

# Functions

Rupesh Nasre.

IIT Madras  
January 2021

# CUDA Function Declarations

	Executed on the:	Callable from only the:
<code>__device__ float DeviceFunc()</code>	device	device
<code>__global__ void KernelFunc()</code>	device	host
<code>__host__ float HostFunc()</code>	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)

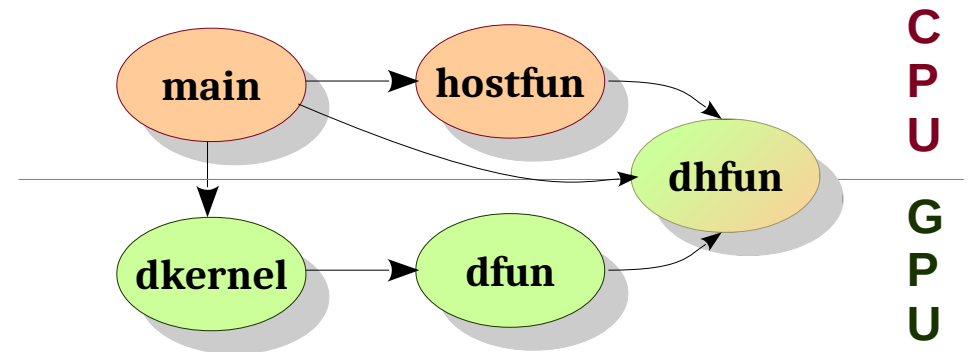
```
#include <stdio.h>
#include <cuda.h>
__host__ __device__ void dhfun() {
    printf("I can run on both CPU and GPU.\n");
}
__device__ unsigned dfun(unsigned *vector, unsigned vectorsize, unsigned id) {
    if (id == 0) dhfun();
    if (id < vectorsize) {
        vector[id] = id;
        return 1;
    } 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: Pranav's idea

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.
- Makes use of C++ features such as functors, and `__host__ __device__` functions.

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <iostream>
int main(void) {
    // H has storage for 4 integers
    thrust::host_vector<int> H(4);
    // initialize individual elements
    H[0] = 14; H[1] = 20; H[2] = 38; H[3] = 46;
    // H.size() returns the size of vector H
    std::cout << "H has size " << H.size() << std::endl;
    // print contents of H
    for(int i = 0; i < H.size(); i++) std::cout << "H[" << i << "] = " << H[i] << std::endl;
    // resize H
    H.resize(2);
    std::cout << "H now has size " << H.size() << std::endl;
    // Copy host_vector H to device_vector D
    thrust::device_vector<int> D = H;
    // elements of D can be modified
    D[0] = 99; D[1] = 88;
    // H and D are automatically deleted when the function returns
    return 0;
}
```

```

#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/sequence.h>
#include <iostream>

int main(void) {
    // initialize all ten integers of a device_vector to 1
    thrust::device_vector<int> D(10, 1);

    // set the first seven elements of a vector to 9
    thrust::fill(D.begin(), D.begin() + 7, 9);

    // initialize a host_vector with the first five elements of D
    thrust::host_vector<int> H(D.begin(), D.begin() + 5);

    // set the elements of H to 0, 1, 2, 3, ...
    thrust::sequence(H.begin(), H.end());

    // copy all of H back to the beginning of D
    thrust::copy(H.begin(), H.end(), D.begin());

    // print D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    return 0;
}

```



# Thrust Details

```
thrust::host_vector<int> hnums(1024);  
thrust::device_vector<int> dnums;  
  
dnums = hnums; // calls cudaMemcpy  
  
// initialization.  
thrust::device_vector<int> dnum2(hnums.begin(), hnums.end());  
hnums = dnum2; // array resizing happens automatically.  
  
std::cout << dnums[3] << std::endl;  
  
thrust::transform(dsrc.begin(), dsrc.end(), dsrc2.begin(),  
                  ddst.begin(), addFunc);
```

# Thrust Functions

- `find(begin, end, value);`
- `find_if(begin, end, predicate);`
- `copy, copy_if.`
- `count, count_if.`
- `equal.`
- `min_element, max_element.`
- `merge, sort, reduce.`
- `transform.`
- ...

# Thrust Algorithms

- Dual implementations: **host** and **device**
- Iterators as arguments must be on the same device
  - except *copy*, which can copy across devices
  - Otherwise, compiler issues error

```

#include <thrust/device_vector.h>
#include <thrust/transform.h>
#include <thrust/sequence.h>
#include <thrust/copy.h>
#include <thrust/fill.h>
#include <thrust/replace.h>
#include <thrust/functional.h>
#include <iostream>
int main(void) {
    // allocate three device_vectors with 10 elements
    thrust::device_vector<int> X(10);
    thrust::device_vector<int> Y(10);
    thrust::device_vector<int> Z(10);
    // initialize X to 0,1,2,3, ....
    thrust::sequence(X.begin(), X.end());
    // compute Y = -X
    thrust::transform(X.begin(), X.end(), Y.begin(), thrust::negate<int>());
    // fill Z with twos
    thrust::fill(Z.begin(), Z.end(), 2);
    // compute Y = X mod 2
    thrust::transform(X.begin(), X.end(), Z.begin(), Y.begin(), thrust::modulus<int>());
    // replace all the ones in Y with tens
    thrust::replace(Y.begin(), Y.end(), 1, 10);
    // print Y
    thrust::copy(Y.begin(), Y.end(), std::ostream_iterator<int>(std::cout, "\n"));
    return 0;
}

```

# Thrust User-Defined Functors

```
// calculate result[] = (a * x[]) + y[]
struct saxpy {
    const float _a;
    saxpy(int a) : _a(a) { }

    __host__ __device__
    float operator()(const float& x, const float& y) const {
        return _a * x + y;
    }
};

thrust::device_vector<float> x, y, result;    ... // populate x, y.
thrust::transform(x.begin(), x.end(), y.begin(), result.begin(), saxpy(a));
```

# Classwork

- Create two 32-element vectors:
  - $X$  on host,  $Y$  on device
- Fill  $X$  with 10, fill  $Y$  with sequence 0..31
- Compute  $X = X - Y$
- Compute  $Z = X * Y$ 
  - // element-wise multiplication

# Thrust Reductions

- Recall reductions in  $\log(n)$  barriers
- No need to worry about blocks, synchronization.

```
int x, y;  
thrust::host_vector<int> hvec;  
thrust::device_vector<int> dvec;  
// (thrust::reduce is a sum operation by default)  
x = thrust::reduce(hvec.begin(), hvec.end()); // on CPU  
y = thrust::reduce(dvec.begin(), dvec.end()); // on GPU
```

```
y = thrust::reduce(dvec.begin(), dvec.end(),  
                  (int)0, thrust::plus<int>());
```

**Classwork:** Implement **count** using reduction.  
For instance, I want to find the number of  
occurrences of an element in a vector.

```

struct mycount {
    int _a;
    mycount(int a):_a(a){}
    __host__ __device__
    int operator()(const int x, const int y) const {
        return (y == _a ? x + 1 : x);
    }
};

int main() {
    thrust::host_vector<int> vec(10, 0);
    vec[1] = 5;
    vec[4] = 5;
    vec[9] = 5;

    int result = thrust::reduce(vec.begin(), vec.end(),
                                (int)0, mycount(5));
    std::cout << result << std::endl;
    return 0;
}

```



# Prefix Sum / Scan

```
#include <thrust/scan.h>
```

```
int data[6] = {1, 0, 2, 2, 1, 3};
```

```
// inclusive scan
```

```
thrust::inclusive_scan(data, data + 6, data);
```

```
// data is now {1, 1, 3, 5, 6, 9}
```

```
thrust::exclusive_scan(data, data + 6, data);
```

```
// data is now {0, 1, 1, 3, 5, 6}
```

# Classwork

- What is the output of the following code?

```
int data[] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
int sizedata = sizeof(data) / sizeof(*data);
thrust::plus<int> binop;

thrust::exclusive_scan(data, data + sizedata, data, 5, binop);

for (unsigned ii = 0; ii < sizedata; ++ii) {
    std::cout << data[ii] << " ";
}
std::cout << std::endl;
```

5 0 0 2 -1 1 5 5 4 6

# Classwork

- What is the output of the following code?

```
int data[] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
int sizedata = sizeof(data) / sizeof(*data);
thrust::plus<int> binop;

thrust::inclusive_scan(data, data + sizedata, data, 5, binop);

for (unsigned ii = 0; ii < sizedata; ++ii) {
    std::cout << data[ii] << " ";
}
std::cout << std::endl;
```

**Compile-time error: Why?**

# Classwork

- What is the output of the following code?

```
int data[] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
int sizedata = sizeof(data) / sizeof(*data);
thrust::plus<int> binop;

thrust::inclusive_scan(data, data + sizedata, data, 5, binop);

for (unsigned ii = 0; ii < sizedata; ++ii) {
    std::cout << data[ii] << " ";
}
std::cout << std::endl;
```

**-5 -5 -3 -6 -4 0 0 -1 1 9**

# Classwork

- What is the output of the following code?

```
int data [] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
int data2[] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};
int sizedata = sizeof(data) / sizeof(*data);
thrust::plus<int> binop;

thrust::inclusive_scan(data, data+sizedata, data, 5, binop);
thrust::exclusive_scan(data, data+sizedata, data2, 0, binop);

for (unsigned ii = 0; ii < sizedata; ++ii)
    std::cout << data2[ii] << " ";
std::cout << std::endl;
```

**0 -5 -10 -13 -19 -23 -23 -23 -24 -23**

# Classwork: Find output

```
int main() {  
    int data[] = {-5, 0, 2, -3, 2, 4, 0, -1, 2, 8};  
    int sizedata = sizeof(data) / sizeof(*data);  
    thrust::maximum<int> binop;  
    thrust::exclusive_scan(data, data + sizedata,  
                           data, 1, binop);  
    for (unsigned ii = 0; ii < sizedata; ++ii) {  
        std::cout << data[ii] << " ";  
    }  
    std::cout << std::endl;  
    return 0;  
}
```

1 1 1 2 2 2 4 4 4 4

# Set Operations

```
#include <thrust/set_operations.h>
```

```
...
```

```
int A1[6] = {0, 1, 3, 4, 5, 6, 9};
```

```
int A2[5] = {1, 3, 5, 7, 9};
```

```
int result[N];
```

```
thrust::set_difference(A1, A1+6, A2, A2+5, result);
```



Must be sorted

result is {0, 4, 6}.

# Set Operations

```
#include <thrust/set_operations.h>
```

```
...
```

```
int A1[] = {9, 6, 5, 4, 3, 1, 0};
```

```
int A2[5] = {9, 7, 5, 3, 1};
```

```
int result[N];
```

```
thrust::set_difference(A1, A1+7, A2, A2+5, result,  
                      thrust::greater<int>());
```

result is {6, 4, 0}.



# Sorting

```
#include <thrust/sort.h>
```

```
...
```

```
const int N = 6;
```

```
int A[N] = {1, 4, 2, 8, 5, 7};
```

```
thrust::sort(A, A + N);
```

```
// A is now {1, 2, 4, 5, 7, 8}
```

```
int keys[N] = { 1, 4, 2, 8, 5, 7};
```

```
char values[N] = {'a', 'b', 'c', 'd', 'e', 'f'};
```

```
thrust::sort_by_key(keys, keys + N, values);
```

```
// keys is now { 1, 2, 4, 5, 7, 8}
```

```
// values is now {'a', 'c', 'b', 'e', 'f', 'd'}
```