HES SO // Master

Information and Communications Technologies

2012 06 28

GPGPU

A look into OpenCL

Author : Frédéric Dubouchet

Professor : Paul Albuquerque

Plan

- What is OpenCL?
- Practical simple example
- Advance examples in OpenCL
- Conclusion
- Questions

What is OpenCL?

Open Computing Language

Language and API

OpenCL Objects

OpenCL Device Model

OpenCL C++ Interface

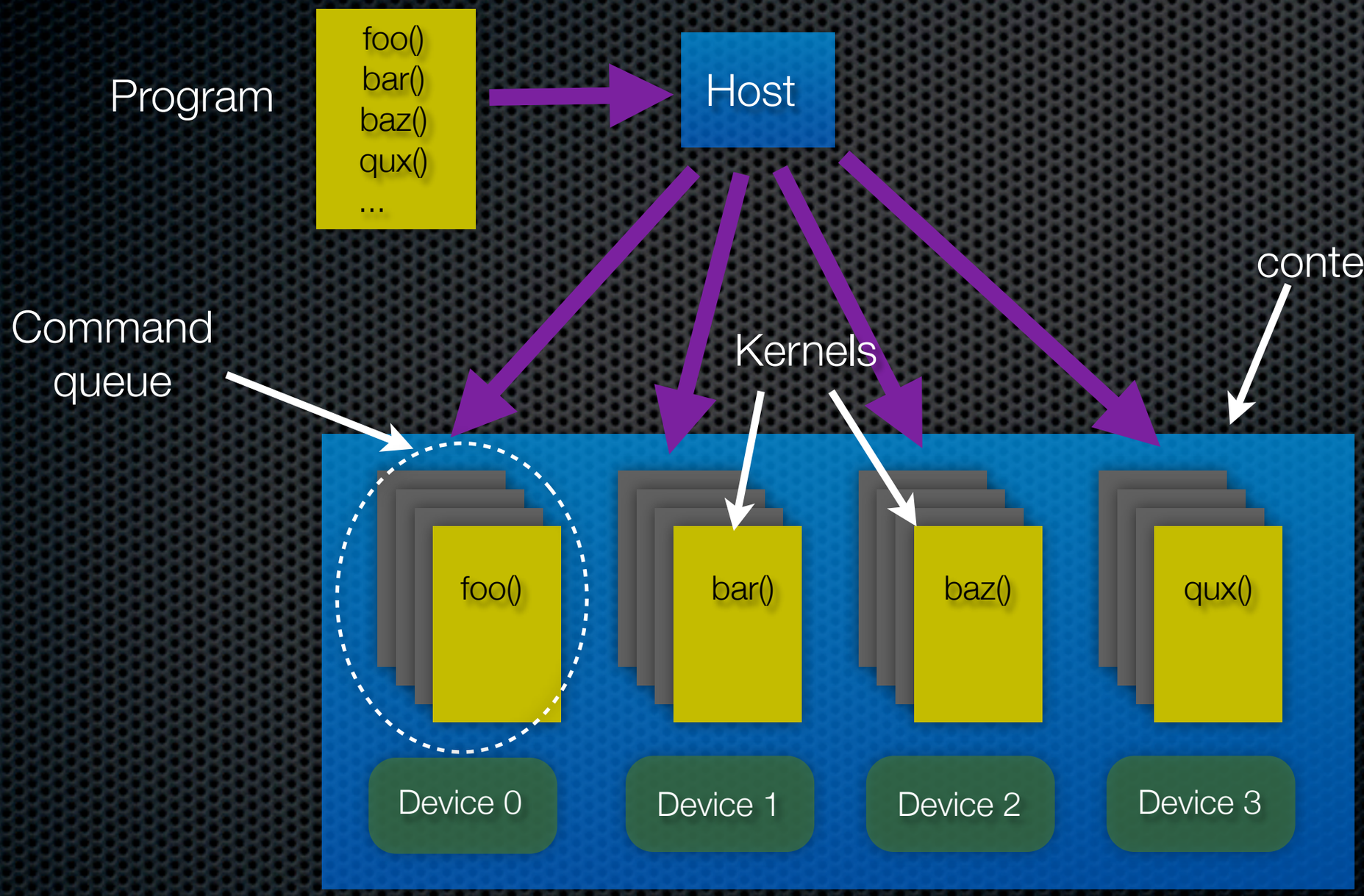
Open Computing Language

- General Purpose Computing Language!
  - Parallel computing on many platform
    - computer mobile web
  - GPGPU but not only!
    - CPU / GPU / Specialized Hardware
  - Maintain by the Khronos group (OpenGL)
    - Apple, Nvidia, AMD, Intel and many others...
  - Open! not another proprietary interface

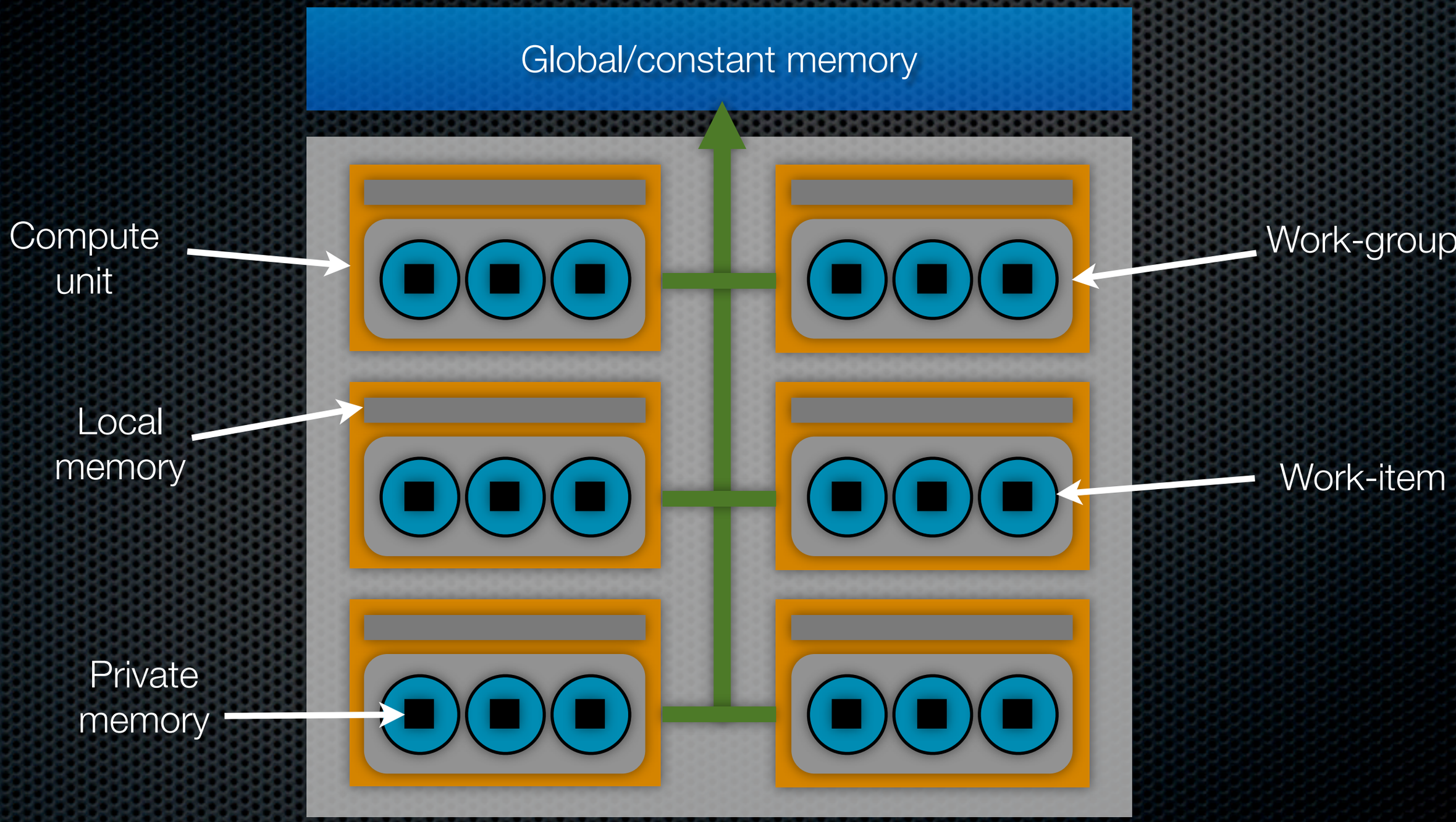
Language and API

- An API (run on the Host)
  - in C but with interface to many other languages
- A language (run on the Device)
  - vector oriented
  - C99 inspired

OpenCL Objects

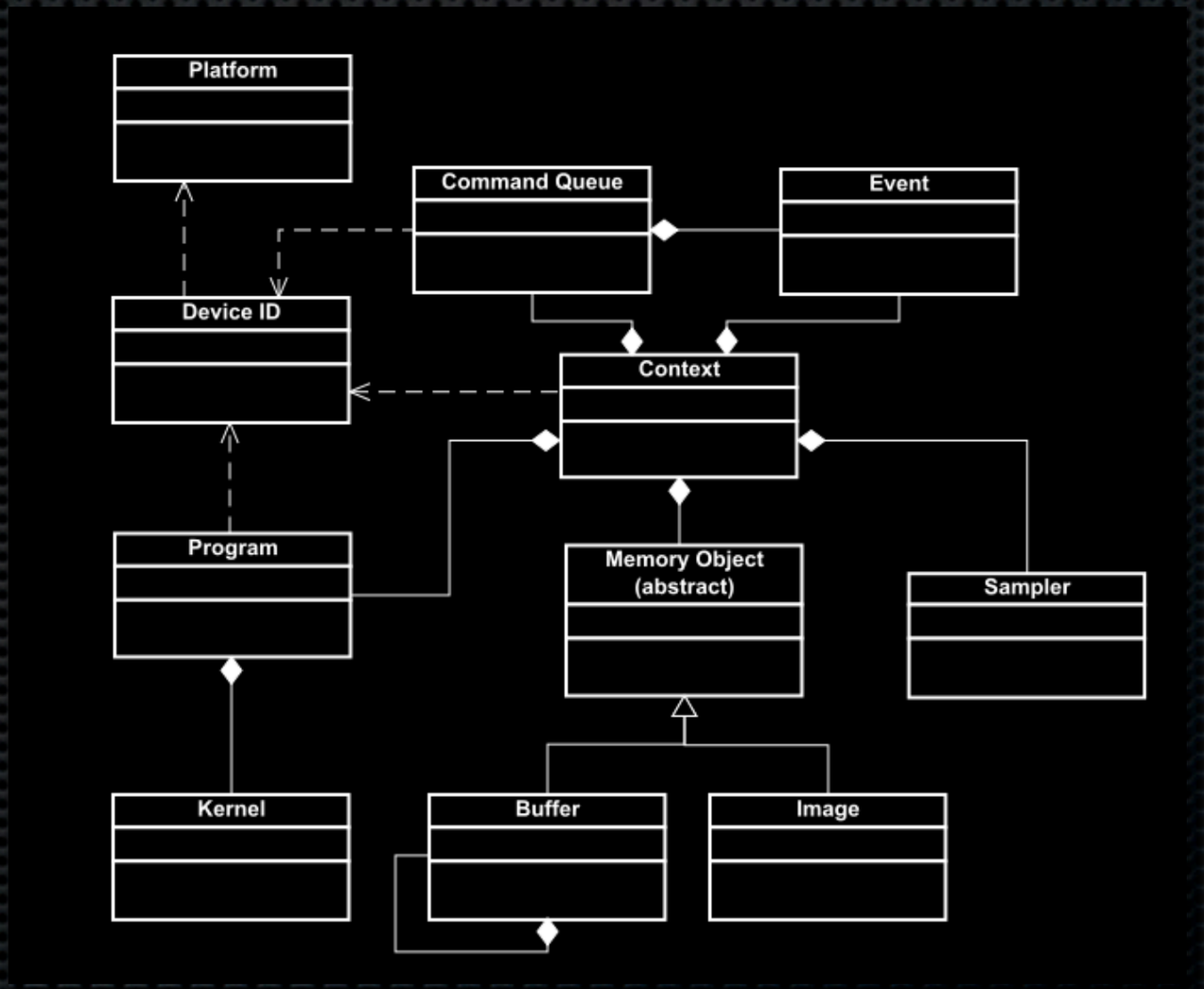


OpenCL Device Model



OpenCL C++ interface

- Officially published by Khronos
- Template based
- Header only
- 100% portable
- Object Oriented



Practical Simple Example

Host code

Device code

Host code (1/2)

- Headers include
- Get Platform
- Get Device
- Create Context
- Create Command Queue
- Load & Build
- Select Kernel

```
// simple OpenCL example

#include <iostream>
#include <fstream>
#include <vector>
#define __CL_ENABLE_EXCEPTIONS
#include <CL/cl.hpp>

const unsigned int vector_size = 1024;

int main(int ac, char** av) {
   try {
      // get the device (here the GPU)
      std::vector<cl::Platform> platforms;
      std::vector<cl::Device> devices_;
      cl::Platform::get(&platforms);
      platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices_);
      cl_context_properties properties[] = {
         CL_CONTEXT_PLATFORM,
         (cl_context_properties)(platforms[0])(),
         0
      };
      cl::Context context_ = cl::Context(CL_DEVICE_TYPE_GPU, properties);
      devices_ = context_.getInfo<CL_CONTEXT_DEVICES>();
      cl::CommandQueue queue_(context_, devices_[0]);
      // load the kernel code
      std::ifstream ifs("./simple.cl");
      std::string str(
            (std::istreambuf_iterator<char>(ifs)),
            std::istreambuf_iterator<char>());
      ifs.close();
      // compile
      cl::Program::Sources source(1, std::make_pair(str.c_str(), str.size()));
      cl::Program program_(context_, source);
      program_.build(devices_);
      // create the kernel
      cl::Kernel kernel_(program_, "simple");
```

Host code (2/2)

- Create and fill up Host buffers
- Create Device Buffers (2 in and 1 out)
- Set Kernel Arguments
- Execute the Kernel
- Read the buffers
- Exception Handling

```
    // prepare the buffers
    std::vector<float> in1(vector_size);
    std::vector<float> in2(vector_size);
    for (std::vector<float>::iterator ite = in1.begin(); ite != in1.end(); ++ite)
       (*ite) = (float)random();
    for (std::vector<float>::iterator ite = in2.begin(); ite != in2.end(); ++ite)
       (*ite) = (float)random();
    cl::Buffer buf_in1_ = cl::Buffer(
          context_,
          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
          sizeof(float) * vector_size,
          (void*)&in1);
    cl::Buffer buf_in2_ = cl::Buffer(
          context_,
          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
          sizeof(float) * vector_size,
          (void*)&in2);
    cl::Buffer buf_out_ = cl::Buffer(
          context_,
          CL_MEM_WRITE_ONLY,
          sizeof(float) * vector_size);
       // set the arguments
       kernel_.setArg(0, buf_in1_);
       kernel_.setArg(1, buf_in2_);
       kernel_.setArg(2, buf_out_);
       // wait to the command queue to finish before proceeding
       queue_.finish();
       // run the kernel
       std::vector<float> out(vector_size);
       queue_.enqueueNDRangeKernel(
             kernel_,
             cl::NullRange,
             cl::NDRange(vector_size),
             cl::NullRange);
       queue_.finish();
       // get the result out
       queue_.enqueueReadBuffer(
             buf_out_,
             CL_TRUE,
             0,
             vector_size * sizeof(float),
             &out[0]);
       queue_.finish();
       std::cout << "Operation successfull" << std::endl;
    } catch (cl::Error& er) {
       std::cerr << "Exception(CL) : " << er.what() << std::endl;
    } catch (std::exception& ex) {
       std::cerr << "Exception(STL) : " << ex.what() << std::endl;
    }
    return 0;
}
```

Device Code

- Kernel simple
- global in buffers (in1 & in2)
- global out buffer
- get the work item ID
- compute

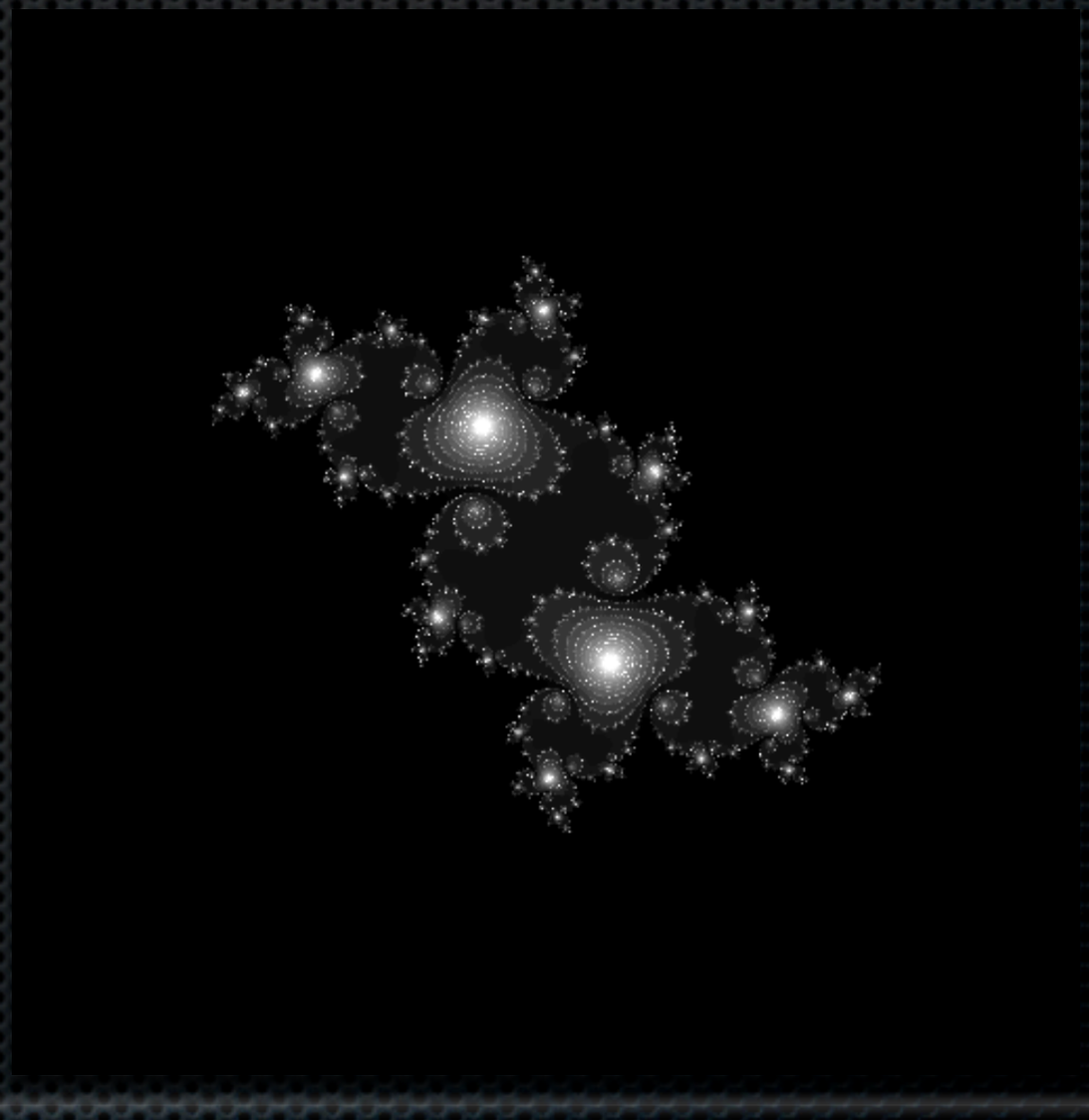
```
// very simple kernel

__kernel void simple(
        __global read_only float* in1,
        __global read_only float* in2,
        __global write_only float* out)
{
    const uint pos = get_global_id(0);
    out[pos] = in1[pos] * in2[pos];
}
```

Advance examples in OpenCL

- Julia Set
- Cellular automata
- Floyd-Warshall
- Fast Fourier Transform

Julia Set

Kernel view
Performances

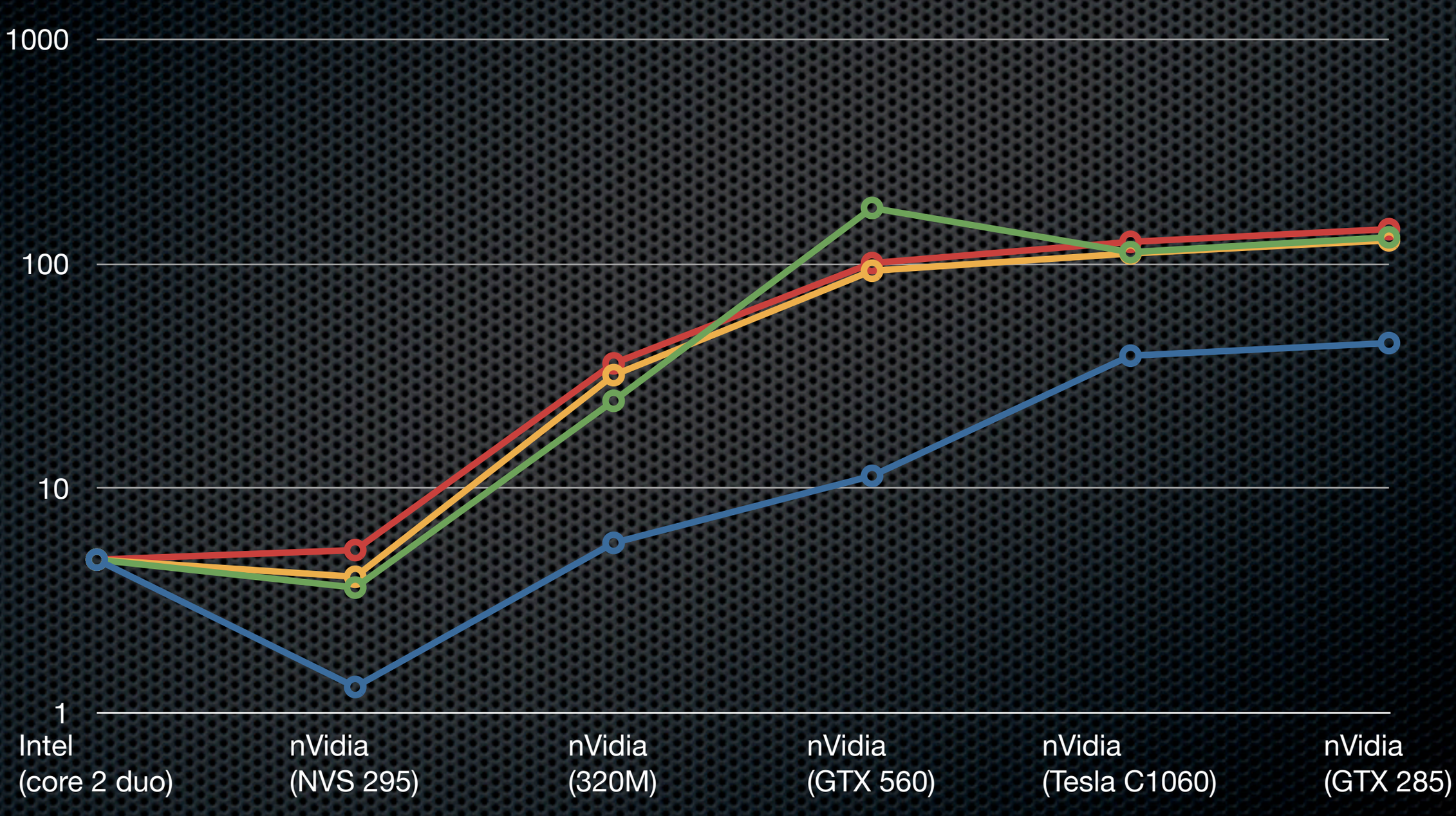


Kernel View

- Utility complex operators
- Get WorkItem position (d)
- Get grid size (m)
- Compute the position in C
- Compute Julia
- Set the output buffer

```
float2 square_c(float2 c) {
   return (float2)(c.x * c.x - c.y * c.y, 2.0f * c.y * c.x);
}

float2 add_c(float2 c1, float2 c2) {
   return (float2)(c1.x + c2.x, c1.y + c2.y);
}

float square_norm_c(float2 c) {
   return c.x * c.x + c.y * c.y;
}

__kernel void julia_buffer(
   __global float* out,
   float2 c,
   uint max_iteration)
{
   const uint2 d = (uint2)(get_global_id(0), get_global_id(1));
   const uint2 m = (uint2)(get_global_size(0), get_global_size(1));
   float2 pos = (float2)(
      (((float)d.x / (float)m.x) * 4.0f) - 2.0f,
      (((float)d.y / (float)m.y) * 4.0f) - 2.0f);
   uint ite = 0;
   while ((square_norm_c(pos) < 4.0f) && (ite < max_iteration)) {
      pos = add_c(square_c(pos), c);
      ite++;
   }
   float val = (float)ite / (float)max_iteration;
   out[d.y * m.x + d.x] = val;
}
```

Performances

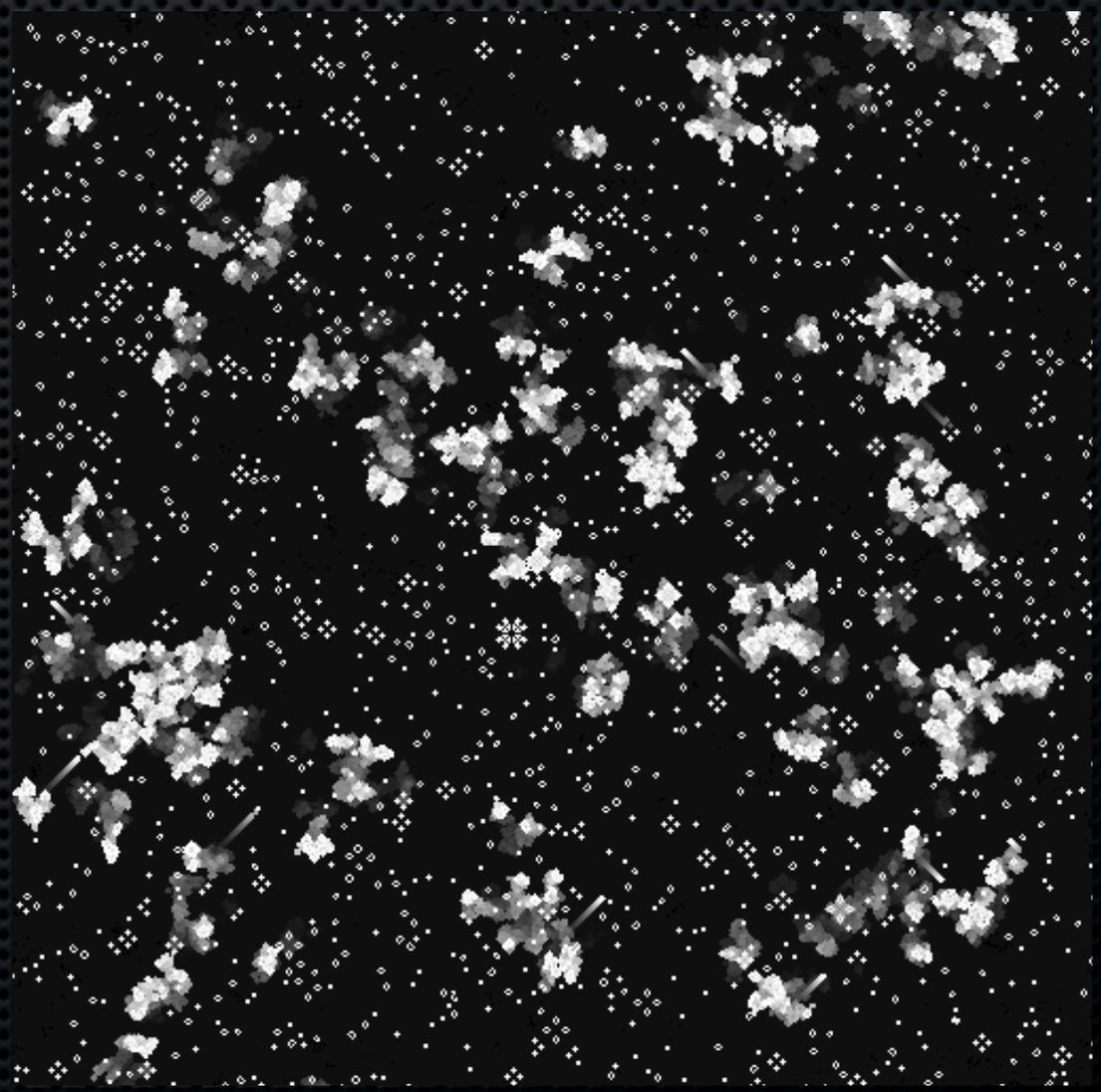


Cellular Automata

Kernel view

Performances

Against CUDA

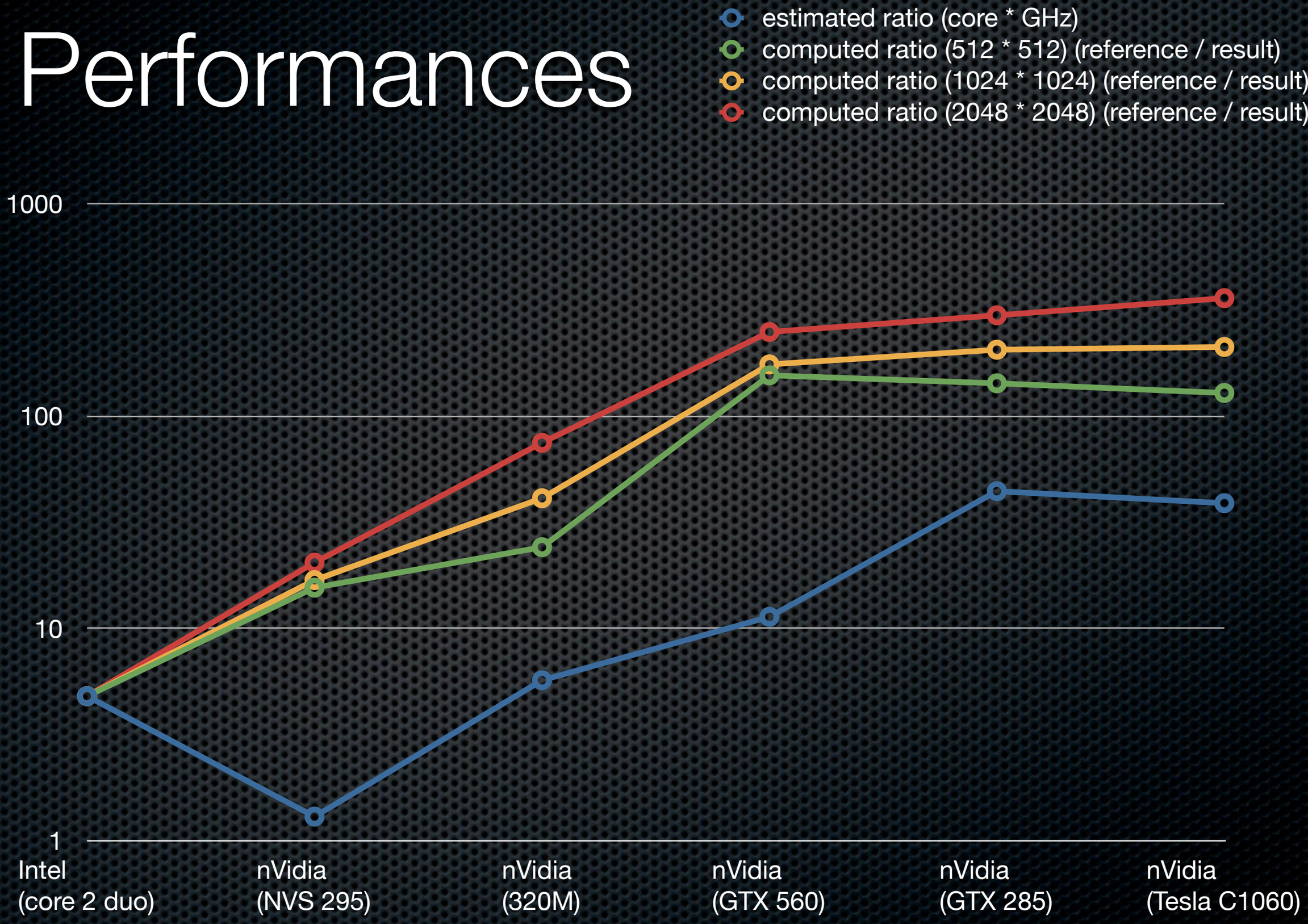


Kernel View

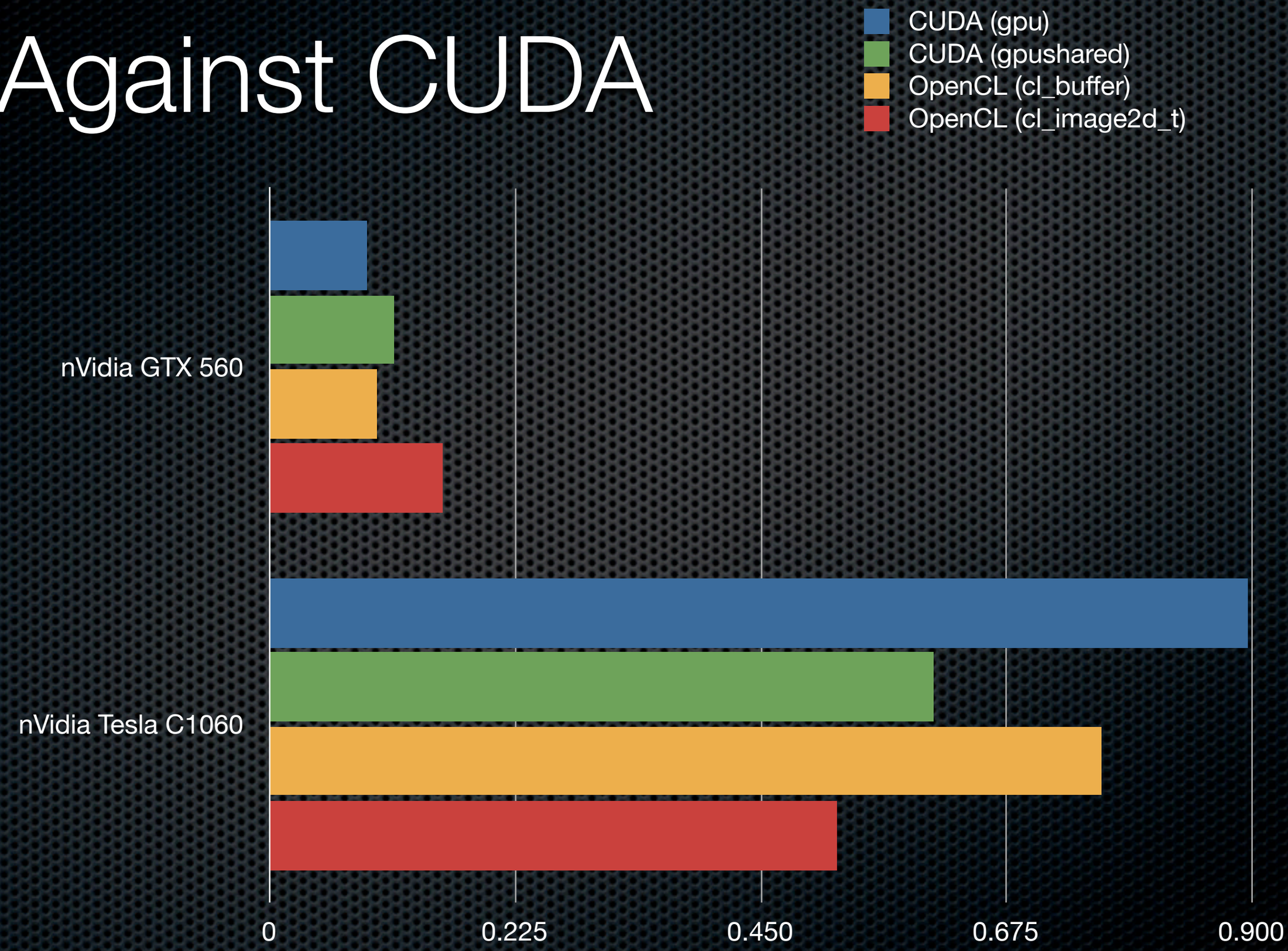
- Utility complex operators
- Get WorkItem position (d)
- Get grid size (m)
- accumulate the value of neighbor
- Compute the result
  - using ternary operator

```
int pos2linear(
   const int2 position,
   const int2 m)
{
   return position.y * m.x + position.x;
}

int add(__global const float* in, const int2 position, const int2 m) {
   // empty border
   return
      ((position.x < 0 || position.x >= m.x) || (position.y < 0 || position.y >= m.y))
         ? 0.0f
         : (in[pos2linear(position, m)]) ? 1.0f : 0.0f;
}

__kernel void life_buffer(
   __global float* in,
   __global float* out,
   unsigned int max_iterations)
{
   const int2 d = (int2)(get_global_id(0), get_global_id(1));
   const int2 m = (int2)(get_global_size(0), get_global_size(1));

   int position = pos2linear(d, m);
   int sum = 0;
   sum += add(in, (int2)(d.x - 1, d.y + 1), m);
   sum += add(in, (int2)(d.x    , d.y + 1), m);
   sum += add(in, (int2)(d.x + 1, d.y + 1), m);
   sum += add(in, (int2)(d.x - 1, d.y    ), m);
   sum += add(in, (int2)(d.x + 1, d.y    ), m);
   sum += add(in, (int2)(d.x - 1, d.y - 1), m);
   sum += add(in, (int2)(d.x    , d.y - 1), m);
   sum += add(in, (int2)(d.x + 1, d.y - 1), m);

   out[position] = ((sum == 3) || (in[position] && (sum == 2))) ? 1.0f : 0.0f;
}
```

Performances



Against CUDA



Floyd-Warshall

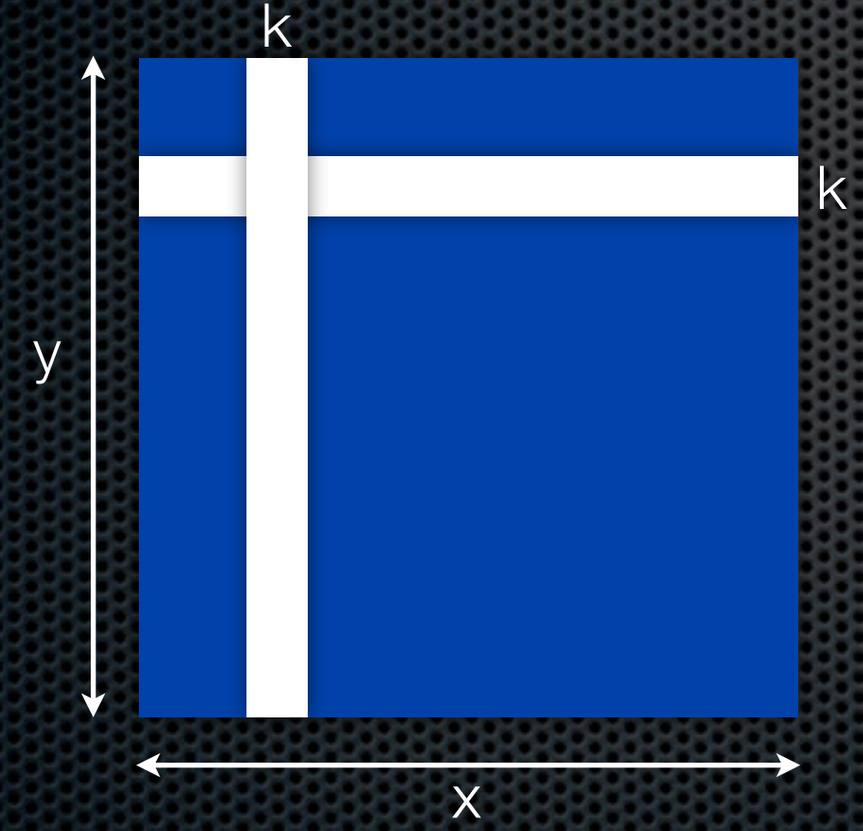
Method

Performances

Method

For each values of k (in 0..N -1)

The Matrix

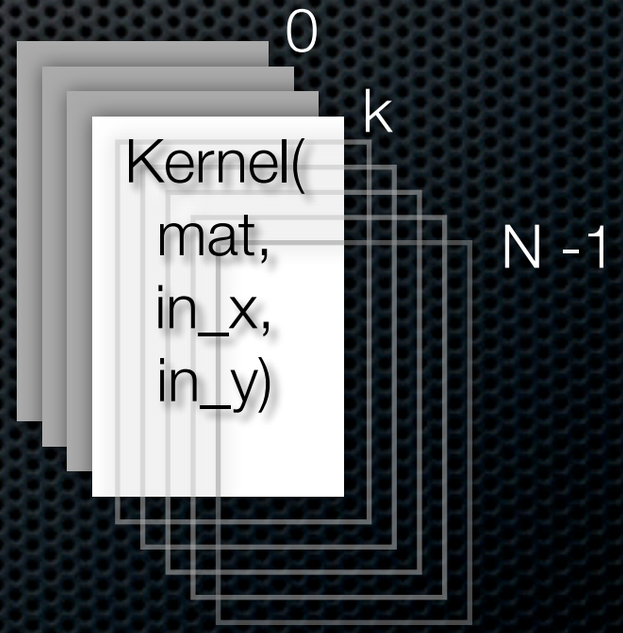


mat (whole)

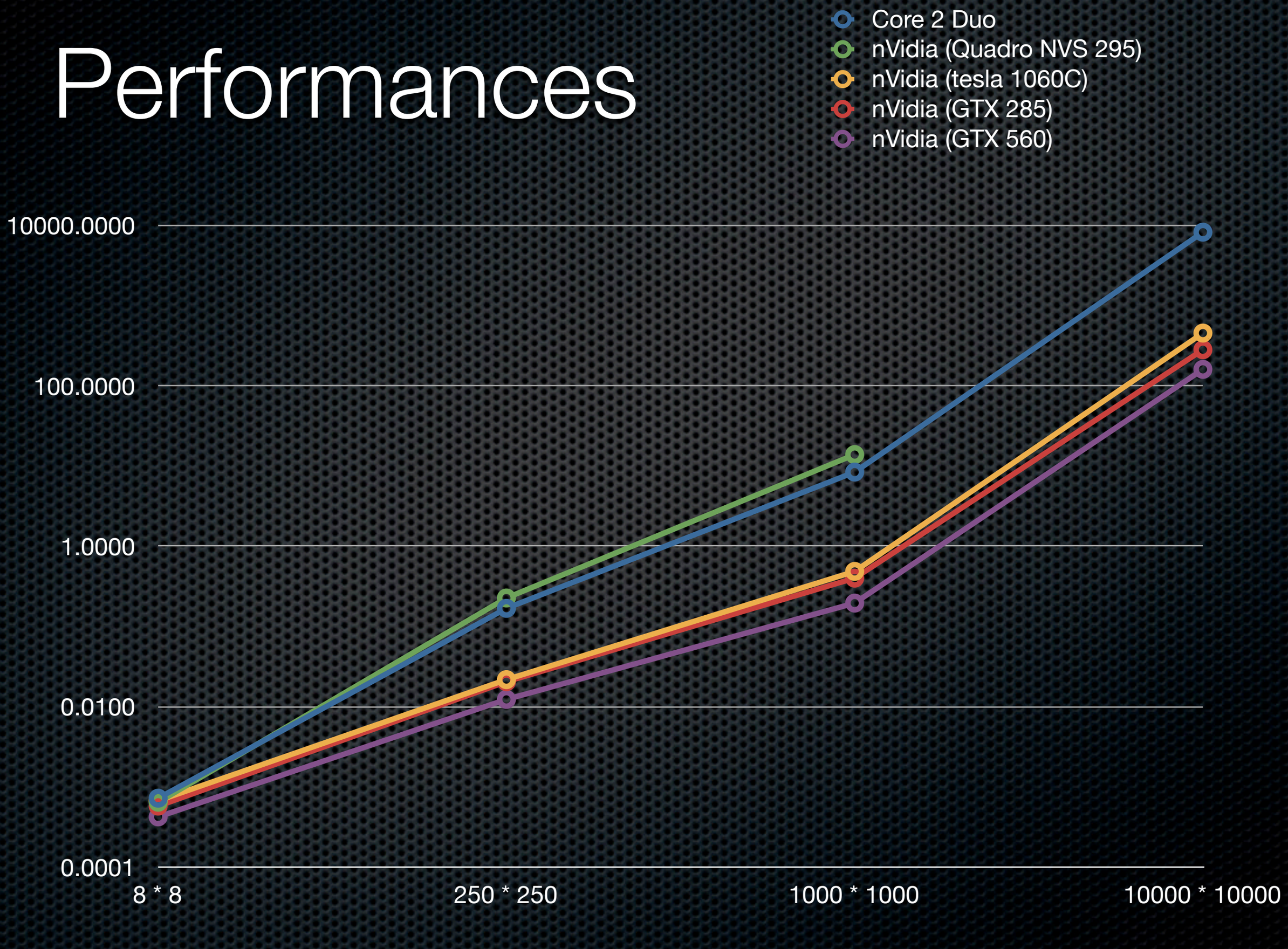
in_x = y[k]

in_y = x[k]

The Kernel Stack



Performances



Fast Fourier Transform

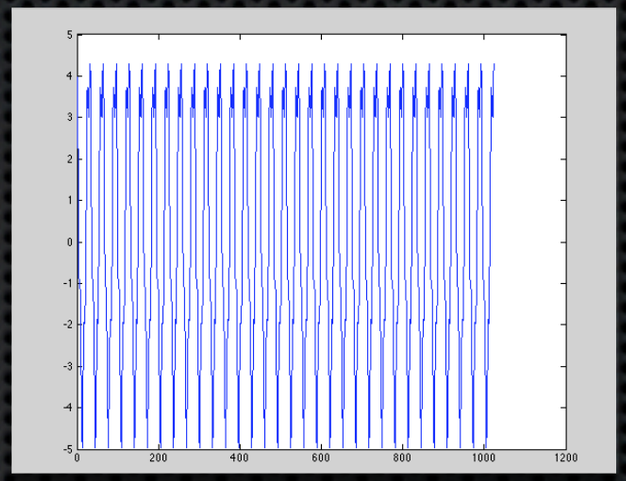
Fast Introduction

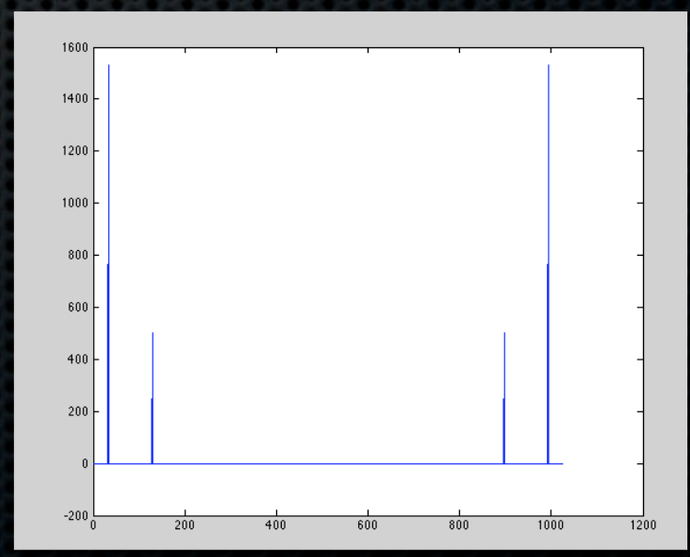
Kernel & FFTW

Performances

Fast Introduction

- From Temporal space to Spectral
- Discrete transformation
- Based on power of prime number
- Divide & Conquer algorithm
- N log(N) complexity



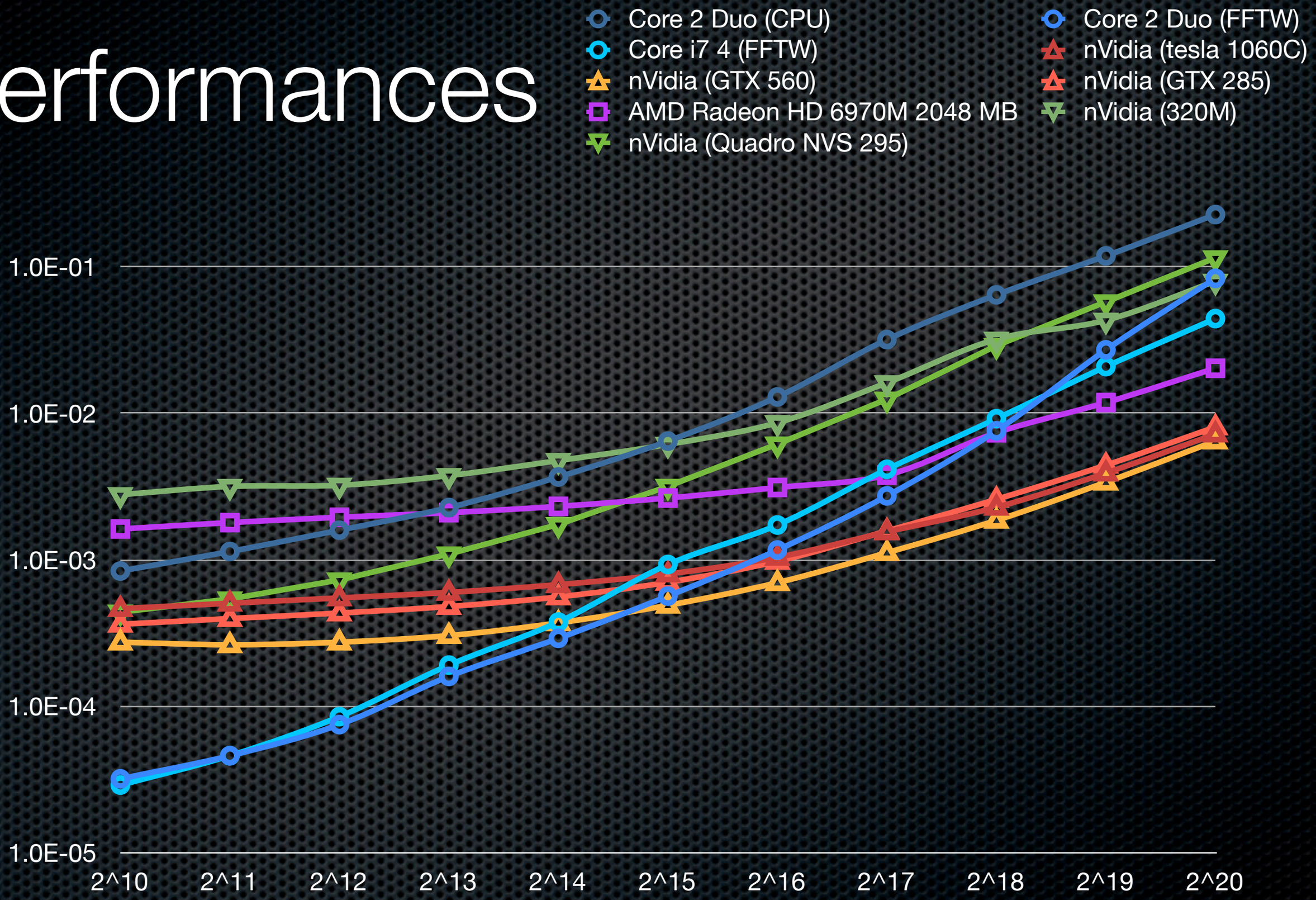


Kernel & FFTW

- From the creator of the official OpenCL FFT library
  - Release by Apple under MIT type license
- Has similar performances to the CUDA FFT library

- FFTW is a library written for CPU
  - Used by Matlab in fft
  - Suppose to be the best library for FFT on CPU
  - GPL license

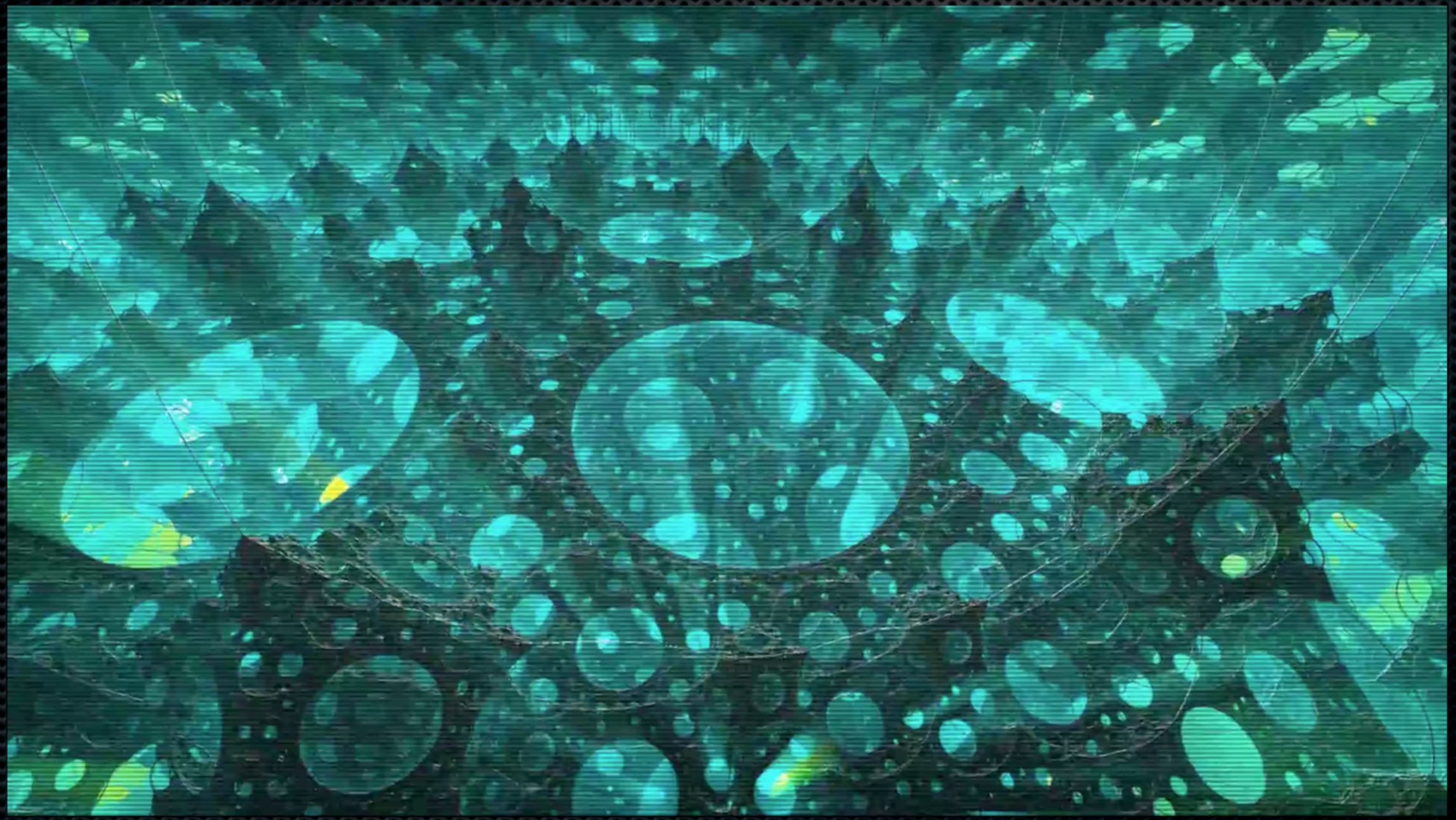
Performances



Conclusion

- OpenCL is a mature technology
- Performances are at least at the level of CUDA
- Ability to use it on CPU is a plus
- GPGPU offers excellent performances at a reduce cost
  - Cheaper than Hardware
  - Shorter development time

Questions?



ray marching using OpenCL+OpenGL - Hartverdrahtet - Akronyme Analogiker | 4k Revision 2012