Fast Convolutions Via the Overlap-and-Save Method Using Shared Memory FFT

Karel Adámek, Sofia Dimoudi, Mike Giles, Wes Armour

Content

1. Convolutions and motivation
2. Overlap-and-save method
3. Custom shared memory FFT
4. Results
5. Conclusions

DEPARTMENT OF ENGINEERING SCIENCE

UNIVERSITY OF OXFORD

OXFORD e-Research CENTRE

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Convolution (time-domain)

Convolution is one of the fundamental signal filtering techniques widely used in natural sciences and signal processing.

Convolution is given by

$$\mathrm{y[n]} = \mathrm{h[k]} \star \mathrm{s[n]} = \sum_{k=0}^{M-1} s[n-k]h[k],$$

s the input signal of size N, h is the filter of length M, and y is the convolved signal N-M+1,

- Complexity is NM
- Suited for very small filters

Convolution (frequency-domain)

We could also invoke convolution theorem and perform convolution using frequency-domain

$$h[k] \star s[n] = FT^{-1}(H[m] \cdot S[m])$$

H and S are Fourier pairs in frequency domain of h and s which are in time domain.

In frequency domain the convolution is just a point-wise complex multiplication.

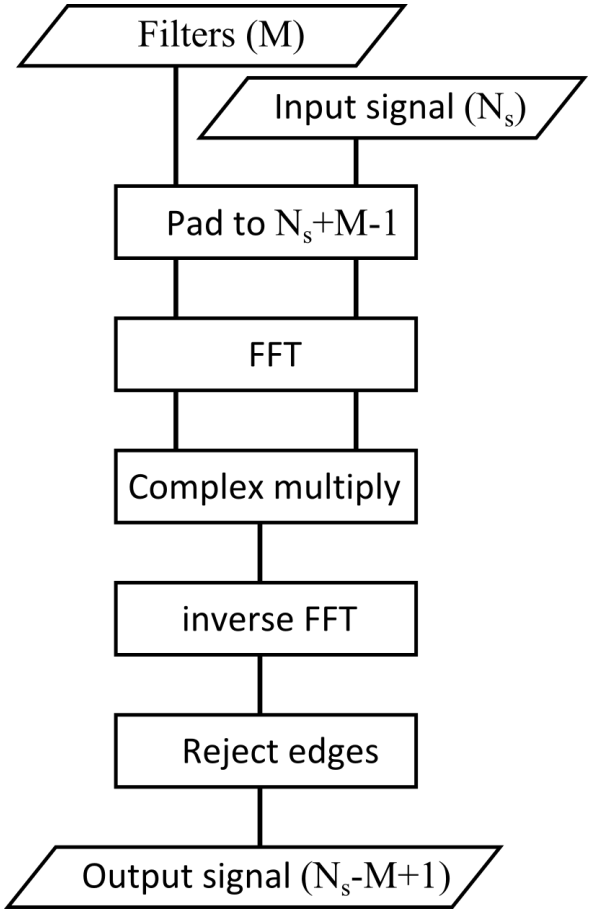
Complexity of convolution through frequency domain is $3N \log_2 N + 2N$

How to do convolution in frequency-domain

Doing convolution via frequency domain means we are performing circular instead of a linear convolution.

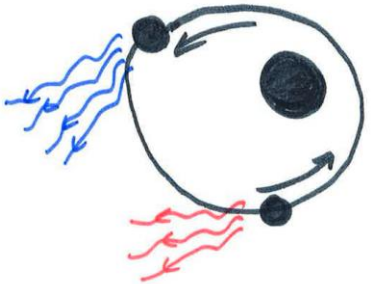
Frequency domain convolution:

- Signal and filter needs to be padded to N+M-1 to prevent aliasing
- It is suited for convolutions with long filters
- Less efficient when convolving long input signal with a short filter, because due to padding of the filter we processing a lot of “zeroes”.



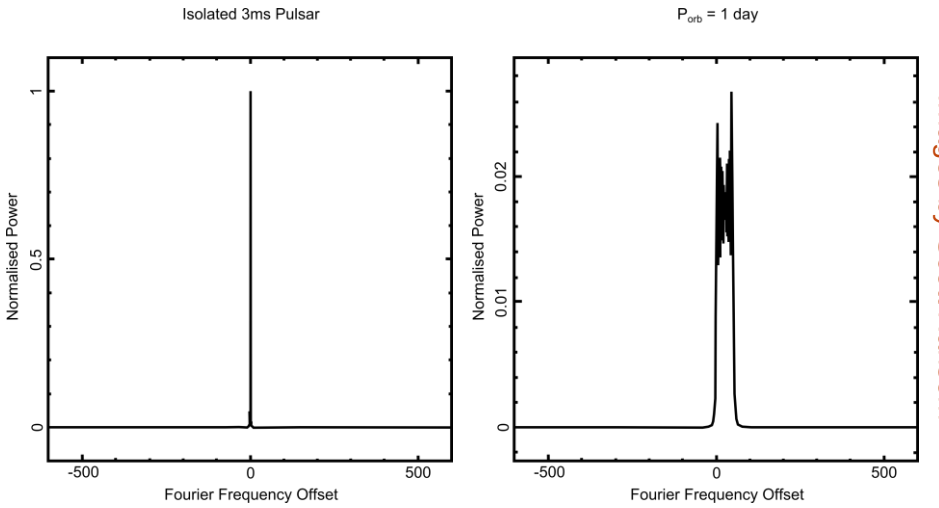
Our motivation

Motivation – Fourier Domain Acceleration Search



Normal pulsar

$P>10T_{obs}$



Images by Scott Ransom

Signals from binary systems can undergo a Doppler shift due to accelerated motion experienced over the orbital period.

- signal is no longer periodic
- standard pulsar searches are less sensitive

This can be corrected by using a matched filter approach.

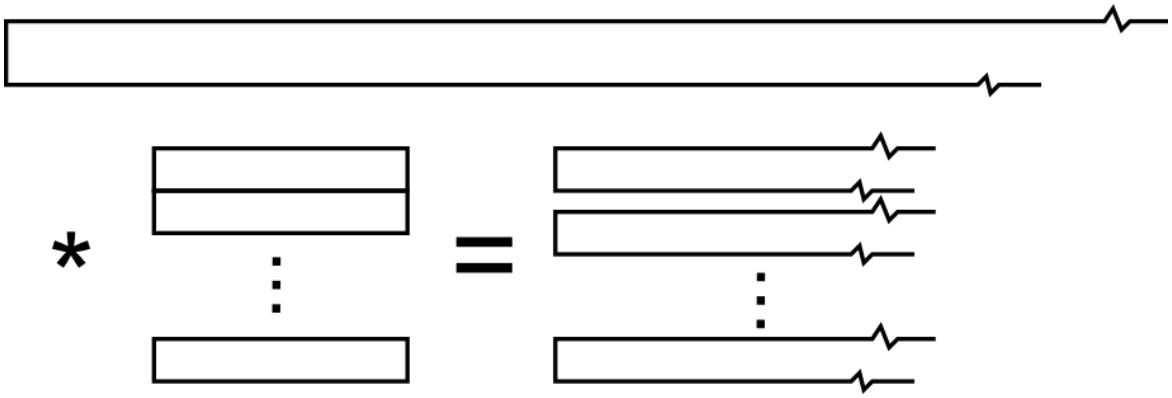
Motivation – Fourier Domain Acceleration Search

Fourier domain accelerated search[1,2] (FDAS) uses multiple matched filters, where each filter fits a specific acceleration.

- Number of filters F depends on FDAS precision (SKA: 1-200)
- Size of the filters M depends on maximum acceleration searched (SKA: ~200)
- Size of the signal depends on observation time (SKA 8M+ samples)

Also we would like to do interbining of the output.

What is the best technique?



[1] Dimoudi Sofia et. al. A GPU implementation of the Correlation Technique for Real-time Fourier Domain Pulsar Acceleration Searches, 2018
[2] Ransom Scott et. al. A New Search Technique for Short Orbital Period Binary Pulsars 2003

UNIVERSITY OF OXFORD
Oxford e-Research Centre

Our approach is general

Our convolution presented here is for general case.

So If you have

- long input signal

- and a set of short (<2048) filters

- and require non-local operations on convolution result (like interbinning in FDAS), but even without it.

Then our approach could be useful to you…

Overlap-and-Save & Overlap-and-Add

Overlap-and-save(add) method is a hybrid method which combines advantages of time-domain convolution and frequency domain convolution.

It allows us to separate input signal into segments which are convolved separately using frequency domain convolution.

Overlap-and-save method:

- Especially suited for long input signals and short filters
- No need for long paddings of filters
- No synchronization needed for Overlap-and-save method. Overlap-and-add needs to know about its neighbors.
- GPU friendly

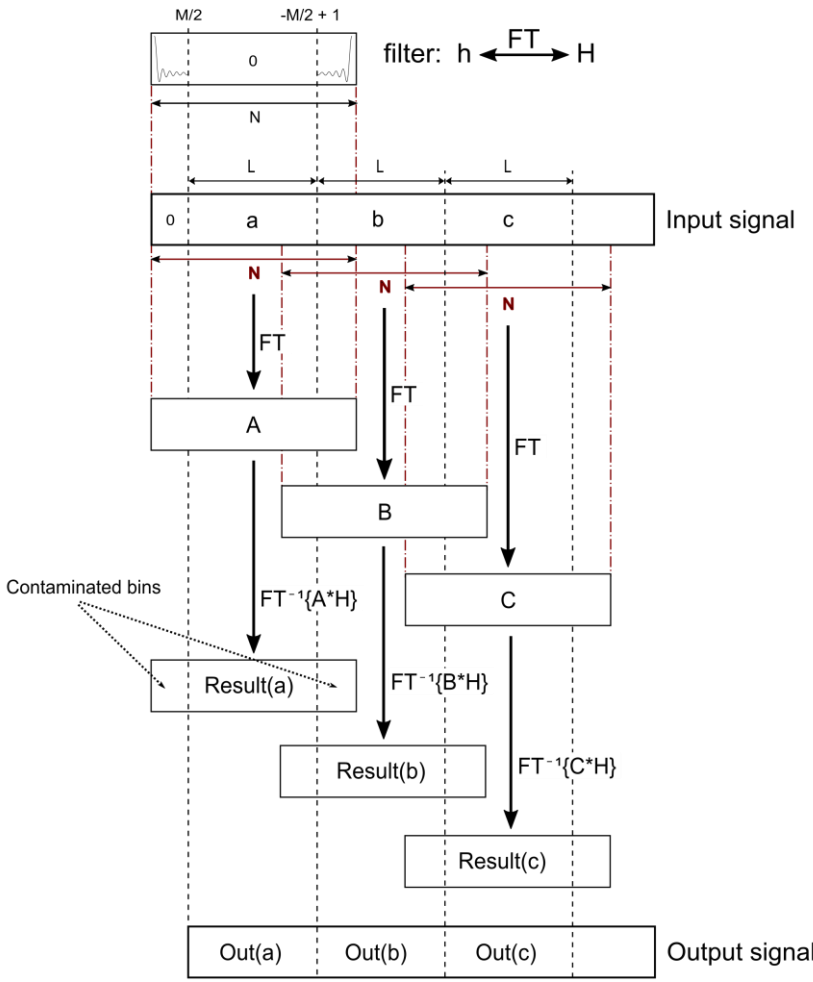
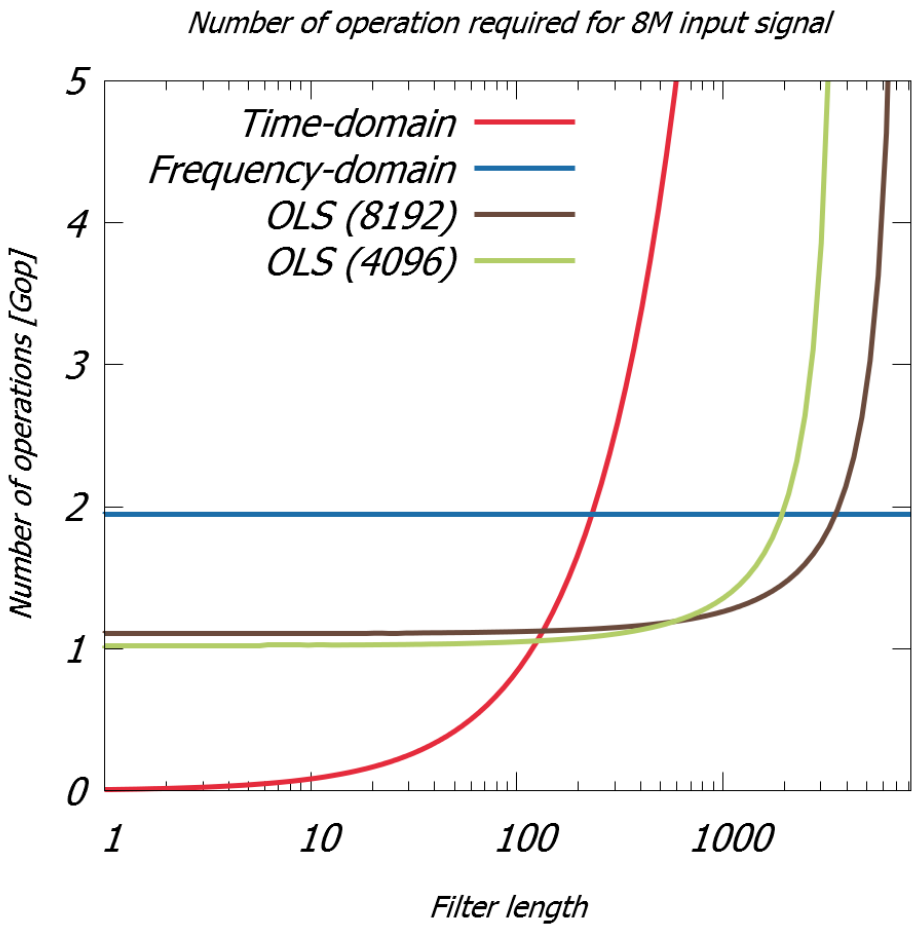


Image by Sofia Dimoudi

Number of operations

- Time-domain convolution is most efficient for tiny filter sizes
- Frequency-domain convolution is best when filter is long
- Overlap-and-save is hybrid method suited for short filters

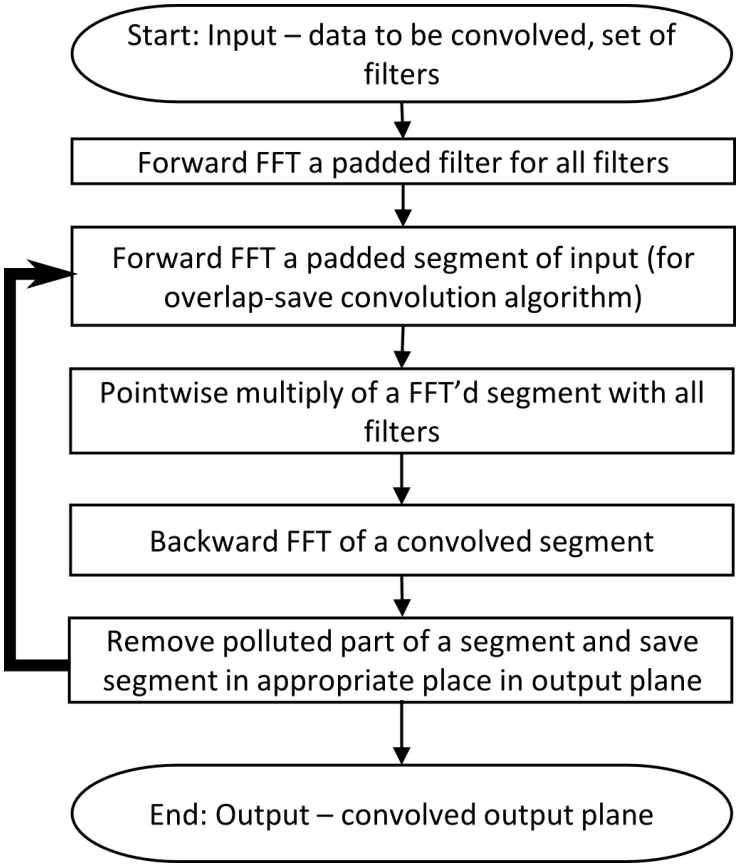
Number of operations is only one of many parameters affecting performance.



Implementation of OLS using cuFFT

RIGHT: Flow diagram of the OLS method.

- Forward FFT and inverse FFT is calculated using cuFFT library
- Best performing FFT length for cuFFT is 8192 samples.
- Custom GPU kernels are needed for point-wise multiplication and removing aliased parts
- Each segment is convolved with same set of filters, these are reused



Point-wise complex multiplication kernel

Parallelization of point-wise multiplication of a segment with set of filters

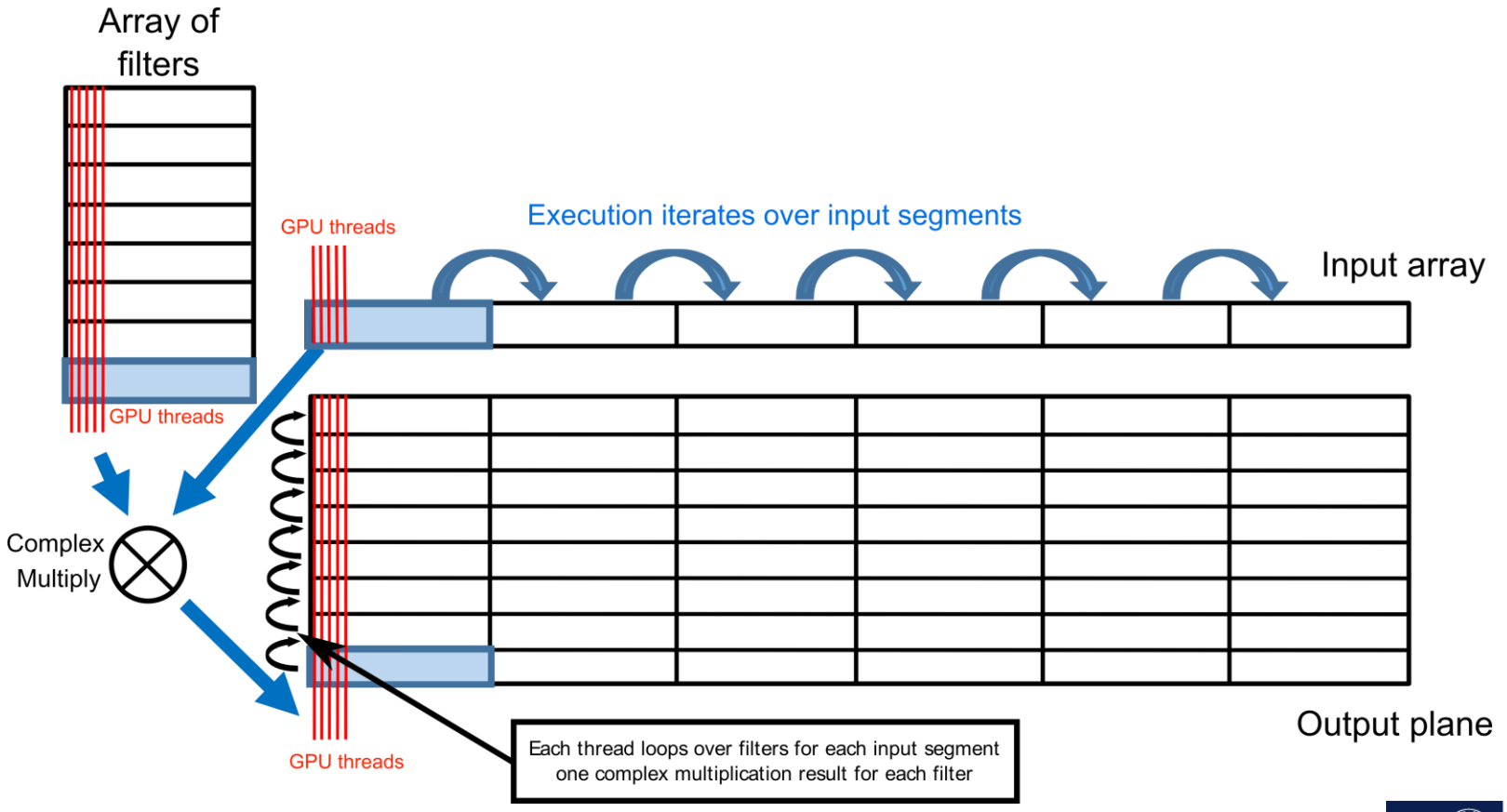
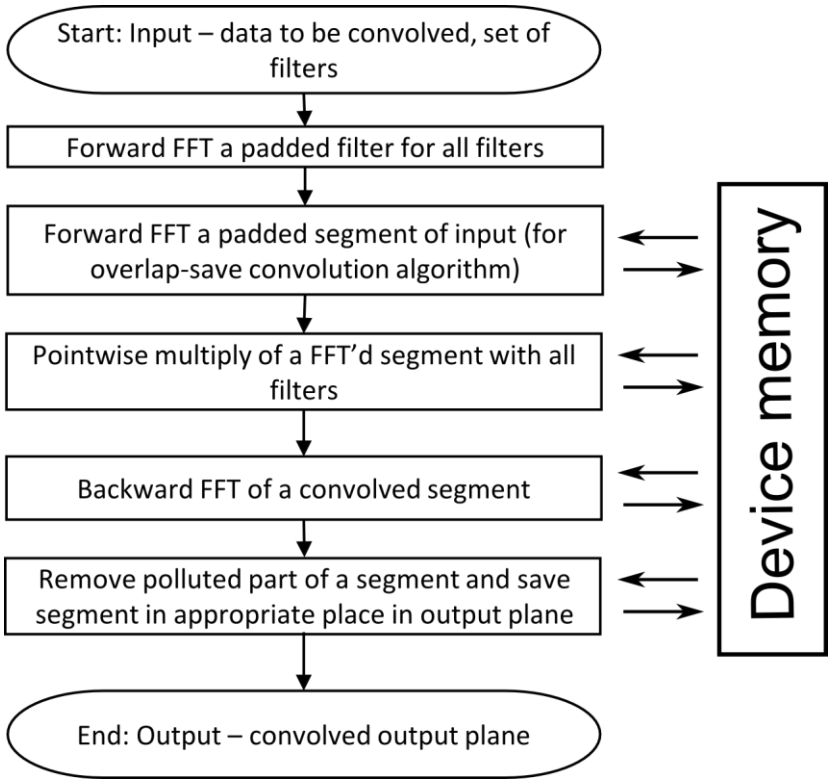


Image by Sofia Dimoudi

Can we do better?

What is the limiting factor in the cuFFT implementation of Overlap-and-save?

- Accesses to the device memory



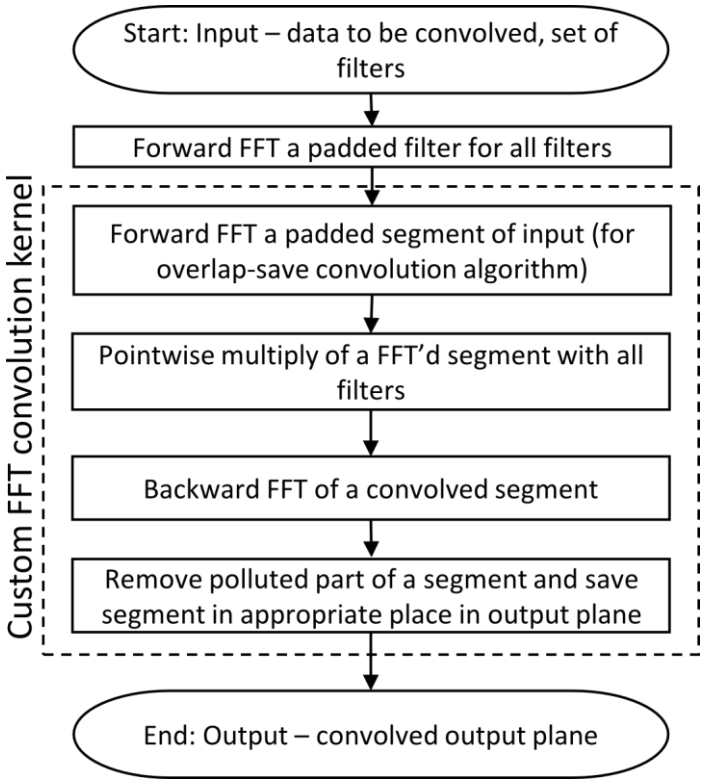
Can we do better?

What is the limiting factor in the cuFFT implementation of Overlap-and-save?

- Accesses to the device memory

We can eliminate these by having an FFT implementation invokable from the thread-block.

- This would allow us to perform all steps of the overlap-and-save method inside the thread-block



Shared Memory FFT

What FFT algorithm to choose

The custom FFT algorithm should

- be best suited to our needs; aim is to develop a convolution not general purpose FFT
- be fast but does not need to be the best
- be using shared memory
- In-place
- consume as little registers as possible so it would not impact the kernel which is calling it
- focus on FFT size $N=2^t$

There are three basic algorithms

1) Cooley-Tukey
   + Simple access pattern
   + Local to the warp for first 5 iterations
   - Needs reordering of the output

2) Pease
   + Memory access pattern does not change
   - Needs reordering of the output

3) Stockham
   + Does not need reordering of the output
   + Great for stand alone FFT code

Custom FFT

We have chosen Cooley-Tukey implementation

1) Getting rid of the reordering step
   - Convolution in frequency domain is point-wise multiplication which is order invariant we can leave FFT result in wrong order as long as we correct it during inverse FFT.
   - Using combination of DIF and DIT Cooley-Tukey algorithm will do the trick.

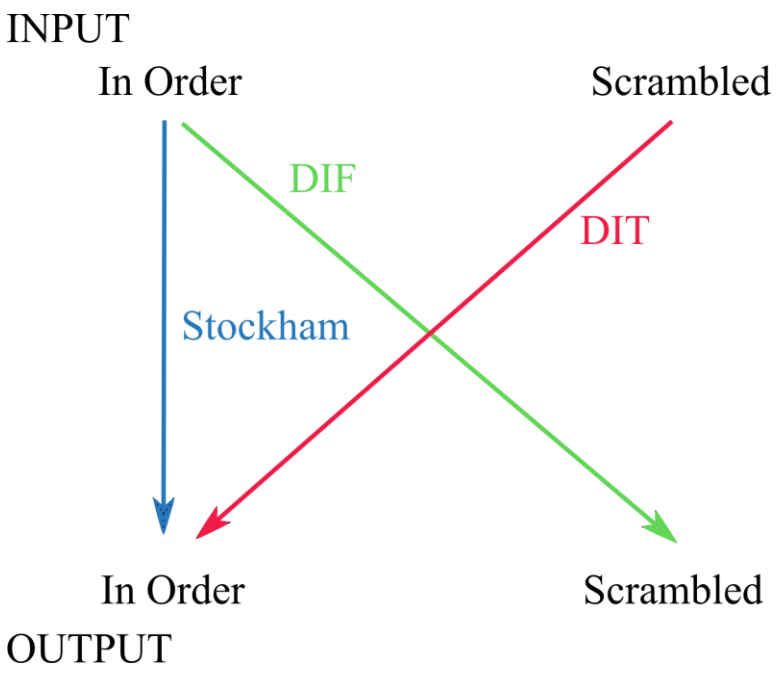
2) Simple data access pattern

3) Small butterflies
   - Butterflies smaller than warp could performed using shuffles without synchronization

4) Large butterflies
   - Performed using shared memory

Calculation of twiddle factors requires evaluating exp(), we use fastmath intrinsics for that.

Decimation in time or in frequency?



Cooley-Tukey FFT

The discrete Fourier transformation is given

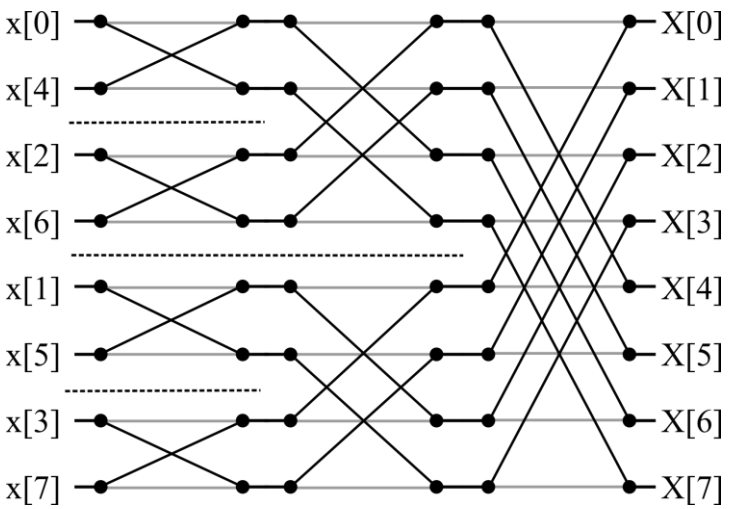
$$\mathrm{X}[n] = \sum_{k=0}^{N} x[n] e^{-\frac{i2\pi kn}{N}}$$

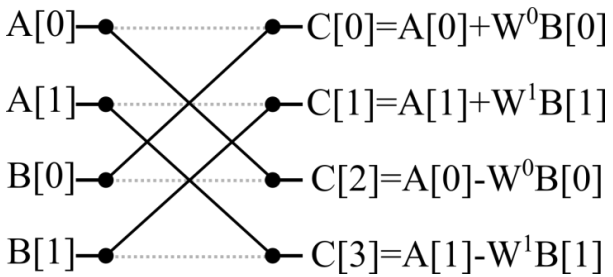
$$W^k = e^{-\frac{i2\pi kn}{N}}$$

W is called twiddle factor.

FFT algorithm is based on divide and conquer, two smaller FFTs (A, B) are combined into new bigger one C

$$\mathrm{C}[k] = A\left[k \bmod \left(\frac{N}{2}\right)\right] + W^k B\left[k \bmod \left(\frac{N}{2}\right)\right]$$





Initial implementation:

- One thread calculates two different elements of C from the same FFT which share the same input data and uses the same twiddle factor (C[0], C[2])

Cooley-Tukey FFT

The discrete Fourier transformation is given

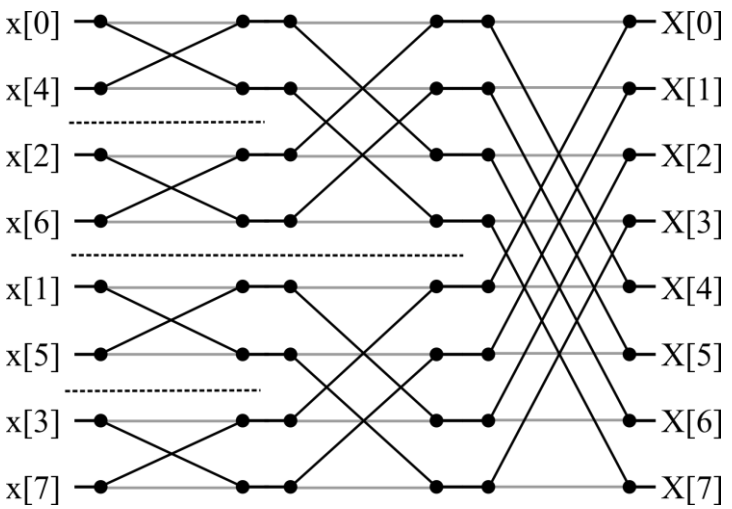
$$X[n] = \sum_{k=0}^{N} x[n] e^{-\frac{i2\pi kn}{N}}$$

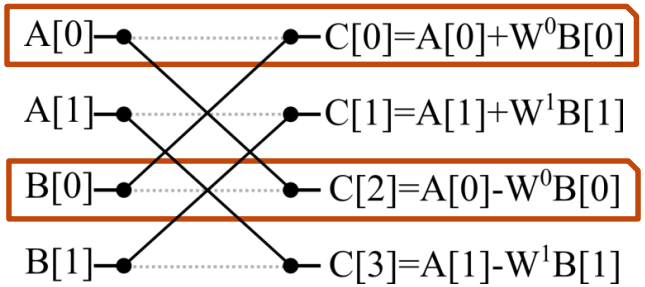
$$W^k = e^{-\frac{i2\pi kn}{N}}$$

W is called twiddle factor.

FFT algorithm is based on divide and conquer, two smaller FFTs (A, B) are combined into new bigger one C

$$C[k] = A\left[k \bmod \left(\frac{N}{2}\right)\right] + W^k B\left[k \bmod \left(\frac{N}{2}\right)\right]$$





Initial implementation:

- One thread calculates two different elements of C from the same FFT which share the same input data and uses the same twiddle factor (C[0], C[2])

Custom FFT progression

Basic:

- Limited by shared memory bandwidth
- High special function unit (SFU) utilisation
- Shared Mem. bank conflicts
- Low twiddle factor reuse
- Low instruction level parallelism

Basic:

Shared memory bandwidth: 10,248 TB/s; (73%)

Synchronization: 31.4%; pipe busy: 33.5%;

Theoretical occupancy: 100%;

Load/Store instructions: 50%; single: 50%;

Kernel	Time (ms)	Speed-up	Total Speed-up
Basic	2.22	X	X

Execution time for TitanV is for 100k FFTs each 1024 samples long. Code performs 100 FFTs per kernel to avoid being device memory bandwidth limited.

UNIVERSITY OF OXFORD

Oxford e-Research Centre

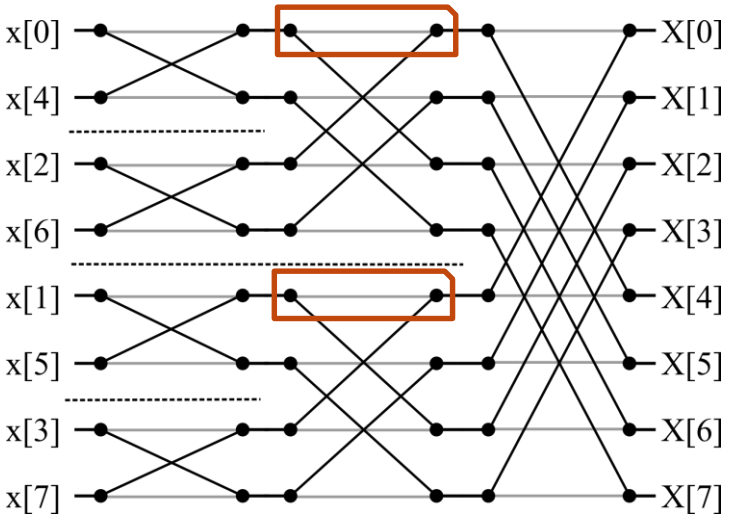
Introduction of shuffle instruction

Shared memory bank conflicts are caused by small butterflies.

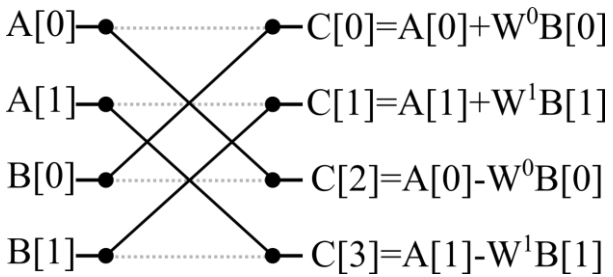
For butterflies smaller then 32 use shuffle instructions.

Different parallelization:

- One thread calculates the same element C from independent sub-FFTs (for example C[0])
- Allows us to use shuffle instructions



- No share memory bank conflicts
- No synchronization required
- Increases Load/Store instruction utilization



Introduction of shuffle instruction

Shared memory bank conflicts are caused by small butterflies.

For butterflies smaller then 32 use shuffle instructions.

Different parallelization:

- One thread calculates the same element C from independent sub-FFTs (for example C[0])
- Allows us to use shuffle instructions

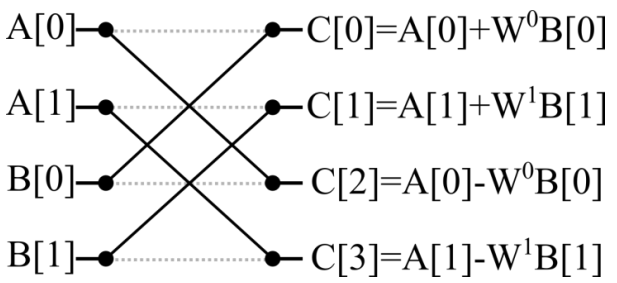
```
//--> Second through Fifth iteration (no synchronization)
PoT=2;
PoTp1=4;
for(q=1;q<5;q++){
    m_param = (local_id & (PoTp1 - 1));
    itemp=m_param>>q;
    W=Get_W_value(PoTp1,m_param); // Twiddle factors

    // Reading B
    f2temp.x=__shfl_xor(A_DFT_value.x,(1-itemp)*PoT);
    f2temp.y=__shfl_xor(A_DFT_value.y,(1-itemp)*PoT);
    // Calculation C = A + W*B
    A_DFT_value.x = __shfl_xor(A_DFT_value.x,itemp*PoT)
                  + W.x*ftemp2.x-W.y*f2temp.y;
    A_DFT_value.y = __shfl_xor(A_DFT_value.y,itemp*PoT)
                  + W.x*ftemp2.y+W.y*f2temp.x;

    ...

    PoT=PoT<<1;
    PoTp1=PoTp1<<1;
}
```

- No share memory bank conflicts
- No synchronization required
- Increases Load/Store instruction utilization



Shuffle instructions

Unroll:

First 5 iteration of the FFT algorithm are calculated using shuffle inst.

Less synchronization

No bank conflicts

- Limited by Load/Store instructions
- Medium special function unit (SFU) utilisation
- NO Shared Mem. bank conflicts
- Low twiddle factor reuse
- Low instruction level parallelism

Unroll:

Shared memory bandwidth: **5,225 TB/s**;

Synchronization: **24.9%**; pipe busy: 33%;

Theoretical occupancy: **50%**;

Load/Store instructions: **70%**; single: 50%;

Kernel	Time (ms)	Speed-up	Total Speed-up
Basic	2.22	X	X
Unroll	1.95	1.13	1.13

Execution time for TitanV is for 100k FFTs each 1024 samples long. Code performs 100 FFTs per kernel to avoid being device memory bandwidth limited.

Four elements per thread

Four elements:

Four FFT element are processed per thread.

Good:

- Better twiddle factor reuse
- More instruction level parallelism
- Less threads per thread-block

Bad:

- More registers

Status:

- Limited by Load/Store instructions
- Low SFU utilisation
- NO Shared Mem. bank conflicts
- Good twiddle factor reuse

Four elements:

Shared memory bandwidth: **6,365 TB/s**;

Synchronization: **17%**; pipe busy: 43%;

Theoretical occupancy: 50%;

Load/Store instructions: 75%; single: 50%;

Kernel	Time (ms)	Speed-up	Total Speed-up
Basic	2.22	X	X
Unroll	1.95	1.13	1.13
Four elements	1.49	1.31	1.49

Execution time for TitanV is for 100k FFTs each 1024 samples long. Code performs 100 FFTs per kernel to avoid being device memory bandwidth limited.

UNIVERSITY OF OXFORD

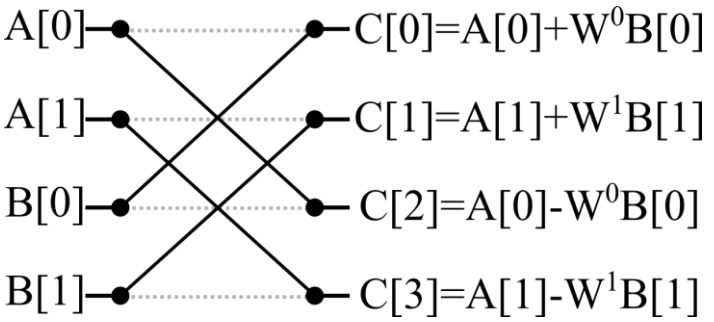
Oxford e-Research Centre

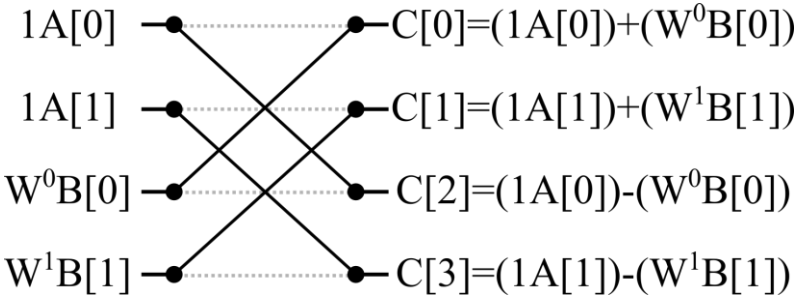
Less shuffle instruction per thread

Shuffle instructions increasing Load/Save instruction utilization which limits the code.

In previous version thread does not know which element (A or B) is local to the thread and which it needs to load. This is a problem since B needs to be multiplied by twiddle factors while A should not.

Multiplying with modified twiddle factor, which is for A W=1, before shuffle instructions means we no longer need to know which element is local to the thread, since shuffled element has proper value.





UNIVERSITY OF OXFORD

Oxford e-Research Centre

Less shuffle instructions

Final:

Less shuffle instruction, reduced number of registers

Good:

- Occupation 62.5% but number of active blocks is 2.5x larger (from 2 to 5)
- Shared Mem. B. 60% utilized

- Limited by floating point operations and Load/Store instructions
- Low SFU utilisation
- NO Shared Mem. bank conflicts
- Good twiddle factor reuse

Final:

Shared memory bandwidth: **8,374 TB/s**;

Synchronization: 21%; pipe busy: 41%;

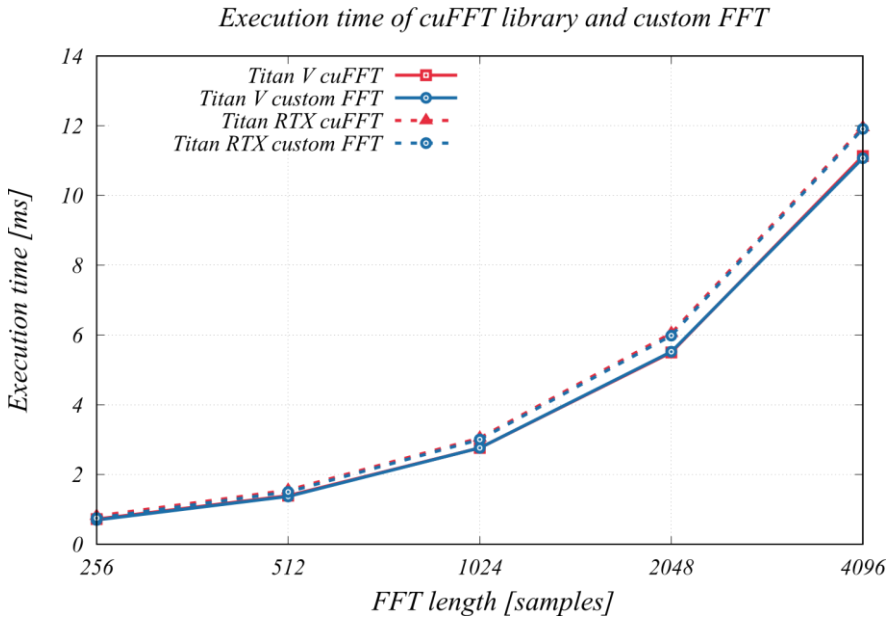
Theoretical occupancy**: 62.5%**;

Load/Store instructions: 75%; single: **75%**;

Kernel	Time (ms)	Speed-up	Total Speed-up
Basic	2.22	X	X
Unroll	1.95	1.13	1.13
Four elements	1.49	1.31	1.49
Final	1.18	1.26	1.89

Execution time for TitanV is for 100k FFTs each 1024 samples long. Code performs 100 FFTs per kernel to avoid being device memory bandwidth limited.

FFT performance – fair and unfair comparison



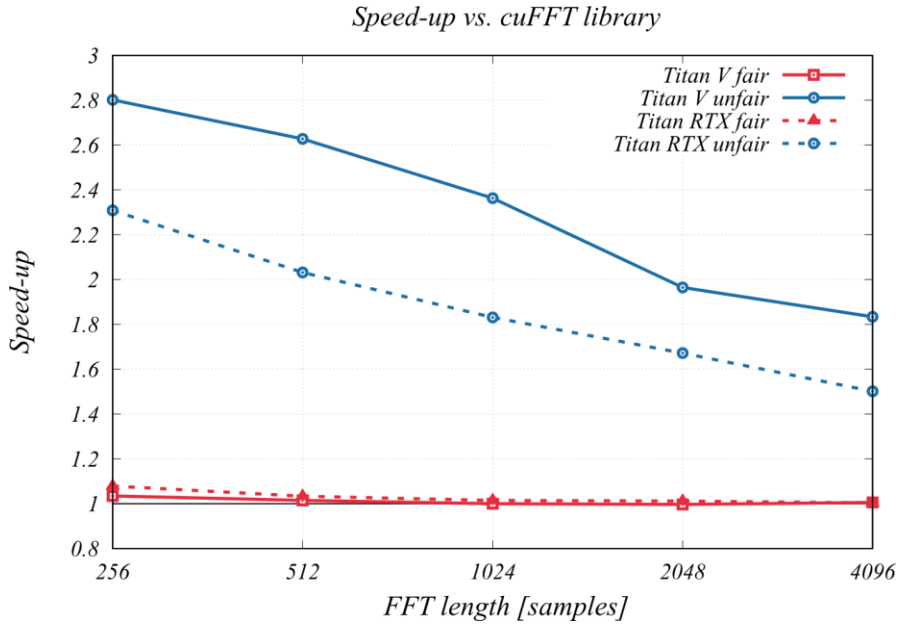
Fair comparison is when both codes are limited by device memory bandwidth.

In fair comparison custom FFT is as fast as cuFFT for tested FFT sizes.

FFT size is limited by size of the shared memory or number of threads.

Unfair comparison with custom FFT is when custom FFT is not limited by device memory bandwidth while cuFFT is.

We have calculated 100 FFTs per kernel to avoid being device memory bandwidth limited.

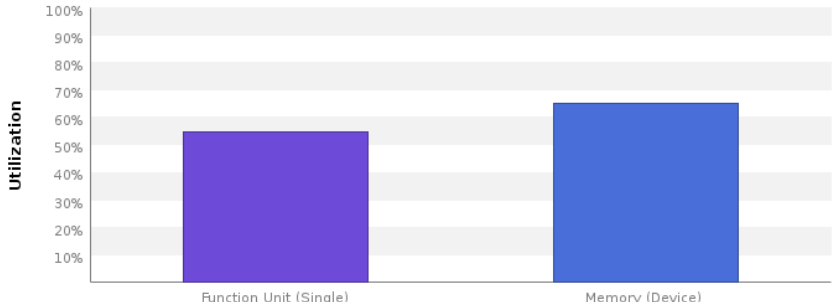


Putting convolution kernel together

Convolution kernel is using same implementation of point-wise complex multiplication as in cuFFT convolution.

For 2M points, filter M=192, convolution = 1024, F=64 filters

- FP32 instructions and Load/Store instructions are high
- Device memory bandwidth 67%
- Shared memory bandwidth 53%
- L2 hit rate is 62%
- L1 is sacrificed for shared memory
- Occupancy 50%



sed by this kernel for the various types of memory on the device. The table also shows the utilization
oughput supported by the memory. More...

Bandwidth	Utilization
3,822.827 GB/s	
3,854.209 GB/s	
7,677.037 GB/s	Idle Low Medium High Max
514.353 GB/s	
467.121 GB/s	
981.473 GB/s	Idle Low Medium High Max
0 B/s	
0 B/s	
528.966 GB/s	
467.121 GB/s	
6,784.412 GB/s	
7,780.499 GB/s	Idle Low Medium High Max
11.005 GB/s	
426.483 GB/s	
437.488 GB/s	Idle Low Medium High Max

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Results

Callbacks

Callbacks are user defined functions which enable modification of input or output of the cuFFT.

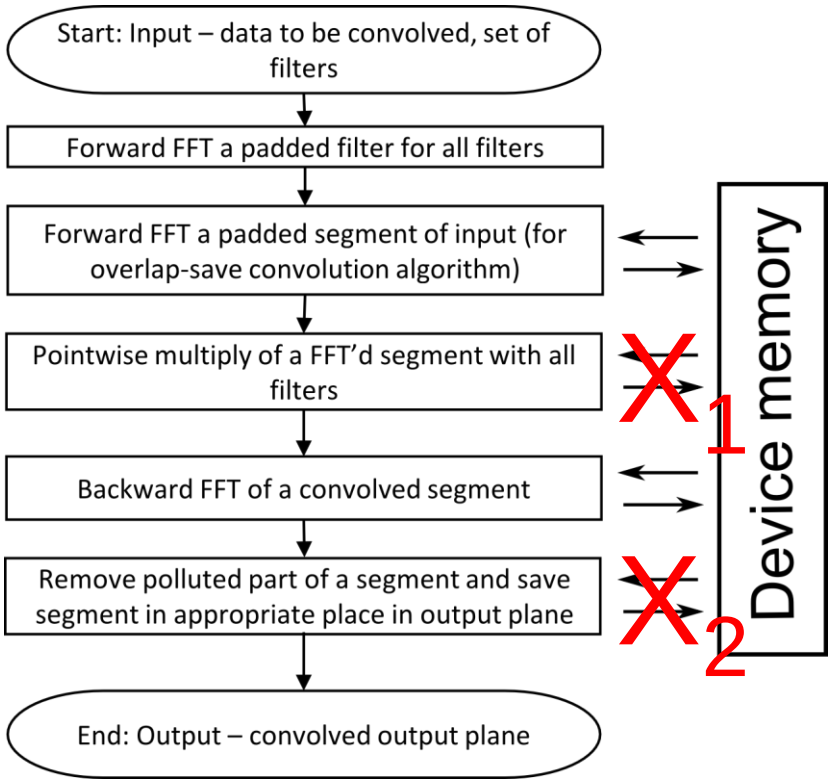
Access to the data is provided per element basis.

Two places where we could use callbacks

X_1 – when we would multiply output from forward FFT with filters

X_2 – when would remove polluted parts after inverse FFT

However for non-local operation on elements (like interbinning) callbacks cannot be used. They are not suitable for FDAS example.



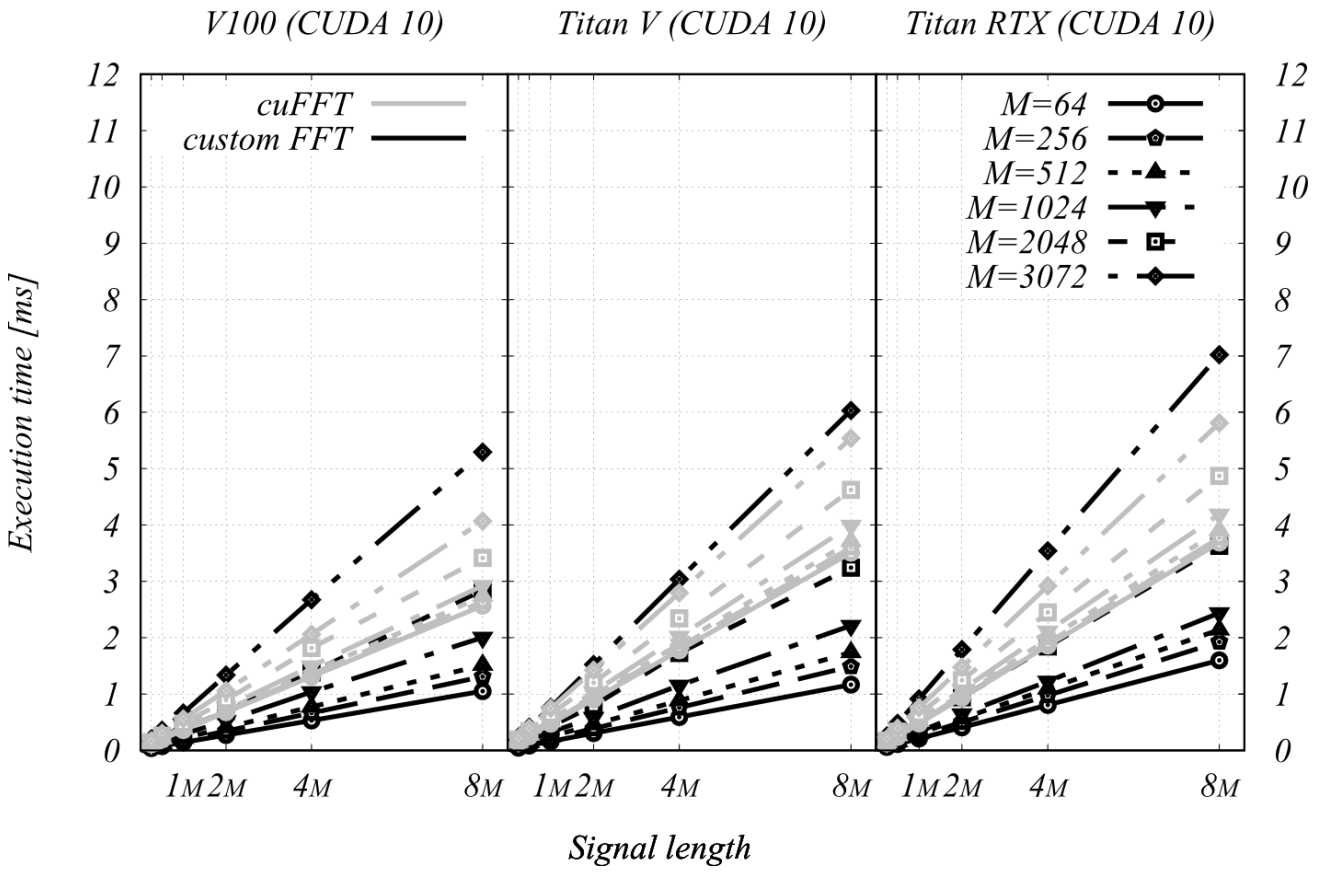
Performance comparison

Input and output are complex. M is length of the filter. For number of filters F=8.

LEFT:
Shows the execution time of custom FFT OLS in black and cuFFT OLS with callbacks in gray.

Spread in ex. time is due to segment size. The custom FFT is restricted to 4096.

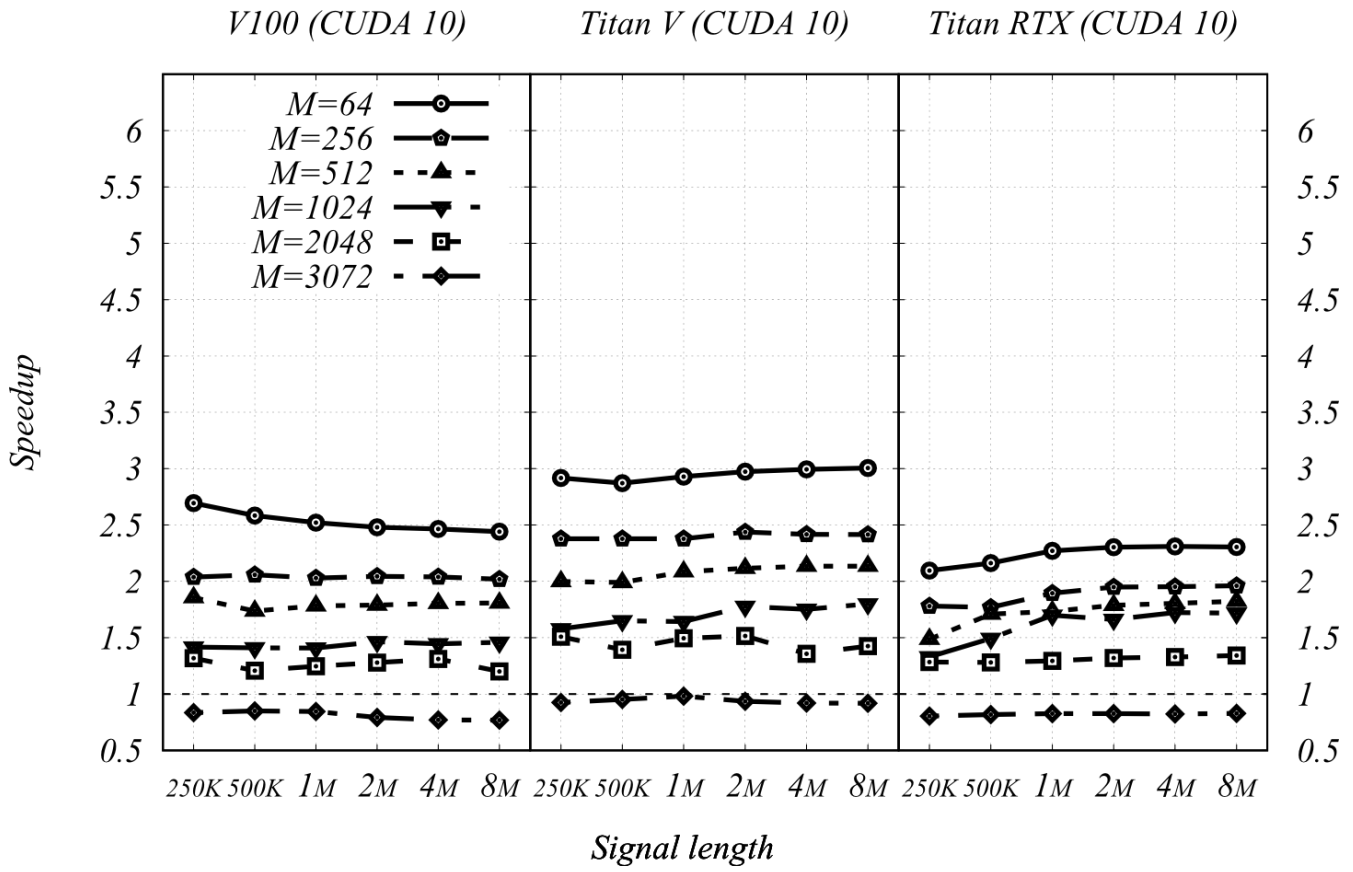
The execution time scales linearly with increasing input signal length.



Performance comparison with callbacks

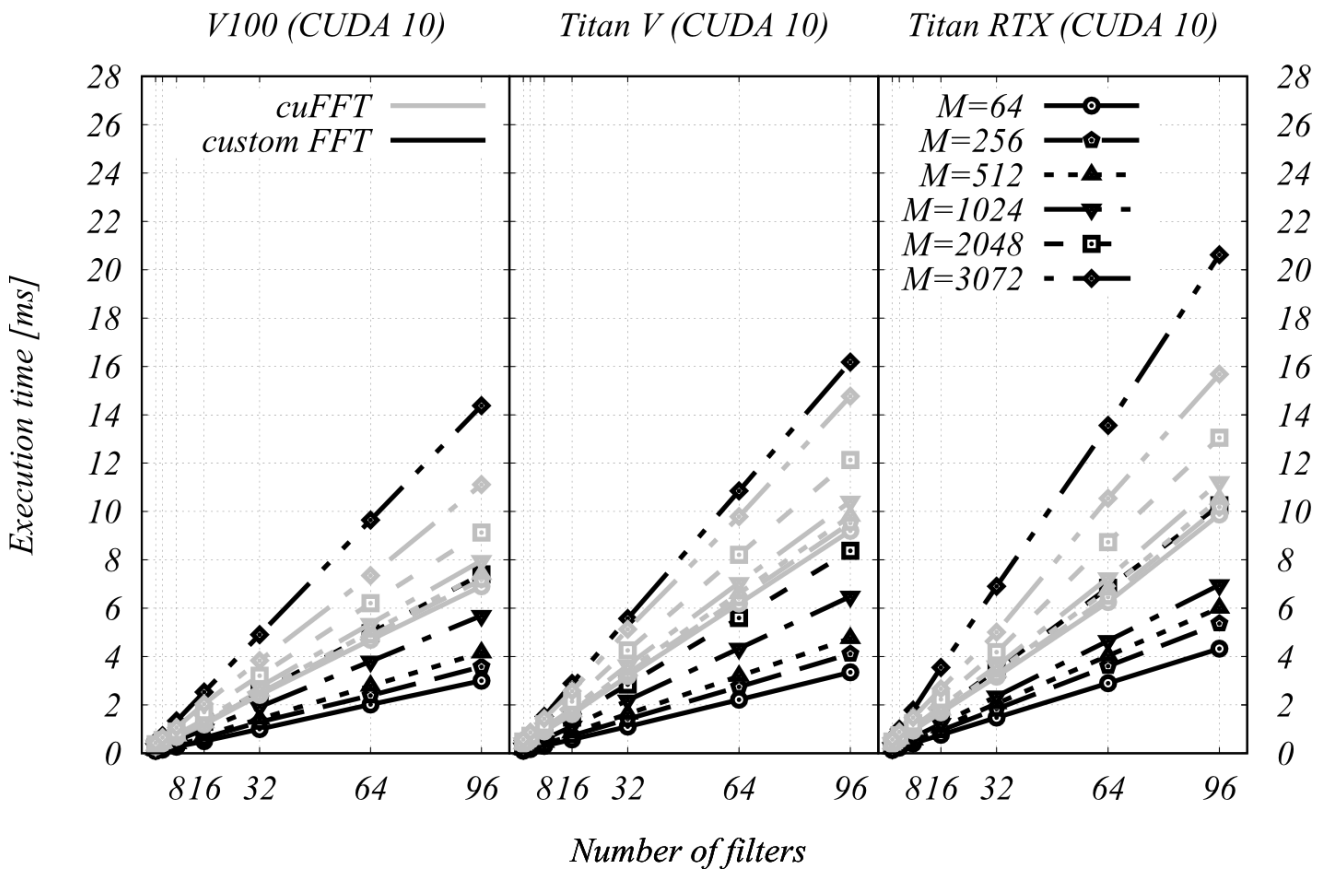
Input and output are complex. M is length of the filter.

LEFT:
Even with callbacks used in the OLS with cuFFT, the speed-ups are decent for filter sizes <1024 samples.



Performance comparison

Input and output are complex. M is length of the filter. Signal length N=2Mil.



RIGHT:
Shows the execution time of custom FFT OLS in black and cuFFT OLS with callbacks in gray.

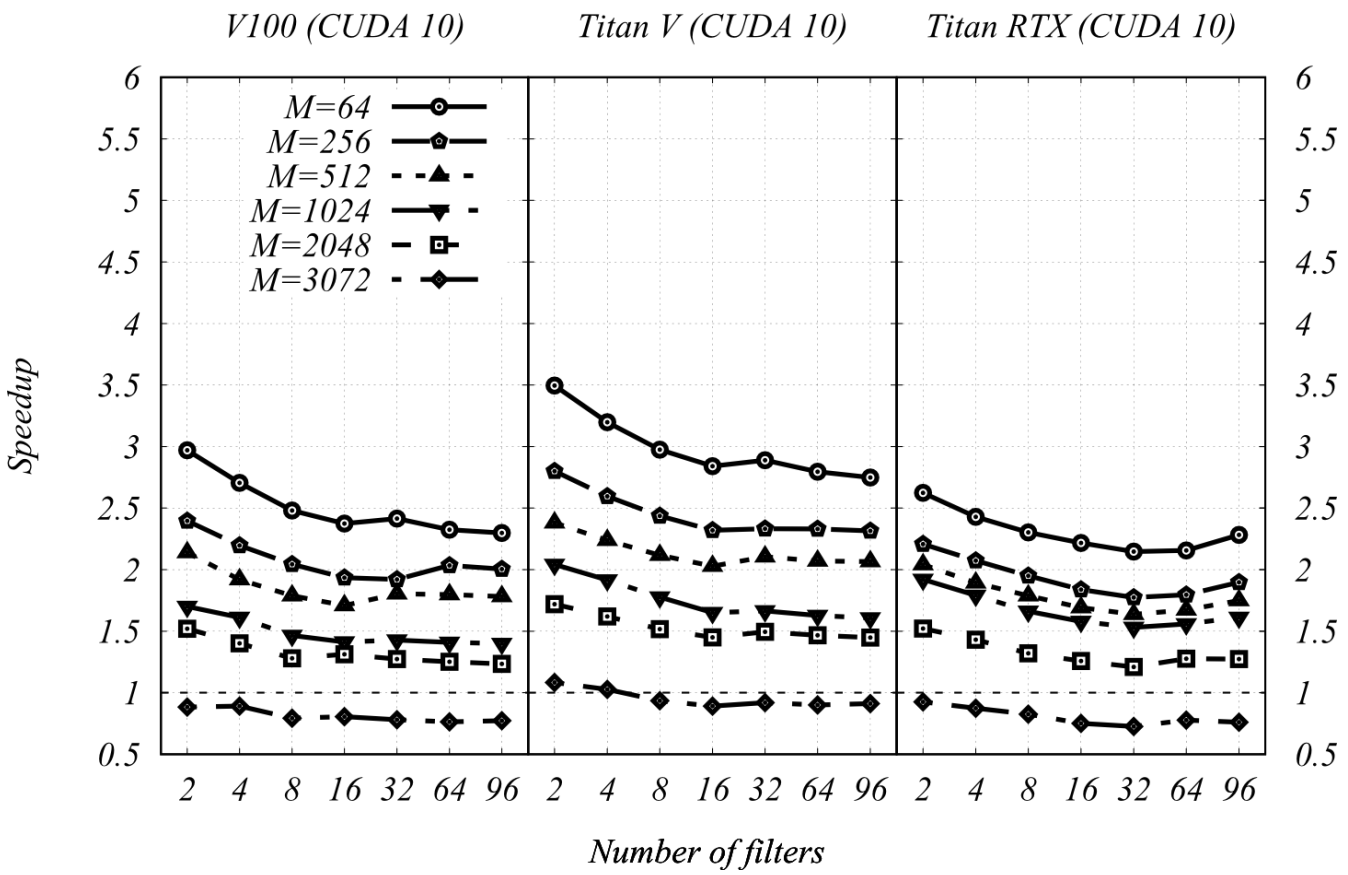
The execution time scales linearly with increasing number of filters.

UNIVERSITY OF OXFORD

Oxford e-Research Centre

Performance comparison with callbacks

Input and output are complex. M is length of the filter.

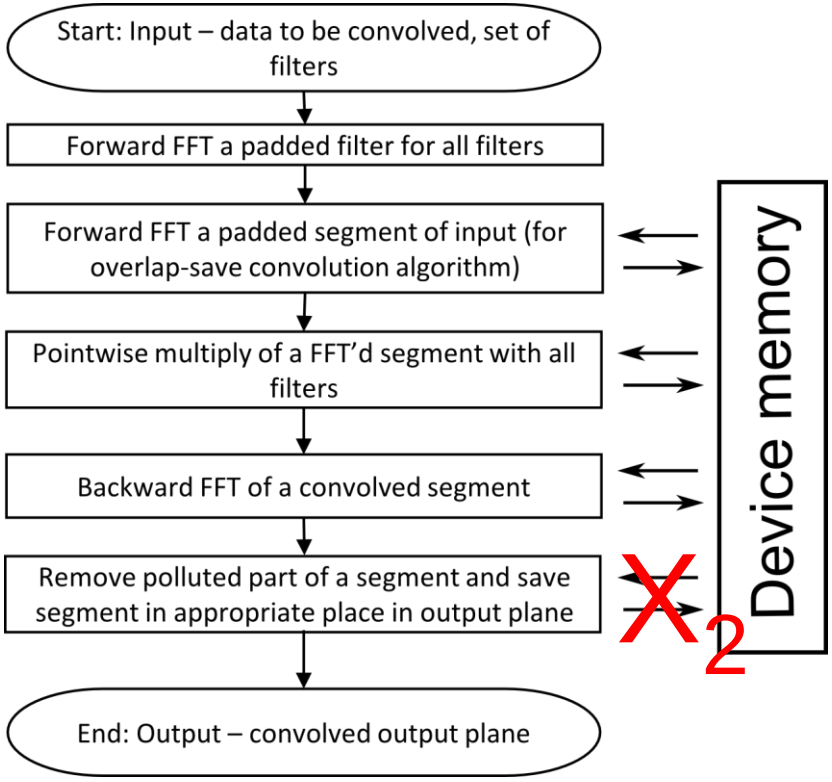


RIGHT:
Even with callbacks used in the OLS with cuFFT, the speed-ups are, decent for filter sizes <1024 samples.

Convolutions with post-processing

Convolution with post-processing

For non-local post-processing of the convolution callbacks are not usable (like in FDAS).



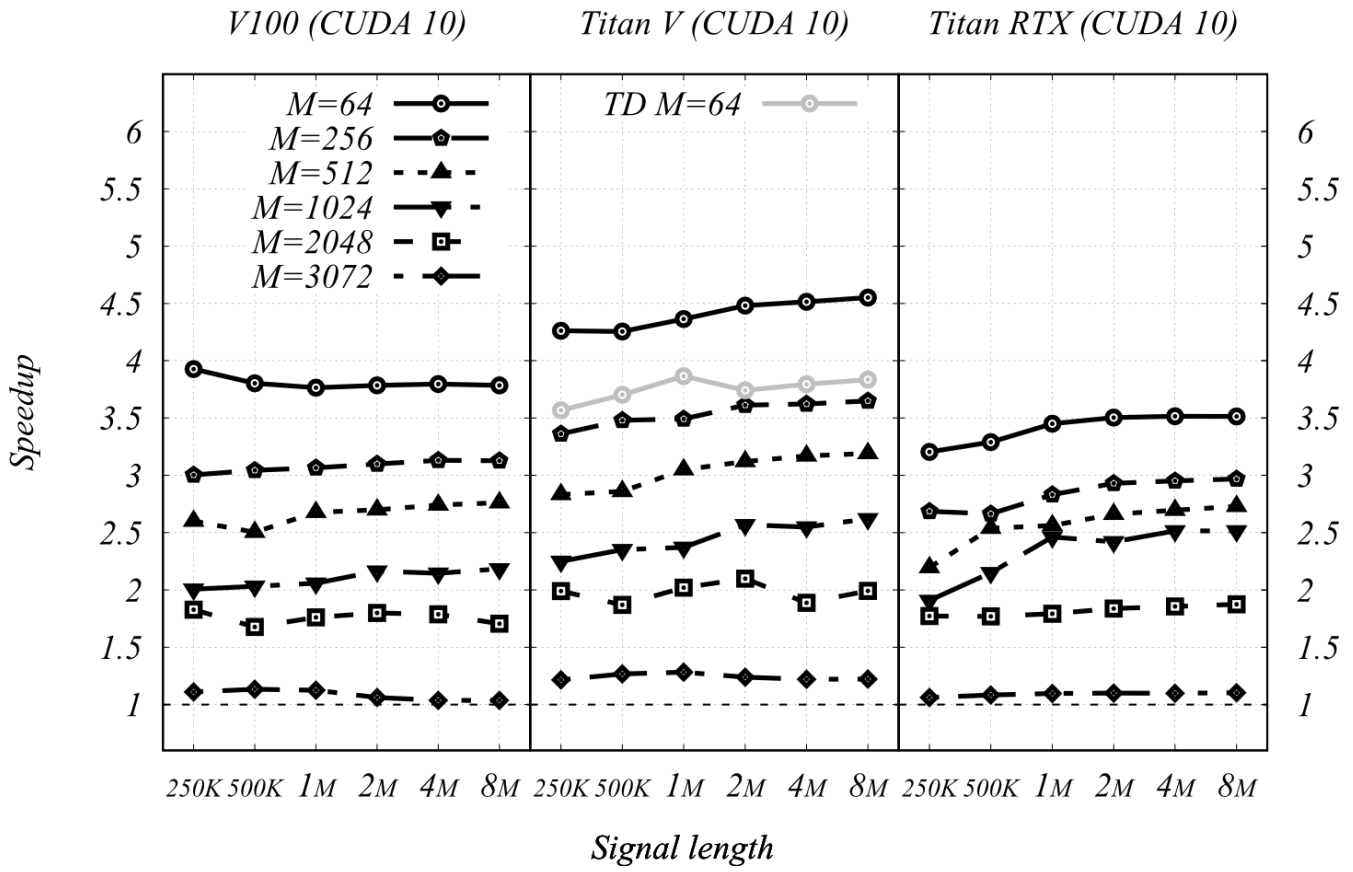
Performance comparison

Input and output are complex. M is length of the filter. For number of filters F=8.

LEFT:
Shows speed-up of custom FFT OLS over cuFFT OLS.
Speed-up to time-domain convolution in gray.

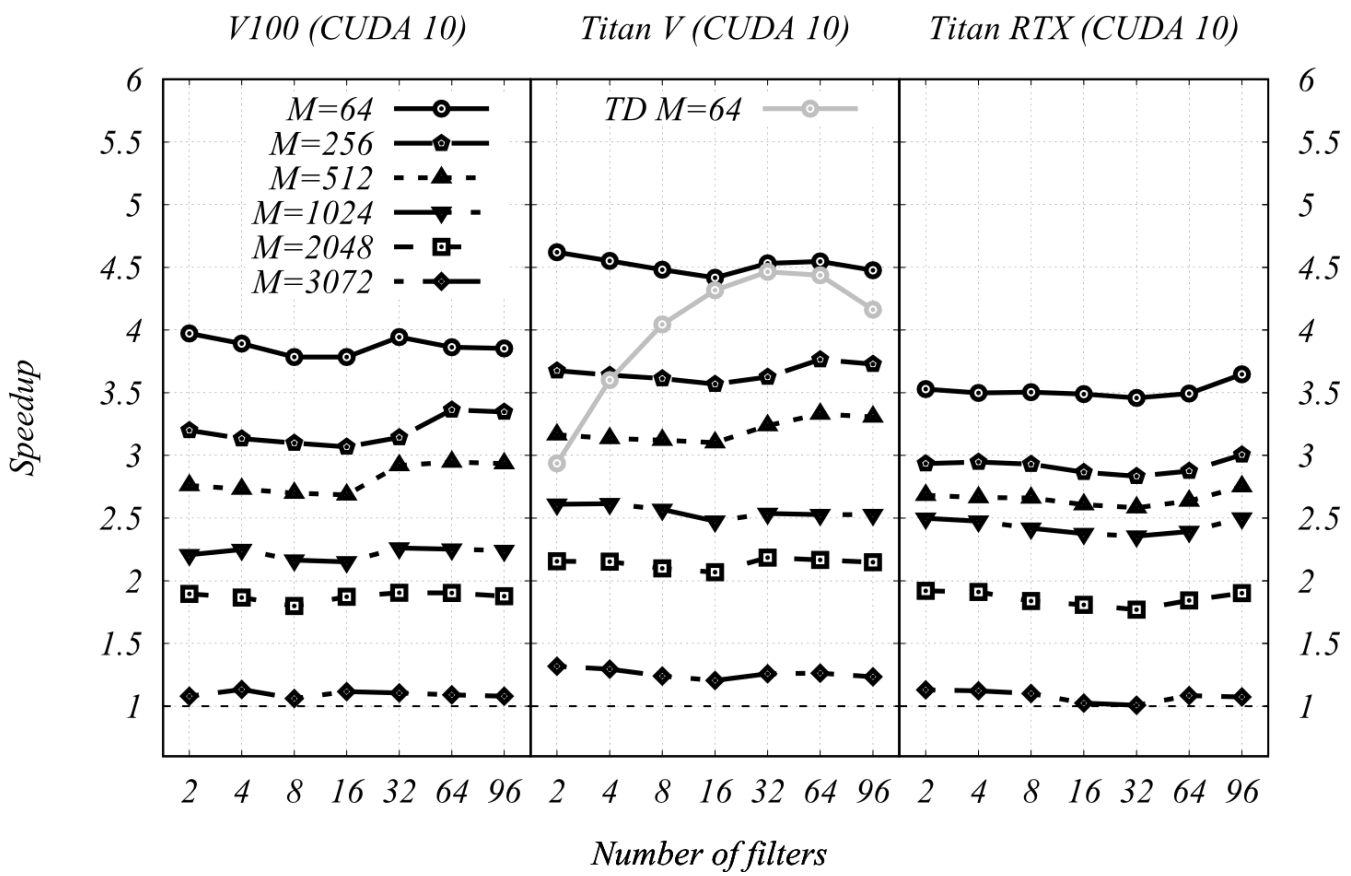
Speed-up is more or less constant as signal length is increasing.

Performance is gained by not performing global accesses. These scales with number of segments.



Performance comparison

Input and output are complex. M is length of the filter. Signal length N=2Mil.



RIGHT:
Shows speed-up of custom FFT OLS over cuFFT OLS.

Speed-up is more or less constant as number of filters is increasing.

Performance is gained by not performing global accesses. These scales with number of elements (segments).

UNIVERSITY OF OXFORD

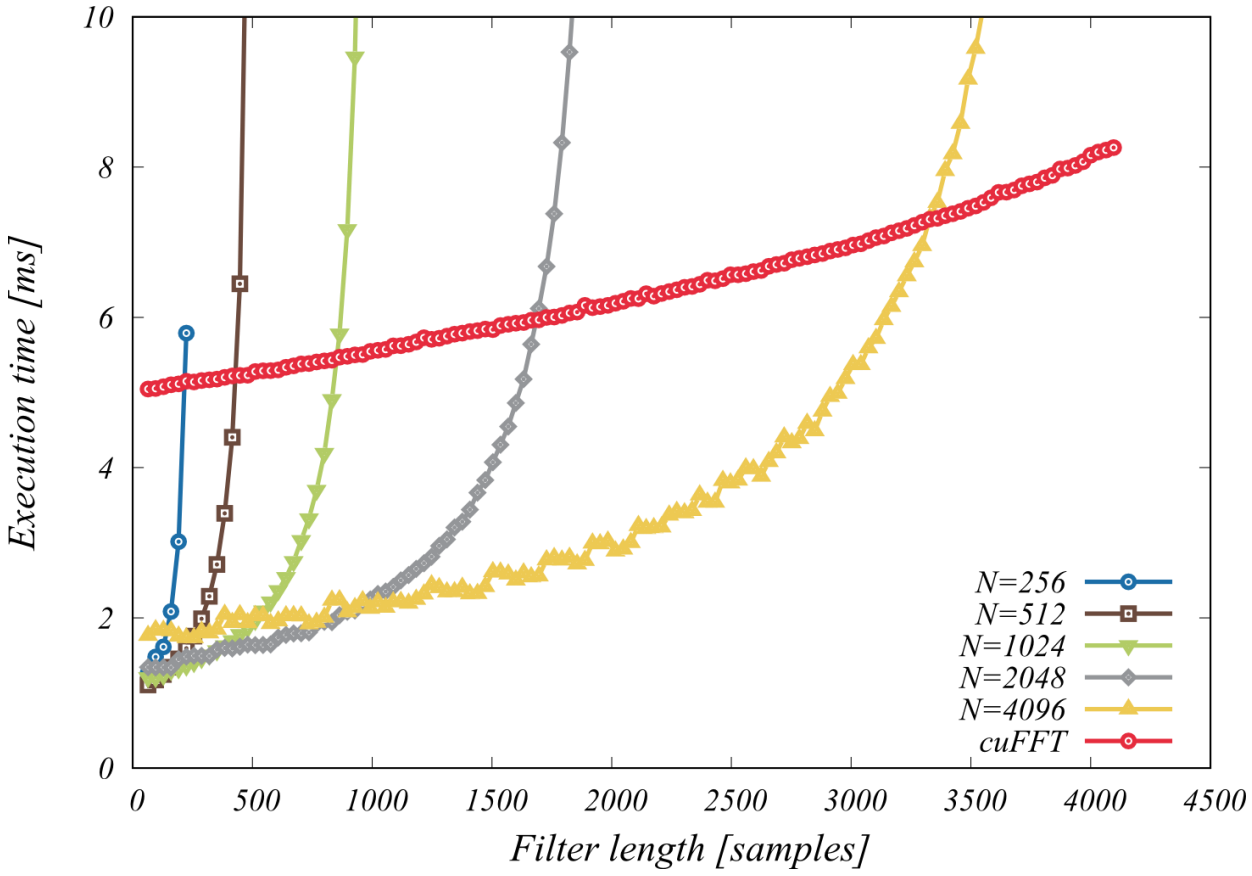
Oxford e-Research Centre

When cuFFT wins

Convolution of N=2Mil samples, 32 filters.

LEFT:
Shows execution time of OLS with custom FFT for different segment sizes (FFT size) and how it depends on filter size.

The execution time for OLS with cuFFT with optimal segment size (usually 8192) is shown in red.



Conclusions

- Implemented custom shared memory FFT and integrated into overlap-and-save method
- For convolution with post-processing on V100 using OLS with custom FFT we have achieved speedup 2x up to 4x over OLS with cuFFT depending on filter size (<2048)
- For convolutions without postprocessing on V100 we have achieved speedups 1.8x up to 2.5x for filters <512. (2x-2.8x for Titan V)
- Investigating possibility of 2D convolutions

End

Thank you for your attention!