CUDA PROGRAMMING

Yu-Hang Tang

June 23-26, 2015

CSRC, Beijing

WELCOME

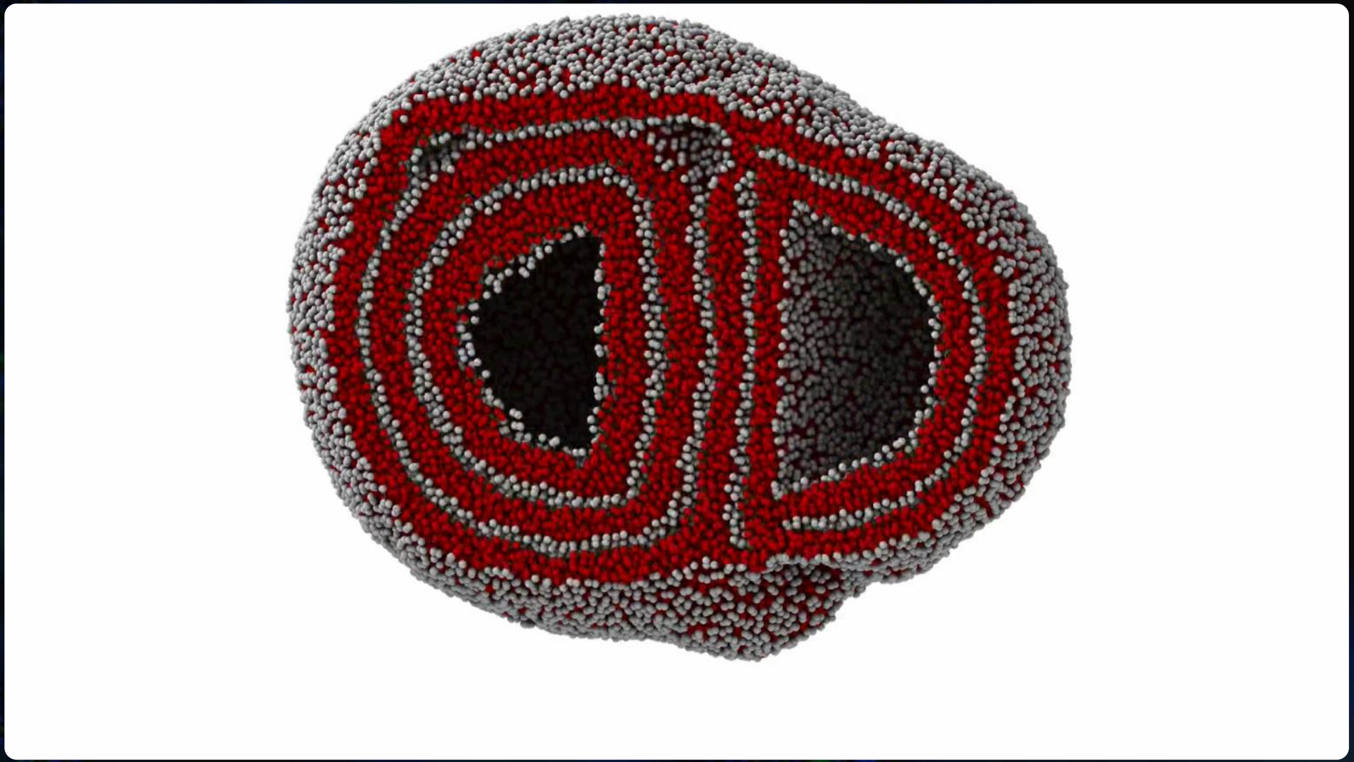
- slides at `/home/ytang/slides`
  - posted after each day
- exercise at `/home/ytang/exercise` `– make your own copy!`
- solution at `/home/ytang/solution`
  - permission granted after each exercise

Online CUDA API documentation

`http://docs.nvidia.com/cuda/index.html`

DEMO - SELF ASSEMBLY

- 134,217,728 particles
- 12,400,000 steps
- 1024 GPUs



GPU = GRAPHICS PROCESSING UNIT

- Traditional GPUs
  - Fixed-function pipelines
  - Earliest consumer application of multi-core architecture
- General-purpose GPUs
  - Fully programmable
  - massively parallel

MOTIVATION

WHO USED MY POWER

- Frequency vs. Power

$$P \propto V^2 f$$

- Higher voltages are necessary for higher frequency
- Rule of thumb: 2x frequency, 4x power consumption

WHO USED MY POWER

- **ILP**: Instruction Level Parallelism

```
a = b + c;
d = c + a;
f = c + e;
```

- Hardware ILP extraction logic
  - Pipelining
  - Superscalar
  - Branch prediction / Speculative execution

ILP extraction is expensive!

WHO USED MY POWER

- Data transfer is expensive
  - Computation appears to be **FREE**
- **Cache**: a smaller but faster memory storing copies of data in frequently used main memory locations.

	64-bit DP FMA	256-bit On-chip SRAM	256-bit Off-chip DRAM
Energy	20 pJ	50 pJ	16 nJ

Cache is Cash

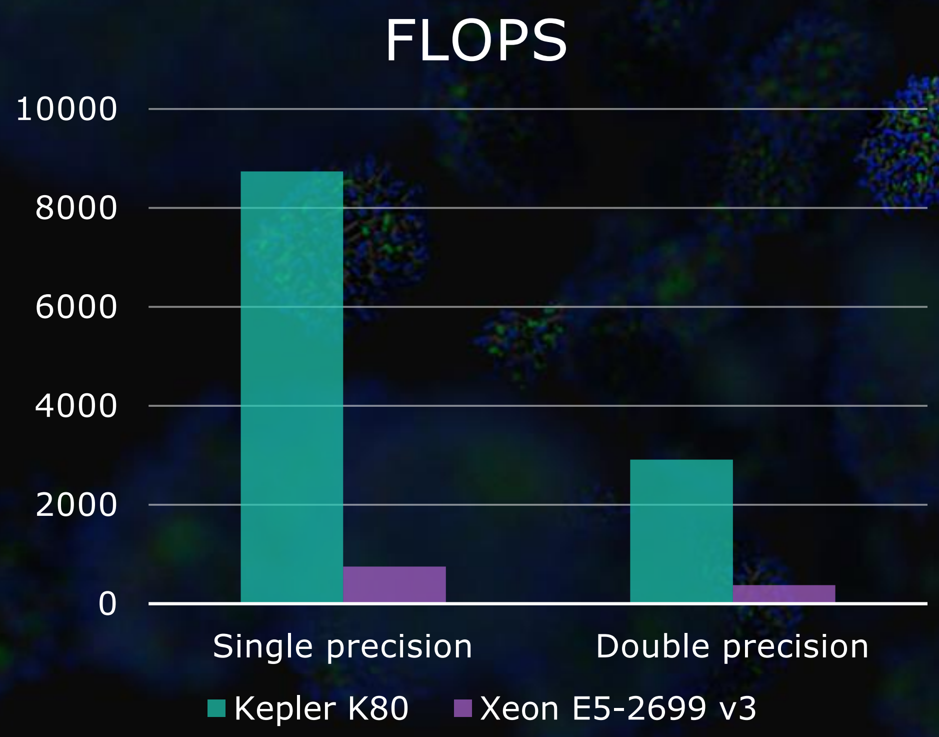
WHY GPU

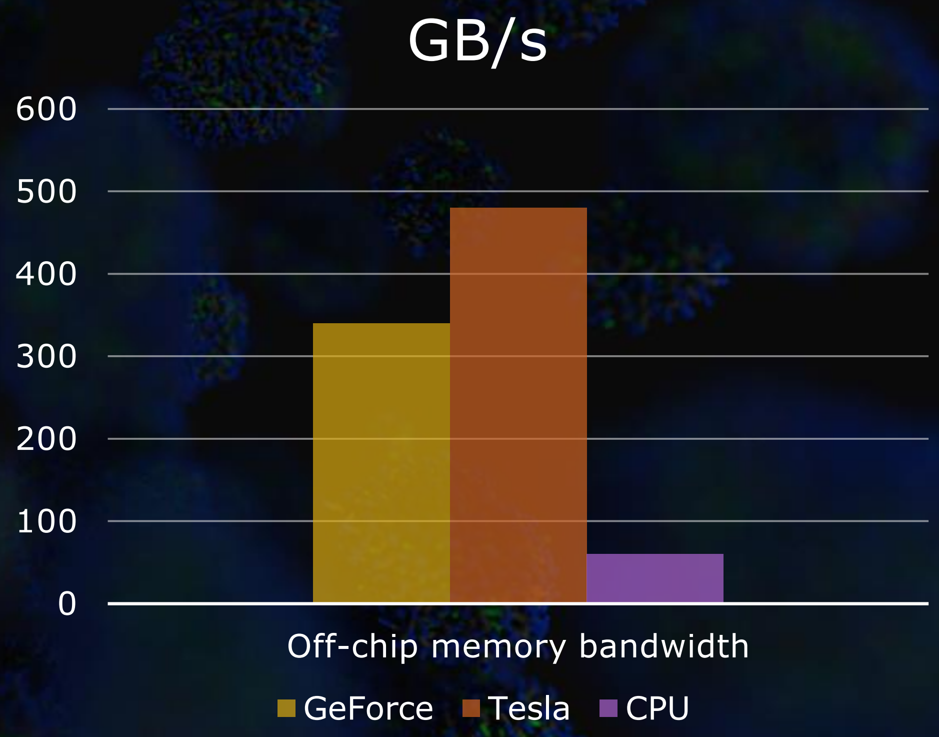
- **New Moore's Law: parallelism increases exponentially**
- Failure of Moore's Law
  - Frequency wall
  - Power wall
  - ILP wall
- Green Computing
  - FLOPS/watt matters
  - Green500.org
    - Kepler GPUs dominate

Rank	Name	GFLOPS/W	Configuration
1	L-CSC	5.3	ASUS ESC4000 FDR/G2S, Intel Xeon E5-2690v2 10C 3GHz, Infiniband FDR, **AMD FirePro S9150**
2	Suiren	4.9	ExaScaler 32U256SC Cluster, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, PEZY-SC
3	Tsubame-KFC	4.5	Intel Xeon E5-2620v2 6C 2.100GHz, Infiniband FDR, **NVIDIA K20x**
4	Storm1	4.0	Cray CS-Storm, Intel Xeon E5-2660v2 10C 2.2GHz, Infiniband FDR, **Nvidia K40m**
5	Wilkes	3.6	Intel Xeon E5-2630v2 6C 2.600GHz, Infiniband FDR, **NVIDIA K20**
6	iDataPlex DX360M4	3;5	Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband, **NVIDIA K20x**
7	HA-PACS TCA	3.5	Intel Xeon E5-2680v2 10C 2.800GHz, Infiniband QDR, **NVIDIA K20x**
8	Cartesius Accelerator Island	3.5	Bullx B515 cluster, Intel Xeon E5-2450v2 8C 2.5GHz, InfiniBand 4× FDR, Nvidia **K40m**
9	Piz Daint	3.2	Xeon E5-2670 8C 2.600GHz, Aries interconnect , **NVIDIA K20x**

A QUICK FACT

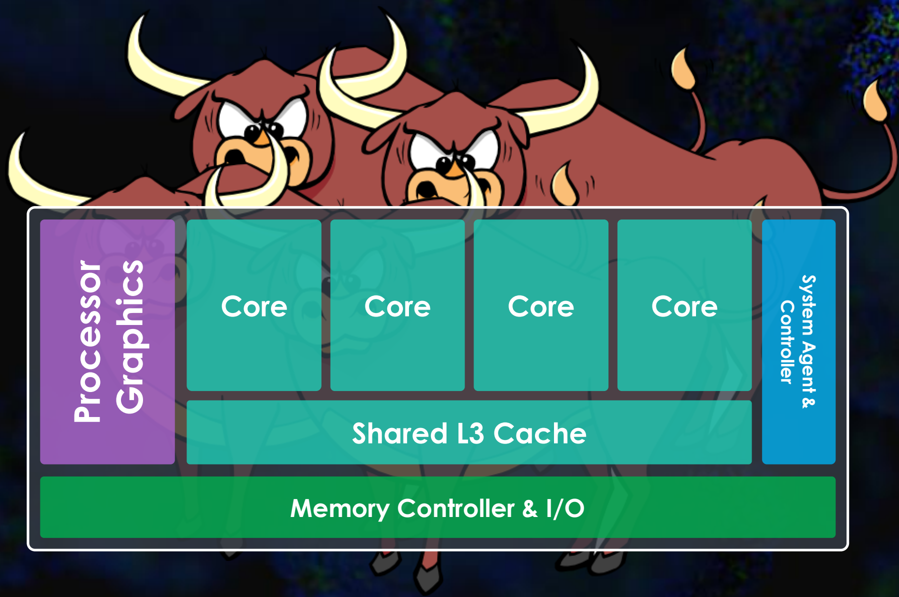
- GPU vs. CPU performance comparison



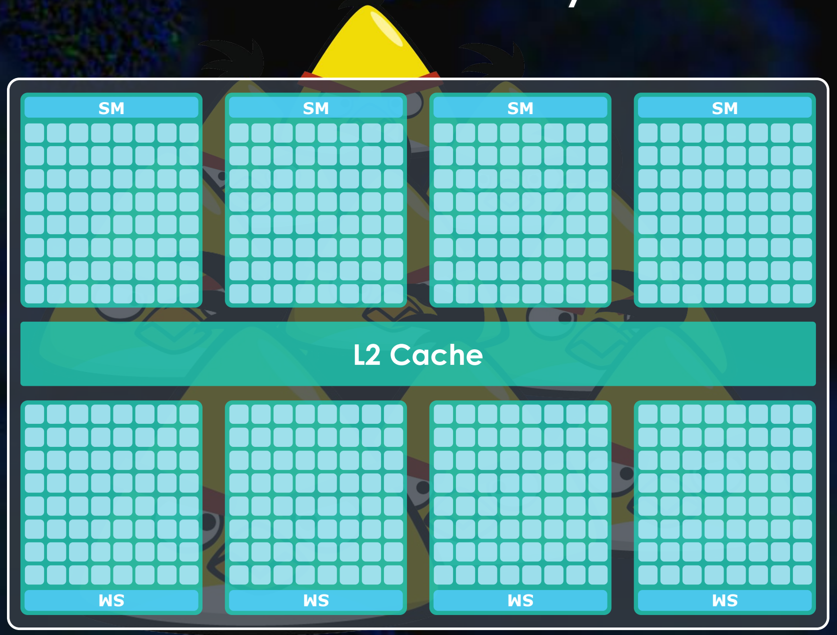


MORE FACTS

- Programming GPU is less trivial
- *"If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"* – Seymour Cray



CPU



GPU

GPU PROS & CONS 知己知彼，百战不殆

- ↗ Data Parallel
- ↗ Intensive FP Arithemtic
- ↗ Fine-grained parallelism

- ↘ Task Parallel
- ↘ Thread Dependencies
- ↘ Serial work
- ↘ Coarse-grained parallelism

BROWN

HOW TO USE GPUS

Language Extensions	C	C++	Fortran	…	
Directives	OpenACC	OpenMP4	…		
Libraries	cuBLAS	cuSPARSE	cuFFT	cuRAND	…
Scripting	PyCUDA	MATLAB	…		

MATLAB GPUARRAY

- gpuArray: Create arrays on GPU

```
m = magic(64);         % m is on CPU
M = gpuArray( m );     % M is on GPU now
```

- GPU-enabled functions

```
n = fft2( m );         % FFT on CPU
N = fft2( M );         % FFT on GPU
```

- Collect result

```
L = gather( N );       % transfer N back to CPU
find( abs( L – n ) > 1e-9 );
```

- Reference: http://www.mathworks.com/discovery/matlab-gpu.html

CUDA C/C++ OVERVIEW

- Subset of C++ with CUDA-specific extensions

Feature	Availability	Remark
Control flow	Y	
Built-in data types: char, int, float, etc.	Y	vector types: int2, float4…
Built-in operators	Y	including new/delete
Overloading	Y	
Object-oriented programming	Y	Inheritance virtual methods
Templates	Y	
C standard library	Partial	printf, malloc, free supported
C++ standard library	N	
C++11 extensions	**Y**	variadic template, lambda

EXAMPLE #0: HELLO WORLD

- Compilation command

```
nvcc -arch=sm_35 hello.cu -o hello.x
```

- -arch: specifies target compute capability

1.0 → 1.1 → 1.2 → 1.3 → 2.0 → 2.1 → 3.0 → **3.5*** → 5.0 → ...

```
#include <cstdio>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void hello_gpu() {
    printf( "\"Hello, world!\", says the GPU.\n" );
}

void hello_cpu() {
    printf( "\"Hello, world!\", says the CPU.\n" );
}

int main( int argc, char **argv )
{
    hello_cpu();
    hello_gpu<<< 1, 1>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

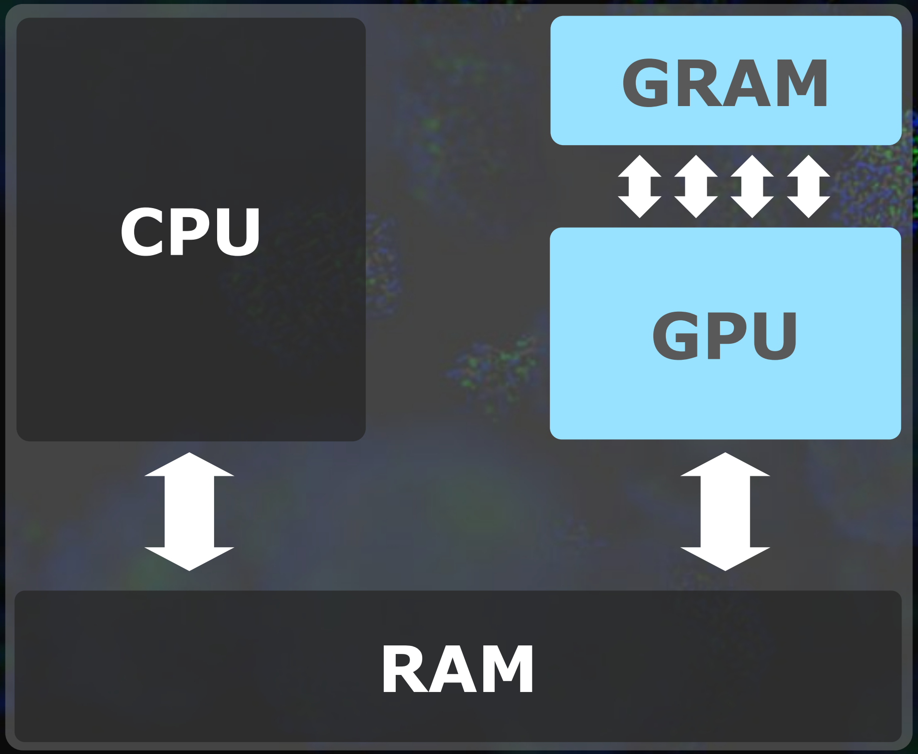
BEHIND THE SCENE

- NVCC compiler
  - a wrapper around a host compiler, e.g.
    - GCC
    - Intel Compiler
    - MSC
  - -arch: specify compute capability
  - -ccbin: specify host compiler (default to g++ on Linux)
- CUDA runtime
  - Automatic initialization
  - Asynchronous Execution

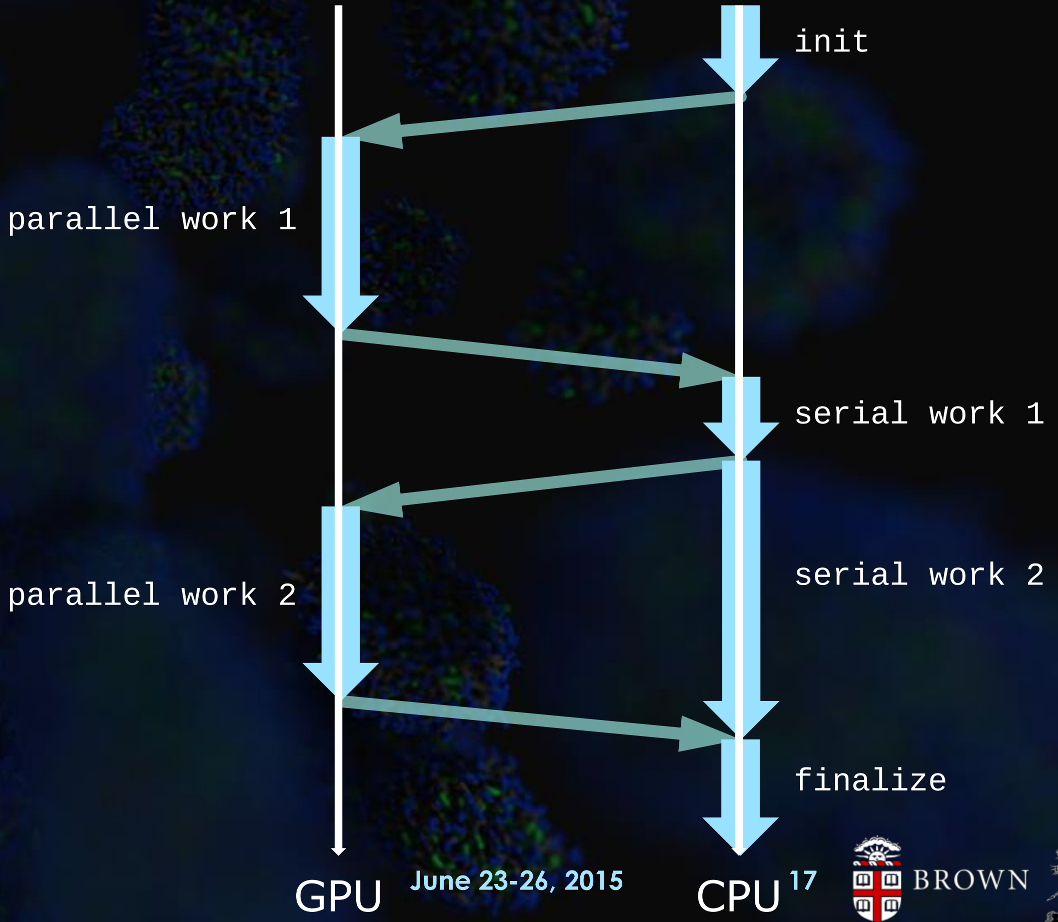
```
#include <cstdio>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void hello_gpu() {
    printf( "\"Hello, world!\", says the GPU.\n" );
}

void hello_cpu() {
    printf( "\"Hello, world!\", says the CPU.\n" );
}

// host code entrance
int main( int argc, char **argv )
{
    hello_cpu();
    hello_gpu<<< 1, 1>>>();
    cudaDeviceSynchronize();
}
```

PROGRAM FLOW

- Hardware



- Software

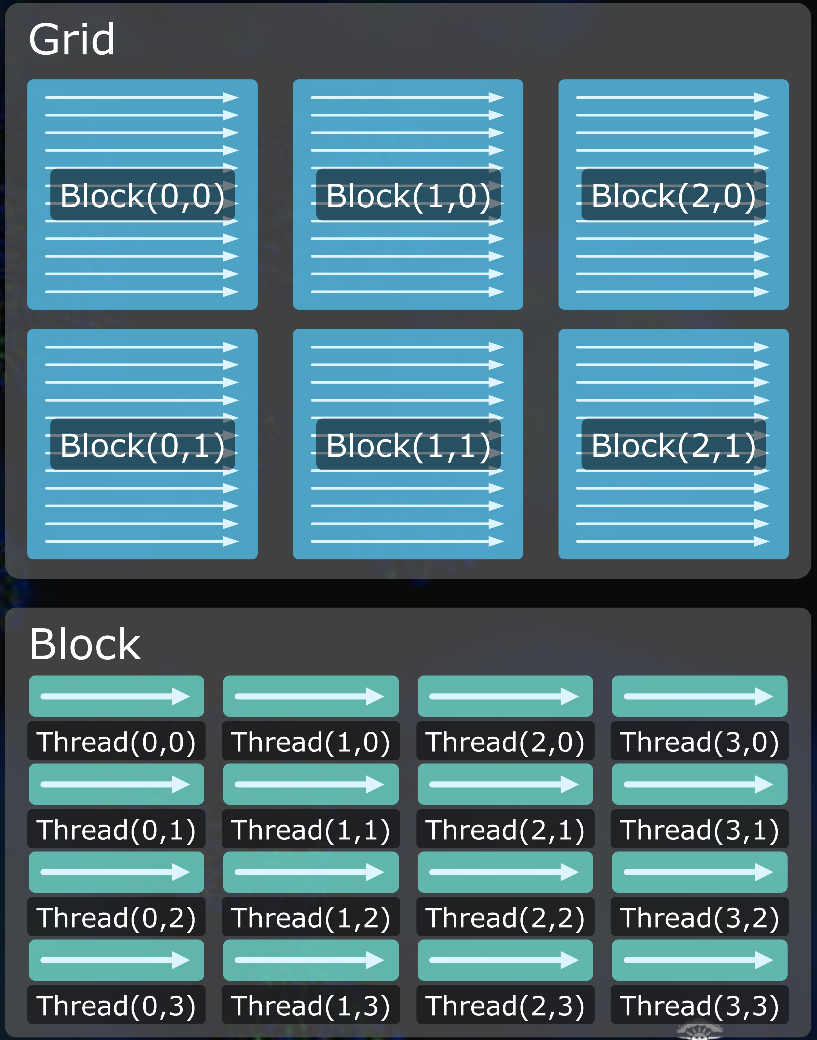


THREAD HIERARCHY

- To manage thousands of threads

  divide et impera

- **Block**: a group of at most 1024 threads
  - Why: hardware limitation
  - Threads arranged in 1D/2D/3D
  - threads share resources within a block:
    - register, shared memory, etc.
  - synchronization available
- **Grid**: group of blocks
  - 1D/2D/3D arrangement
  - Practically unlimited number of blocks
  - No synchronization between blocks



KERNELS

- Executed in parallel by many CUDA threads
  - begin with the `__global__` qualifier

```
// each thread will print once
__global__ void hello() {
    printf( "\"Hello, world!\", says the GPU.\n" );
}
```

  - Need to configure parallelism when launching

```
kernel<<<numBlocks,threadsPerBlock>>>(args);
```

  - Each thread has an id defined in built-in `threadIdx` variable.
  - Must be launched from host

FUNCTION QUALIFIER

- **Kernels:** `__global__`
- **Device functions**: `__device__`
  - callable from kernels
- Host functions: `__host__`
  - functions without a qualifier are default to be host functions
- A function can be both `__device__` and `__host__`
  - No qualifier can be used together with `__global__`

```
__inline__ __host__ __device__ double force( double x ) {
    return -0.5 * K * ( x - x0 );
}
```

PARALLELISM CONFIGURATION

- `struct dim3 { uint x,y,z; };`
    - CUDA built-in type for describing the thread configuration
- Built-in variables for each thread:

`threadIdx`	thread index within the current block
`blockIdx`	block index within the current grid
`blockDim`	block size
`gridDim`	grid size, i.e. number of blocks in each dimension

- Launch configuration <<<numBlocks, threadsPerBlock>>>
    - <<<uint,uint>>>
    - <<<dim3,dim3>>>

GPU GLOBAL MEMORY

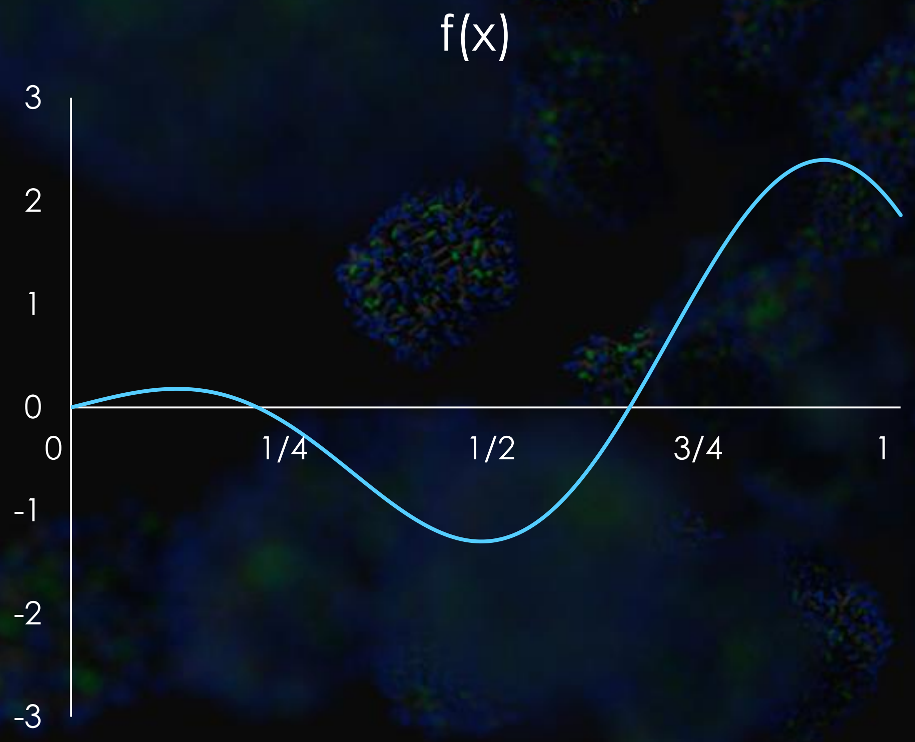
- GPU has its own on-board memory
  - Accessing CPU RAM is another story
- Allocatable from host/device
  - `cudaError_t cudaMalloc ( void** devPtr, size_t size );`
  - `cudaError_t cudaFree ( void* devPtr ) ;`
  - `device-side malloc/new/free/delete`
- Accessible from device

```
ptr[ index ] = value;
```

- Copiable from host
  - `cudaError_t cudaMemcpy ( void* dst, const void* src, size_t count, cudaMemcpyKind kind );`
  - `cudaError_t cudaMemset ( void* devPtr, int  value, size_t count );`

EXAMPLE 1A: TRANSCENDENTAL FUNCTIONS

- $f(x) = \sin x \cdot \cos 7x \cdot e^x, x \in [0,1]$



```
#include <cstdio>
#include <iostream>
#include <vector>
#include <limits>

#include <cuda.h>
#include <cuda_runtime.h>
#include <omp.h>

#include "../util/util.h"

__inline__ __host__ __device__ double f( double x ) {
          return sin( 2.0*x ) * cos( 7.0*x ) * exp( x );
}

__global__ void evaluate( double *y, const int n )
{
    int i = global_thread_id();
    y[i] = f( (double)i / (double)n );
}

// host code entrance
int main( int argc, char **argv )
{
    int N = 128 * 1024 * 1024;

    // timing register
    double t_CPU_0, t_CPU_1, t_GPU_0, t_GPU_1, t_GPU_2;
    // allocate host memory
    double *hst_y, *ref_y;
    hst_y = new double[N];
    ref_y = new double[N];
    // allocate device memory
    double *dev_y;
    cudaMalloc( &dev_y, N * sizeof( double ) );

    t_GPU_0 = get_time();

    // do computation on GPU
    evaluate <<< N / 1024, 1024 >>> ( dev_y, N );
    cudaDeviceSynchronize();

    t_GPU_1 = get_time();

    // copy result back to CPU
    cudaMemcpy( hst_y, dev_y, N * sizeof( double ),
cudaMemcpyDefault );

    t_GPU_2 = get_time();

    t_CPU_0 = get_time();

    // calculate reference value
          #pragma omp parallel for
    for( int i = 0; i < N; i++ ) ref_y[i] = f( (double)i /
(double)N );

    t_CPU_1 = get_time();

    // compare
    bool match = true;
    for( int i = 0; i < N; i++ ) {
        match = match &&
                ( fabs( ref_y[i] - hst_y[i] ) < 8 *
std::numeric_limits<double>::epsilon() );
    }

    // output
    std::cout << "Computation on CPU took " << t_CPU_1 -
t_CPU_0 << " secs." << std::endl;
    std::cout << "Computation on GPU took " << t_GPU_1 -
t_GPU_0 << " secs." << std::endl;
    std::cout << "Data transfer from GPU took " << t_GPU_2
- t_GPU_1 << " secs." << std::endl;
    std::cout << "CPU/GPU result match: " << ( match ?
"YES" : "NO" ) << std::endl;

    // free up resources
    delete [] hst_y;
    delete [] ref_y;
    cudaDeviceReset();
}
```

EXAMPLE 1B: AXPY

- AXPY = $a\,x + y$
  - a: scalar
  - x, y: N-by-1 vectors

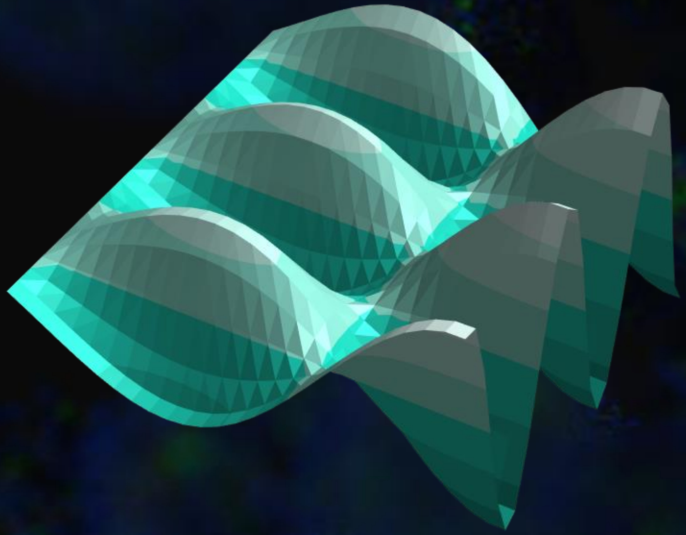
?

EXAMPLE 2: HELLO WORLD 2D

```
#include <cstdio>
#include <cuda.h>
#include <cuda_runtime.h>

__global__ void hello_gpu()
{
    printf( "\"Hello, world!\", says GPU block (%d,%d) thread (%d,%d).\n",
            blockIdx.x, blockIdx.y, threadIdx.x, threadIdx.y );
}

void hello_cpu()
{
    printf( "\"Hello, world!\", says the CPU.\n" );
}

// host code entrance
int main( int argc, char **argv )
{
    hello_cpu();
    printf( "launching 2x2 blocks each containing 4 threads\n" );
    hello_gpu <<< dim3( 2, 2, 1 ), dim3( 4, 1, 1 ) >>>();
    cudaDeviceSynchronize();
    printf( "launching 2x2 blocks each containing 2x2 threads\n" );
    hello_gpu <<< dim3( 2, 2, 1 ), dim3( 2, 2, 1 ) >>>();
    cudaDeviceSynchronize();
    cudaDeviceSynchronize();
}
```

EXAMPLE 3A: 2D FUNCTION

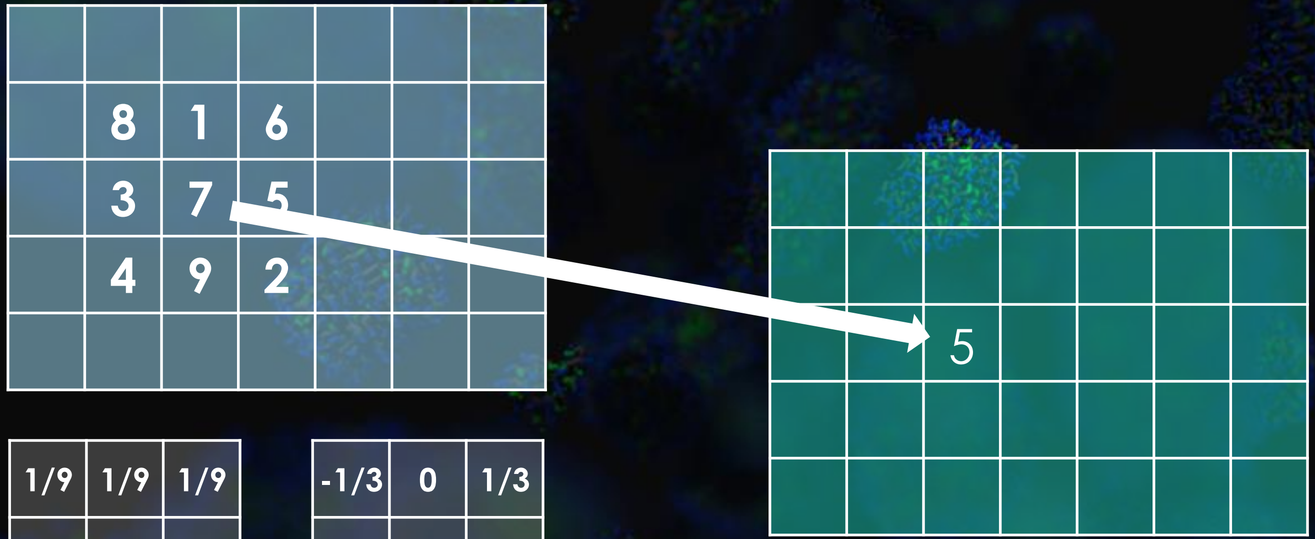
- $f(x,y) = \sin 5x \cdot \cos 16y \cdot e^x,\ \ x \in [0,1], y \in [0,1]$



?

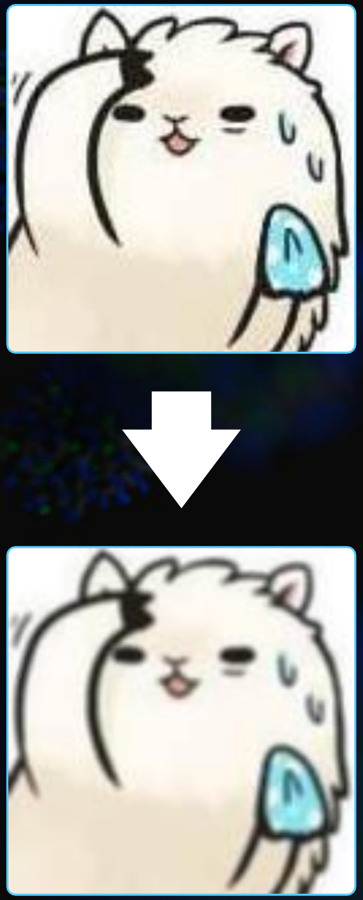
EXAMPLE 3B: IMAGE FILTERING

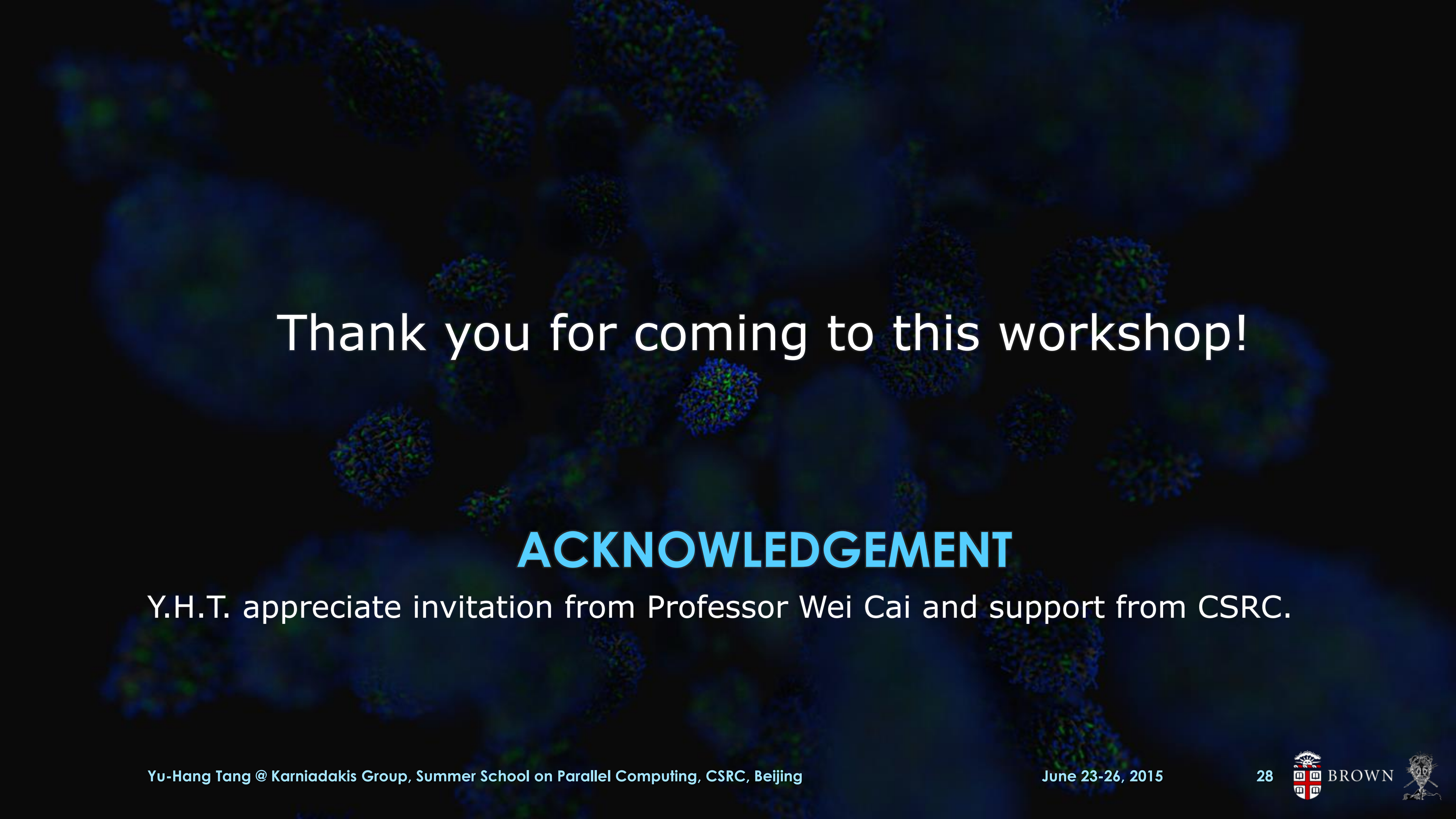
- 2D convolution



1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

-1/3	0	1/3
-1/3	0	1/3
-1/3	0	1/3





Thank you for coming to this workshop!

ACKNOWLEDGEMENT

Y.H.T. appreciate invitation from Professor Wei Cai and support from CSRC.