GPU Computing introduction, library and examples

Rogy Liu(刘文志)
HPC Developer Technology, NVIDIA

Links to get started

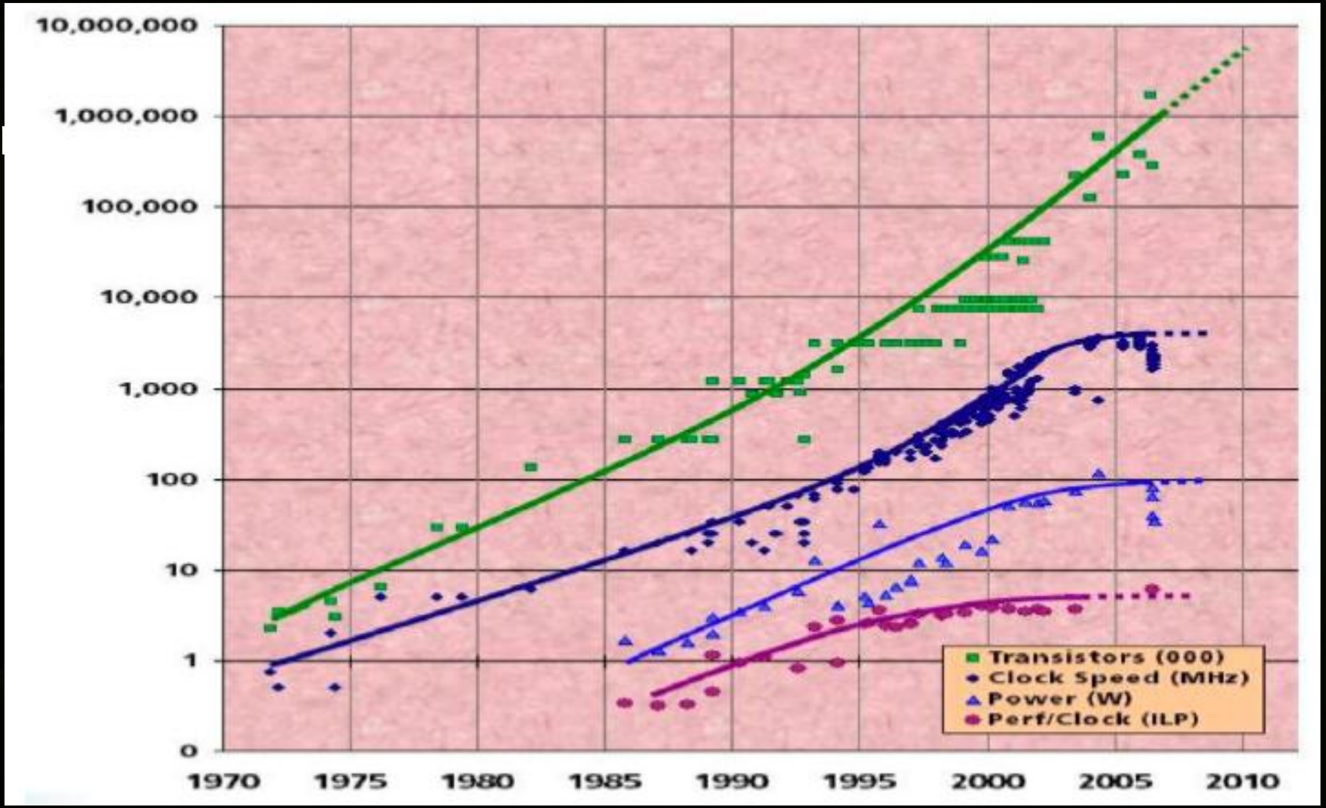
- Get CUDA: www.nvidia.com/getcuda
- Nsight IDE: www.nvidia.com/nsight
- Programming Guide/Best Practices…
  - docs.nvidia.com
- Questions:
  - http://cudazone.nvidia.cn/forum/forum.php
  - http://blog.sina.com.cn/u/3260774114
- General: www.nvidia.com/cudazone

Overview

- Heterogeneous Parallel Computing
- GPU Programming
- GPU Accelerated library
- Some Examples
- Some successful stories if possible

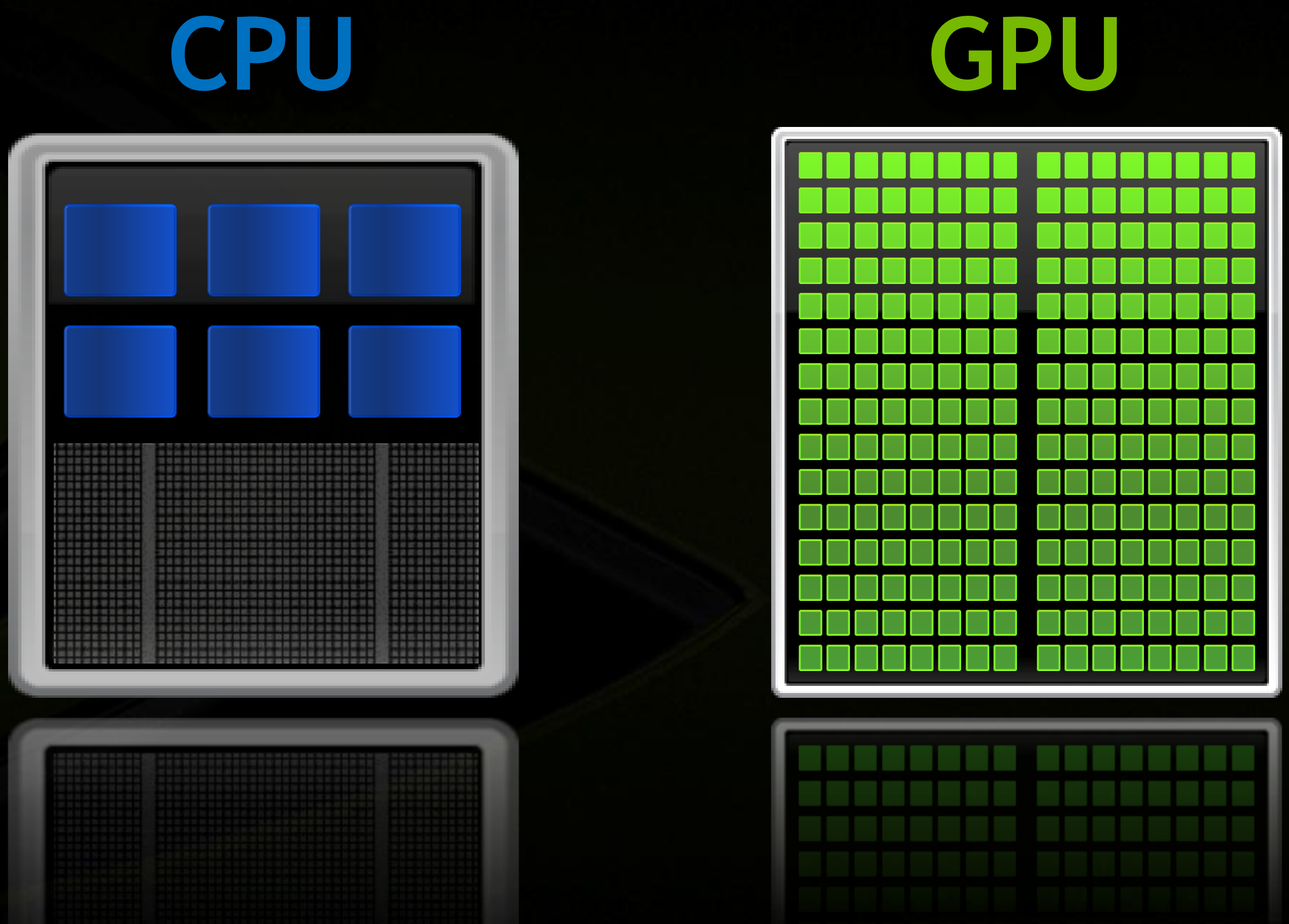
New Era of Computing: Parallel Computing

- **Modified Moore's law**
  - **Only transistor density is scaling, clock frequency (perf/clock) no longer increases linearly**
  - **general purpose single processor hits the power**

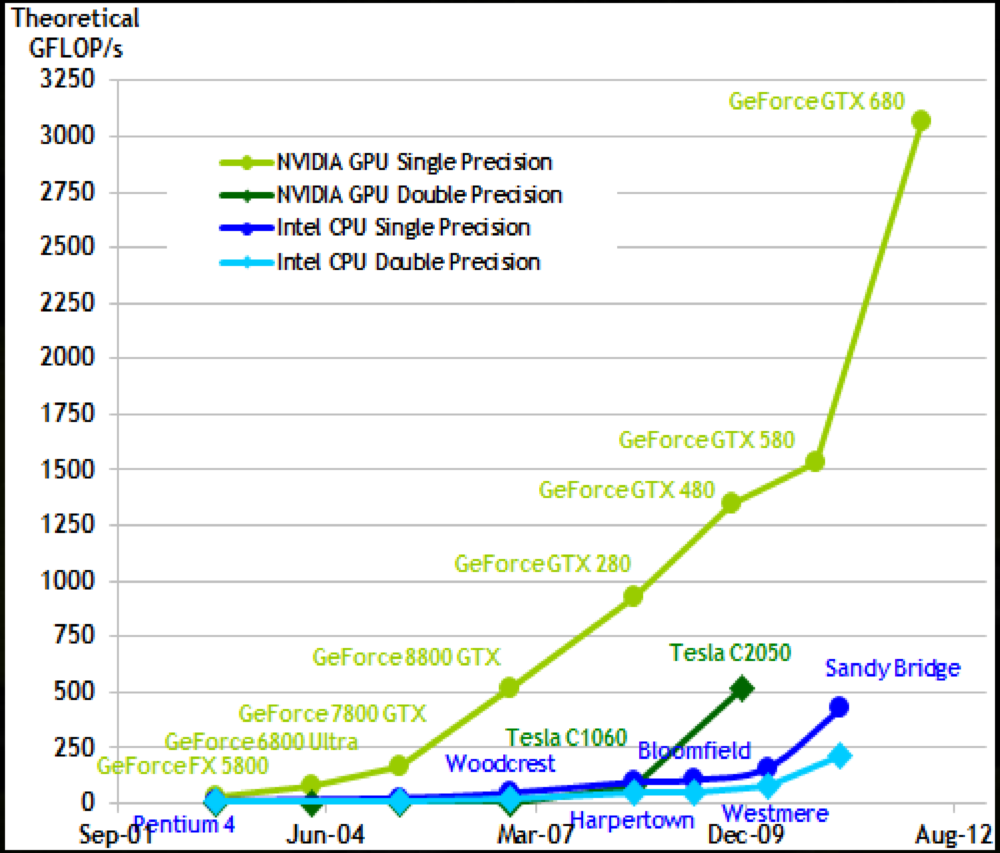


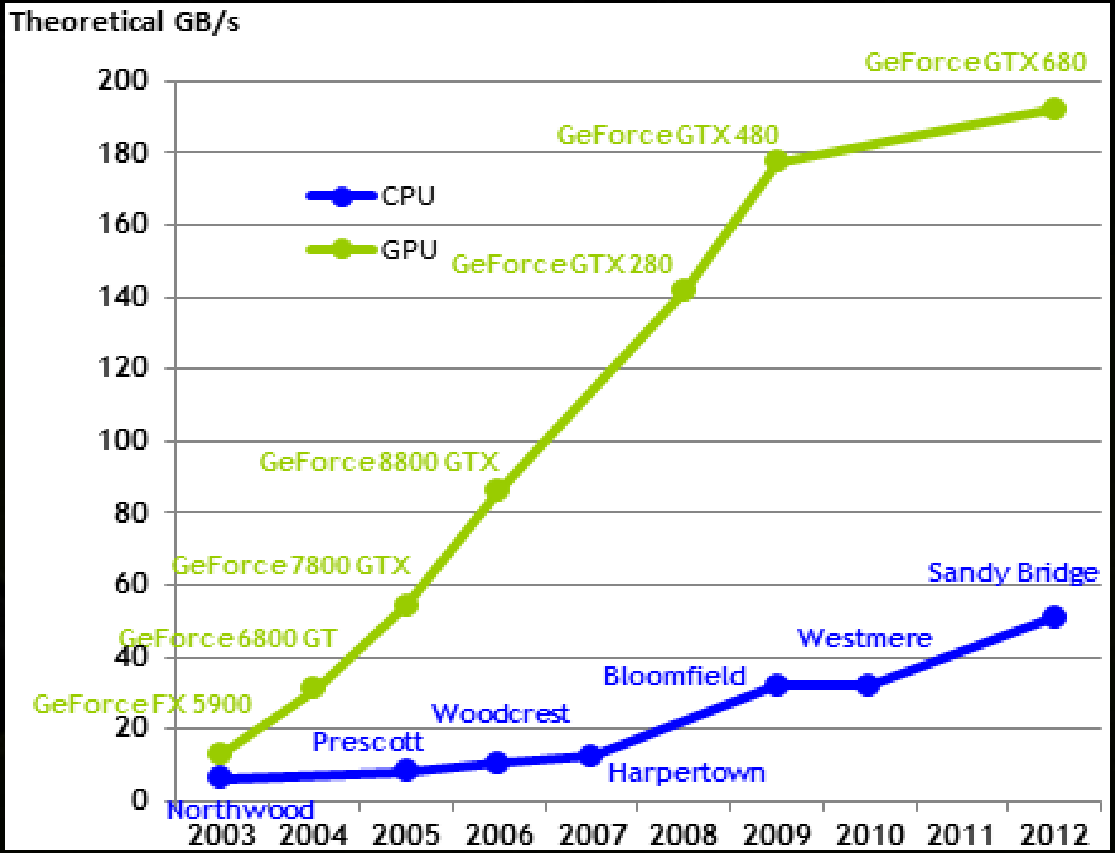
The free lunch is over.

CPU vs. GPU

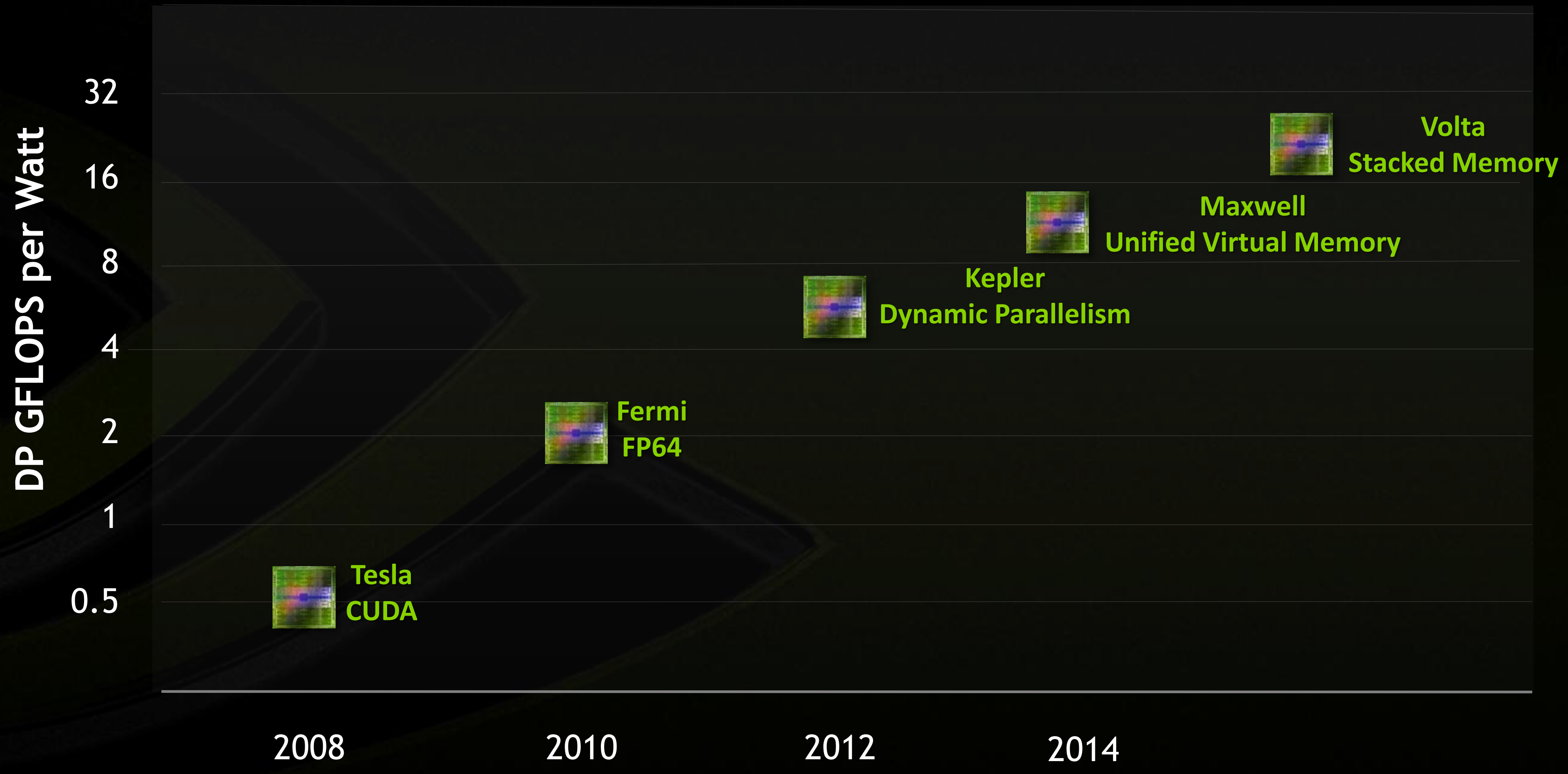


GPUs = Higher Flops and Memory Bandwidth

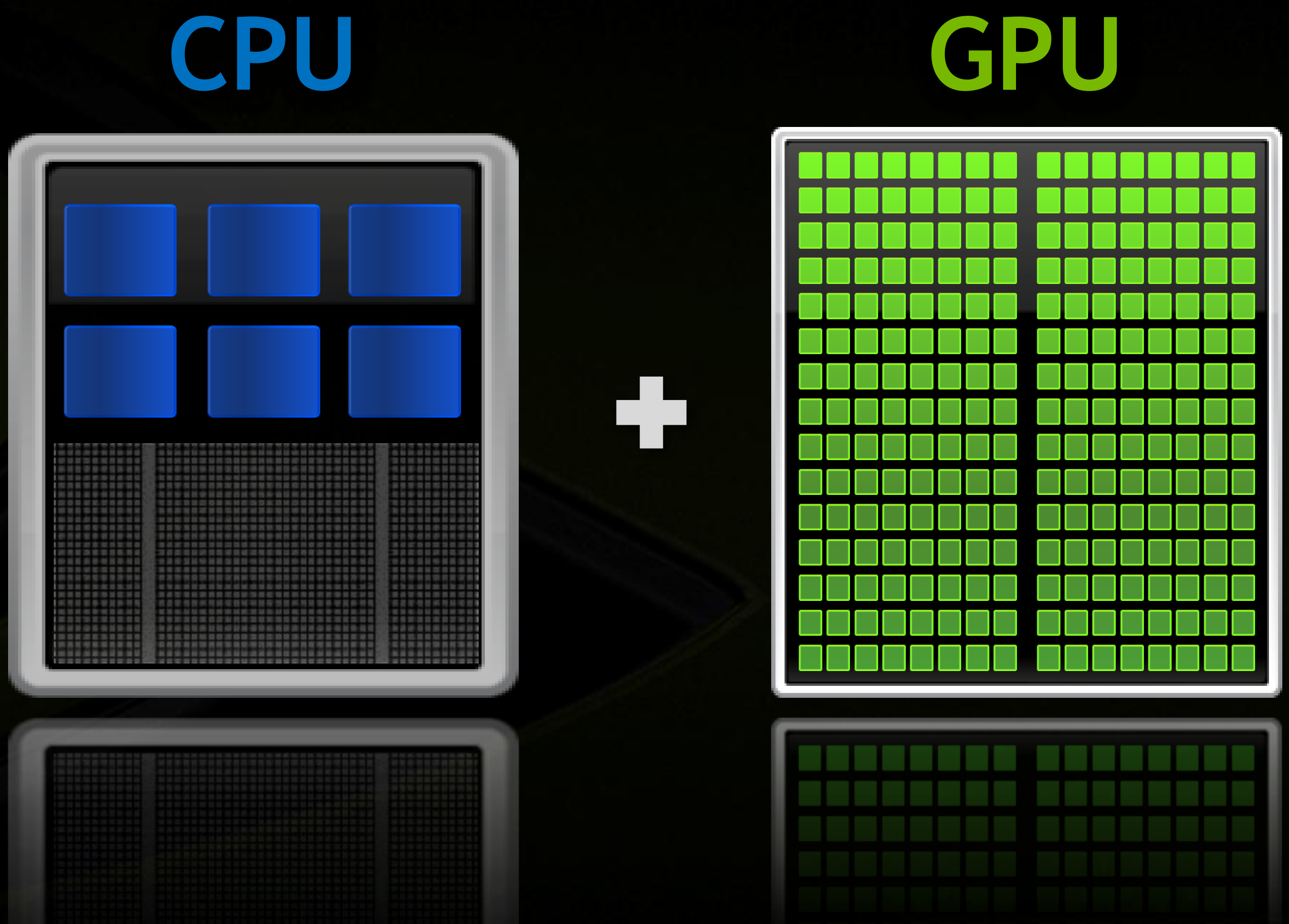




GPUs : Two Year Heart Beat



CPU+GPU?



Heterogeneous Parallel Computing

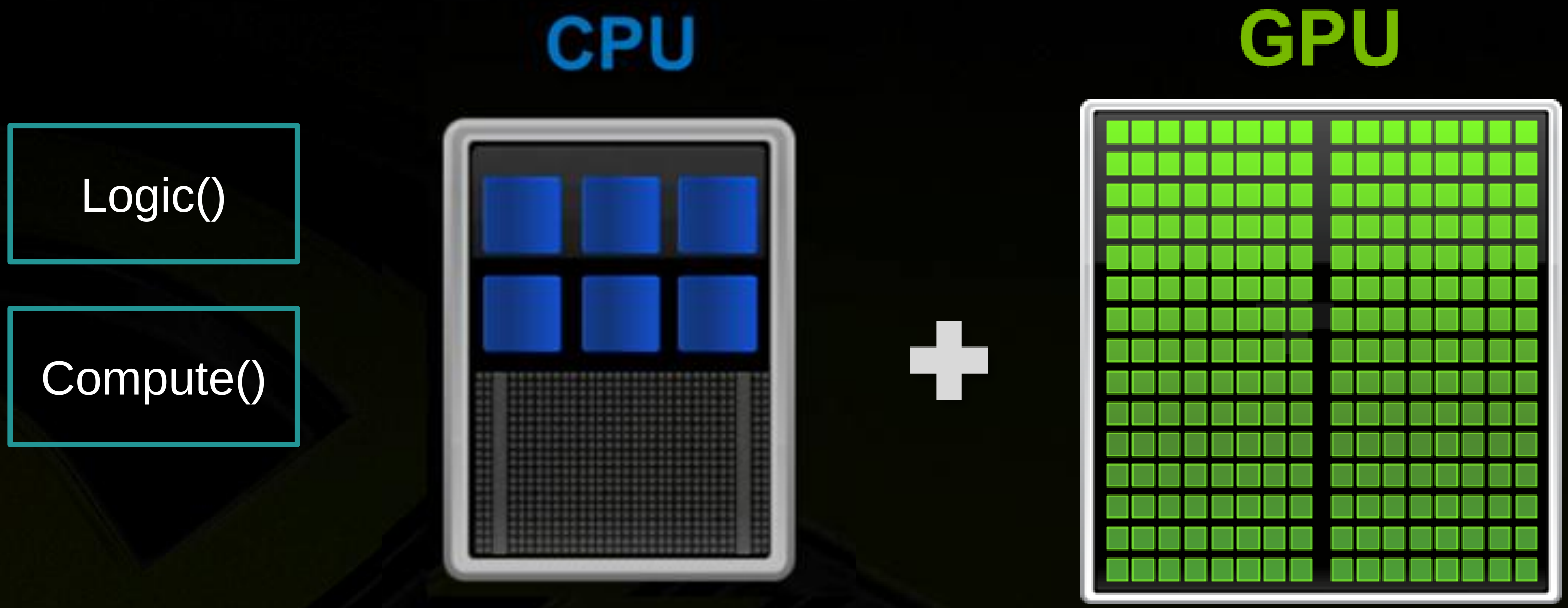
Logic()

Compute()

CPU

Latency-Optimized
Fast Serial Processing

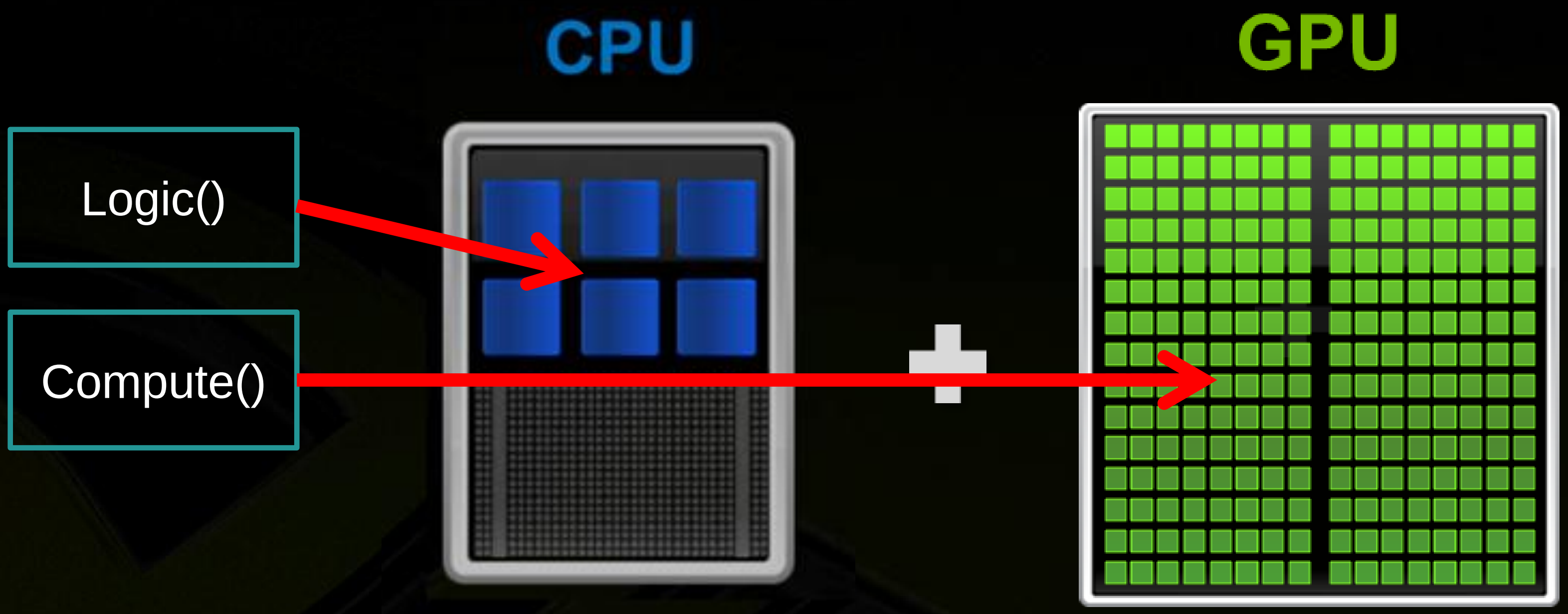
Heterogeneous Parallel Computing



Latency-Optimized
Fast Serial Processing

Throughput-Optimized
Fast Parallel Processing

Heterogeneous Parallel Computing

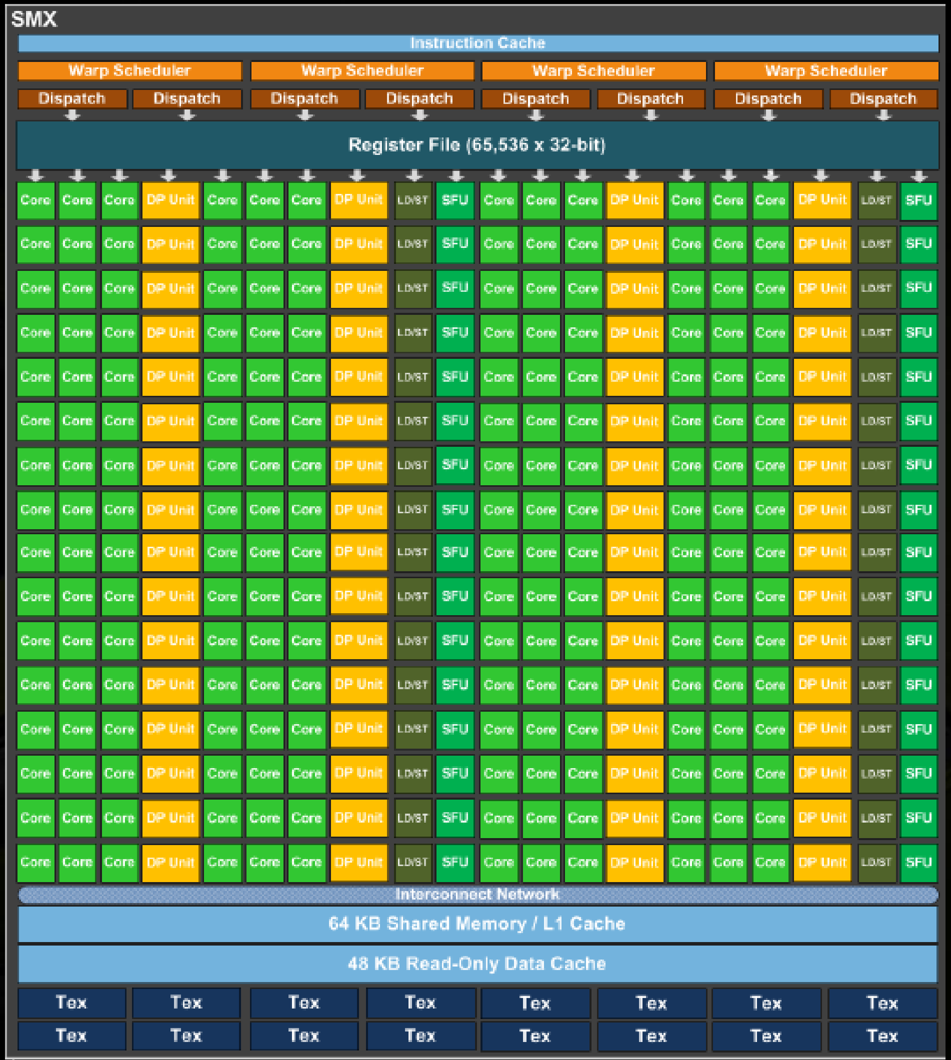


Latency-Optimized
Fast Serial Processing

Throughput-Optimized
Fast Parallel Processing

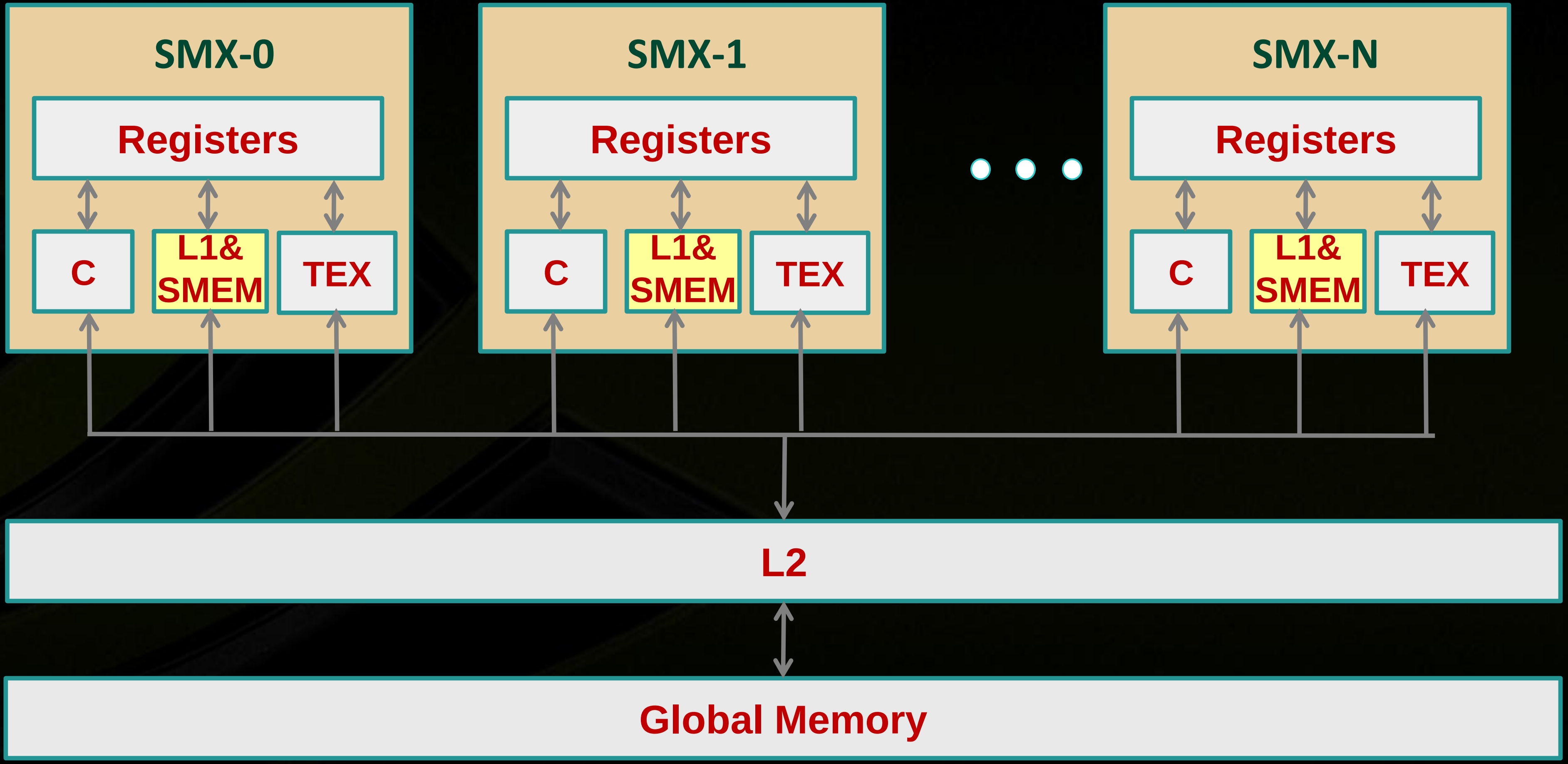
Kepler GK110 Block Diagram

GK110 SMX

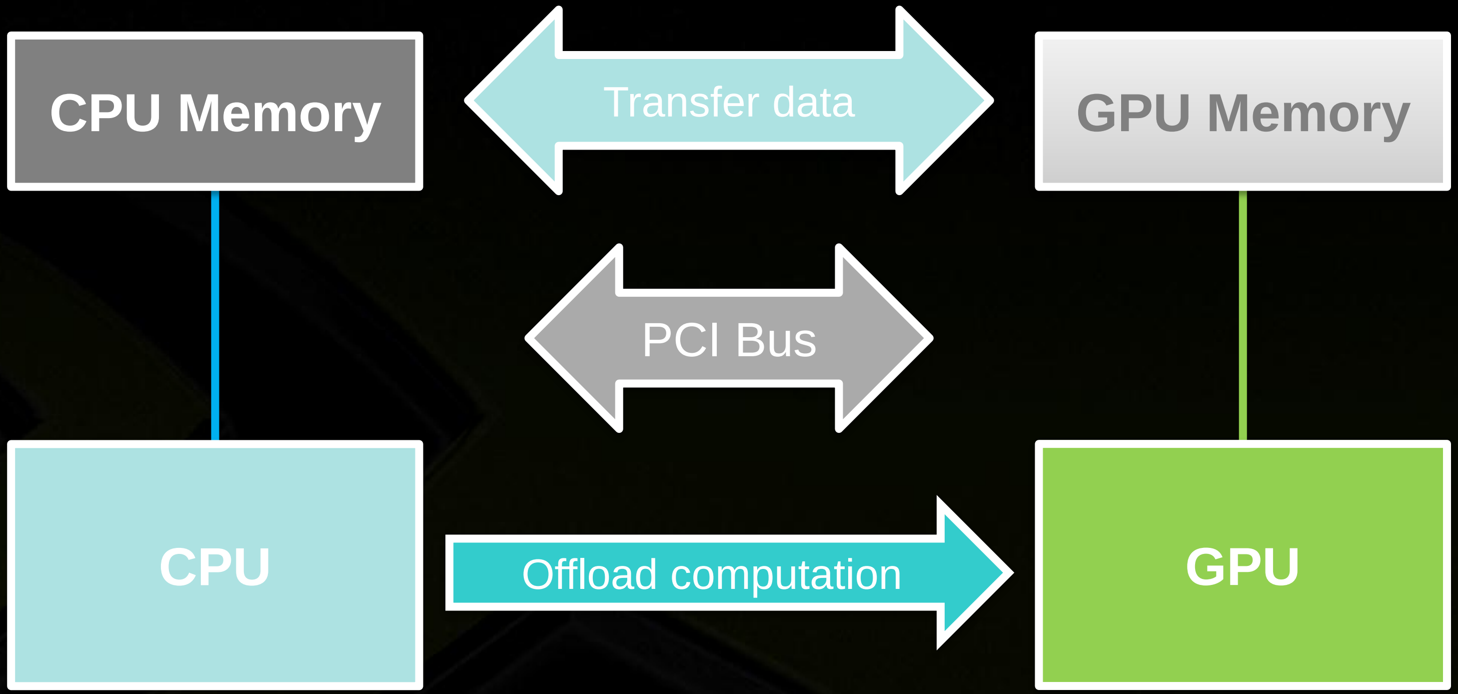


- Control unit
  - 4 Warp Scheduler
  - 8 instruction dispatcher
- Execution unit
  - 192 single-precision CUDA Cores
  - 64 double-precision CUDA Cores
  - 32 SFU, 32 LD/ST
- Memory
  - Registers: 64K 32-bit
  - Cache

Memory Hierarchy



GPU Computing



GPU computing is all about 2 things:
- Transfer data between CPU-GPU
- Do parallel computing on GPU

3 Ways to Accelerate Applications

Applications

- **OpenACC Directives** — Easily Accelerate Applications
- **Programming Languages** — Maximum Flexibility
- **Libraries** — “Drop-in” Acceleration

3 Ways to Accelerate Applications

Applications

OpenACC Directives	Programming Languages	Libraries
Easily Accelerate Applications	Maximum Flexibility	“Drop-in” Acceleration

Vector Addition using OpenACC C

```c
void vecadd(float *x,float *y,int n)
{

  for (int i=0;i<n;++i)
    y[i]=x[i]+y[i];
}

float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
vecadd(x,y,n);
free(x);
free(y);
```

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

float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
vecadd(x,y,n);
free(x);
free(y);
```

`#pragma acc kernels: run the loop in parallel on GPU`

Vector Addition using OpenACC Fortran

```
subroutine vecadd(a, b, c, n)
    real(4) :: a(n), b(n), c(n)
    integer(4) :: n, i
    do i = 1, n
        c(i) = a(i) + b(i)
    end do
end


allocate(a(1:len), b(1:len), c(1:len))
call vecadd(a, b, c, len)
do i = 1, len
   write(*, *) c(i)
end do
```

```
subroutine vecadd(a, b, c, n)
    real(4) :: a(n), b(n), c(n)
    integer(4) :: n, i
    !$acc kernels
    do i = 1, n
        c(i) = a(i) + b(i)
    end do
    !$acc end kernels
end

allocate(a(1:len), b(1:len), c(1:len))
call vecadd(a, b, c, len)
do i = 1, len
   write(*, *) c(i)
end do
```

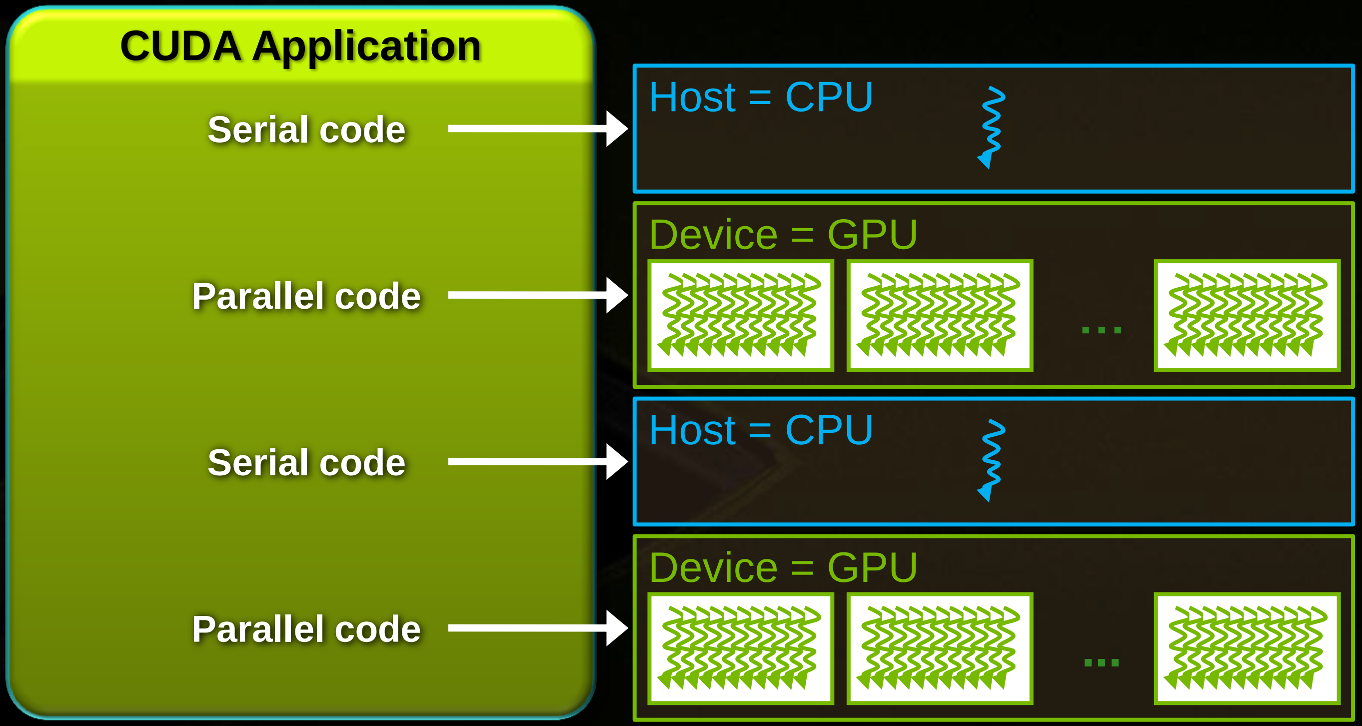
3 Ways to Accelerate Applications

Applications

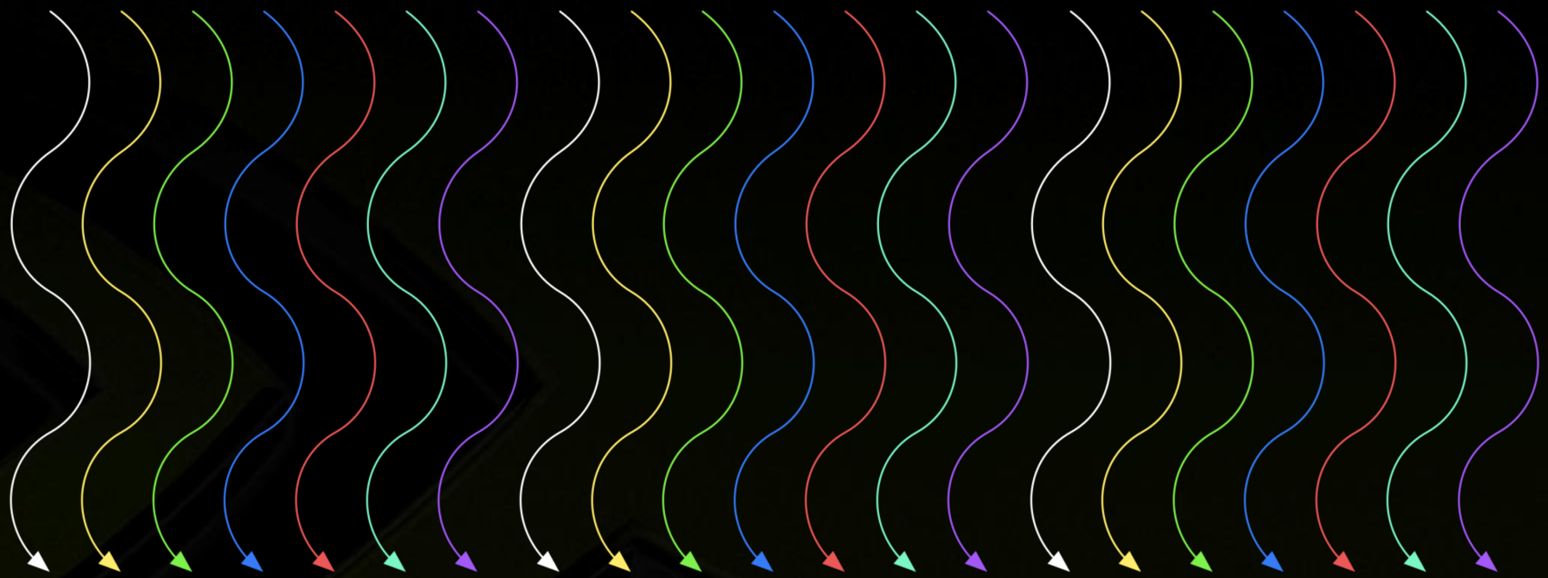
OpenACC Directives	Programming Languages	Libraries
Easily Accelerate Applications	Maximum Flexibility	"Drop-in" Acceleration

Anatomy of a CUDA Application

- Serial code executes in a Host (CPU) thread
- Parallel code executes in many Device (GPU) threads across multiple processing elements

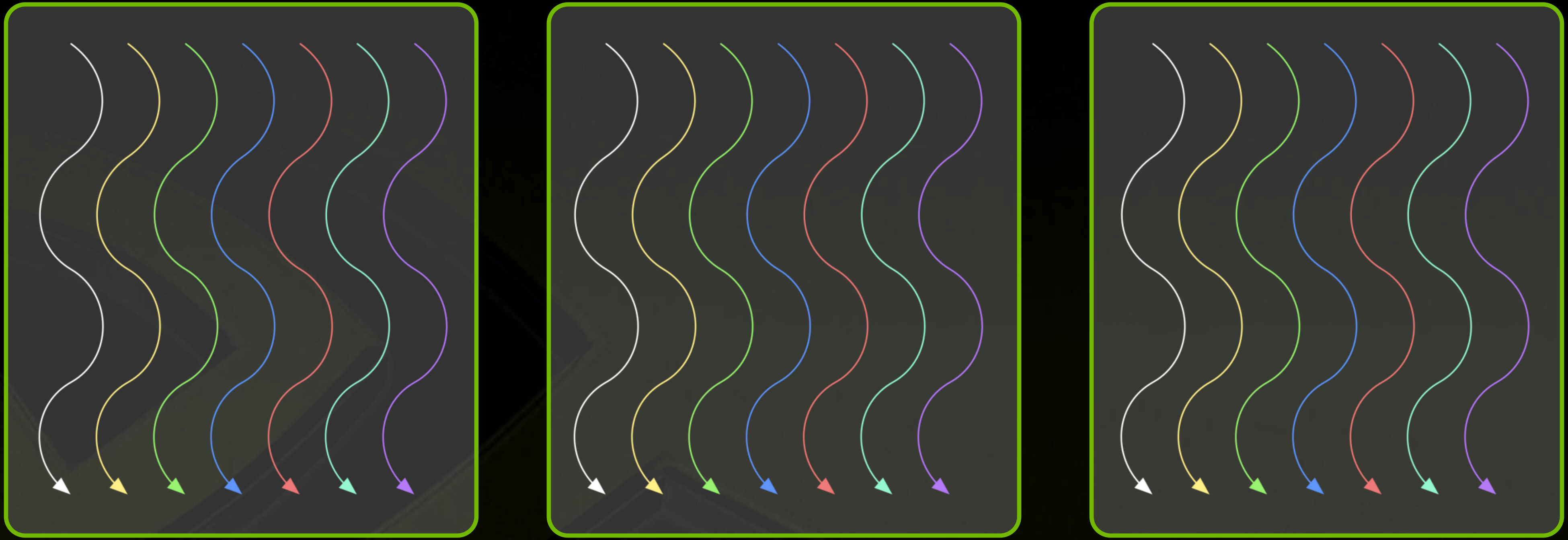


CUDA Kernels: Subdivide into Blocks



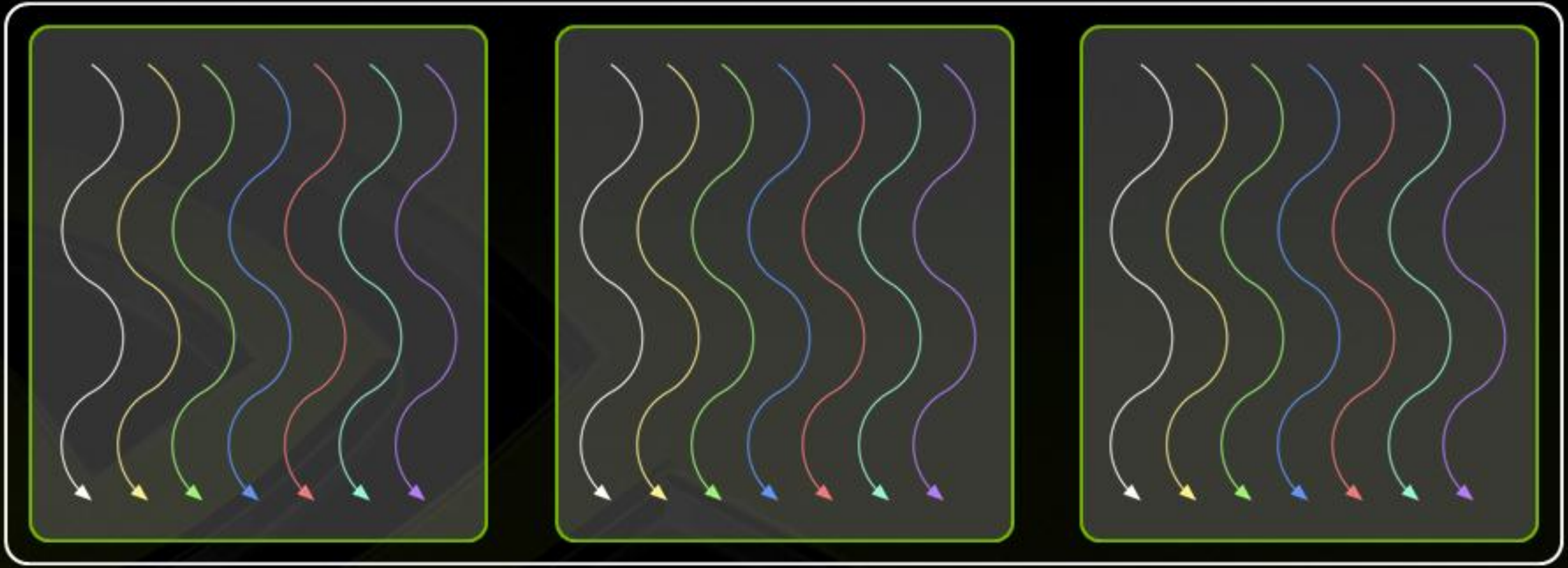
- Threads are grouped into **blocks**

CUDA Kernels: Subdivide into Blocks



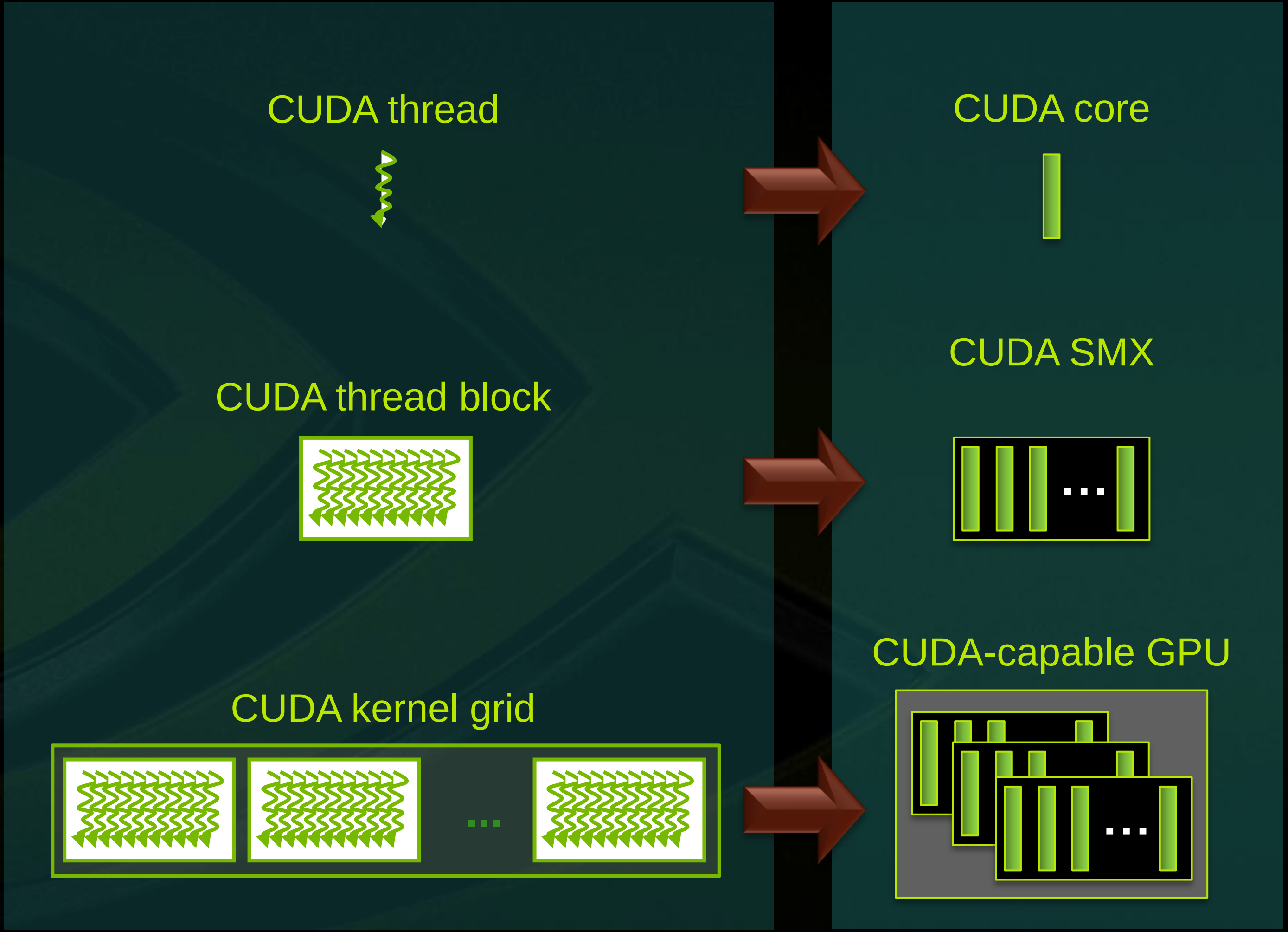
- Threads are grouped into **blocks**
- **Blocks** are grouped into **a grid**

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



- Each thread is executed by a core
- Each block is executed by one SMX and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Vector Addition using CUDA C

```c
void vec_add(float *x,float *y,int n)
{
  for (int i=0;i<n;++i)
    y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));

vec_add(x,y,n);

free(x);free(y);
```

```c
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostToDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
cudaFree(d_x); cudaFree(d_y);
free(x); free(y);
```

Vector Addition using CUDA C

```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostT
oDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostT
oDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDevic
eToHost);
cudaFree(d_x); cudaFree(d_y);
free(x); free(y);
```

Keyword for CUDA kernel

Vector Addition using CUDA C

```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostT
oDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostT
oDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDevic
eToHost);
cudaFree(d_x); cudaFree(d_y);
free(x); free(y);
```

Thread index computation to replace loop

Vector Addition using CUDA C

```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostT
oDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostT
oDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDevic
eToHost);
cudaFree(d_x); cudaFree(d_y);
free(x); free(y);
```

Like malloc
allocate device memory

Vector Addition using CUDA C

```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostToDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
cudaFree(d_x); cudaFree(d_y);
free(x); free(y);
```

cudaMemcpy to transfer data from CPU to GPU

Vector Addition using CUDA C

```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostToDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
cudaFree(d_x); cudaFree(d_y);
free(x); free(y);
```

<<<*,*>>> to specify size of block and grid

Vector Addition using CUDA C

```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostT
oDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostT
oDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDevic
eToHost);
cudaFree(d_x); cudaFree(d_y);
free(x); free(y);
```

Another cudaMemcpy to transfer result back

Vector Addition using CUDA C

```
__global__ void vec_add(float *x,float *y,int n)
{
  int i=blockIdx.x*blockDim.x+threadIdx.x;
  y[i]=x[i]+y[i];
}
float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
float *d_x,*d_y;
cudaMalloc(&d_x,n*sizeof(float));
cudaMalloc(&d_y,n*sizeof(float));
cudaMemcpy(d_x,x,n*sizeof(float),cudaMemcpyHostToDevice);
cudaMemcpy(d_y,y,n*sizeof(float),cudaMemcpyHostToDevice);
vec_add<<<n/128,128>>>(d_x,d_y,n);
cudaMemcpy(y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
cudaFree(d_x); cudaFree(d_y);
free(x); free(y);
```

cudaFree to free GPU memory

Vector Addition using CUDA Fortran

```
subroutine vecadd(a, b, c, n)
    real(4) :: a(n), b(n), c(n)
    integer(4) :: n, i
    do i = 1, n
        c(i) = a(i) + b(i)
    end do
end



allocate(a(1:len), b(1:len), (1:len))
call vecadd(a, b, c, len)
do i = 1, len
   write(*, *) c(i)
end do
```

```
module m
implicit none
contains
           attributes(global) subroutine vecadd(a, b, c, n)
           real(4), device :: a(:), b(:), c(:)
           integer, value :: n
           integer :: i
           i = blockDim%x *(blockIdx%x-1) + threadIdx%x
           c(i) = a(i) + b(i)

           end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

Vector Addition using CUDA Fortran

```fortran
module m
implicit none
contains
          attributes(global) subroutine vecadd(a, b, c, n)
          real(4), device :: a(:), b(:), c(:)
          integer, value :: n
          integer :: i
          i = blockDim%x *(blockIdx%x-1) + threadIdx%x
          c(i) = a(i) + b(i)
          end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

CUDA code must be in module

Vector Addition using CUDA Fortran

```
module m
implicit none
contains
           attributes(global) subroutine vecadd(a, b, c, n)
           real(4), device :: a(:), b(:), c(:)
           integer, value :: n
           integer :: i, j
           i = blockDim%x *(blockIdx%x-1) + threadIdx%x
           c(i) = a(i) + b(i)

           end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

Keyword for CUDA kernel

Vector Addition using CUDA Fortran

```
module m
implicit none
contains
           attributes(global) subroutine vecadd(a, b, c, n)
           real(4), device :: a(:), b(:), c(:)
           integer, value :: n
           integer :: i, j
           i = blockDim%x *(blockIdx%x-1) + threadIdx%x

           c(i) = a(i) + b(i)
           end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

Thread index computation to replace for loop

Vector Addition using CUDA Fortran

```
module m
implicit none
contains
            attributes(global) subroutine vecadd(a, b, c, n)
            real(4), device :: a(:), b(:), c(:)
            integer, value :: n
            integer :: i, j
            i = blockDim%x *(blockIdx%x-1) + threadIdx%x
            c(i) = a(i) + b(i)
            end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

include module m and cudafor

Vector Addition using CUDA Fortran

```
module m
implicit none
contains
          attributes(global) subroutine vecadd(a, b, c, n)
          real(4), device :: a(:), b(:), c(:)
          integer, value :: n
          integer :: i, j
          i = blockDim%x *(blockIdx%x-1) + threadIdx%x
          c(i) = a(i) + b(i)
          end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

Array are located on GPU memory

Vector Addition using CUDA Fortran

```
module m
implicit none
contains
          attributes(global) subroutine vecadd(a, b, c, n)
          real(4), device :: a(:), b(:), c(:)
          integer, value :: n
          integer :: i, j
          i = blockDim%x *(blockIdx%x-1) + threadIdx%x
          c(i) = a(i) + b(i)
          end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

copy host data to GPU memory

Vector Addition using CUDA Fortran

```
module m
implicit none
contains
          attributes(global) subroutine vecadd(a, b, c, n)
          real(4), device :: a(:), b(:), c(:)
          integer, value :: n
          integer :: i, j
          i = blockDim%x *(blockIdx%x-1) + threadIdx%x
          c(i) = a(i) + b(i)
          end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

Invoke kernel in module m

Vector Addition using CUDA Fortran

```
module m
implicit none
contains
           attributes(global) subroutine vecadd(a, b, c, n)
           real(4), device :: a(:), b(:), c(:)
           integer, value :: n
           integer :: i, j
           i = blockDim%x *(blockIdx%x-1) + threadIdx%x
           c(i) = a(i) + b(i)
           end subroutine
end module

use m
use cudafor
real(4), device, allocatable :: d_a(:), d_b(:), d_c(:)
allocate(a(1:n), b(1:n), c(1:n))
allocate(d_a(1:n), d_b(1:n), d_c(1:n))
d_a = a
d_b = b
call vecadd<<< 10, 64 >>>(d_a, d_b, d_c, n)
c = d_c
```

copy back

OpenACC Execution Model on CUDA

- The OpenACC execution model has three levels: gang, worker, and vector
- For GPUs, the mapping is implementation-dependent. Some possibilities:
  - gang==block, worker==warp, and vector==threads of a warp
  - omit “worker” and just have gang==block, vector==threads of a block
- Depends on what the compiler thinks is the best mapping for the problem

Data Clauses

`copy ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
`copyin ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region.
`copyout ( list )`	Allocates memory on GPU and copies data to the host when exiting region.
`create ( list )`	Allocates memory on GPU but does not copy.
`present ( list )`	Data is already present on GPU from another containing data region.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

Review vector Addition using OpenACC C

```
void vec_add(float *x,float *y,int n)
{

  for (int i=0;i<n;++i)
    y[i]=x[i]+y[i];
}

float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
vec_add(x,y,n);
free(x);
free(y);
```

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

float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
saxpy(x,y,n);
free(x);
free(y);
```

`#pragma acc kernels: run the loop in parallel on GPU`

Review vector Addition using OpenACC C

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

float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
vec_add(x,y,n);
free(x);
free(y);
```

```
void vec_add(float *x,float *y,int n)
{
#pragma acc kernels copyin(x[0:n]) copy(y[0:n])
gang(grid) vector(block)
  for (int i=0;i<n;++i)
    y[i]=x[i]+y[i];
}

float *x=(float*)malloc(n*sizeof(float));
float *y=(float*)malloc(n*sizeof(float));
vec_add(x,y,n);
free(x);
free(y);
```

`#pragma acc kernels`: run the loop in parallel on GPU

Hello World on CPU

hello_world.c:

```
#include <stdio.h>

void hello_world_kernel()
{
    printf("Hello World\n");
}

int main()
{
    hello_world_kernel();
}
```

Compile & Run:

```
gcc hello_world.c
./a.out
```

Hello World on GPU

hello_world.cu:

```
#include <stdio.h>

__global__ void hello_world_kernel()
{
    printf(“Hello World\n”);
}

int main()
{
    hello_world_kernel<<<1,1>>>();
}
```

Compile & Run:

```
nvcc hello_world.cu
./a.out
```

Hello World on GPU

hello_world.cu:

```
#include <stdio.h>

__global__ void hello_world_kernel()
{
    printf("Hello World\n");
}

int main()
{
    hello_world_kernel<<<1,1>>>();
}
```

Compile & Run:

```
nvcc hello_world.cu
./a.out
```

- **CUDA kernel within .cu files**
- **.cu files compiled by nvcc**
- **CUDA kernels preceded by "__global__"**
- **CUDA kernels launched with "<<<…,…>>>"**

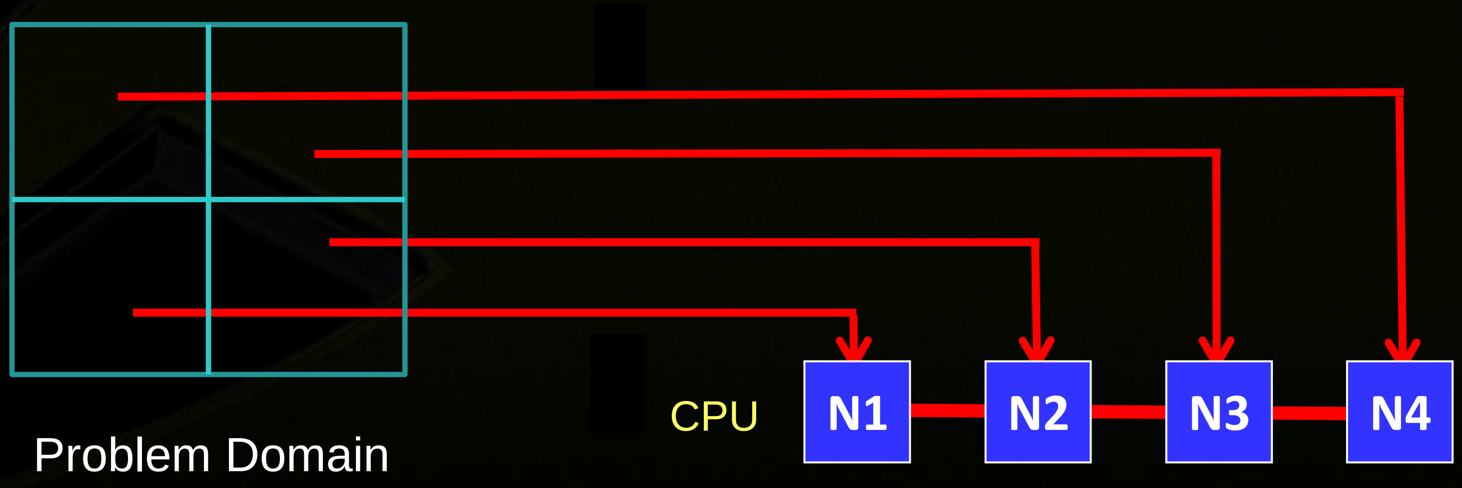
GPU Cluster Programming

- **Programming: straightforward**

Problem Domain

GPU Cluster Programming

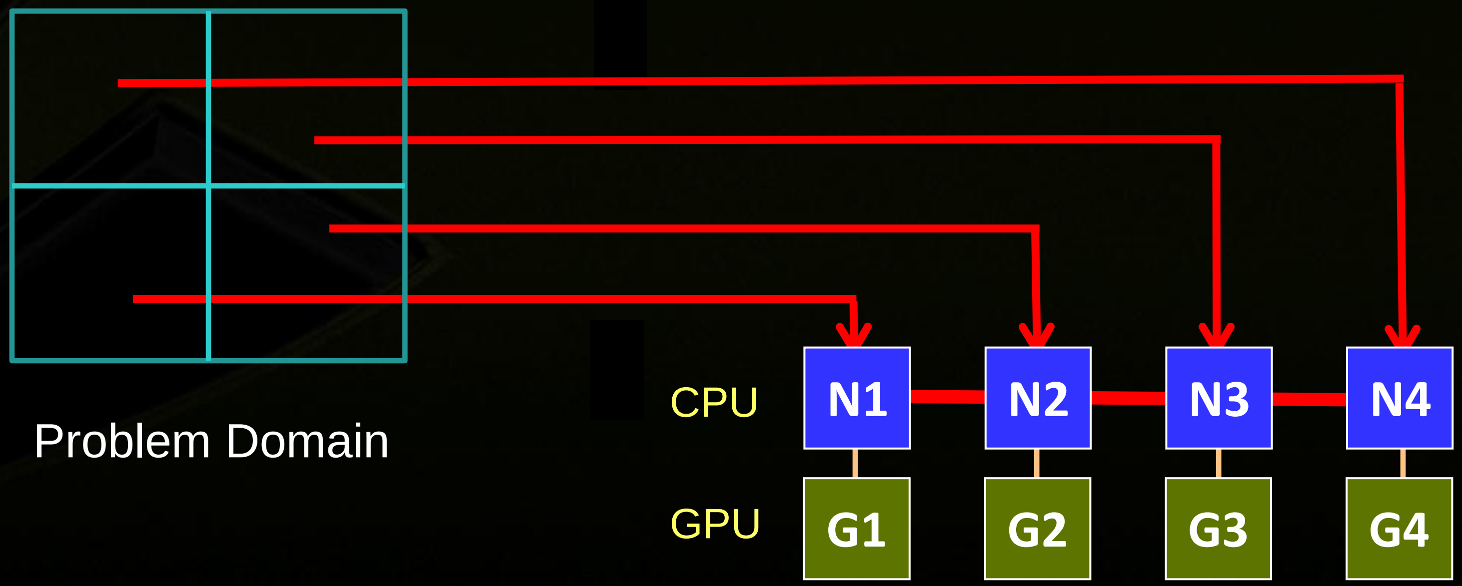
- **Programming: straightforward**
  - **MPI takes care of inter-node communication**



GPU Cluster Programming

- Programming: straightforward
  - MPI takes care of inter-node communication
  - GPU speeds up the computation of each node

No modification required in the MPI part.



MATLAB

- Parallel Computing Toolbox
- Fourier Analysis, Linear Algebra, ...
- NVIDIA GPUs supported through gpuArray, reference: http://www.mathworks.com/help/distcomp/using-gpuarray.html

How to invoke CUDA in matlab

- **Idea:**
  - **Matlab invokes C using mex file**
  - **C invokes CUDA with dll or object file**
- **How to build C program in matlab**
  - **Type mex –setup to choose C compiler step by step**
- **How to create dll with nvcc**
  - **-Xcompiler option of nvcc**

How to invoke CUDA in matlab(CUDA kernel)

```
__global__ void test(int *a, int size){
        int idx = blockDim.x*blockIdx.x + threadIdx.x;
        if(idx < size)
                a[idx] += idx;
}
void w(int *a, int size){
        int *d_a;
        cudaMalloc((void**)&d_a, size*sizeof(int));
        cudaMemcpy(d_a, a, size*sizeof(int), cudaMemcpyHostToDevice);

        int blocksize = 256;
        test<<<(size+blocksize-1)/blocksize, blocksize>>>(d_a, size);
        cudaMemcpy(a, d_a, size*sizeof(int), cudaMemcpyDeviceToHost);

        cudaFree(d_a);
}
```

compile to dll with:

```
nvcc -c -shared -Xcompiler=-fPIC test.cu -o libtest.so
```

How to invoke CUDA in matlab(mex file)

```
#include<string.h>
#include <stdio.h>
#include <stdlib.h>
#include "mex.h"
void w(int*, int);
void mexFunction(int nlhs, mxArray **plhs, int hrhs, const mxArray *prhs[]){
        int size = *(mxGetPr(prhs[0]));
        int *a = (int*) malloc(size*sizeof(int));
        memset(a, 0, sizeof(int)*size);
        w(a, size);

        for(int i = 0; i < size; i++){
                printf("%d %d\n", i, a[i]);
        }
}
```

compile mex file:

```
mex invoke.cpp -ltest -L. -lcudart -L/usr/local/cuda/lib64
```

Open matlab on the directory of invoke.cpp, type:

```
invoke(6)
```

Thanks， Q&A

3 Ways to Accelerate Applications

Applications		
OpenACC Directives	Programming Languages	Libraries
Easily Accelerate Applications	Maximum Flexibility	“Drop-in” Acceleration

Why should you use libraries?

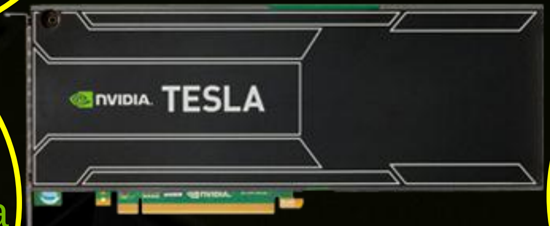
- No need to reinvent the wheel
  - Implement complex algorithms
  - Deal with details of the platform
- High Performance
  - Expert: In depth knowledge of architecture
- Low Maintenance
  - Rigorous testing/quality-assurance
  - Have someone to file bugs against

Accelerated Compute Libraries in CUDA toolkit

math.h
C99 floating-point

Thrust
Parallel Algorithms

cuFFT
Fast Fourier Transform



cuBLAS
Dense Linear Algebra

cuSPARSE
Sparse Linear Algebra

cuRAND
Random # Generation

NPP
Image & Video Processing

Download free at: www.nvidia.com/getcuda

Patterns in Scientific Computing and CUDA lib

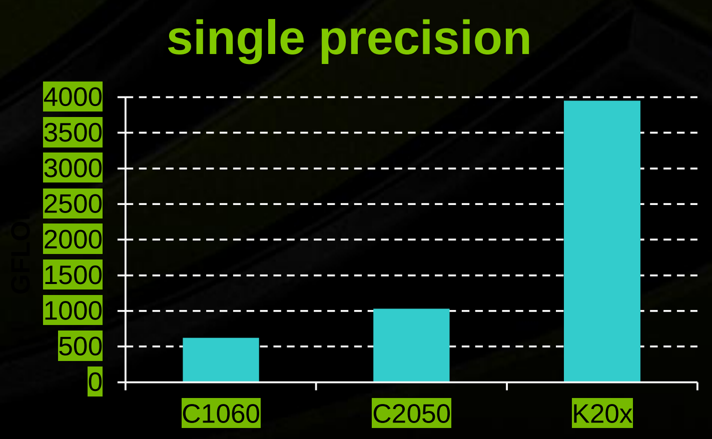
- **Structured grids (SG, cublas)**
- **Unstructured grids (UG, cusparse)**
- **Fast Fourier Transform (FFT, cufft)**
- **Dense Linear Algebra (DLA, cublas)**
- **Sparse Linear Algebra (SLA, cusparse)**
- **Particles (P, thrust, cub)**
- **Monte Carlo (MC, curand)**

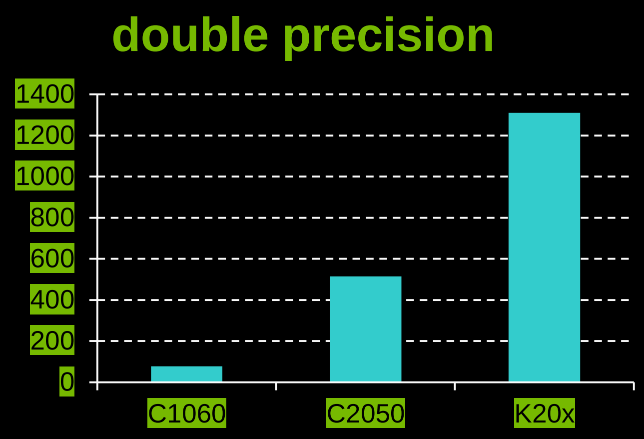
from "Defining Software Requirements for Scientific Computing", Phillip Colella, 2004

math.h

- C99 floating point operations + extras
  - IEEE-754 accurate single and double (+, *, fma, /, ...)
  - Exponential (exp, log, ...)
  - Trigonometric (sin, cos, tan, ...)
  - Special (lgamma, tgamma, erf, erfc, ...)

- Peak (Theoretical) Performance





CUDA C++ Template Library

- Optimized parallel algorithms, include
  - Scan
  - Sort
  - Transform
  - Reduce

- Interface
  - Host and device containers that mimic the C++ STL
  - Allows quick development
  - OpenMP backend for portability

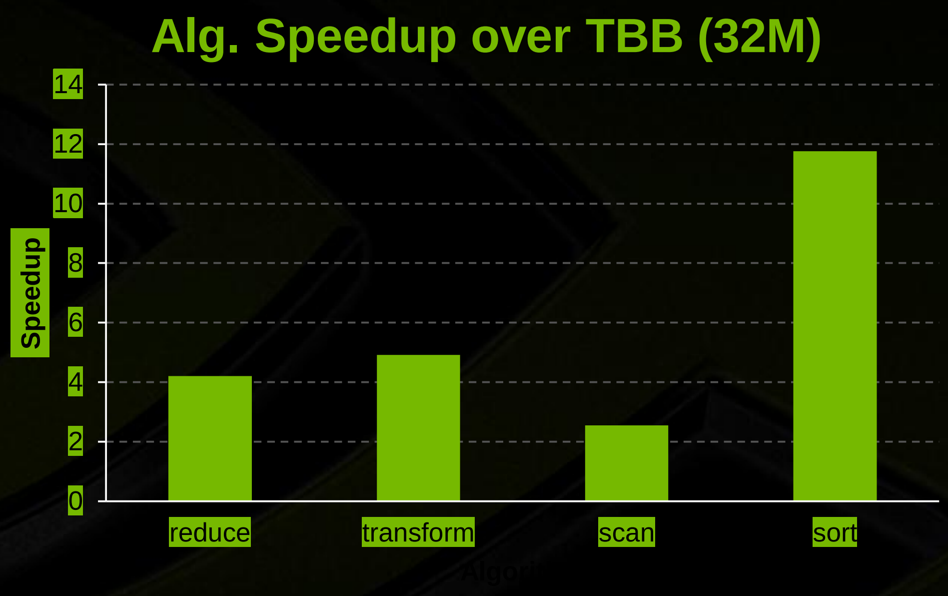
Also available on: http://thrust.github.com

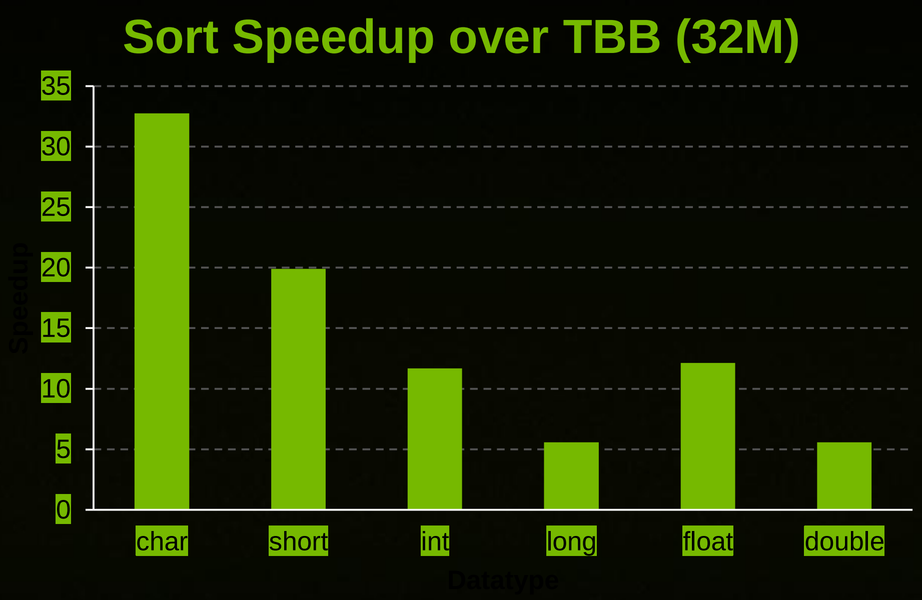
CUDA C++ Template Library

```
int main(void){
    // generate random data serially
    thrust::host_vector<int> h_vec(100);
    std::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer to device and compute sum
    thrust::device_vector<int> d_vec = h_vec;
    int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0,
  thrust::plus<int>());
    return 0;
}
```


Also available on: http://thrust.github.com

Thrust Performance





- Thrust 5.0 on K20X, input and output data on device
- TBB 4.1 on Intel SandyBridge E5-2687W @3.10GHz

Performance may vary based on OS version and motherboard configuration

CUB

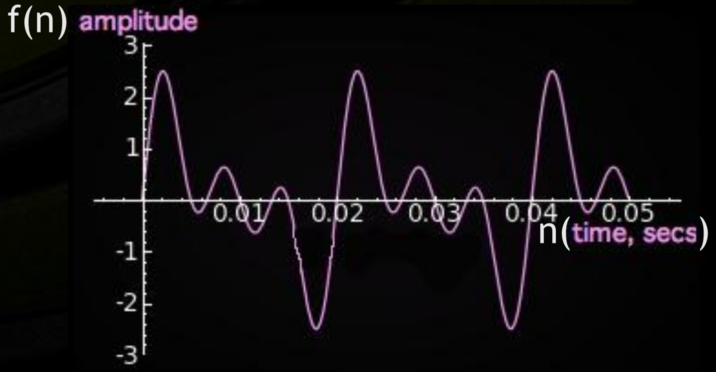
- **open-source high performance CUDA library developed by nvidia research. provides reusable components for every layer of CUDA programming model:**
  - **Device-wide primitives**
    - **global histogram, reduction, etc. (More coming soon..)**
  - **Block-wide primitives**
    - **local radix sort, prefix scan, histogram, reduction, I/O, etc.**
  - **Warp-wide primitives**
    - **local prefix scan, reduction, etc.**
  - **Thread, grid dispatch, and resource utilities**
    - **PTX intrinsics, device reflection, work distribution, caching memory allocators, etc.**

Block sort with cub

```
#include <cub/cub.cuh>
template <int BLOCK_THREADS, int ITEMS_PER_THREAD, typename T>
__global__ void TileSortKernel(T *d_in, T *d_out){

        const int TILE_SIZE = BLOCK_THREADS * ITEMS_PER_THREAD;

        typedef cub::BlockRadixSort<T, BLOCK_THREADS> BlockRadixSort;

        __shared__ typename BlockRadixSort::SmemStorage smem_storage;

        T data[ITEMS_PER_THREAD];

        BlockLoadVectorized(data, d_in + (blockIdx.x * TILE_SIZE));

        BlockRadixSort::SortBlocked(smem_storage, data);

        BlockStoreVectorized(data, d_out + (blockIdx.x * TILE_SIZE));
}
```

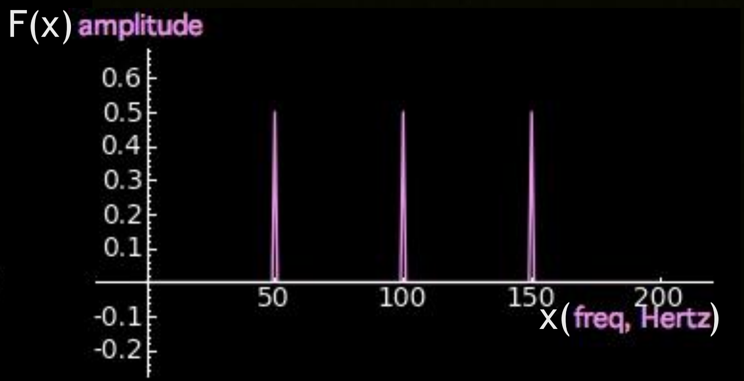
cuFFT Library

- **Features**
  - **Single and double precision**
  - **Real and complex data types**
  - **Radix 2, 3, 5 and 7 natively supported**
  - **1D, 2D and 3D batched transforms**
- **Interface**
  - **Similar to the FFTW “Advanced Interface”**



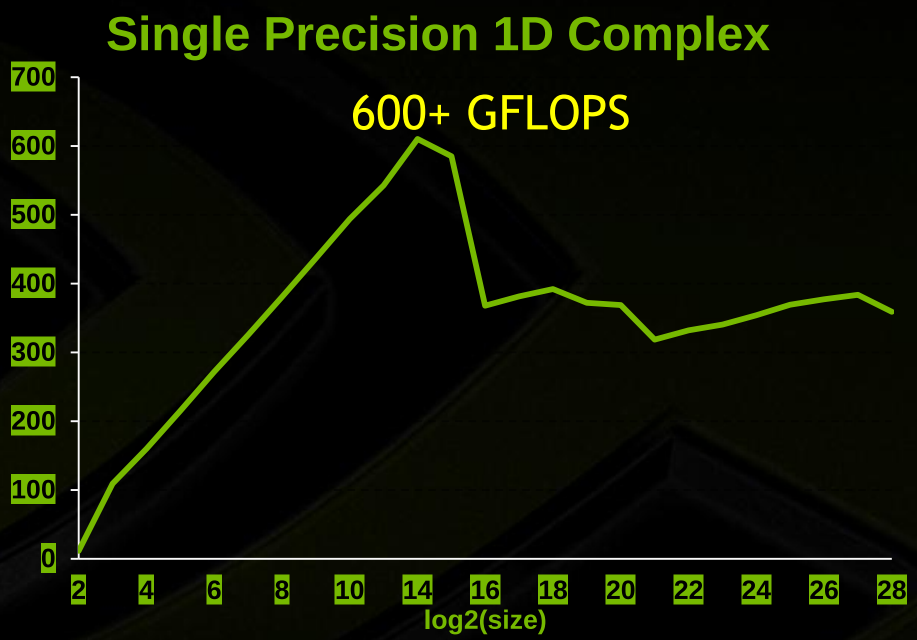
$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(x\frac{n}{N})}$$

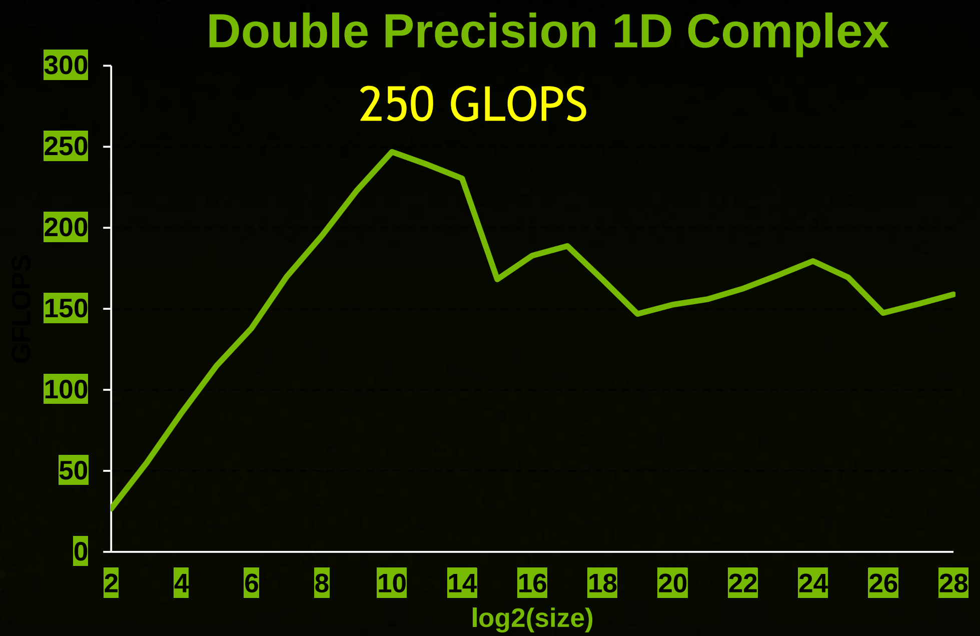
$$f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x) e^{j2\pi(x\frac{n}{N})}$$



cuFFT Performance

1D used in audio processing and as a foundation for 2D and 3D FFTs





Performance may vary based on OS version and motherboard configuration

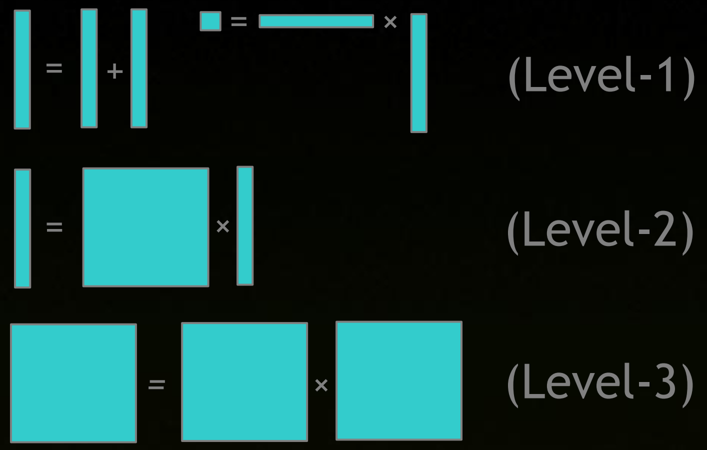
• cuFFT 5.0 on K20X, input and output data on device

Cufft example

```
cufftHandle plan;
cufftPlan1d(&plan, new_size, CUFFT_C2C, 1);

 // Transform signal and kernel
printf("Transforming signal cufftExecC2C\n");
cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex *)d_signal, CUFFT_FORWARD);

cufftDestroy(plan);
cudaFree(d_signal);
```

cuBLAS Library

- Dense Linear Algebra
  - Single and double precision
  - Real and complex data types
  - Vector- and matrix-vector operations
  - Matrix-matrix operations
  - Fortran style, column first, based 1

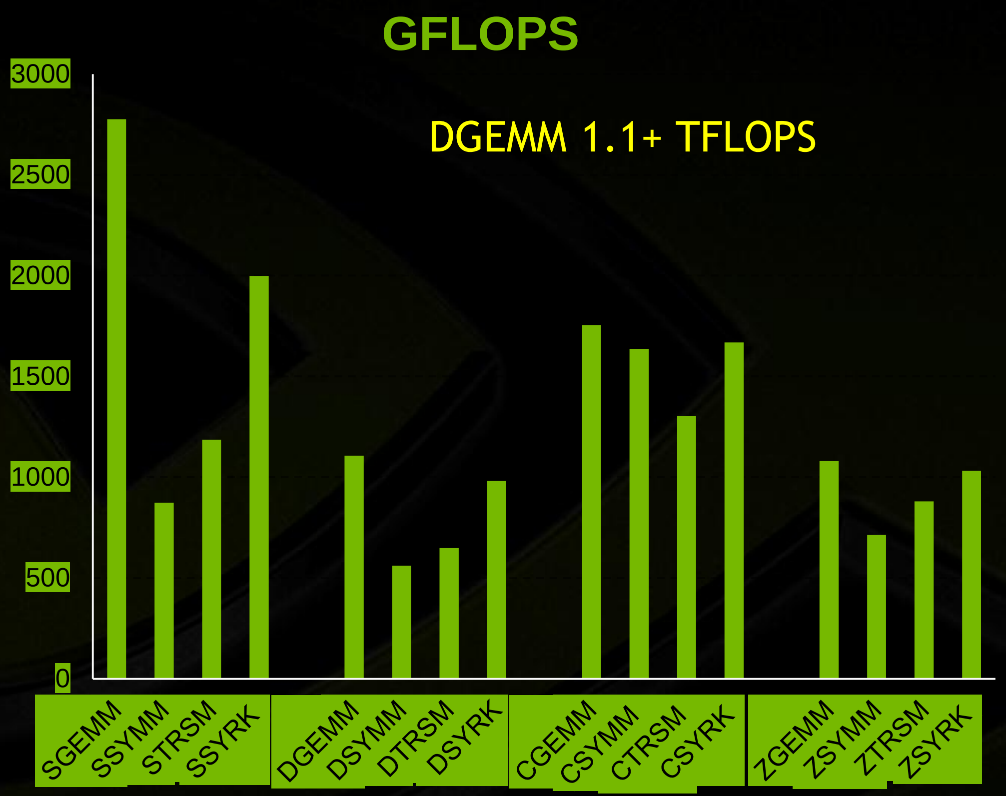


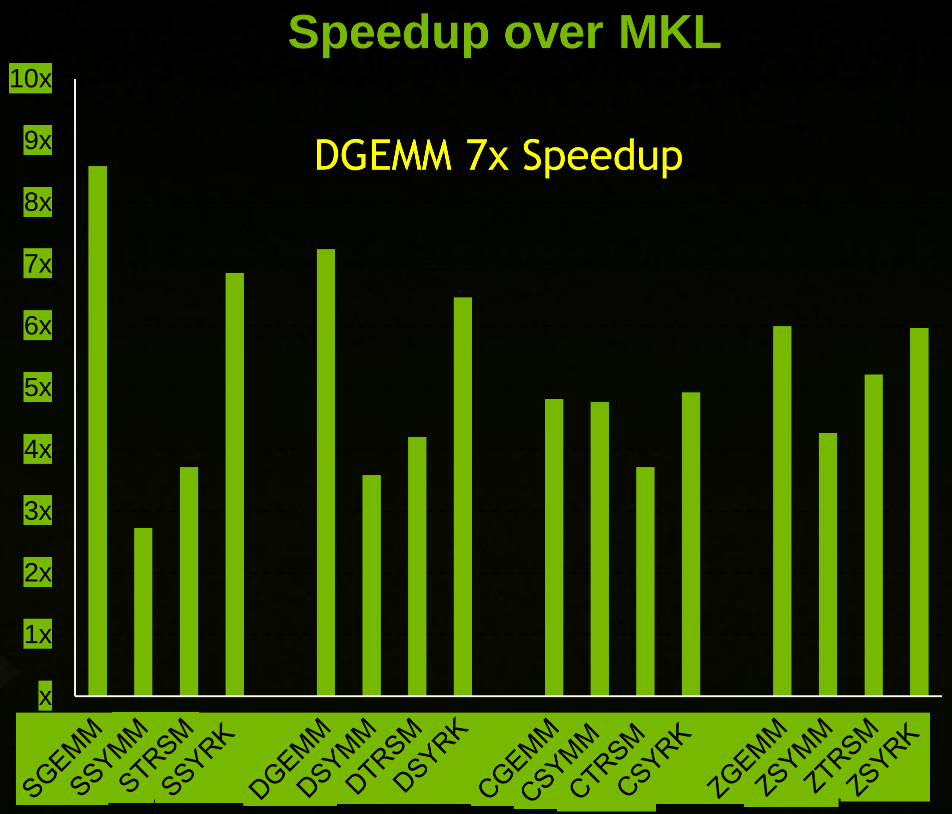
- Interface
  - Similar to Basic Linear Algebra Subprograms (BLAS)
  - Supports dynamic parallelism (on K20)

Matrix multiply matrix example

```
cublasHandle_t handle;
cublasCreate(&handle);

const float alpha = 1.0f;
const float beta  = 0.0f;

cublasStatus_t ret = cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N,
matrix_size.uiWB, matrix_size.uiHA, matrix_size.uiWA, &alpha, d_B, matrix_size.uiWB,
d_A, matrix_size.uiWA, &beta, d_C, matrix_size.uiWA);

cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

cublasDestroy(handle);
```

cuBLAS Performance



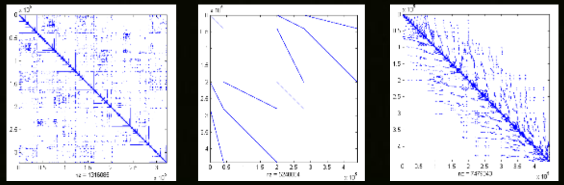


- cuBLAS 5.0 on K20X, input and output data on device
- MKL 10.3.6 on Intel SandyBridge E5-2687W @ 3.10GHz

Performance may vary based on OS version and motherboard configuration

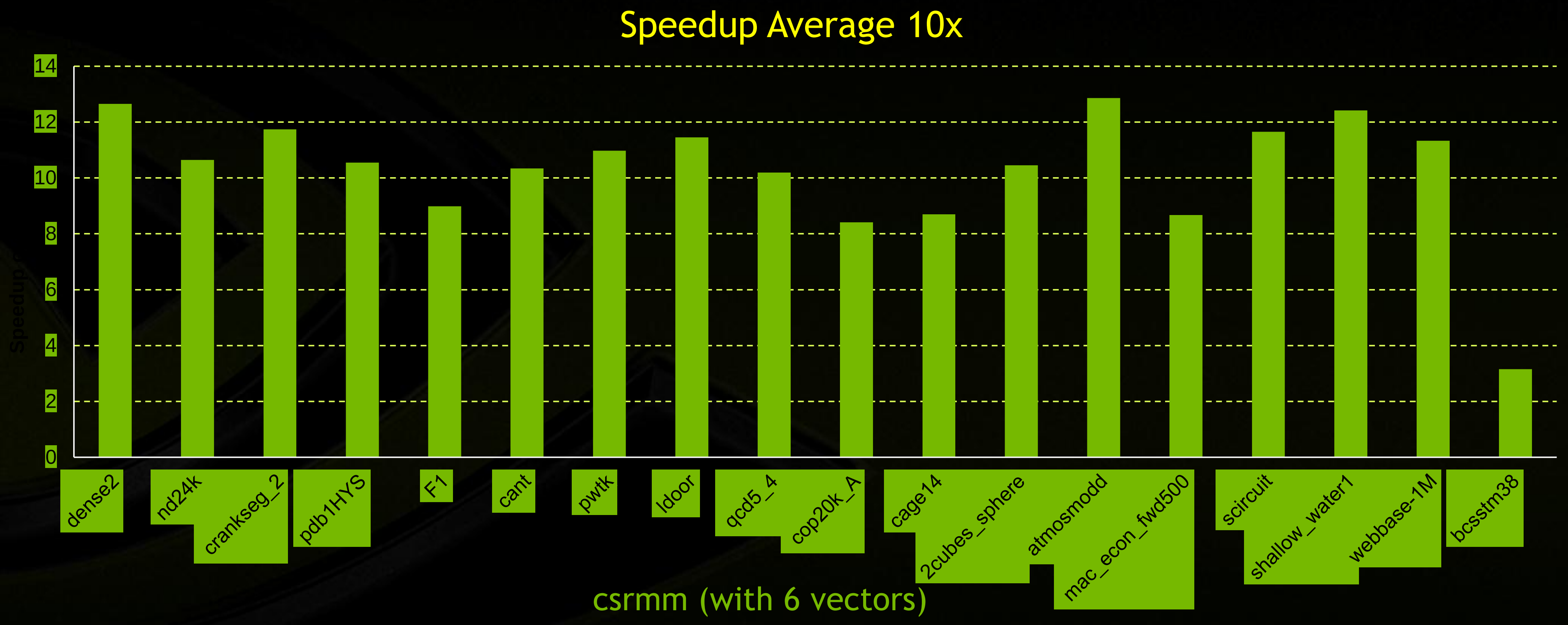
cuSPARSE Library

- Features
  - Format conversion (dense, CSR, block-CSR, ...)
  - Sparse-dense (matrix-vector multiply and triangular solve)
  - Sparse-sparse (matrix-matrix add and multiply)
  - Preconditioners (incomplete-LU, tridiagonal, ...)
- Interface
  - C API with Fortran wrappers



different matrix sparsity patterns

Matrix-vector multiply performance (with multiple vectors)

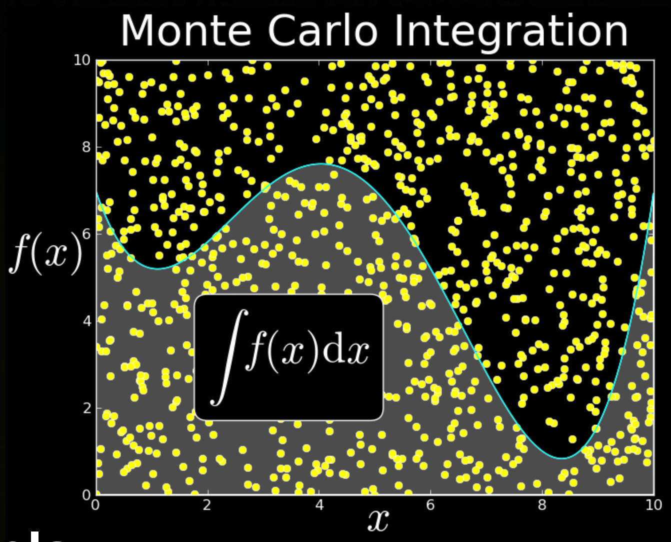


- Average of s/d/c/z routines
- cuSPARSE 5.0 on K20X, input and output data on device
- MKL 10.3.6 on Intel SandyBridge E5-2687W @ 3.10GHz

Performance may vary based on OS version and motherboard configuration

cuRAND Library

- **Features**
  - **Pseudo-RNGs* (XORWOW, MRG32k3a, MTGP)**
  - **Quasi-RNGs (Sobol32, Sobol64)**
  - **Uniform, Normal and Poisson distributions**
  - **Statistical test results in documentation**



- **Interface**
  - **May be called from host routines and device kernels**

*: Random Number Generators

NPP

- Collection of high-performance GPU processing
  - Initial focus on Image, Video and Signal processing
    - Growth into other domains expected
  - Support for multi-channel integer and float data

- C API => name disambiguates between data types, flavor nppiAdd_32f_C1R (…)
  - “Add” two single channel (“C1”) 32-bit float (“32f”) images, possibly masked by a region of interest (“R”)

NPP features a large set of functions

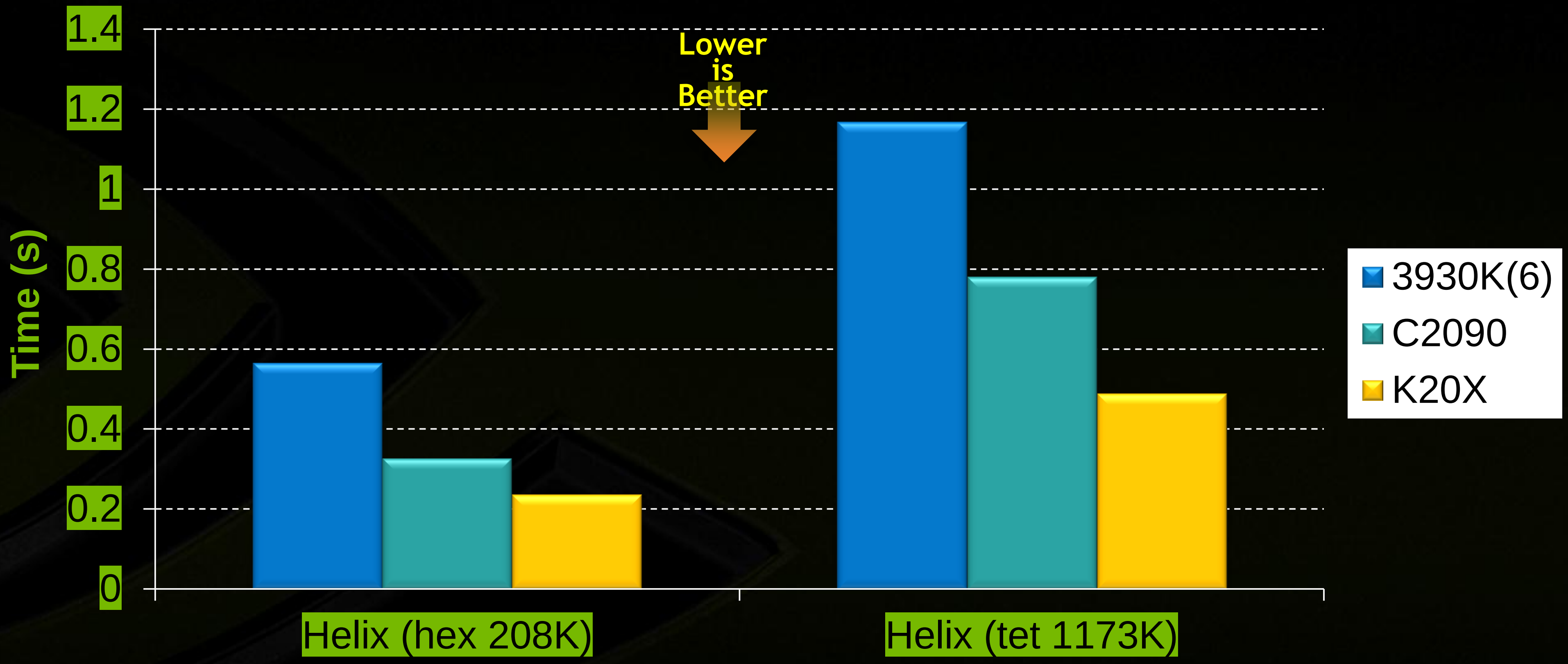
- Arithmetic and Logical Operations
  - Add, mul, clamp, ..
- Threshold and Compare
- Geometric transformations
  - Rotate, Warp, Perspective transformations
  - Various interpolations
- Compression
  - jpeg de/compression
- Image processing
  - Filter, histogram, statistics

NVIDIA NPP

Solution of (Sparse) Linear Systems

- nvAMG: Algebraic MultiGrid
  - Aggregation and Classical MultiGrid
  - Allows third party plug-ins
  - Supports MPI
- GLU: LU re-factorization on the GPU
  - Sparse direct solver
  - Applicable when solving a set of linear systems $A_i x_i = f_i$ for i=1,...,k

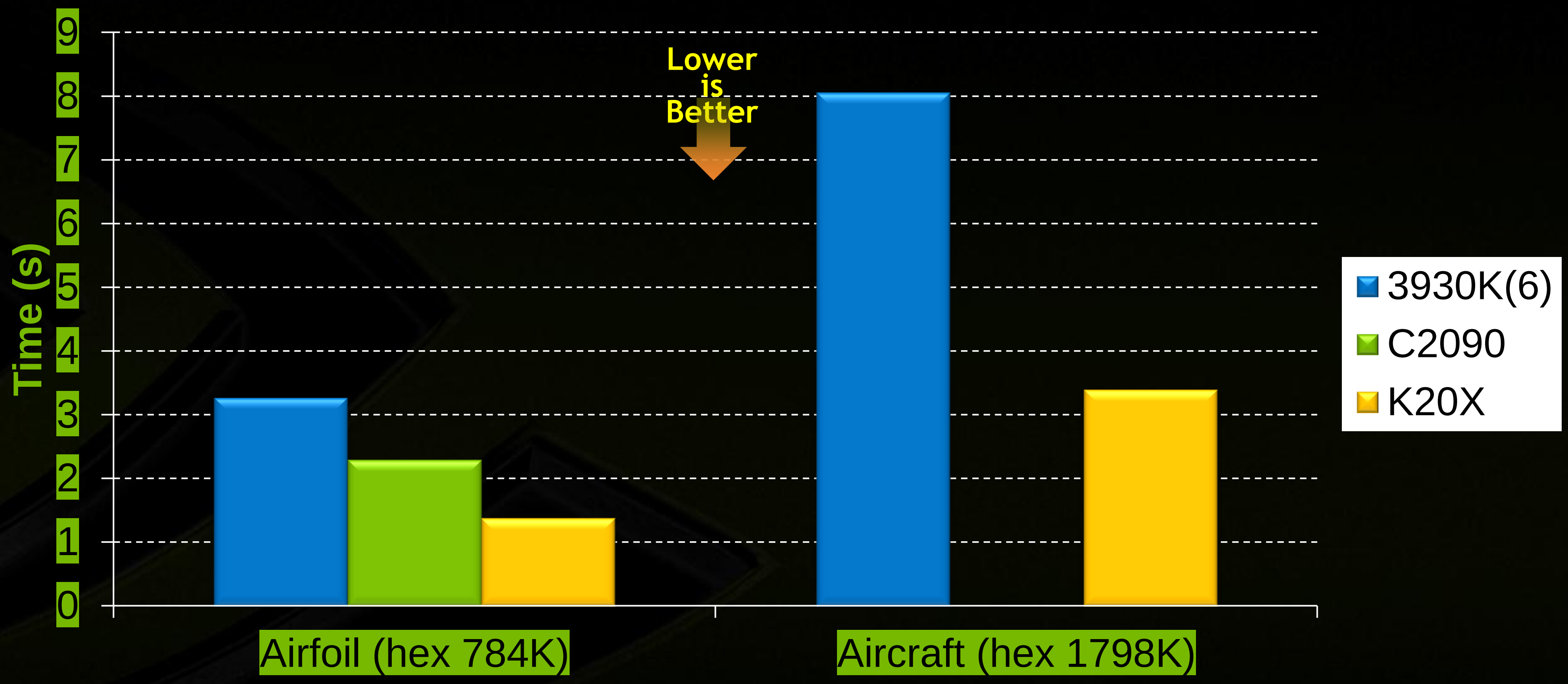
nvAMG on Regular Grids



Performance may vary based on OS version and motherboard configuration

- GPU nvAMG (V-cycle, agg8, MC-DILU, 0pre, 3post) on C2090 and K20X
- CPU Fluent AMG (F-cycle, agg8, DILU, 0pre, 3post) on Intel i7-3930K (Sandy Bridge, 6 Core™)

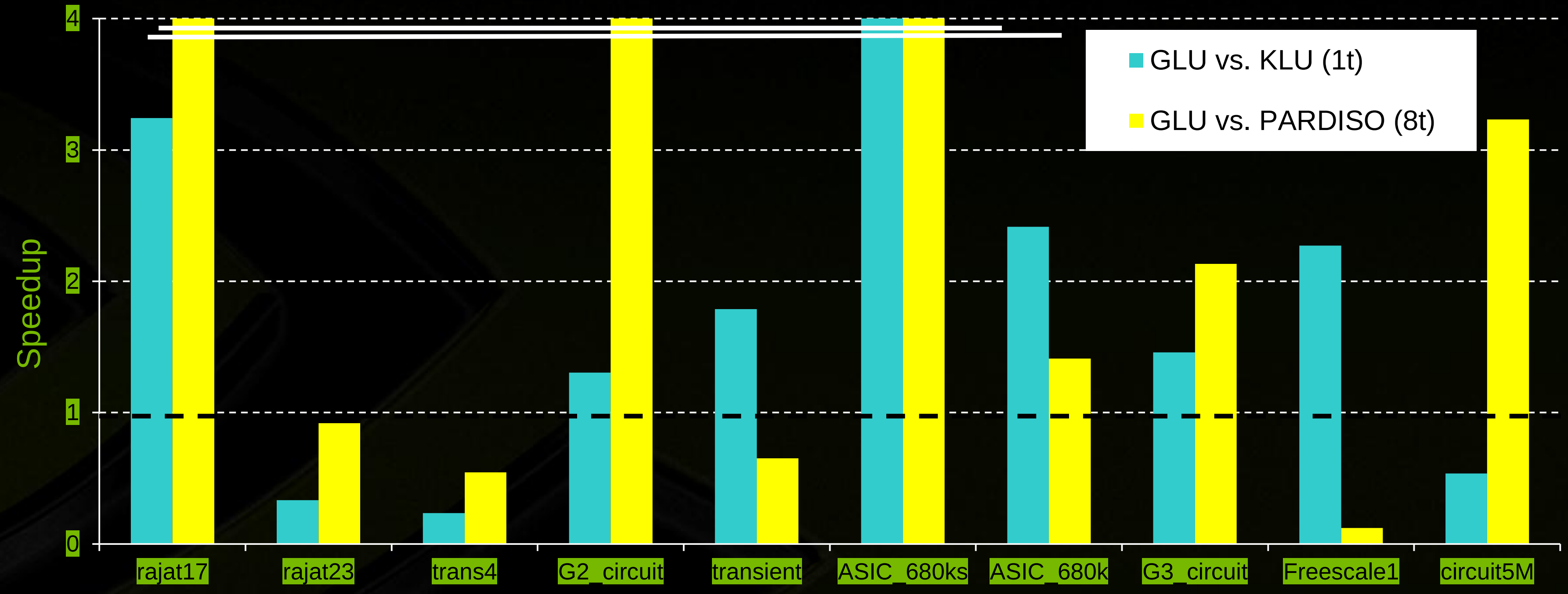
nvAMG on Irregular Grids



Performance may vary based on
OS version and motherboard configuration

- GPU nvAMG (V-cycle, agg2, MC-DILU, 0pre, 3post) on C2090 and K20X
- CPU Fluent AMG (F-cycle, agg8, DILU, 0pre, 3post) on Intel i7-3930K (Sandy Bridge, 6 Core™) @3.2GHz

GLU Library Speedup (K20x)

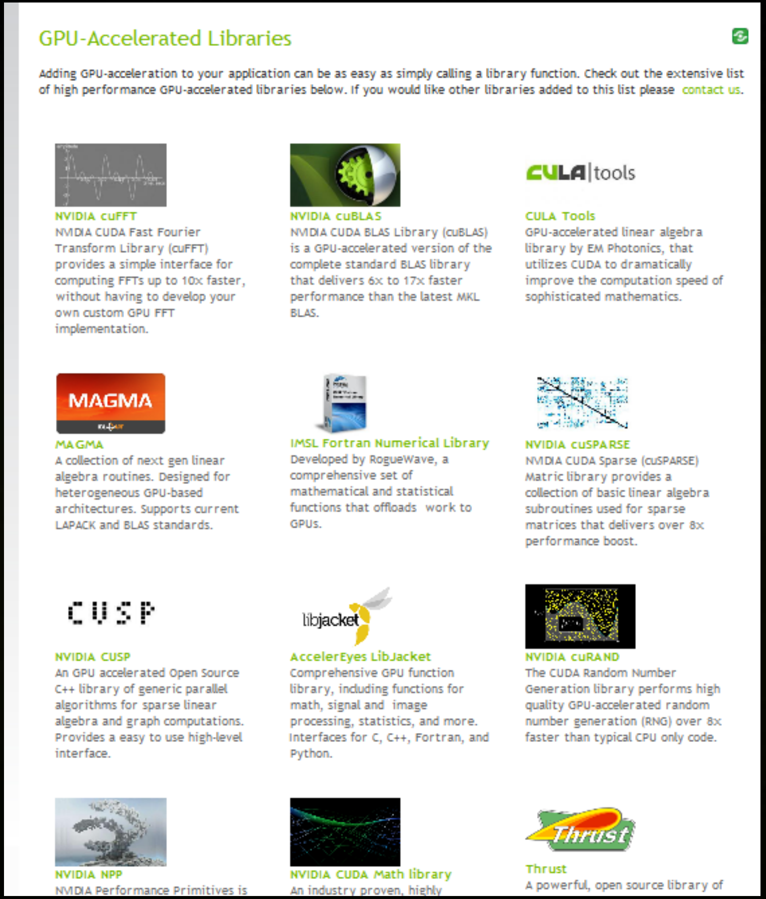


Performance may vary based on OS version and motherboard configuration

- NVIDIA K20, ECC on
- Intel E5-2687w (Sandy Bridge, 8 Core^{TM}) @ 3.1GHz, MKL 10.3.6

The Ecosystem

- There are many great libraries
  - I apologize in advance if I have not mentioned your favorite one
- Explore the library ecosystem
  - Well maintained libraries make life much easier for developers



GPU-Accelerated Libraries

Adding GPU-acceleration to your application can be as easy as simply calling a library function. Check out the extensive list of high performance GPU-accelerated libraries below. If you would like other libraries added to this list please contact us.

NVIDIA cuFFT
NVIDIA CUDA Fast Fourier Transform Library (cuFFT) provides a simple interface for computing FFTs up to 10x faster, without having to develop your own custom GPU FFT implementation.

NVIDIA cuBLAS
NVIDIA CUDA BLAS Library (cuBLAS) is a GPU-accelerated version of the complete standard BLAS library that delivers 6x to 17x faster performance than the latest MKL BLAS.

CULA Tools
GPU-accelerated linear algebra library by EM Photonics, that utilizes CUDA to dramatically improve the computation speed of sophisticated mathematics.

MAGMA
A collection of next gen linear algebra routines. Designed for heterogeneous GPU-based architectures. Supports current LAPACK and BLAS standards.

IMSL Fortran Numerical Library
Developed by RogueWave, a comprehensive set of mathematical and statistical functions that offloads work to GPUs.

NVIDIA cuSPARSE
NVIDIA CUDA Sparse (cuSPARSE) Matric library provides a collection of basic linear algebra subroutines used for sparse matrices that delivers over 8x performance boost.

NVIDIA CUSP
An GPU accelerated Open Source C++ library of generic parallel algorithms for sparse linear algebra and graph computations. Provides a easy to use high-level interface.

AccelerEyes LibJacket
Comprehensive GPU function library, including functions for math, signal and image processing, statistics, and more. Interfaces for C, C++, Fortran, and Python.

NVIDIA cuRAND
The CUDA Random Number Generation library performs high quality GPU-accelerated random number generation (RNG) over 8x faster than typical CPU only code.

NVIDIA NPP
NVIDIA Performance Primitives is

NVIDIA CUDA Math library
An industry proven, highly

Thrust
A powerful, open source library of

https://developer.nvidia.com/gpu-accelerated-libraries

3 Examples

- Matrix multiply vector
  - Very common in scientific computing
  - Use cub to implement
- Mediate filter
  - C, CUDA, also try OpenACC, but failed because of compiler bug
  - Compare performance
- Compute PI with curand
  - MC method

Matrix multiply vector with cub

- Row first
- Algorithm: block bid computes row bid of matrix multiply vector, thread tid in block bid computes data tid of row bid, loop over if block size is smaller than column size

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} AX + BY + CZ \\ DX + EY + FZ \\ GX + HY + IZ \end{bmatrix}$$

Matrix multiply vector with cub(kernel)

```
template<int THREADSPERBLOCK>
void __global__ mxvBlock(int rowSize, int columnSize, const float* __restrict__ d_matrix,
        const float* __restrict__ d_vec, float* __restrict__ d_r){
        int tid = threadIdx.x;
        int bid = blockIdx.x;
        float temp = 0.0f;
        for(int i = tid; i < columnSize; i += blockDim.x){
                temp += d_matrix[bid*columnSize+i]*d_vec[i];
        }

        typedef cub::BlockReduce<float, THREADSPERBLOCK> BR;
        __shared__ typename BR::SmemStorage smem;

        float ret = BR::Sum(smem, temp, blockDim.x);

        if(0 == tid) d_r[blockIdx.x] = ret;
}
```

3*3 Mediate filter

- 2D thread block, every thread computes one pixel
- Every thread need one array with size 9
- Use texture to load pixel

- Testing environment
  - 4096*4096 pixels
  - CPU: Intel Core i7 3930K 6cores, HT
  - GPU: K20c
  - CUDA 5.5, nvcc –O3

3*3 Mediate filter(C version)

```
void medianFilter(int height, int width, unsigned char* restrict src, unsigned char* restrict dst){
        for( int i = 1; i < height-1; i++){
                for( int j = 1; j < width-1; j++){
                        unsigned char a [9];
                        a[0] = src[ i*width+j];                 a[1] = src[ i*width+j+1];
                        a[2] = src[ i*width+j-1];               a[3] = src[(i+1)*width+j];
                        a[4] = src[(i+1)*width+j+1];            a[5] = src[(i+1)*width+j-1];
                        a[6] = src[(i-1)*width+j];              a[7] = src[(i-1)*width+j+1];
                        a[8] = src[(i-1)*width+j-1];
                        for( int ji = 0; ji < 5; ji++)
                                for( int jj = ji+1; jj < 9; jj++)
                                        if (a[ ji ] > a[ jj ]){
                                                unsigned char tmp = a[ji];a[ ji ] = a[ jj ];a[ jj ] = tmp;
                                        }
                        dst[i*width+j] = a[4];
                }
        }
        for( int i = 0; i < width; i++)
                {dst[i] = src[i]; dst[(height-1)*width+i] = src[(height-1)*width+i];  }
        for(int i = 0; i < height; i++)
                {dst[i*width] = src[i*width];           dst[i*width+width-1] = src[i*width+width-1]; }
}
```

1744 ms

3*3 Mediate filter(CUDA version)

```
__global__ void medianFilterInternal(int height, int width, unsigned char* __restrict__ src, unsigned
char* __restrict__ dst){
        int tidx = blockDim.x*blockIdx.x + threadIdx.x;
        int tidy = blockDim.y*blockIdx.y + threadIdx.y;

        bool flag = (tidx > 0) && (tidx < width-1) && (tidy > 0) && (tidy < height-1);
        if(flag){
                unsigned char a [9];
                a[0] = src[ tidy*width+tidx];            a[1] = src[ tidy*width+tidx+1];
                a[2] = src[ tidy*width+tidx-1];          a[3] = src[(tidy+1)*width+tidx];
                a[4] = src[(tidy+1)*width+tidx+1];       a[5] = src[(tidy+1)*width+tidx-1];
                a[6] = src[(tidy-1)*width+tidx];         a[7] = src[(tidy-1)*width+tidx+1];
                a[8] = src[(tidy-1)*width+tidx-1];
        #pragma unroll
                for( int ji = 0; ji < 5; ji++){
                        unsigned char max = a[ji];
                        int index = ji;
                        for( int jj = ji+1; jj < 9; jj++)
                                if(a[jj] > max){ max = a[jj]; index = jj;}
                        if(index != ji){max = a[ji];a[ji] = a[index];  a[index] = max;}
                }
                dst[tidy*width+tidx] = a[4];
        }
}
```

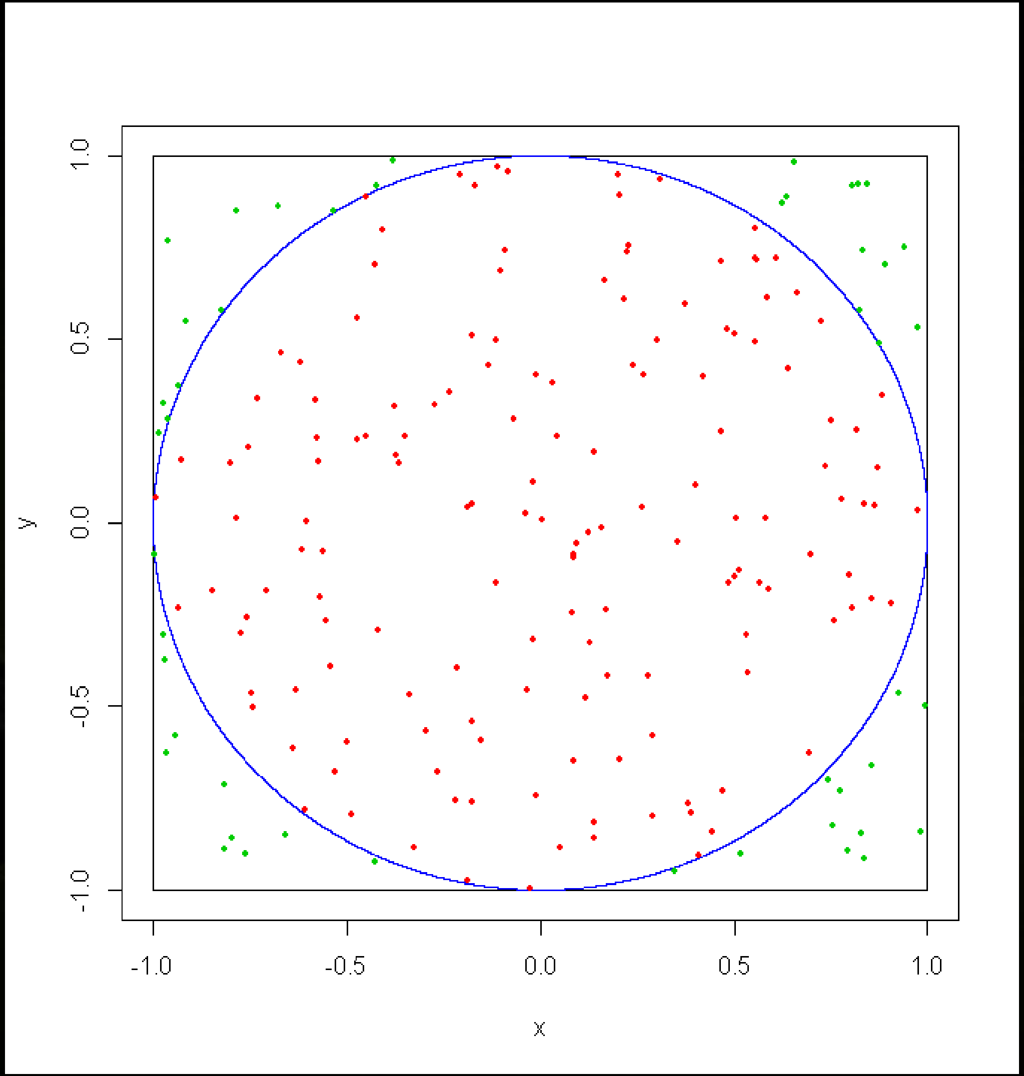
3*3 Mediate filter(CUDA version)

```
void __global__ fillBoundary(int height, int width, const unsigned char* __restrict__ src, unsigned char*
__restrict__ dst){
        int tid = blockDim.x*blockIdx.x + threadIdx.x;
        for( int i = tid; i < width; i += blockDim.x*gridDim.x){
                dst[i] = src[i]; //first row
                dst[(height-1)*width+i] = src[(height-1)*width+i];//the last row
        }

        for( int i = tid; i < height; i += gridDim.x*blockDim.x){
                dst[i*width] = src[i*width];//first column
                dst[i*width+width-1] = src[i*width+width-1];//last column
        }
}
```

9 ms

Compute PI with curand

Area of circle: PI
Area of square: 4



Compute PI with curand(kernel)

```
__device__ inline void getPoint(double &x, double &y, curandState &state){
    x = curand_uniform_double(&state);
    y = curand_uniform_double(&state);
}
__global__ void computeValue(int* results, curandState* rngStates, int seed, int numSamples){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    curand_init(seed, tid, 0, rngStates+tid);
    curandState localState = rngStates[tid];

    int pointsInside = 0;
    for ( int i = tid; i < numSamples; i += blockDim.x*gridDim.x)    {
        double x;
        double y;
        getPoint(x, y, localState);
        if (x * x + y * y < 1.0)        {
            pointsInside++;
        }
    }

    results[tid] = pointsInside;
}
```