Parallelizing sparse FFT using programming models on state-of-the-art systems

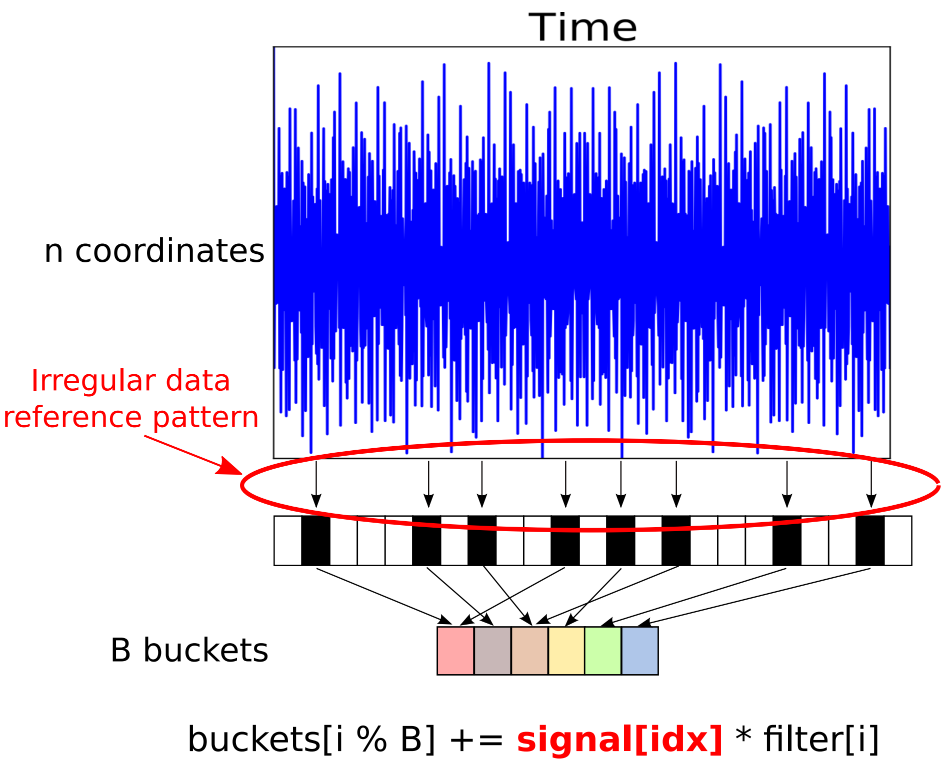
Sunita Chandrasekaran, University of Delaware
Cheng Wang, Microsoft
Arnov Sinha, Numeca
Barbara Chapman, Stony Brook University
Detlef Hohl, Mauricio Araya, Amit St. Cyr, Shell

November 15, 2017

About :

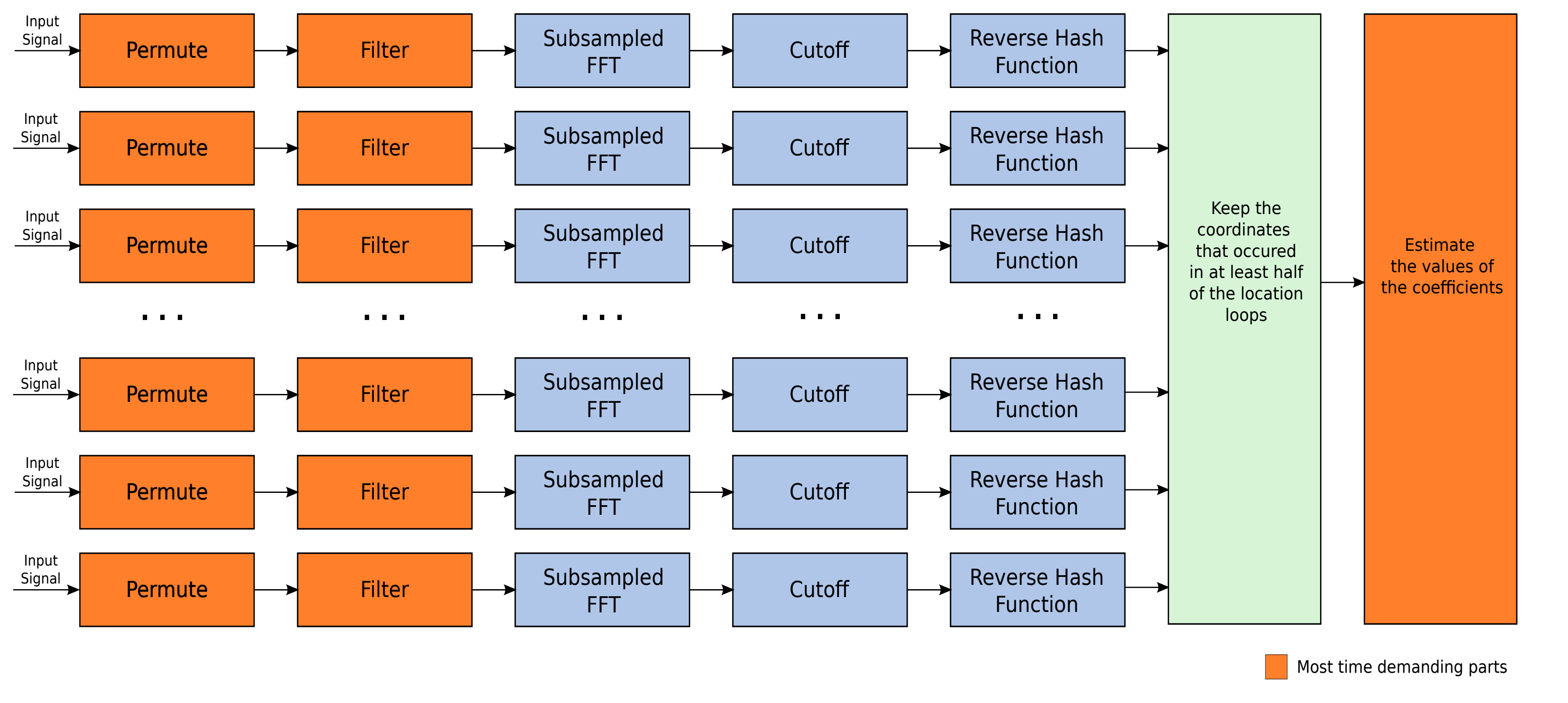
- MIT's sparse FFT 2012
- Computing FFT in a sub-linear time to efficiently locate the most significant output (very few "large" coefficients present in the frequency domain)
- Profiled & Parallelized sFFT
  - Multicore using OpenMP (~4.5x on 6 threads)
  - ARM + DSP using OpenMP
  - GPUs using CUDA ($\sim 25x$ vs the MIT sFFT, ~10x faster than cuFFT for large data size)
  - GPUs using OpenACC (performance close to CUDA)
- Dynamic irregular memory access patterns makes parallelization most challenging
- A runtime transformation algorithm to exploit temporal and spatial locality

Irregular memory access pattern



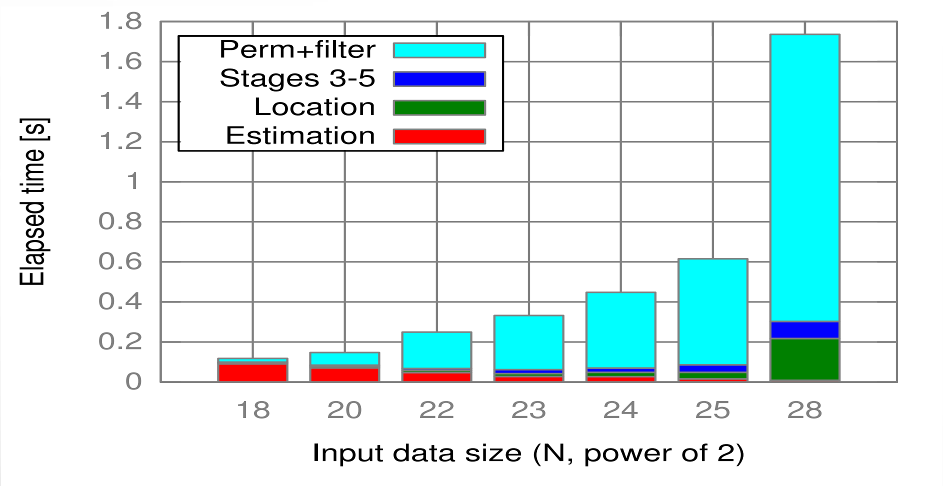
- Randomly permutes the signal spectrum and bins into a small number of buckets
- Irregular memory access pattern

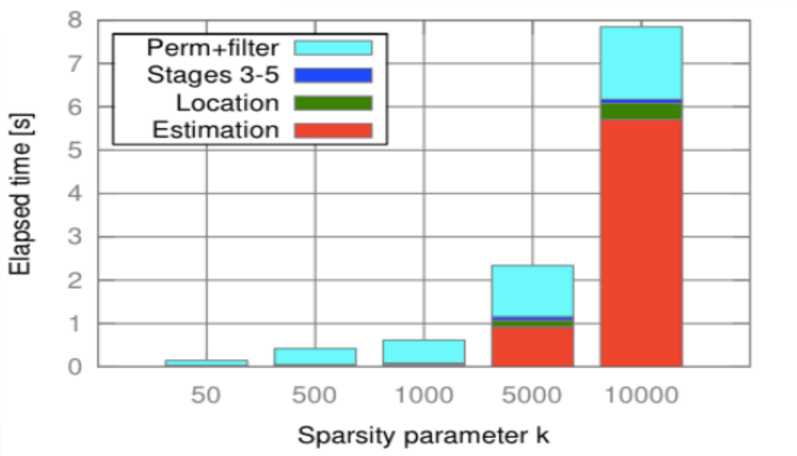
sFFT stages



Input Signal
Permute
Filter
Subsampled FFT
Cutoff
Reverse Hash Function
Input Signal
Permute
Filter
Subsampled FFT
Cutoff
Reverse Hash Function
Input Signal
Permute
Filter
Subsampled FFT
Cutoff
Reverse Hash Function
Input Signal
Permute
Filter
Subsampled FFT
Cutoff
Reverse Hash Function
Input Signal
Permute
Filter
Subsampled FFT
Cutoff
Reverse Hash Function
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

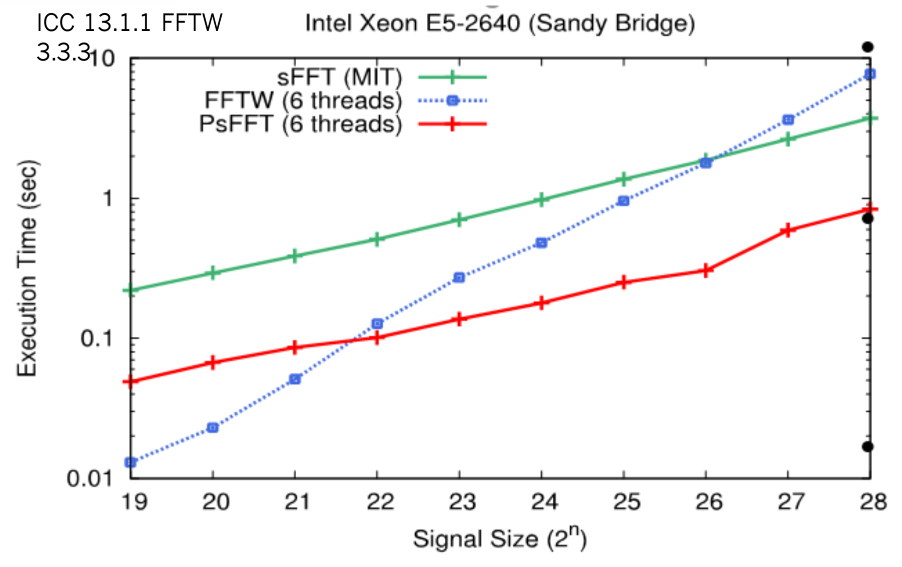




Computational hotspot in the algorithm – Permutation + Filter, dominant
K is fixed to 1000

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

Using OpenMP to parallelize sFFT on Multicore

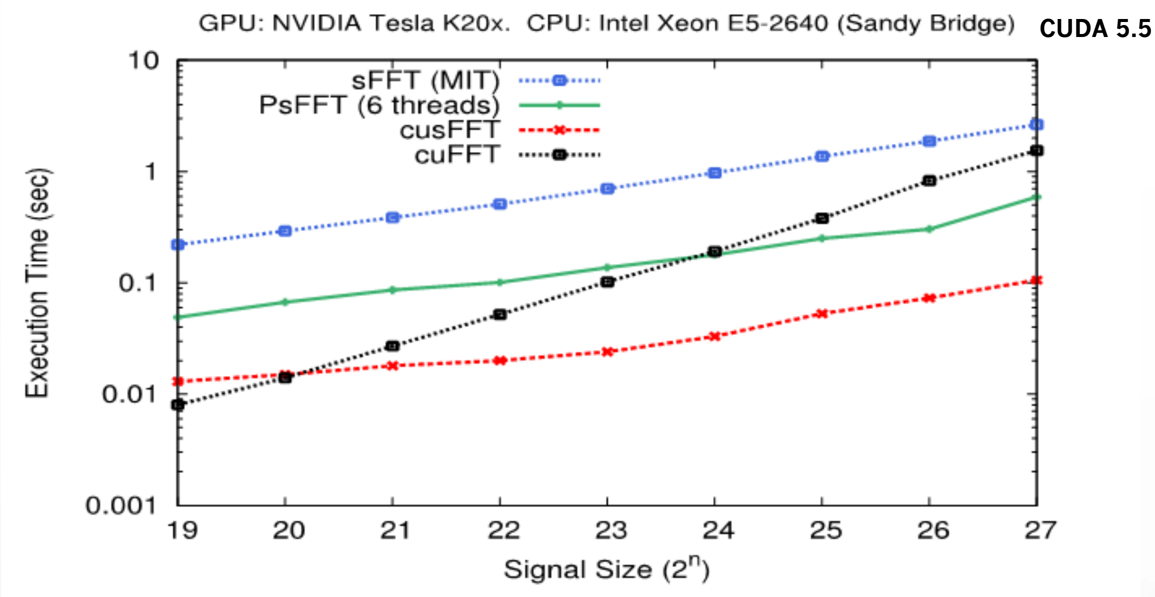


K= 1000

- PsFFT (6 threads) is ~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

Using CUDA to paralellize cusFFT on GPUs

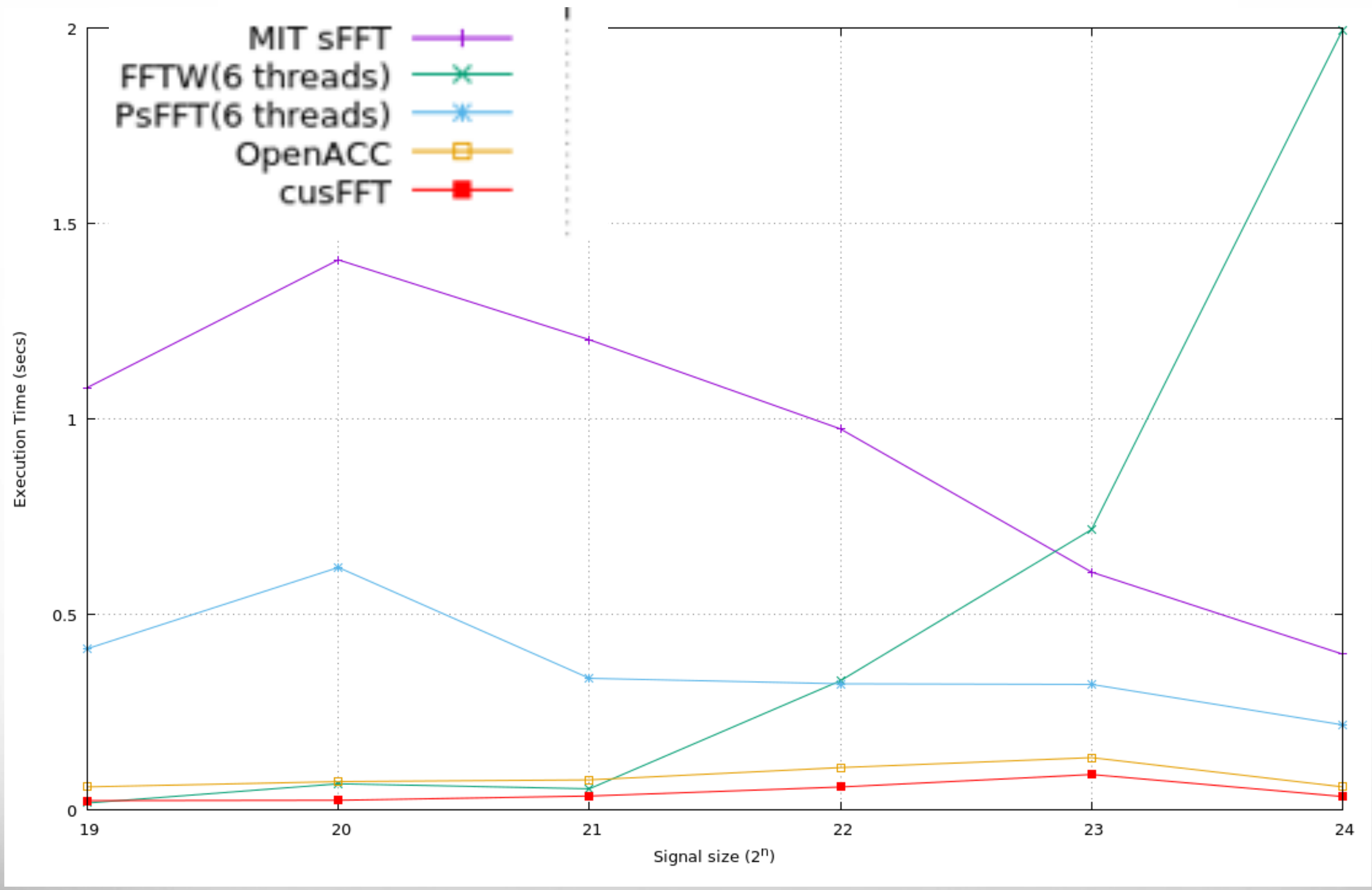


K= 1000

- 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.

Using OpenACC – Parallel sFFT, cusFFT, sFFT & FFTW

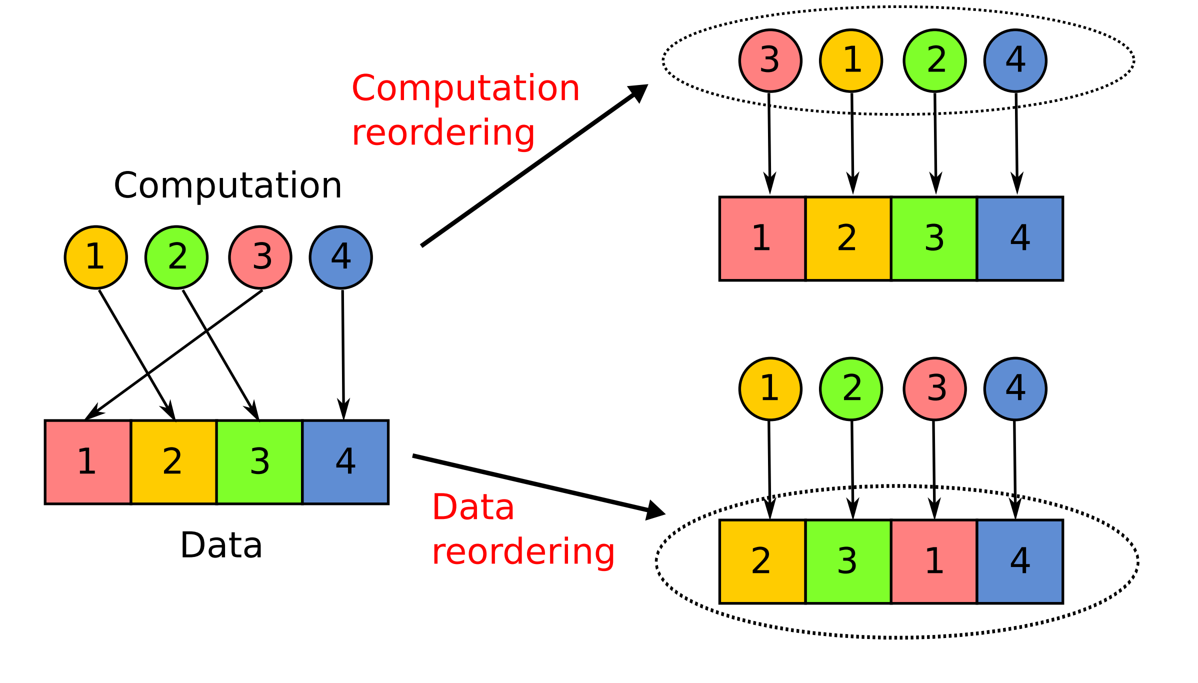


K= 1000 constant and N varied and vice versa

Presented at GTC 2017

Computation and Data Reordering

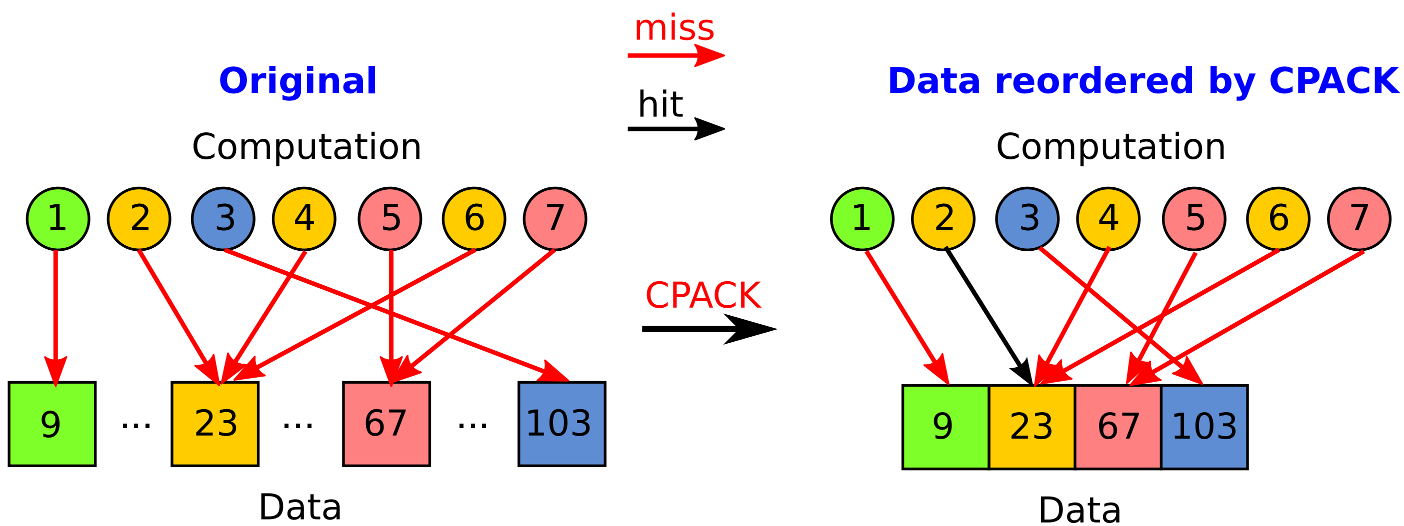
A runtime transformation algorithm to exploit temporal and spatial locality



Wang, Cheng, Sunita Chandrasekaran, and Barbara Chapman. " An Efficient Data Layout Transformation Algorithm for Locality-Aware Parallel Sparse FFT." *IA3, Workshop at SC17 <to be published>*

CPACK algorithm

CPACK: A greedy algorithm which packs data into consecutive locations in the order they are first accessed by the computation



Data access order: 9, 23, 103, **23, 67, 23, 67**

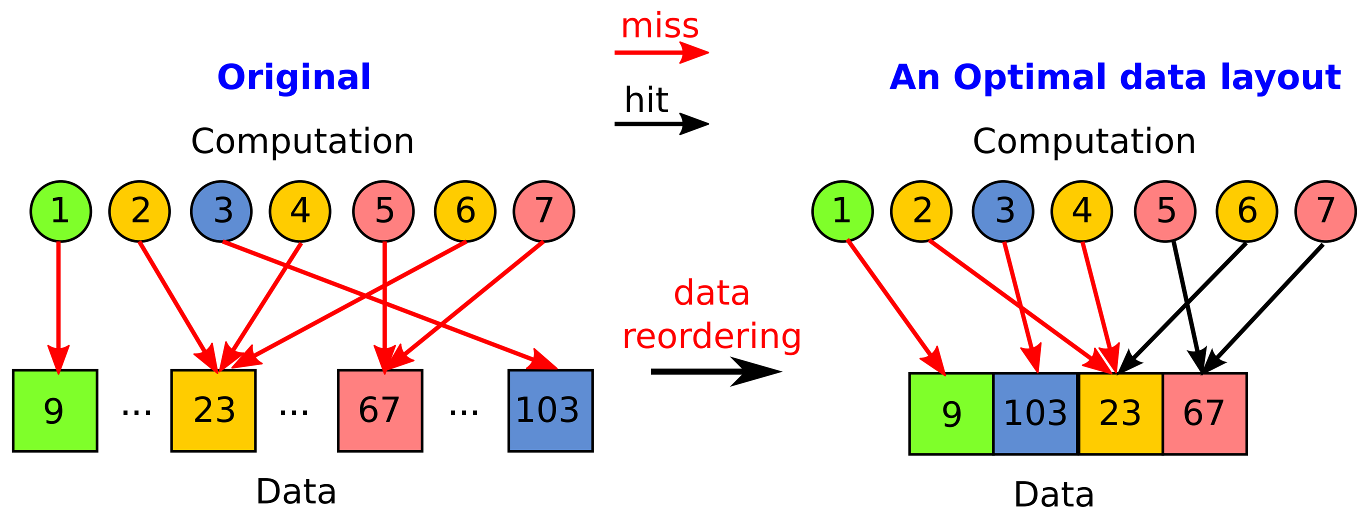
7 cache misses

6 cache miss

- First-touch policy packs (9,23) together
- Not optimal

Rethinking CPACK algorithm

Affinity-conscious data reordering ...



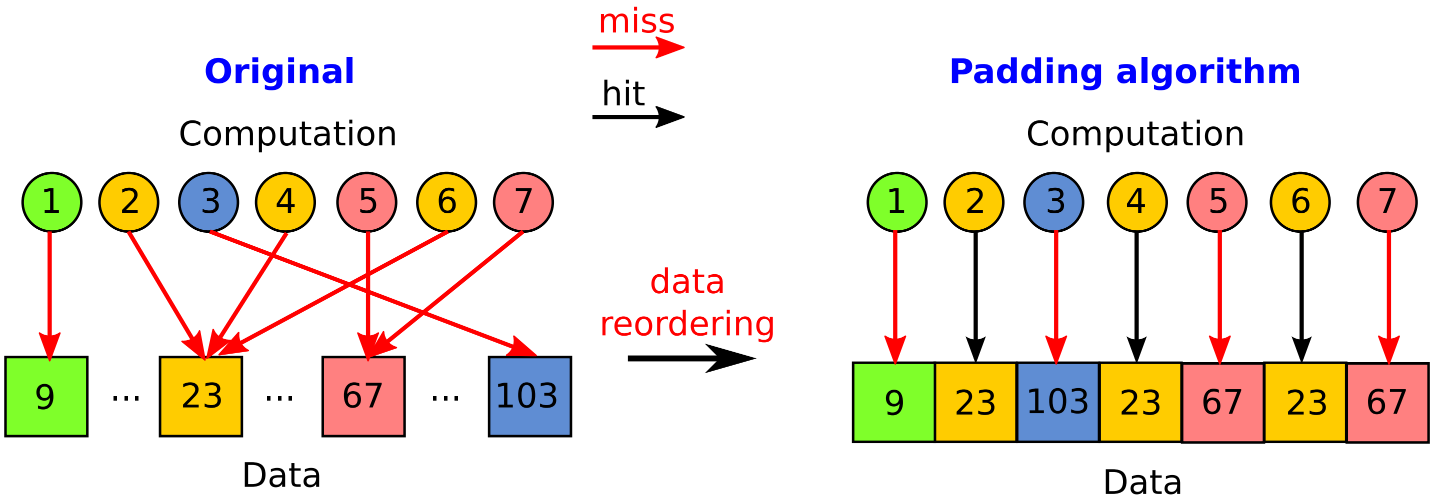
Data access order: 9, 23, 103, 23, 67, 23, 67
7 cache misses

4 cache miss

- CPACK does not consider data affinity (i.e., how close the nearby data elements are accessed together)
- Packs (23,67) rather than (9,23) should yield better locality

CPACKE Algorithm

CPACKE Algorithm: Extends the CPACK by creating duplicated copies of each repeatedly accessed data entry

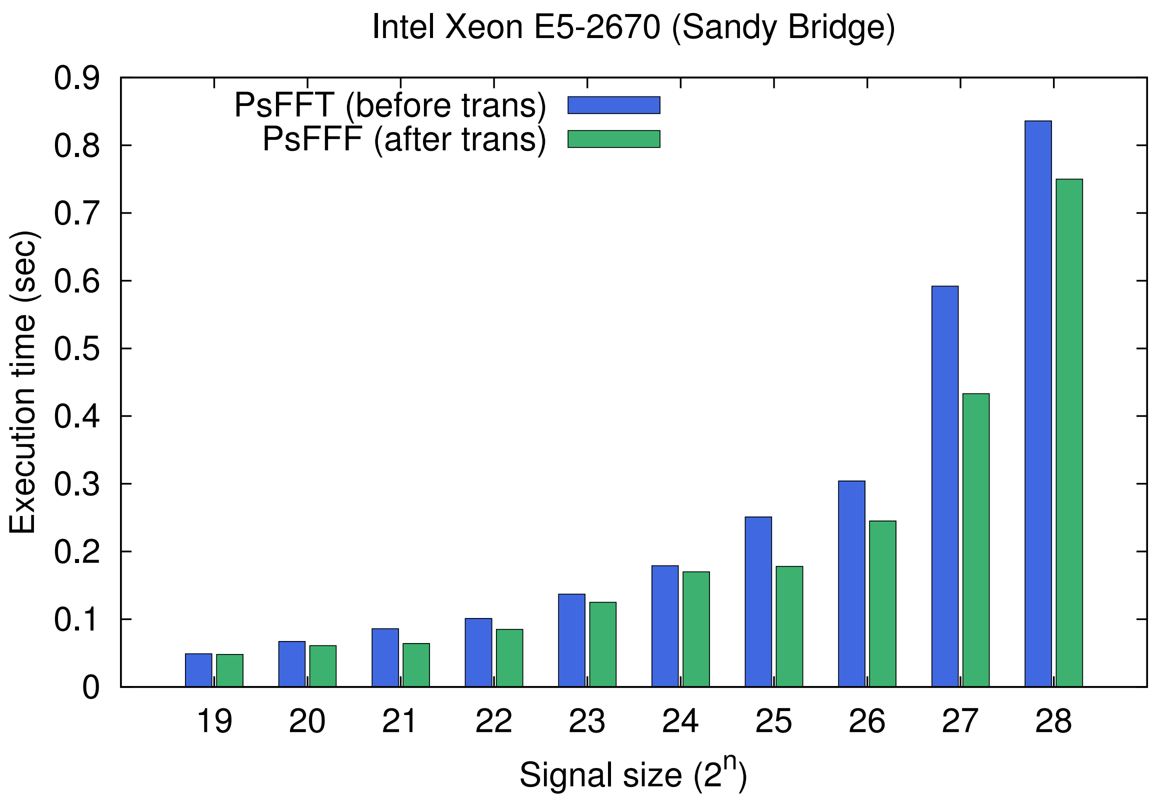


Data access order: 9, 23, 103, 23, 67, 23, 67
7 cache misses

4 cache miss

- **Advantage**: Better locality than CPACK
- **Disadvantage**: Slight space overhead

Performance Evaluation



- Applies the CPACKE to the *perm+filter* stage in sFFT
- Improves the performance by 30% for the irregular kernel
- Improves the overall performance of PsFFT by 20%

Questions?

1) Why did you write your own FFT?

2) What considerations are important for you in an FFT implementation?

3) What might you look for if there were to be a unified FFT interface (similar to BLAS, LAPACK and SCALAPACK interfaces)?

4) How important are performance, portability, and scalability for you?

5) Will FFT be needed in exascale computing and if so how will it be achieved?

6) What would be a good FFT benchmark or a good way to include the FFT in a high-performance computer benchmark?