SDP Memo 090: Comparison of convolution methods for GPUs

Document number……SDP Memo 090
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.

Comparison of convolution methods for GPUs

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

---

1. Convolution in Radio Astronomy

Several steps in the processing of radio telescope images involve convolution, mathematically expressed as

$$C(r) = \int A(r')B(r - r') \tag{1}$$

where A, B and C are 2-dimensional images and r and r' are 2-dimensional vectors. This operation is useful in building 2-dimensional functions (kernels) which describe the distortion of a celestial image observed by an instrument with limited spatial frequency.

2. Computing convolution

Practically, convolution can be computed in a number of different ways. A direct convolution could be coded as follows:

```
for (int x=xmin; x<xmax; x++)
for (int y=ymin; y<ymax; y++)
{
   C[x][y] = 0.0;
   for (int xp=-xwid; xp<xwid; xp++)
   for (int yp=-ywid; yp<ywid; yp++)
   {
      C[x][y] += A[xp][yp]*B[x-xp][y-yp];
   }
}
```

The code above follows directly from the mathematical definition, but does not check boundary conditions. It also makes no effort to optimize memory accesses and is not very efficient.

Convolution can also be computed using a Fourier transform. This is a well-known technique in signal processing motivated by noting that the Fourier transform of the convolved image is the product of the Fourier transform of the two input images. If $\otimes$ indicates convolution and $\mathcal{F}$ represents Fourier transform, we can express this as

$$\mathcal{F}(A \otimes B) = \mathcal{F}(A)\mathcal{F}(B) \tag{2}$$

Here, the Fourier transforms of A and B are multiplied point-by-point.

Whether this is more efficient than the direct convolution described above depends on a number of attributes of the images and of the compute hardware. At a high level, Fourier transform scales as $\mathcal{O}(n^2 \ln n)$, where n is the size of a (square) image. Two Fourier transforms, one inverse transform and a direct point-by-point multiply would scale as

$$\mathcal{O}(n^2 \ln n) + \mathcal{O}(m^2 \ln m) + \mathcal{O}((m+n)^2(m+n)) + \mathcal{O}((m+n)^2) \tag{3}$$

Meanwhile, direct convolution scales as $\mathcal{O}(n^2 m^2)$, with m and n the sizes of the two input images. For large images, the Fourier transform method is the obvious winner since the fourth order scaling of the direct convolution causes compute times to quickly increase. But, for smaller image and kernel sizes, we expect to find regimes in which the direct convolution method is best. Further, when we compare the favorable, highly local data access pattern of the direct convolution to the strided access of the Fourier transform, the direct convolution looks even more attractive.

The goal of this project is to examine and quantify the performance of these two approaches for present GPUs for a range of image sizes and to predict, as best we can, performance for future GPUs.

2.1. cuDNN and cuFFT

NVIDIA produces and maintains a highly-optimized library for computing a Fast Fourier Transform (FFT) of an image. This library is called cuFFT. It is optimized for each generation of GPU, leveraging very specific knowledge of NVIDIA hardware. We will use cuFFT for the Fourier transforms in the work described here. NVIDIA strongly encourages the use of

libraries, where possible. NVIDIA specialists tune library kernels for performance on each new architecture. Libraries not only give better performance today, but also ensure good performance in the future.

Convolution of a kernel with a large image is a common operation in image recognition with deep neural networks. This has been an area of emphasis for NVIDIA and other companies and organizations. NVIDIA has produced a library, called cuDNN, to facilitate deep learning on GPUs. Since image recognition most often uses very small kernels, the algorithms in the cuDNN library includes highly-tuned functions for computing convolutions.

The cuDNN library includes several convolution algorithms. In this paper, we will be focused on a method called "implicit GEMM". In this method, the image and kernel are transformed into specialized 2-D matrices which are then multiplied. In development of cuDNN, this method was found to perform better than an optimized direct convolution written in CUDA.

cuDNN also includes a Winograd transform method and an FFT-based method. The current iteration of cuDNN has important limitations for some of these methods. Winograd transform, for example, is only available for 3x3 kernels. As cuDNN expands its capabilities, these limitations may be relaxed and enable wider use in SKA. For the purpose of this paper, we have used implicit GEMM exclusively in cuDNN.

At the time of this report, cuDNN has no explicit support for complex numbers. We accomplished the complex multiplication by treating the real and complex parts of each image field as a "color". We separate the real and complex parts in memory and keep two copies of each to form a 4D tensor as follows

$$\begin{bmatrix} [A_r] & [A_i] \\ -[A_i] & [A_r] \end{bmatrix}$$

In the above expression, $[A_r]$ represents the real part of the entire 2-D image called A. The other image is also separated, but stored only once as follows

$$\begin{bmatrix} [B_r] \\ [B_i] \end{bmatrix}$$

The product of these two under the cuDNNForwardConvolution is then

$$\begin{bmatrix} [A_r] \otimes [B_r] - [A_i] \otimes [B_i] \\ [A_r] \otimes [B_i] + [A_i] \otimes [B_r] \end{bmatrix}$$

where $\otimes$ again represents convolution. This 3-D tensor (with an outer dimension of 2) is the real and imaginary parts of the complex matrix multiplication.

3. Batching

For the small image sizes of interest to SKA, we may struggle to find sufficient parallelism to achieve maximum performance on the GPU. To mitigate this, we batch several convolutions. For direct convolution, the computation should scale as the product of the two batch sizes. If we convolve 10 w-kernels with 10 A-kernels, we expect computation to increase by 100X. For FFT-based convolutions, the number of inverse transforms scales as the product of the batch sizes (the same 100X as before), but the forward transforms do not. For 10 w- and 10 A-kernels, the forward transforms are just 10X more computation.

cuDNN supports parallel convolution of many images with many kernels. There is some additional complication since our complex convolution of a single image *already* involves a set of 2 real images, each with 2 colors. Because of this, we need a matrix transpose after the convolution to collect the real and imaginary parts of each image.

4. Comparisons of methods

Figure 1 shows the performance of the cuFFT method and the cuDNN method for the convolution of an image of fixed size 512×512, varying the size of the other image. The performance is computed by assuming the same number of flops as required in the naive implementation in Sec. 2. For two images of size $\mathtt{n} \times \mathtt{n}$ and $\mathtt{m} \times \mathtt{m}$, the flops are $8n^2m^2$. The measured time includes all data manipulation, but excludes the data transfer to and from the GPU and also excludes the plan creation for both cuFFT and cuDNN. All experiments were done on a K40c with boost clocks and with ECC disabled.

As expected, for large kernel sizes, the FFT-based algorithm is superior. Figure 2 is the same data, but at a finer scale focused on the small kernel end of the scale. For some very small kernel sizes, cuDNN out performs cuFFT. The sporadic nature of the cuFFT performance makes this difficult to predict. As mentioned, cuFFT may use different algorithms even for image sizes which are quite close to one another. Our problem is further complicated by the fact that we may have FFT operations on images of three different sizes.

Each of the input images is transformed, then the output image, which is one pixel smaller than the combined width of the two input images, is inverse transformed to obtain the final convolved image. For best performance, it is worth paying attention to cuFFT performance for different image sizes since often very small changes in size can make large differences in performance.

These unpredictable changes in performance with small changes in image size make it dangerous to interpolate between data points. The spline fit on the graphs should not be regarded as predictive.

In figure 3, we plot the same data, but add the data for different fixed image sizes. The size of one image is, again fixed, but at the value specified in the legend next to the image.

All cases are very similar with performance for both methods very similar for very small kernel sizes and cuFFT performing much better for large kernels. For the largest images, we can find kernel sizes for which cuDNN outperforms an FFT-based convolution, but even for medium-sized images ($<$ 200 pixels wide), FFT-based convolution always gives better performance. Even for the large images, the regions in which cuDNN performs can be avoided by careful selection of kernel sizes.

Figure 4 shows performance for an all-to-all convolution of 512 kernels with one another as a function of the kernel size. This task is quite different from the task above. When both kernels are very small, it is easier to stay below the regime in which the $\mathcal{O}(n^2m^2)$ scaling for the direct convolution dominates.

5. Performance limiters and scaling

5.1. cuFFT

In general, performance of cuFFT varies considerably and rapidly with image size. Best performance is achieved for image sizes which are powers of 2. For sizes outside that narrow category, performance is best when the dimension is a multiple of 2, 3, 5 or 7, and preferably two from that list. The memory requirements of FFT also vary by the algorithm used. An image size not divisible by 2, 3, 5 or 7 will allocate much more temporary memory, which can cause problems. We need to pay attention not only to the sizes of the input kernels and images, but also to the size of the output since that image, too, will be Fourier transformed. The output image dimension is the sum of the two input image dimensions minus 1.

$$d_{out} = d_{in,1} + d_{in,2} - 1 \tag{4}$$

We examined three test cases in depth. For the first case, we convolved 10 512×512 images with 40 50×50 kernels. This is a wonderful case for the FFT-based solution. The floating point instruction pipes are 72% utilized and 75% of warp stalls are due to too many issueable instructions waiting for a full instruction pipe.

For the second case, we reduced the size of the images in the second batch to 5×5. This is a case for which direct convolution outperforms FFT-based convolution. The cuFFT kernels are exposed to the latency of memory instructions. The floating points units are utilized at just 57% and almost half of stalled warps are due to an unresolved memory dependency. The hit rate in L2 is a respectable 62% due to the frequent reuse of the small kernel. As a result, the achieved bandwidth from L1 is a higher fraction of peak (28%) than from L2 or from DRAM. Finally, the occupancy is 40% and limited by the size of the register file.

The last case we examined is the all-to-all convolution of 197 11×11 kernels. For the FFT-based convolution, the small FFTs have insufficient parallelism to cover the memory latency. The occupancy is capped at 75% due to register use, but achieved occupancy is just 58%.

5.2. cuDNN

The performance analysis of the cuDNN kernels looks very similar for each of these three cases. For the first case (one batch of 10 512×512 images and one batch of 40 50×50 images), total instruction throughput is 75% of peak, but only 80% of those are arithmetic instructions. More than two-thirds of the remaining operation are memory operations.

The other two cases have a similar issue. For the second case, instruction throughput falls slightly to 70% and falls to 66% for the third case. In all of these cases, the memory instructions are primarily shared memory operations. Global loads are sent through the texture path and the hit rates in texture are 77%, 91% and 65%. So, the large number of memory operations is a sign that memory is being effectively managed. The real problem with the cuDNN implementation is the $\mathcal{O}(n^2m^2)$ scaling.

6. Conclusion

There turn out to be very few sets of parameters for which the implicit GEMM algorithm, implemented in cuDNN, outperforms an FFT-based convolution. However, for images larger than 200×200 and kernels smaller

than 15×15, the cuDNN convolution library can beat an FFT-based convolution. The cuDNN routines perform convolution very efficiently, but the favorable scaling of the FFT-based convolution makes it the better choice for all most cases. The performance of cuDNN libraries is more predictable than the cuFFT libraries, which can show large changes in performance for small changes in image size. This is due to the different algorithms used inside cuFFT which depend on specific factorizations of the image width.

Future versions of cuDNN may make additional convolution algorithms available. It is our opinion that the cuDNN library should be revisited at a later date.

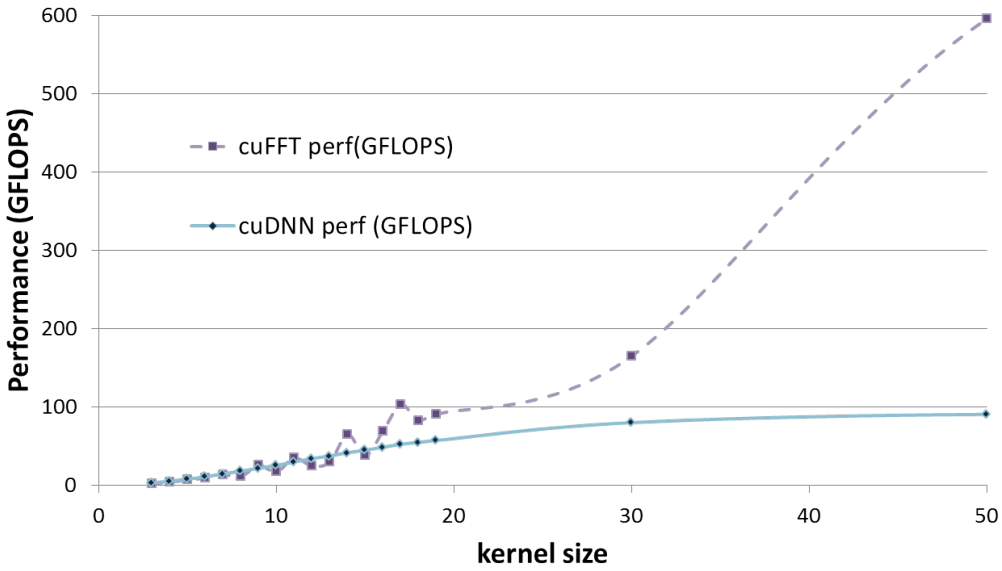

Figure 1: Performance of convolution with a 512×512 image as a function of the size of the second image.

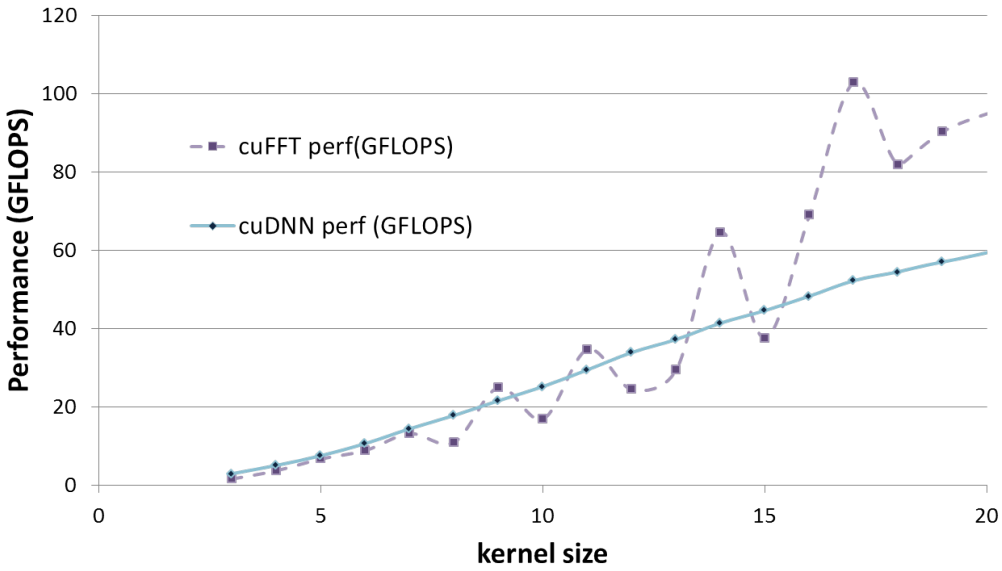

Figure 2: Performance of convolution with a 512×512 image as a function of the size of the second image

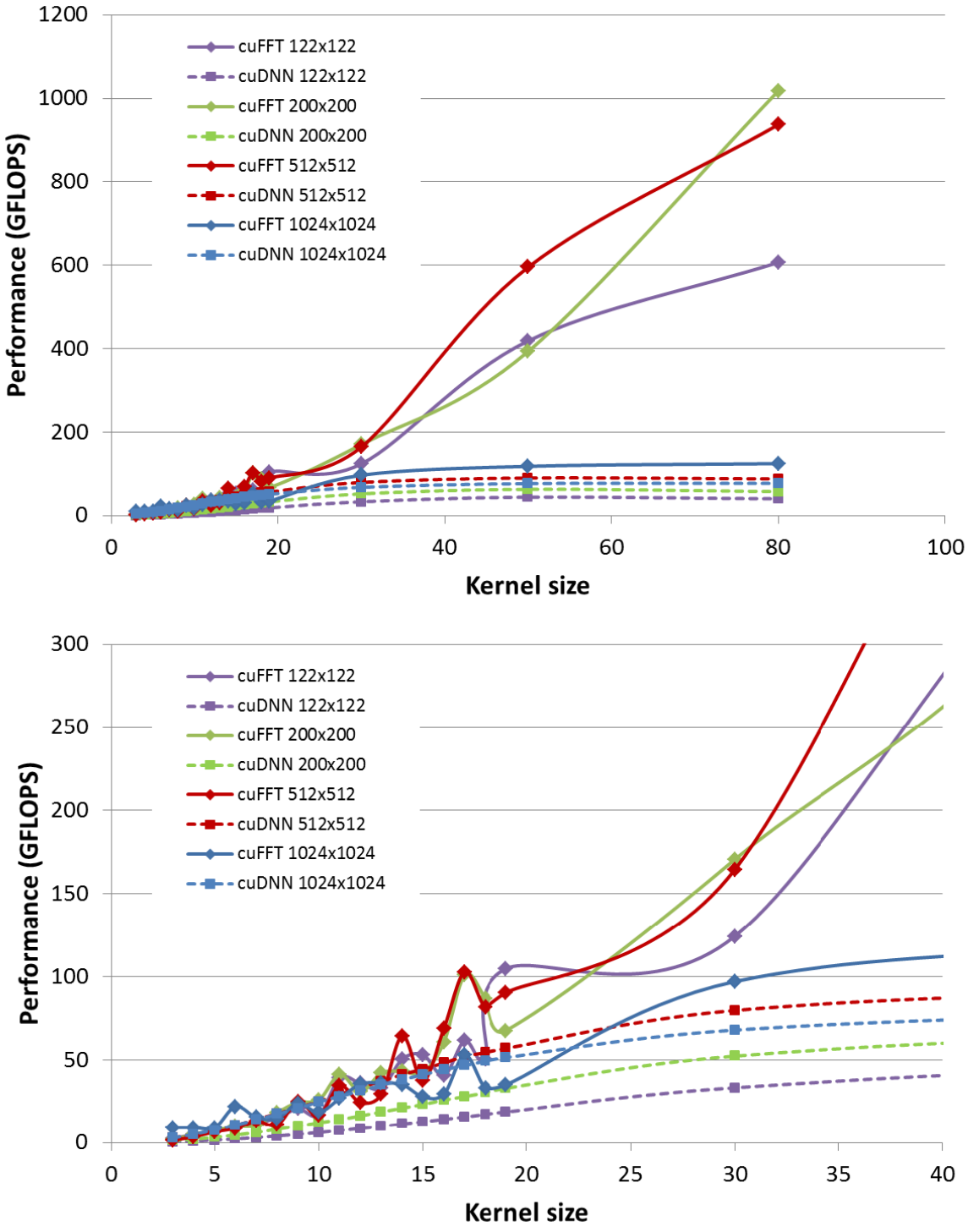

Figure 3: Performance of convolution as a function of the size of the second image for multiple image sizes

Figure 4: Performace of all-to-all convolution as a function of kernel size