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

300m
Dutch antenna field,
Inset: low band antenna
Phased Arrays

Receiver

Receiving array

Physical delay

Combiner

Artificial delay

Output
Current compute cluster: IBM Blue Gene/P
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

Station
Station
Station
Station

80 * 3 GB*St/s = 240 GB/s

Receive
Compute
Send

240 GBit/s All to all
80 GBit/s All to all
80 GBit/s

Post processing and storage

18 * GPU (K10)

13.3 GBit/s Input
4.43 GBit/s Output (per GPU)
Hardware prototypes (Mar 2013)

First Design
Dell R720

Second Design
Dell T620

CPU1

CPU2

GPU1

GPU2

40GbE

PCIe balance load on QPI

LOFAR

IB1

IB2

2x 10GbE

LOFAR

IB1

IB2

2x 10GbE
Hardware Prototype

- **GPU idle temperatures:**

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Tesla K10.G2.8GB Off 0000:04:00.0 Off N/A 75C P0 43W / ERR! 0% 9MB / 3583MB 0% Default</td>
</tr>
<tr>
<td>1</td>
<td>Tesla K10.G2.8GB Off 0000:05:00.0 Off N/A 76C P0 42W / ERR! 0% 9MB / 3583MB 0% Default</td>
</tr>
<tr>
<td>2</td>
<td>Tesla K10.G2.8GB Off 0000:45:00.0 Off N/A 62C P0 42W / ERR! 0% 9MB / 3583MB 0% Default</td>
</tr>
<tr>
<td>3</td>
<td>Tesla K10.G2.8GB Off 0000:46:00.0 Off N/A 46C P0 36W / ERR! 0% 9MB / 3583MB 0% Default</td>
</tr>
</tbody>
</table>
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  | 0000:00:00:00 |
|                                      | N/A 48C P0 92W / ERR! | 2%               |
|                                      +-------------------+-------------------|
|                                      | 1 Tesla K10.G2.8GB Off  | 0000:00:00:00   |
|                                      | N/A 52C P0 91W / ERR! |                  |
|                                      +-------------------+-------------------|
|                                      | 2 Tesla K10.G2.8GB Off  | 0000:00:00:00   |
|                                      | N/A 51C P0 92W / ERR! |                  |
|                                      +-------------------+-------------------|
|                                      | 3 Tesla K10.G2.8GB Off  | 0000:00:00:00   |
|                                      | 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

The diagram illustrates the bidirectional bandwidth for inter-node communication as a function of message size in bytes. Two curves are shown: one representing the original system and the other the optimized system. As the message size increases, the bandwidth also increases significantly, with the optimized system showing a more gradual increase compared to the original system.
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)

![Pie charts showing data distribution for different configurations.](image-url)
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
<table>
<thead>
<tr>
<th>Month</th>
<th>Hardware</th>
<th>Development</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>2012</td>
<td></td>
<td></td>
<td>Prototype written</td>
</tr>
<tr>
<td>Feb ’13</td>
<td>Design</td>
<td>Sprints (3w), Agile</td>
<td>Refactor</td>
</tr>
<tr>
<td>Mar</td>
<td>Prototype</td>
<td>GTC 2013</td>
<td></td>
</tr>
<tr>
<td>Apr</td>
<td>Air-flow guides</td>
<td>Automated Tests (Jenkins + CTest)</td>
<td>Port OpenCL -&gt; CUDA</td>
</tr>
<tr>
<td>May</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jun</td>
<td>Arrives</td>
<td>Code reviews</td>
<td>MPI: multi-machine</td>
</tr>
<tr>
<td>Jul</td>
<td>Installed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aug</td>
<td>Configured, tuned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sep</td>
<td>System ready</td>
<td>Code reviews</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Network reconfig</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Oct</td>
<td>System back up</td>
<td>One-click roll out</td>
<td>Stability, Tuning</td>
</tr>
<tr>
<td>Nov</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jan ’14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Feb</td>
<td>Performance drop</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mar</td>
<td>iDRAC reboot</td>
<td>Production</td>
<td>Rewrite MPI stack</td>
</tr>
</tbody>
</table>

補充：
- **Project Time Line**
- **Hardware**
- **Development**
- **Software**
- **Features**
- **Commissioning**
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

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

Kernel performance difference: \(\sim 2\%\),
*but* CUDA also has GPUDirect, etc.
OpenCL -> CUDA port

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

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

<table>
<thead>
<tr>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Device</td>
</tr>
<tr>
<td>Global Memory</td>
<td>Global Memory</td>
</tr>
<tr>
<td>Shared Memory</td>
<td><strong>Local Memory</strong></td>
</tr>
<tr>
<td><strong>Local Memory</strong></td>
<td>Private Memory</td>
</tr>
<tr>
<td>Grid</td>
<td>Index Space</td>
</tr>
<tr>
<td>Block</td>
<td>Work Group</td>
</tr>
</tbody>
</table>
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:

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

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

```c
CUmodule m;
CUfunction f;

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

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

// Launch kernel
cuLaunchKernel(f, gridSize, gridSize, gridSize, blockIdx, blockIdx, blockIdx, 0, stream, NULL, NULL);
```
CUDA JIT compilation: Why?

- `#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 parallelism
- 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

```cpp
#define checkCuCall(func) 
  do {
    CResult result = func;
    if (result != CUDA_SUCCESS) 
      throw CUDAEException(#func, errorMessage(result)); 
  } while (0)
```

- C++ resource management -> no leaks

```cpp
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
We use many HPC libraries:

- CUDA driver API (GPU parallelisation...)
- OpenMPI (parallelisation over cluster, 1 process/CPU)
- OpenMP (CPU core parallelisation)
- Pthreads (CPU core parallelisation)
- 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 parallelism 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)

135 CS TABs  1 IS + 101 CS TABs