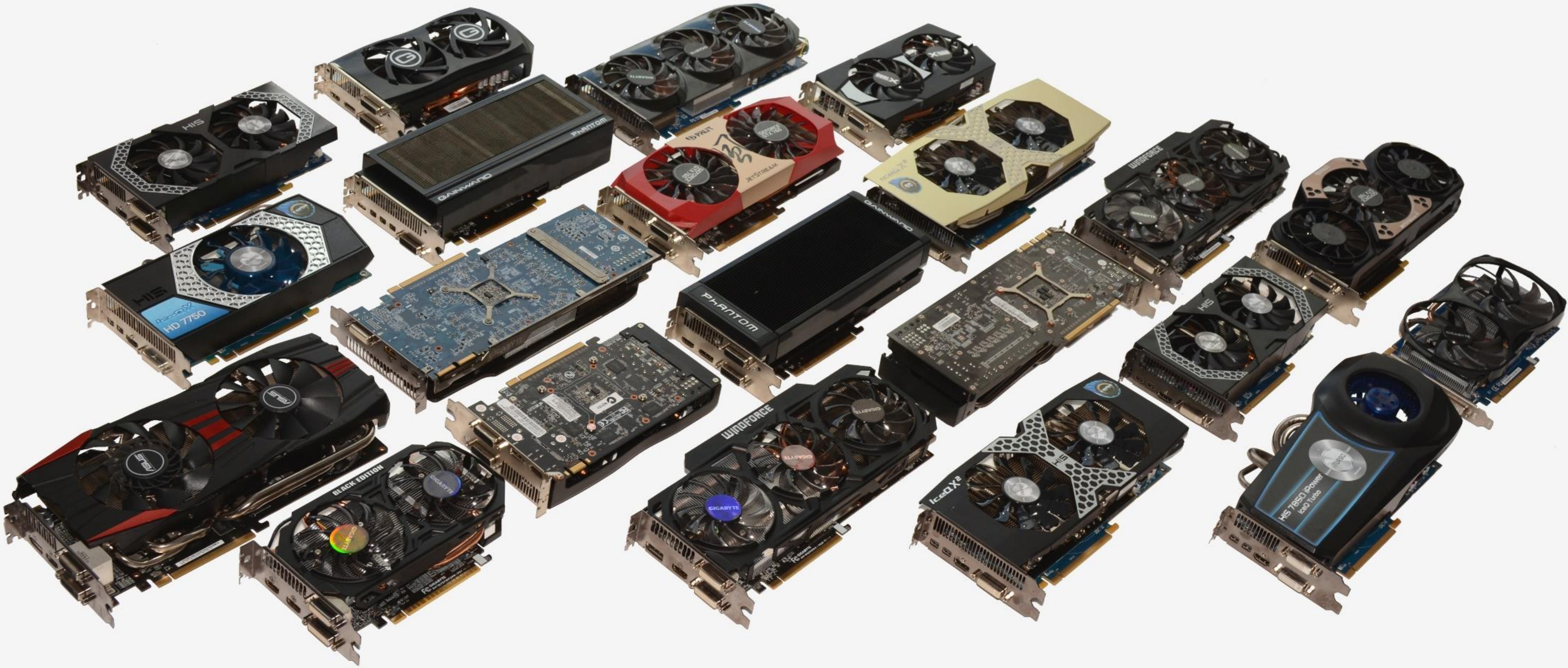
Parallel Computing with CUDA

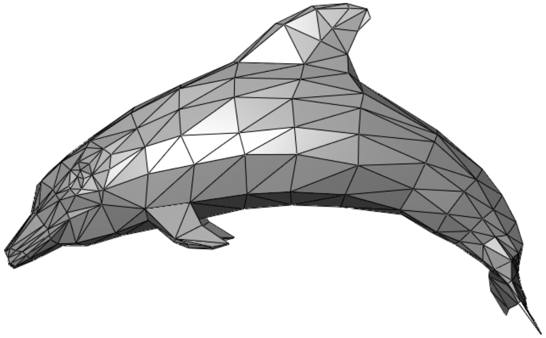
Diogo R. Ferreira

Graphics cards

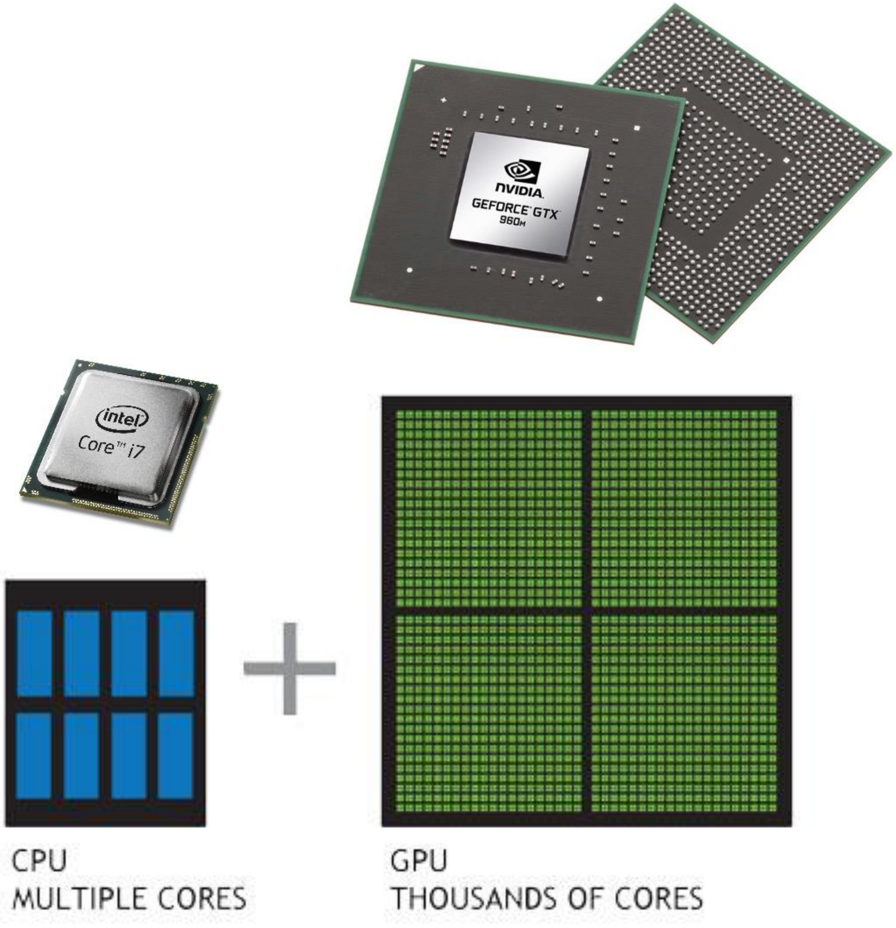


GPU

- The Graphics Processing Unit (GPU)
  - computer graphics and image processing
  - massively parallel and more powerful than the CPU
  - CUDA makes the GPU accessible for general-purpose programming



CPU vs. GPU


NVIDIA
GEFORCE GTX
960M
intel
Core i7
CPU
MULTIPLE CORES
GPU
THOUSANDS OF CORES

CPU vs. GPU

- CPU
    - higher clock speed
    - sequential tasks done in a certain order
    - ad-hoc operations on small arrays
- GPU
    - lower clock speed
    - independent tasks done in any order
    - same operation on very large arrays

CUDA

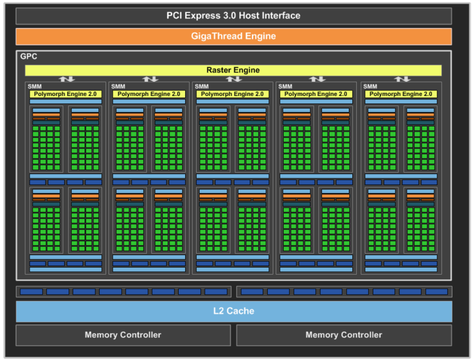
- CUDA was developed by NVIDIA
  - and works on NVIDIA GPUs
  - a more standard (but less popular) alternative is OpenCL

NVIDIA CUDA®

OpenCL

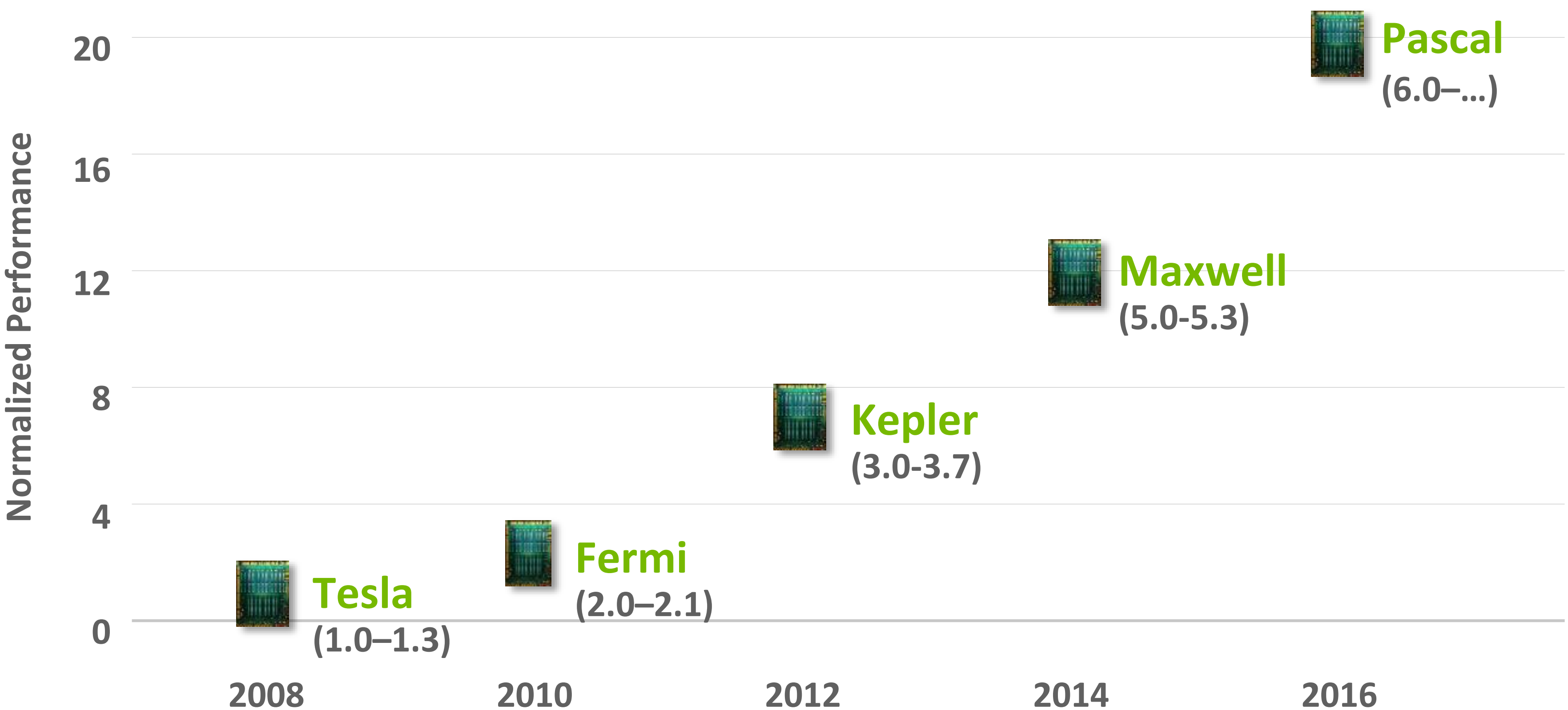
CUDA

- CUDA = architecture + programming model
  - architecture: hardware structure and design
  - programming model: how to develop software on top of that hardware
    - for maximum performance, software must be tuned to hardware




```
__global__ void kernel(int* d_array, int size)
{
        ...
}

kernel<<<blocks, threads>>>(d_array, size);
```

CUDA architecture

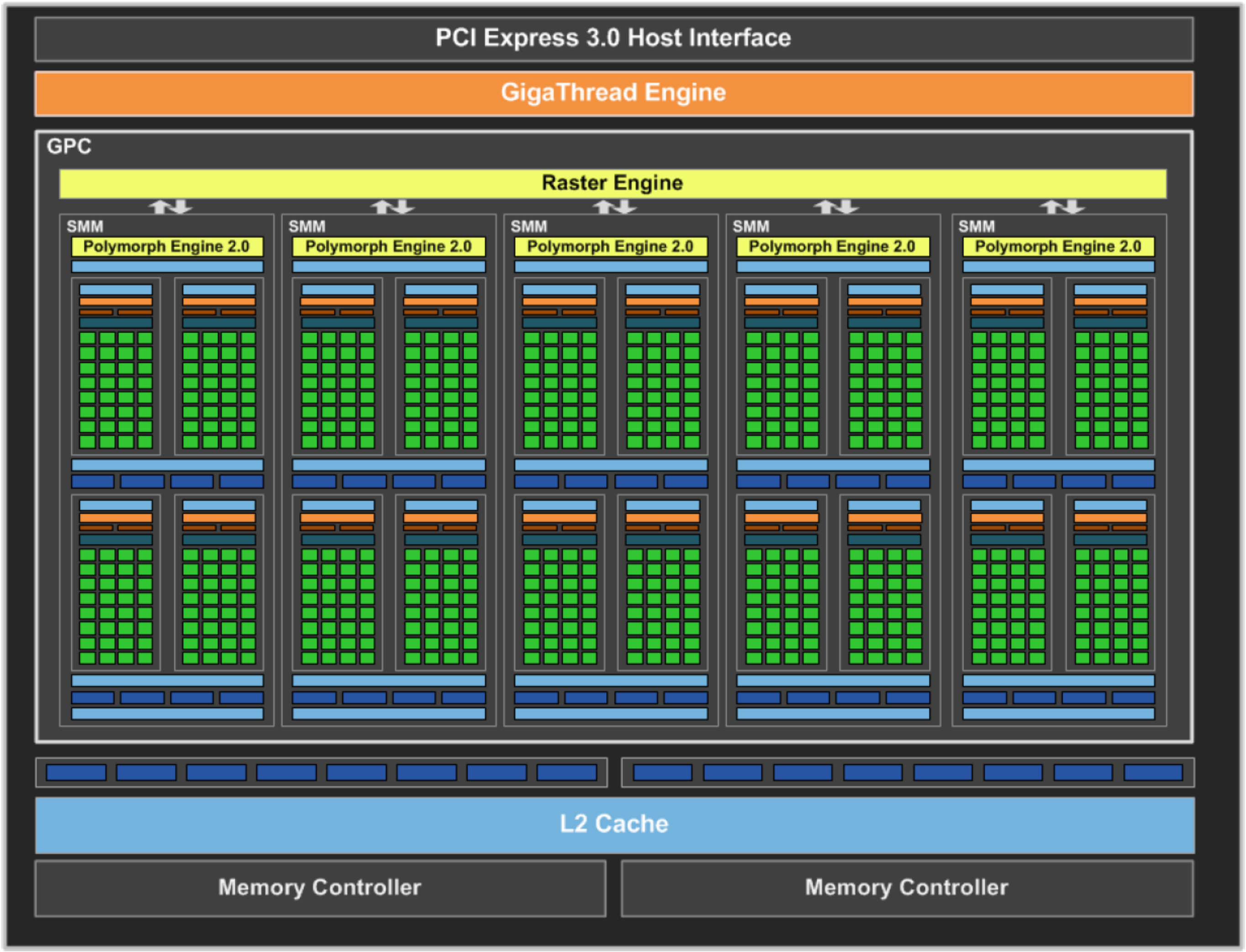


Anatomy of a CUDA GPU

GeForce GTX 750 Ti

Maxwell (5.0)

GM107 chip



The Streaming Multiprocessor

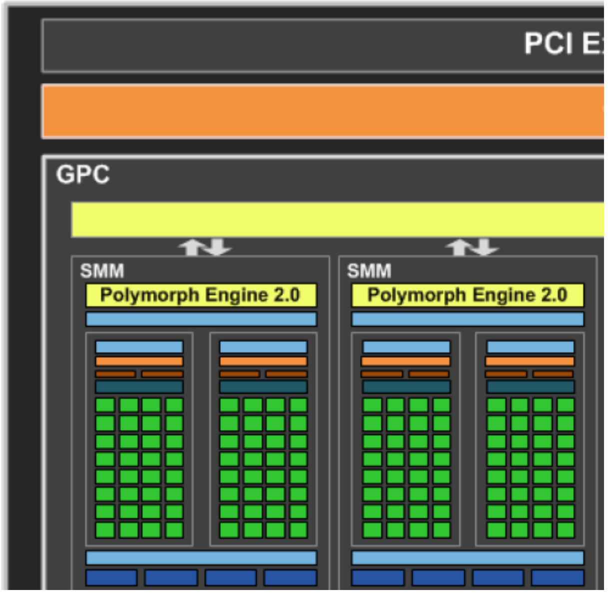
(only upper half is shown)

1 warp = 32 threads

32 cores running “in sync”

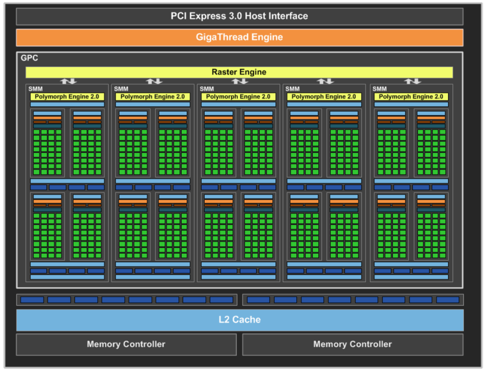
CUDA architecture

- CUDA architecture version 5.x (“Maxwell”)
  - a GPU has one or more graphics processing clusters (GPC)
  - each GPC has one or more streaming multiprocessors (SM)
  - each SM has one or more processing units (4)
  - each processing unit has a number of cores (32)



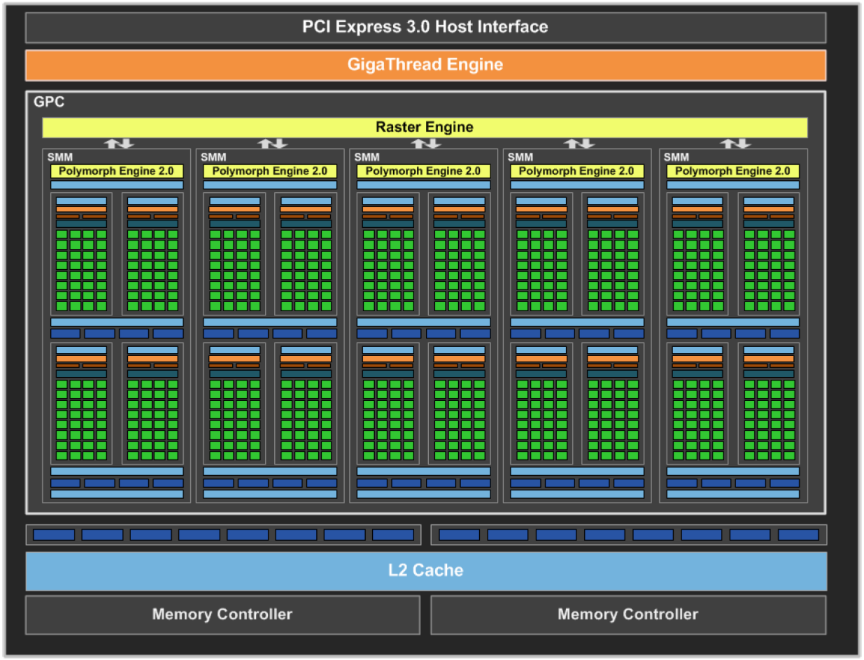
CUDA architecture

- Some examples
  - GTX 750 Ti
    - 1 GPC
    - 5 SMs
    - 4x32 cores per SM
    - 1x5x4x32 = 640 cores (total)
  - GTX 980
    - 4 GPCs
    - 4 SMs per GPC
    - 4x32 cores per SM
    - 4x4x4x32 = 2048 cores (total)



CUDA programming model

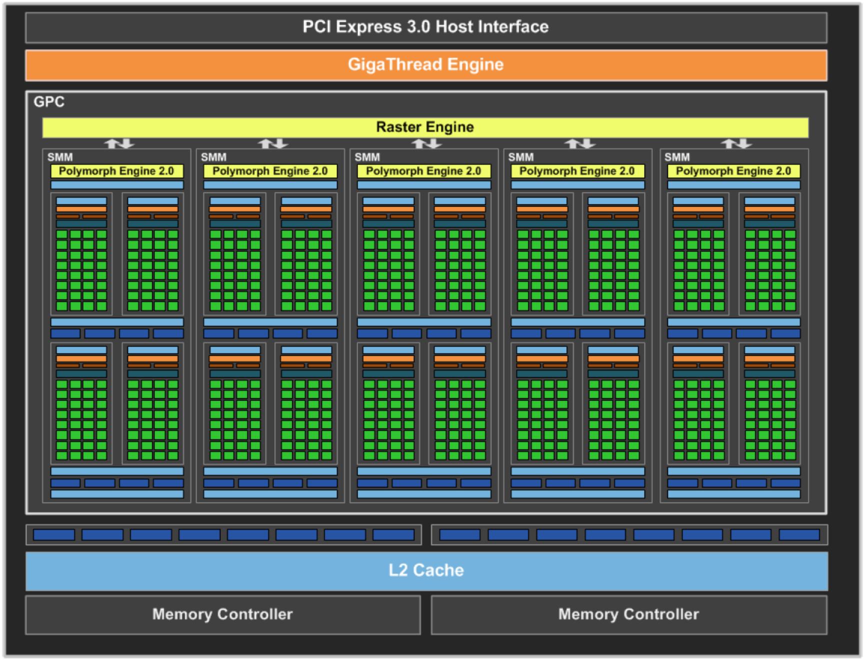
- Basic ideas
  - run multiple threads
  - prefer many short-lived threads over fewer, longer threads (why?)
  - what happens when (number of threads) > (number of cores) ?



CUDA programming model

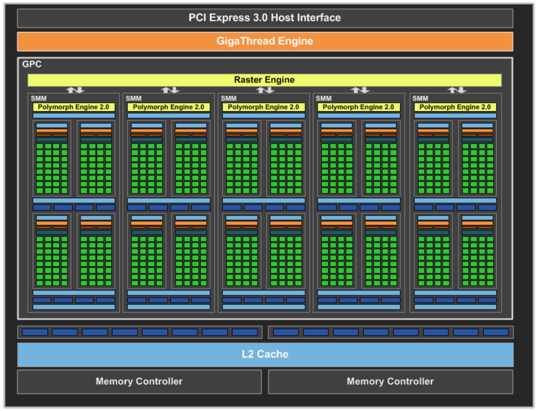
- Threads are organized into <u>blocks</u>
  - each SM receives a block for processing
  - when finished, receives another block (if there are more)

- Questions
  - how many blocks?
  - what is the block size?



CUDA programming model

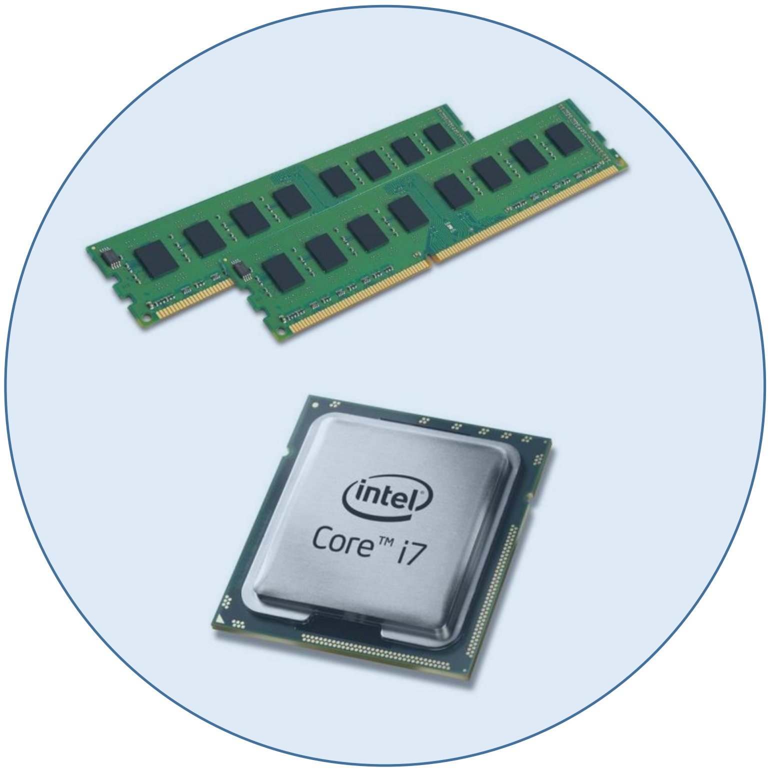
- Example
  - GPU has 5 SMs, 128 cores per SM
  - we need to run 2000 threads
  - what is the ideal block size?



threads	block size	no. blocks	no. blocks	no. threads	wasted
2000	128	15.6	16	2048	48
2000	256	7.8	8	2048	48
2000	384	5.2	6	2304	304
2000	512	3.9	4	2048	48
2000	640	3.1	4	2560	560
2000	768	2.6	3	2304	304
2000	896	2.2	3	2688	688
2000	1024	2.0	2	2048	48

Part 2
Kernels

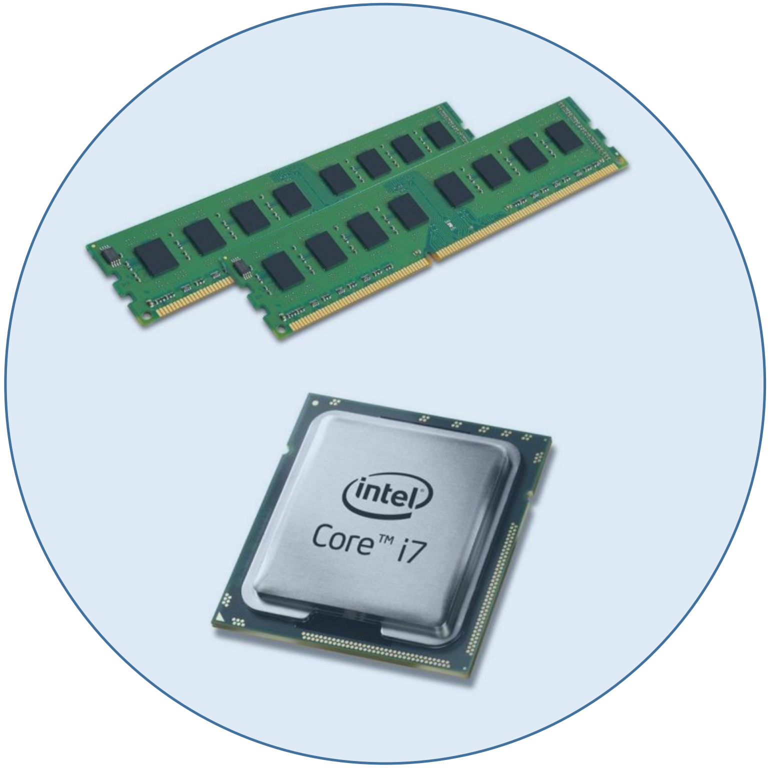
Host vs. device



host
host memory (i.e. RAM)
host functions

device
device memory (i.e. GPU mem)
device functions

Host vs. device



called on the host **kernel** executed on the device

(a.k.a. global function)

Host vs. device

- The host (CPU) calls kernels
  - the CPU “coordinates” the sequence of tasks
- The device (GPU) executes kernels
  - the GPU parallelizes (i.e. “accelerates”) each task

kernel 1

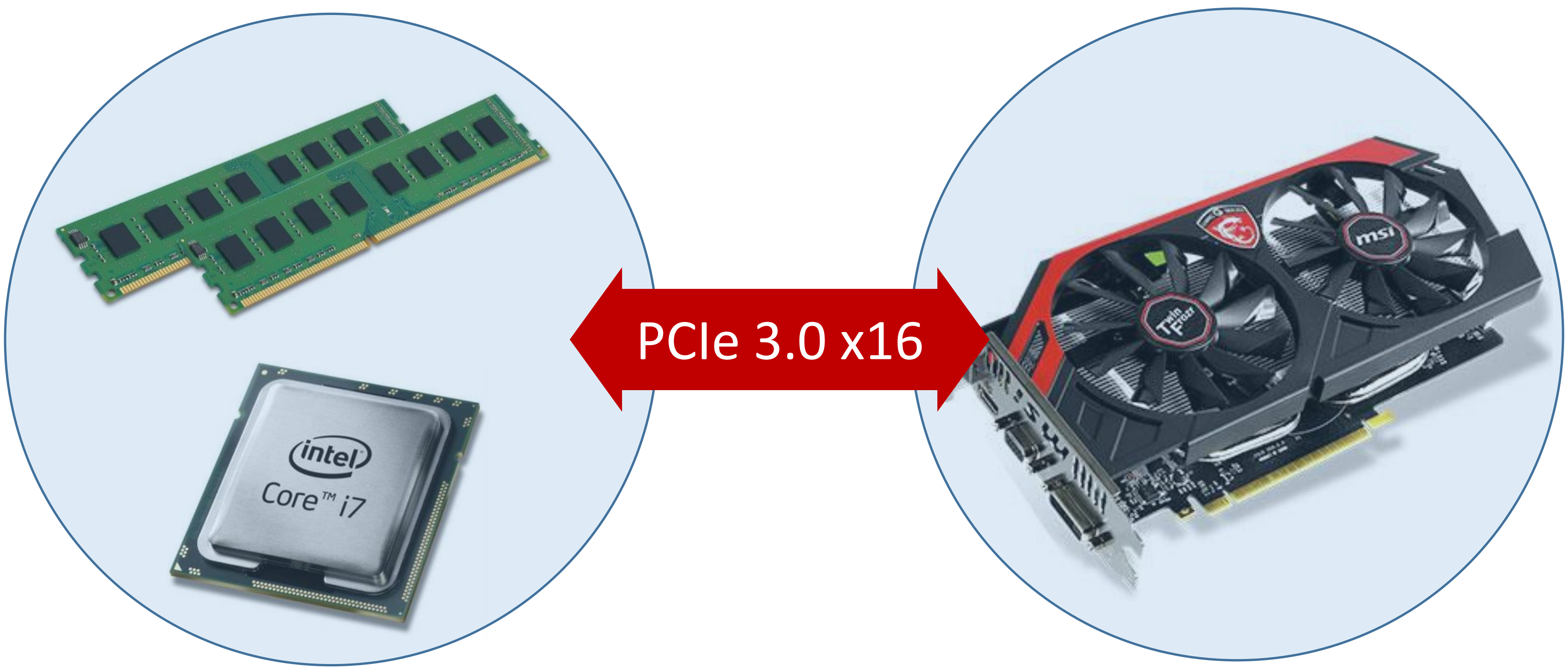
kernel 2

kernel 3

sequence of tasks

Host vs. device

- Kernels operate on data in GPU (device) memory
- Data must be transferred between host and device



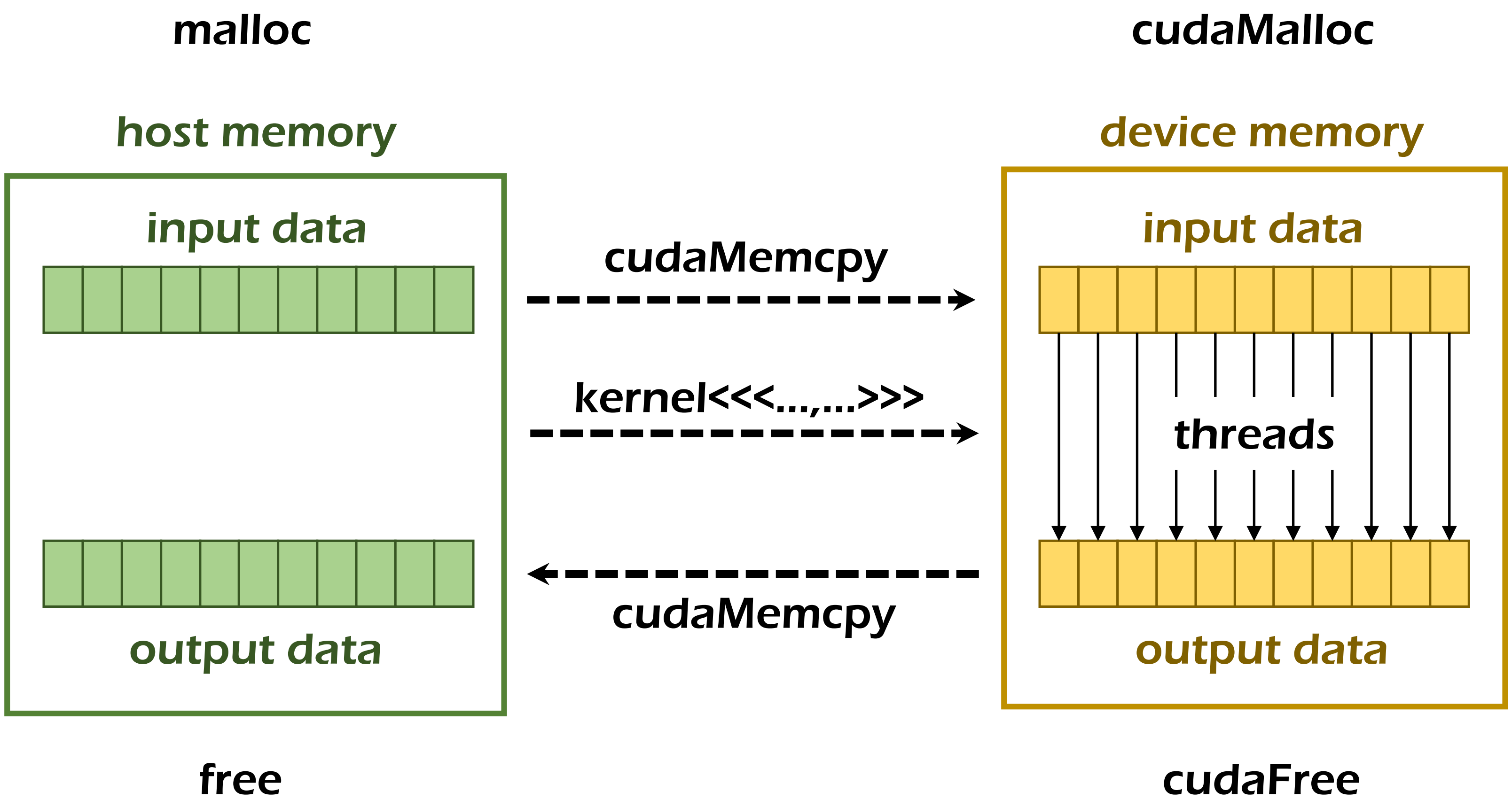
CUDA programming

- Typical CUDA programming
  - allocate memory on device
  - transfer data from host to device
  - execute kernel(s)
  - transfer results from device to host
  - free memory on device

CUDA programming

- Typical CUDA programming
  - `cudaMalloc(...)`
  - `cudaMemcpy(...)`
  - `kernel<<<blocks, threads>>>(...)`
  - `cudaMemcpy(...)`
  - `cudaFree()`

CUDA programming



Kernels and threads

- A kernel is replicated into many threads
  - recap: threads are organized into blocks

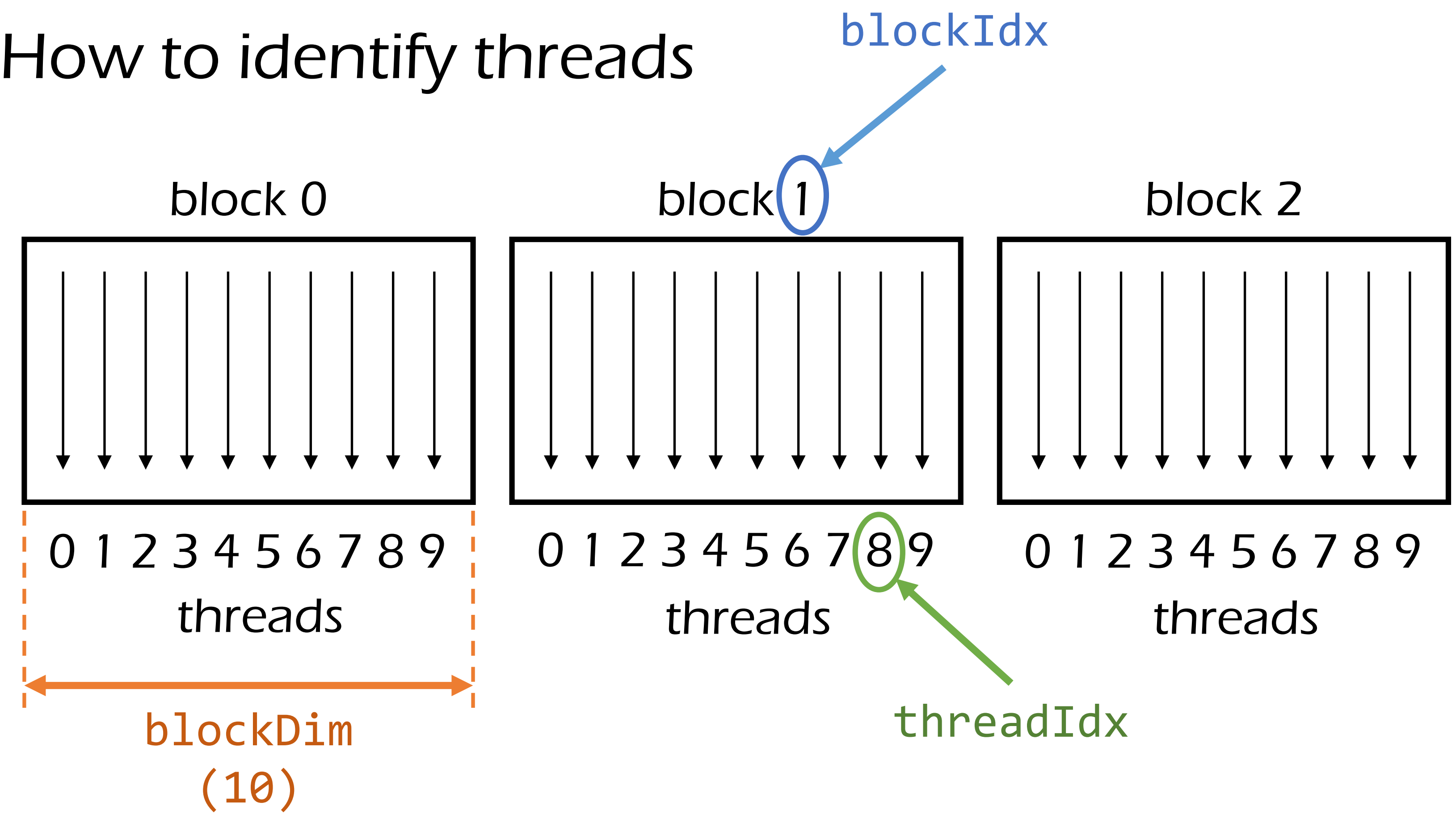
```
kernel<<<blocks, threads>>>(...)
```

number of blocks x number of threads per block

(according to hardware architecture)

Thread indexing

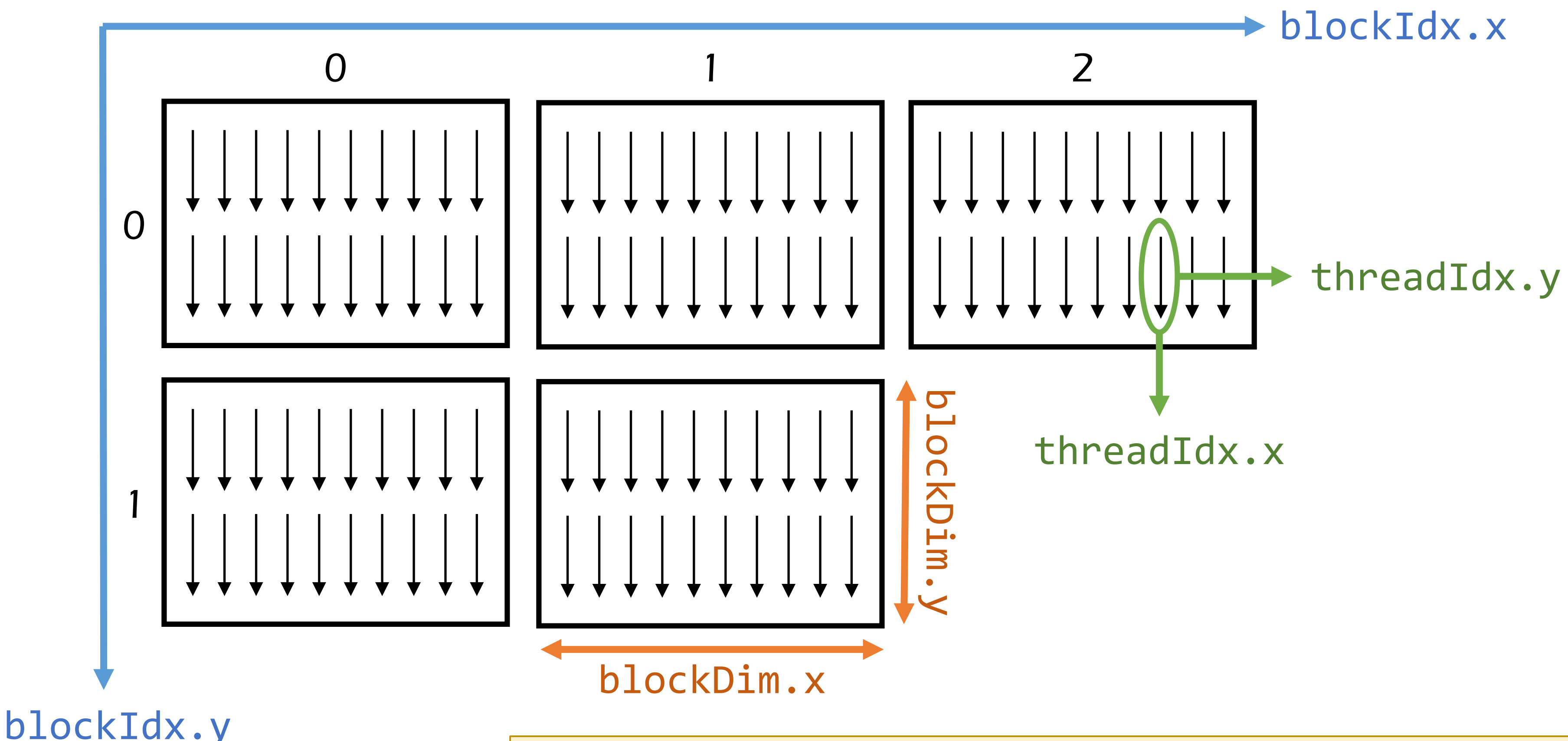
- How to identify threads




```
i = blockIdx * blockDim + threadIdx
```

Thread indexing

- Threads in a 2D grid




```
i = blockIdx.x * blockDim.x + threadIdx.x
j = blockIdx.y * blockDim.y + threadIdx.y
```

Thread indexing

- Threads in a 3D grid

```
i = blockIdx.x * blockDim.x + threadIdx.x
j = blockIdx.y * blockDim.y + threadIdx.y
k = blockIdx.z * blockDim.z + threadIdx.z
```

- CUDA supports 1D, 2D, 3D grids
  - in the following example we will use 1D

```
i = blockIdx.x * blockDim.x + threadIdx.x
```

Example

- *Squaring numbers*

```
void square(double* A, double* B, int N)
{
        for(int i=0; i<N; i++)
        {
                B[i] = A[i]*A[i];
        }
}
```

A	B
0.643	0.413
0.308	0.095
0.302	0.091
0.943	0.890
0.197	0.039
0.061	0.004
0.321	0.103
0.528	0.279
0.180	0.033
0.056	0.003
0.389	0.151
0.596	0.356

```
__global__ void kernel_square(double* d_A, double* d_B, int N)
{
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < N)
        {
                d_B[i] = d_A[i]*d_A[i];
        }
}
```

Example

- *Squaring numbers*

N	square()	kernel_square()
10^1	0.000000	0.000019
10^2	0.000000	0.000019
10^3	0.000004	0.000020
10^4	0.000042	0.000021
10^5	0.000393	0.000078
10^6	0.003152	0.000119
10^7	0.029324	0.000678
10^8	0.289635	0.006197

- 10^8 *is approaching the limit of GPU memory*
  - $8 \times 10^8 \sim$ 800 MB to store each array

CPU version

```
int N = ...

double* A = (double*)malloc(N*sizeof(double));
double* B = (double*)malloc(N*sizeof(double));

for(int i=0; i<N; i++)
{
        A[i] = ...
}
```

```
square(A, B, N);
```

```
free(A);
free(B);
```

```
void square(double* A, double* B, int N)
{
        for(int i=0; i<N; i++)
        {
                B[i] = A[i]*A[i];
        }
}
```

GPU version

```
double* d_A;
double* d_B;

cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));

cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);


...
```

GPU version

```
double* d_A;
double* d_B;

cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));

cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);

dim3 threads(128);
dim3 blocks((int)ceil((double)N/(double)threads.x));

kernel_square<<<blocks, threads>>>(d_A, d_B, N);


...
```

GPU version

```
double* d_A;
double* d_B;

cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));

cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);

dim3 threads(128);
dim3 blocks((int)ceil((double)N/(double)threads.x));

kernel_square<<<blocks, threads>>>(d_A, d_B, N);

...
```

```
__global__ void kernel_square(double* d_A, double* d_B, int N)
{
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < N)
        {
                d_B[i] = d_A[i]*d_A[i];
        }
}
```

GPU version

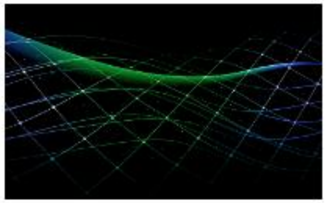
```
double* d_A;
double* d_B;

cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));

cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);

dim3 threads(128);
dim3 blocks((int)ceil((double)N/(double)threads.x));

kernel_square<<<blocks, threads>>>(d_A, d_B, N);

cudaMemcpy(B, d_B, N*sizeof(double), cudaMemcpyDeviceToHost);

cudaFree(d_A);
cudaFree(d_B);
```

Part 3
Libraries

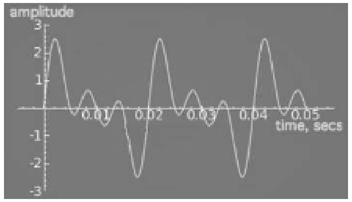
CUDA libraries

cuBLAS

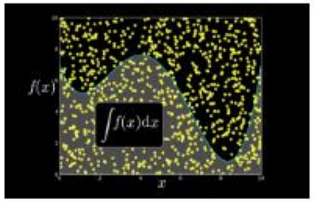
cuDNN



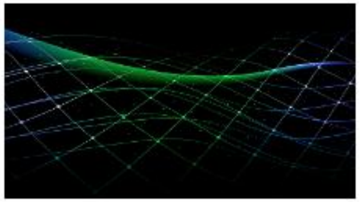
CUDA Math Library



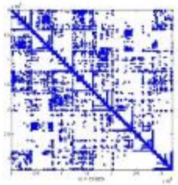
cuFFT



cuRAND

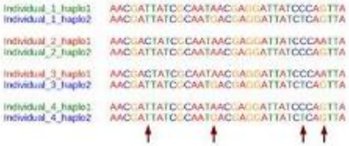


cuSOLVER

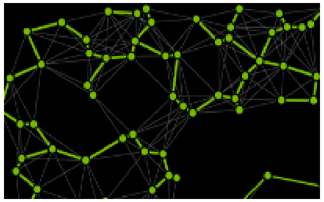


cuSPARSE

NPP



NVBIO



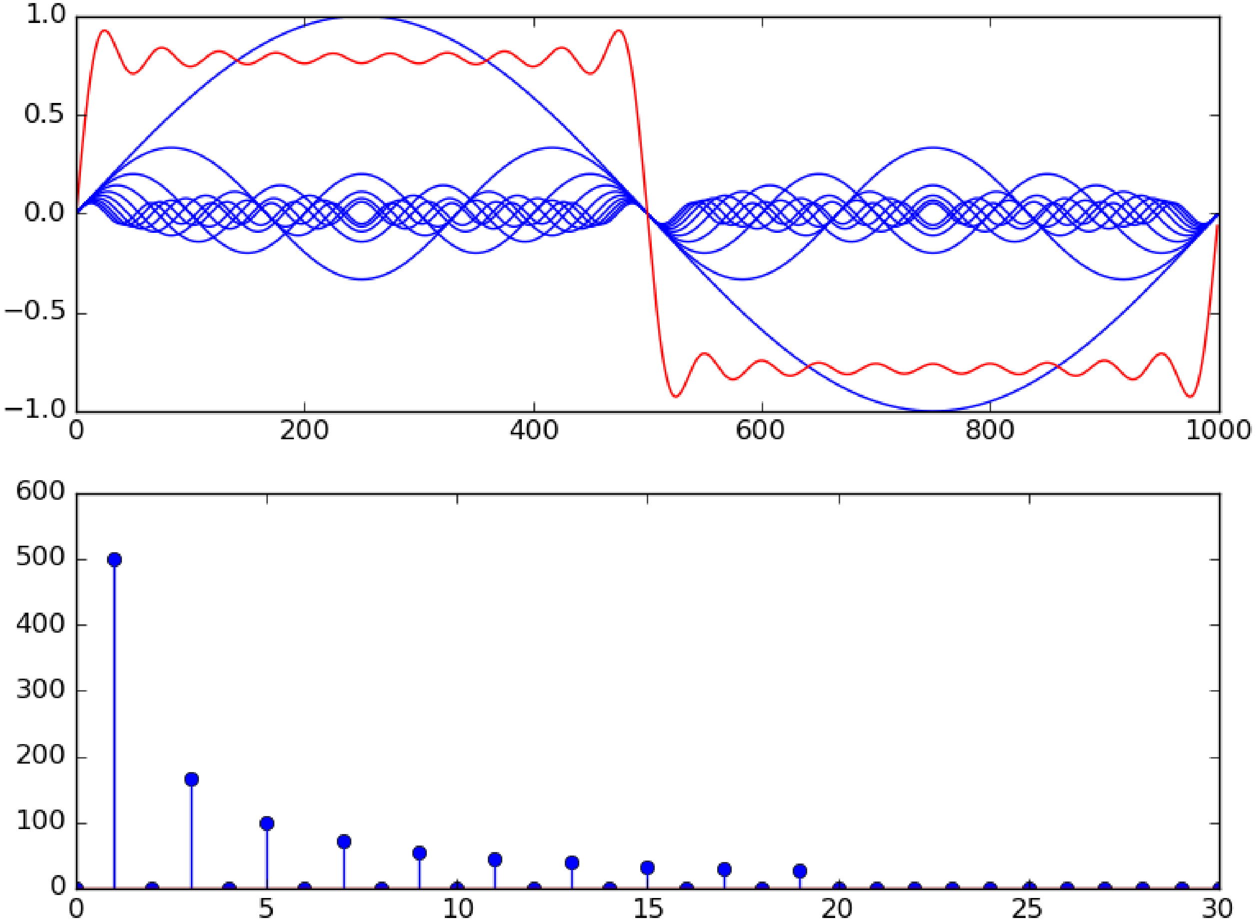
nvGRAPH

Thrust

...

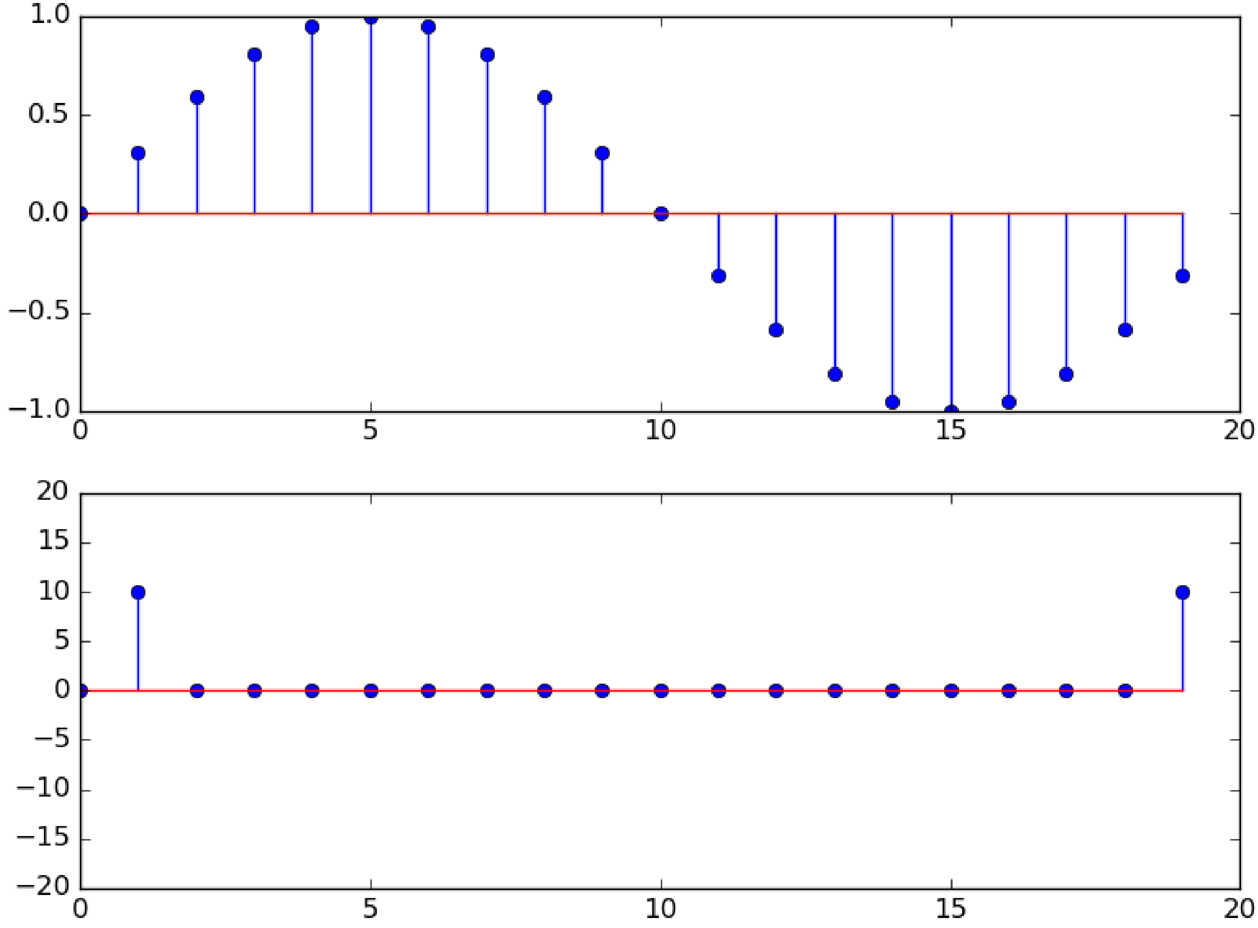
Fourier Transform

- Fourier Transform of a periodic signal

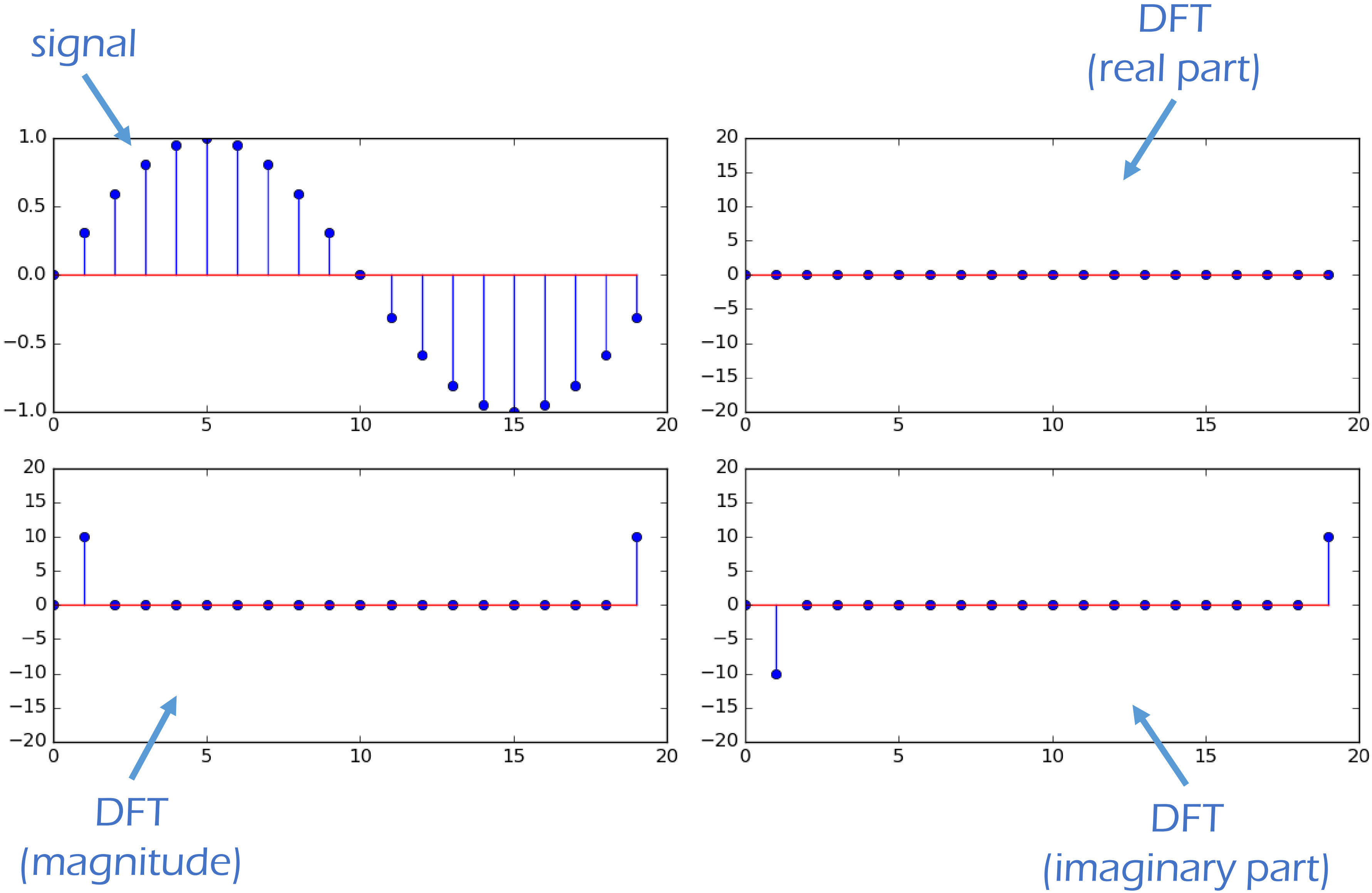


Discrete Fourier Transform (DFT)

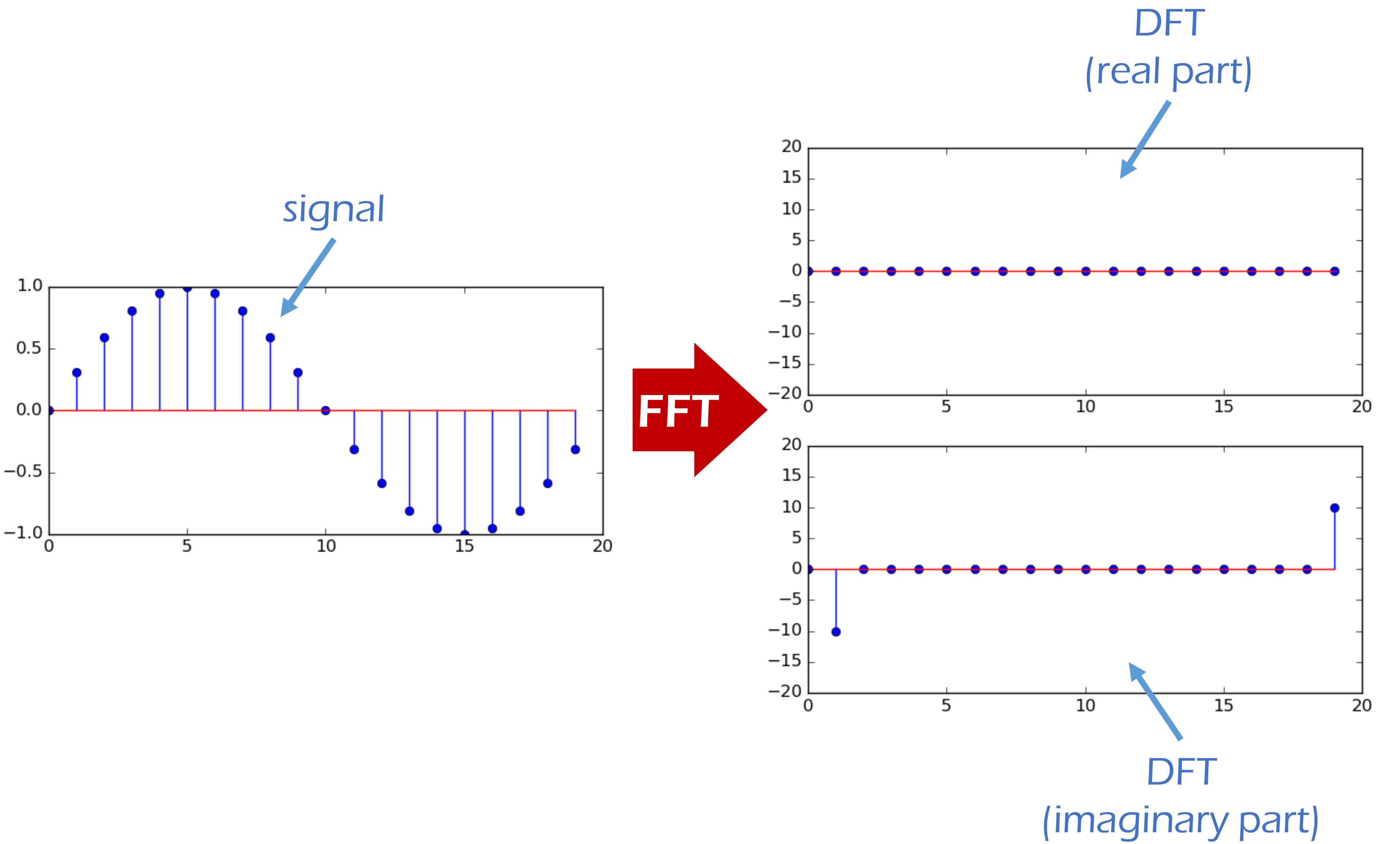
- Fourier Transform of a discrete and periodic signal



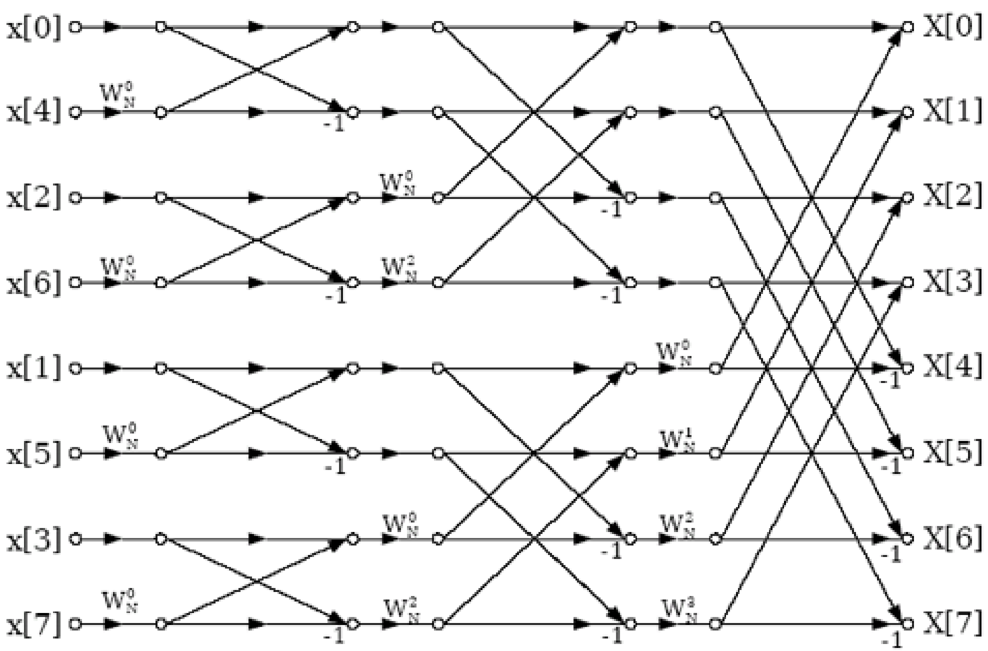
Discrete Fourier Transform (DFT)

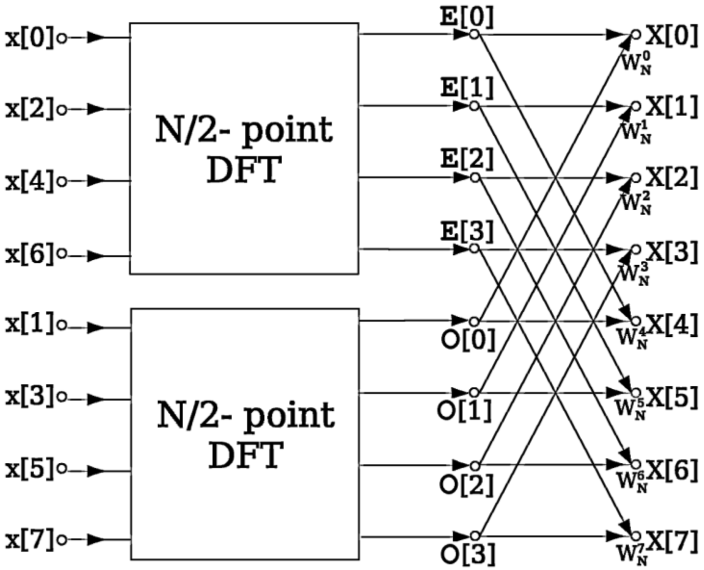


Fast Fourier Transform (FFT)



Fast Fourier Transform (FFT)





cuFFT

- The cuFFT library
  - performs complex-to-complex DFTs
  - requires setting up a “plan” (for a given N)
  - calculates FFT by executing the plan with data
  - the data (signal) must be in device memory
  - the result (FFT) will be in device memory
  - memory allocation and transfers

cuFFT

- Typical cuFFT programming
  - *use data type* `cufftComplex`
  - *allocate and copy signal* `cudaMalloc/cudaMemcpy`
  - *allocate space for result* `cudaMalloc`
  - *set up the plan* `cufftPlan1d()`
  - *execute the plan* `cufftExecC2C()`
  - *copy result to host* `cudaMemcpy`

Example

- Calculating the FFT of a sine wave
  - signal as A
  - FFT as B
  - both A and B complex
  - x, y are the real, imaginary parts

```
struct float2
{
        float x, y;
};

typedef float2 cufftComplex;
```

A.x	A.y	B.x	B.y
0.000	0.0	0.0	0.0
0.309	0.0	0.0	-10.0
0.588	0.0	0.0	0.0
0.809	0.0	0.0	0.0
0.951	0.0	0.0	0.0
1.000	0.0	0.0	0.0
0.951	0.0	0.0	0.0
0.809	0.0	0.0	0.0
0.588	0.0	0.0	0.0
0.309	0.0	0.0	0.0
0.000	0.0	0.0	0.0
-0.309	0.0	0.0	0.0
-0.588	0.0	0.0	0.0
-0.809	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-1.000	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-0.809	0.0	0.0	0.0
-0.588	0.0	0.0	0.0
-0.309	0.0	0.0	10.0

Example

- *Initializing*

```
int N = ...

cufftComplex* A = (cufftComplex*)malloc(N*sizeof(cufftComplex));
cufftComplex* B = (cufftComplex*)malloc(N*sizeof(cufftComplex));

for(int i=0; i<N; i++)
{
        A[i].x = sin(2*PI*i/N);
        A[i].y = 0.0;
}

...
```

A.x	A.y	B.x	B.y
0.000	0.0	0.0	0.0
0.309	0.0	0.0	-10.0
0.588	0.0	0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
[illegible]	[illegible]	0.0	0.0
-1.000	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-0.809	0.0	0.0	0.0
-0.588	0.0	0.0	0.0
-0.309	0.0	0.0	10.0

Example

- *Allocate and transfer data*

```
...

cufftComplex* d_A;
cufftComplex* d_B;

cudaMalloc(&d_A, N*sizeof(cufftComplex));
cudaMalloc(&d_B, N*sizeof(cufftComplex));

cudaMemcpy(d_A, A, N*sizeof(cufftComplex), cudaMemcpyHostToDevice);

...
```

A.x	A.y	B.x	B.y
0.000	0.0	0.0	0.0
0.309	0.0	0.0	-10.0
0.588	0.0	0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
-0.951	0.0	0.0	0.0
-1.000	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-0.809	0.0	0.0	0.0
-0.588	0.0	0.0	0.0
-0.309	0.0	0.0	10.0

Example

- Create plan and execute FFT

```
...

cufftHandle plan;
cufftPlan1d(&plan, N, CUFFT_C2C, 1)

cufftExecC2C(plan, d_A, d_B, CUFFT_FORWARD);

...
```

A.x	A.y	B.x	B.y
0.000	0.0	0.0	0.0
0.309	0.0	0.0	-10.0
0.588	0.0	0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
-0.309	0.0	0.0	0.0
-0.588	0.0	0.0	0.0
-0.809	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-1.000	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-0.809	0.0	0.0	0.0
-0.588	0.0	0.0	0.0
-0.309	0.0	0.0	10.0

Example

- Transfer results back to host

A.x	A.y	B.x	B.y
0.000	0.0	0.0	0.0
0.309	0.0	0.0	-10.0
0.588	0.0	0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
-0.809	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-1.000	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-0.809	0.0	0.0	0.0
-0.588	0.0	0.0	0.0
-0.309	0.0	0.0	10.0

```
...

cudaMemcpy(B, d_B, N*sizeof(cufftComplex), cudaMemcpyDeviceToHost);

for(int i=0; i<N; i++)
{
        printf("%f %f %f %f\n", A[i].x, A[i].y, B[i].x, B[i].y);
}

...
```

Example

- Free resources

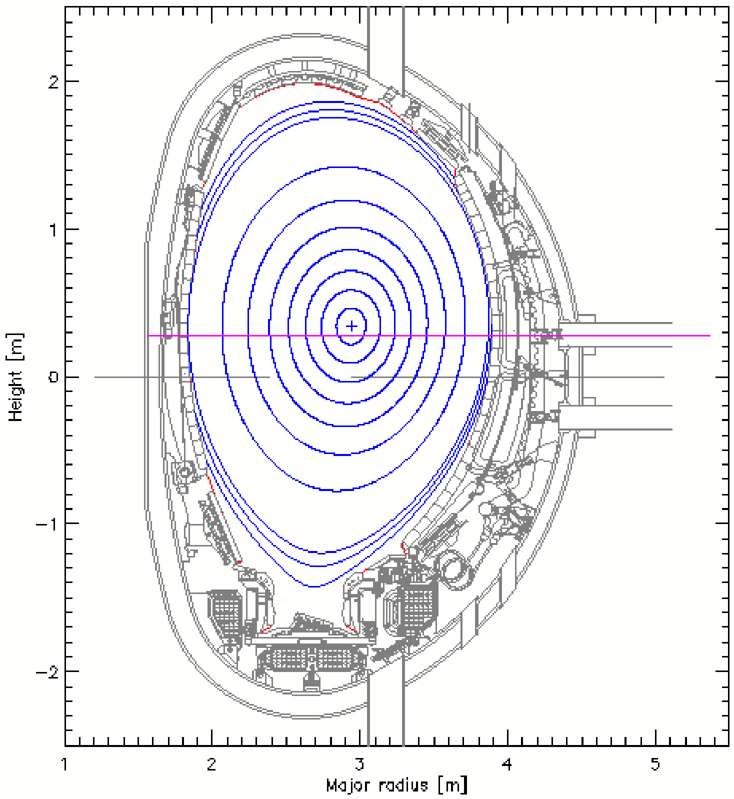
```
...

cufftDestroy(plan);

cudaFree(d_A);
cudaFree(d_B);

free(A);
free(B);

...
```

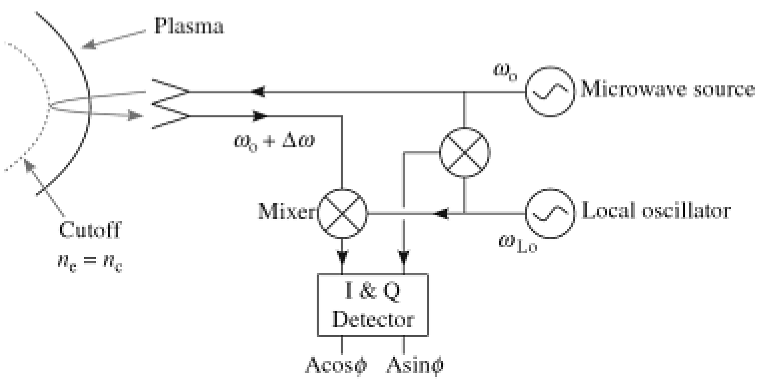
A.x	A.y	B.x	B.y
0.000	0.0	0.0	0.0
0.309	0.0	0.0	-10.0
0.588	0.0	0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
		0.0	0.0
-0.951	0.0	0.0	0.0
-1.000	0.0	0.0	0.0
-0.951	0.0	0.0	0.0
-0.809	0.0	0.0	0.0
-0.588	0.0	0.0	0.0
-0.309	0.0	0.0	10.0

Part 4
Applications

Plasma reflectometry

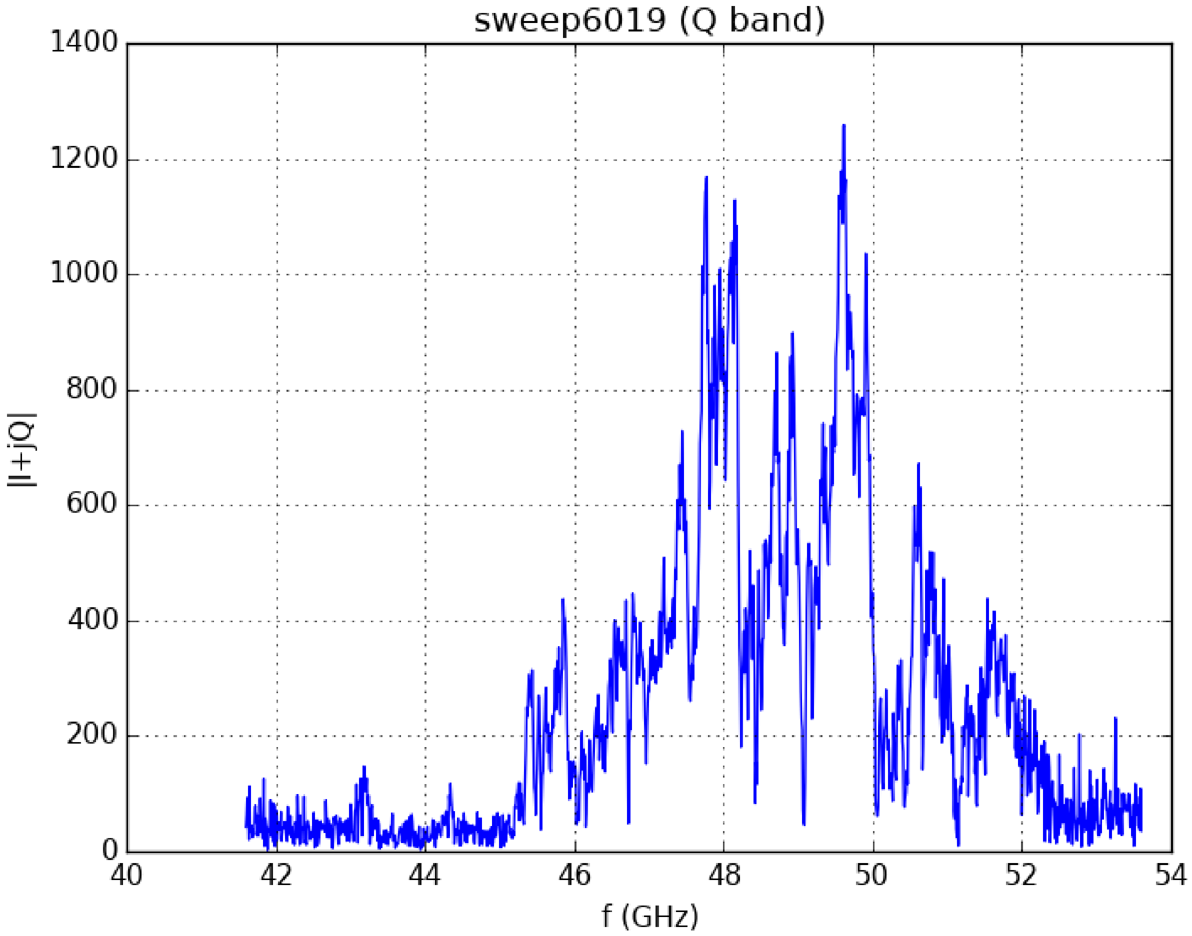
- Probing the plasma with microwave frequencies





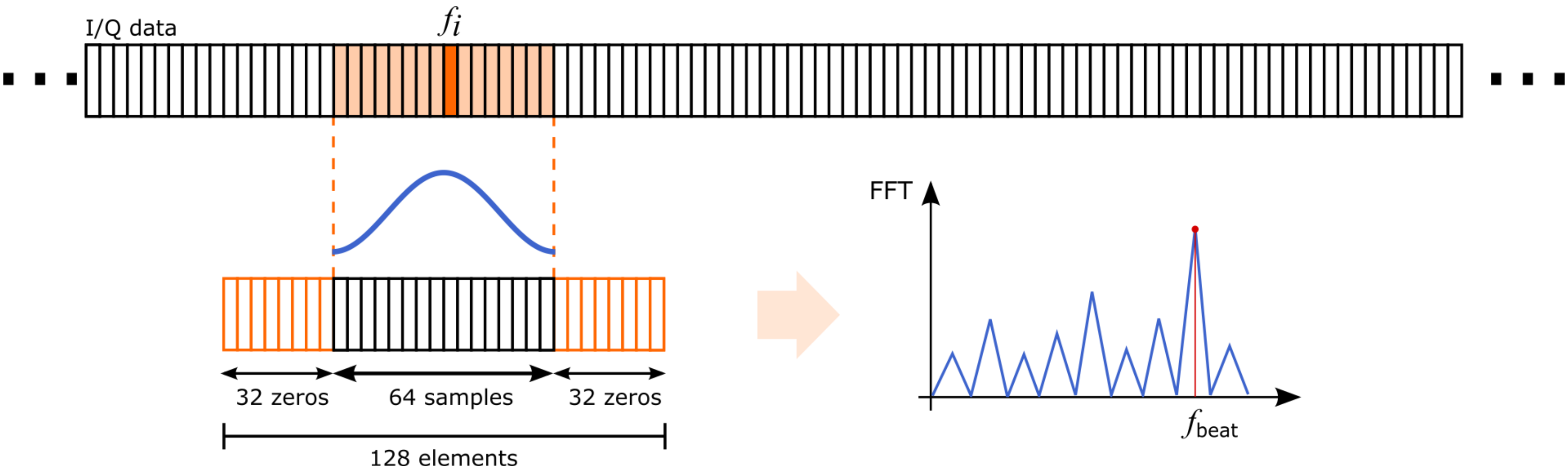
Plasma reflectometry

- *I/Q signal magnitude*



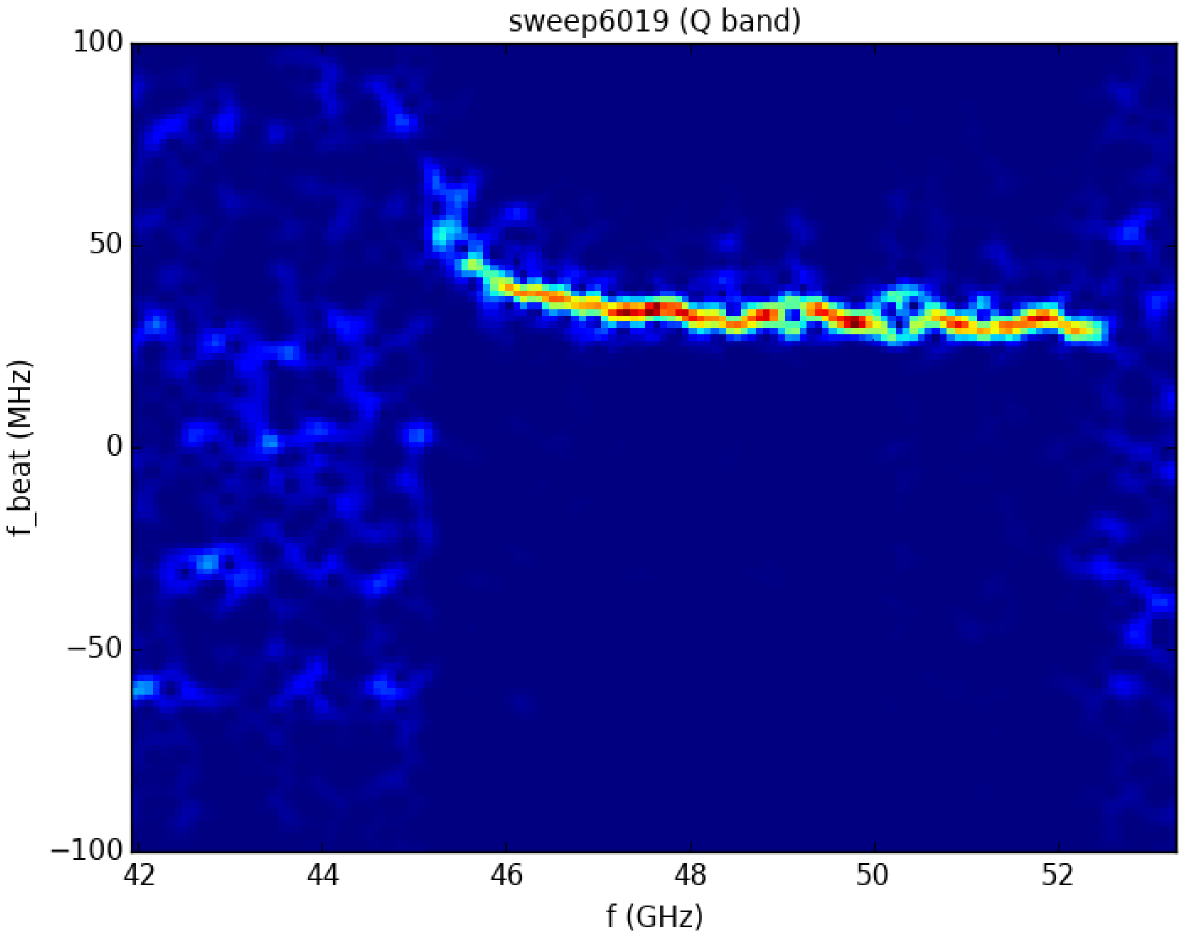
Plasma reflectometry

- Computing the beat frequencies
  - FFT on a sliding-window segment



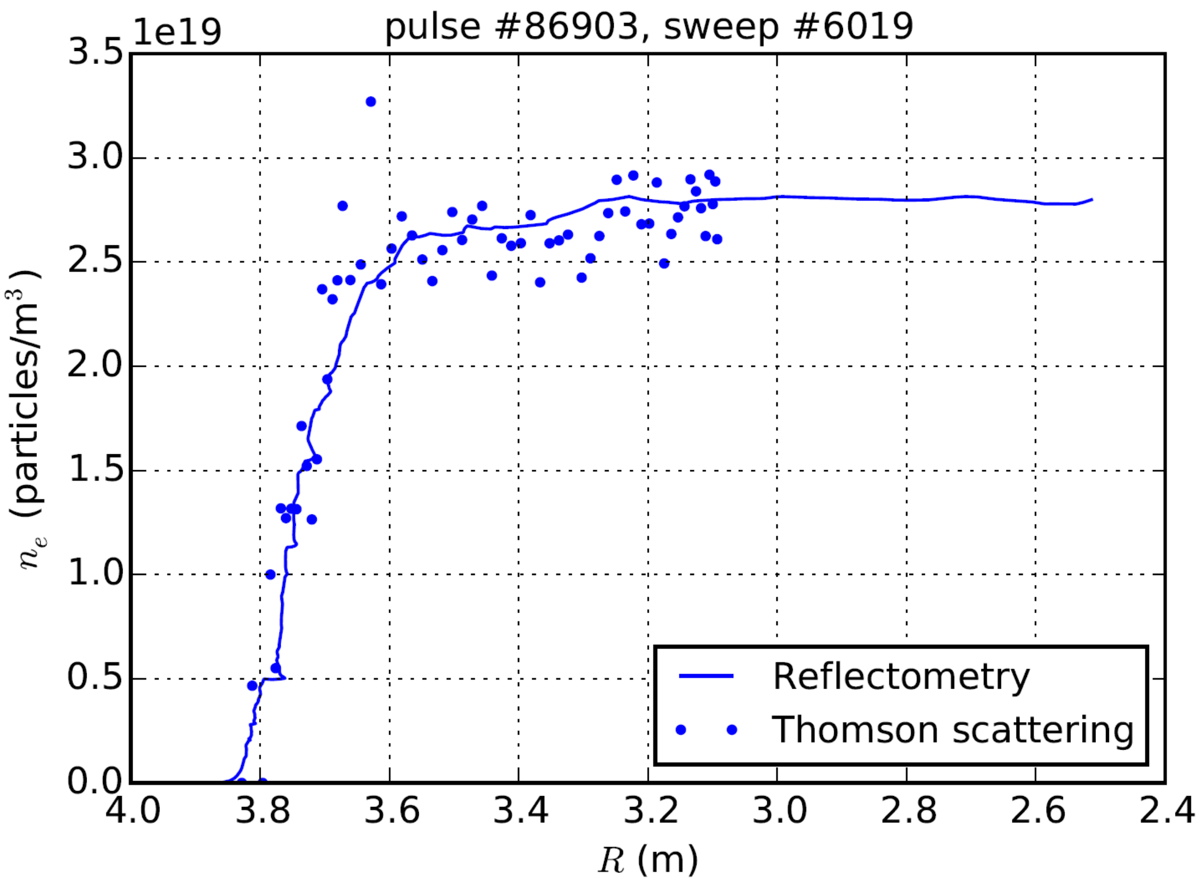
Plasma reflectometry

- Spectrogram



Plasma reflectometry

- Density profile



Plasma reflectometry

- Computing the density profile

I/Q signal

independent segments,
FFTs, peak detection, etc.

parallelizable

beat frequencies

iterative, sequential algorithm
where r_{i+1} depends on r_i

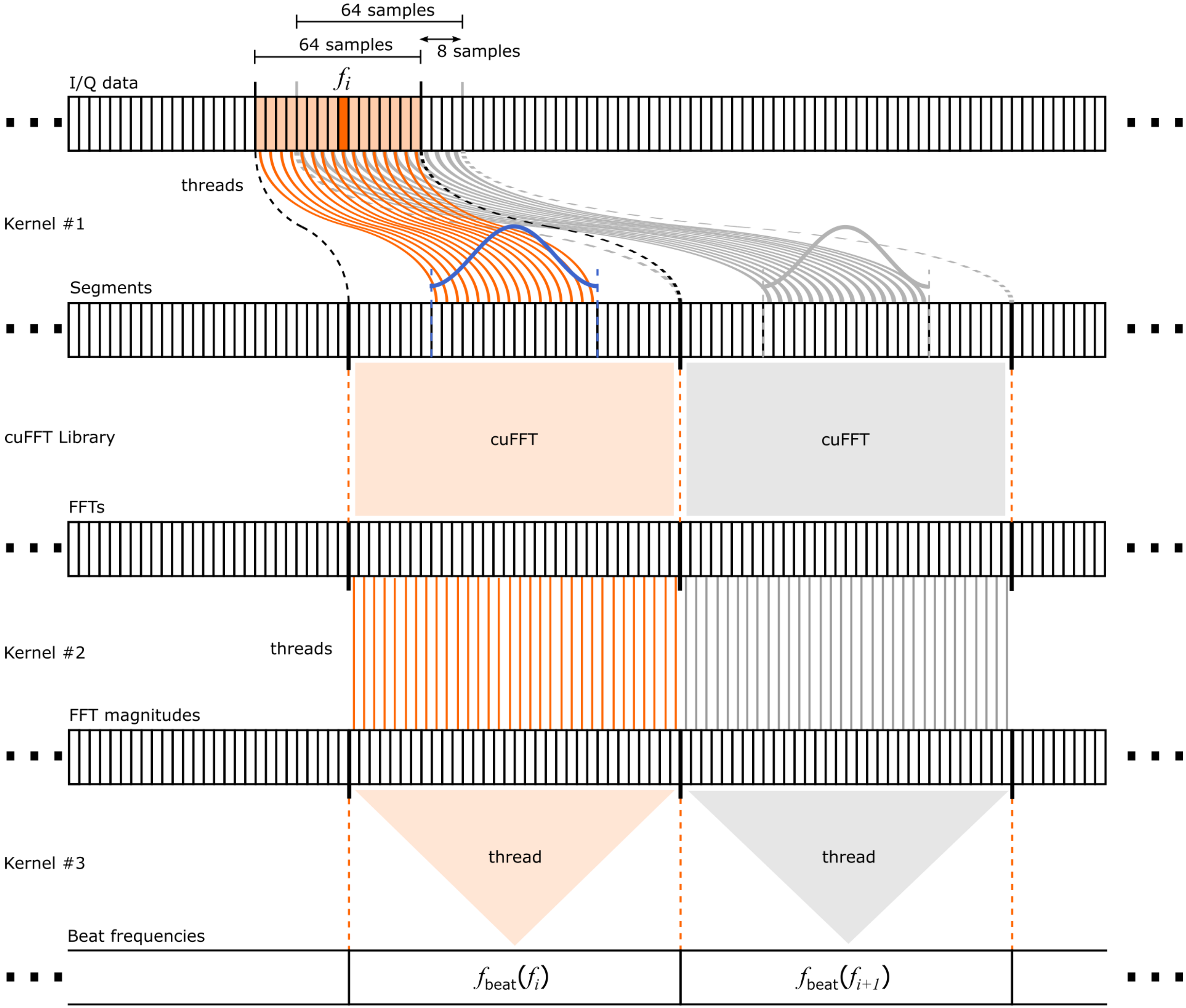
non-parallelizable

density profile

total
time

Plasma reflectometry

- Computing the beat frequencies with CUDA
  - build all segments at once (kernel 1)
  - compute the FFT for each segment (cuFFT)
  - compute the magnitude of each FFT (kernel 2)
  - find the peak of each FFT (kernel 3)



64 samples
64 samples
8 samples
I/Q data
f_i
threads
Kernel #1
Segments
cuFFT Library
cuFFT
cuFFT
FFTs
Kernel #2
threads
FFT magnitudes
Kernel #3
thread
thread
Beat frequencies
$f_{\text{beat}}(f_i)$
$f_{\text{beat}}(f_{i+1})$

Plasma reflectometry

- Computing the density profile – results

CPU: Intel Core i5-4690 @ 3.5 GHz GPU: NVIDIA GeForce GTX 750 Ti SM5.0 with 640 CUDA cores @ 1137 MHz			
Run time (s)	C version	CUDA version	Performance gain
sweep #6019	0.011204	0.003446	3.3x
sweep #10019	0.011178	0.003402	3.3x
sweep #73185	0.011247	0.003429	3.3x

Table 1: Processing time for three sample sweeps from pulse #86903

Conclusion

- GPU Computing is having a profound impact in many fields
  - speedups of 10x~100x are common
  - GPUs are more powerful than CPUs
  - scientific computing can leverage GPUs
- GPU parallelization requires a different mindset
  - use large arrays and many threads
  - check for existing/applicable libraries
  - optimize for hardware

Further reading

- CUDA programming
  - NVIDIA Corporation, “CUDA Toolkit Documentation”, http://docs.nvidia.com/cuda/
  - J. Cheng, M. Grossman, T. McKercher, “Professional CUDA C Programming”, Wrox Press, 2014
- Applications
  - D. R. Ferreira, P. J. Carvalho, H. Fernandes, L. Meneses, “Towards real-time density profile reconstruction with CUDA”, 1st EPS Conference on Plasma Diagnostics, Frascati, Italy, April 14-17, 2015
  - D. R. Ferreira, P. J. Carvalho, H. Fernandes, “Robust regression with CUDA and its application to plasma reflectometry”, Review of Scientific Instruments, vol. 86, no. 11, November 2015