Distributed GPGPU Computing

Martin Stumpf

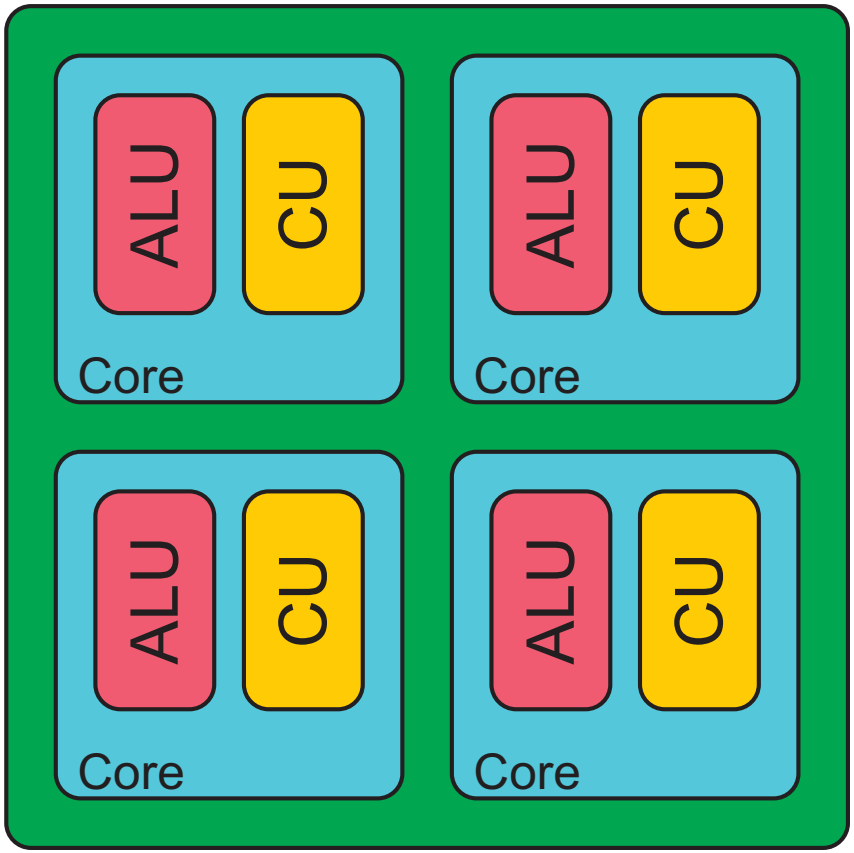
Ste||ar Group, LSU

September 23, 2014

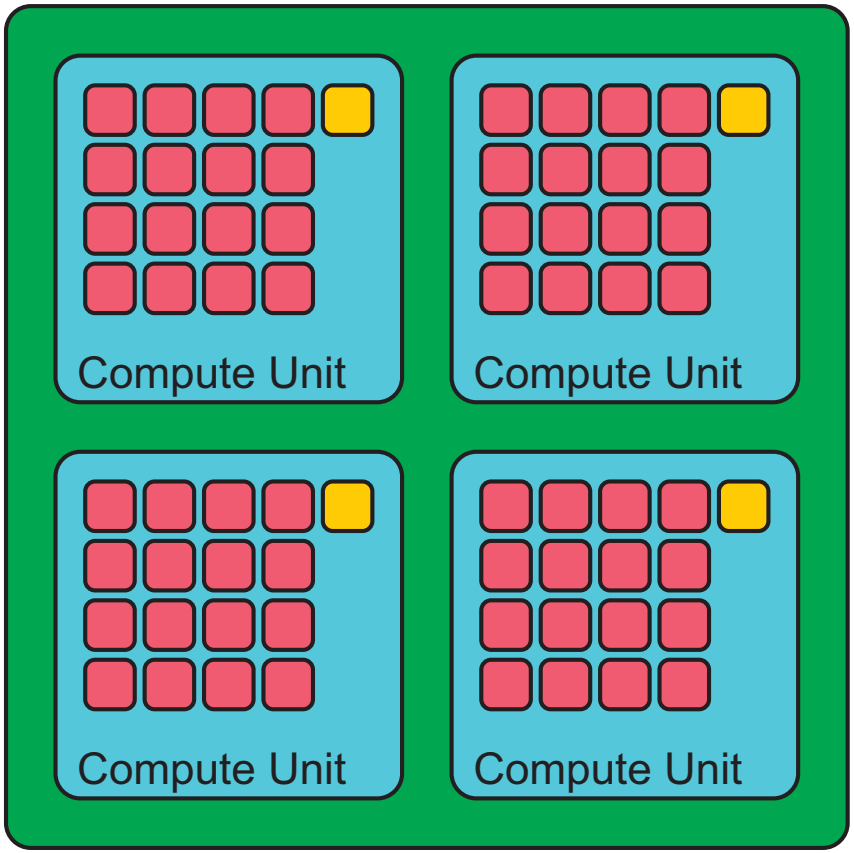
Table of Contents

Outline

CPU vs GPU



CPU



GPU

Why GPGPU?

The theoretical calculation power of a GPU is much higher than a CPU.

Example

CPU (Intel Xeon E5-2670 v3):

- 12 Cores, 2.3 GHz, 32 FLOPS/cycle
  - 884 GFLOPS
- Prize: $\sim$ \$ 1500

GPU (NVidia Tesla K40):

- 2880 Cores, 745 MHz, 2 FLOPS/cycle
  - 4291 GFLOPS
- Prize: $\sim$ \$ 4000

So, what computational tasks are actually suitable for GPGPU?

Problems suitable for GPGPU

Every problem that fits the SPMD programming scheme, can benefit greatly from GPGPU.

Examples:

- Fluid Simulations
- Mathematical Vector Operations
- Image Processing
- Stencil Based Simulations

SPMD based programming languages:

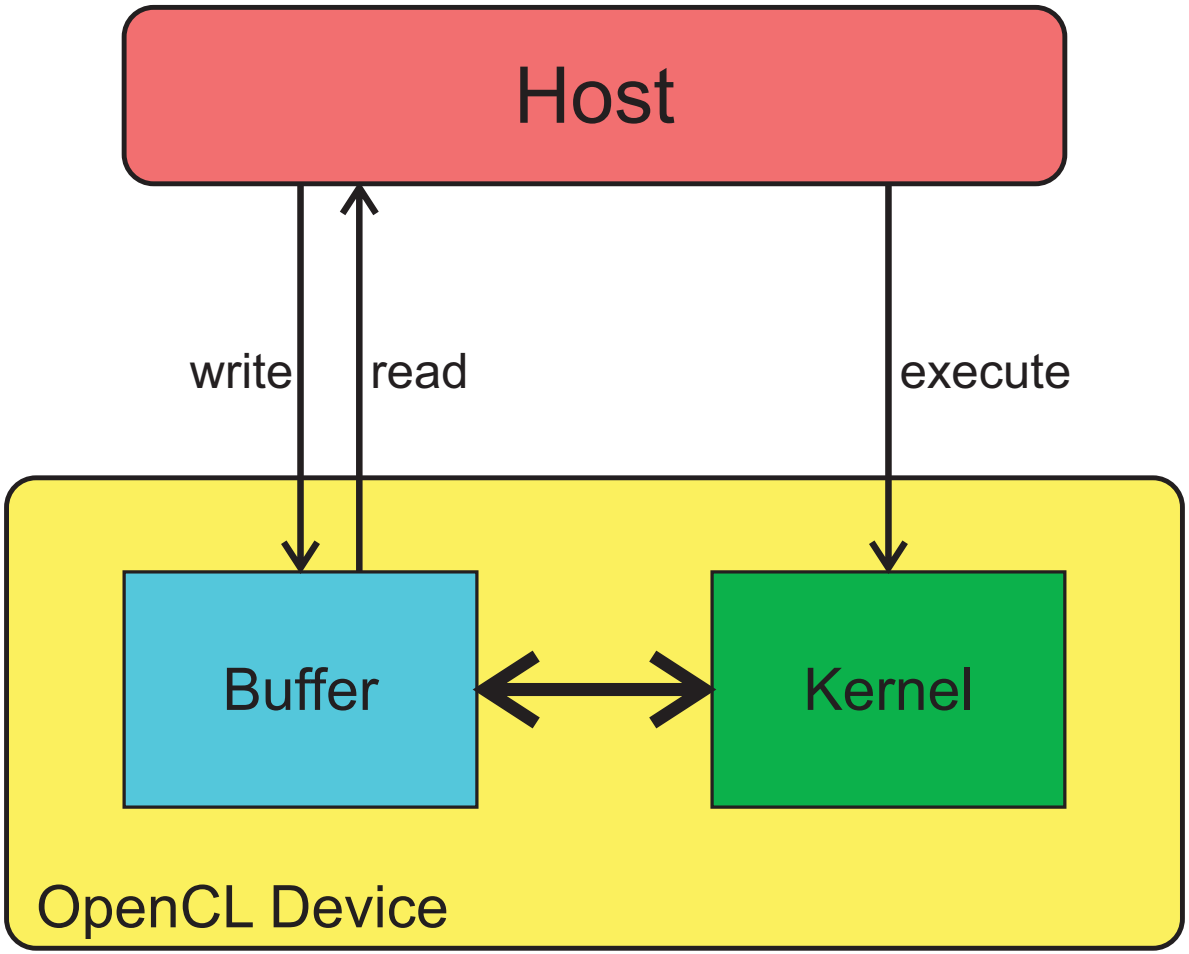
- CUDA (NVidia)
- OpenCL (Vendor independent)
- C++ AMP (Microsoft)

OpenCL

- An OpenCL device is split in two components:
  - The Buffer: Represents memory on the device
  - The Kernel: A C-style function that modifies one or multiple elements of a buffer
- Kernel source code stays plain text and gets compiled at runtime

  $\Longrightarrow$ OpenCL programs are device independent
- Kernel executions on the device run asynchronous to the host program

OpenCL



Outline

Distributed OpenCL with MPI

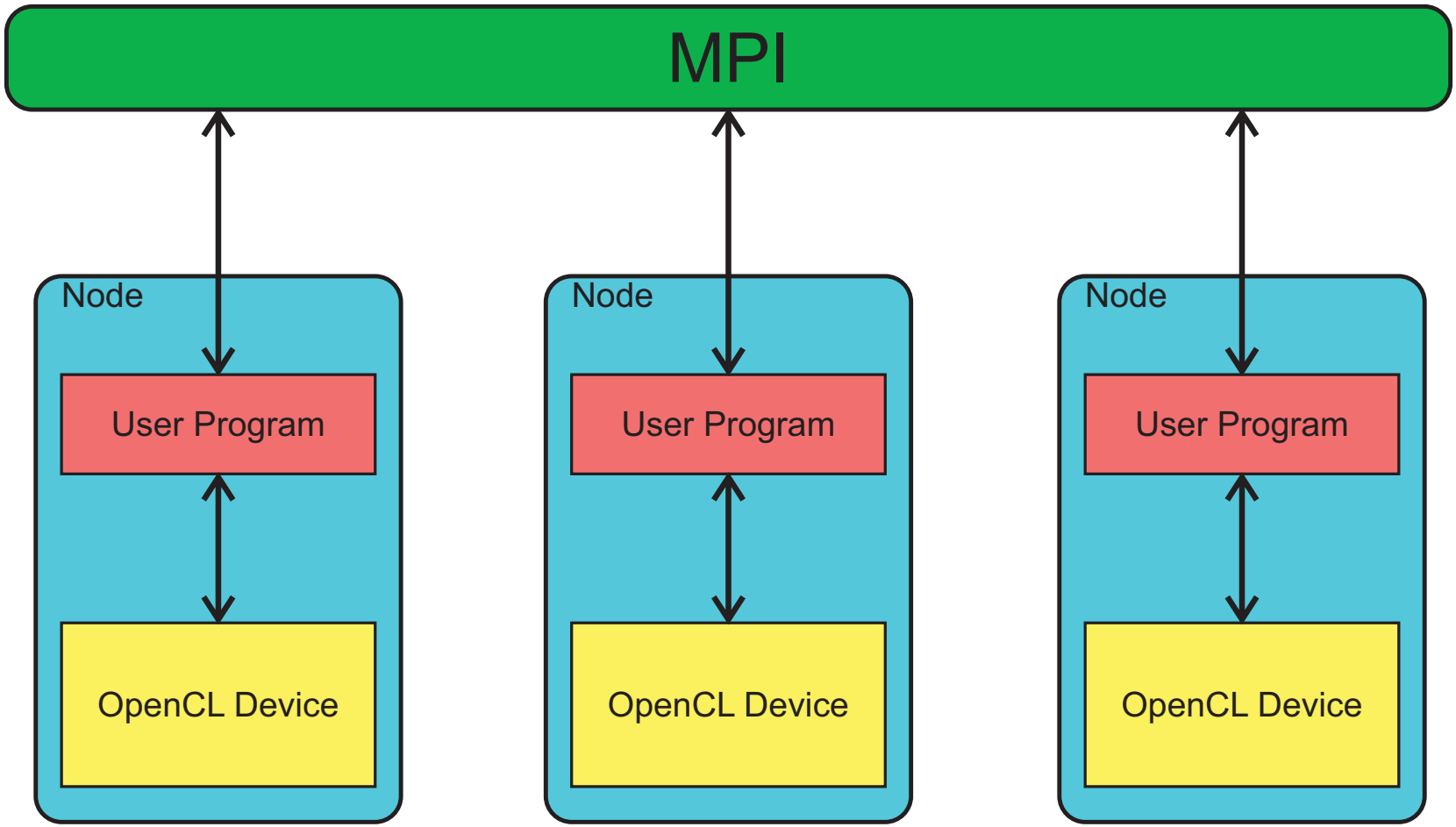
Disadvantages:

- MPI and OpenCL are independent from each other

  $\Longrightarrow$ Connection between computation and data exchange has to be implemented manually

- Every OpenCL device can only be accessed within its own node
- If no further methodes are used, the whole cluster will run in lockstep

Distributed OpenCL with MPI



Outline

HPX

What is HPX?

- A scaling C++ runtime system for parallel and distributed applications
- Based on the ParalleX model

Advantages for distributed OpenCL:

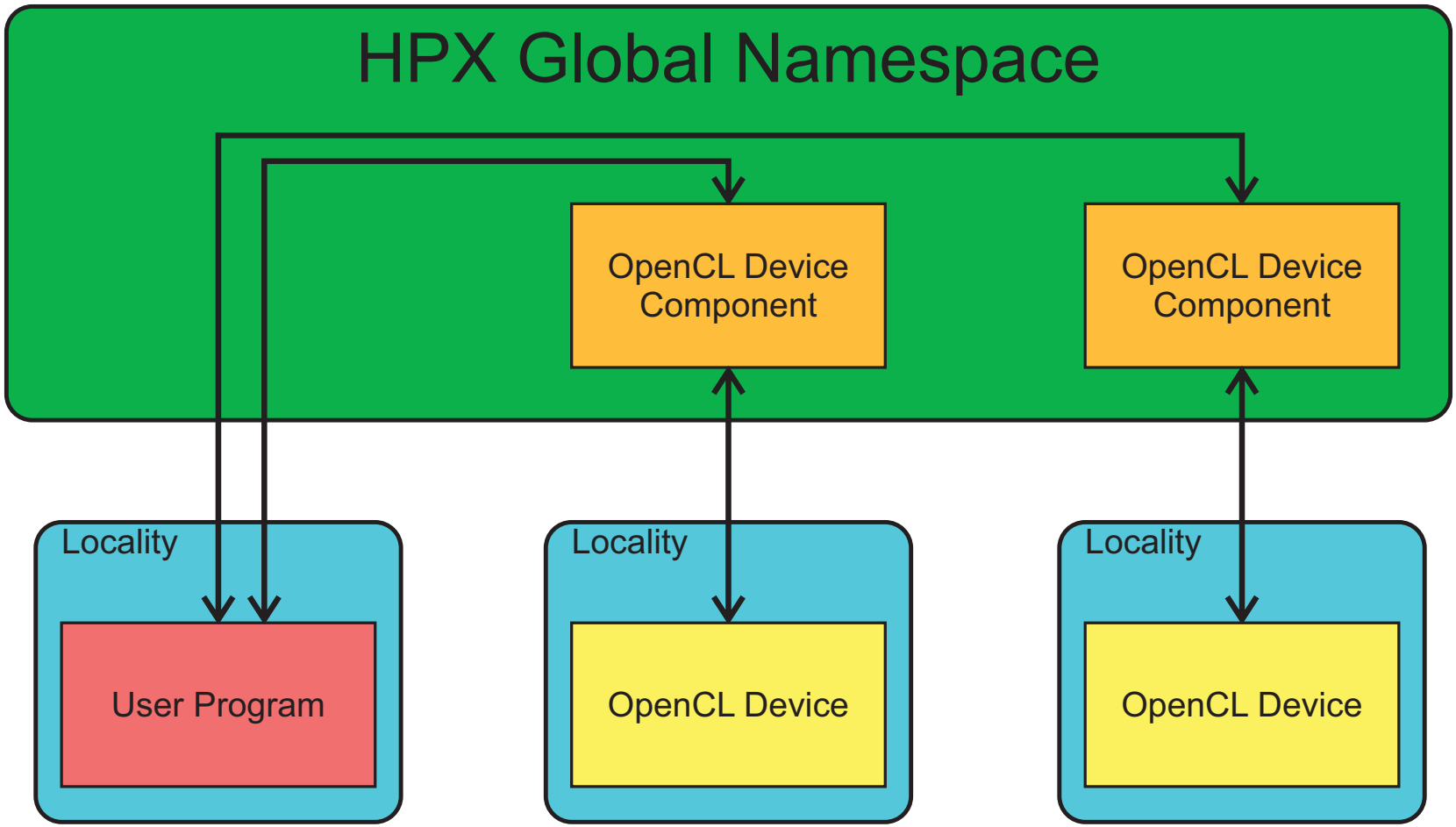
- Global Namespace
- Cluster as "one large machine" (MPI: every Node is autonomous)
- Data dependencies (futures) (MPI: Send-Wait)

HPXCL

- Is our implementation of a distributed OpenCL runtime
- Uses HPX as distribution mechanism
- Wraps every OpenCL datastructure in an HPX component:

OpenCL	HPXCL
cl_device	hpx::opencl::device
cl_program	hpx::opencl::program
cl_kernel	hpx::opencl::kernel
cl_mem	hpx::opencl::buffer
cl_event	hpx::opencl::event
	(soon: hpx::future)

Distributed OpenCL with HPXCL



Effect on distributed GPGPU programming

- Abstracting the whole cluster as one machine
- Simpler, no need to think in a distributed way
- Data dependencies
  - faster due to prevention of lockstep
  - possible to apply standard OpenCL synchronization techniques
- Seamless integration of additional OpenCL nodes into the system
- Possibility to run heterogeneous nodes/devices in one system
- Easy to port non-distributed code to distributed OpenCL whilst maintaining descent scaling

Outline

Implementing "Hello, World!" with HPXCL

- Retrieving an OpenCL device:

```
30 // Get list of available OpenCL devices
31 std::vector<hpx::opencl::device> devices =
32     hpx::opencl::get_all_devices( CL_DEVICE_TYPE_ALL,
33                                   "OpenCL 1.1" ).get();
34
35 // Check whether there are any devices
36 if(devices.size() < 1)
37 {
38     hpx::cerr << "No OpenCL devices found!" << hpx::endl;
39     return hpx::finalize();
40 }
41
42 // Choose the first device found
43 hpx::opencl::device cldevice = devices[0];
44
```

Implementing "Hello, World!" with HPXCL

- Creating a buffer:

```
40 // Create a buffer
41 hpx::opencl::buffer buf =
42     cldevice.create_buffer(CL_MEM_READ_WRITE, 14);
43
```

- Writing to the buffer:

```
44 // Create some data
45 const char some_data[] = { '\x47', '\x64', '\x6b', '\x6b',
46                            '\x6e', '\x2b', '\x1f', '\x56',
47                            '\x6e', '\x71', '\x6b', '\x63',
48                            '\x20', '\xff' };
49
50 // Write data to buffer
51 auto write_done = buf.enqueue_write(0, 14, some_data);
52
```

Implementing "Hello, World!" with HPXCL

- Creating a kernel:

```
53 const char hello_world_src[] =
54 "   __kernel void hello_world(__global char * buf)       \n"
55 "   {                                                    \n"
56 "       size_t tid = get_global_id(0);                   \n"
57 "       buf[tid] = buf[tid] + 1;                         \n"
58 "   }                                                    \n";
59
60 // Create the program
61 hpx::opencl::program prog =
62     cldevice.create_program_with_source(hello_world_src);
63 prog.build();
64
65 // Create the kernel
66 hpx::opencl::kernel hello_world_kernel =
67     prog.create_kernel("hello_world");
68
```

Implementing "Hello, World!" with HPXCL

- Connecting the buffer to the kernel:

```
69 // Set the buffer as kernel argument
70 hello_world_kernel.set_arg(0, buf);
71
```

- Executing a kernel:

```
72 // Create the work dimensions
73 hpx::opencl::work_size<1> dim;
74 dim[0].offset = 0;
75 dim[0].size = 14;
76
77 // Run the kernel
78 auto kernel_done = hello_world_kernel.enqueue(dim,
79                                               write_done);
80
```

Implementing "Hello, World!" with HPXCL

- Reading the result from the buffer:

```
81 // Read from the buffer
82 auto read_result = buf.enqueue_read(0, 14, kernel_done);
83
84 // Get the data (blocking call)
85 hpx::serialize_buffer<char> data_ptr = read_result.get();
86
87 // Print the data. This will print "Hello, World!".
88 hpx::cout << data_ptr.data() << hpx::endl;
89
90 // Gracefully shut down HPX
91 return hpx::finalize();
92
```

Outline

The Mandelbrot Renderer

Mandelbrot Algorithm

The Mandelbrot set is based on the complex series:

$$z_{n+1} = {z_n}^2 + c, \quad z_0 = 0$$

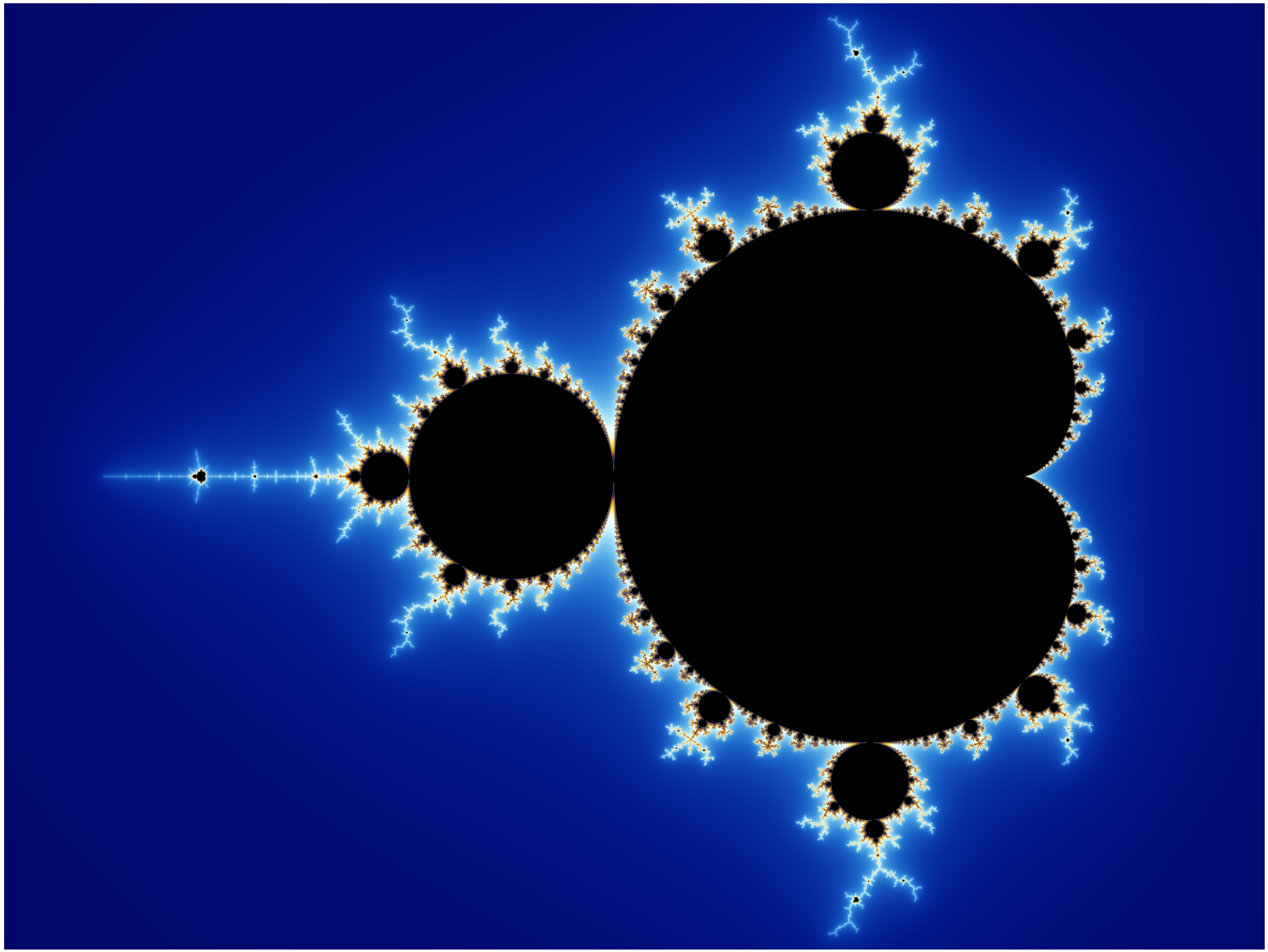
The set itself is defined as: $\left\{c \in \mathbb{C} : \exists s \in \mathbb{R}, \forall n \in \mathbb{N}, |z_n| < s\right\}$

Creating Mandelbrot Images

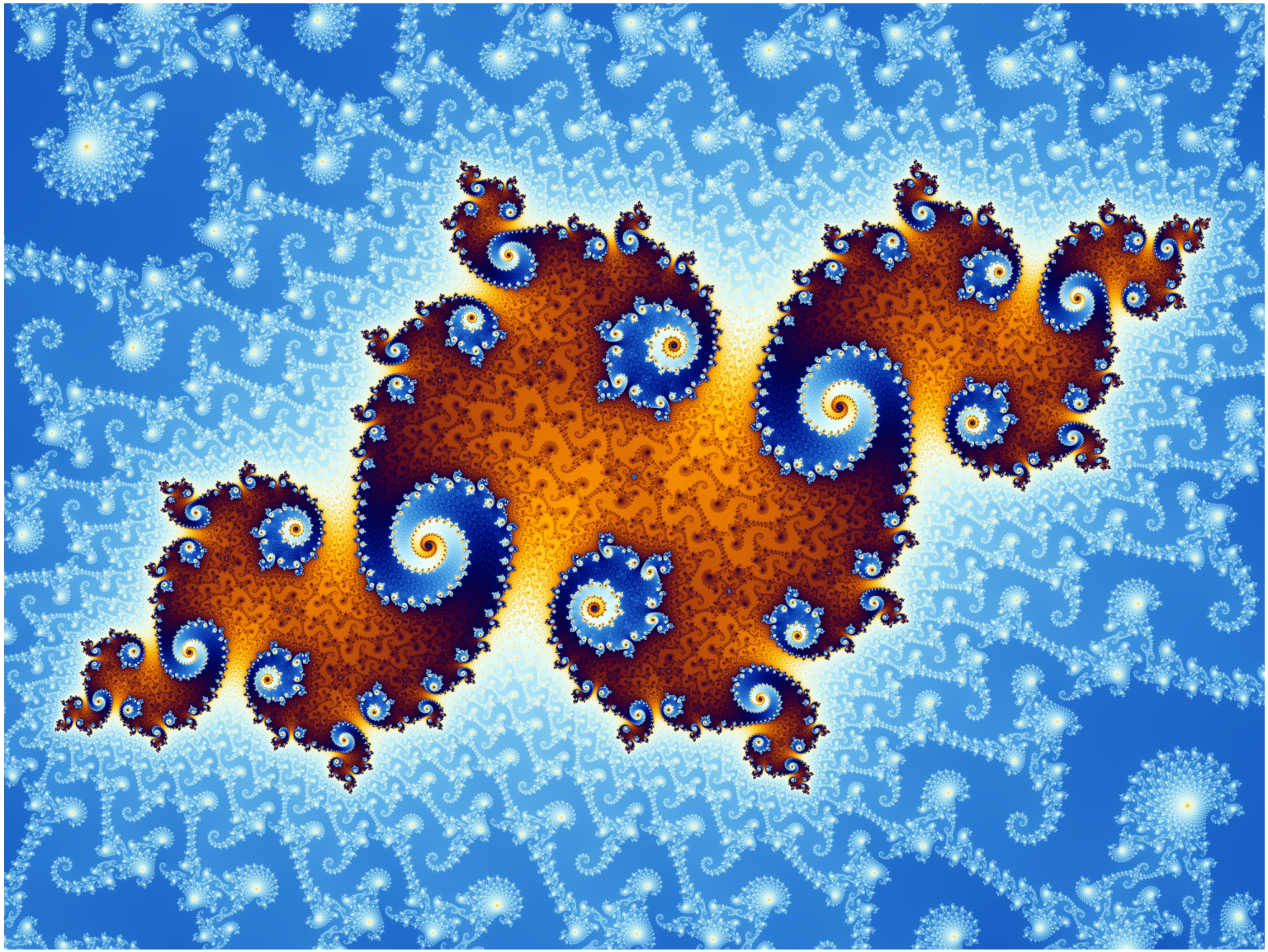
The mandelbrot set can be visualized by associating every pixel with a coordinate $\begin{pmatrix} x \\ y \end{pmatrix}$ and then setting $c = y * i + x$.

Coloring pixels by how fast the series diverges can create impressive images.

The Mandelbrot Renderer



The Mandelbrot Renderer



The Mandelbrot Renderer

Stats for Nerds

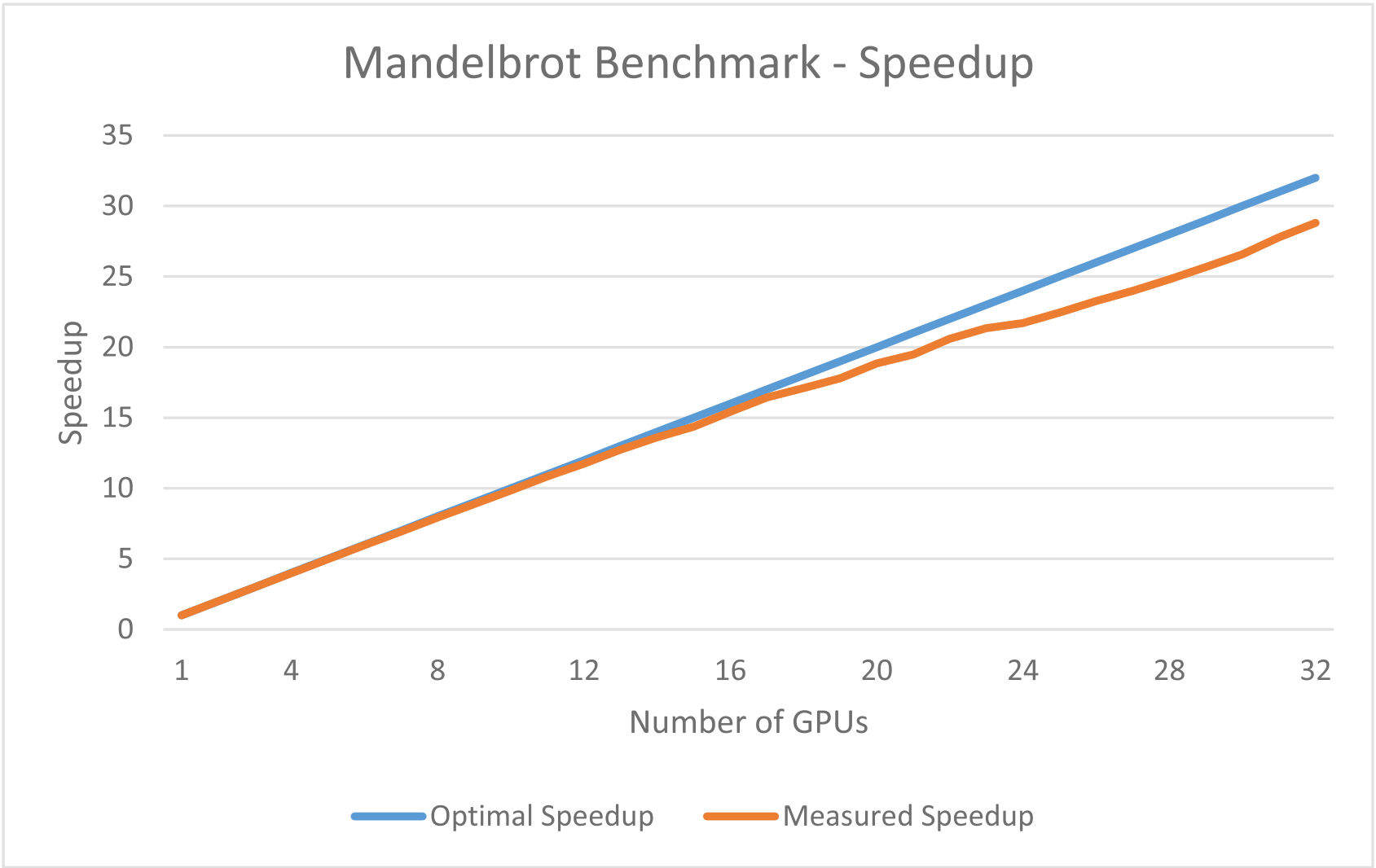
- Resolution: 2560x1920
- Smoothing: 8x8 Supersampling
- Bailout: 10000
- Maximum iterations: 50000
- GPUs: 32x NVidia Tesla K20
- Render time:

The Mandelbrot Renderer

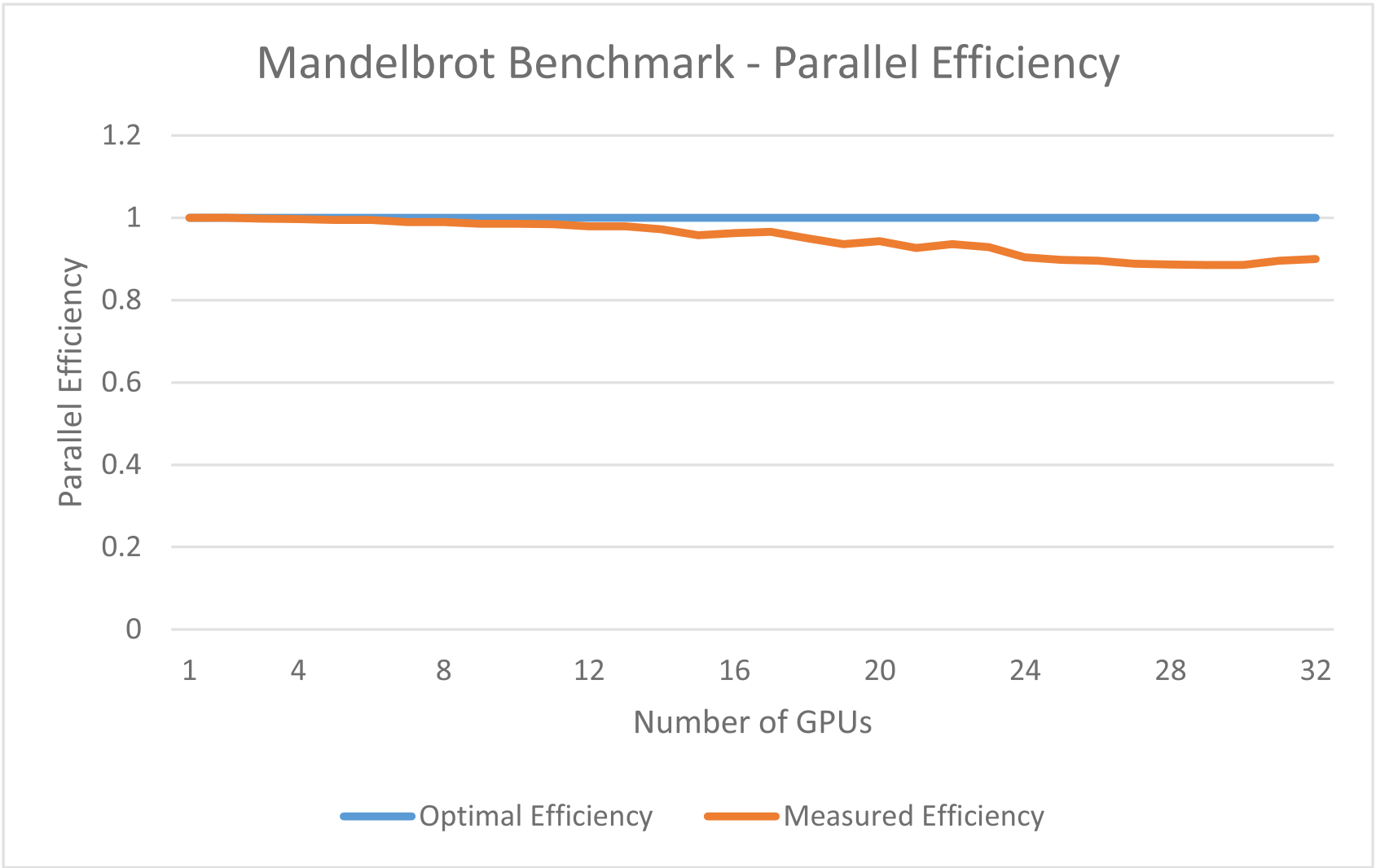
Stats for Nerds

- Resolution: 2560x1920
- Smoothing: 8x8 Supersampling
- Bailout: 10000
- Maximum iterations: 50000
- GPUs: 32x NVidia Tesla K20
- Render time: 0.6 seconds

Speedup


Mandelbrot Benchmark - Speedup
Speedup
35
30
25
20
15
10
5
0
1
4
8
12
16
20
24
28
32
Number of GPUs
Optimal Speedup
Measured Speedup

Parallel Efficiency



Adding Google Maps

- We combined the renderer with the Google Maps API v3: