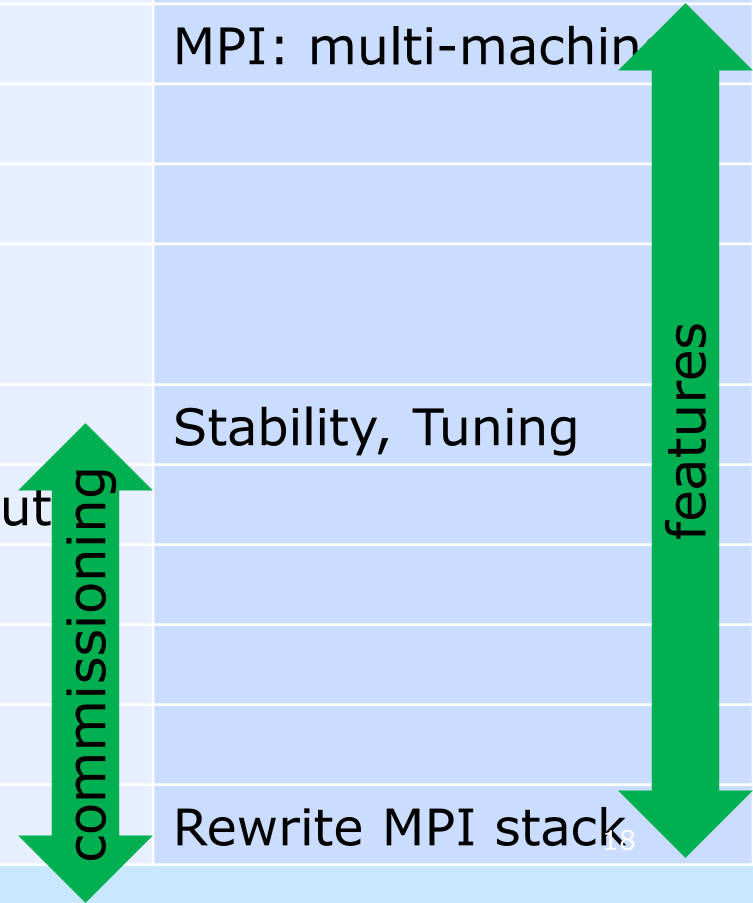


Software

- 4 coders (3 FTE)
- Project lead (management, 1 FTE)
- Project scientist (commissioning, 1 FTE)

- February 2013: Project start
- December 2013: Intended deadline
- March 2014: Delivery

Month	**Hardware**	**Development**	**Software**
2012			Prototype written
Feb ’13	Design	Sprints (3w), Agile	Refactor
Mar	Prototype	GTC 2013	
Apr	Air-flow guides	Automated Tests (Jenkins + Ctest)	Port OpenCL -> CUDA
May			
Jun	Arrives		MPI: multi-machin
Jul	Installed	Code reviews	
Aug	Configured, tuned		
Sep	System ready Network reconfig		
Oct	System back up		Stability, Tuning
Nov		One-click roll out	
Dec			
Jan ’14			
Feb	Performance drop		
Mar	iDRAC reboot	***Production***	Rewrite MPI stack



Refactoring proof-of-concept

- Research software -> development -> production
- From single 5 KLOC file to .hpp + .cpp per class
- Tests in separate sources
- No global variables

- Refactor before major changes:
    - Kill the God Class
    - Separate functionality in on purpose classes
    - Testable and maintainable

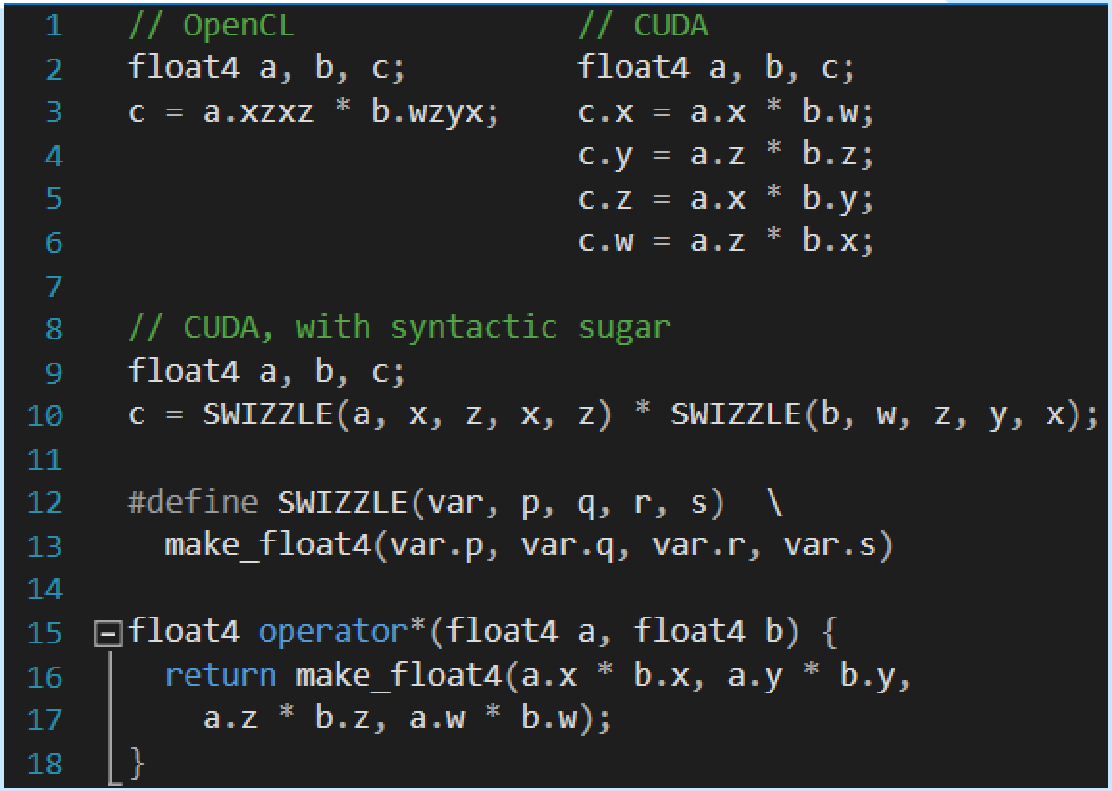
CUDA vs OpenCL

Feature	CUDA	OpenCL
AMD support	No	Yes (OpenCL 1.2)
NVIDIA support	Yes	Poor (OpenCL 1.1)
Vendor lock-in	Yes	No
Platform lock-in	Yes (GPU)	No (GPU, CPU, FPGA)
Debugger/profiler	Yes (Nsight)	Poor (CodeXL)
Learning material	Yes	Yes
Ease of use	Easy learning curve	Good syntactic sugar

Kernel performance difference: ~2%,
but CUDA also has GPUDirect, etc.

OpenCL -> CUDA port

- First a 1:1 port
    - ‘Easy’
    - Great way to learn
- Verify output and
- performance!



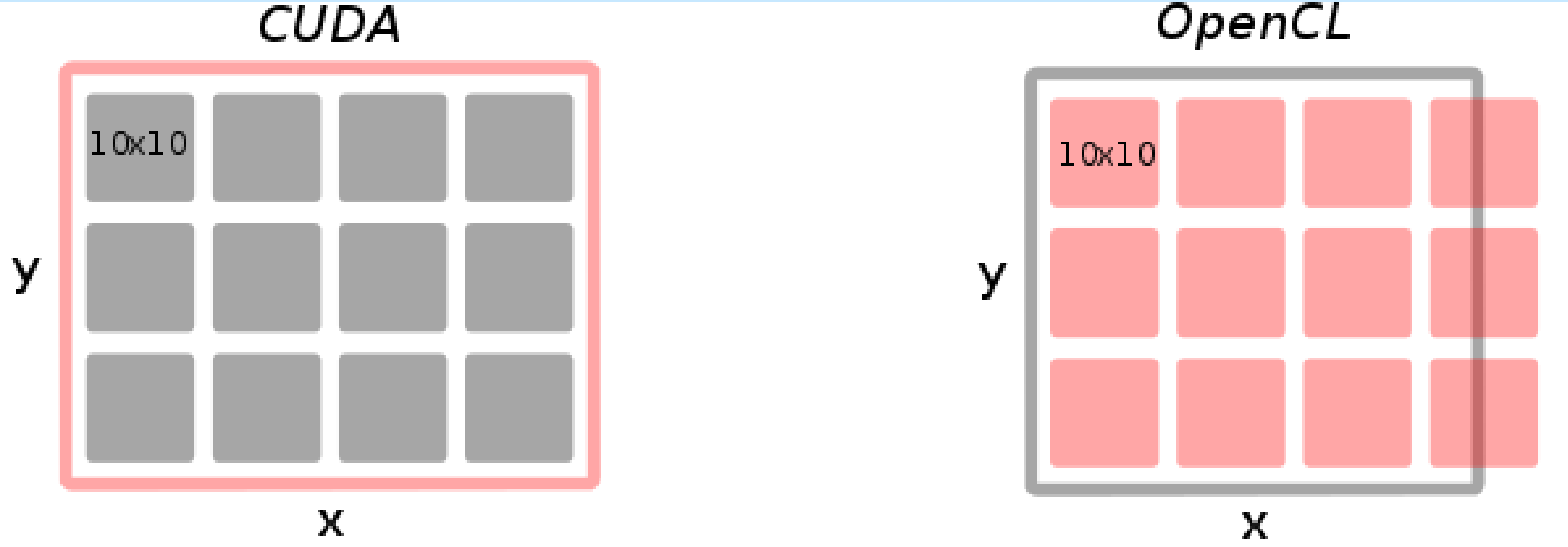
```
// OpenCL                // CUDA
float4 a, b, c;          float4 a, b, c;
c = a.xzxz * b.wzyx;     c.x = a.x * b.w;
                         c.y = a.z * b.z;
                         c.z = a.x * b.y;
                         c.w = a.z * b.x;

// CUDA, with syntactic sugar
float4 a, b, c;
c = SWIZZLE(a, x, z, x, z) * SWIZZLE(b, w, z, y, x);

#define SWIZZLE(var, p, q, r, s)  \
  make_float4(var.p, var.q, var.r, var.s)

float4 operator*(float4 a, float4 b) {
  return make_float4(a.x * b.x, a.y * b.y,
    a.z * b.z, a.w * b.w);
}
```

- Obstacles:
    - No SWIZZLE in CUDA -> compact code expands in port
    - No JIT in CUDA -> we can fake it
    - Terminology differences

CUDA vs OpenCL: Terminology

CUDA	**OpenCL**
GPU	Device
Global Memory	Global Memory
Shared Memory	**Local Memory**
Local Memory	Private Memory
Grid	Index Space
Block	Work Group

CUDA vs OpenCL: Work loads

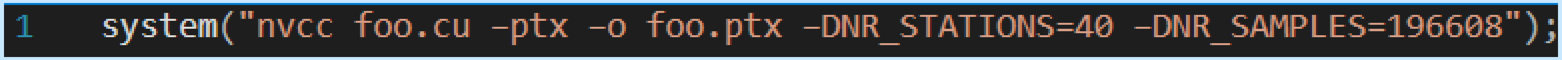


Define #blocks (4, 3)
Derives work load (40, 30)

Define work load (35, 30)
Derives #blocks (4, 3)

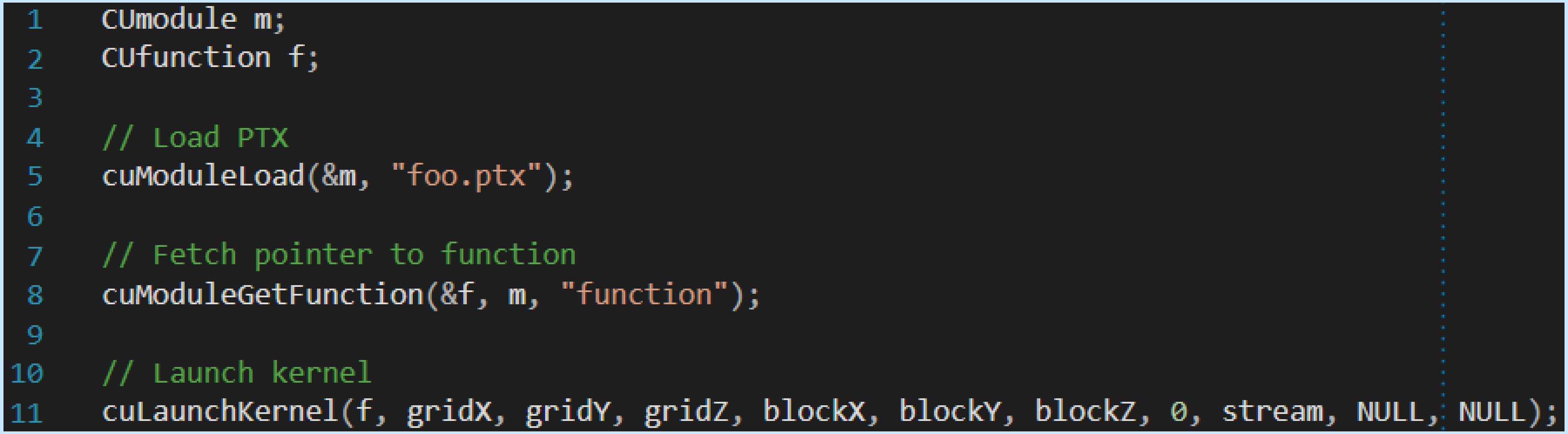
CUDA JIT compilation: How?

- Put your code in a .cu file
- Run nvcc from your program:



```
system("nvcc foo.cu -ptx -o foo.ptx -DNR_STATIONS=40 -DNR_SAMPLES=196608");
```

- Load the module, call the function (need Driver API...):



```
CUmodule m;
CUfunction f;

// Load PTX
cuModuleLoad(&m, "foo.ptx");

// Fetch pointer to function
cuModuleGetFunction(&f, m, "function");

// Launch kernel
cuLaunchKernel(f, gridX, gridY, gridZ, blockX, blockY, blockZ, 0, stream, NULL, NULL);
```

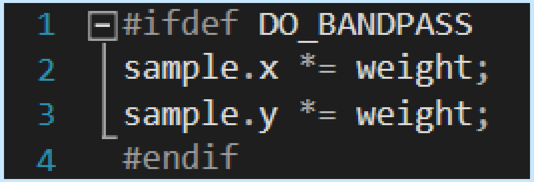
CUDA JIT compilation: Why?

JIT gives us the C preprocessor
to optimise code

- #define/nvcc -D: input parameters -> runtime constants

```
typedef  float2 InputData[NR_STATIONS][NR_SAMPLES][NR_POLARIZATIONS];
```

- #ifdef: Tune/skip functionality



```
#ifdef DO_BANDPASS
sample.x *= weight;
sample.y *= weight;
#endif
```

- Fewer instructions -> faster code
- Fewer registers needed -> more parallellism
- Fewer dynamic constructs -> simpler code

JIT caveats

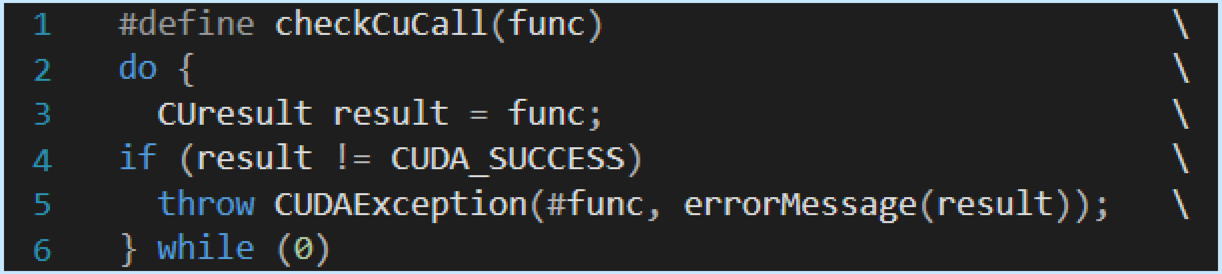
- fork() required to call nvcc.

- Problem: MPI stack is not fork() safe!
    - Solution: move all runtime compilation before MPI_Init.

- Problem: Parallel nvcc invocation caused crashes in nvcc
    - Solution: serialize & early initialization of run.

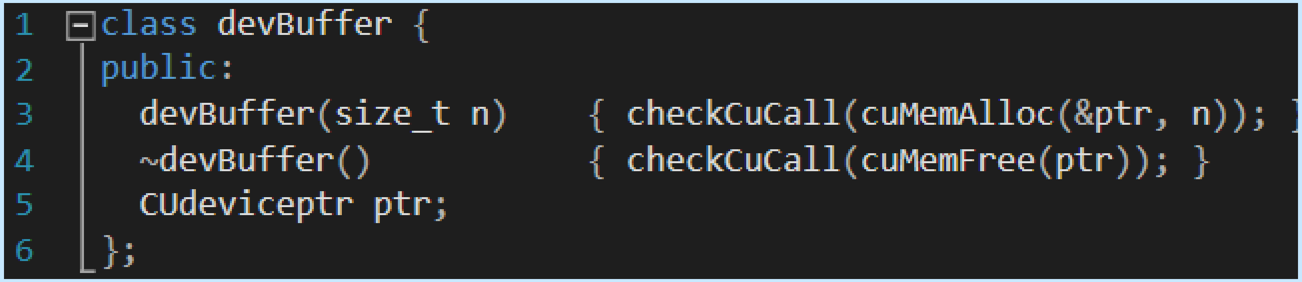
C++ CUDA abstraction layer (1)

- Abstraction layer on CUDA (and OpenCL)
  - Inspired by OpenCL C++ bindings
  - Wrap each resource in a class
- C++ exception handling -> no silent failure



```
#define checkCuCall(func)                             \
do {                                                  \
  CUresult result = func;                             \
if (result != CUDA_SUCCESS)                           \
  throw CUDAException(#func, errorMessage(result));   \
} while (0)
```

- C++ resource management -> no leaks



```
class devBuffer {
public:
  devBuffer(size_t n)    { checkCuCall(cuMemAlloc(&ptr, n)); }
  ~devBuffer()           { checkCuCall(cuMemFree(ptr)); }
  CUdeviceptr ptr;
};
```

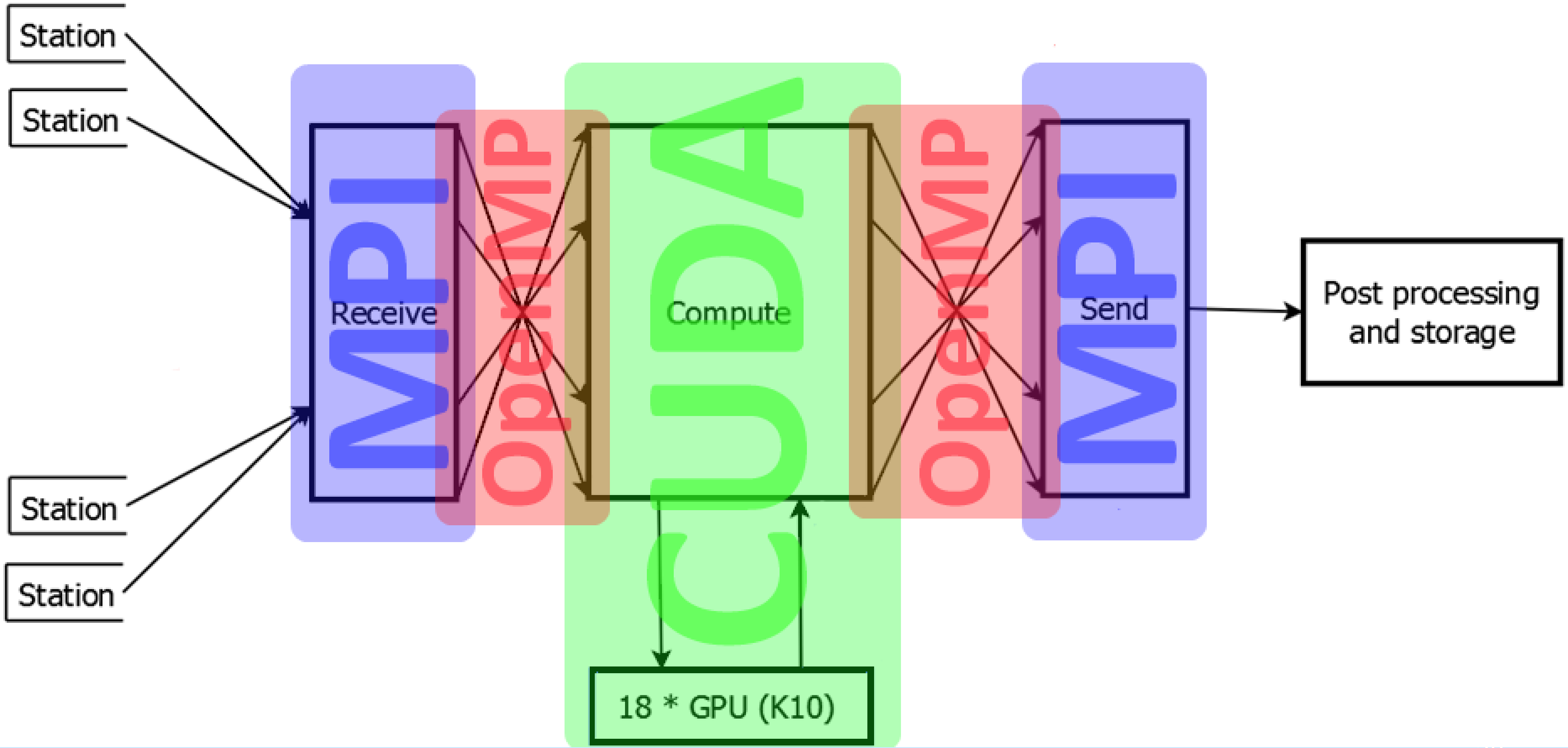
- Cleaner code, easier to debug, easier to test, simpler tests

More layers

- Rich “Kernel” class:
  - Run-time compilation
  - Buffer sizes & initialization
  - Execution
  - Performance monitoring
  - Sanity checks
- Pipelines chain “Kernel” classes
- Buffer classes combining GPU/CPU memory
  - ‘Automate’ transfers
  - Data inspection

Allows path back to OpenCL

Parallelization methods



Libraries libraries libraries

- We use many HPC libraries:
    - CUDA driver API (GPU parallisation…)
    - OpenMPI (parallisation over cluster, 1 process/CPU)
    - OpenMP (CPU core parallelisation)
    - Pthreads (CPU core parallisation)
    - LibNUMA (binds hardware used by process)
    - Casacore, HDF5, FFTW (astronomy/DSP)
    - LibSSH2 (remote process invocation)
    - POSIX (network/system programming):
        - Networking
        - Shared memory

CPU Multithreading

- We use:
    - 60% of CPU (`top’)
    - 53% of DRAM bandwidth (‘Intel PCM’)

- OpenMP + pthreads:
    - OpenMP merges parallellism into your control flow
    - Pthreads needed for background tasks

Multithreading caveats

Numerous libraries are not thread safe!

- OpenMPI still sensitive to forking, threading
  - Written for single-threaded applications

- Some libraries need global lock:
  - Casacore, HDF5
  - OpenMPI (also, MVAPICH2 in practice)

Library conflicts

Some libraries do not cooperate

- Libraries want their own allocation yet do similar things:
    - cuMemHostAlloc
    - MPI_Alloc
    - shmget

- OpenMPI + shared memory = leaks and crashes

Numerical Stability

- Slight instability (output jitter) unavoidable:
    - Differences in GPU architecture (Fermi, Kepler, etc.)
    - Differences in compiler (CUDA 4 vs 5, etc.)
    - Differences in compiler flags (--use-fast-math, etc.)
    - Code changes (optimizations, etc.)

- Careful analysis needed if output changes
    - Whether benign or critical
    - A newly blessed output might be in the order of GBytes!

Conclusions

- Hardware/software co-design = smooth operations
- Design choices depend on hardware + OS + libraries interoperability

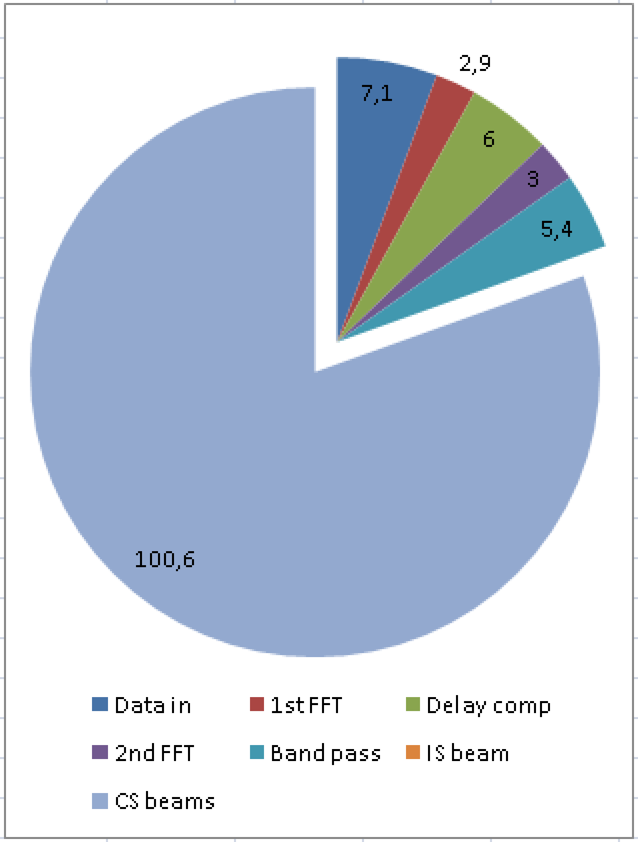
- JIT gives faster code
- OpenCL-like C++ wrapper provides cleaner code
- A GPU production cluster is more than CUDA alone

- 4 developers without GPU experience got COBALT in production in 1 year.

mol@astron.nl klijn@astron.nl

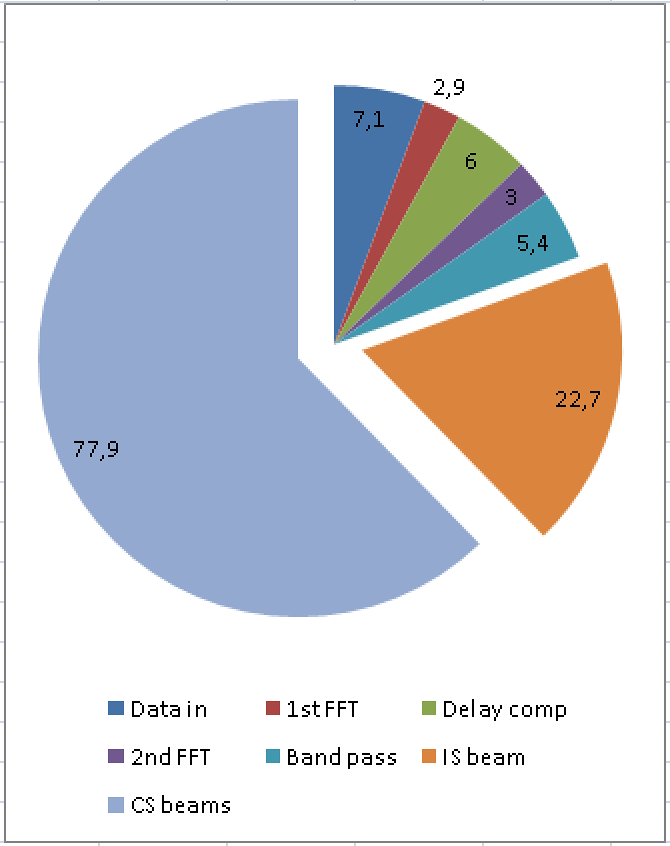
Beam Former: GPU Performance (16 bit, full bw)

CS



135 CS TABs

CS + IS



1 IS + 101 CS TABs