Back-end building blocks: Flask, Mongoose, Boost, CUDA

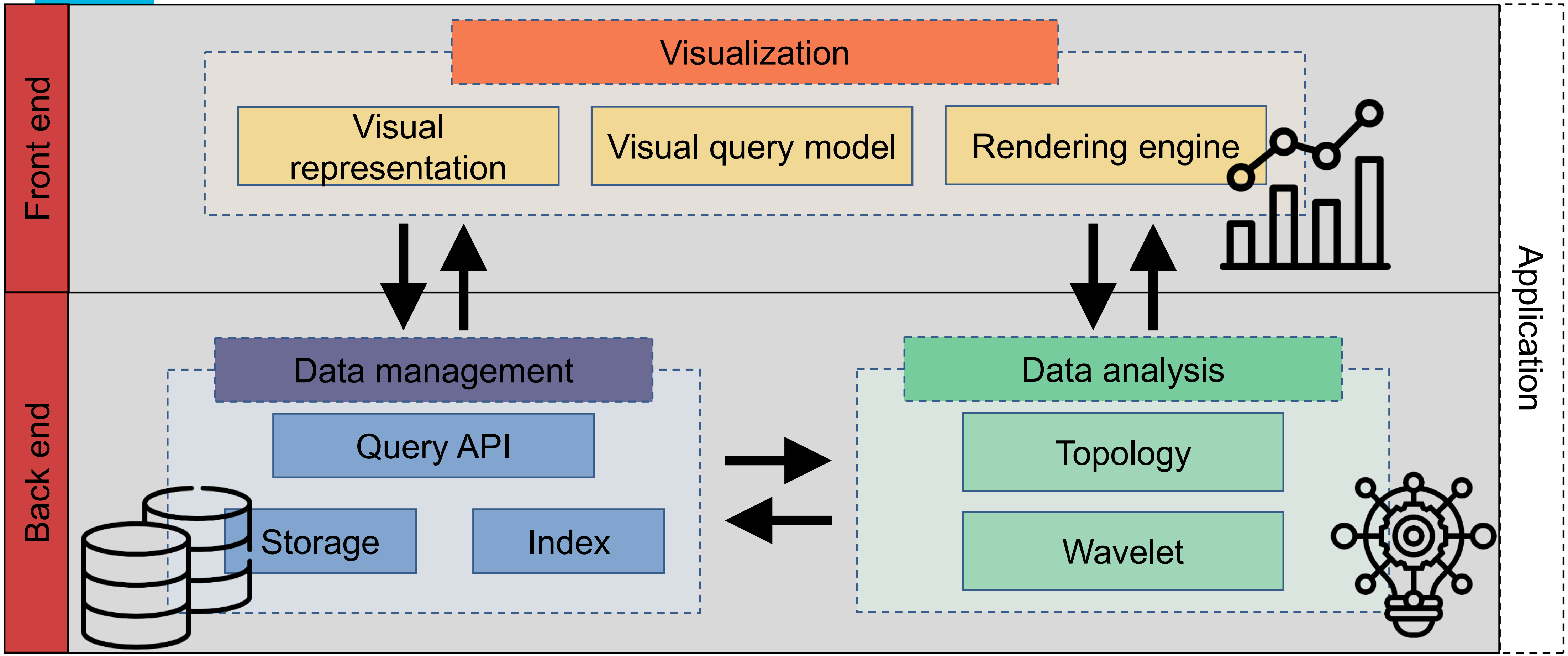
CS594: Big Data Visualization & Analytics

Fabio Miranda

https://fmiranda.me

UIC COMPUTER SCIENCE

Big data visualization system



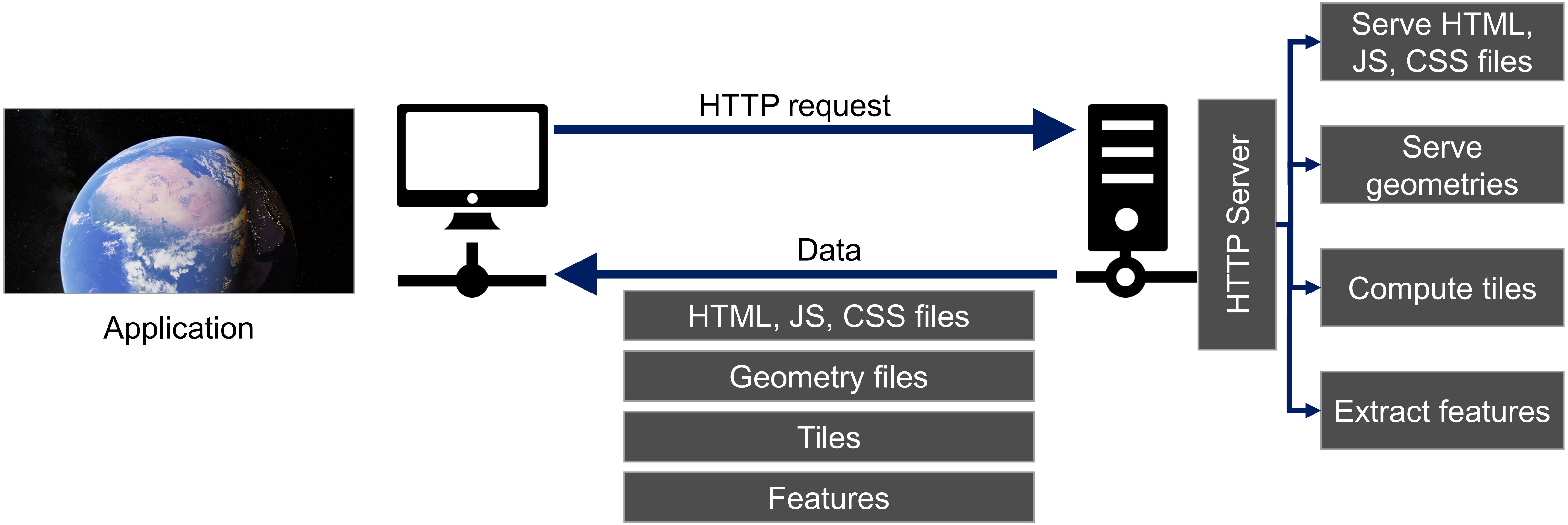
Big data visualization system

- Why separate front-end and back-end development?
  - Separation of concerns between presentation layer (front end) and data layer (back end).
  - Easily mapped to a client-server model.
    - Client: front end
    - Server: back end
  - Easy deployment.

Overview

- Front-end and back-end communication:
    - Flask (Python)
    - Mongoose (C / C++)
- Back-end building blocks:
    - Boost
    - Qt
    - CUDA

Client and server



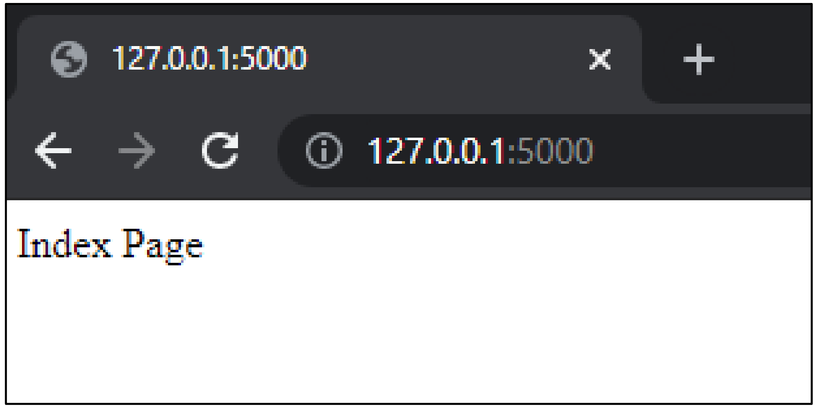
Flask

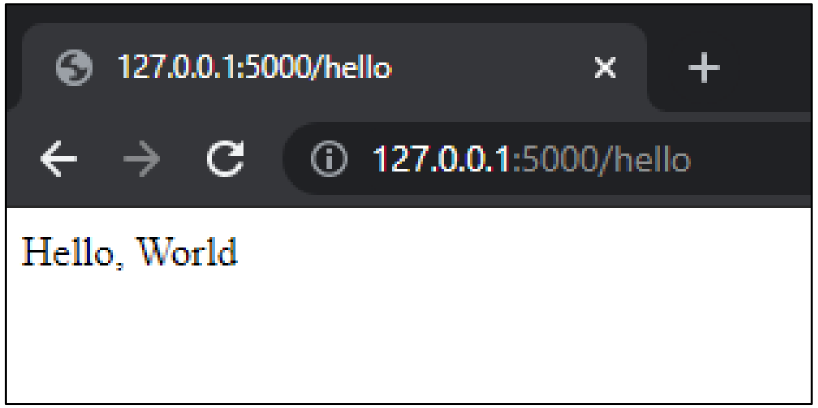
- Python framework for developing web applications.
- Lightweight applications (when compared to Django).
- Easy integration between front-end and back-end components.

Flask: minimal application

```python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return 'Index Page'

@app.route('/hello')
def hello():
    return 'Hello, World'
```

```
user@DESKTOP MINGW64 ~/
$ export FLASK_APP=example FLASK_ENV=development
$ flask run
```





Flask: minimal application

- Web applications use different HTTP methods when accessing URLs.
- You can use the `methods` argument to handle different HTTP methods.

```
from flask import request

@app.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == 'POST':
        return do_the_login()
    else:
        return show_the_login_form()
```

HTTP request methods

- HTTP is designed to enable communication between clients and servers.
- HTTP works as a request-response protocol between a client and a server.
- HTTP methods:
  - **GET**
  - **POST**
  - PUT
  - HEAD
    DELETE
  - PATCH
  - OPTIONS

HTTP request methods

- GET:
  - Used to request data from a specified resource.
  - One of the most common HTTP methods.

```
/test?name1=value1&name2=value2
```

- POST:
  - Used to send data to a server.
  - Data sent to the server with POST is stored in the **request body** of the HTTP request.

```
POST /test HTTP/1.1
Host: w3schools.com
name1=value1&name2=value2
```

HTTP request methods

	GET	POST
Back button / Reload	Harmless	Data will be re-submitted
Bookmarked	Can be bookmarked	Cannot be bookmarked
Cached	Can be cached	Not cached
History	**Parameters remain in browser history**	**Parameters are not saved in browser history**
Restrictions on data length	**Length of a URL is limited: 2048 characters**	**No restrictions**
Restrictions on data type	**Only ASCII characters**	**No restrictions. Binary data is also allowed**
Security	Less secure, data sent is part of the URL	Safer, parameters are not stored in browser history
Visibility	Data is visible to everyone in the URL	Data is not displayed in the URL

From: https://www.w3schools.com/tags/ref_httpmethods.asp

Flask and HTTP methods

```python
from flask import Flask
from flask import request

@app.route('/example/name1=<value1>&name2=<value2>', methods = ['GET', 'POST'])
def example(value1, value2):
    if request.method == 'GET':
        # ...
        pass
    if request.method == 'POST':
        data = request.form # a multidict containing POST data
        # ...
        pass
    else:
        # POST Error 405 Method Not Allowed
        pass
```

Mongoose

- Networking library for C/C++.
- Event-driven non-blocking APIs for TCP, UDP, HTTP, …
- Easy to integrate: mongoose.c and mongoose.h, that is it.

Mongoose: minimal application

- Declare and initialize an event manager:

```
struct mg_mgr mgr;
mg_mgr_init(&mgr);
```

- Create connections with an event handler:

```
struct mg_connection *c = mg_http_listen(&mgr, "0.0.0.0:8000", fn, arg);
```

- Create an event loop:

```
for (;;) {
  mg_mgr_poll(&mgr, 1000);
}
```

Mongoose: minimal application

- Event handler function defines connection's behavior

```
static void fn(struct mg_connection *c, int ev, void *ev_data, void *fn_data) {
  if (ev == MG_EV_HTTP_REQUEST)
  {
    struct http_message *hm = (struct http_message *) p;
    QString uri = QString::fromStdString(std::string(hm->uri.p+1,hm->uri.len));
    QString poststr = QString::fromStdString(std::string(hm->body.p,hm->body.len));
    QJsonDocument post = QJsonDocument::fromJson(poststr.toUtf8());

    if(uri.startsWith("example"))
    {
      QString json;
      Server::getInstance().startQuery(uri, post, json);
      mg_send_head(c, 200, json.length(), "Content-Type: text/plain");
      mg_printf(c, "%.*s", json.length(), json.toStdString().c_str());
    }
    else
    {
      mg_serve_http(c, (struct http_message *) p, s_http_server_opts); //Serve static content
    }
  }
}
```

Back-end building blocks

- Boost
- QT
- CUDA

Boost

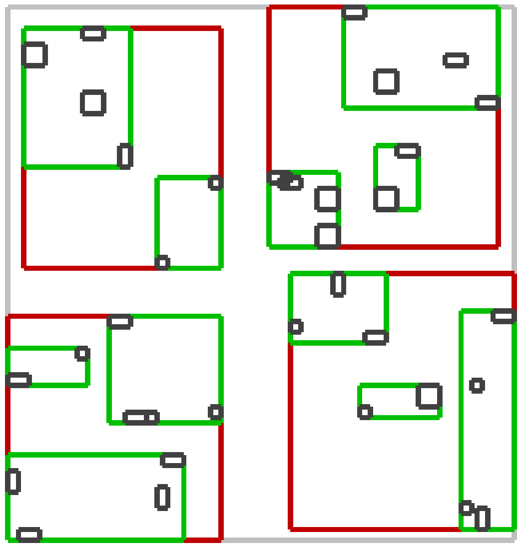
- Libraries for C++ that provide support for linear algebra, multithreading, image processing, etc.
- The most used C++ library (apart of the STL library).
- Supported in most operating systems.
- Integration with other programming languages:
  - Python
  - Java

Boost

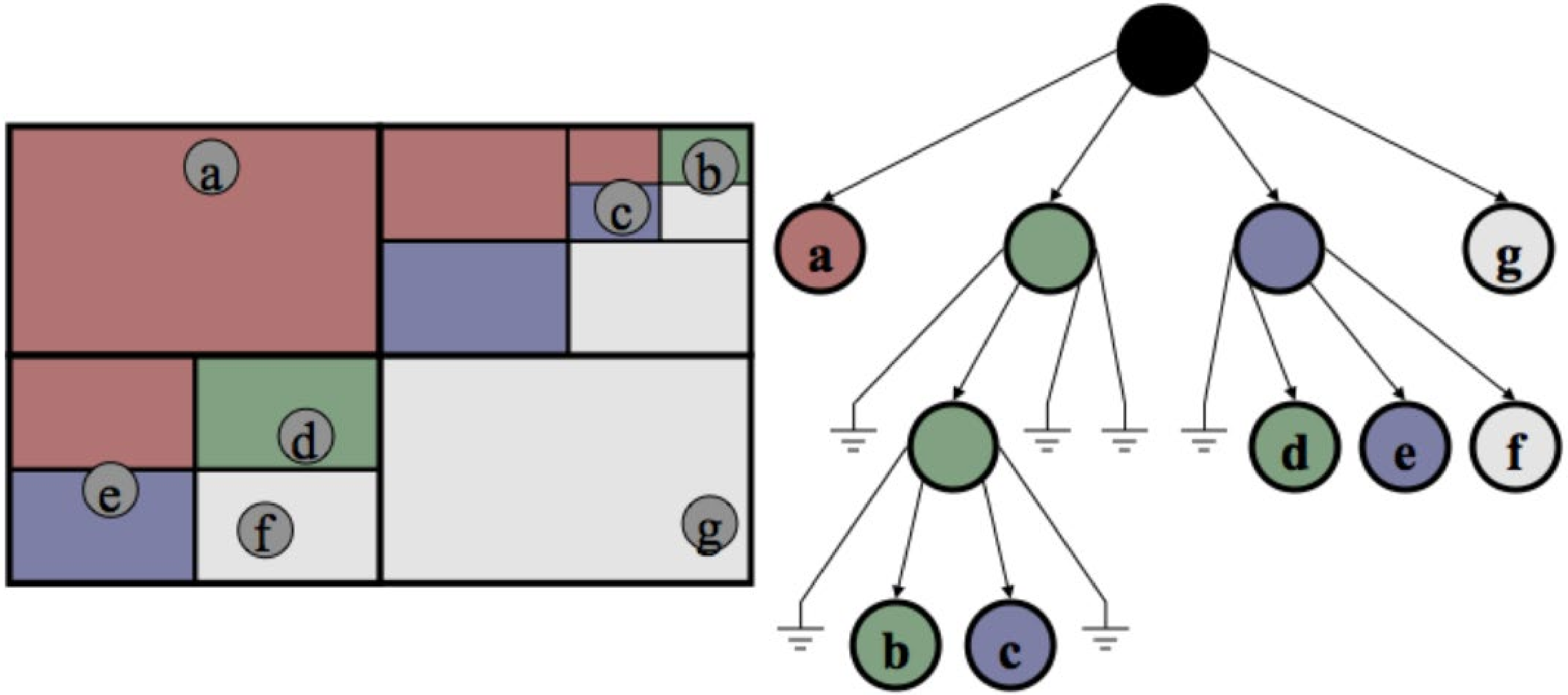
- Example of boost libraries:
  - Algorithms
  - Concurrent programming
  - Containers
  - Data structures
  - Image processing
  - Threads
  - String and text processing
  - Iterators
  - Streams
  - Parsing
  - Memory management
  - …

Boost: spatial indices

- Boost.Geometry.Index collects data structures for spatial indexing of data.
- Goal: accelerate searching for objects in space.
- R-tree is a self-balanced data structure for spatial access methods.
  - Indexes multi-dimensional information (points, rectangles, polygons).
  - Group nearby objects and represent them with their minimum bounding rectangle.



Quadtree



Boost: r-tree example

```
#include <boost/geometry.hpp>
#include <boost/geometry/geometries/point.hpp>
#include <boost/geometry/geometries/box.hpp>

#include <boost/geometry/index/rtree.hpp>

// to store queries results
#include <vector>

// just for output
#include <iostream>
#include <boost/foreach.hpp>

namespace bg = boost::geometry;
namespace bgi = boost::geometry::index;
```

Boost: r-tree example

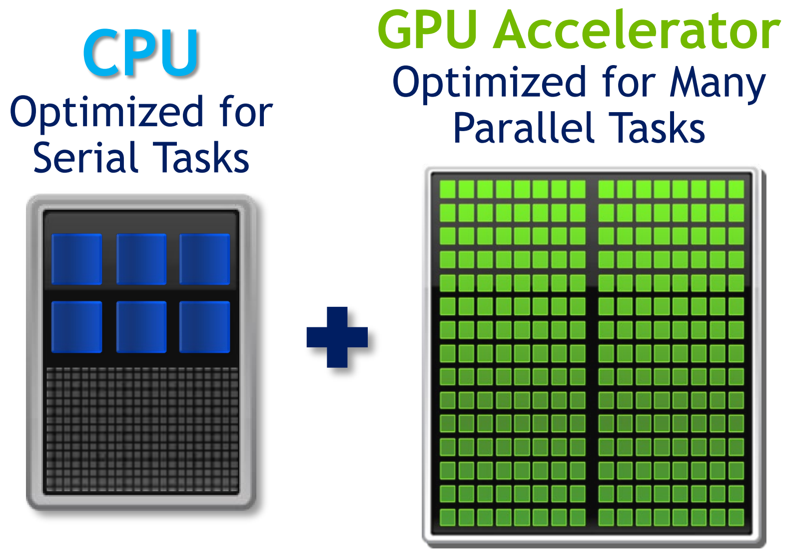
```cpp
int main()
{
    typedef bg::model::point<float, 2, bg::cs::cartesian> point;
    typedef bg::model::box<point> box;
    typedef std::pair<box, unsigned> value;
    // create the rtree using default constructor
    bgi::rtree< value, bgi::quadratic<16> > rtree;
    // create some values
    for ( unsigned i = 0 ; i < 10 ; ++i )
    {
        // create a box
        box b(point(i + 0.0f, i + 0.0f), point(i + 0.5f, i + 0.5f));
        // insert new value
        rtree.insert(std::make_pair(b, i));
    }
    // find values intersecting some area defined by a box
    box query_box(point(0, 0), point(5, 5));
    std::vector<value> result_s;
    rtree.query(bgi::intersects(query_box), std::back_inserter(result_s));
    // find 5 nearest values to a point
    std::vector<value> result_n;
    rtree.query(bgi::nearest(point(0, 0), 5), std::back_inserter(result_n));
    return 0;
}
```

CUDA

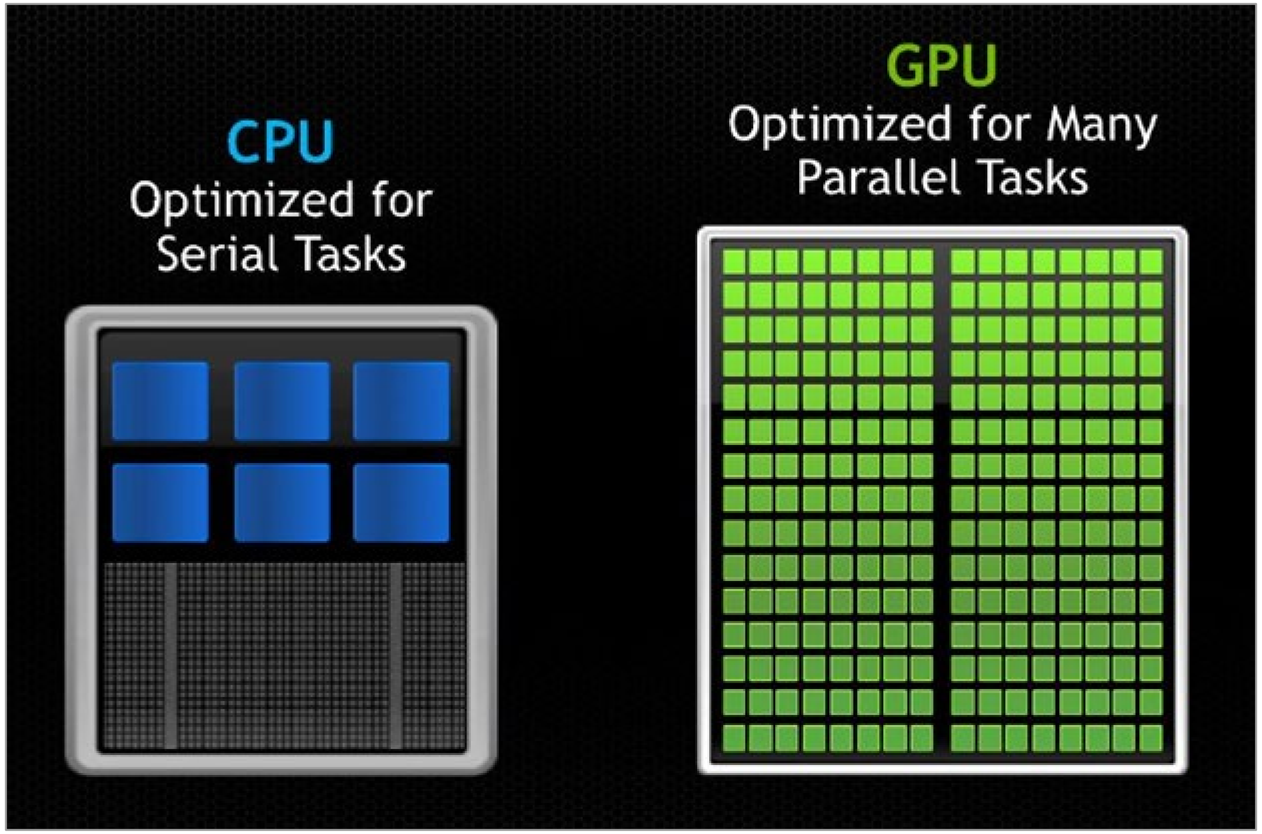
- Parallel computing platform and API that uses the GPU for general purpose computing.
- Software layer that gives direct access to the GPU's parallel computational elements.
- Design to work with other programming languages, such as C, C++, Fortran.

CUDA

- GPUs are designed to perform high-speed parallel calculations for real-time rendering (embarrassingly parallel task).
- 10-100x speed-ups over CPUs when applied in GPGPU.
- Why?
  - CPU contains few powerful cores, GPU contains hundreds of smaller cores.
  - CPU: individual threads execute instructions independently (SISD). GPU: single instruction, multiple threads (SIMT).
  - Shared memory for algorithms with a high degree of locality.



Modern GPUs



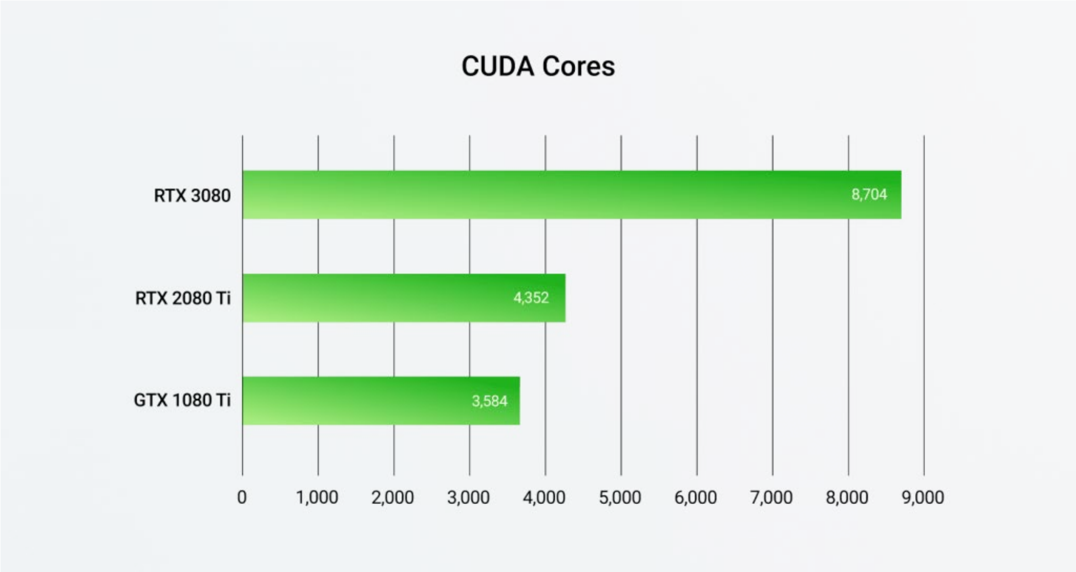
CPU
Optimized for Serial Tasks
GPU
Optimized for Many Parallel Tasks

SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

NVIDIA Titan RTX

GPU architecture

- Global memory:
  - Similar to CPU's RAM.
  - Accessible by both CPU and GPU.
  - Limited: < 24 GB
- Streaming multiprocessors (SMs)
  - Perform the actual computations.
  - Each SM has its own control units, registers, caches, **<u>execution pipeline</u>**.
  - 3080 RTX: 68 SMs, each with 128 CUDA cores.



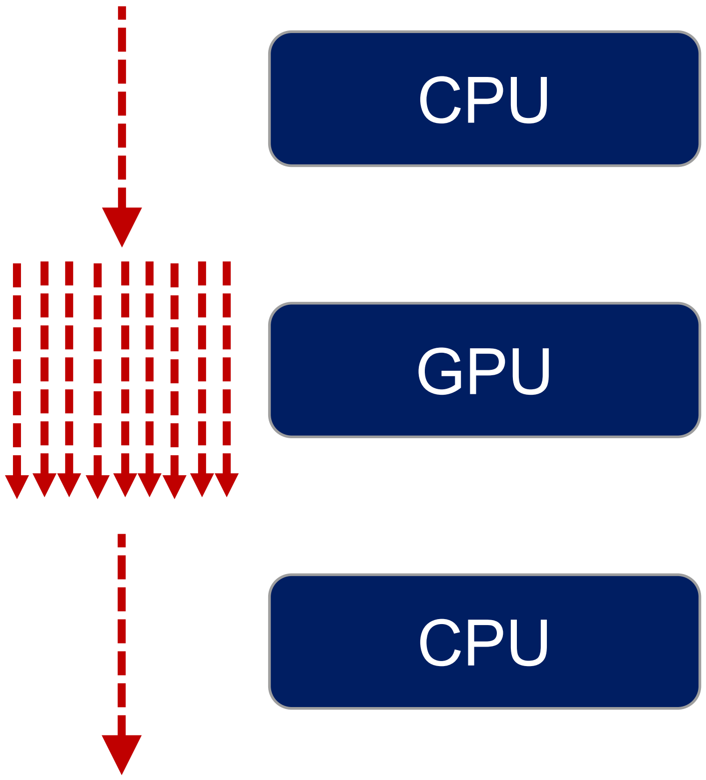
Heterogeneous computing

- Host: CPU and its memory.
- Device: GPU and its memory.

```
texture<float, 2, cudaReadModeElementType> tex;

void foo()
{
  cudaArray* cu_array;

  // Allocate array
  cudaChannelFormatDesc description = cudaCreateChannelDesc<float>();
  cudaMallocArray(&cu_array, &description, width, height);

  // Copy image data to array
  cudaMemcpyToArray(cu_array, image, width*height*sizeof(float), cudaMemcpyHostToDevice);

  // Set texture parameters (default)
  tex.addressMode[0] = cudaAddressModeClamp;
  tex.addressMode[1] = cudaAddressModeClamp;
  tex.filterMode = cudaFilterModePoint;
  tex.normalized = false; // do not normalize coordinates

  // Bind the array to the texture
  cudaBindTextureToArray(tex, cu_array);

  // Run kernel
  dim3 blockDim(16, 16, 1);
  dim3 gridDim((width + blockDim.x - 1)/ blockDim.x, (height + blockDim.y - 1) / blockDim.y, 1);
  kernel<<< gridDim, blockDim, 0 >>>(d_data, height, width);

  // Unbind the array from the texture
  cudaUnbindTexture(tex);
} //end foo()

__global__ void kernel(float* odata, int height, int width)
{
   unsigned int x = blockIdx.x*blockDim.x + threadIdx.x;
   unsigned int y = blockIdx.y*blockDim.y + threadIdx.y;
   if (x < width && y < height) {
      float c = tex2D(tex, x, y);
      odata[y*width+x] = c;
   }
}
```

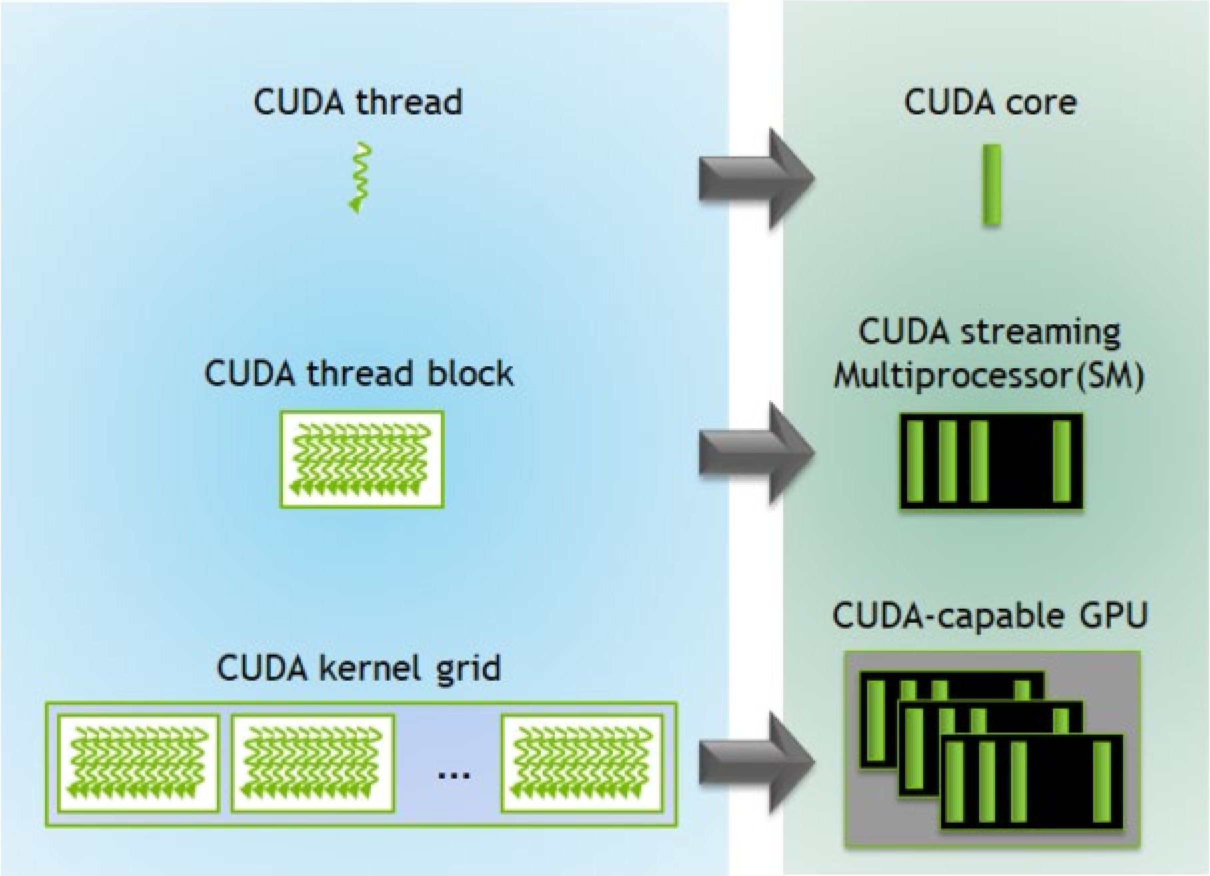
CPU

GPU



Processing flow

1. Copy input data from CPU to GPU memory.
2. Load GPU program and execute.
   - Group of threads is called a CUDA block, executed by one streaming multiprocessor (SM).
   - Set of blocks is referred to as a grid.
3. Copy results from GPU memory to CPU memory.



CUDA: vector addition

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main( int argc, char* argv[] )
{
    // Size of vectors
    int n = 100000;
    // Host input vectors
    double *h_a, *h_b;
    //Host output vector
    double *h_c;
    // Device input vectors
    double *d_a, *d_b;
    //Device output vector
    double *d_c;
    // Size, in bytes, of each vector
    size_t bytes = n*sizeof(double);
    // Allocate memory for each vector on host
    h_a = (double*)malloc(bytes); h_b = (double*)malloc(bytes); h_c = (double*)malloc(bytes);

    // Allocate memory for each vector on GPU
    cudaMalloc(&d_a, bytes); cudaMalloc(&d_b, bytes); cudaMalloc(&d_c, bytes);
```

Allocating memory on the host and device

CUDA: vector addition

```
// Initialize vectors on host
for( int i = 0; i < n; i++ ) {
    h_a[i] = sin(i)*sin(i);
    h_b[i] = cos(i)*cos(i);
}

// Copy host vectors to device
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);

int blockSize, gridSize;

// Number of threads in each thread block
blockSize = 1024;

// Number of thread blocks in grid
gridSize = (int)ceil((float)n/blockSize);

// Execute the kernel
vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
```

Copying to device

Executing the kernel, with 1024 threads per thread block

CUDA: vector addition

```
// CUDA kernel. Each thread takes care of one element of c
__global__ void vecAdd(double *a, double *b, double *c, int n)
{
    // Get our global thread ID
    int id = blockIdx.x*blockDim.x+threadIdx.x;

    // Make sure we do not go out of bounds
    if (id < n)
        c[id] = a[id] + b[id];
}
```

CUDA kernel, runs on device

From: https://www.olcf.ornl.gov/tutorials/cuda-vector-addition/

CUDA: vector addition

Copying from device to host

```
    // Copy array back to host
    cudaMemcpy( h_c, d_c, bytes, cudaMemcpyDeviceToHost );

    // Sum up vector c and print result divided by n, this should equal 1 within error
    double sum = 0;
    for(i=0; i<n; i++)
        sum += h_c[i];
    printf("final result: %f\n", sum/n);

    // Release device memory
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    // Release host memory
    free(h_a);
    free(h_b);
    free(h_c);

    return 0;
}
```

CUDA libraries

- Math:
  - cuBLAS: basic linear algebra.
  - cuFFT: fast Fourier transforms.
  - cuTENSOR: tensor linear algebra.
  - cuSPARSE: BLAS for sparse matrices.
- Vision, image and video libraries
  - OpenCV: computer vision, machine learning.
  - Gunrock: graph analytics and processing.
- Deep learning:
  - cuDNN: primitives for deep neural networks.
  - Riva: conversation apps.
- Parallel algorithm:
  - Thrust: parallel algorithms and data structures.

Thrust

- C++ template library for CUDA.
- Containers
  - `thrust::host_vector<T>`
  - `thrust::device_vector<T>`
- Algorithms
  - `thrust::sort()`
  - `thrust::reduce()`
  - `thrust::inclusive_scan()`
  - …

Thrust

- Containers to hide `cudaMalloc`, `cudaMemcpy`, `cudaFree`.

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;

// vector memory automatically released w/ free() or cudaFree()
```