SDP Memo 091: Antenna gain calibration on Graphical Processing Units

Document number……SDP Memo 091
Document Type……MEMO
Revision……...1
Author……..NVIDIA Corporation
Release Date……...2018-10-25
Document Classification……..…… Unrestricted

Lead Author:
NVIDIA Corporation

Released by:

Name	Designation	Affiliation
Bojan Nikolic	SDP Project Engineer	University of Cambridge
Signature & Date:		

SDP Memo Disclaimer

The SDP memos are designed to allow the quick recording of investigations and research done by members of the SDP. They are also designed to raise questions about parts of the SDP design or SDP process. The contents of a memo may be the opinion of the author, not the whole of the SDP.

Antenna gain calibration on Graphical Processing Units

NVIDIA Corporation, 2701 San Tomas Expressway,
Santa Clara, CA 95050 USA

---

1. Antenna gain

In practical radio astronomy, the response of the various antennae to celestial stimuli are uncertain and vary with time. Reliable imaging requires periodic calibration of the gains of each antenna against a known source. The appropriate gain of each antenna can be discovered by examining the response of the system to a known stimulus. The response of the p-th antenna to Q sources is assumed to take the form

$$x_p(t) = g_p \sum_{q=1}^{Q} a_{p,q} s_q(t) + n_p(t) \tag{1}$$

where g_p is the gain of the p-th antenna, $a_{p,q}$ is the response of the p-th antenna to the q-th source, $s_q(t)$, and $n_p(t)$ represents noise on the p-th antenna.

As illustrated by Salvini and Wijnholds[1], we can minimize the Frobenius norm of the noise, by assigning g_p to minimize

$$\sum_{i,j} |\hat{R}_{i,j} - g_i M_{i,j} g_j|^2 \tag{2}$$

where $\hat{R}$ is the array covariance matrix

$$\hat{R}_{i,j} = \frac{1}{K} \sum_{k} x_i(kT) x_j^*(kT) \tag{3}$$

and M is given by

$$M_{i,j} = \sum_{q} \sum_{p} \sum_{k} a_{i,q} s_q(kT) s_p^*(kT) a_{j,p}^*. \tag{4}$$

In the above expressions, T is the sampling period. We assume the sources and the response are known at a set of regularly-spaced times, kT.

Algorithm 1: StEFCal

foreach *iteration, i* **do**
  for *p = 1, 2, ... P* **do**
    $z_q \leftarrow g_q^{[i-1]} M_{p,q}$
    $g_p^{[i]} \leftarrow (\sum_q \hat{R}_{q,p}^* z_q)/(\sum_q z_q^* z_q)$
  end
end
if $mod_2(i) = 0$ **then**
  $g_p^{[i]} = (g_p^{[i]} + g_p^{[i-1]})/2$
end

1.1. Basic Algorithm

Salvini and Wijnholds present an alternating least-squares algorithm for finding the optimal antenna gains. They refer to this algorithm as "Statisically Efficient and Fast Calibration," or StEFCal. The iteration loop can be described as in Algorithm 1.

There may also be a stopping criterion, which is not shown in the algorithm.

1.2. GPU implementation

Calibration on the GPU is relatively straightforward. We chose, initially, to assign one threadblock to compute the gain for one antenna. Each thread is assigned one or more values in the dot products. Following the computation by individual threads, the values from each thread in the block are "reduced" to compute a single complex gain. The number of threads in each block can be varied. If the number of threads is less than the number of elements, then each thread computes more than one term in the dot product. The number of threadblocks can also be varied if each threadblock computes more than one gain. Both of these changes can improve the number of independent instructions, allowing for better latency hiding.

1.3. Precomputing

In algorithm 1, the write to the temporary storage, z_q, is unnecessary. The expression for $g_p^{[i]}$ can be alternatively expressed as

$$g_p^{[i]} = (\sum_q \hat{R}_{q,p}^* M_{p,q} g_q)/(\sum_q g_q M_{p,q} M_{p,q}^* g_q^*) \tag{5}$$

Here, the $[i-1]$ superscripts have been dropped. All of the g_p on the right-hand side refer to the gains computed in the last iteration.

We can precompute matrices A and B given by

$$A_{p,q} = \hat{R}^*_{q,p} M_{p,q} \tag{6}$$

and

$$B_{p,q} = M_{p,q} M^*_{p,q}. \tag{7}$$

Then, expression (5) becomes

$$g_p^{[i]} = (\sum_q A_{p,q} g_q)/(\sum_q g_q B_{p,q} g_q^*). \tag{8}$$

From the standpoint of computation, this is a more favorable expression. The matrices, A and B can be read in a coalesced fashion and the matrix B is purely real. In principle, the set of real numbers $g_p g_p*$ can also be computed once, but the performance impact of this is not favorable since the algorithm is dominated by data movement. The repeated computation can be hidden underneath data memory reads and writes.

This change improves performance of the CPU-only implementation as well.

2. Performance

The GPU implementation of this algorithm performs 300 calibration iterations for 1000 elements in 25 ms. In order to avoid data-dependence in the runtime, we ran the tests described in this report with a fixed number of iterations. Adding a stopping criterion will increase runtime. All tests are using Tesla K40c with boost clocks and ECC off. The time reported is only for the calibration iterations. The generation of the $\hat{R}$ and M matrices is not included.

Optimal performance uses 128 threads per threadblock and 1000 threadblocks. The bandwidth between the L1 cache and the L2 cache is 70% of peak performance. The hit rate in the L2 cache is 40%, a good result, but one that is likely to drop with larger problem sizes.

The computational throughput is just 35%, indicating that data movement is the limiting factor.

This is dramatically faster than CPU-only implementation parallelized using OpenMP. A comparison to a well-tuned CPU implementation has not been made.

The code used for these tests can be found in the Science Data Processor git repository[1].

3. Optimization

3.1. Texture Load

A small performance improvement can be made by loading the gains from the previous iteration through the texture path. With CUDA, this is a very simple change. If the standard load is

```
cuComplex thisgain = g[z];
```

then, to instruct the compiler to load this through the texture path, we change this line to

```
cuComplex thisgain = __ldg(&g[z]);
```

This will change the path the data travels to get from the L2 cache to the registers. The first line will send the data through the L1 cache with a standard global load instruction. The second will compile to a texture load instruction and pass the data through the texture path. Because reuse of the gains is much higher than the rest of the data, it makes sense to pass this data through the texture cache to avoid evicting it in favor of data which is less likely to be reused.

3.2. Independent warps

The reduction step requires some synchronization of the individual threads in the threadblock. This can limit parallelism as threads stall waiting on the remaining threads in the threadblock to reach the sync point. To avoid this, we can write our kernels so that contiguous groups of 32 threads, called warps, are independent. Threads in a warp operate synchronously as a result of the GPU architecture so no additional synchronization is necessary when a warp works independently on a single computation, even if the threadblock contains more than one warp.

[1] https://github.com/SKA-ScienceDataProcessor/GPULeastSquaresSolve

With this in mind, we modified the code to allow each warp to compute one dot product independently. In practice, we set the width (blockDim.x) of the threadblock to 32 and the height (blockDim.y) to 4, for a total of 128 threads per block. Each row of the threadblock computes a different dot product.

The result of this change is actually a slight slowdown because the hit rate in the texture cache falls with this new approach.

4. Performance for varying problem size

Figure 1 shows the performance for varying problem size. All measurements were made with 300 iterations. Performance rises as more elements provides more parallelism, then begins to fall as larger array sizes begin to exhaust the caches.

The above analysis neglects the time for computing the A and B matrices. This step is dominated by matrix multiplication which scales as $\mathcal{O}(n^3)$. For very large numbers of antennas, generation of $\hat{R}$ and M matrices on GPU dominates the runtime.

5. Batched execution

For some systems, it may make sense to calibrate more than one set of gains at once. For example, a system may be collecting images at more than one frequency of more than one polarization at the same time. In such cases, the gains for several systems may be computed simultaneously. Figure 2 shows the performance for the calibration of several systems at once, varying the number of systems to be simultaneously calibrated. Data are shown for sets of 500 and 2000 antennas. In the case of 500 antennas, parallelism is clearly not sufficient for peak performance for a single calibration. But as more calibrations are batched, performance, measured in GFLOPS, rises, achieving 95% of best performance with 20 batched calibration problems. This peak performance, 168 GFLOPS, is short of the performance for a single calibration for 2000 antennas, but is more than double the performance for a single calibration for 500.

6. Conclusion

Antenna calibration on the GPU is relatively simple to implement and is limited by data movement. Some small changes can be made to reduce data

traffic and to make better use of caches, resulting in a 30% improvement over a simple implementation. Best performance requires a few thousand antennas. Performance falls slowly beyond about 5000 antennae. Better performance with fewer antennas can be achieved by batching several calibration computations together.

7. References

[1] Stefano Salvini and Stefan J. Wijnolds. Fast gain calibration in radio astronomy using alternating direction implicit methods: Analysis and applications *Astronomy and Astrophysics* 571, A97, 2014.

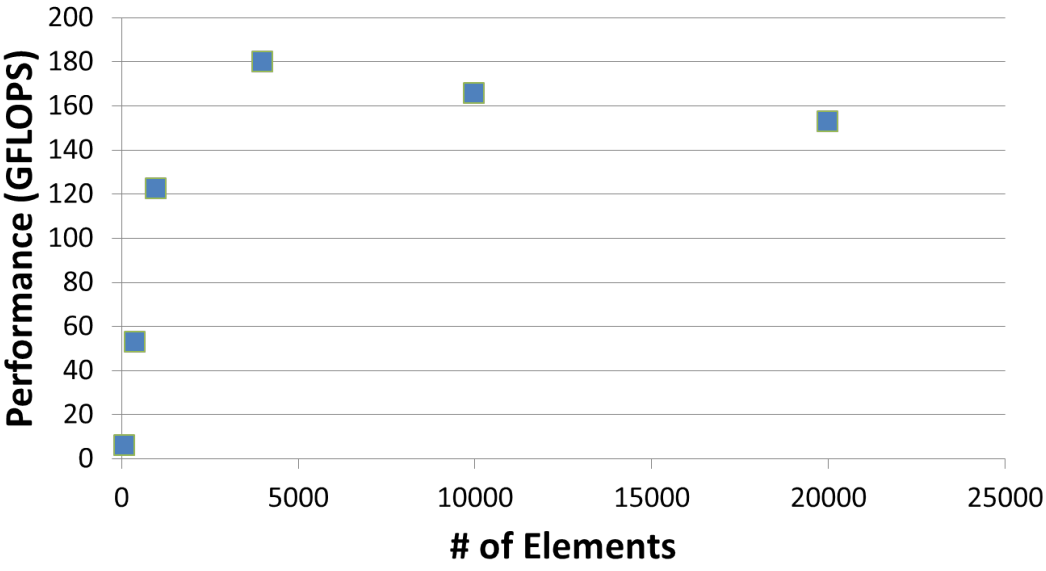

Figure 1: Performance for GPU-StEFCal as a function of the number of elements

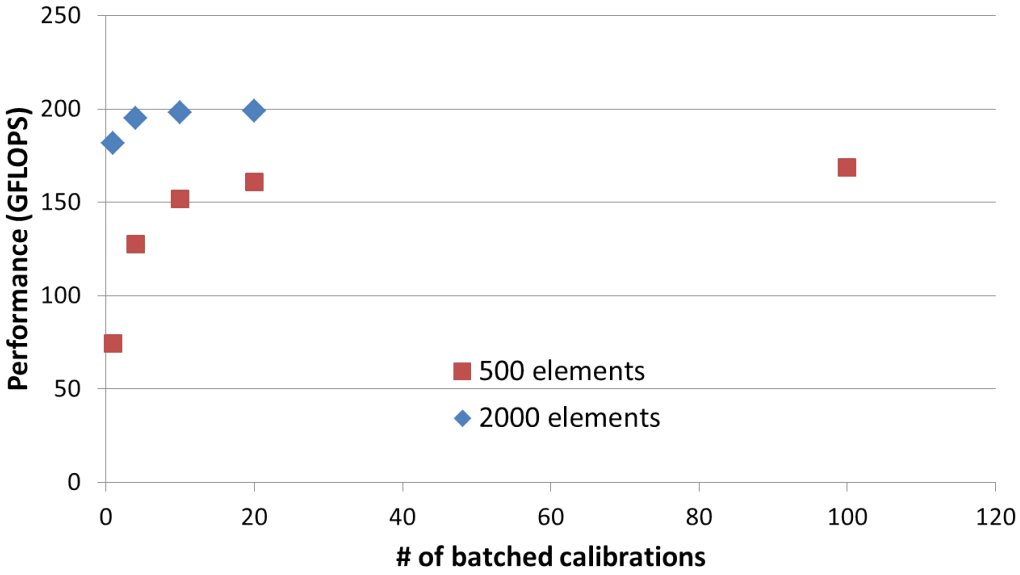

Figure 2: Performance for GPU-StEFCal as a function of the number of simultaneously calibrations for 500 and 2,000 antennas.