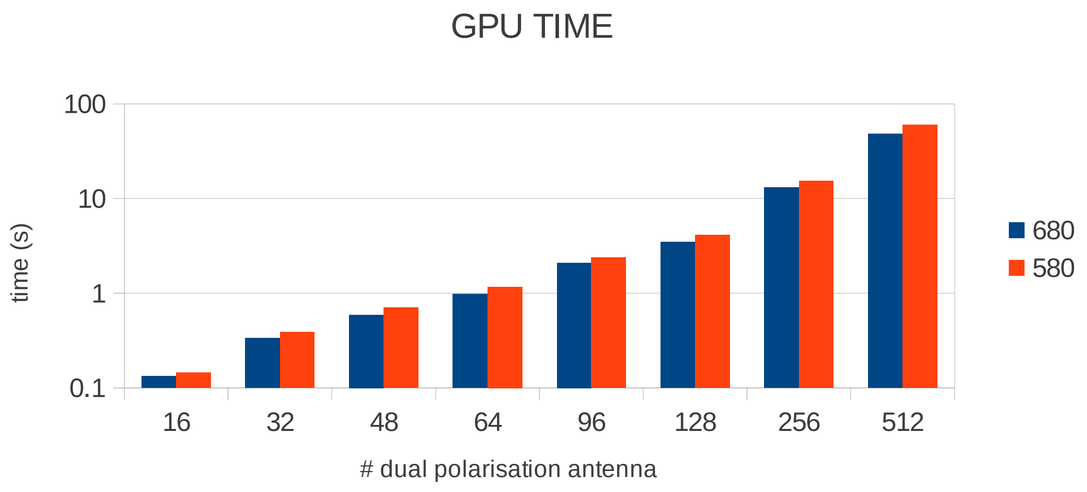
Profiling xGPU code on the GTX580 and GTX680

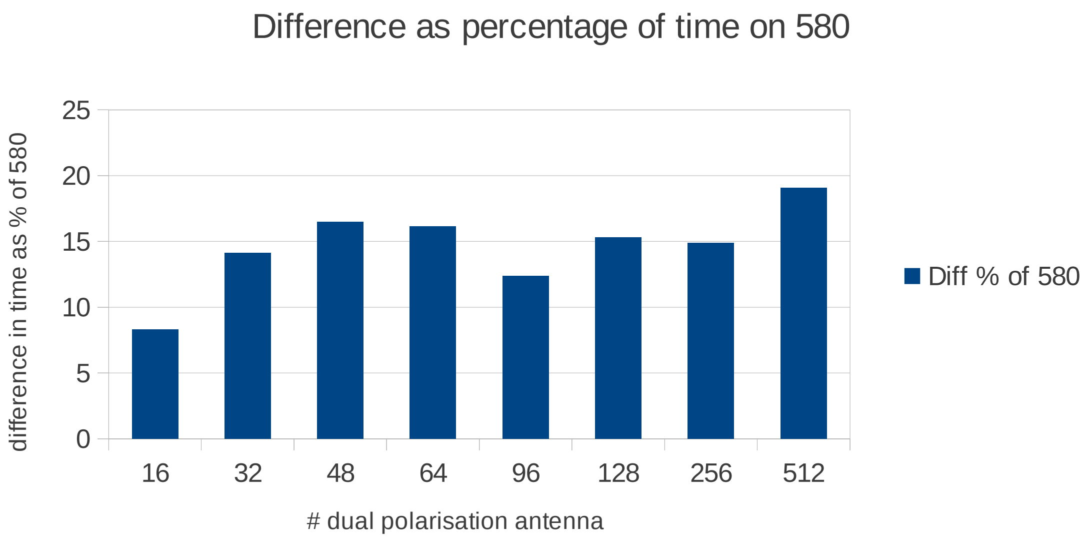
Christopher Schollar 09/07/2012

Over the last week I conducted a number of tests on running the xGPU code on the GTX580 and the GTX680. I was using the cuda-correlator benchmarker which was given to me by Dave MacMahon which essentially generates random data on a CPU, pushes it onto the GPU and correlates the data. I ran my tests on a number of different configurations.

For all the results here I ran X-engines working on 64 separate channels and changed the number of antenna cross correlated. Each antenna provides two polarisations, so for a test with 32 antenna we are correlating 64 inputs. The number of samples correlated in each case is 256 * Nantennas * 2Polarisations * Nchannels, with each sample being a 2 byte complex sample.

One thing we realised during the process is that you must use a multiple of 16 as the number of antennas you are inputting or the code falls over. This is because of implementation details which I don't really understand. This isn't a problem for PAPER or LEDA. If you wanted to use a non-multiple of 16 apparently you can pad your data, sending through 0's for all antenna between your total number of antenna and the next multiple of 16.

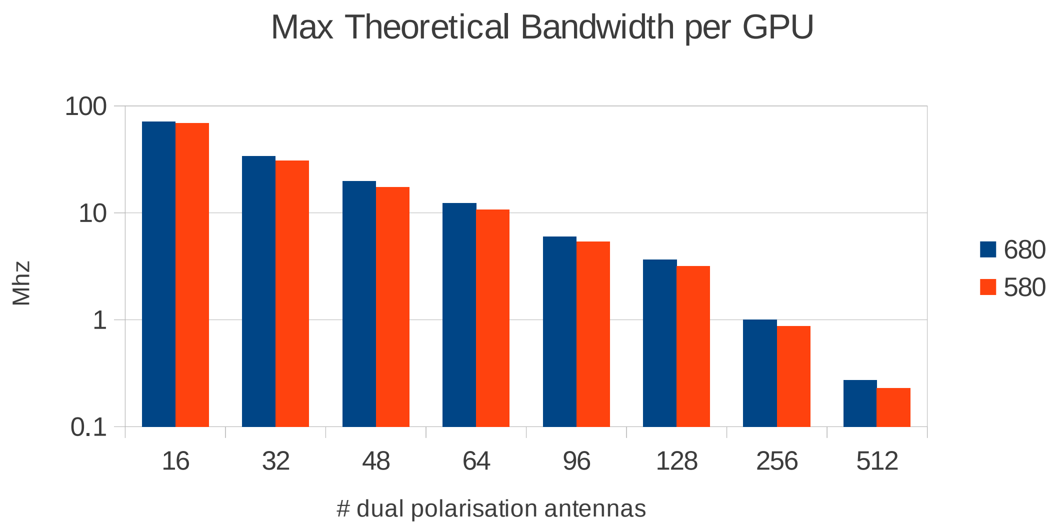


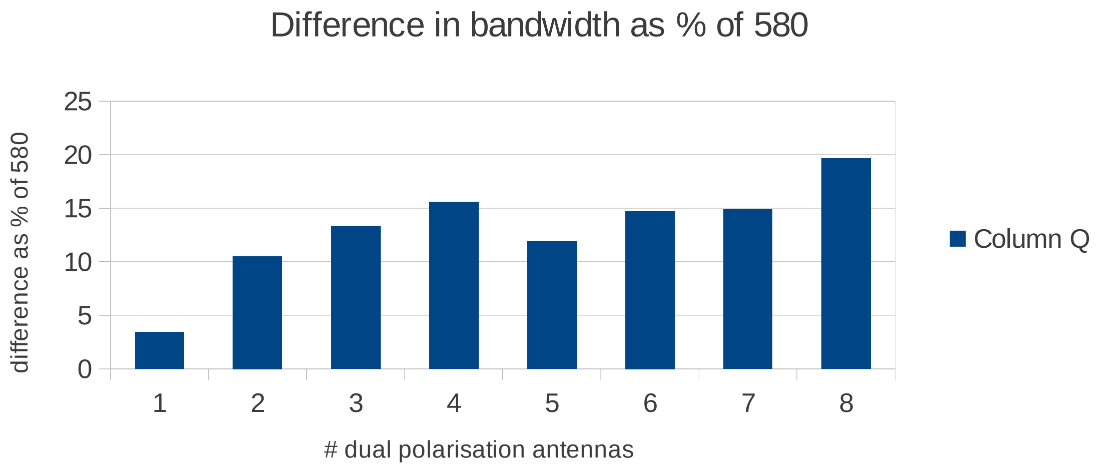


The graphs show that the GTX680 is faster than the GTX580 for all values I tested. It seems we can reliably expect a 10-20% increase in the speed of computation. This is before any real changes have been made to the code to take advantage of the new features of the Kepler architecture. The only difference between the code run on the GTX580 and that run on the GTX680 is that I used the compiler flag -arch=sm_20 for the 580 and -arch=sm_30 for the 680.

The 580 has 512 cuda cores running at 1544 Mhz and the 680 has 1536 cores running at 1006 Mhz. As the problem is compute bound (at least for 64+ antennas), we would expect to get an increase of something like 1.95x speed-up over the 580 if the code were optimised for the 680. Something to consider is that the 580 uses 244 Watts while the 680 uses 195 Watts with a price difference of around $50 on Amazon.

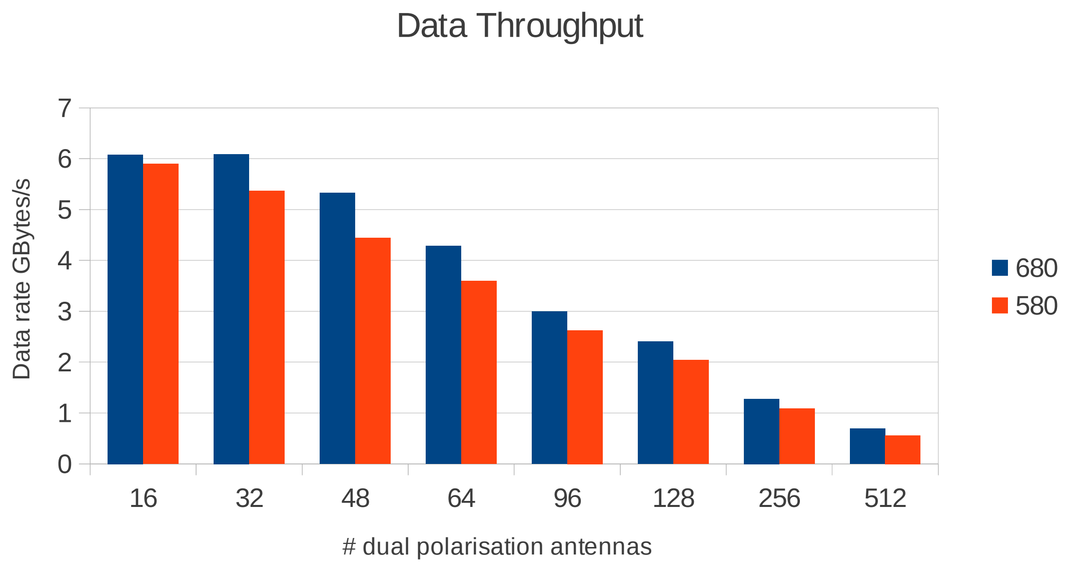
Below are the tables of how the different processors differ in terms of the maximum bandwidth they can process as part of a correlator. This data was collected using the output of the cuda-correlator benchmark which calculates the maximum theoretical bandwidth as well as timing the code.





Data Throughput

In the chart below I show how the computation time starts to dominate the memory transfer time at 16 antenna for the GTX580 and 48 for the GTX680. The maximum speed that the bus between the CPU and GPU can transfer is around 6.7GBytes/s. These tests were run on data randomly generated by the CPU, it is only testing the throughput from the CPU to the GPU. The real correlator has other factors which affect the speed at which data gets to the GPU, and these tests do not include the affects of collecting data from multiple F-Engines, transporting that data from the NIC to the CPU or converting the data into a format suitable to be fed into the GPU. In these tests the GTX 580 was connected to a Supermicro X8DTG-QF motherboard and the GTX 680 was connected to a Supermicro X9SRA.



In the xGPU code each complex sample is 2 bytes, 1 byte of real data and 1 byte of imaginary data.

As the memory transfers to the GPU can be performed asynchronously, I got these numbers by dividing the amount of data transferred by the maximum of the time to run the memory transfer and the cross correlation kernel.

When the time for memory transfer is longer than the cross correlation kernel, then the problem is IO-limited. In this case the number here should be a significant percentage of the maximum transfer rate, which is the case for 16/32 antenna.

When the time for the correlation is longer than the time for memory transfer then the problem is compute bound. In this case the number I calculated shows the speed data needs to be transferred to the GPU to make sure it always has the data it needs to compute. The memory transfer always transfers data at around 6 GBytes/s, however the number here shows the speed that the GPU is 'consuming' data.

These numbers do not include the time to transfer data from the GPU back to the CPU. I chose to do this as this transfer is performed synchronously and must be performed after computation is complete. This is the case for the cuda_correlator benchmark, however it is not necessarily true for all cases.

Tables
I have included the tables I made the charts from for those of you who like numbers

Time for the cross-correlation kernel

# Antennas	680 (s)	580 (s)	difference (ms)	Difference as % of 580
16	0.13	0.15	0.01	8.29
32	0.34	0.39	0.06	14.13
48	0.59	0.71	0.12	16.49
64	0.98	1.17	0.19	16.14
96	2.10	2.40	0.30	12.40
128	3.49	4.12	0.63	15.30
256	13.11	15.40	2.30	14.90
512	48.30	59.68	11.38	19.06

GPU data consumption (GBytes/s)

# Antennas	680 Throughput GBytes/s	580 Throughput GBytes/s
16	6.08	5.89
32	6.09	5.37
48	5.32	4.44
64	4.28	3.59
96	3.00	2.62
128	2.41	2.04
256	1.28	1.09
512	0.69	0.56

Max Theoretical Bandwidth (Mhz)

# Antennas	680 (Mhz)	580 (Mhz)	Difference as % of 580
16	71.52	69.14	3.44
32	34.21	30.95	10.51
48	19.82	17.48	13.36
64	12.35	10.69	15.59
96	6.00	5.36	11.95
128	3.65	3.18	14.71
256	1.00	0.87	14.87
512	0.27	0.23	19.65

Profiling Nvidia GPU's
In order to profile the xGpu code I used the Nvidia command line profiler. Although I suggest using the visual profiler when possible. However I couldn't use the visual profiler as I was sshing onto a machine at the NRAO in Green Bank, it was not possible to run the visual profiler on the NRAO machine as it became unusable due to latency. Also, the Cuda 4.0 toolkit uses a different visual profiler to the Cuda 4.2 toolkit, so if you want to compare code across the different platforms you have to learn how to use two profilers.

In order to profile with the command line you need to set a few environment variables.

```
export COMPUTE_PROFILE=1
```
This enables profiling for any GPU code that you run.

```
export COMPUTE_PROFILE_CSV=1
```
This ensures that the format of the profiler output is in CSV format. This is a personal preference of mine. The profiler automatically saves in a Key-Value-Pair format. I wanted to further analyse the data with a spreadsheet program so it seemed like a better plan to output as csv.

```
export COMPUTE_PROFILE_LOG=[log-file-name]
```
This sets the file name (and path) of the file which will contain the profiling information after your code is run

```
export COMPUTE_PROFILE_CONFIG=[profiling-config-filename]
```
This sets the name of the configuration file which describes which counters to use to profile your code.

After you have set these variables every time you run code on the GPU from your environment, the profiler will save the values of the counters you specify in your configuration file into the log file you specify.

Configuration Files
The configuration files are very simple. You simply write down the name of each counter you want profiled. You put each counter name on a new line. One thing to remember is that you can only profile 4 counters at a time, so it is likely you will want to run multiple configurations to get all of the information you want.

You can find all of the possible counters in the Compute_command_line_profiler_user_guide which is in the Nividia GPU Computing SDK. If you don't know where this is saved on the computer you are using it is very easy to find on google.

Most of the counters are not particularly useful on their own, what you really want to get is the derived statistics which are automatically computed when you use the Compute Visual Profiler (which is why you should use the visual profiler whenever you can). However, all of the statistics which are automatically calculated by the visual profiler are explained in the Compute_visual_profiler_user_guide.pdf under the supported derived statistics section. This guide is also freely available online.

Example configuration file

```
gpustarttimestamp
gpuendtimestamp
fb_subp1_read_sectors
fb_subp1_write_sectors
memtransfersize
memtransferdir
```