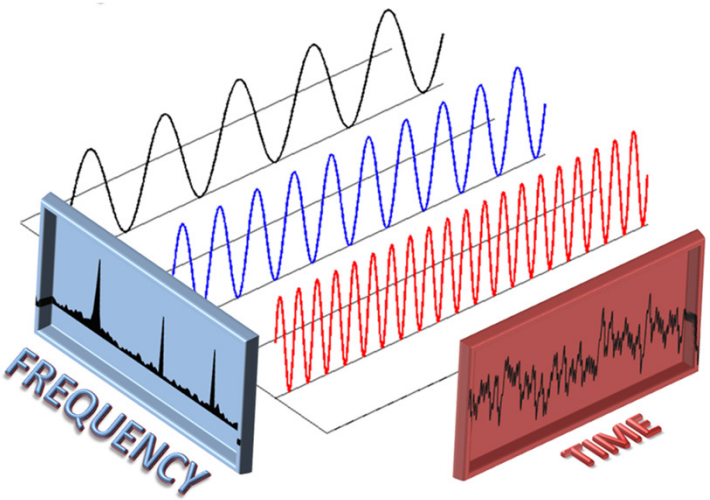
Using OpenACC to parallelize irregular computation
(Session:S7478)

Arnov Sinha (arnov@udel.edu)
M.S. (Graduating Summer'17)

Sunita Chandrasekaran
(schandra@udel.edu)
Assistant Professor

University of Delaware, DE, USA

May 10, GTC 2017, Marriott Ballroom 03

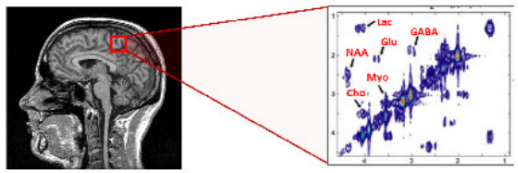


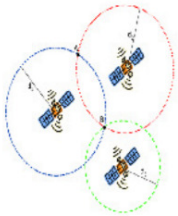
- Sparse FFT (sFFT) - a sub-optimal time linear transform used to convert Time to Frequency domain
  - An irregular algorithm
- More sparsity and larger signal size, the more difficult it gets to locate the data

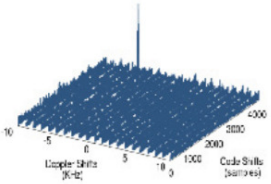
Courtesy: http://groups.csail.mit.edu/netmit/sFFT/

sFFT

Applications

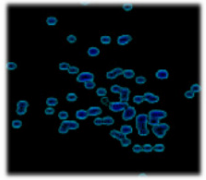






Paper: [ISMRM'13]
MRS Sparse-FFT: Reducing Acquisition Time and Artifacts for In Vivo 2D Correlation Spectroscopy

Paper: [MOBICOM'12]
Faster GPS Via the Sparse Fourier Transform



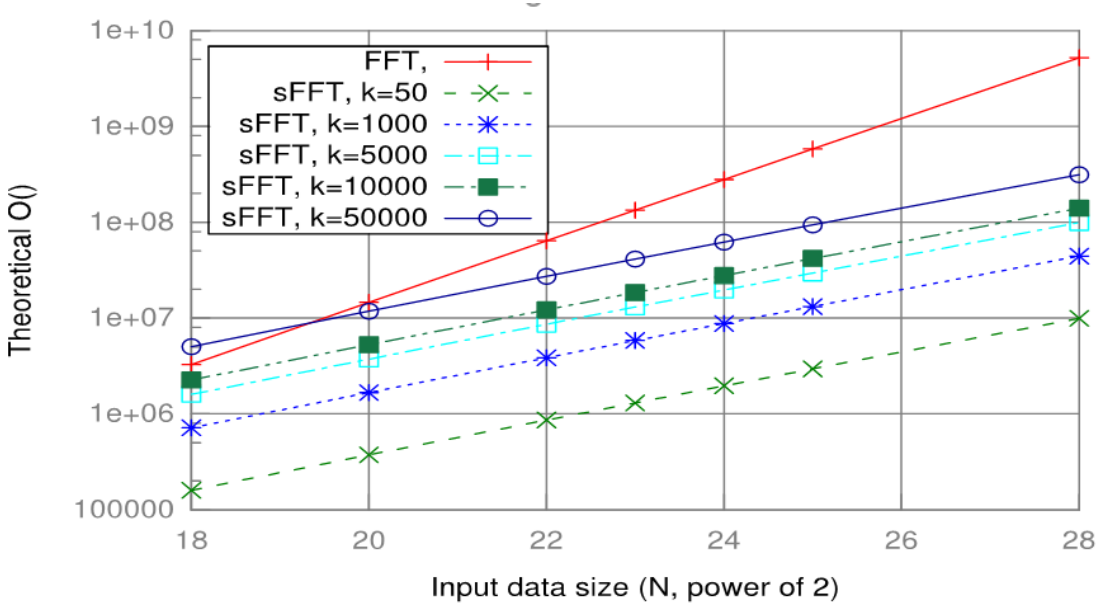
Nuclear Magnetic Resonance (NMR)

Radio Astronomy

DNA Sequencing

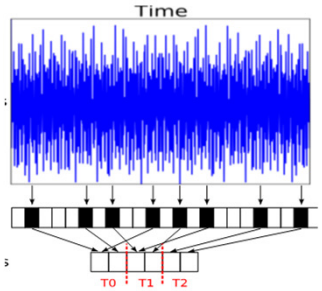
Seismic Data Analysis

MIT's sFFT



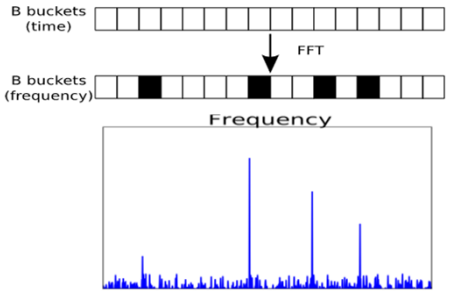
- MIT CSAIL, 2012
- Compute the k-sparse fourier transform with much lower time complexity than FFTW
- Algorithm faster than full size FFT for k, up to O(n/logn)

Random Spectrum, Permutation + filtering



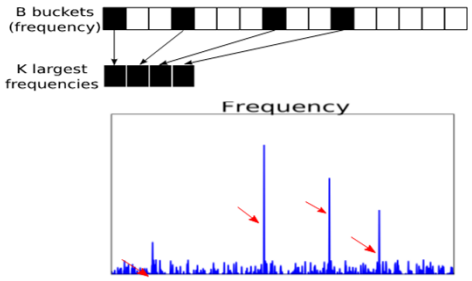
- Separating nonzero coefficients
- Ensure different locations of the signal spectrum is permuted

Subsampling FFT



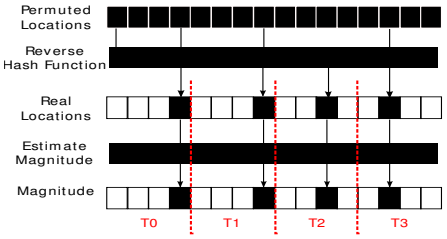
- Smoothen the sampling
- Gaussian filter

Selecting k largest Fourier coefficients



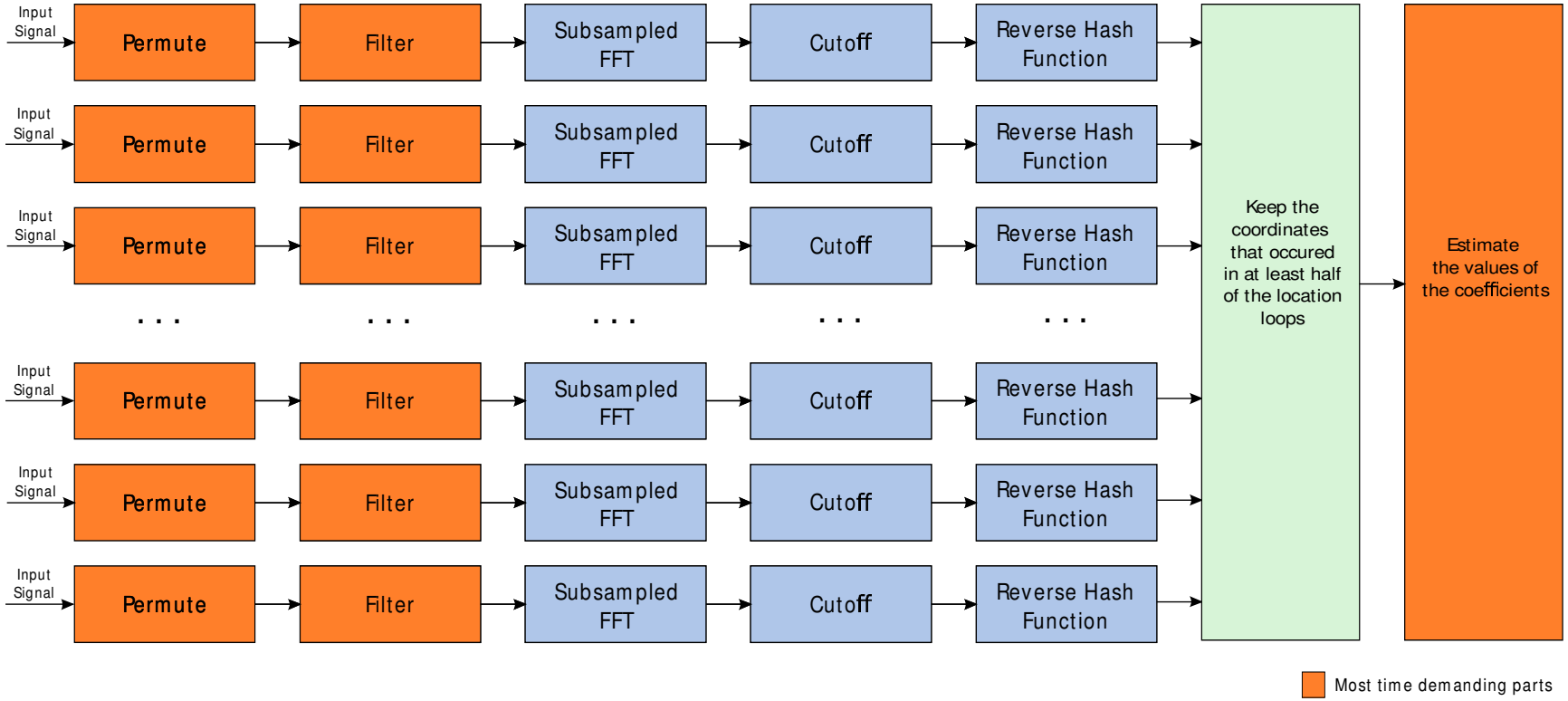
- Signal spectrum is sparse
- Most of the buckets are small
- Select top k largest coefficients from B sized buckets
- Heap sort O(B) time

Reverse hash function for location recovery, value estimation

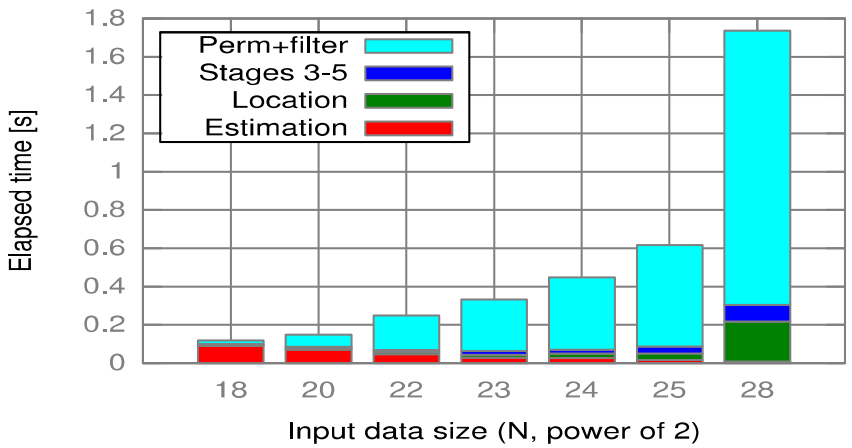


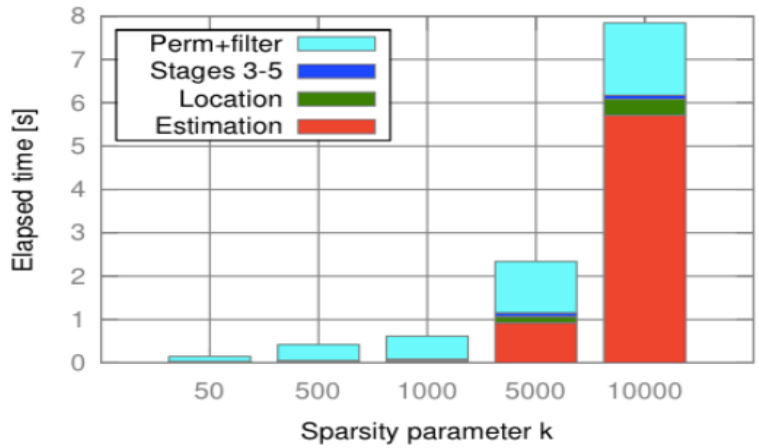
- Find the location of the large coefficients
- Recover magnitudes of coefficients found

sFFT stages


Input Signal
Permute
Filter
Subsampled FFT
Cutoff
Reverse Hash Function
Keep the coordinates that occured in at least half of the location loops
Estimate the values of the coefficients
Most time demanding parts

Profiling sparse FFT



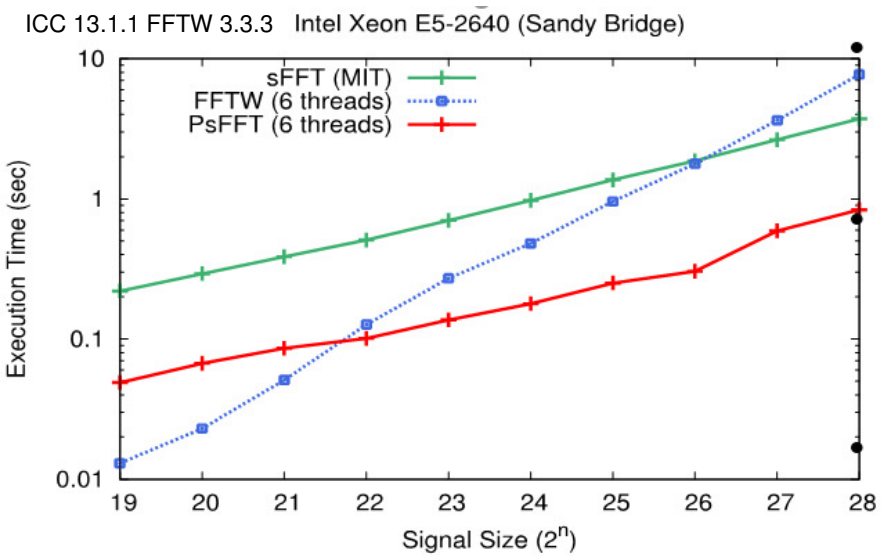


Computational hotspot in the algorithm – Permutation + Filter, dominant
K is fixed to 1000

Computational hotspot in the algorithm – Estimation is dominant
N is fixed to 2^25

Parallel sFFT on Multicore using OpenMP

K= 1000

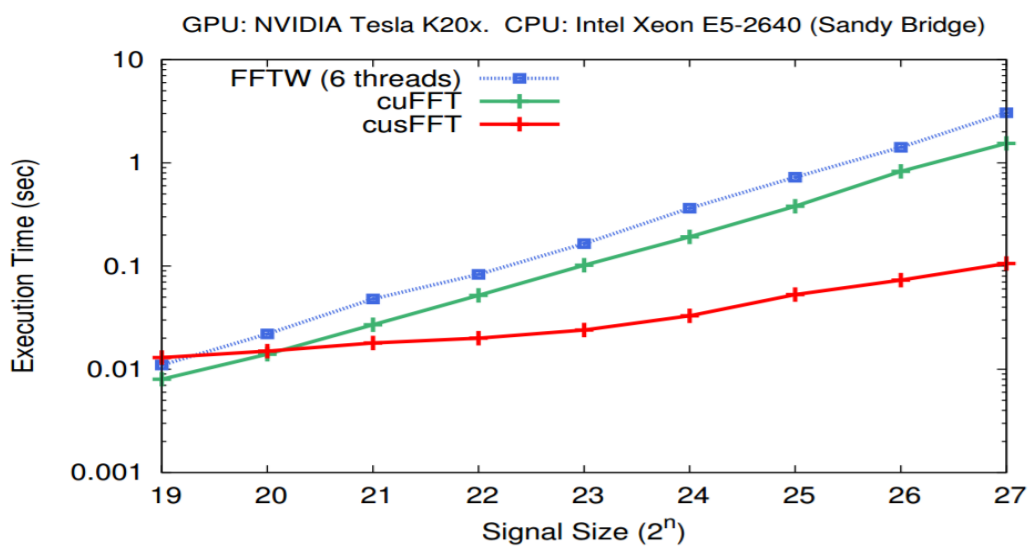


- PsFFT (6 threads) is $\sim 4-5x$ faster than the original MIT sFFT
- From, $n = 2^{22}$ onwards, PsFFT reduces execution time compared to FFTW
- PsFFT is faster than FFTW up to $9.23x$

Wang, Cheng, et al. "Parallel sparse FFT." *Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms*. ACM, 2013

cusFFT on GPUs using CUDA

K= 1000

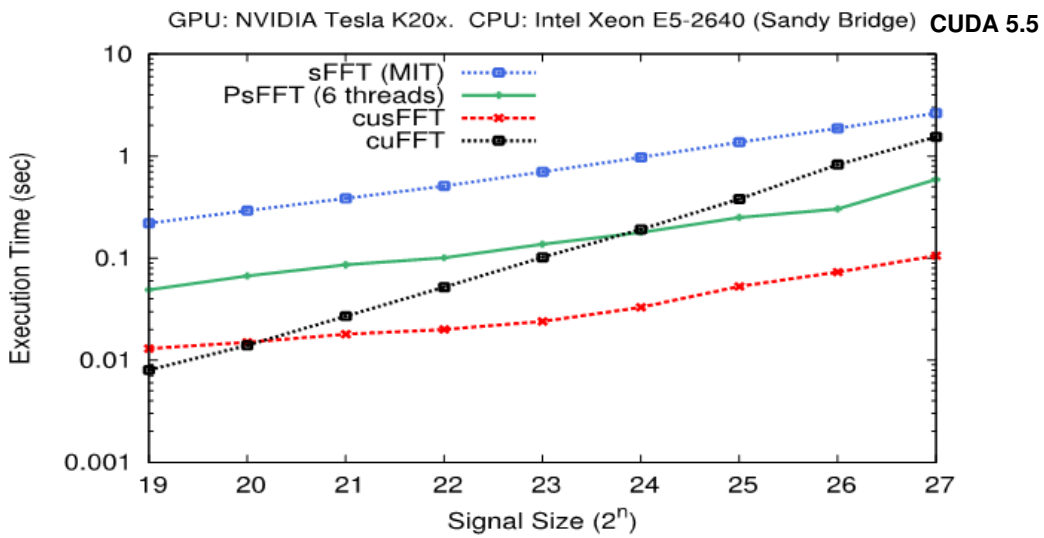


- cusFFT is $\sim 28x$ faster than parallel FFTW on multicore CPU
- $\sim 6.6x$ for n^{24} (goes down for larger signal size)

Wang, Cheng, Sunita Chandrasekaran, and Barbara Chapman. "cusFFT: A High-Performance Sparse Fast Fourier Transform Algorithm on GPUs." *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016.

cusFFT on GPUs using CUDA

K= 1000



- cusFFT is $\sim 4x$ faster than PsFFT on CPU, $\sim 25x$ vs the MIT sFFT
- cusFFT is $\sim 10x$ faster than cuFFT for large data size

Wang, Cheng, Sunita Chandrasekaran, and Barbara Chapman. "cusFFT: A High-Performance Sparse Fast Fourier Transform Algorithm on GPUs." *Parallel and Distributed Processing Symposium, 2016 IEEE International*. IEEE, 2016.

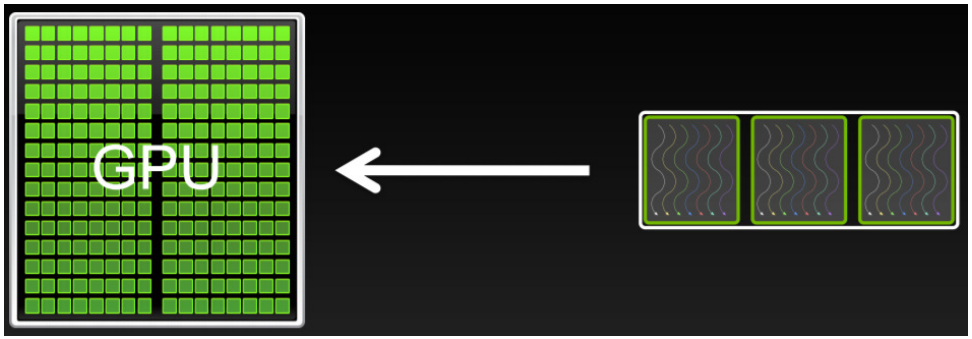
OpenACC – Parallel Programming Model

- Large user base: MD, weather, particle physics, CFD, seismic
  - Directive-based, high level, allows programmers to provide hints to the compiler to parallelize a given code
- OpenACC code is portable across a variety of platforms and evolving
  - Ratified in 2011
  - Supports X86, OpenPOWER, GPUs. Development efforts on KNL and ARM have been reported publicly
  - Mainstream compilers for Fortran, C and C++
  - Compiler support available in PGI, Cray, GCC and in research compilers OpenUH, OpenARC, Omni Compiler

```
#pragma acc kernels
for( i = 0; i < n; ++i )
    a[i] = b[i] + c[i];
```

```
#pragma acc parallel loop
for( i = 0; i < n; ++i )
    a[i] = b[i] + c[i];
```

Gang, Worker, Vector



Source: Profiling and Tuning OpenACC code, Cliff Woolley, NVIDIA

CUDA vs OpenACC (Example Saxpy Code)

```
__global__
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

...
int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, x, y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

```
void saxpy(int n, float a, float * restrict x, float * restrict y)
{
#pragma acc kernels
  for (int i = 0; i < n; ++i)
      y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
```

Source code example from: devblogs.nvidia.com/parallelforall/six-ways-saxpy/

CUDA sFFT

```
cudaMalloc((void**)&d_x,
n*sizeof(complex_t));
  cudaMemcpy(d_x, x,
n*sizeof(complex_t),cudaMemcpyHostToDevice);

  for(int i = 0; i < repetitions; i++){
    err = cufftExecZ2Z(……);
    ....
    ....
    }
  }
  cudaMemcpy(cufft_x_f, d_x_f,
n*sizeof(complex_t),
cudaMemcpyDeviceToHost);

  cudaFree(….);
```

```
 __global__ void PermFilterKernel(

cuDoubleComplex* d_origx,
cuDoubleComplex* d_filter,
int* d_permute,
cuDoubleComplex* d_x_sampt)
{
  if(i < loops*B)
    ....
    cuDoubleComplex tmp_value1,
tmp_value2;
    for(int j=0; j<round; j++){
      ....
      tmp_value1 =
cuCmul(d_origx[index],d_filter[off+j]);
      tmp_value2 = cuCadd(tmp_value1,
tmp_value2);
    }
}
```

```
 cudaMalloc((void**)&d_origx,
n*sizeof(complex_t));
  cudaMemcpy(d_origx, origx,
n*sizeof(complex_t),
cudaMemcpyHostToDevice);

….
…. //similar instructions three times
more

cudaFree(d_origx);
cudaFree(d_filter);
cudaFree(d_x_sampt);
cudaFree(d_permute);
```

OpenACC code

```
#pragma acc data copyin(d_origx[0:2*n], \
d_filter[0:2*filter_size], \
    permute[0:loops]) copyout(d_x_sampt[0:loops*B_2])
  {
    #pragma acc kernels loop gang vector(8) independent
    for (int ii=0; ii<loops; ii++){
      #pragma acc loop gang vector(64) independent
      for(int i=0; i<B; i++){
        .....
        .....
        for(int j=0; j<round_2; j+=4){
          tmp = ((unsigned)((i_2+j*B)*ai));
          index = tmp & n2_m_1;
          COMPLEX_MULT(index,off3,j);

          index = (unsigned)(tmp + B*2*ai) & n2_m_1;
          COMPLEX_MULT(index,off3,j+2);
        }
        .....
        .....
      }
      .....
      //Step B -- cuFFT of B-dimensional FFT
      #pragma acc host_data use_device(d_x_sampt)
      {
        .....
        if (err != CUFFT_SUCCESS){
          .....
          exit(-1);
        }
      }
    }/*End of ACC data region*/
```

OpenACC code

```
#pragma acc kernels

 for(int i = 0; i < my_hits_found; i++) {
  int position = 0;

  #pragma acc kernels async(1)

  for(int j = 0; j < loops; j++) {
   int permuted_index= timesmod(permute[j], hits[i], n);
   int hashed_to = permuted_index / (n / B);
   int dist = permuted_index % (n / B);
   if (dist > (n / B) / 2) {
    hashed_to = (hashed_to + 1) % B;
    dist -= n / B;
```

Experimental Setup

Software

- CUDA v5.5
- PGI v17.3 (PGI 16.10 CE)
- FFTW v3.3.6

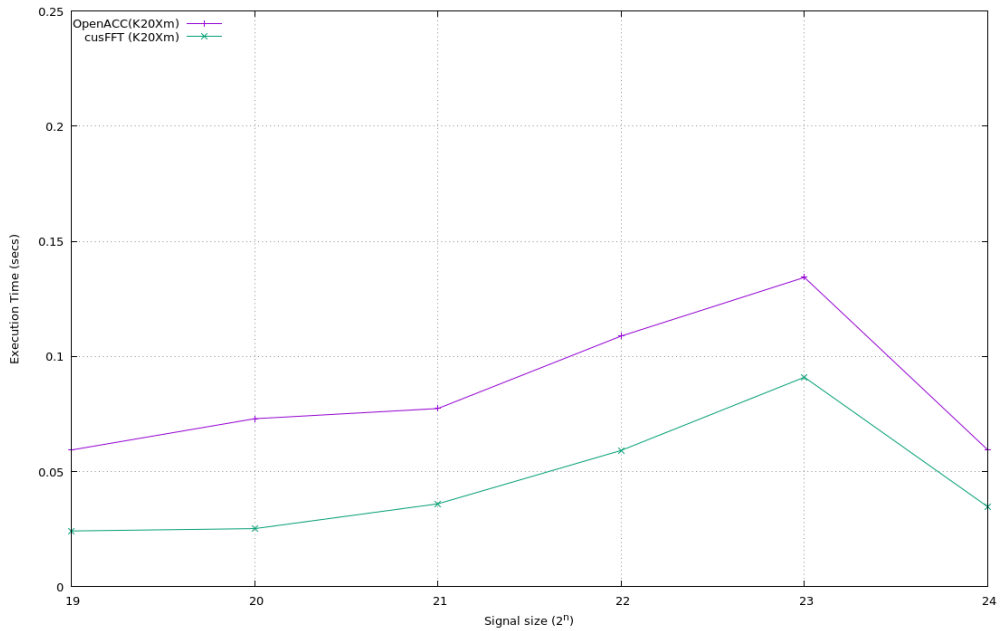
Hardware

- NVIDIA K20Xm
- Intel Xeon E5 (12 cores)

Yes, We realize we have used an older CUDA version and an older GPU card. Unfortunately we had reproducibility issues with CUDA 7 - 8.0 on K40, K80, P100 and have not been successful determining what's causing this issue. So we are limited with the experimental setup that worked OK for CUDA sFFT.

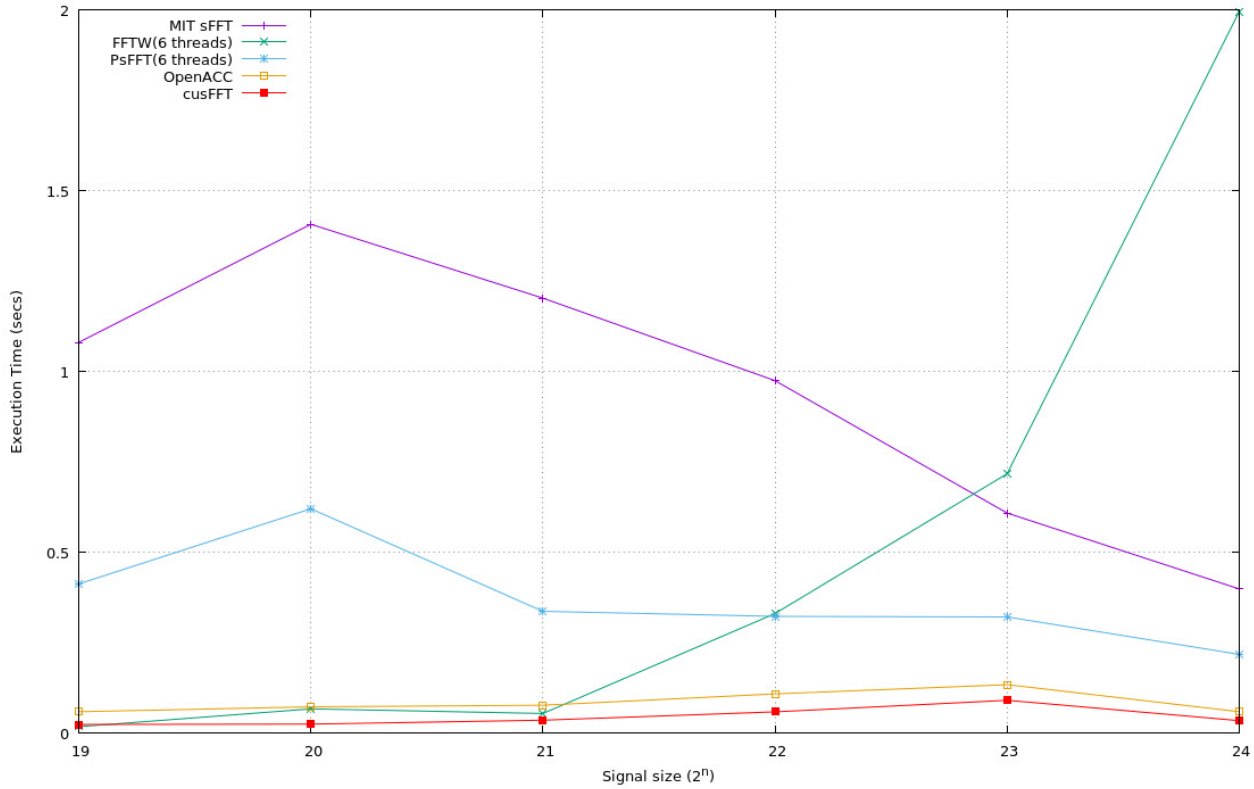
OpenACC Vs CUDA sFFT Performance

K= 1000

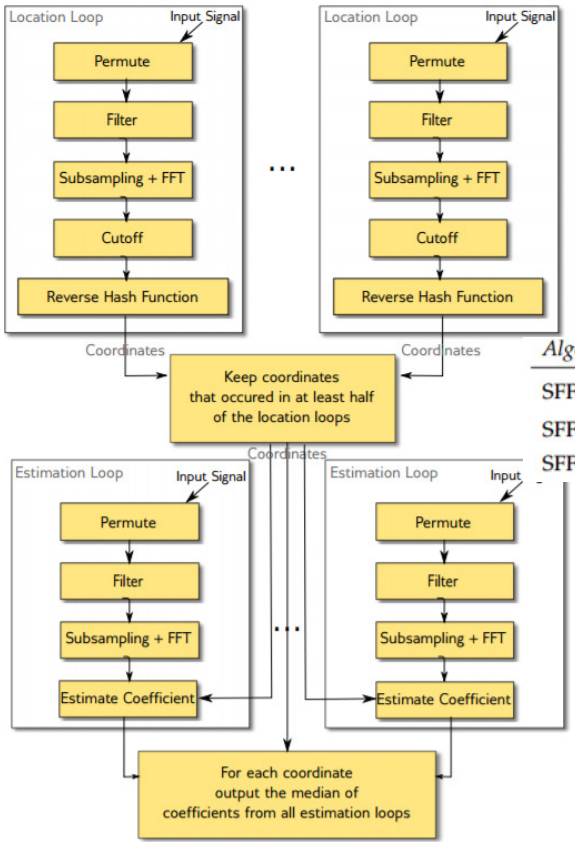


sFFT, Parallel sFFT, cusFFT, OpenACC-sFFT & FFTW

K= 1000 constant and N varied and vice versa

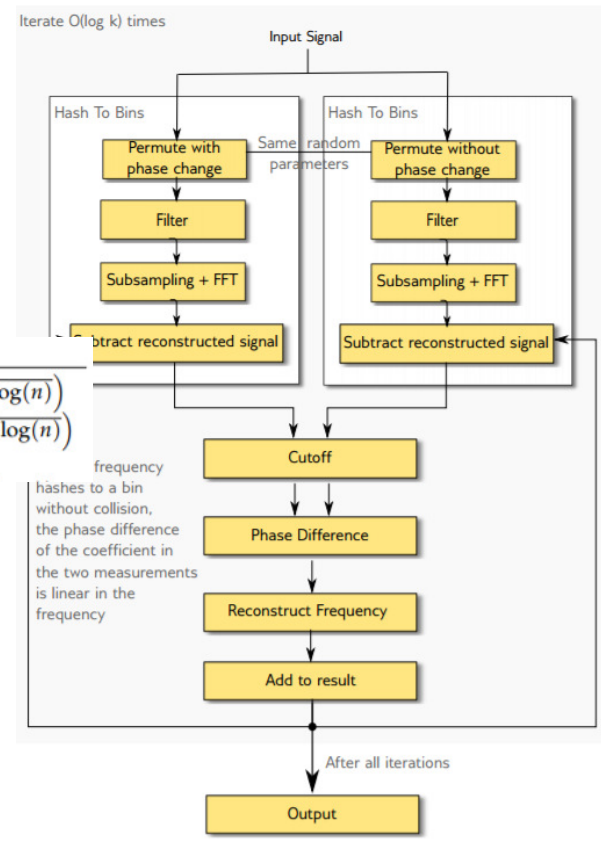


sFFT 1, 2



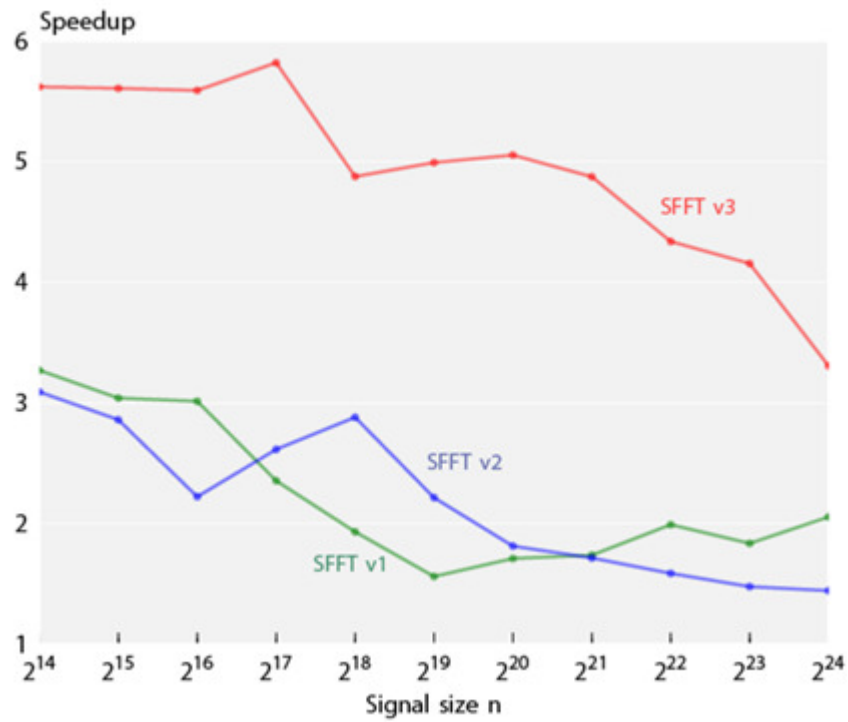
Algorithm Version	*Cost*
SFFT 1	$\mathcal{O}\left(\log n \sqrt{nk \log(n)}\right)$
SFFT 2	$\mathcal{O}\left(\log n \sqrt[3]{nk^2 \log(n)}\right)$
SFFT 3	$\mathcal{O}\left(k \log n\right)$

sFFT 3



sFFT v3.0

- Optimized sFFT serial version
  - Iteration in chunks
  - Interleaved data layout
  - Vectorization
  - Gaussian Filter, along with Mansour for better heuristics
  - Loop unroll by using fixed size HashToBins (Generally 2)
  - SSE intrinsics



Schumacher, Jorn, and Markus Puschel. "High-performance sparse fast Fourier transforms." *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*. IEEE, 2014.

Conclusion and Future Work

- Conclusions
  - Created an OpenACC sFFT codebase
    - Can be incrementally improved
    - Can be easily maintained
    - Can be executed just as a serial code (ignoring directives)
    - Can run on multicore platform as well or target other supported platforms
  - For selective cases OpenACC achieves parallelism close to CUDA
- Future Work
  - Explore parallelizing sFFT 3.0 for GPUs using OpenACC
  - Apply parallelized sFFT algorithms on real-world applications

Ack: Many thanks to Mat Colgrove, Mark Harris, Pat Brooks, Chandra Cheij, Chris Gottbrath