Portable Parallel Performance

Where stands SYCL in this landscape ?

Les Jeudis Gray Scott

David Chamont, march 2024

OpenCL (2008)

vectorization + acceleration,
portable across CPU, GPU, FPGA ...
damned !?!

Les Jeudis Gray Scott
David Chamont, march 2024

OpenCL example / kernel

```
#include <CL/opencl.h>

.....

const char * KernelSource = "                \n"
"  __kernel void square(                     \n"
"    __global float* input,                  \n"
"    __global float* output,                 \n"
"    const unsigned int count)               \n"
"  {                                         \n"
"    int i = get_global_id(0);               \n"
"    if(i < count)                           \n"
"      output[i] = input[i]*input[i];         \n"
"  }                                         \n"
"                                            \n";

.....
```

OpenCL example / main

```
.....

int main(int argc, char** argv)
{
  ..... create and initialize host input/output arrays .....

  // Prepare kernel
  cl_int err;
  cl_platform_id platform_id;
  err = clGetPlatformIDs(1, &platform_id, NULL);
  cl_device_id device_id ;
  err = clGetDeviceIDs(platform_id,CL_DEVICE_TYPE_GPU,1,&device_id,NULL);
  cl_context context = clCreateContext(0,1,&device_id,NULL,NULL,&err);
  cl_command_queue queue = clCreateCommandQueue(context,device_id,0,&err);
  cl_program prog = clCreateProgramWithSource(context,1,(const char **)&KernelSource,...
  err = clBuildProgram(prog,0,NULL,NULL,NULL,NULL);
  cl_kernel kernel = clCreateKernel(prog,"square",&err);

  // Device io arrays
  cl_mem input_buffer = clCreateBuffer(context,CL_MEM_READ_ONLY,sizeof(float)*count,...
  cl_mem output_buffer = clCreateBuffer(context,CL_MEM_WRITE_ONLY,sizeof(float)*count,...

  .....
```

OpenCL example / main

```
.....

// Write our data set into the input array in device memory
err = clEnqueueWriteBuffer(queue,input_buffer,CL_TRUE,0,sizeof(float)*count,input,0,...

// Set arguments to kernel
err  = clSetKernelArg(kernel,0,sizeof(cl_mem),&input_buffer);
err |= clSetKernelArg(kernel,1,sizeof(cl_mem),&output_buffer);
err |= clSetKernelArg(kernel,2,sizeof(unsigned int),&count);

// Queue kernel and wait for its execution end
size_t global = count;
err = clEnqueueNDRangeKernel(queue,kernel,1,NULL,&global,NULL,0,NULL,NULL);
clFinish(commands);

// Read back the results from the device to verify the output
err = clEnqueueReadBuffer(queue,output_buffer,CL_TRUE,0,sizeof(float)*count,output,0,...

.....
```

OpenCL example / main

```
.....

// Process results
.....

// Shutdown and cleanup
clReleaseMemObject(input);
clReleaseMemObject(output);
clReleaseProgram(prog);
clReleaseKernel(kernel);
clReleaseCommandQueue(commands);
clReleaseContext(context);
return 0 ;
}
```

SYCL (2013)

rise to C++

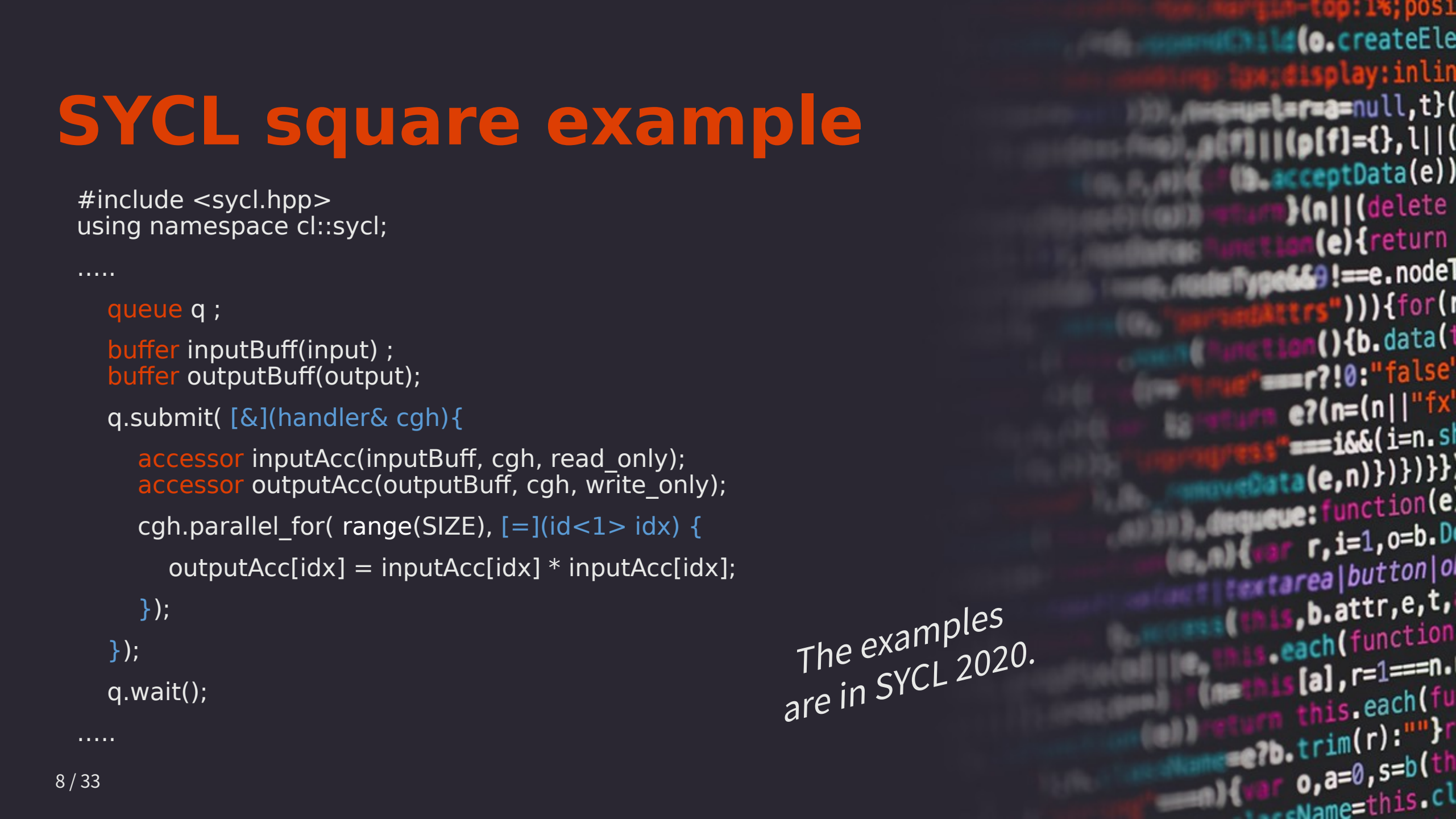
Les Jeudis Gray Scott

David Chamont, march 2024

SYCL square example

```
#include <sycl.hpp>
using namespace cl::sycl;

.....

    queue q ;

    buffer inputBuff(input) ;
    buffer outputBuff(output);

    q.submit( [&](handler& cgh){

        accessor inputAcc(inputBuff, cgh, read_only);
        accessor outputAcc(outputBuff, cgh, write_only);

        cgh.parallel_for( range(SIZE), [=](id<1> idx) {

            outputAcc[idx] = inputAcc[idx] * inputAcc[idx];

        });

    });

    q.wait();

.....
```

The examples are in SYCL 2020.



SYCL selector

```
.....
    queue q ;                          // selection of «best» device
    queue q(default_selector_v); // selection of «best» device
    queue q(cpu_selector_v);      // prefers CPU
    queue q(gpu_selector_v);      // prefers GPU

.....
    queue q([&]( device const & d ){
        if (arg_cpu) return cpu_selector_v(d) ;
        else if (arg_gpu) return gpu_selector_v(d) ;
        else .....
    });

.....
    std::cout
        << "Name: "<<q.get_device().get_info<info::device::name>()<< "\n";
        << "Vendor: "<< q.get_device().get_info<info::device::vendor>()<< "\n";

.....
```

SYCL explicit nd-range

```
.....
        cgh.parallel_for( nd_range(SIZE,SUBSIZE), [=](nd_item<1> it) {
            int ig = it.get_global_id(0);
            int il = it.get_local_id(0) ;
            outputAcc[ig] = inputAcc[ig] * inputAcc[ig];
        });
.....
```

SYCL shared memory

```
.....
    auto * input  = malloc_shared<float>(SIZE, q);
    auto * output = malloc_shared<float>(SIZE, q);

    .....
    q.submit( [&](handler& cgh){
        cgh.parallel_for( range<1>(SIZE), [=](id<1> idx) {
            output[idx] = input[idx] * input[idx];
        }) ;
    }) ;
.....
```

SYCL explicit copy

```
.....
    auto * dinput  = malloc_device<float>(SIZE, q);
    auto * doutput = malloc_device<float>(SIZE, q);

    .....
    q.memcpy(dinput,input,SIZE*sizeof(float)).wait();
    q.submit( [&](handler& cgh){
        cgh.parallel_for( range<1>(SIZE), [=](id<1> idx) {
            doutput[idx] = dinput[idx] * dinput[idx];
        }) ;
    }).wait() ;
    q.memcpy(output,doutput,SIZE*sizeof(float)).wait();
.....
```

SYCL in-order queues

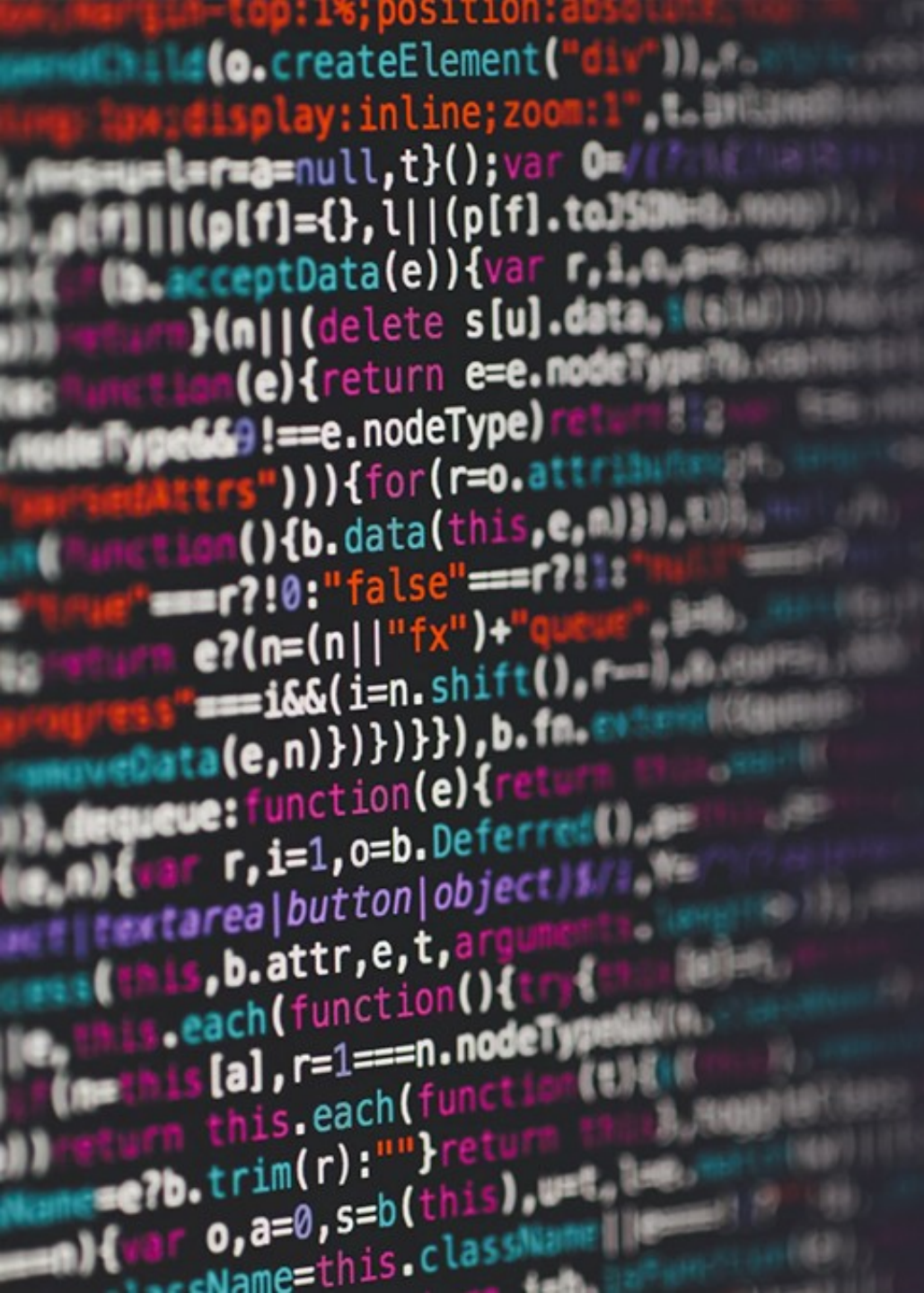
```
.....
    queue q(property::queue::in_order());
    .....
    q.memcpy(dinput,input,SIZE*sizeof(float));
    q.submit( [&](handler& cgh){
        cgh.parallel_for( range<1>(SIZE), [=](id<1> idx) {
            doutput[idx] = dinput[idx] * dinput[idx];
        });
    });
    q.memcpy(output,doutput,SIZE*sizeof(float));
    q.wait() ;
.....
```

SYCL events

```
.....
    queue q;
    .....
    event e1 = q.memcpy(dinput,input,SIZE*sizeof(float));
    event e2 = q.submit( [&](handler& cgh){
       cgh.depends_on(e1) ;
       cgh.parallel_for( range<1>(SIZE), [=](id<1> idx) {
          doutput[idx] = dinput[idx] * dinput[idx];
       });
    });
    q.submit( [&](handler& cgh){
       cgh.depends_on(e2);
       cgh.memcpy(output,doutput,SIZE*sizeof(float));
    });
    q.wait() ;
.....
```

SYCL buffers & accessors

```
#include <sycl.hpp>
using namespace cl::sycl;

.....

    queue q ;

    buffer inputBuff(input) ;
    buffer outputBuff(output);

    q.submit( [&](handler& cgh){

        accessor inputAcc(inputBuff, cgh, read_only);
        accessor outputAcc(outputBuff, cgh, write_only);

        cgh.parallel_for( range(SIZE), [=](id<1> idx) {

            outputAcc[idx] = inputAcc[idx] * inputAcc[idx];

        });

    });

    q.wait();

.....
```



Behind the curtain

Installation...compilation...execution

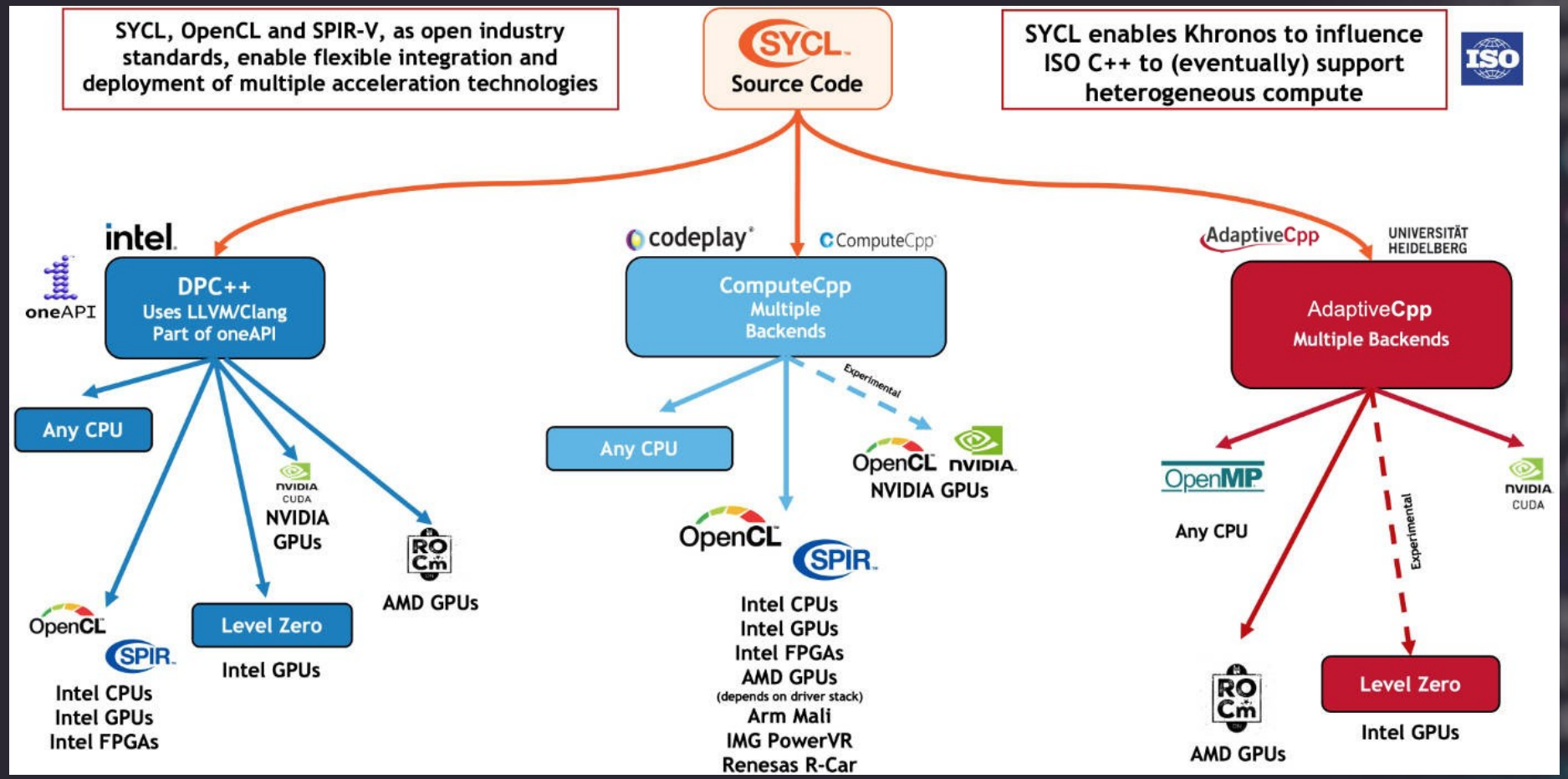
Les Jeudis Gray Scott

David Chamont, march 2024

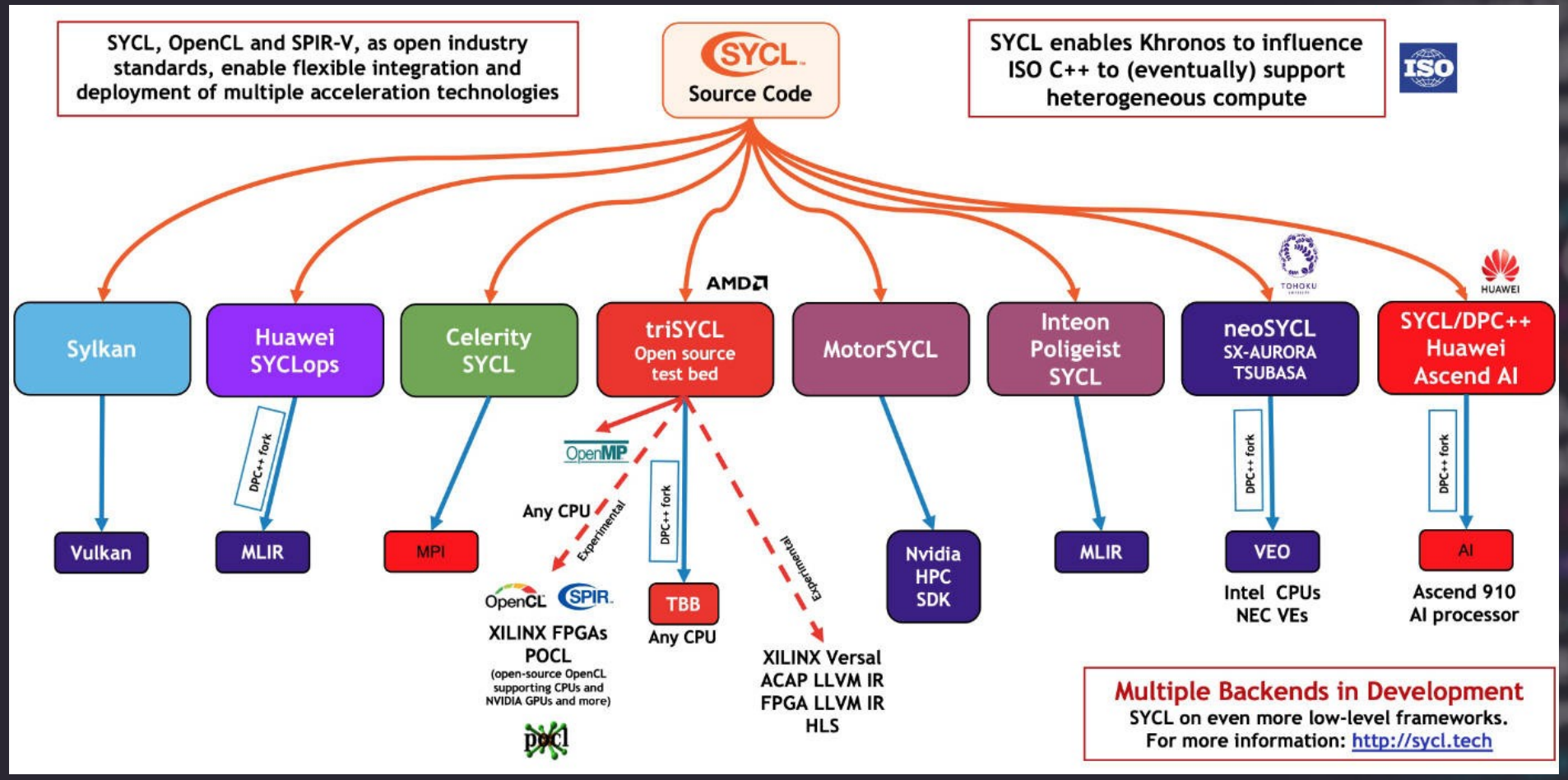
SYCL history

- 2013: initial proposal from Codeplay Software
- 2015: SYCL 1.2, released by the Khronos Group
- 2016: integrated into Intel FPGA SDK for OpenCL
- 2016: SYCL 2.2, target C++14 and OpenCL 2.2
- 2018: Xilinx announced support for SYCL
- 2020: SYCL 2020, open the door for CUDA backend
- 2022: Intel acquire Codeplay
- ⋯ support for C++17/20/23, OpenCL 3.0, types for AI (half, bfloat16), backends (ROCm 5.0, oneAPI, CUDA 11.x)

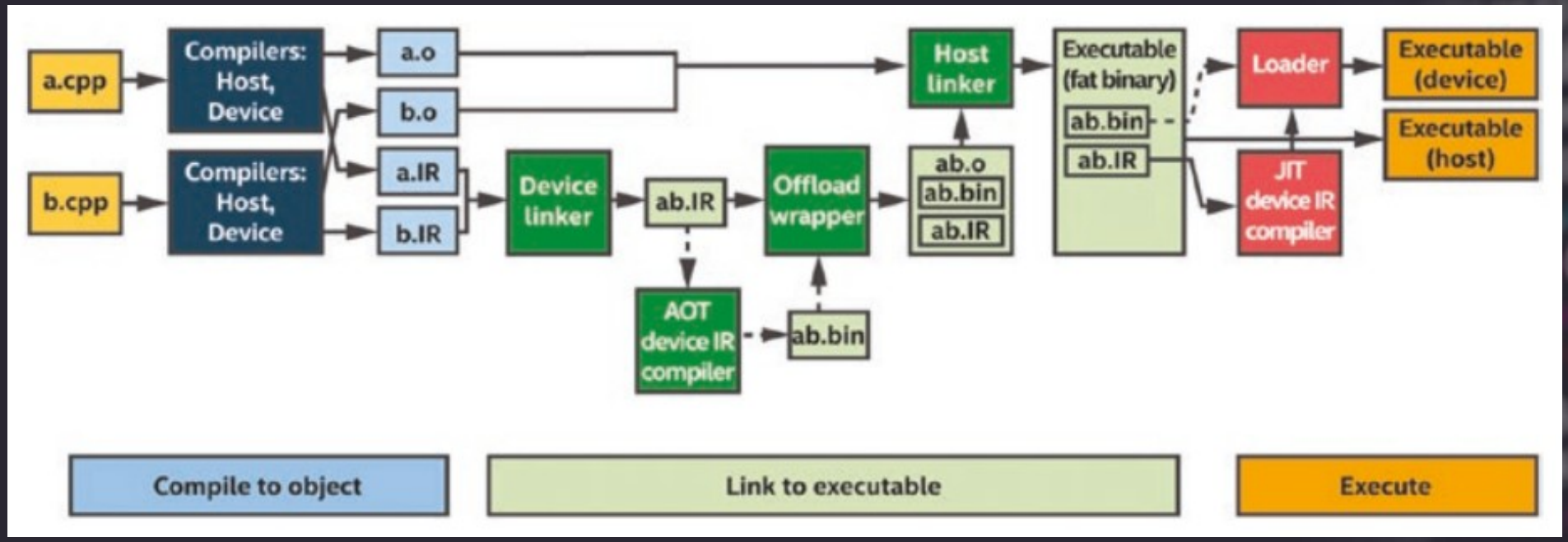
SYCL implems & backends



SYCL implems & backends



DPC++ : AoT and/or JiT



oneAPI (2018)

the Intel touch

Les Jeudis Gray Scott

David Chamont, march 2024

oneAPI history

- 2018: announced at the Intel Developer Forum
- 2019: Base Toolkit released
- 2020: oneAPI 1.0, devcloud, Level Zero API, support for AMD and NVIDIA hardware.
- 2022: support for ARM and RISC-V
- 2023: oneAPI evolves into the Unified Acceleration Fundation (UXL), under control of the Linux Fundation.

oneAPI ecosystem

- Level Zero: the low-level hardware interface
- DPC++: an implementation of SYCL with extensions (powered by LLVM)
- SYCLomatic: CUDA-to-SYCL code migration tool
- oneDPL (algorithms), oneMKL (math), oneDAL (machine learning), oneDNN (deep learning), oneCCL (communication), oneTBB (threads), oneVPL (video), ...

And yet...

… in high energy physics,
no consensus on some common
portable parallelization solution

Les Jeudis Gray Scott
David Chamont, march 2024

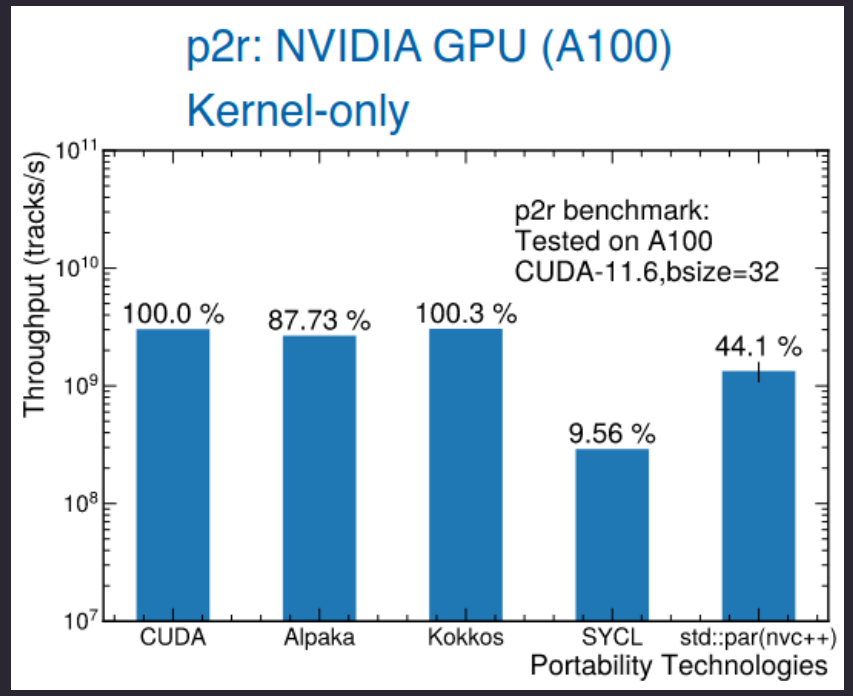
Hardware vs solutions (2023)

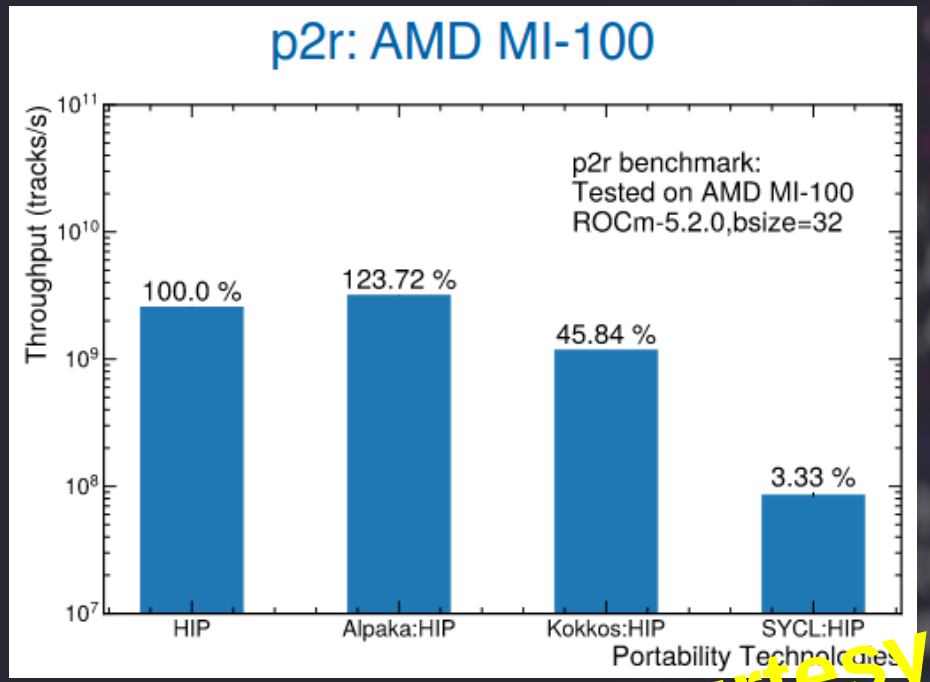
	CUDA	Kokkos	SYCL	HIP	OpenMP	alpaka	std::par
NVIDIA GPU				*hipcc*	*nvc++ LLVM, Cray GCC, XL*		*nvc++*
AMD GPU			*openSYCL intel/llvm*	*hipcc*	*AOMP LLVM Cray*		
Intel GPU			*oneAPI intel/llvm*	CHIP-SPV: *early prototype*	*Intel OneAPI compiler*	*prototype*	*oneapi::dpl*
x86 CPU			*oneAPI intel/llvm openSYCL*	*via HIP-CPU Runtime*	*nvc++ LLVM, CCE, GCC, XL*		
FPGA				*via Xilinx Runtime*	*prototype compilers (OpenArc, Intel, etc.)*	*protytype via SYCL*	

Courtesy of HEP-CCE

CHEP'23

"There are a number of portability layers... each of these has its own characteristics, performing better at some tasks and worse at others, or placing limitations on aspects of the application."





Courtesy of HEP-CCE

ACAT’24

- GenVectorX (lorentz vectors algebra): “...SYCL can achieve a good performance portability with respect to CUDA and even outperform it. Furthermore, we have compared two main different implementations of SYCL compiler, namely AdaptiveCpp and Intel(R) OneAPI, showing that it is possible to achieve comparable performance with these tools... SYCL has shown impressive scaling on CPU backends as well, confirming its performance portability capacity with near-zero code divergence...”

IOWCL'24

- MadGraph (events simulation): "...SYCL not only ensures portability but also maintains competitive performance. The adoption of SYCL achieves this without the complexities of managing multiple code bases or extensive use of preprocessor directives..."
- RdataFrame (columnar data): "...This (use-case) leads to suboptimal performance of the application in CUDA, which is further exacerbated when using SYCL due to the additional overhead of managing the runtime. However, ... we acquired support for accelerators beyond NVIDIA GPUs and parallelism within bulks on the CPU... thanks to abstractions such as the SYCL2020 reduction interface, our code has become more concise..."
- SOFIE (inference optimisation): "In this work, we extend the functionality of SOFIE to generate SYCL code for machine learning model inference that can run on various GPU platforms and is only dependent on Intel MKL BLAS and portBLAS libraries"

Conclusions

Personal, subjective & questionable

Les Jeudis Gray Scott

David Chamont, march 2024

Bittersweet news

- Less API boilerplate in user code, which is nice, but a more complex installation procedure, compilation chain and execution runtime.
- Debugging and profiling will benefit from trying several hardware, several implementations, several backends. But when NVidia/AMD hardware is involved, one has to use NVidia/HIP tools and mentally relocate their results in the SYCL code context. This proves painful and not beginner-friendly.

Good news

- More and more hardware is adressed by one SYCL implementation or the other.
- Intel is fully involved, and the SYCL ecosystem seems to close the gap with CUDA.
- Whatever the performance, porting to SYCL will improve your code quality, and ease the transition to any other solution.
- All solutions improve and converge:
  - Kokkos & Alpaka experiments a SYCL backend for Intel GPUs.
  - AdaptiveCpp has a support for C++ std::par.
  - Kokkos includes std::mdspan reference implementation.

Gray Scott Reloaded

- Prerequisites:
  - C++ lambda
  - SYCL setup
- Content: SYCL port of Gray-Scott
  - on CPU, with multi-threading and simd
  - on GPU, optimizing data transfer
- On which platform?
  - CodeReckons
  - MUST, Intel DevCloud, Docker, ...

See you there !

Questions ?

https:// **DPC++ Book**
https:// Khronos Standard
https:// API Reference Guide
https:// SYCL Academy
https:// IOWCL Workshop
https:// UXL Fundation

Les Jeudis Gray Scott
David Chamont, march 2024