Distributed GPGPU Computing

Martin Stumpf

Ste||ar Group, LSU

September 19, 2014

Table of Contents

Outline

CPU vs GPU

"<IMAGE GPU vs CPU>"

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:

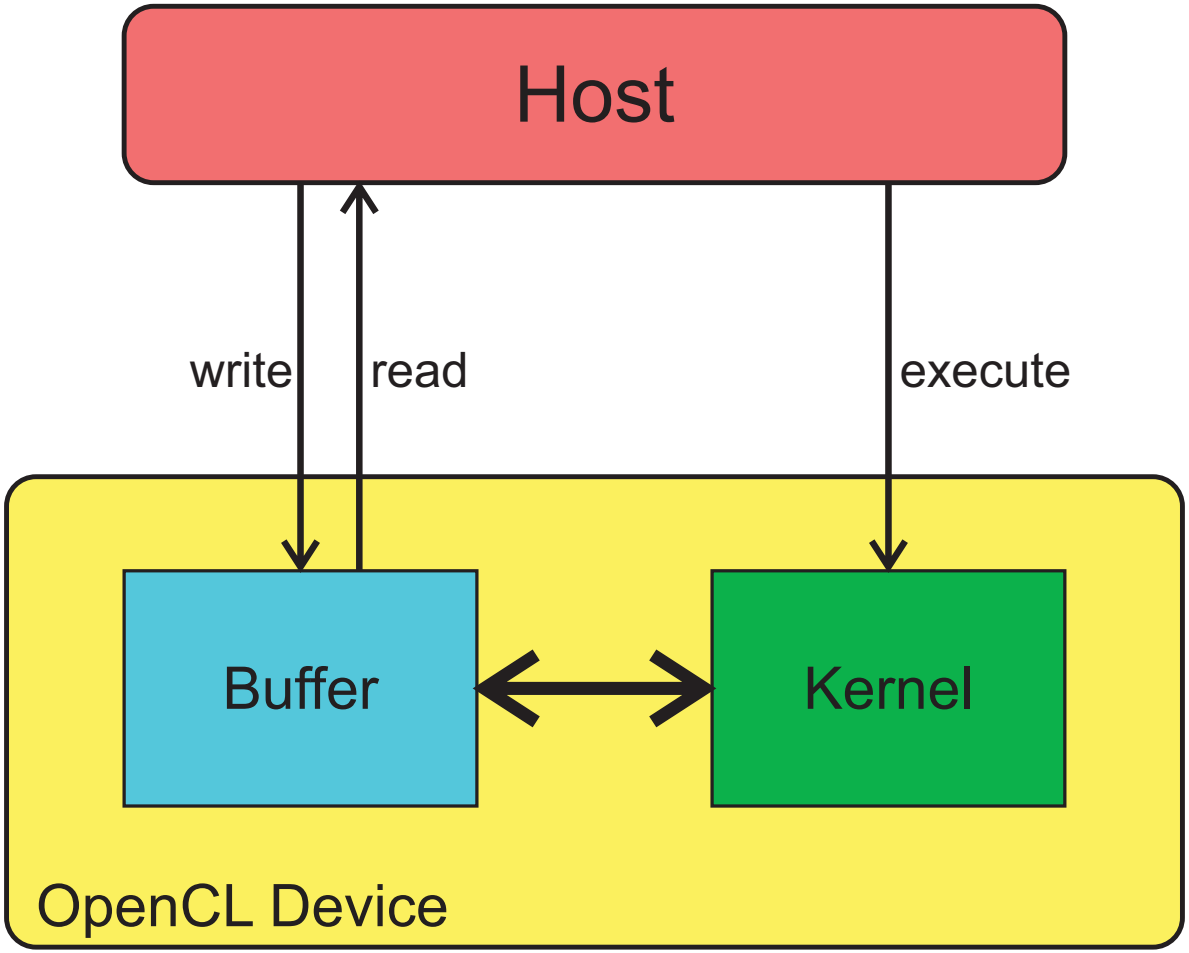
- CUDA (NVidia)
- OpenCL (Platform independent)

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

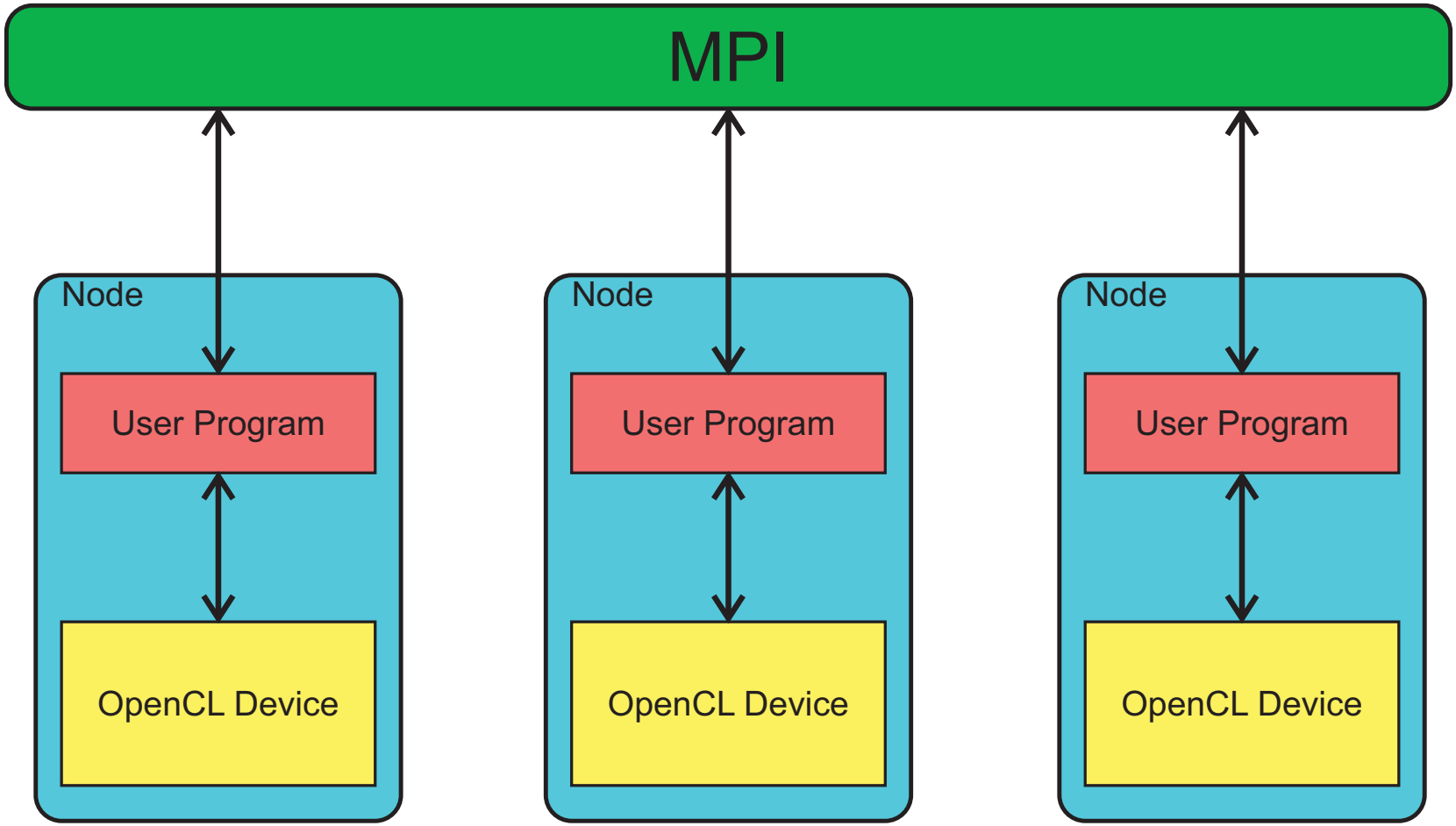
  $\Longrightarrow$ OpenCL programs are device independent
- Kernel executions on the device run asynchronous to the host program

OpenCL



Outline

Distributed OpenCL with MPI



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

Outline

HPX

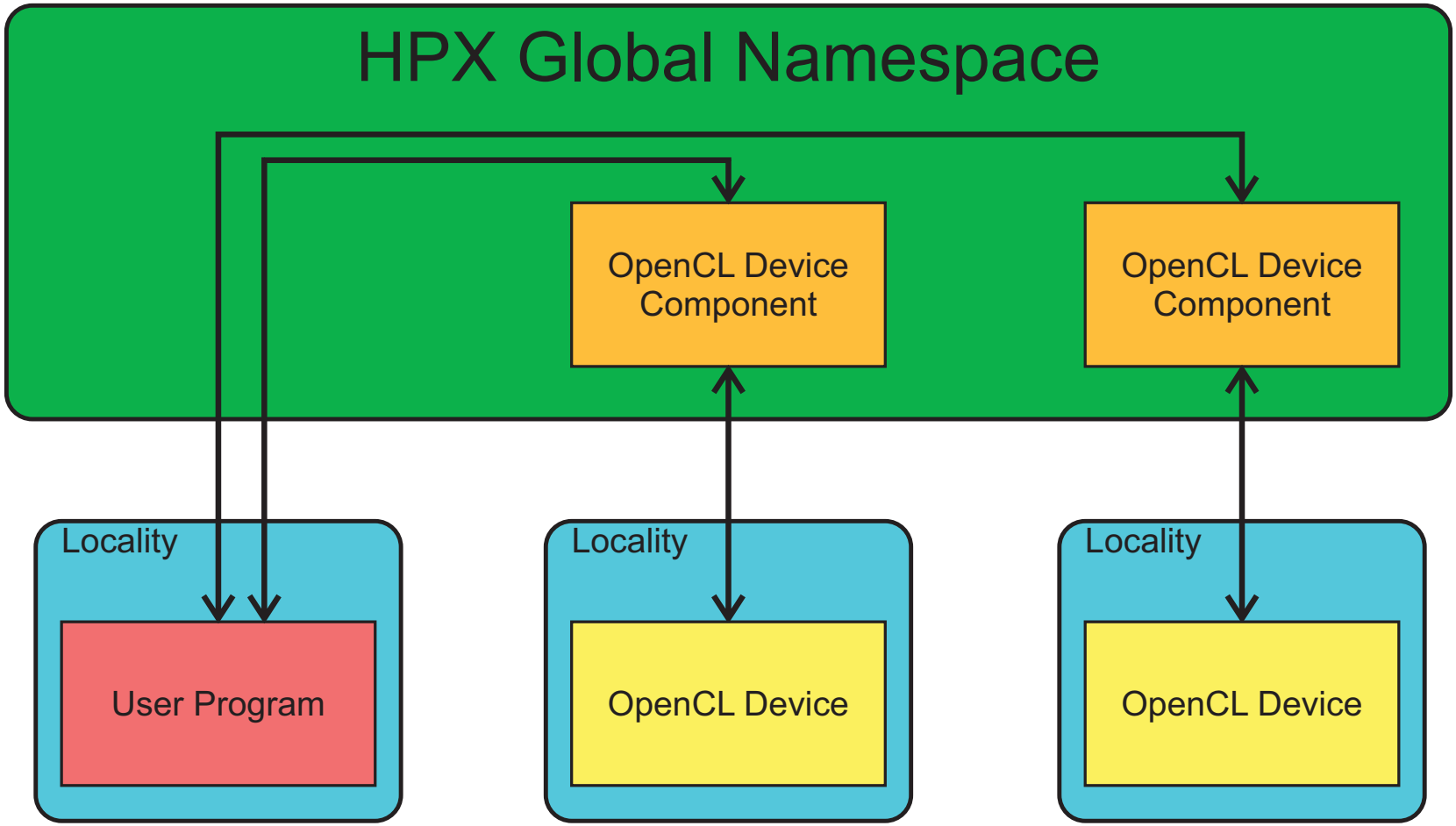
What is HPX?

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

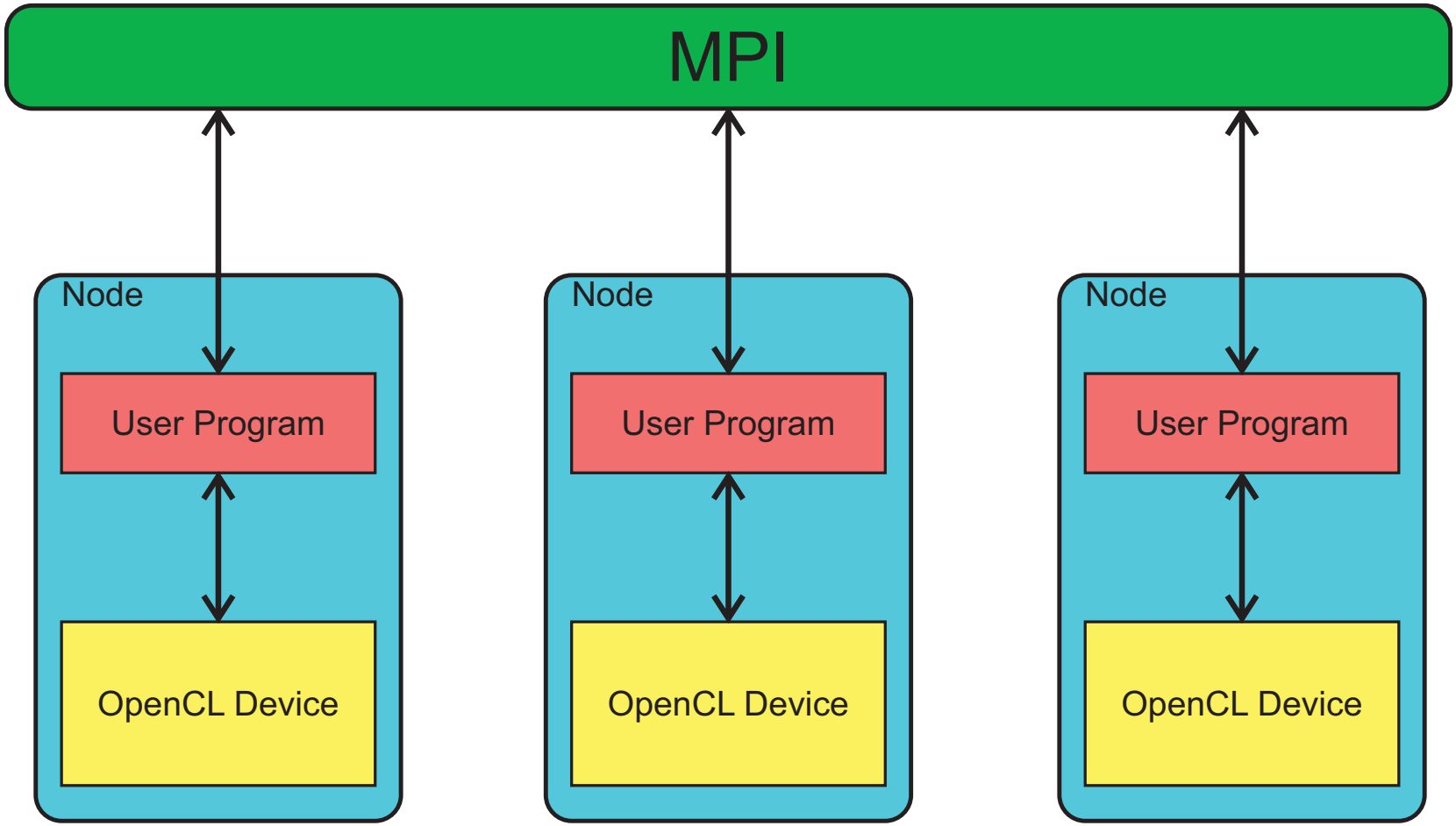
Advantages for distributed OpenCL:

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

Distributed OpenCL with HPX



Distributed OpenCL with MPI



Affect on distributed GPGPU programming

- Abstracting the whole cluster as one machine
- Simpler, not 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 more 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

HPXCL - Overview

- Is an implementation of that concept.
- Wraps every OpenCL datastructure in a 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)

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', '\x65', '\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

- Executing a kernel:

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

Implementing "Hello, World!" with HPXCL

- Reading the result from the buffer:

```
78 // Read from the buffer
79 auto read_result = buf.enqueue_read(0, 14, kernel_done);
80
81 // Get the data (blocking call)
82 hpx::serialize_buffer<char> data_ptr = read_result.get();
83
84 // Print the data. Will print "Hello, World!".
85 hpx::cout << data_ptr.data() << hpx::endl;
86
87 // Gracefully shut down HPX
88 return hpx::finalize();
89
```

Outline

bla

Scaling

Parallel Efficiency