Functions

CUDA Function Declarations

	Executed on the:	Callable from only the:
`__device__ float DeviceFunc()`	device	device
`__global__ void  KernelFunc()`	device	host
`__host__   float HostFunc()`	host	host

- __global__ defines a kernel. It must return void.
- A program may have several functions of each kind.
- The same function of any kind may be called multiple times.
- Host == CPU, Device == GPU.

Function Types (1/2)

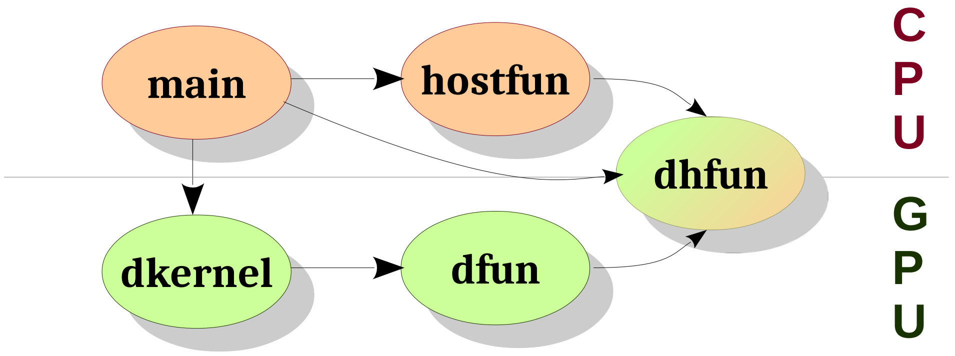
```
#include <stdio.h>
#include <cuda.h>
__host__ __device__ void dhfun() {
        printf("I can run on both CPU and GPU.\n");
}
__device__ unsigned dfun(unsigned *vector, unsigned vectorsize, unsigned id) {
        if (id == 0) dhfun();
        if (id < vectorsize) {
                vector[id] = id;
                return 1;
        } else {
                return 0;
        }
}
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
        unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
        dfun(vector, vectorsize, id);
}
__host__ void hostfun() {
        printf("I am simply like another function running on CPU. Calling dhfun\n");
        dhfun();
}
```

Function Types (2/2)

```
#define BLOCKSIZE       1024
int main(int nn, char *str[]) {
        unsigned N = atoi(str[1]);
        unsigned *vector, *hvector;
        cudaMalloc(&vector, N * sizeof(unsigned));
        hvector = (unsigned *)malloc(N * sizeof(unsigned));

        unsigned nblocks = ceil((float)N / BLOCKSIZE);
        printf("nblocks = %d\n", nblocks);

        dkernel<<<nblocks, BLOCKSIZE>>>(vector, N);
        cudaMemcpy(hvector, vector, N * sizeof(unsigned), cudaMemcpyDeviceToHost);
        for (unsigned ii = 0; ii < N; ++ii) {
                printf("%4d ", hvector[ii]);
        }
        printf("\n");
        hostfun();
        dhfun();
        return 0;
}
```



What are the other arrows possible in this diagram?
How about **dhfun** to **dfun**?

with HostAlloc'ed Memory

__host__ __device__ functions are friends with HostAlloc’ed memory.

```
__host__ __device__ void fun(int *counter) {
        ++*counter;
}
__global__ void printk(int *counter) {
        fun(counter);
        printf("printk (after fun): %d\n", *counter);
}
int main() {
        int *counter;
        cudaHostAlloc(&counter, sizeof(int), 0);

        *counter = 0;
        printf("main: %d\n", *counter);

        printk<<<1, 1>>>(counter);
        cudaDeviceSynchronize();

        fun(counter);
        printf("main (after fun): %d\n", *counter);

        return 0;
}
```

What is the output of this code?

with a Device-only Function

```
__host__ __device__ void fun(int *counter) {
        ++*counter;
        __syncthreads();
}
__global__ void printk(int *counter) {
        fun(counter);
        printf("printk (after fun): %d\n", *counter);
}
int main() {
        int *counter;
        cudaHostAlloc(&counter, sizeof(int), 0);

        *counter = 0;
        printf("main: %d\n", *counter);

        printk <<<1, 1>>>(counter);
        cudaDeviceSynchronize();

        fun(counter);
        printf("main (after fun): %d\n", *counter);

        return 0;
}
```

__syncthreads() is not available on CPU.

with a CPU-only Memory

```
__host__ __device__ void fun(int *counter) {
        ++*counter;
}
__global__ void printk(int *counter) {
        fun(counter);
        printf("printk (after fun): %d\n", *counter);
}
int main() {
        int *counter;
        // cudaHostAlloc(&counter, sizeof(int), 0);
       cudaMalloc(&counter, sizeof(int));

        *counter = 0;
        printf("main: %d\n", *counter);

        printk <<<1, 1>>>(counter);
        cudaDeviceSynchronize();

        fun(counter);
        printf("main (after fun): %d\n", *counter);

        return 0;
}
```

counter cannot be accessed on CPU.

Global Variables

```
int counter;

__host__ __device__ void fun() {
        ++counter;
}
__global__ void printk() {
        fun();
        printf("printk (after fun): %d\n", counter);
}
int main() {

        counter = 0;
        printf("main: %d\n", counter);

        printk <<<1, 1>>>();
        cudaDeviceSynchronize();

        fun();
        printf("main (after fun): %d\n", counter);

        return 0;
}
```

counter cannot be accessed on **GPU**.

Global Variables

```
__host__ __device__ int counter;

__host__ __device__ void fun() {
        ++counter;
}
__global__ void printk() {
        fun();
        printf("printk (after fun): %d\n", counter);
}
int main() {

        counter = 0;
        printf("main: %d\n", counter);

        printk <<<1, 1>>>();
        cudaDeviceSynchronize();

        fun();
        printf("main (after fun): %d\n", counter);

        return 0;
}
```

Variables cannot be declared as __host__.

Global Variables

```
__device__ int counter;

__host__ __device__ void fun() {
        ++counter;
}
__global__ void printk() {
        fun();
        printf("printk (after fun): %d\n", counter);
}
int main() {

        printk <<<1, 1>>>();
        cudaDeviceSynchronize();

        return 0;
}
```

Warning during compilation, but works fine.

Classwork

Write a CUDA code to increment all elements in an array. Call this code from host as well as device.

Classwork: Can you avoid the for loop in **fun**?

```
__host__ __device__ void fun(int *arr) {
        for (unsigned ii = 0; ii < N; ++ii)
                ++arr[ii];
}
__global__ void dfun(int *arr) {
        fun(arr);
}
int main() {
        int arr[N], *darr;

        cudaMalloc(&darr, N * sizeof(int));

        for (unsigned ii = 0; ii < N; ++ii)
                arr[ii] = ii;
        cudaMemcpy(darr, arr, N * sizeof(int),
                                cudaMemcpyHostToDevice);

        fun(arr);
        dfun<<<1, 1>>>(darr);
        cudaDeviceSynchronize();

        return 0;
}
```

Host-centric, sequential on GPU

Classwork

Write a CUDA code to increment all elements in an array. Call this code from host as well as device.

Classwork: Can you avoid the for loop in **fun**?

Classwork: What if I don't like the for loop in main, but still want GPU-parallel code?

```
__host__ __device__ void fun(int *arr) {
                ++(*arr);
}
__global__ void dfun(int *arr) {
        fun(arr + threadIdx.x);
}
int main() {
        int arr[N], *darr;

        cudaMalloc(&darr, N * sizeof(int));

        for (unsigned ii = 0; ii < N; ++ii)
                arr[ii] = ii;
        cudaMemcpy(darr, arr, N * sizeof(int),
                              cudaMemcpyHostToDevice);
        for (unsigned ii = 0; ii < N; ++ii)
                fun(arr + ii);
        dfun<<<1, N>>>(darr);
        cudaDeviceSynchronize();

        return 0;
}
```

Device-centric, sequential on CPU

Classwork

Write a CUDA code to increment all elements in an array. Call this code from host as well as device.

Classwork: Can you avoid the for loop in **fun**?

Classwork: What if I don’t like the for loop in main, but still want GPU-parallel code?

```
__host__ __device__ void fun(int *arr, int nn) {
        for (unsigned ii = 0; ii < nn; ++ii)
                ++arr[ii];
}
__global__ void dfun(int *arr) {
        fun(arr + threadIdx.x, 1);
       // need to change for more blocks.
}
int main() {
        int arr[N], *darr;

        cudaMalloc(&darr, N * sizeof(int));

        for (unsigned ii = 0; ii < N; ++ii)
                arr[ii] = ii;
        cudaMemcpy(darr, arr, N * sizeof(int),
                                    cudaMemcpyHostToDevice);

        fun(arr, N);
        dfun<<<1, N>>>(darr);
        cudaDeviceSynchronize();

        return 0;
}
```

Thrust

- Thrust is a parallel algorithms library (similar in spirit to STL on CPU).
- Supports vectors and associated transforms.
- Programmer is oblivious to where code executes – on CPU or GPU.