SDP Memo 088: Characterizing FFT performance on GPUs for SKA-SDP

Document number……SDP Memo 088
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.

Characterizing FFT performance on GPUs for SKA-SDP

Problem Summary

Throughout the processing of radio telescope data, images must be moved between real space and fourier space. GPUs can be used for this computation. NVIDIA has written and optimized a library, called cuFFT, for this purpose.

2D FFTs using cuFFT

To perform a 2-dimensional FFT in CUDA C using cuFFT, we first create a plan and allocate and populate GPU memory, then execute that plan one or more times. A typical syntax might look like this:

```
float data[2*SIZE], *d_data;

cufftHandle plan;

cudaMalloc(&d_data, sizeof(float)*2*SIZE);

cudaMemcpy(d_data, data, sizeof(float)*2*SIZE);

cufftPlan2D(&plan, SIZE, SIZE, CUFFT_C2C);

cufftExecC2C(plan, d_data, d_data, CUFFT_FORWARD);
```

The above is an in-place forward FFT. The code should end by freeing the GPU memory and destroying the cufftPlan.

The cufftPlan allocates some temporary memory. The size of the temporary memory required is given by cufftGetSize2d.

FFT is typically limited by the memory bandwidth of the architecture. This is the case for GPUs as well.

Performance testing cuFFT

The code used to measure the performance of cuFFT can be found at the following location:

https://github.com/SKA-ScienceDataProcessor/cuFFTBenchmark

In the following charts, the performance numbers for cuFFT assume 5mn (log(n)+log(m)) floating point operations, for a 2-dimensional FFT of an m x n image. Here are the performance numbers for in-place double precision complex-to-complex transforms. N and m are the two dimensions of the image. Storage is the temporary storage used. Elapsed time is in milliseconds. Gcell/s is the number of elements processed per seconds. It is given by nm/time/1e6. Gflps is giga floating point operations per second assuming the complexity discussed above.

N	M	storage(MB)	elapsed	Gcell/s	Gflps
8192	8192	1073	38.4563	1.74507	226.859
4096	4096	268	8.92633	1.87952	225.542
10000	10000	1600	79.6815	1.255	166.76
10000	10001	6480	287.804	0.347494	46.1742
10016	10000	6480	301.883	0.331784	44.0904
10000	10016	6480	286.983	0.349011	46.3796
10016	10016	6490	484.109	0.207227	27.5405
10002	10000	6480	301.768	0.331447	44.0422
10001	10000	6480	301.798	0.33138	44.0331
10003	10000	6480	301.797	0.331448	44.0426
10002	10002	6481	485.397	0.206099	27.3865
10004	10004	1601	239.069	0.418624	55.6279
10005	10005	1601	175.75	0.569558	75.6853
10010	10010	1603	177.927	0.563152	74.8381
10011	10011	1603	312.579	0.320623	42.6086
10032	10032	1610	159.828	0.629682	83.6994
10030	10030	1609	235.255	0.427624	56.8399
10040	10040	6505	484.725	0.207956	27.6446
10050	10050	1616	268.807	0.375743	49.9547
8192	16384	2147	80.9569	1.65789	223.815
16384	10000	2621	128.787	1.27217	173.574
10000	16384	2621	118.323	1.38468	188.924
9984	9984	1594	107.34	0.928639	123.373
9500	9500	1444	102.863	0.877379	115.934
16	16	0	0.055137	0.004643	0.185719
30	30	0	0.021466	0.041928	2.05734
32	32	0	0.063059	0.016239	0.811935
64	64	0	0.06197	0.066096	3.96578
100	100	0	0.02269	0.44072	29.2808
128	128	0	0.062684	0.261376	18.2963
100	128	0	0.042409	0.301826	20.5904
128	100	0	0.046457	0.275526	18.7962
256	256	1	0.064238	1.02021	81.6167
1000	1024	16	0.467341	2.19112	218.737
1024	1000	16	0.461392	2.21937	221.557
1024	1024	16	0.464493	2.25746	225.746

Very close to m=n=10000 there does not seem to be much to be gained by zero-padding the data. Some padded sizes (i.e. 10000 x 16384) have better performance in terms of Gflps, but the elapsed time is higher than for 10000x10000. However, if the entire image can be sized to a power of two such as 8192 or 16384, substantial performance can be gained. Out-of-place execution, in which the

output data is sent to a different location than the input, does not seem to affect performance above the level of noise.

Here is a slightly truncated chart for single precision.

N	M	storage(MB)	elapsed	Gcell/s	Gflps
8192	8192	536	20.0081	3.35409	436.032
4096	4096	134	3.44475	4.87037	584.445
10000	10000	800	42.4837	2.35385	312.772
10004	10004	800	157.131	0.63692	84.6357
10005	10005	800	120.301	0.832081	110.571
10010	10010	801	109.67	0.913652	121.417
10011	10011	801	216.015	0.463949	61.6556
10032	10032	805	99.5626	1.01083	134.363
10030	10030	804	137.087	0.733849	97.5434
10050	10050	808	150.515	0.671044	89.2147
9984	9984	797	64.7087	1.54045	204.654
9500	9500	722	61.7897	1.4606	192.999
16	16	0	0.049854	0.005135	0.2054
30	30	0	0.023202	0.03879	1.90336
32	32	0	0.061683	0.016601	0.830048
64	64	0	0.059665	0.06865	4.11899
100	100	0	0.021923	0.456138	30.3051
128	128	0	0.05936	0.276011	19.3208
100	128	0	0.040466	0.316314	21.5787
128	100	0	0.043886	0.291666	19.8972
256	256	0	0.058586	1.11864	89.4909
1000	1024	8	0.238399	4.29532	428.797
1024	1000	8	0.243493	4.20545	419.826
1024	1024	8	0.233341	4.49375	449.375
8192	16384	1073	36.5583	3.67134	495.63
16384	10000	1310	65.6498	2.49567	340.505
10000	16384	1310	56.0965	2.92068	398.493
16384	16384	2147	86.9907	3.0858	432.011

Exploiting Matrix Symmetry

In many cases, the image either before or after the transform is Hermitian. In these cases, this symmetry can be exploited to save computational work. A Hermitian matrix implies that the transformed matrix is strictly real (not complex). cuFFT provides libraries for real-to-complex and complex-to-real transforms which take advantage of the Hermiticity of the matrix.

Here is the performance chart for double precision complex-to-real transforms.

N	M	storage(MB)	elapsed	Gcell/s	Gflps

8192	8192	536	22.6511	2.96272	385.154
4096	4096	134	4.79626	3.49798	419.758
10000	10000	800	46.3461	2.15768	286.706
10000	10001	6480	259.358	0.385606	51.2385
10016	10000	6480	175.989	0.569125	75.6302
10000	10016	6480	260.271	0.384829	51.1394
10016	10016	6490	348.66	0.287731	38.2395
10002	10000	6480	175.89	0.568651	75.5615
10001	10000	6480	175.867	0.568669	75.5636
10003	10000	6480	175.912	0.568636	75.56
10002	10002	6481	348.111	0.28738	38.187
10004	10004	800	126.446	0.791483	105.175
10005	10005	6483	347.759	0.287843	38.2498
10010	10010	801	89.4141	1.12063	148.922
10011	10011	6487	347.376	0.288507	38.3405
10032	10032	805	86.933	1.15768	153.883
10030	10030	804	117.518	0.856045	113.786
10040	10040	6505	348.208	0.289487	38.4828
10050	10050	808	134.885	0.748807	99.5532
9984	9984	797	57.5096	1.73328	230.273
9500	9500	722	56.8424	1.58772	209.797
16	16	0	0.064436	0.003973	0.158917
30	30	0	0.034353	0.026199	1.28553
32	32	0	0.064927	0.015772	0.788579
64	64	0	0.077693	0.052721	3.16323
100	100	0	0.034118	0.293097	19.4729
128	128	0	0.078029	0.209974	14.6982
100	128	0	0.054709	0.233964	15.9608
128	100	0	0.059435	0.215363	14.6919
256	256	0	0.077501	0.845617	67.6494
1000	1024	8	0.316166	3.2388	323.326
1024	1000	8	0.366433	2.79451	278.973
1024	1024	8	0.303663	3.45309	345.309
8192	16384	1073	47.2133	2.84279	383.777
16384	10000	1310	66.9526	2.4471	333.879
10000	16384	1310	68.314	2.39834	327.226
16384	16384	2147	100.062	2.6827	375.578

Small Image Sizes

Performance for very small images is quite poor since the computation does not provide enough parallelism to occupy the whole GPU. If more than one image is to be transformed, FFT operations can be batched, providing additional parallelism.

The interface for building a plan to batch FFTs is quite flexible, allowing the user to specify the space between subsequent elements within an image as well as between subsequent images. Here is the function specification for cufftPlanMany

```
cufftResult
    cufftPlanMany(cufftHandle *plan, int rank, int *n, int *inembed,
        int istride, int idist, int *onembed, int ostride,
        int odist, cufftType type, int batch);
```

inembed and onembed are arrays specifying the storage dimensions of the input and output images, respectively. These can be different from rank if the images are padded. idist and odist are the distance, in elements between subsequent elements in an image. This can be very useful in performing an FFT on an image in a dimension other than the fastest-changing dimension (i.e. a 2D transform in y and z on a 3D field). istride and ostride specify the distance between the first elements of subsequent images in a batch. The type argument is the same as in cufftPlan2d(). Batch specifies how many images in the batch. So, to transform 10 2-dimensional images each of size 34x32 and stored contiguously in memory, call

```
int asize[2] = {34,32};

cufftPlanMany(plan, 2, asize, asize, 1, 32*34, asize, 1, 32*34, CUFFT_Z2Z,
10);
```

For contiguous, unpadded images, it is actually not necessary to specify most parameters. The following would also work.

```
cufftPlanMany(plan, 2, assize, NULL, 0, 0, NULL, 0, 0, CUFFT_Z2Z, 10);
```

Execute the plan by calling cufftExec as before. An example can be found in the SKA-ScienceDataProcessor git repository in the bench_batch.cu file.

The table below shows performance numbers for batched transforms for some small images. These transforms are double precision complex-to-complex and out-of-place. Temporary storage in all these cases is less than one MB.

dx	dy	batch	storage(MB)	elapsed	Gcell/s	Gflps
32	32	1	0	0.035482	0.02886	1.443
32	32	5	0	0.015265	0.06708	3.35402
32	32	10	0	0.008705	0.11763	5.88149
32	32	15	0	0.005934	0.172575	8.62875
32	32	16	0	0.004842	0.211465	10.5733
32	32	20	0	0.003745	0.273411	13.6705
32	32	32	0	0.002422	0.422721	21.1361
32	32	32	0	0.002225	0.460142	23.0071

32	30	32	0	0.001896	0.506329	25.0807
30	32	32	0	0.001688	0.56872	28.1713
32	32	32	0	0.002425	0.422233	21.1117
32	32	1024	0	0.000304	3.36531	168.265
32	30	1024	0	0.000312	3.07212	152.176
30	32	1024	0	0.000277	3.47072	171.92
30	30	1024	0	0.000288	3.12738	153.457
100	100	1	0	0.032269	0.309897	20.5891
100	100	5	0	0.007809	1.28053	85.0764
100	100	10	0	0.006133	1.63049	108.328
100	100	15	0	0.005166	1.9357	128.605
100	100	16	0	0.005007	1.99712	132.686
100	100	20	0	0.004679	2.13719	141.992
100	100	32	0	0.003864	2.58813	171.951
100	100	10	0	0.006106	1.63784	108.816
100	100	1024	0	0.002643	3.78409	251.409

Batching of small transforms is quite important, allowing us to expose more parallelism to the GPU and achieve performance near that for large images.

Power consumption

<<<THESE MEASUREMENTS ARE SUSPECT>>>>

During the transform, the K40m consumes 156 Watts at peak and with boost clocks. This was found by monitoring nvidia-smi during repeated 10000x10000 double complex transforms for 20 minutes. The temperature and power consumption stabilized after 3 minutes. The clock speed could continue at the boosted value without clamping to reduce temperature.

Future architectures

The profile of a GPU FFT is complicated. The DRAM channel is reasonably well utilized, but latency, especially for texture load instructions, keeps the memory bus from saturating. Texture caches are not utilized at all since data are read through texture only once. The new Maxwell architecture provides improvement in both DRAM bandwidth and computational throughput. Since the gain in computational throughput (50%) is larger than the gain in DRAM bandwidth (10%), we expect the latency issues will be alleviated, giving us an overall speedup between 10 and 50%.

There is no offering of Maxwell in the Tesla product line geared toward high performance, scientific computing. We can make a useful comparison with a Maxwell chip from the Quadro line, aimed at professional graphics, but only for single precision. The table below gives performance numbers for a Quadro M6000, built with a GM200 chip. As seen in the comparison of this table and the single precision complex-to-complex table for Kepler, the GM200 has an advantage over the GK110 of between 40 and 50%.

<<<<THESE DATA ARE INACCURATE!!!!>>>>

N	M	storage(MB)	elapsed	Gcell/s	Gflps
8192	8192	536	14.0483	4.77702	124.203
4096	4096	134	3.31182	5.06586	121.581
16384	16384	2147	79.6541	3.37001	94.3604
10000	10000	800	39.4197	2.5368	67.4166
10004	10004	800	241.982	0.413585	10.9917
10005	10005	800	108.981	0.918505	24.411
10010	10010	801	100.331	0.998698	26.5437
10011	10011	801	316.042	0.31711	8.42833
10032	10032	805	101.362	0.99289	26.3956
10030	10030	804	171.061	0.588098	15.634
10050	10050	808	254.393	0.397033	10.557
9984	9984	797	61.8326	1.6121	42.8348
9500	9500	722	68.5888	1.31581	34.7736
16384	10000	1310	59.9263	2.73402	74.6053
10000	16384	1310	50.7785	3.22656	88.0455
8192	8192	536	14.0476	4.77723	124.208
8192	16384	1073	35.1185	3.82185	103.19
16	16	0	0.024992	0.010243	0.081946
30	30	0	0.023757	0.037884	0.371784
32	32	0	0.029082	0.035211	0.352113
64	64	0	0.029894	0.137016	1.64419
100	100	0	0.029293	0.341381	4.53617
128	128	0	0.031136	0.526208	7.36691
100	128	0	0.024774	0.516662	7.04927
128	100	0	0.029082	0.440141	6.00522
256	256	0	0.032557	2.01297	32.2076
1000	1024	8	0.297786	3.43872	68.6567
1024	1000	8	0.36352	2.8169	56.2416
1024	1024	8	0.263789	3.97506	79.5012

Looking ahead to Pascal, NVIDIA expects to increase memory bandwidth by 3X and instruction throughput by 4X over Kepler, further reducing the influence of memory fetch latency. For this reason, we expect FFT to saturate the channel from DRAM and scale with the bandwidth through that channel. That is, we expect a speedup over Kepler of at least 3X.

Conclusions

The cuFFT library gives best performance for image dimensions that are powers of 2. A dimension of 10000 also appears to be a relative bright spot. Very small images (i.e. less than 2k) must be batched to achieve good utilization of the GPU. We expect the Pascal architecture to boost performance by about 3X.