CUDA C++

25. APRIL 2018 | JAN H. MEINKE

CUDA AND C++

- CUDA host code has been compiled as C++ code since version 2!
- Some C++ features, e.g., templates have been supported since CUDA 1.x
- C++ 11 features supported in host *and* device code since CUDA 7

A SAMPLE OF C++ 11 FEATURES

auto

template

memory management

range-based for loops

lambdas

WRITING KERNELS FOR DIFFERENT DATA TYPES

```
__global__ void saxpy(float alpha, float* x, float* y, size_t n){
    auto i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < n){
        y[i] = a * x[i] + y[i];
    }
}
```

WRITING KERNELS FOR DIFFERENT DATA TYPES

```
__global__ void daxpy(double alpha, double* x, double* y, size_t n){
    auto i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < n){
        y[i] = a * x[i] + y[i];
    }
}
```

WRITING KERNELS FOR DIFFERENT DATA TYPES

```
template <typename T>
__global__ void axpy(T alpha, T* x, T* y, size_t n){
    auto i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < n){
        y[i] = a * x[i] + y[i];
    }
}
```

Exercise

CUDA++/exercises/tasks/gemm

Compile with make.

TRANSPARENT TYPES

```
class Managed {
public:
  void *operator new(size_t len) {
    void *ptr;
    cudaMallocManaged(&ptr, len);
    cudaDeviceSynchronize();
    return ptr;
  }

  void operator delete(void *ptr) {
    cudaDeviceSynchronize();
    cudaFree(ptr);
  }
};
```

Closely modeled after “Unified Memory in CUDA 6” (see Refs)

TRANSPARENT TYPES

```
template <class T>
class Array : public Managed {
    size_t n;
    T* data;

public:
  Array (const Array &a) {
    n = a.n;
    cudaMallocManaged(&data, n);
    memcpy(data, a.data, n);
  }
   // Also have to implement operator[], for example
};
```

TRANSPARENT TYPES

```
// Pass-by-reference version
__global__ void kernel_by_ref(Array &data) { ... }
// Pass-by-value version
__global__ void kernel_by_val(Array data) { ... }

int main(void) {
  Array *a = new Array;
  ...
  // pass data to kernel by reference
  kernel_by_ref<<<1,1>>>(*a);
  // pass data to kernel by value -- this will create a copy
  kernel_by_val<<<1,1>>>(*a);
}
```

FUNCTION OBJECT (AKA FUNCTOR)

```
template <class T>
class In_range {
    const T val1;
    const T val2;
public:
    In_range(const T& v1, const T& v2) : val1(v1), val2(v2) {}
    bool operator()(const T& x) const {return (x >= val1 && x < val2);}
};
```

Can be used, e.g., in std::count():

```
std::count_if(v.begin(), v.end(), In_range<int>(3, 6));
```

LAMBDAS

```
auto lambda = [](const int& x){return (x >= 3 && x < 6);}
```

Can be used, e.g., in std::count_if():

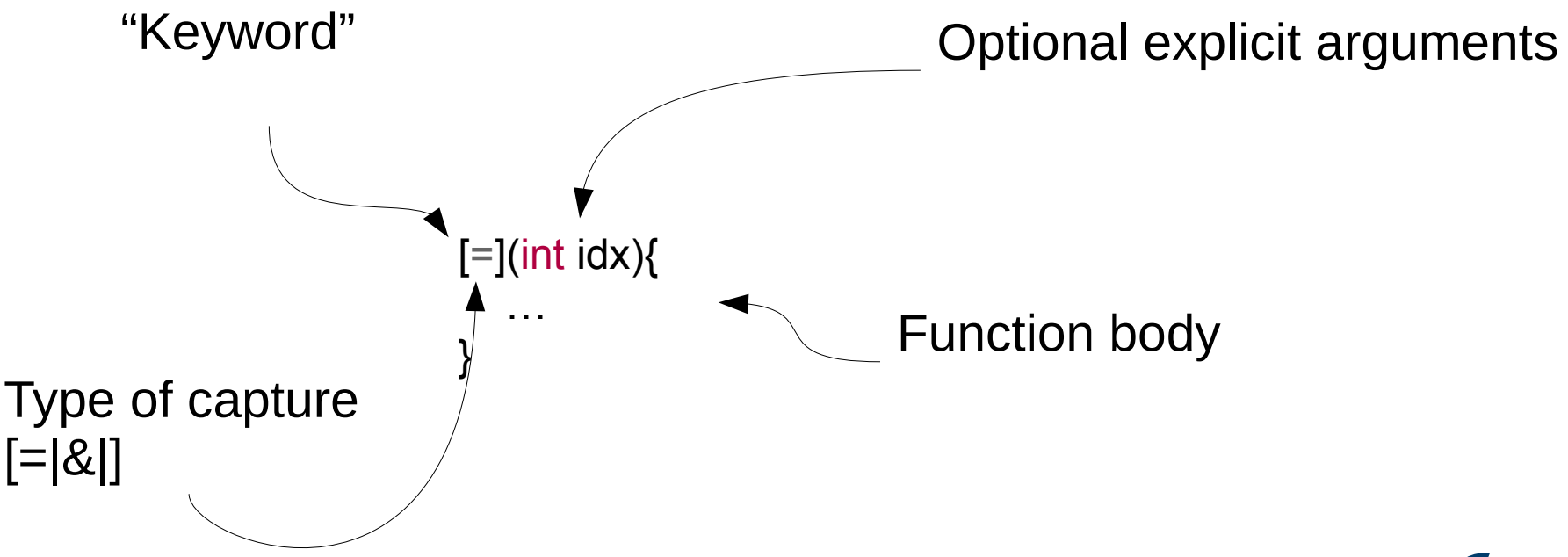
```
std::count_if(v.begin(), v.end(), [](const int& x){return (x >= 3 && x < 6);});
```

LAMBDAS

```
std::vector<int> v {5, 1, 1, 3, 1, 4, 1, 3, 3, 2};
int a = 3;
int b = 6;
auto lambda = [&](const int x){return (x >= a && x < b);}
auto ct36 = std::count_if(v.begin(), v.end(), lambda);
```

LAMBDAS

Lambdas are anonymous functions that can capture variables.



THRUST::FOR_EACH + LAMBDAS

c.f. std::for_each

```
#include <thrust/for_each.h>
#include <thrust/execution_policy.h>

constexpr int gpuThreshold = 10000;
void scale_vector(float *x, float *y, float a, int N) {
    auto r = thrust::counting_iterator<int>(0);
    auto lambda = [=] __host__ __device__ (int i) { // since CUDA 8
      y[i] = a * x[i];};
    if(N > gpuThreshold) // needs to be defined outside
      thrust::for_each(thrust::device, r, r+N, lambda);
    else
      thrust::for_each(thrust::host, r, r+N, lambda);
}
```

Exercise

CUDA++/exercises/tasks/for_each

Compile with make.

THRUST ON DEVICE

```
__global__
void xyzw_frequency_thrust_device(int *count, char *text, int n)
{
  const char letters[] { 'x','y','z','w' };

  *count = thrust::count_if(thrust::device, text, text+n, [=](char c) {
    for (const auto x : letters)
      if (c == x) return true;
    return false;
  });
}
```

REFERENCES

- C++11 in CUDA: Variadic Templates - https://devblogs.nvidia.com/parallelforall/cplusplus-11-in-cuda-variadic-templates
- managed_allocator/README.md at master · jaredhoberock/managed_allocator · GitHub - https://github.com/jaredhoberock/managed_allocator/blob/master/README.md
- C++11 Archives | Parallel Forall - https://devblogs.nvidia.com/parallelforall/tag/c11
- The Saint on Porting C++ Classes to CUDA with Unified Memory - https://devblogs.nvidia.com/parallelforall/the-saint-porting-c-classes-cuda-unified-memory

REFERENCES

- Unified Memory in CUDA 6 - https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6
- Faster Parallel Reductions on Kepler https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler
- CUDA 7.5 https://devblogs.nvidia.com/parallelforall/new-features-cuda-7-5/
- CUDA 8.0 https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/