UNIVERSITÀ DEGLI STUDI FIRENZE

GPU programming basics

Prof. Marco Bertini

Data parallelism: GPU computing

CUDA: libraries

Why use libraries ?

- Libraries are one of the most efficient ways to program GPUs, because they encapsulate the complexity of writing optimized code for common algorithms into high-level, standard interfaces.
- There is a wide variety of high-performance libraries available for NVIDIA GPUs.

Thrust

- Thrust is a library that has been included in CUDA
  - Inspired by STL. Just include the header: it's a template library.
  - Provides many algorithms like reduce, sort, scan, transformations, search.
  - Includes OpenMP backend for multicore programming
  - Allows transparent use of GPU

Thrust Hello World

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

// generate 200K random numbers on the host
thrust::host_vector<int> h_vec(300000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// transfer data to device
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

Thrust vector

- Thrust provides two vector containers, `host_vector` and `device_vector`.
  As the names suggest, `host_vector` is stored in host memory while `device_vector` lives in GPU device memory.

- Thrust's vector containers are just like `std::vector` in the C++ STL.

Thrust iterators

- They point to regions of a vector
- Can be used like pointers
  - Can be converted to raw pointers to interface with CUDA

```
thrust::device_vector<int>::iterator begin =
                                    d_vec.begin();
int * d_ptr = thrust::raw_pointer_cast(begin);
kernel<<<10, 128>>>(d_ptr);
```

From CUDA to Thrust

- Raw pointers can be used in Thrust. Once the raw pointer has been wrapped by a device_ptr it can be used like an ordinary Thrust iterator.

```
int* d_ptr;
cudaMalloc((void**)&d_ptr, N);
thrust::device_ptr<int> d_vec =
            thrust::device_pointer_cast(d_ptr);
thrust::sort(d_vec, d_vec+N);
cudaFree(d_ptr);
```

Thrust: reduction

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <algorithm>

// 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>());
```

CuBLAS

- Many scientific computer applications need high-performance matrix algebra. BLAS is a famous (very optimized) library for such operations.
- The NVIDIA cuBLAS library is a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS).
- Include <cublas.h>
- Link the CuBLAS library files
  - e.g. in CMAKE:
    cuda_add_cublas_to_target(**target_name**)

CuBLAS data layout

- Historically BLAS has been developed in Fortran, and to maintain compatibility CuBLAS uses column-major storage, and 1-based indexing.

- For natively written C and C++ code, one would most likely choose 0-based indexing, in which case the array index of a matrix element in row "i" and column "j" can be computed via the following macro:

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

where `ld` refers to the leading dimension of the matrix, which in the case of column-major storage is the number of rows of the allocated matrix

Example:

```
float* a = (float *)malloc (M * N * sizeof (*a));

for (j = 0; j < N; j++) {
    for (i = 0; i < M; i++) {
        a[IDX2C(i,j,M)] = (float)(i * M + j + 1);
    }
}
```

- For natively written C and C++ code, one would most likely choose 0-based indexing, in which case the array index of a matrix element in row "i" and column "j" can be computed via the following macro:

```
#define IDX2C(i,j,ld) (((j)*(ld))+(i))
```

where `ld` refers to the leading dimension of the matrix, which in the case of column-major storage is the number of rows of the allocated matrix

CuBLAS SAXPY

```
float* d_x;
cudaMalloc((void**)&d_x,N*sizeof(float));
float* d_y;
cudaMalloc((void**)&d_y,N*sizeof(float));

cublasInit(); // init. CuBLAS context
// copy vectors to device memory
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on N elements
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);
cublasShutdown();
```

Fast math

- Adding -use_fast_math option forces to use intrinsic math functions for several operations like divisions, log, exp and sin/cos/tan
    - Work only in device code, of course
- These are single precision functions, so they could be less precise than working on doubles.

C++

- Latest versions of CUDA support (since CUDA 7.0) more and more C++11 features like:
  - auto, lambda expressions, rvalues, nullptr, default and deleted methods.
- C++11 concurrency is completely missing, though.
- Since CUDA 9.0 also almost all C++14 features are supported.

Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufmann - 2nd edition - Chapt. 16

  or

  Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufmann - 3rd edition - Appendix B