Parallel FFT Program Optimizations on Heterogeneous Computers

Shuo Chen, Xiaoming Li

Department of Electrical and Computer Engineering
University of Delaware, Newark, DE 19716

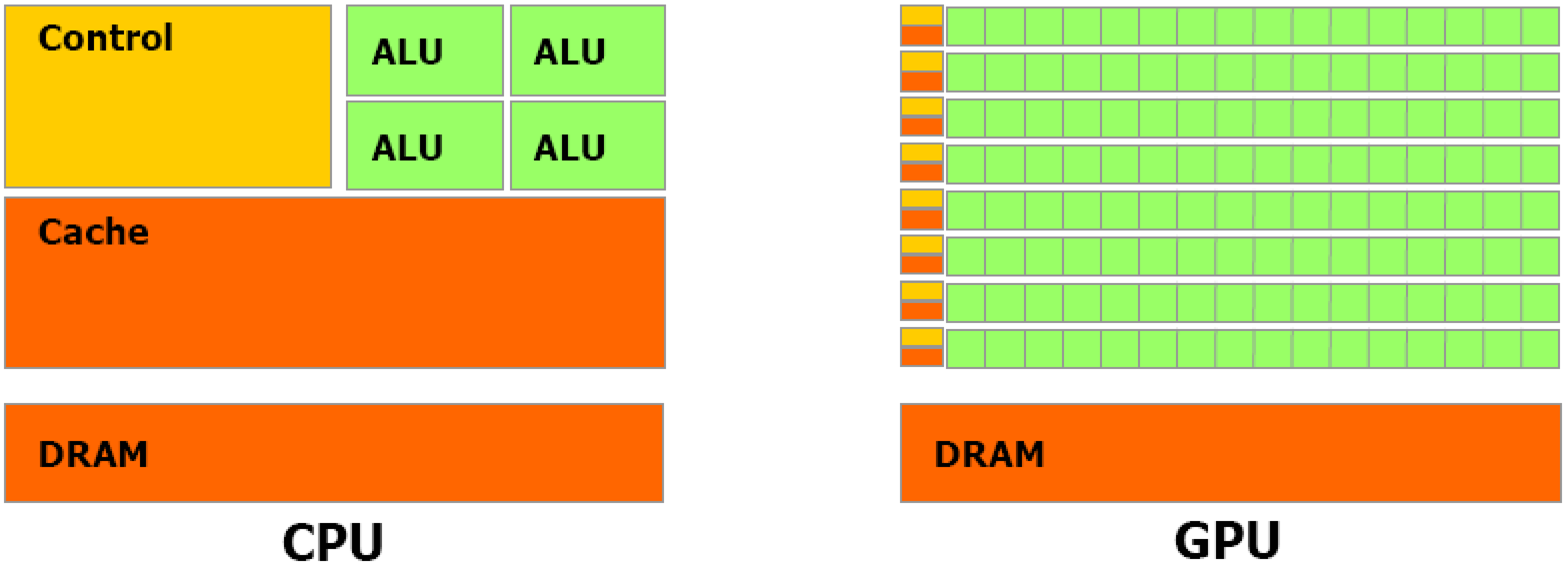
Outline

- *Part I:* A Hybrid GPU/CPU Parallel FFT Library for Large FFT Programs
- *Part II:* An Input Adaptive Algorithm for Parallel Sparse Fast Fourier Transform

Part I: A Hybrid GPU/CPU Parallel FFT Library for Large FFT Programs

Heterogeneous High Performance CPU and GPU System

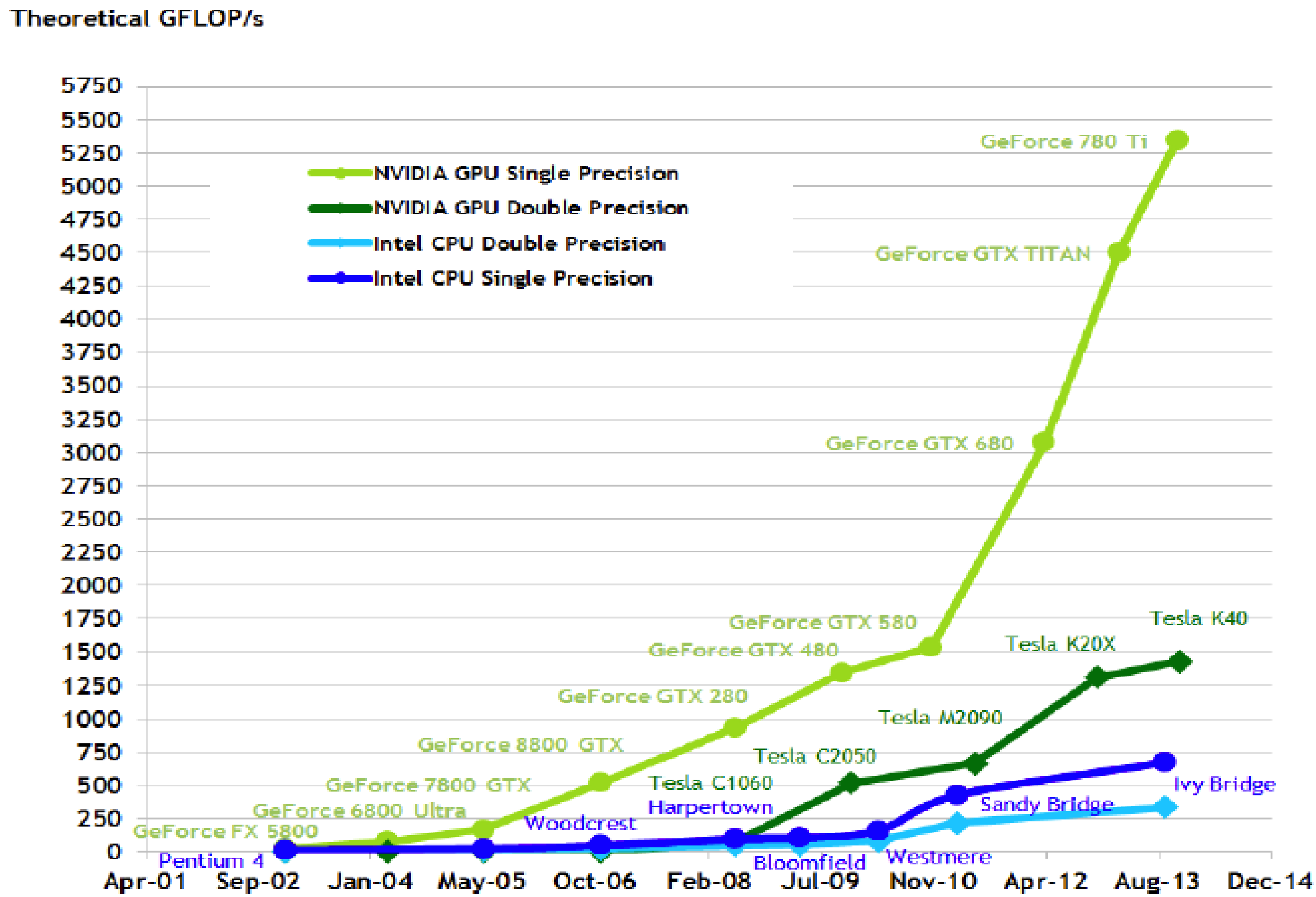
- General Purpose CPU
- GPGPU
  - General-purpose computing on graphics processing units (GPGPU).
  - GPU becomes a highly parallel, multithreaded, many-core processor with tremendous computational power and very high memory bandwidth.



The GPU devotes more transistors to data processing

(Images from the NVIDIA CUDA C Programming Guide 6.5)

CUDA Background



Floating-Point Operations per Second for the CPU and GPU

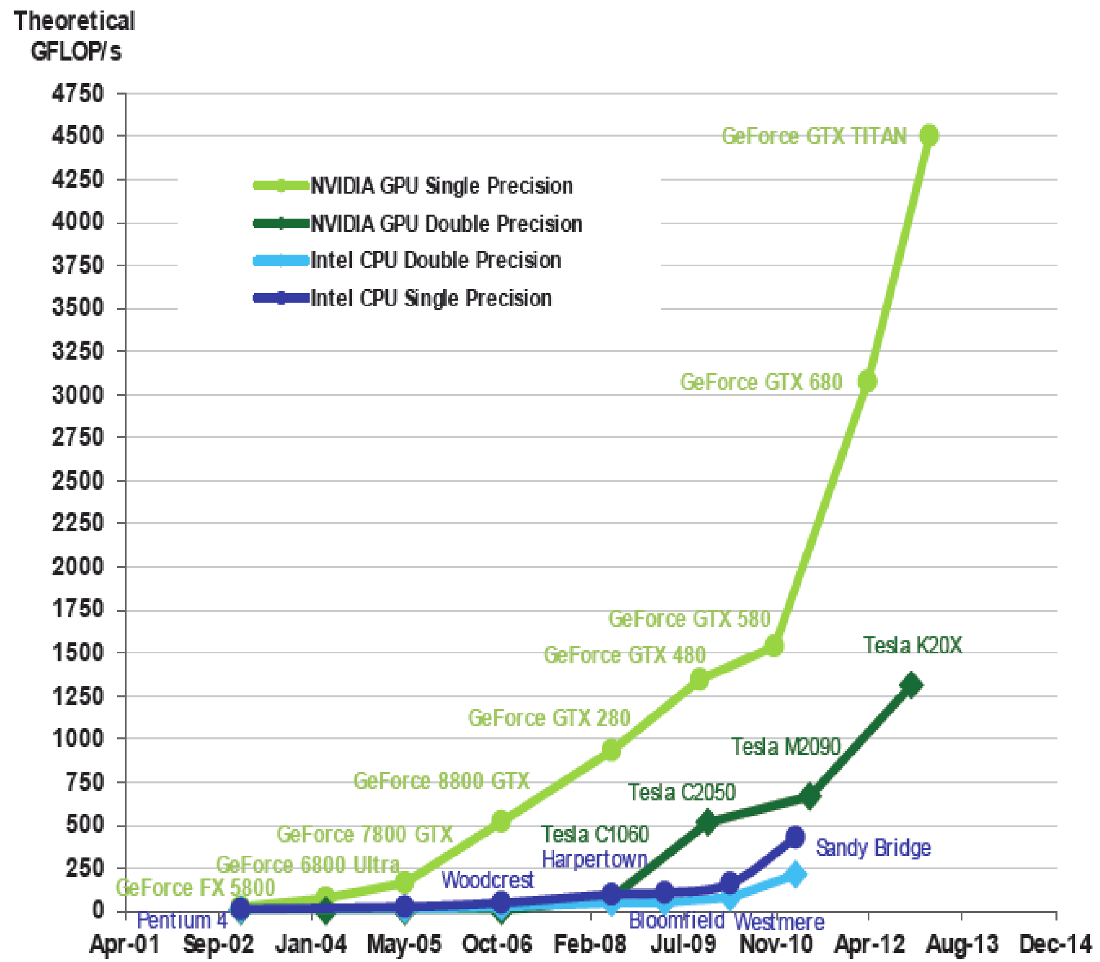
(Images from the NVIDIA CUDA C Programming Guide 6.5)

CUDA Background

– Compute Unified Device Architecture → a parallel programming model created by NVIDIA.

– Higher FLOPS on GPU than CPU. Higher memory bandwidth on GPU than traditional processor's memory.

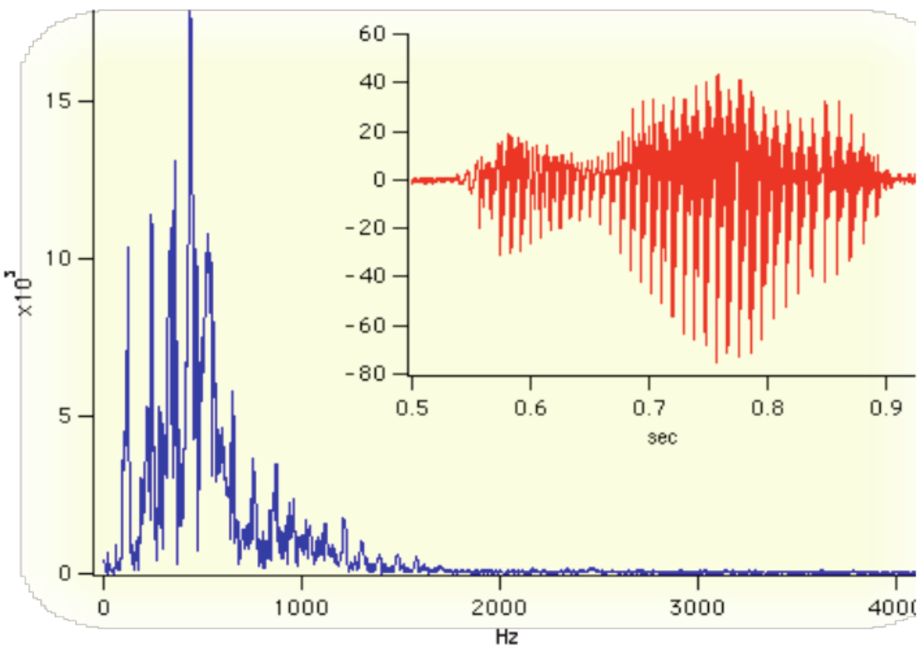
– GPUs support thousands of threads running at the same time.



Floating-Point Operations per Second for the CPU and GPU

(Images from the NVIDIA CUDA C Programming Guide 6.5)

Background of DFT/FFT



Sampled Audio Data (Time)

DFT of Audio Samples (Frequency)

- *Discrete Fourier Transform (DFT)*
  - Given $x \in \mathbb{C}^n$, compute its Fourier transform $\hat{x}$:

$$\hat{x}_d = \sum_i x_i \, \omega^{id} \quad \text{for} \quad \omega^{id} = e^{-j2\pi id/N}$$

- One of the most widely used and expensive computation in science and engineering domains:
  - large-scale physics simulations
  - signal processing
  - data compression

Background of DFT/FFT

- *Fast Fourier transform (FFT)* reduces DFT's complexity from O(N^2) into O($N\log N$).
  - Requires large amount of **computing resources and memory bandwidth**.
  - GPUs is proved to be a more promising platform than CPU.
    - much more parallel computing resources.
    - achieve an order of magnitude performance improvement over CPUs on compute-intensive applications.

Motivation

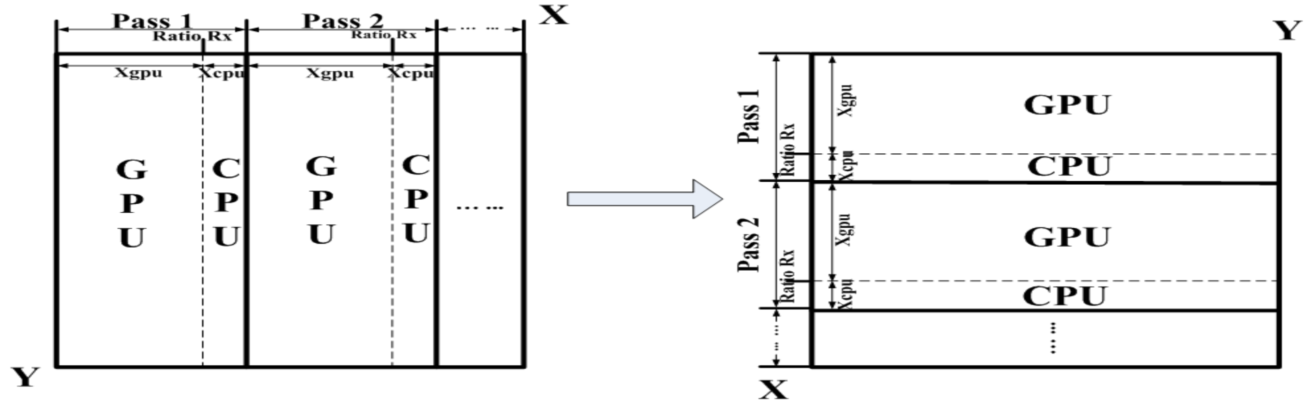
- Previous works
  - Prior FFT works on GPU use only GPU to compute but employ CPU as a mere memory-transfer controller.
    - **In-Card FFT →** CUFFT by Nvidia, Nukada's work, Govindaraju's and Gu's on 2D/3D FFT.
    - **Out-of-Card FFT →** Gu's GPU-based FFT library. Co-optimization for communication and computation.
    - **Distributed FFT →** Chen presented a GPU cluster based FFT implementation.
  - The computing power of CPU is wasted.
  - The GPU performance is restricted by the limited memory size and the low bandwidth of data transfer through PCIe channel.
- Hybridize Concurrent CPU and GPU
  - A hybrid FFT library is proposed to engage both CPU and GPU for parallel FFT.

Motivation

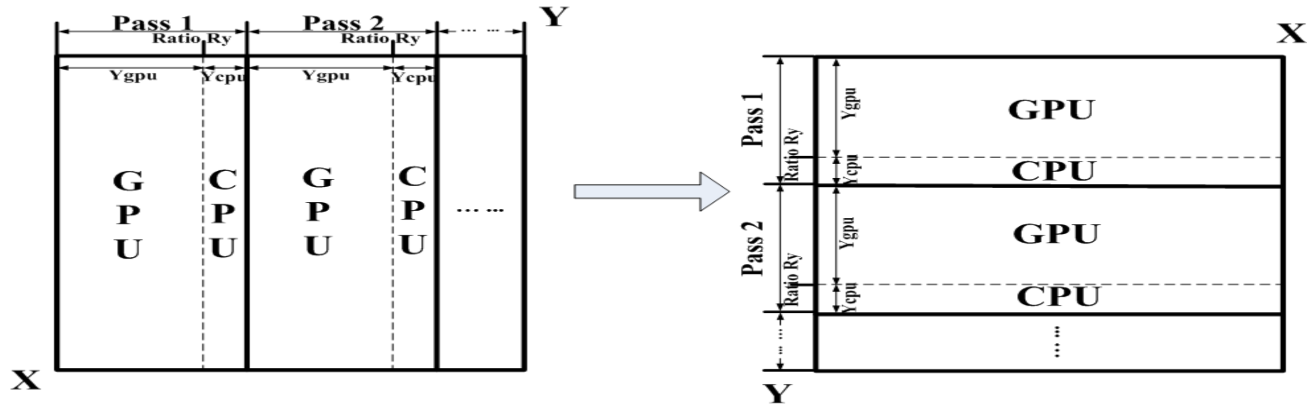
- Challenges
  - We have to handle the low bandwidth channel for data transfer between CPU and GPU?
  - How to solve the locality issues when work is distributed into heterogeneous devices?
  - How to efficiently split the workloads, and how to achieve the workloads balancing between two types of computing devices?
  - Whether or not the computations and communications can be efficiently overlapped ?

Hybrid 2D FFT Library

- Hybrid out-of-card 2D FFT Library on Heterogeneous CPU-GPU system



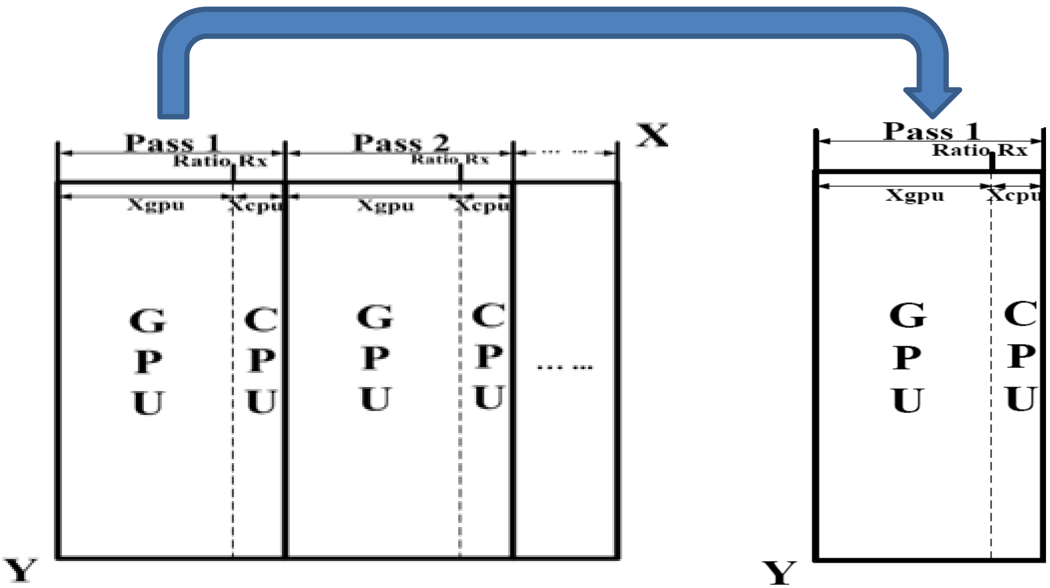
(a) Computation of the first round of 2D FFT



(b) Computation of the second round of 2D FFT

Hybrid 2D FFT Library

- A hybrid large-scale FFT decomposition framework
  - For each pass of 1st-round 2D FFT fitting into GPU memory

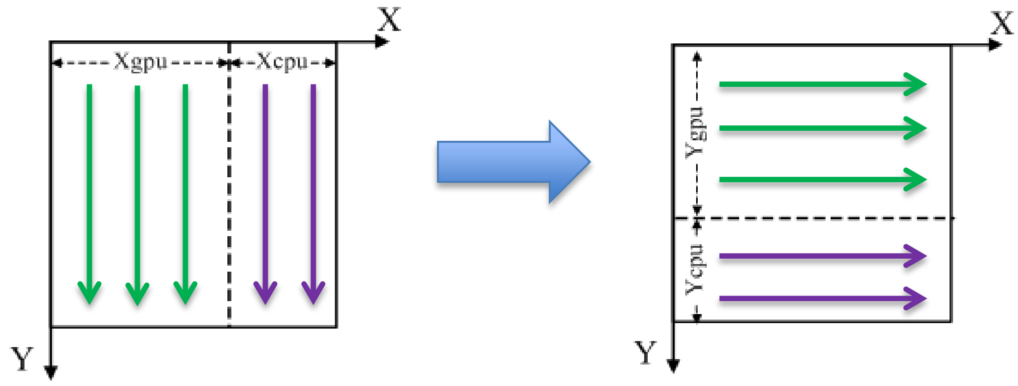


Whether we can further decompose Y dimensional 1D FFT and exploit more parallelism that can make full use of parallel computing resources?

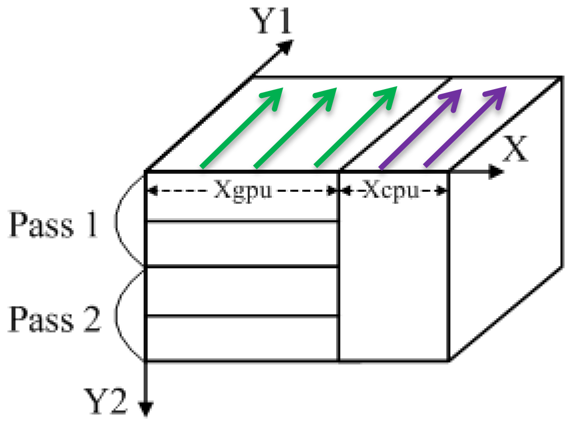
  - Data parallelism and concurrency is exploited along X dimension for both GPU and CPU.
  - However, there is still restriction to performance if size of computational dimension Y is large.

Hybrid 2D FFT Library

- A hybrid large-scale FFT decomposition framework
  - *Two rounds of computation & load distribution*

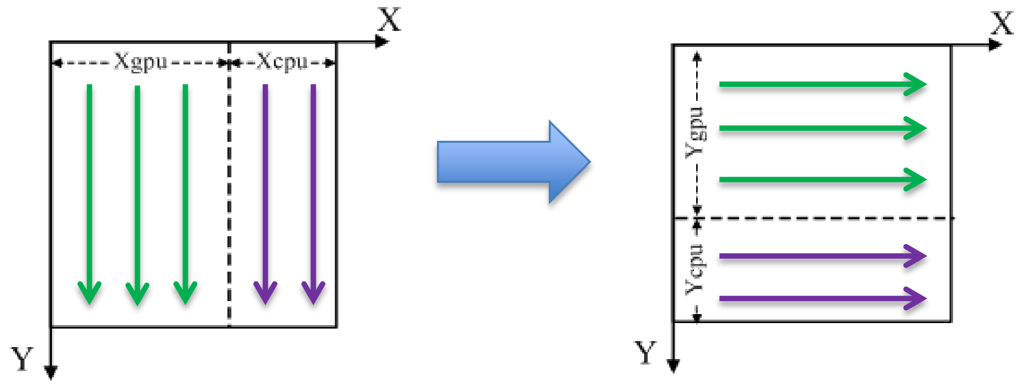


  - *Y-dimensional decomposition & load distribution*

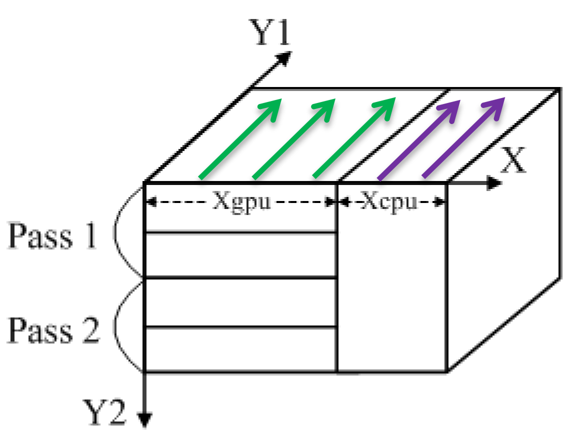


Hybrid 2D FFT Library

- A hybrid large-scale FFT decomposition framework
  - *Two rounds of computation & load distribution*

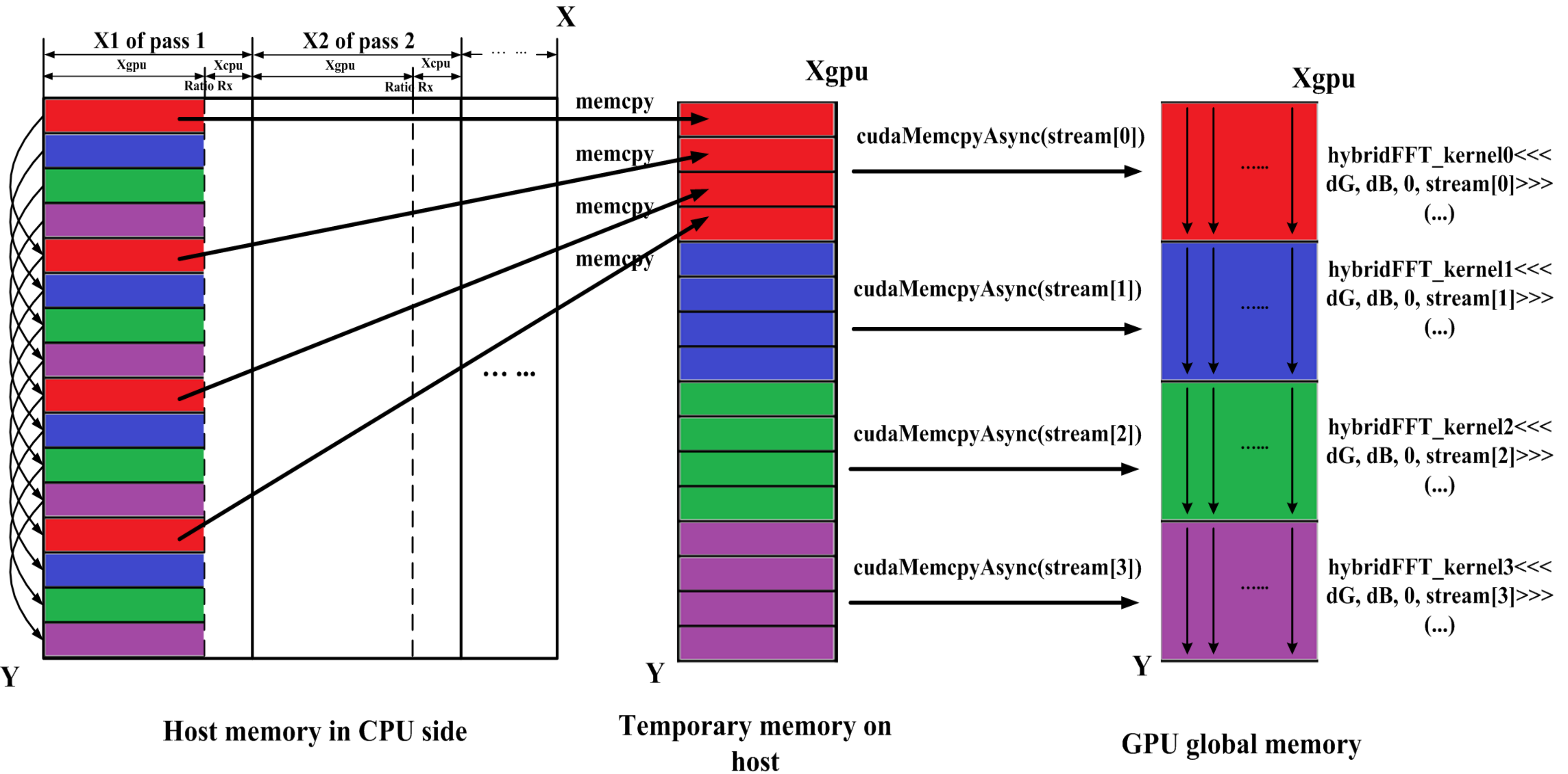


  - *Y-dimensional decomposition & load distribution*

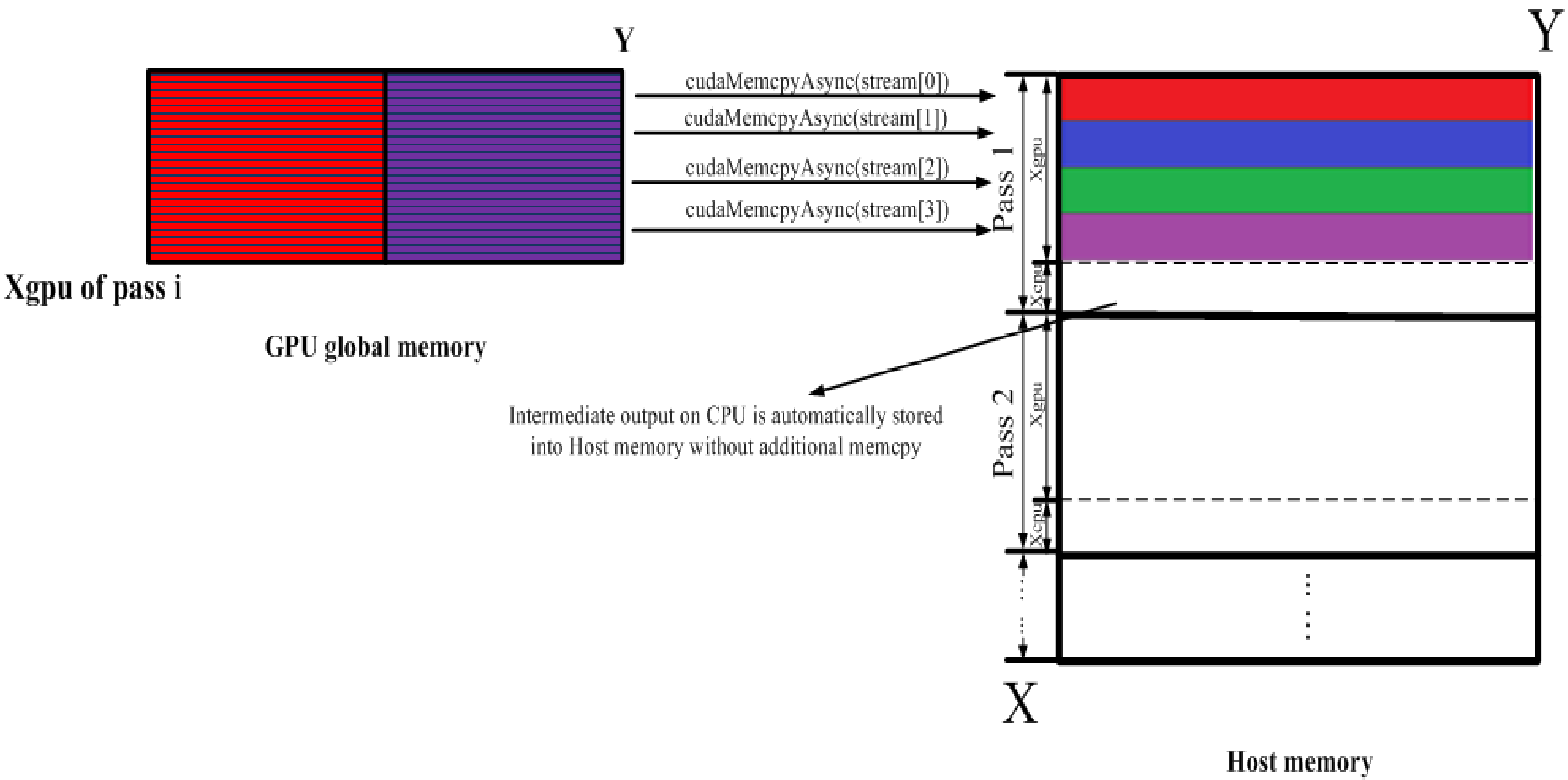


$$
\begin{aligned}
u_{gpu} =& \{d(Y_1, Y_2 Xgpu, Xgpu, Igpu, Ogpu), Sync,\\
& t^{Y_1}_{Y_2} d(Y_2, Y_1 X, Y_1 X, O, O), d(X, 1, 1, O, O)\}\\
u_{cpu} =& \{d(Y_1, Y_2 Xcpu, Xcpu, Icpu, Ocpu), Sync\}
\end{aligned}
$$

- Data Transfer Scheme Through PCIe Channel
  - *Asynchronous strided memory copy via PCIe bus*

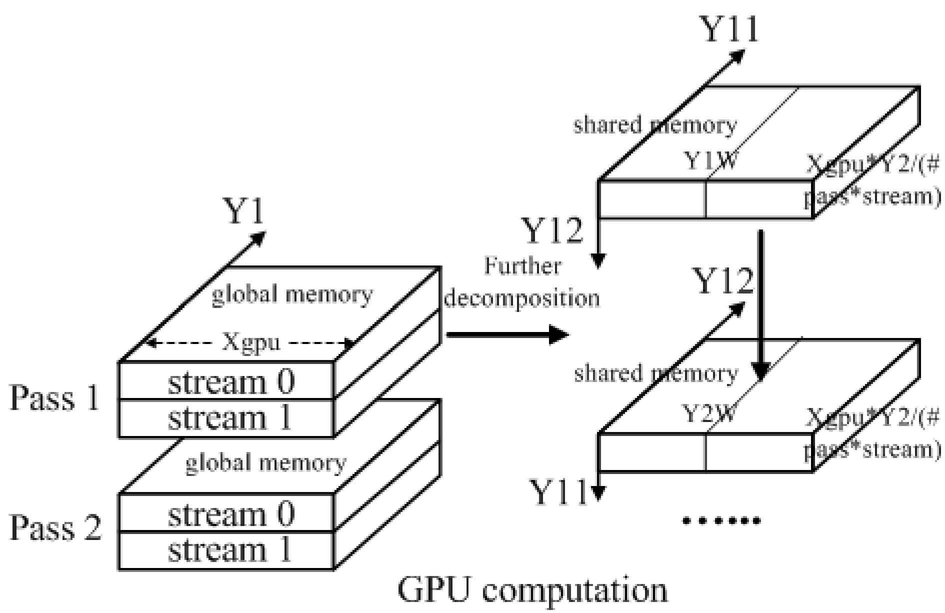


- Data Transfer Scheme Through PCIe Channel
  - *Asynchronous strided memory copy via PCIe bus*



Hybrid 2D FFT Library

- *GPU Computation & Optimization*
  - *Out-of-card FFT → divided into several passes*
  - *Asynchronous strided memory copy via PCIe bus*
  - *Stream-based asynchronous execution*
  - Shared memory increases device memory bandwidth



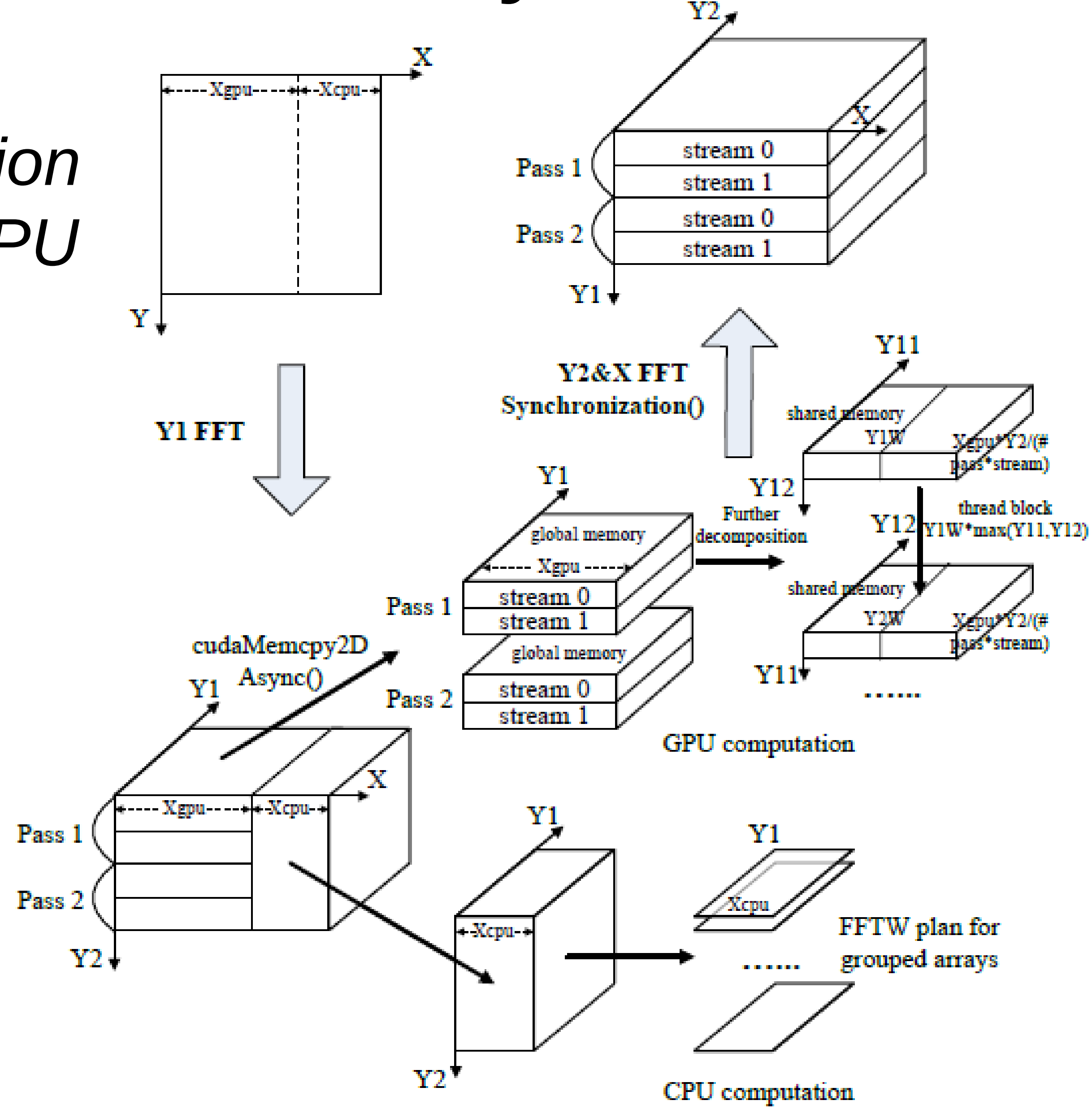
Device Memory Shared Memory

Hybrid 2D FFT Library

- *CPU Computation & Optimization*
  - *Concurrent group operations*
    - → *Each time to operate a group of data*
    - → *Operate on non-contiguous (strided) data*
  - *No input/output transposition performed*
    - → *Pre-set the input/output access stride*
    - → *Save much execution time*
  - *Multi-threaded execution to parallelize the recursive sections*

Hybrid 2D FFT Library

Co-Optimization of CPUs & GPU



Load Balancing between GPU and CPU

- *Performance Modeling*
  - Split the total execution into several ***primitive*** sub-steps to derive a performance model parameter for each primitive.

Load Balancing between GPU and CPU

- *Performance Modeling*
  - Split the total execution into several ***primitive*** sub-steps to derive a performance model parameter for each primitive.
    - ***Model parameter*** provides estimated execution time parameterized with load ratio.

Load Balancing between GPU and CPU

- *Performance Modeling*
  - Split the total execution into several ***primitive*** sub-steps to derive a performance model for each primitive.
    - ***Model parameter*** provides estimated execution time parameterized with load ratio.
    - ***Two profiling runs***, one on CPU and one on GPU, to determine parameter values of different load ratios.

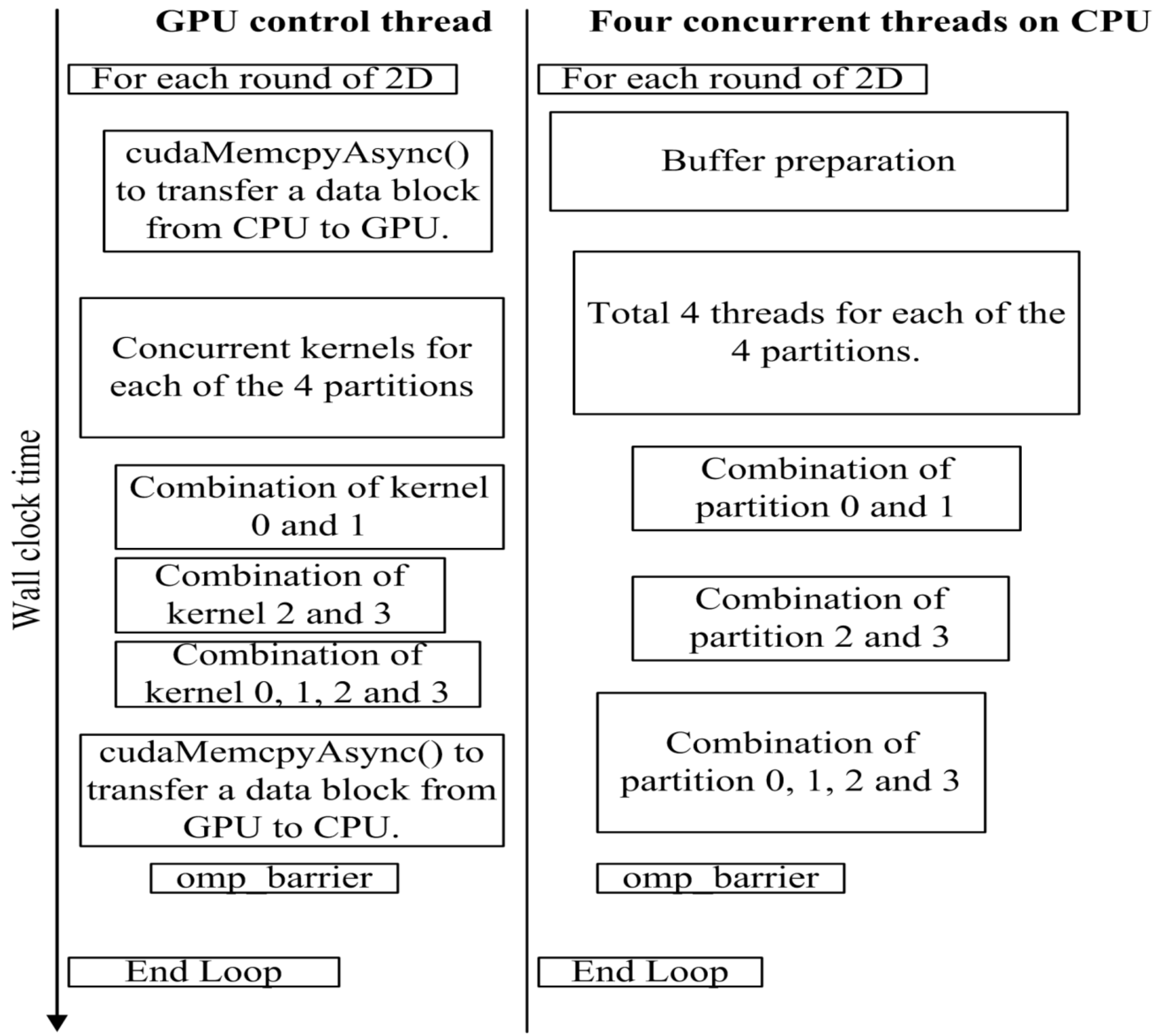
Load Balancing between GPU and CPU

- *Performance Modeling*
  - Split the total execution into several ***primitive*** sub-steps to derive a performance model for each primitive.
    - ***Model parameter*** provides estimated execution time parameterized with load ratio.
    - ***Two profiling runs***, one on CPU and one on GPU, to determine parameter values of different load ratios.
    - ***Automatically estimate***, rather than really measuring, the total execution time of our implementation under varying ratios.

Load Balancing between GPU and CPU

- *Performance Modeling*
  - Split the total execution into several ***primitive*** sub-steps to derive a performance model for each primitive.
    - ***Model parameter*** provides estimated execution time parameterized with load ratio.
    - ***Two profiling runs***, one on CPU and one on GPU, to determine parameter values of different load ratios.
    - ***Automatically estimate***, rather than really measuring, the total execution time of our implementation under varying ratios.
  - *Performance Modeling and Tuning*
    - Performance estimation from model parameters.
    - Accuracy is evaluated → only use it to provide a small region of potentially good choices.

Load Balancing between GPU and CPU



Load Balancing between GPU and CPU

Parameters for 2D FFT Running Time Estimation

Parameters	Description
# passes	Total # of passes. Subproblem of each pass fits into GPU memory.
# streams	Total # of streams that support for asynchronous kernel executions and transfers.
# thds	# of threads of CPU.
$T_{2d\text{H2D}}(i, R_g)$	$= T_{2d\text{H2D-gpu}} \times R_g$. Time of copying a 2D strided array of size $\frac{R_g \times X \times Y_2}{\#\,\text{passes} \times \#\,\text{streams}}$ from host to device in stream i.
$T_{Y_1\,\text{kernel}}(i, R_g)$	$= T_{Y_1\,\text{kernel-gpu}} \times R_g$. Time of Y_1-step FFTs computation of concurrent kernel in stream i. Thread block size is $Y_1 W \times max(Y_{11}, Y_{12})$, grid size is $\frac{R_g \times X \times Y_2}{\#\,\text{passes} \times \#\,\text{streams}}$.
$T_{2d\text{D2H}}(i, R_g)$	$= T_{2d\text{D2H-gpu}} \times R_g$. Time of copying a 2D strided array of size $\frac{R_g \times X \times Y}{\#\,\text{passes} \times \#\,\text{streams}}$ from device to host in stream i.
$T_{Y_1\,\text{fftw}}(1 - R_g)$	$= T_{Y_1\,\text{fftw-cpu}} \times (1 - R_g)$. Time of Y_1-step FFTs on advanced FFTW plan for grouped array of size $(1 - R_g) \times X$ in CPU. Total number of plans is Y_2.
$T_{Y_2 \& X}$	Time of subsequent calculation of Y_2 and X dimensional FFTs.

Load Balancing between GPU and CPU

- *GPU Part of Hybrid 2D FFT*

$$\begin{aligned} TG_{2D} = \quad & \#passes \times max\{[Y_1 \times T_{2d\mathrm{H2D}}(0, R_g)+ \\ & T_{Y_1\mathrm{kernel}}(0, R_g) + T_{2d\mathrm{D2H}}(0, R_g)]; [...]; \\ & [Y_1 \times T_{2d\mathrm{H2D}}(\#\text{ streams-}1, R_g) \\ & + T_{Y_1\mathrm{kernel}}(\#\text{ streams-}1, R_g) \\ & + T_{2d\mathrm{D2H}}(\#\text{ streams-}1, R_g)]; \quad \} \end{aligned}$$

- *CPU Part of Hybrid 2D FFT*

$$TC_{2D} = \frac{Y_2}{\#thds} \times T_{Y_1\mathrm{fftw}}(1 - R_g)$$

- *Estimation of execution time of Hybrid 2D FFT*

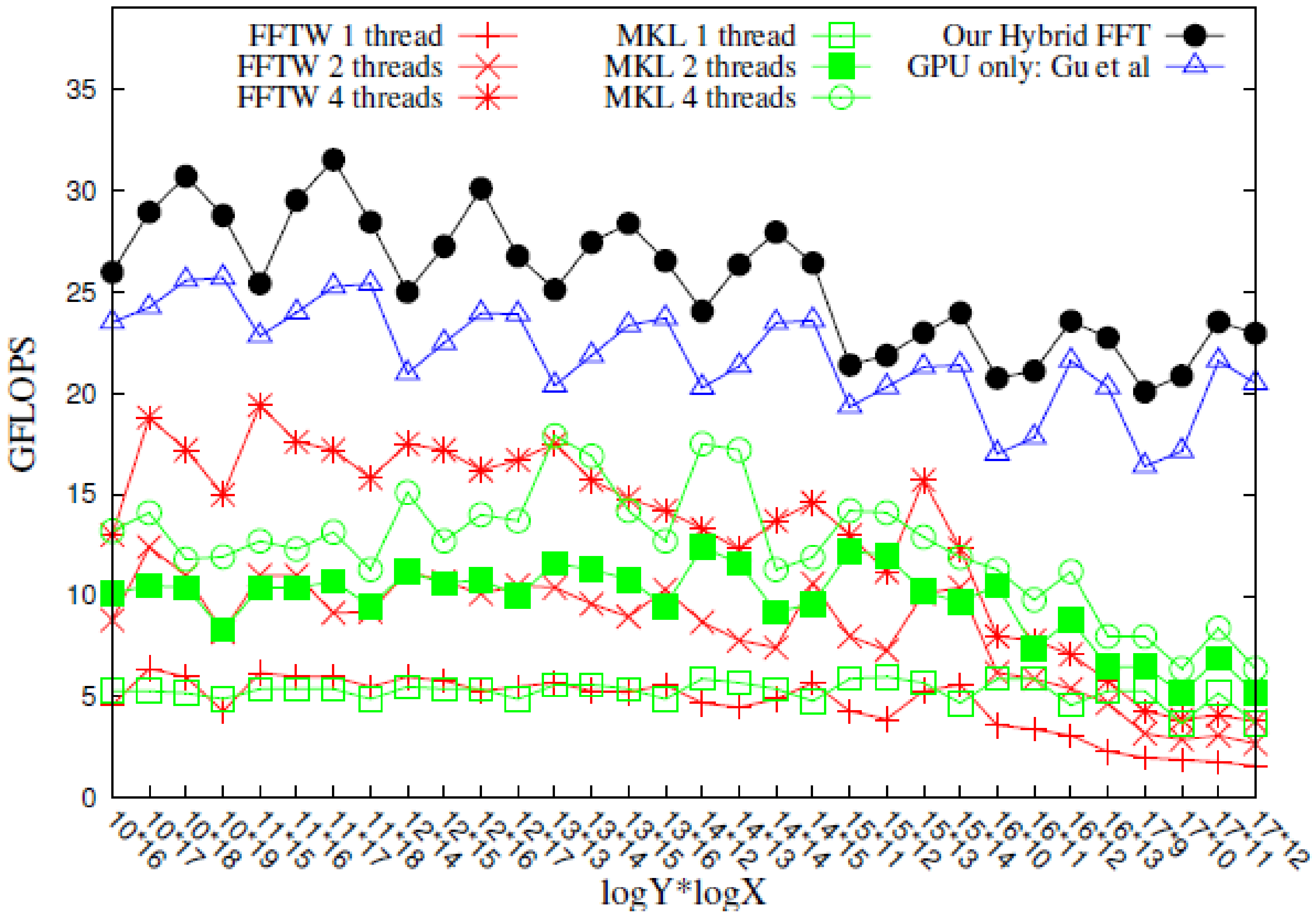
$$T_{Y_1} = max\{TG_{2D}, TC_{2D}\}$$

Evaluation of Preliminary Results

- *Environmental Setup*

GPU	Global Memory	NVCC	PCI
GeForce GTX480	1.5GB	3.2	PCIe2.0 x16
Tesla C2070	6GB	3.2	PCIe2.0 x16
Tesla C2075	6GB	3.2	PCIe2.0 x16
CPU	**Frequency, # of Cores**	**System Memory**	**Cache**
Intel i7 920	2.66GHz, 4 cores	24GB	8192KB

- *Performance Comparison*
  - Test cases are all out-of-card, i.e. ***larger*** than GPU memory.
  - SSE-enabled 1-thread, 2-thread, 4-thread ***FFTW*** 3.3.3 with MEASURE flag.
  - SSE-enabled 1-thread, 2-thread, 4-thread ***Intel MKL*** 10.3.
  - ***Gu***'s out-of-card FFT Library.



2D FFT of size from 2^26 to 2^29 on GTX480

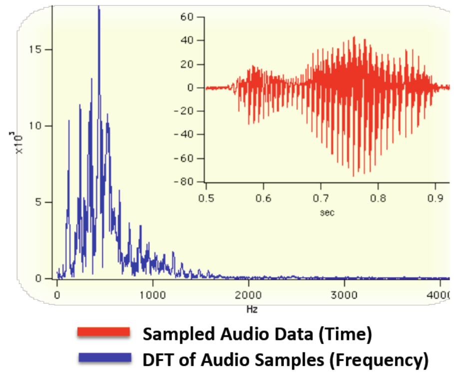
Conclusion

- Our hybrid FFT library concurrently uses both CPU and GPU to compute large FFT problems. The library has three key components:
  - A hybrid large-scale decomposition paradigm to extract concurrency and workload patterns between the two different processor types.
  - A load balancer with empirical performance modeling to determine optimal load balancing between CPU and GPU.
  - An optimizer that exploits substantial parallelism for GPU and CPUs.
  - An effective heuristic to expose opportunities of overlapping communication with computation for FFT decomposition.
- Overall, the preliminary results show that our hybrid library outperforms two best performing FFT implementations by 1.9x and 2.1x, respectively.

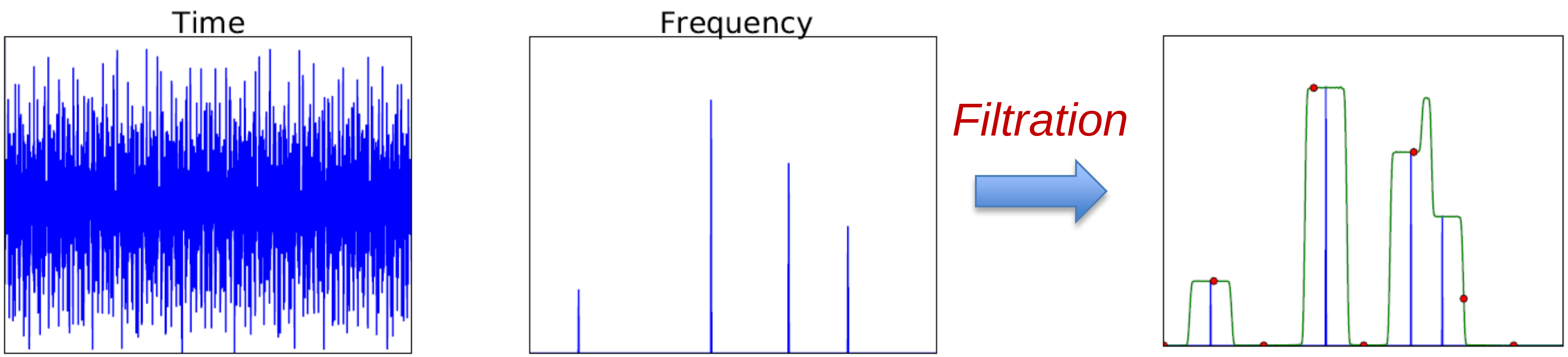
Part II: An Input Adaptive Algorithm for Parallel Sparse Fast Fourier Transform

Motivation

- Original (Dense) DFT/FFT

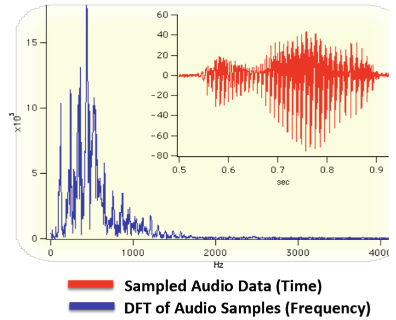


- Sparse FFT

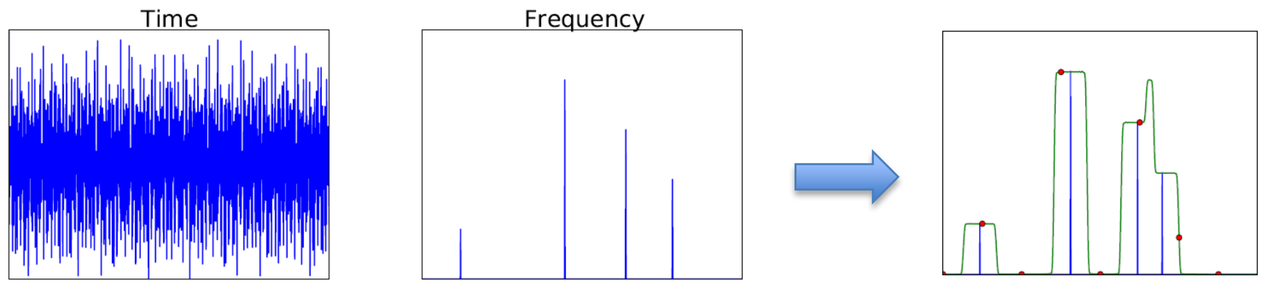


Motivation

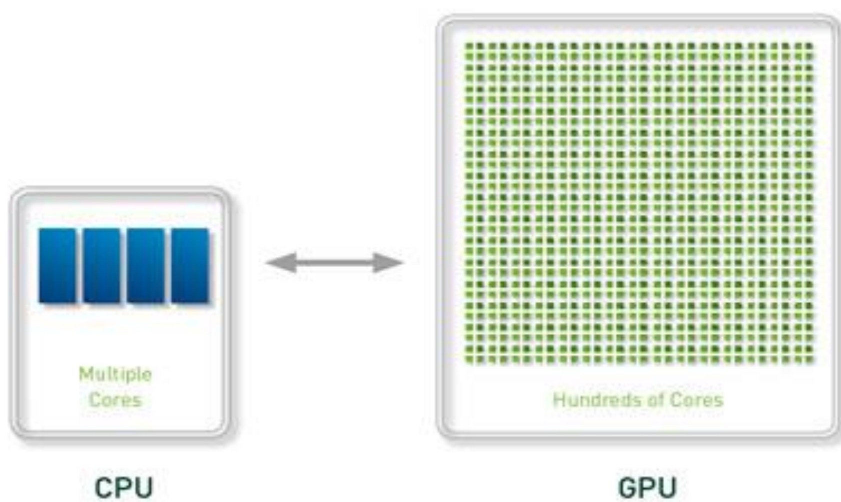
- Original (Dense) DFT/FFT



- Sparse FFT

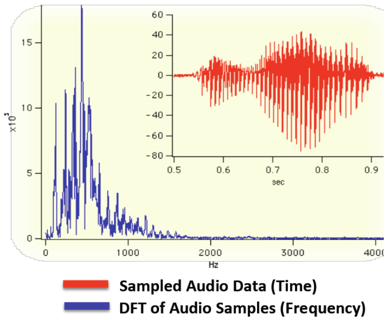


- Parallelization

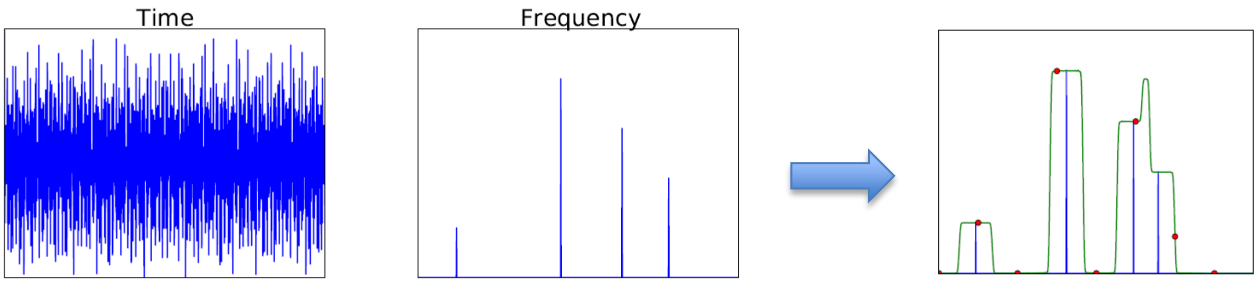


Motivation

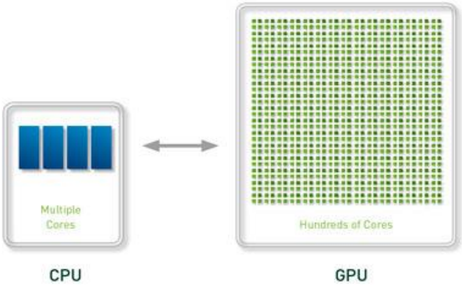
- Original (Dense) DFT/FFT



- Sparse FFT



- Parallelization



- Input Adaptive
  – Spectrum Similarities

Background of DFT/FFT

- Discrete Fourier Transform (DFT)
  - Given $x \in \mathbb{C}^n$, compute its Fourier transform $\hat{x}$:

$$\hat{x}_d = \sum_i x_i \, \omega^{id} \quad \text{for} \quad \omega^{id} = e^{-j2\pi id/N}$$

Background of DFT/FFT

- Discrete Fourier Transform (DFT)
  - Given $x \in \mathbb{C}^n$, compute its Fourier transform $\hat{x}$:

$$\hat{x}_d = \sum_i x_i\, \omega^{id} \quad \text{for} \quad \omega^{id} = e^{-j2\pi id/N}$$

- Fast Fourier transform (FFT) algorithms reduce DFT's operational complexity from O(N^2) into O($N\log N$).
  - Cooley–Tukey algorithm;
  - Prime–Factor (Good-Thomas) algorithm;
  - Rader's algorithm;
  - Bluestein's algorithm.

Background of Sparse FFT

- All FFT algorithms cost time proportional to input size *N*.

Background of Sparse FFT

- All FFT algorithms cost time proportional to input size *N*.
- What if the output of a FFT is K-sparse?

Background of Sparse FFT

- All FFT algorithms cost time proportional to input size *N*.
- What if the output of a FFT is K-sparse? → only K non-zero Fourier coefficients. Its runtime is sublinear to N.

Background of Sparse FFT

- All FFT algorithms cost time proportional to input size *N*.
- What if the output of a FFT is K-sparse?
- Sublinear sparse Fourier algorithm was first proposed by Kushilevitz et.al, and since then, has been extensively studied in many applications.

Background of Sparse FFT

- All FFT algorithms cost time proportional to input size N.
- What if the output of a FFT is K-sparse?
- Sublinear sparse Fourier algorithm was first proposed by Kushilevitz et.al, and since then, has been extensively studied in many applications.
- However, their runtimes have large exponents in the polynomial of k and $\log N$, and their complex algorithmic structures impose restrictions on fast and parallel implementations.

Background of Sparse FFT

- Hassanieh et.al. recently presented improved algorithms with the runtime of

$$O(k\log N\log(N/k))$$ or even faster $$O(k\log N)$$

Background of Sparse FFT

- Hassanieh et.al. recently presented improved algorithms with the runtime of

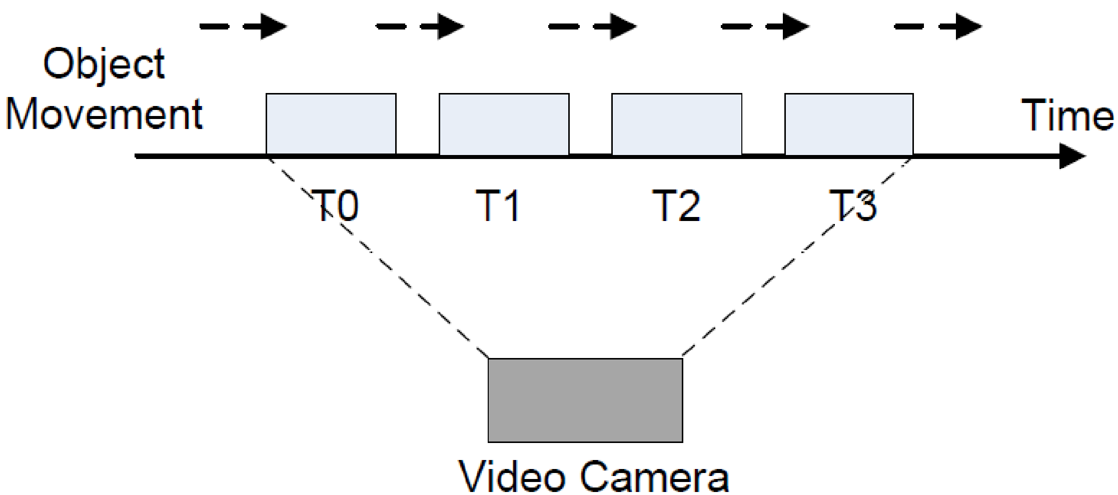
  $O(k\log N\log(N/k))$ or even faster $O(k\log N)$
- Limitations:
  - Iterates over passes;
  - Dependency exists between consecutive iterations;
  - Oblivious to input characteristics.

Motivation

- Advantages
  - Many applications are sparse in the frequency domain and hence can benefit from sparse FFT.

Motivation

- Advantages
  - Many applications are sparse in the frequency domain and hence can benefit from sparse FFT.
  - Homogeneity exists in specific spectrums.

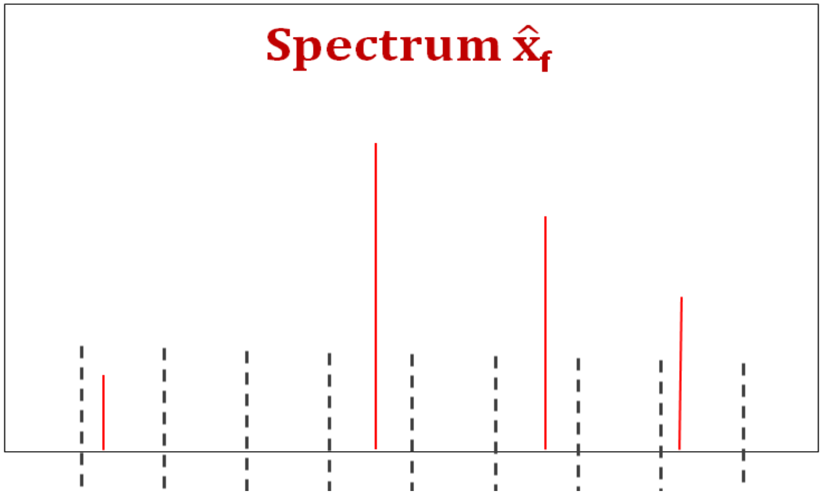


Motivation

- Advantages
  - Many applications are sparse in the frequency domain and hence can benefit from sparse FFT.
  - Homogeneity exists in the spectrums.
  - Parallelism is always needed to be exploited as much as possible from the sparse FFT algorithms.

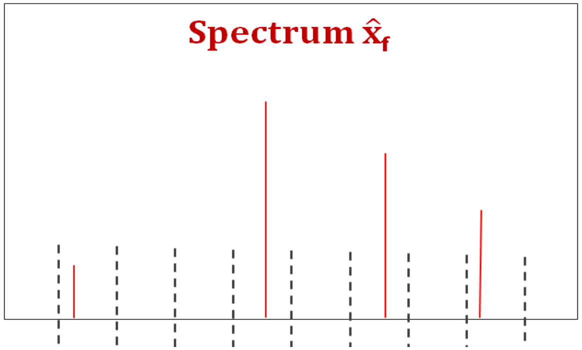
General Input Adaptive Sparse FFT

- Basics
  - **Bucketize:** Hash the spectrum into a few buckets. Each bucket is to have only one large coefficient.
  - **Estimate:** Estimate large coefficient in each non-empty bucket.

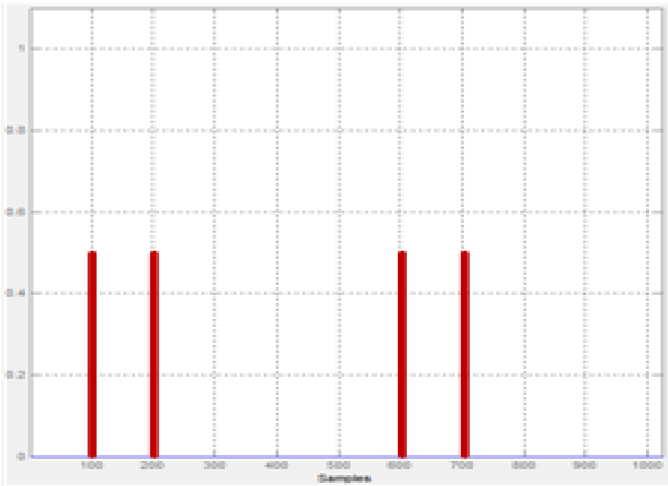


General Input Adaptive Sparse FFT

- Basics
  - **Bucketize:** Hash the spectrum into a few buckets. Each bucket is to have only one large coefficient.
  - **Estimate:** Estimate large coefficient in each non-empty bucket.
- Prerequisite
  - Need to generate Fourier template once. The template includes the locations of non-zero frequencies and is made use of for the spectrums in the following samples.

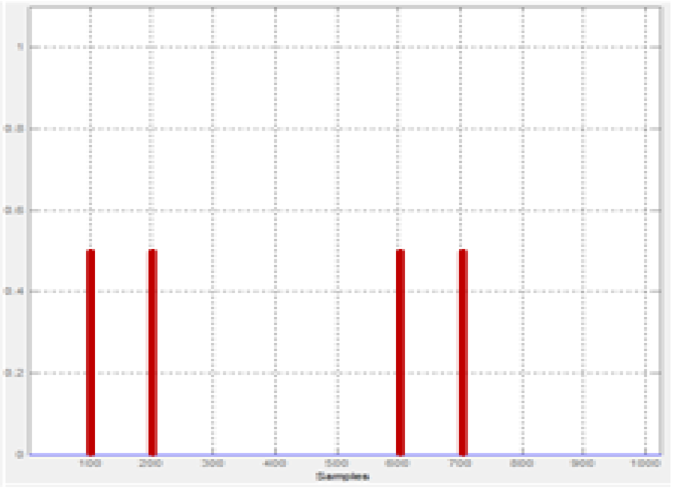


General Input Adaptive Sparse FFT

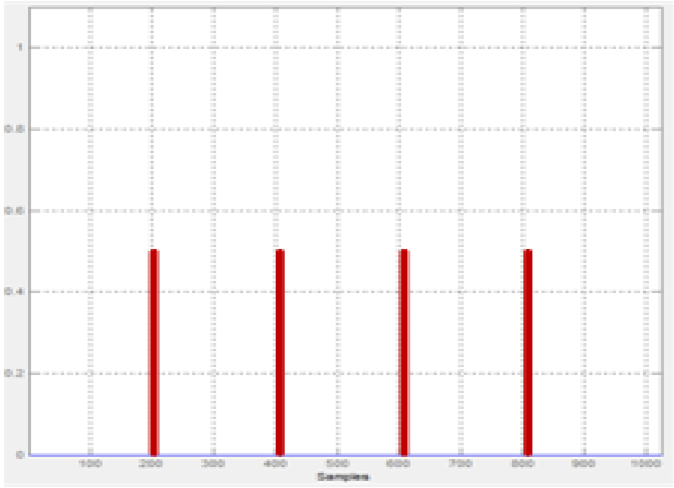


(a) Original Spectrum of Signal

General Input Adaptive Sparse FFT

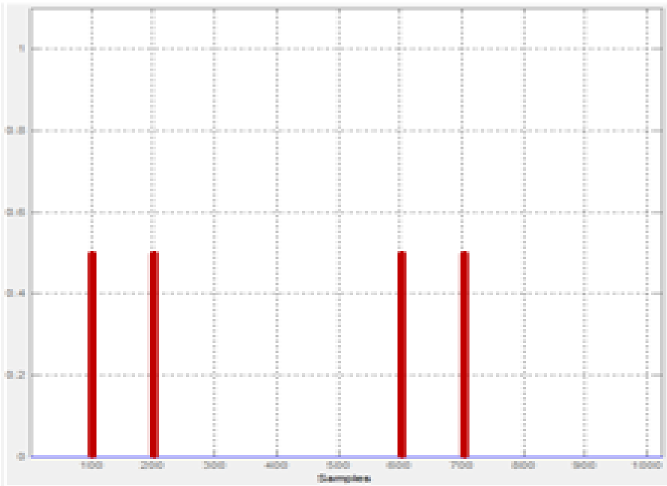


(a) Original Spectrum of Signal

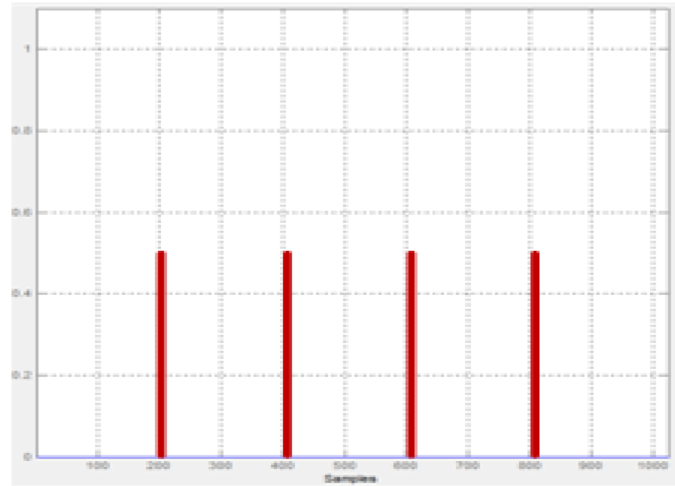


(b) Permuted Spectrum of Signal

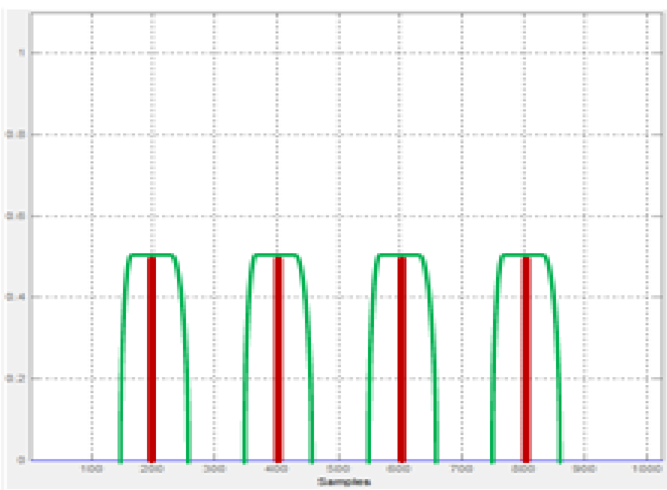
General Input Adaptive Sparse FFT



(a) Original Spectrum of Signal

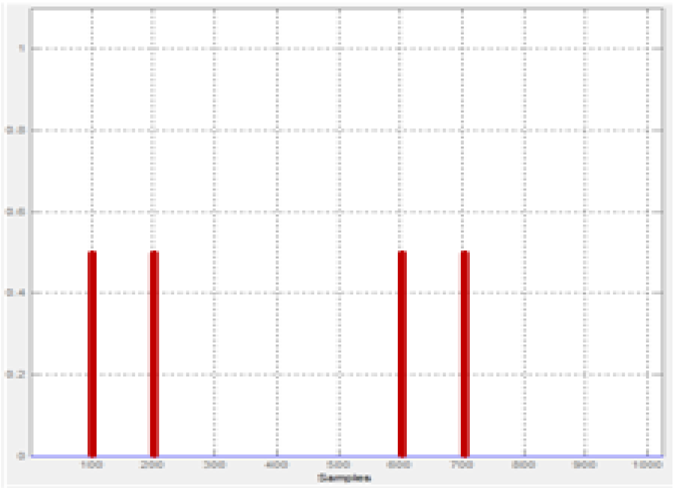


(b) Permuted Spectrum of Signal

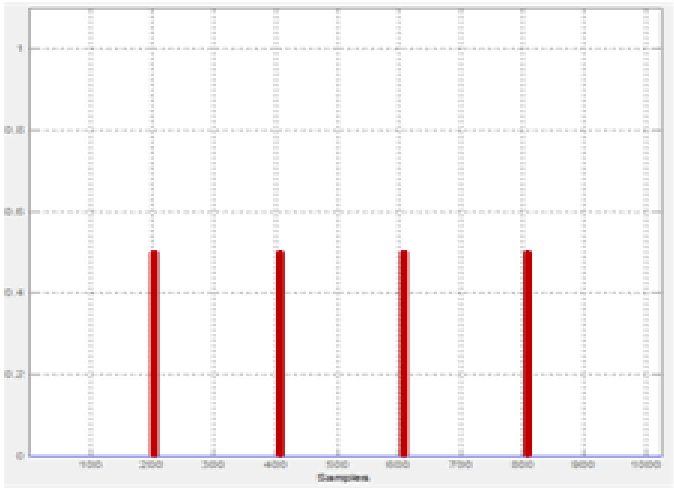


(c) Applying Filter and Subsampled FFT

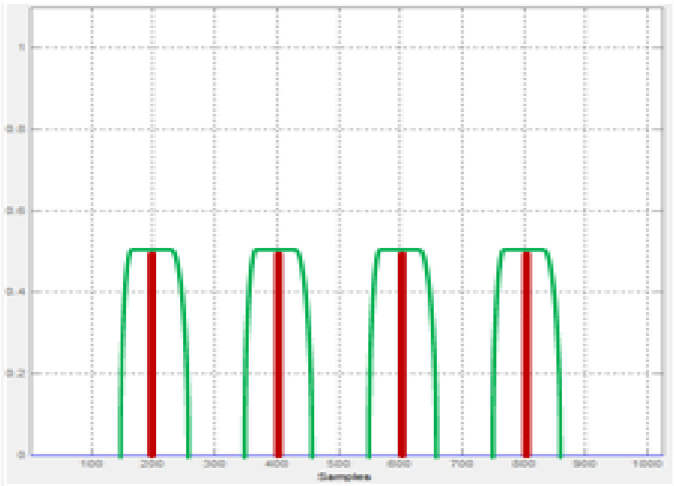
General Input Adaptive Sparse FFT



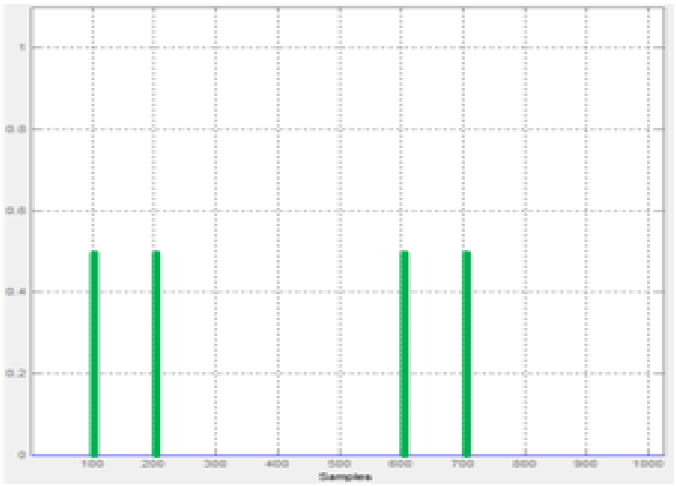
(a) Original Spectrum of Signal



(b) Permuted Spectrum of Signal



(c) Applying Filter and
Subsampled FFT



(d) Recovered Spectrum of Output

Parallel Input Adaptive Sparse FFT

- Parallelism Exploitation
  - Input adaptive approach directly permutes sparse coefficients in spectrum domain, rather than to estimate permutation in time domain.
  - To get rid of dependency existing between consecutive iterations in the traditional permutation estimation process.
- Data Parallelism and Kernel Execution
  - Graphic Processing Units (GPUs) for well-suited data parallel computations.
  - Data parallelism exists in subsections of hashed index computation, filtering and permuting input, subsampling FFT and location recovery.
  - GPU computational kernels are constructed for each subsection.

Parallel Input Adaptive Sparse FFT

- Parallelism Exploitation and Kernel Execution

Kernel	# of threads	Functionality
HashFunc()	k	Compute hashed indices of permuted coefficients and determine shift factors. The loop of size k is decomposed and each scalar thread in kernel concurrently works as each index j in the algorithm.
Perm()	$k^2 \log N$	Apply filter and permutation to input. Each thread multiplies filter as well as shifting factor with input for one element.
Subsample()	k	Parallelize subsampling to input.
TunedFFT()	$(K_1 \times K_2)$	Well-tuned GPU based FFT Library.
Recover()	k	Parallelize the loop of location estimation.
Filtering()	$k\log N$	Parallelize loop size $O(k\log N)$ of applying filter to the input.
Shifting()	$\min\{k, \frac{N}{k\log N}\} k\log N$	Make each thread shift one input element by a factor. For each shifting event, TunedFFT() is launched before we gain output.

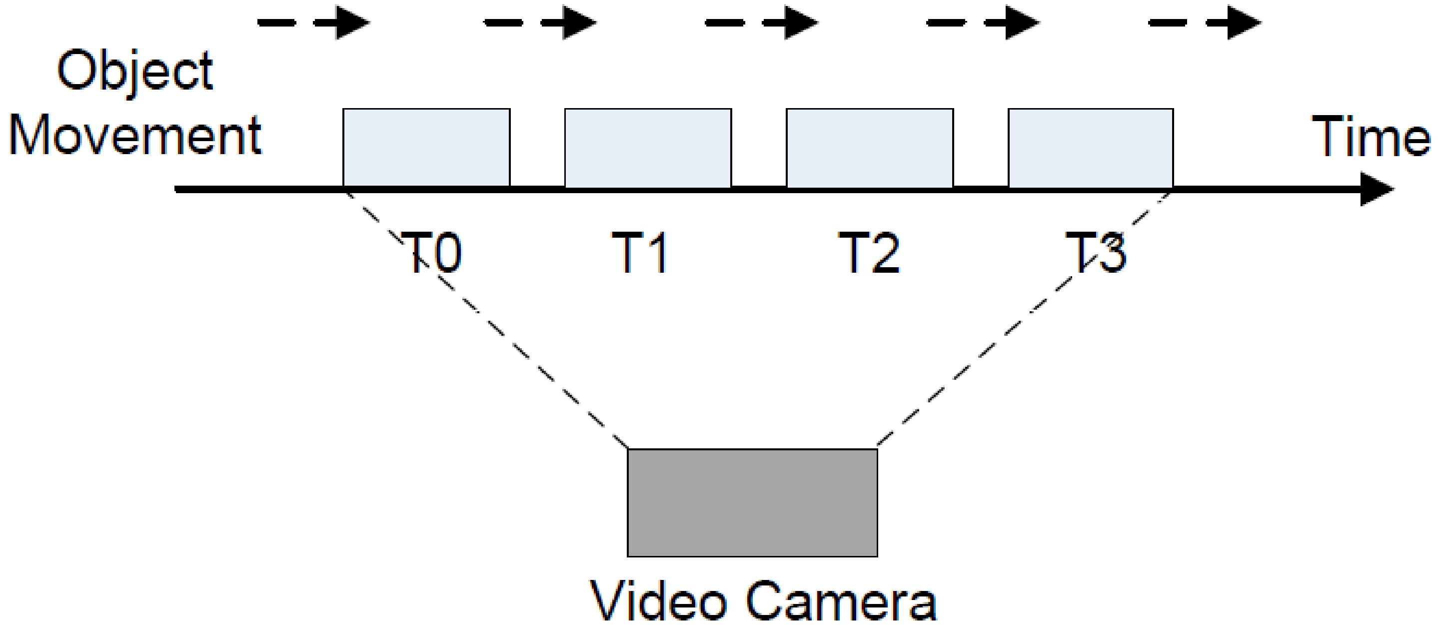
GPU computational kernels

Parallel Input Adaptive Sparse FFT

- Performance Optimizations
  - Maximize GPU device memory bandwidth.
    - Coalesced memory accesses.
      - ➢ Coalesced accesses are enabled by setting the size of thread block be 16×2^p where $p \geq 0$, and set grid size to $\frac{\#threads}{block\_size}$.
  - Maximize data sharing between kernels.
    - Increase PCI bandwidth significantly.
    - Only two transfers between CPU and GPU to maximize PCI bandwidth.

SFFT Real-World Application

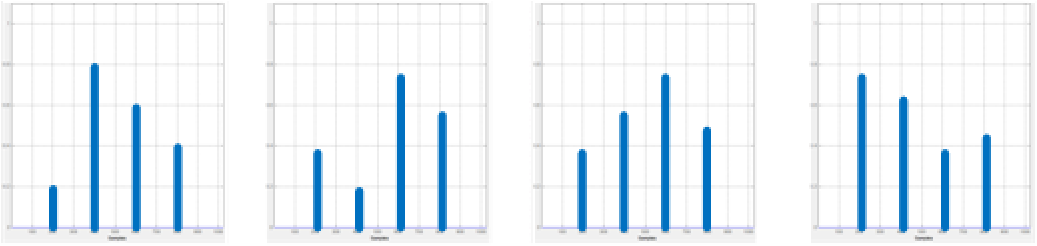
- Video recording of object movement
  - Fix a video camera to record a 2D object movement for a duration of time $\{T_0, T_1, \ldots, T_t\}$.



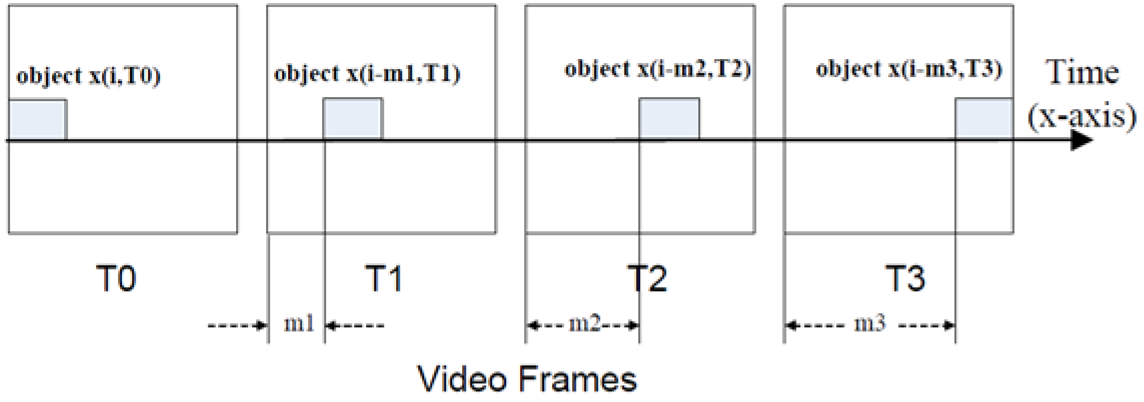
SFFT Real-World Application

- Video recording of object movement
  - Fix a video camera to record a 2D object movement for a duration of time $\{T_0, T_1, \ldots, T_t\}$.
  - The object in video frame is denoted as 2D image matrix and as a 1D row-major array.
  - Similarity in the spectrum.

Spectrums



Time Domain



Evaluation of Preliminary Results

- Environmental Setup

GPU	Global Memory	NVCC	PCI
GeForce GTX480	1.5GB	3.2	PCIe2.0 x16
Tesla C2070	6GB	3.2	PCIe2.0 x16
Tesla C2075	6GB	3.2	PCIe2.0 x16

CPU	Frequency, # of Cores	System Memory	Cache
Intel i7 920	2.66GHz, 4 cores	24GB	8192KB

Evaluation of Preliminary Results

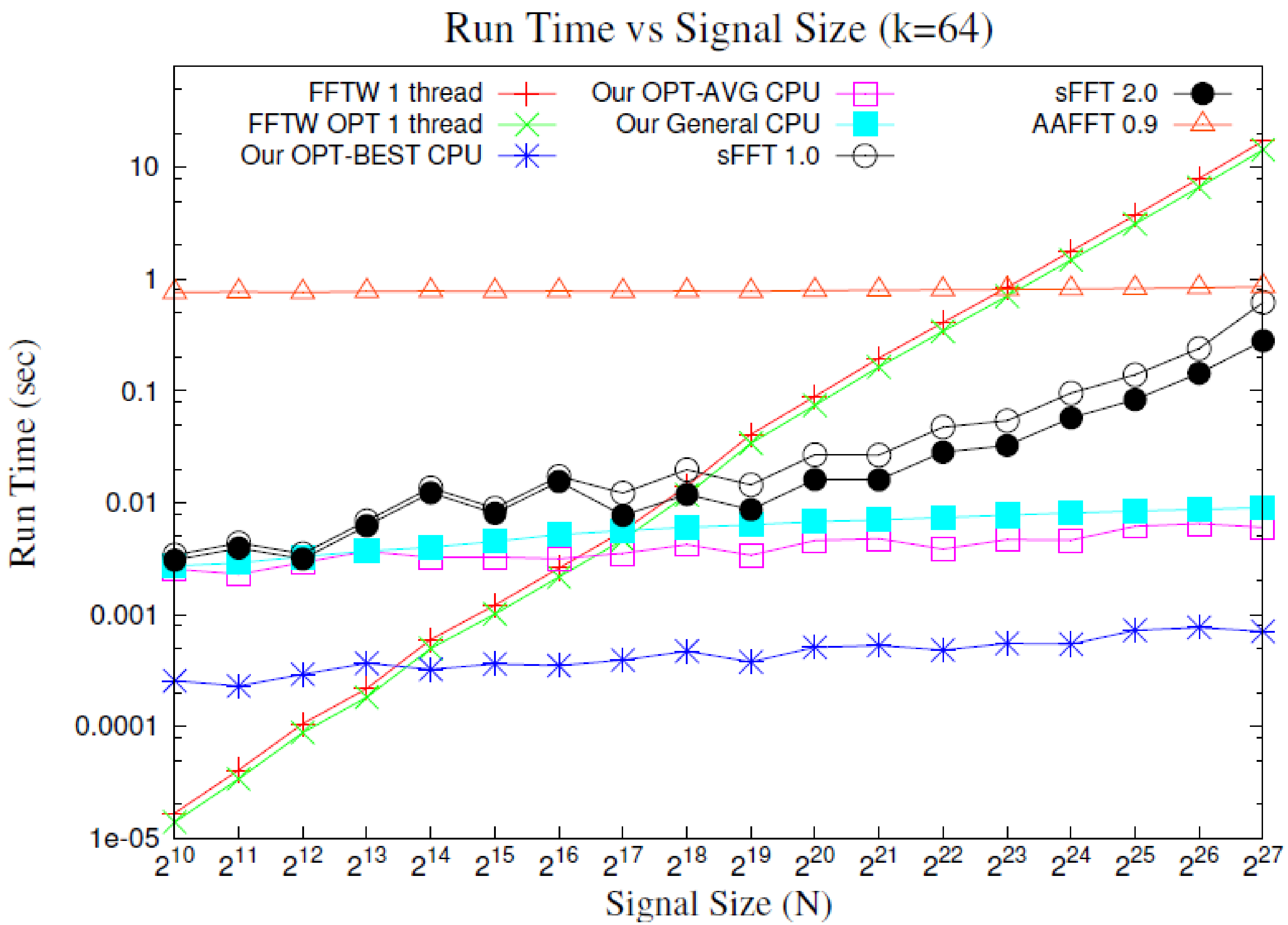
- Environmental Setup

GPU	Global Memory	NVCC	PCI
GeForce GTX480	1.5GB	3.2	PCIe2.0 x16
Tesla C2070	6GB	3.2	PCIe2.0 x16
Tesla C2075	6GB	3.2	PCIe2.0 x16

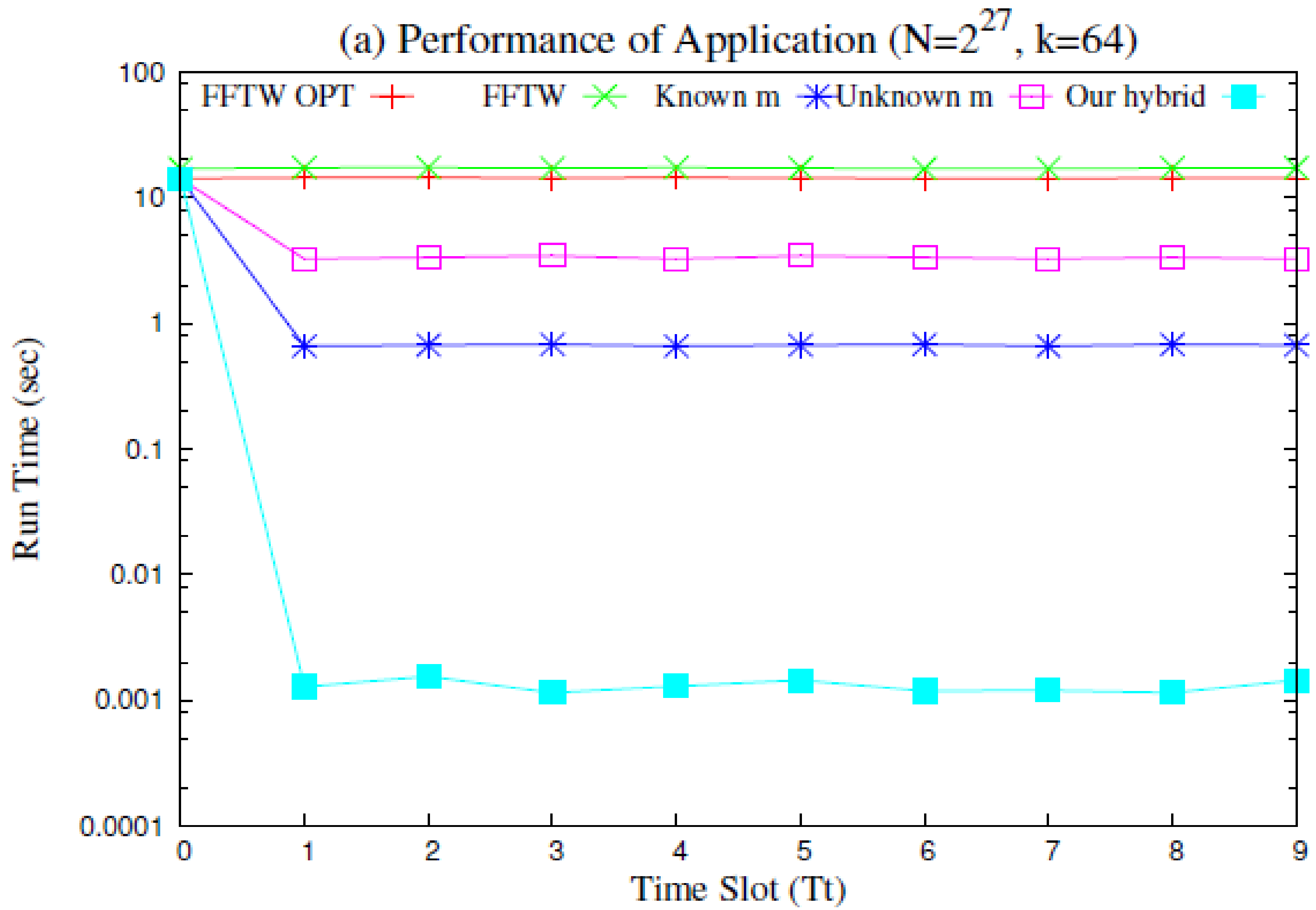
CPU	Frequency, # of Cores	System Memory	Cache
Intel i7 920	2.66GHz, 4 cores	24GB	8192KB

- Sequential version
  - 1-thread SSE-enabled FFTW 3.3.3
    - basic version (ESTIMATE)
    - optimal version (MEASURE)
  - sFFT 1.0 and 2.0
  - AAFFT 0.9
- Parallel version
  - 4-thread SSE-enabled FFTW 3.3.3 with basic and optimal version
  - CUFFT 3.2

General sparse FFT in sequential case: Run Time vs. Signal Size

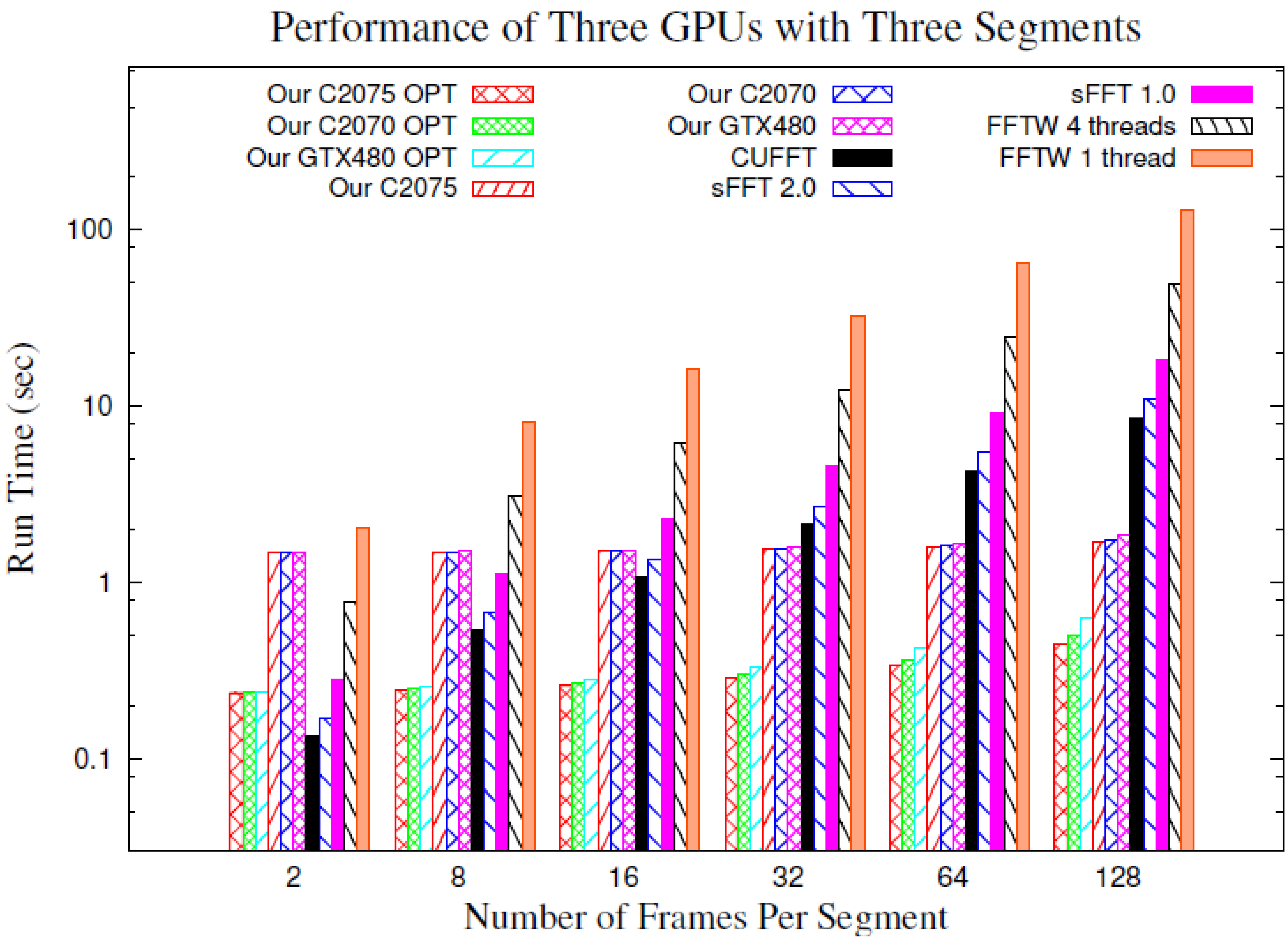


SFFT Real-World Application

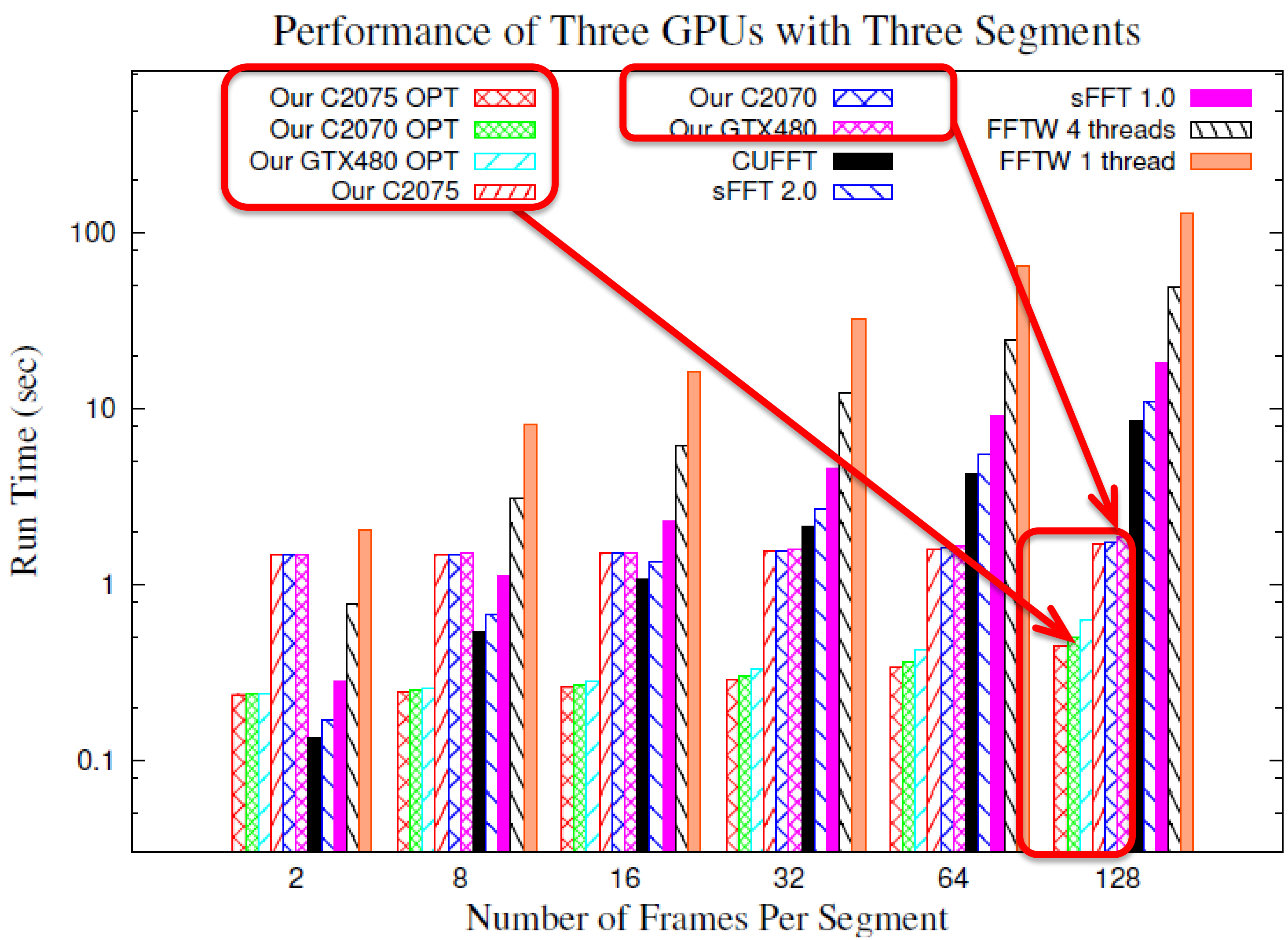


Performance of a real-world application

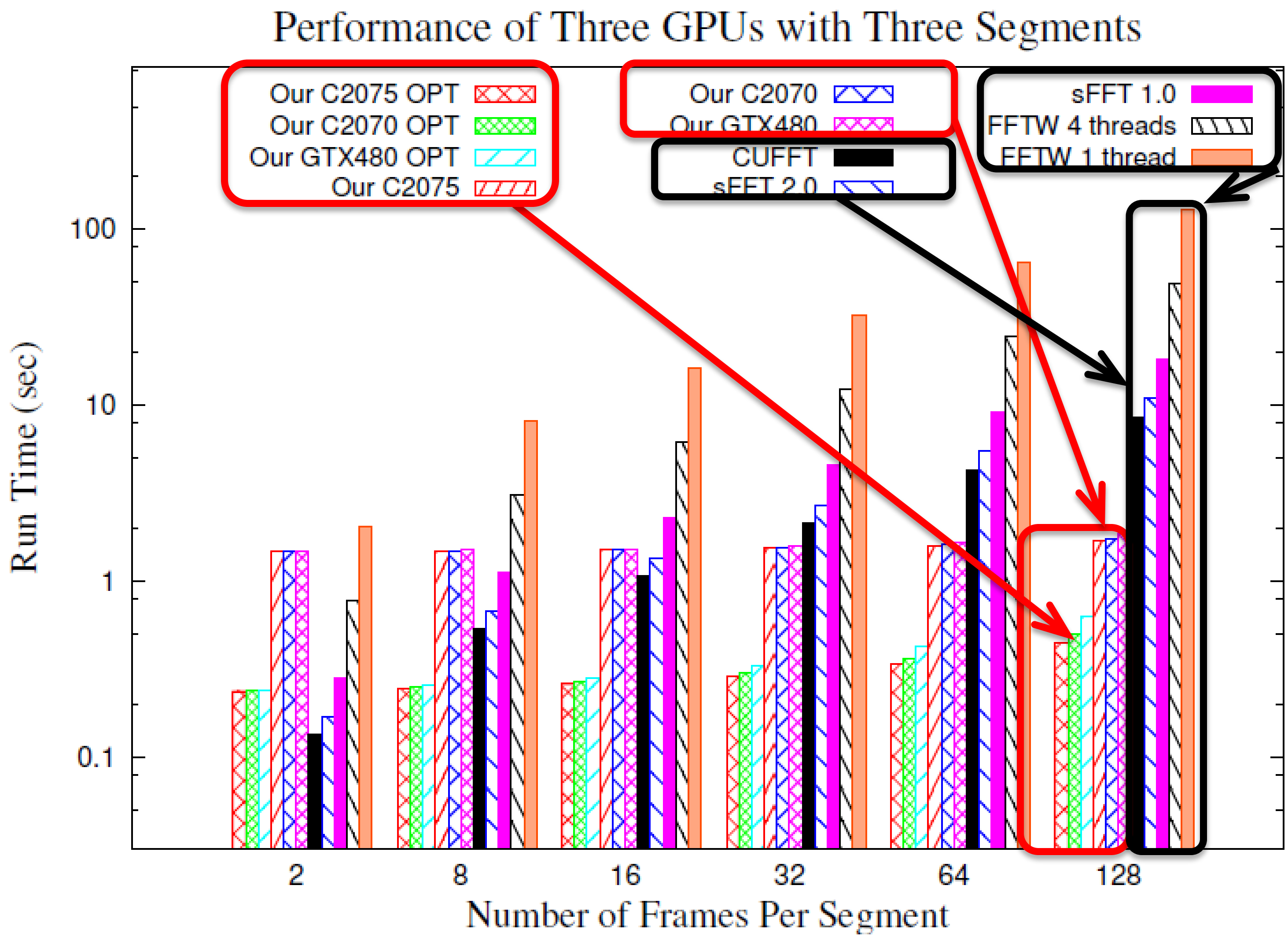
GPU Performance of Input Adaption Process with 3 Video Segments.



GPU Performance of Input Adaption Process with 3 Video Segments.



GPU Performance of Input Adaption Process with 3 Video Segments.



Conclusion

- An input-adaptive sparse FFT algorithm that extracts input similarities and tunes adaptive filters to package non-zero Fourier coefficients into sparse bins.
- Non-iterative with high computation intensity such that substantial parallelism is exploited for CPUs and GPU to improve performance.
- Overall, our algorithm aims to be faster than other FFTs both in theory and implementation.

Thank You !

Questions?