Fast Fourier Transforms (FFTs) and Graphical Processing Units (GPUs)

Kate Despain

CMSC828e

Outline

- Motivation
- Introduction to FFTs
  - Discrete Fourier Transforms (DFTs)
  - Cooley-Tukey Algorithm
- CUFFT Library
- High Performance DFTs on GPUs by Microsoft Corporation
  - Coalescing
  - Use of Shared Memory
  - Calculation-rich Kernels

Motivation: Uses of FFTs

- Scientific Computing: Method to solve differential equations

  For example, in Quantum Mechanics (or Electricity & Magnetism) we often assume solutions to Schrodinger's Equation (or Maxwell's equations) to be plane waves, which are built on a Fourier basis

$$A = \sum_{k=-\infty}^{\infty} A_0 e^{ikx}$$

  Then, in k-space, derivative operators become multiplications

$$\frac{\partial A}{\partial x} = i \sum_{k=-\infty}^{\infty} A_0 k e^{ikx}$$

Motivation: Uses of FFTs

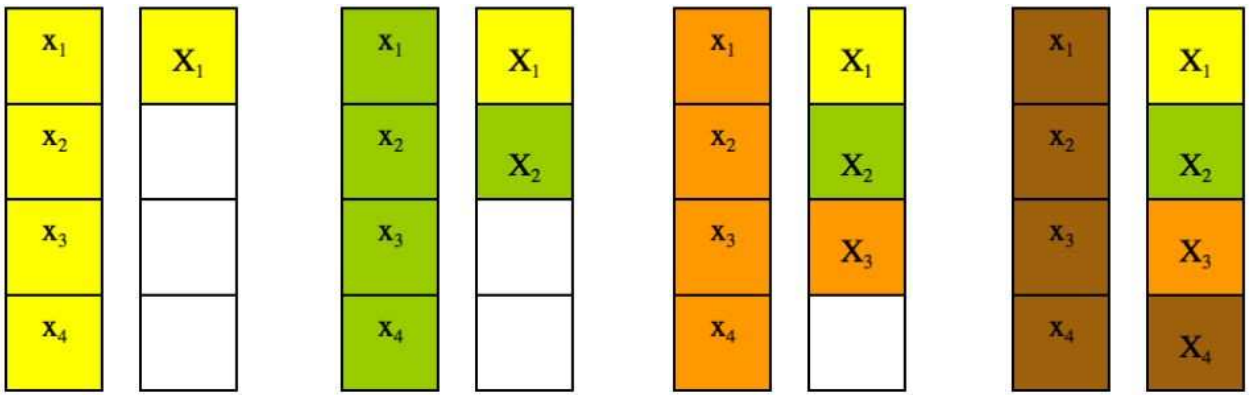
- Digital Signal Processing & Image Processing
  - Receive signal in the time domain, but want the frequency spectrum
- Convolutions/Filters
  - Filter can be represented mathematically by a convolution
  - Using the convolution theorem and FFTs, filters can be implemented efficiently

Convolution Theorem: The Fourier transform of a convolution is the product of the Fourier transforms of the convoluted elements.

Introduciton: What is an FFT?

- Algorithm to compute Discrete Fourier Transform (DFT)
  - Straightforward implementation requires $\mathcal{O}\left(N^2\right)$ MADD operations

$$X_k = \sum_{n=0}^{N-1} x_n \exp -\frac{2\pi i}{N} kn$$



Introduction: Cooley-Tukey

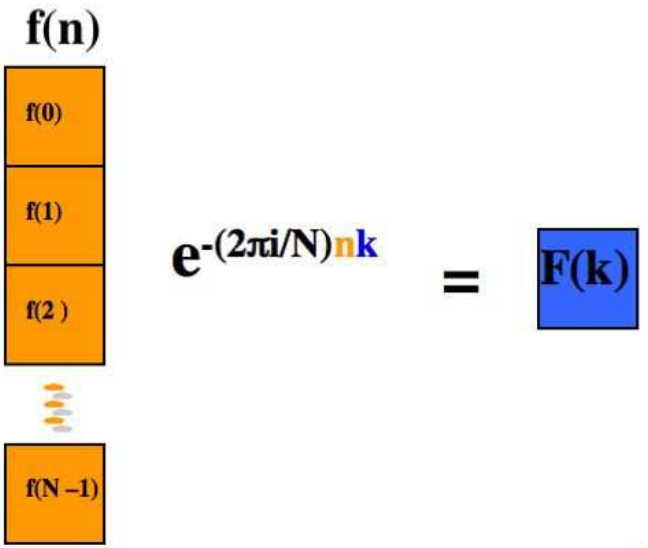
- FFTs are a subset of efficient algorithms that only require $\mathcal{O}\left(N \log N\right)$ MADD operations
- Most FFTs based on Cooley-Tukey algorithm (originally discovered by Gauss and rediscovered several times by a host of other people)

Consider N as a composite, $N = r_1 r_2$. Let $k = k_1 r_1 + k_0$ and $n = n_1 r_2 + n_0$. Then,

$$X\left(k_1, k_0\right) = \sum_{n_0=0}^{r_2-1} \sum_{n_1=0}^{r_1-1} x\left(n_1, n_0\right) \exp -\frac{2\pi i}{N} k\left(n_1 r_2 + n_0\right)$$

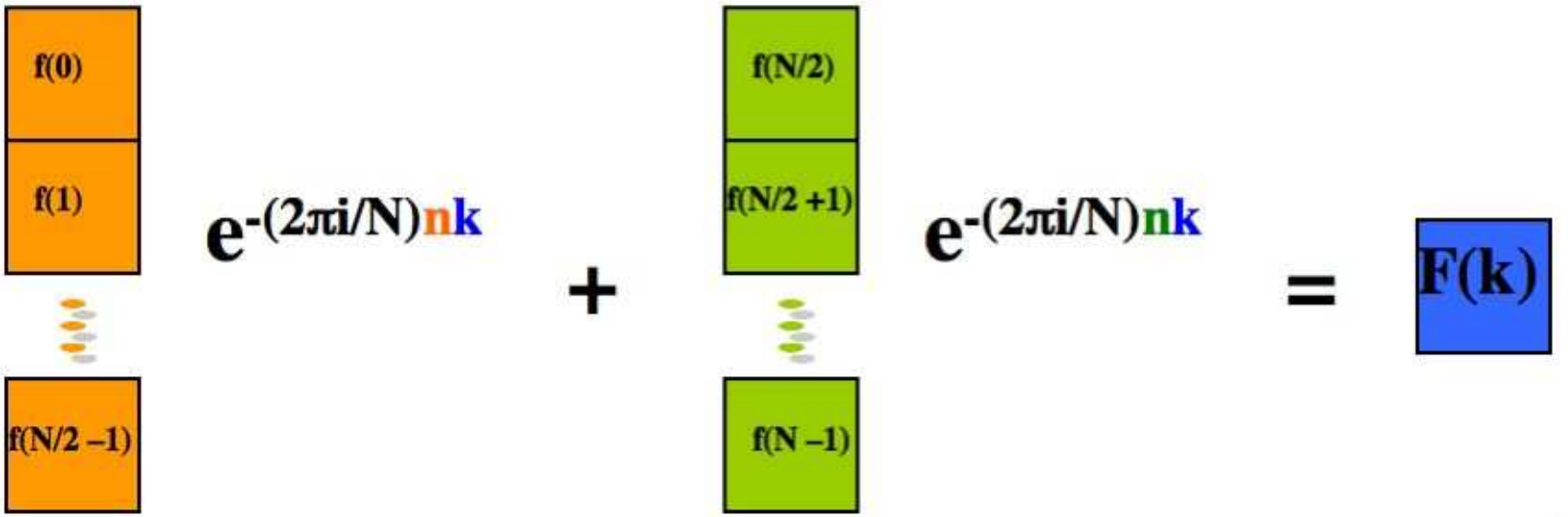
The sum over n_1 only depends on k_0, leading to r_1 operations for calculating a single k_0 output value. The second sum over n_0 requires r_2 operations for calculating a single k_1 output value, for a total of $r_1 + r_2$ operations per output element. If you divide the transform into m equal composites, you ultimately get $rN \log_r N$ operations.

Cartoon Math for FFT - I



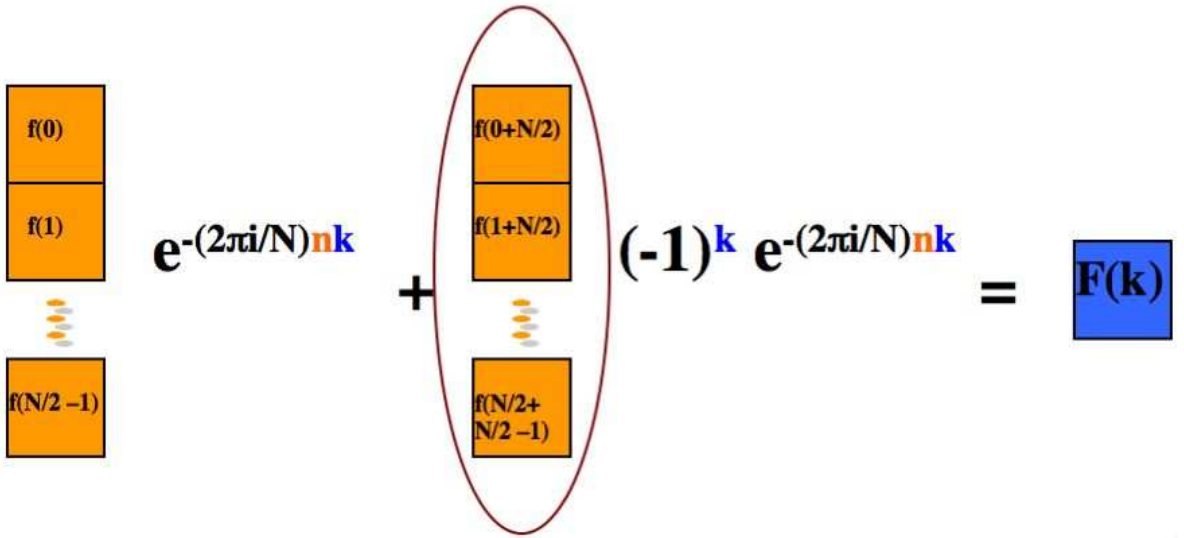
For each element of the output vector $\mathbf{F}(\mathbf{k})$, we need to multiply each element of the input vector, $\mathbf{f}(\mathbf{n})$ by the correct exponential term, $\mathbf{e}^{-\frac{2\pi \mathbf{i}}{\mathbf{N}}\mathbf{nk}}$ where $\mathbf{n}$ is the corresponding index of the element of the input vector and $\mathbf{k}$ is the index of the element of the output vector.

Cartoon Math for FFT - II



We could divide the input vector into two and create two separate sums - one going from $\mathbf{n = 0...N/2 - 1}$ and one going from $\mathbf{n = N/2...N - 1}$.

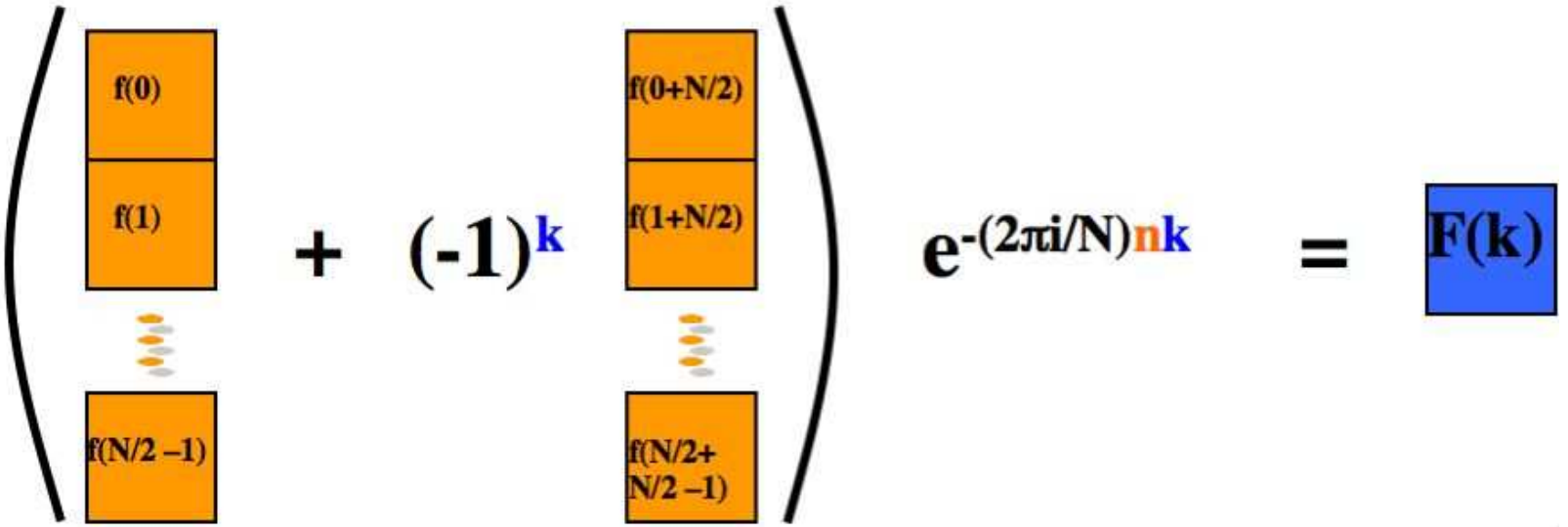
Cartoon Math for FFT - III



We can change the summation index on the second sum to match that of the first sum. We let $\mathbf{n} \rightarrow \mathbf{n} + \mathbf{N/2}$ in the second sum. This introduces an exponential term of the form

$$\mathbf{e}^{-\frac{2\pi \mathbf{i}}{\mathbf{N}} \frac{\mathbf{N}}{2} \mathbf{k}} = \mathbf{e}^{-\pi \mathbf{i} \mathbf{k}} = \left(-\mathbf{1}\right)^{\mathbf{k}}$$

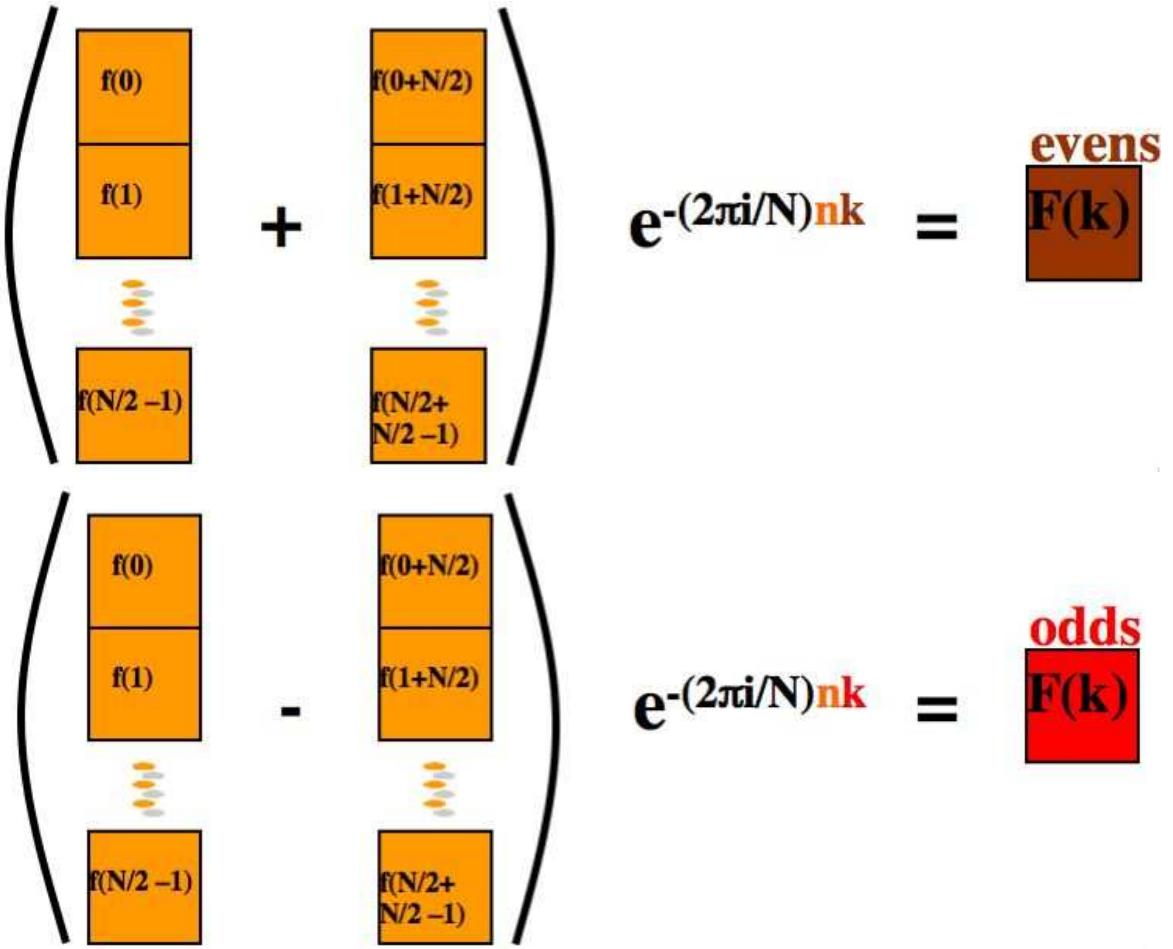
Cartoon Math for FFT - IV



If we factor appropriately, we can eliminate one multiplication per input element! (This translates to a net savings of $N/2$ MADD operations.)

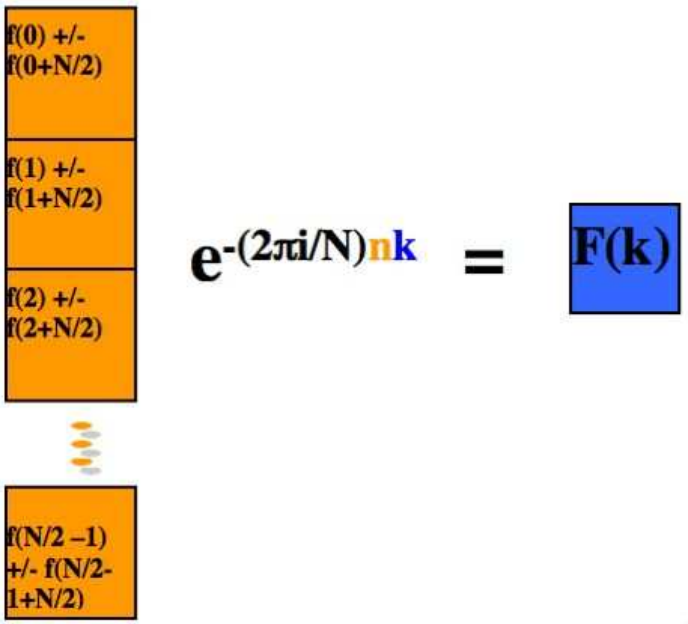
But wait, there's more...

Cartoon Math for FFT - V



We could take the output vector and divide it into two this time - according to whether **k** is even or odd.

Cartoon Math for FFT - VI

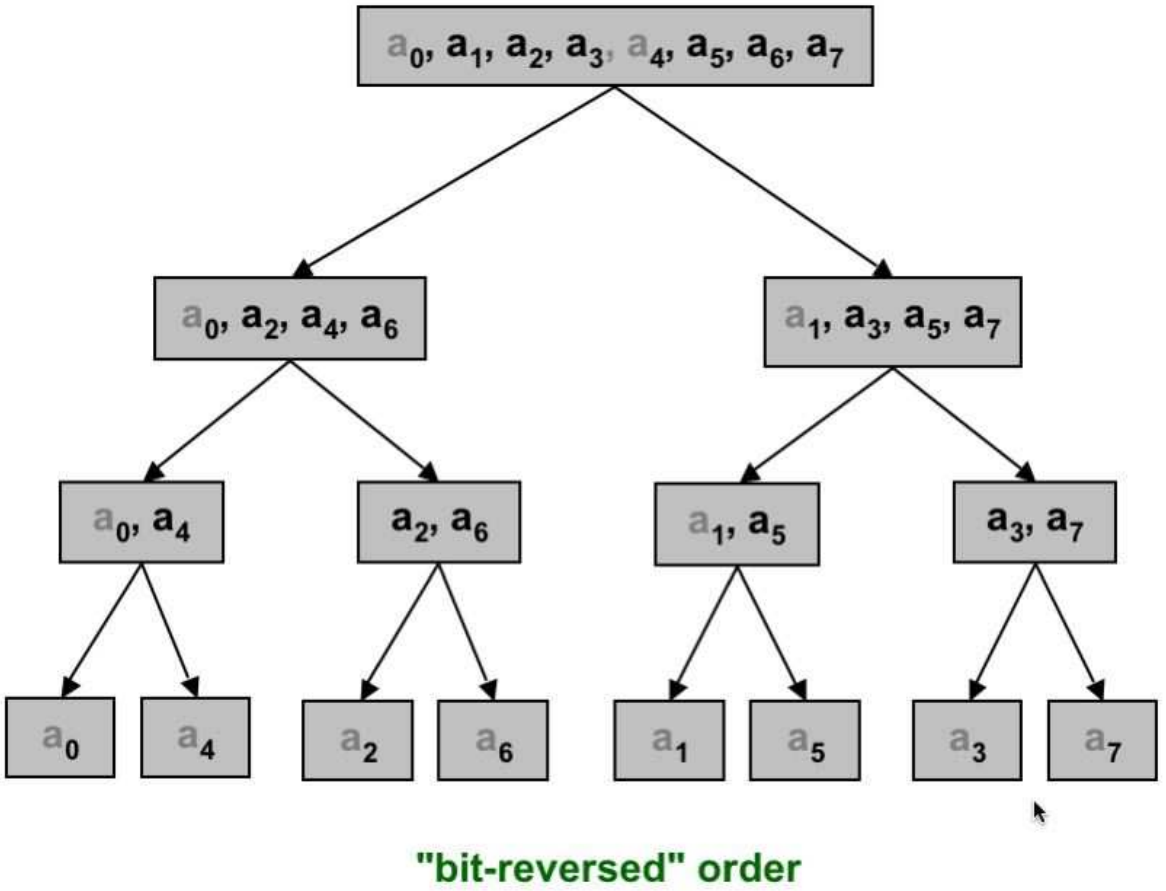


For any given **k** we now have something that looks similar to our original Fourier Transform. We can repeat this procedure recursively. Each output element requires $\sim \log_2 \mathbf{N}$ operations, and since there are **N** output elements, we get $\mathcal{O}(\mathbf{N} \log_2 \mathbf{N})$ operations as promised.

Additional FFT Information

- Radix-r algorithms refer to the number of r-sums you divide your transform into at each step
- Usually, FFT algorithms work best when r is some small prime number (original Cooley-Tukey algorithm optimizes at $r = 3$)
- However, for $r = 2$, one can utilize bit reversal on the CPU
  - When the output vector is divided into even and odd sums, the location of the output vector elements can get scrambled in memory. Fortunately, it's usually in such a way that you can bit reverse adjacent locations in memory and work your way back to ordered output.

Scrambled Output of the FFT



Duraiswami, Ramani. "Fourier transform." *Computational Methods* CMSC/AMSC/MAPL 460 course, UMCP.

CUFFT - FFT for CUDA

- Library for performing FFTs on GPU
- Can Handle:
  - 1D, 2D or 3D data
  - Complex-to-Complex, Complex-to-Real, and Real-to-Complex transforms
  - Batch execution in 1D
  - In-place or out-of-place transforms
  - Up to 8 million elements in 1D
  - Between 2 and 16384 elements in any direction for 2D and 3D

1D Complex-to-Complex

Example for batched, in-place case:

```
#include <cufft.h>
#define NX 256
#define BATCH 10
cufftHandle plan;
cufftComplex *data;

cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);

/* Create a 1D FFT plan.  */
cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);

/* Use the CUFFT plan to transform the signal in place.  */
cufftExecC2C(plan, data, data, CUFFT_FORWARD);

/* Destroy the CUFFT plan.  */
cufftDestroy(plan);
cudaFree(data);
```

CUDA CUFFT Library, v. 2.1 (2008) Santa Clara, CA: NVIDIA Corporation

2D Complex-to-Real

Example for out-of-place case:

```
#define NX 256
#define NY 128
cufftHandle plan;
cufftComplex *idata;
cufftReal *odata;

cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftReal)*NX*NY);

/* Create a 2D FFT plan.  */
cufftPlan2d(&plan, NX, NY, CUFFT_C2R);

/* Use the CUFFT plan to transform the signal out of place.  */
cufftExecC2R(plan, idata, odata);

/* Destroy the CUFFT plan.  */
cufftDestroy(plan);
cudaFree(idata); cudaFree(odata);
```

CUDA CUFFT Library, v. 2.1 (2008) Santa Clara, CA: NVIDIA Corporation

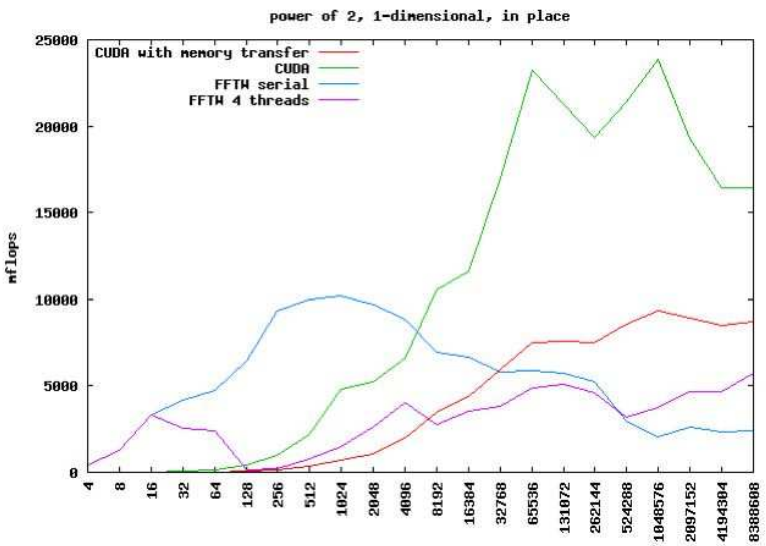
CUFFT Performance vs. FFTW

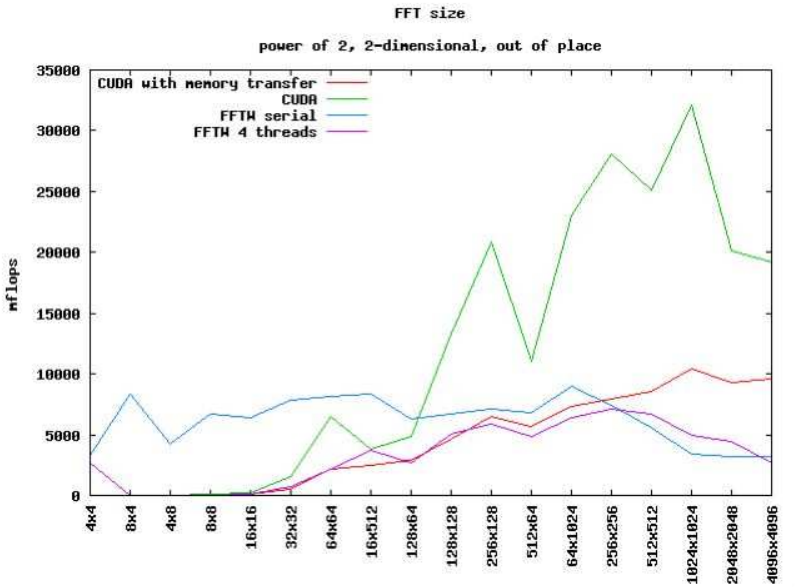
Group at University of Waterloo did some benchmarks to compare CUFFT to FFTW. They found that, in general:

- CUFFT is good for larger, power-of-two sized FFT's
- CUFFT is not good for small sized FFT's
  - CPUs can fit all the data in their cache
  - GPUs data transfer from global memory takes too long

University of Waterloo. (2007). http://www.science.uwaterloo.ca/~hmerz/CUDA_benchFFT/

CUFFT Performance vs. FFTW





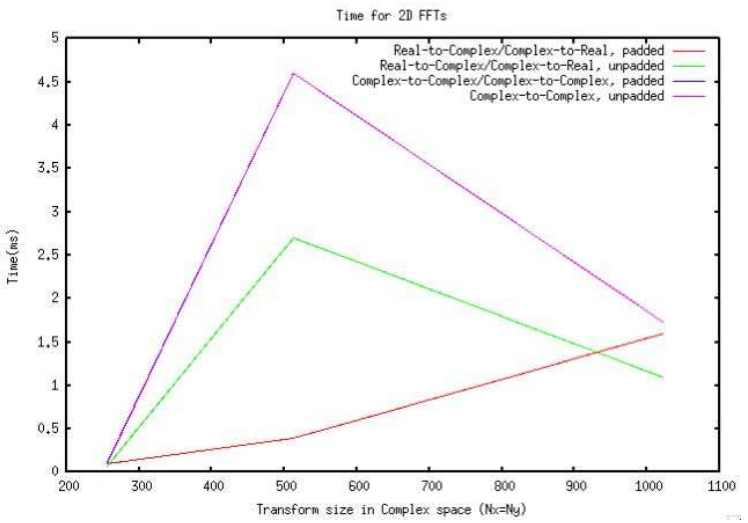
CUFFT starts to perform better than FFTW around data sizes of 8192 elements. Though I don't show it here, **nflops** for CUFFT do decrease for non-power-of-two sized FFT's, but it still beats FFTW for most large sizes ($> \sim$ 10,000 elements)

University of Waterloo. (2007). http://www.science.uwaterloo.ca/~hmerz/CUDA_benchFFT/

CUFFT Performance

CUFFT seems to be a sort of "first pass" implementation. It doesn’t appear to fully exploit the strengths of mature FFT algorithms or the hardware of the GPU.

For example, "Many FFT algorithms for real data exploit the conjugate symmetry property to reduce computation and memory cost by roughly half. However, CUFFT does not implement any specialized algorithms for real data, and so there is no direct performance benefit to using real-to-complex (or complex-to-real) plans instead of complex-to-complex." - *CUDA CUFFT Library, v. 2.1* (2008) Santa Clara, CA: NVIDIA Corporation



Latest Developments

"High Performance Discrete Fourier Transforms on Graphics Processors" – Govindaraju, NK, et al. Presented at *SC08*.

- Use the Stockham algorithm (requires out-of-place transform)
- Exploit coalescing with global memory
- Exploit fast access to shared memory
- Calculation-rich kernels

Coalescing I

- Refers to global memory access
- Memory transferred in "segments"
  - For Compute Capability 1.0 or 1.1(e.g. 9800s and below), segment size = 64- or 128-bytes
  - For Compute Capability 1.2 and higher, segment size = 32-, 64-, or 128-bytes
- To achieve *coalescing*
  - A half-warp should utilize all bytes in a given memory transfer (e.g. each thread accesses a 16-bit word)
  - Adjacent threads should access adjacent memory

NVIDIA CUDA Programming Guide, v.21. (2008).

Coalescing II

Left: coalesced `float` memory access, resulting in a single memory transaction.
Right: coalesced `float` memory access (divergent warp), resulting in a single memory transaction.

Figure 5-1. Examples of Coalesced Global Memory Access Patterns

Can still have *coalescence* with divergent threads, but the bandwidth is lower.

NVIDIA CUDA Programming Guide, v.21. (2008). p. 59

Coalescing III -Code Example

Fig 2 shows a bit of pseudo-code that employs *coalescence*. (From: "High Performance Discrete Fourier Transforms on Graphics Processors" – Govindaraju, NK, et al. *SC08*).

```
17  void FftIteration(int j, int N, int R, int Ns,
18                    float2* data0, float2*data1){
19    float2 v[R];
20    int idxS = j;
21    float angle = -2*M_PI*(j%Ns)/(Ns*R);
22    for( int r=0; r<R; r++ ) {
23      v[r] = data0[idxS+r*N/R];
24      v[r] *= (cos(r*angle), sin(r*angle));
25    }
26    FFT<R>( v );
27    int idxD = expand(j,Ns,R);
28    for( int r=0; r<R; r++ )
29      data1[idxD+r*Ns] = v[r];
30  }
```

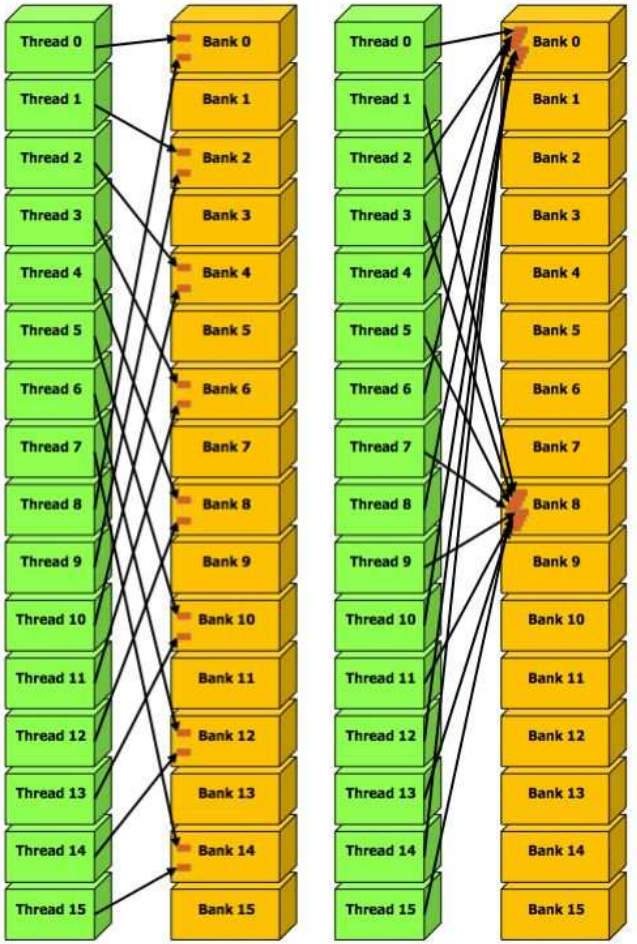
- For each iteration of the loop, `r`, `idxS` = thread ID and `T` = `N/R`, the # of threads per block. (I believe `v[R]` lives in registers if `R` is small enough.)
- `N` and `R` can be chosen such that `T` threads participate in coalesced reads. (e.g. `T= 64`; each thread reads 16-bits)
- Coalesced writes are more complicated and require the use of shared memory.

Challenges of Shared Memory

- Limited to roughly 16kB/multiprocessor (or block)
- Organized into 16 banks, and 32-bit words are distributed in a round-robin fashion
  - Bank conflicts if two threads from the same half-warp try to access the same bank at the same time (anything bigger than `float` is going to have problems)
  - Bank conflicts are handled through serialization
- Some overhead resides there
  - Function arguments
  - Execution configurations

NVIDIA CUDA Programming Guide, v.21. (2008).

Bank Conflicts



Left: Linear addressing with a stride of two 32-bit words causes 2-way bank conflicts.
Right: Linear addressing with a stride of eight 32-bit words causes 8-way bank conflicts.

Figure 5-7. Examples of Shared Memory Access Patterns with Bank Conflicts

NVIDIA CUDA Programming Guide, v.21. (2008). p. 69

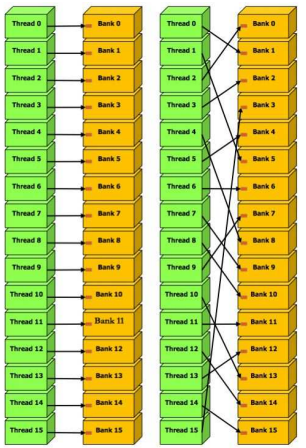
Advantages of Shared Memory

- Access can take as little as two clock cycles!
- Atomic operations allowed
- Employs a broadcast mechanism

NVIDIA CUDA Programming Guide, v.21. (2008).

Shared Memory Solutions

- Bank conflict solution: Break up your `floatN` variable into `N float` variables when storing in shared memory
- If the size of the array is not a multiple of 16, you may consider padding it. (There's a trade off between calculating the padded indices and the serialization of the bank conflicts.)



Left: linear addressing with a stride of one 32-bit word.
Right: random permutation.

Figure 5-5. Examples of Shared Memory Access Patterns without Bank Conflicts

NVIDIA CUDA Programming Guide, v.21. (2008). p. 67

Calculation-rich kernels

To increase flops and floating point operations/memory load, it helps to create kernels that involve lots of floating point operations.

- The $e^{-\frac{2\pi i}{N}nk}$ term in an FFT can be represented by a matrix on a memory-rich CPU. But, the strength of the GPU is its ALUs - it's better to calculate this term as you need it.
- Rearranging of the scrambled output array is faster using shared memory. (Once the output is unscrambled, it can be written out to global memory in a coalesced fashion.)
- Transposes (which are needed for multi-dimensional FFTs as well as for transforms that are too big to fit into shared memory all at once) are interleaved with computations and are done in the kernel itself.

Additional Tricks

- Hierarchal FFTs - break large FFT into small FFTs that will fit into shared memory
- Mixed-radix FFTs - handles non-power-of-two sizes
- Multi-dimensional FFTs- handled similar to hierarchal FFTs
- Real FFTs - exploit symmetry

Batched 1D Power-of-Two Data

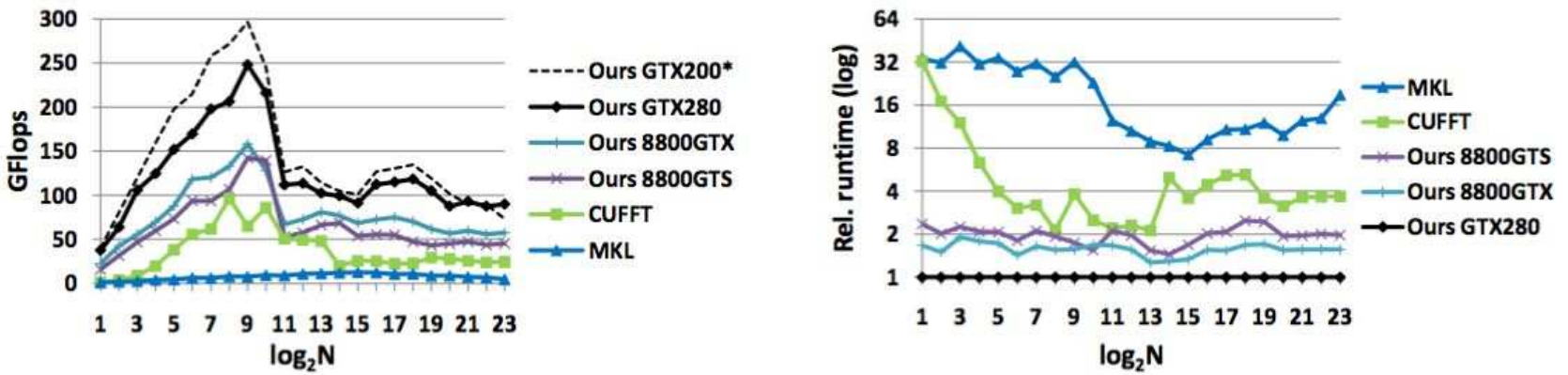


Fig. 12. **Batched 1D power-of-two FFTs.** (Left) Performance of our algorithms on multiple GPUs, CUFFT on the GTX280, and MKL. The dashed line is for performance on an older driver. (Right) Run time relative to our algorithms on GTX280 (zoomed on large values of N). The number of FFTs M is chosen as E/N, where $E = 2^{23}$, the largest value supported by CUFFT. For large N on the GTX280, our FFTs are up to 4 times faster than CUFFT and 19 times faster than MKL.

"For large N ... our FFT's are up to 4 times faster than CUFFT and 19 times faster than MKL"

"High Performance Discrete Fourier Transforms on Graphics Processors" – Govindaraju, NK, et al. *SC08*).

2D Power-of-Two Data

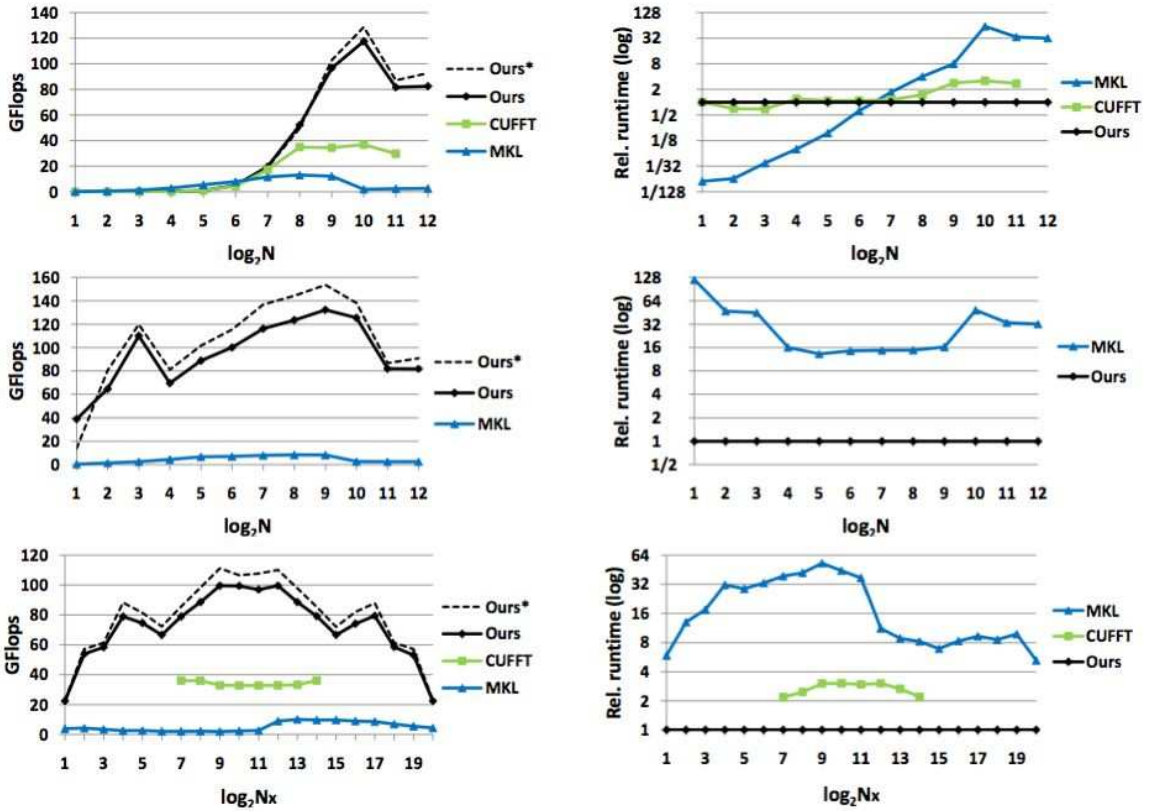


Fig. 14. **2D power-of-two FFTs.** (Top) Performance for single 2D FFTs of size $N \times N$. (Middle) Performance for M 2D FFTs of size $N \times N$, where $M = E/N^2$ and $E = 2^{24}$. CUFFT not shown because it does not support batched 2D FFTs. (Bottom) Performance of a single 2D FFT of size $N_x \times N_y$ where $N_x N_y = 2^{24}$. CUFFT currently only supports 2D FFTs with $N_x, N_y \in [2, 2^{14}]$. The dashed line is for performance on an older driver.

Top: Single 2D FFTs of size NxN; Middle: Batched 2D FFTs; Bottom: 2D FFTs of fixed size 2^{24}

"High Performance Discrete Fourier Transforms on Graphics Processors" – Govindaraju, NK, et al. *SC08*).