codeplay

# SYCL: An Abstraction Layer for Leveraging C++ and OpenCL

**Alastair Murray**
**Compiler Research Engineer, Codeplay**

**Visit us at**
**www.codeplay.com**

45 York Place
Edinburgh
EH1 3HP
United Kingdom

# Overview

- SYCL for OpenCL
- SYCL example
- Hierarchical parallelism
- Parallel STL
- Embedding DSLs
- Conclude

# SYCL for OpenCL

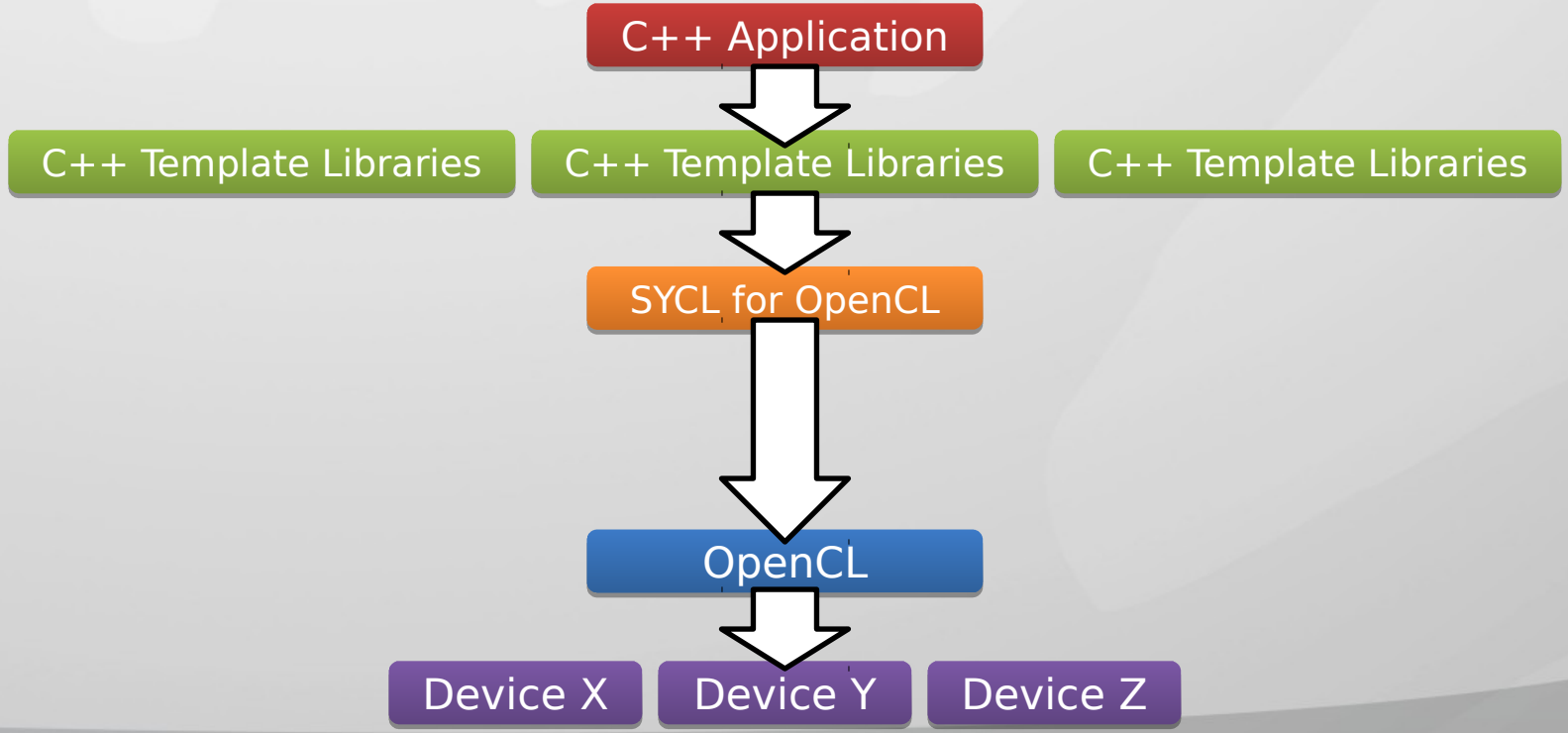
# SYCL for OpenCL


- Cross-platform heterogeneous, parallel C++ programming model.
  - Developed within the Khronos Group.
  - Builds on concepts, portability and efficiency of OpenCL.
  - Ease of use and flexibility of C++.
- Single-source C++ development.
- Anything possible in core OpenCL 1.2 should be possible in SYCL.
- Specification officially released last week.

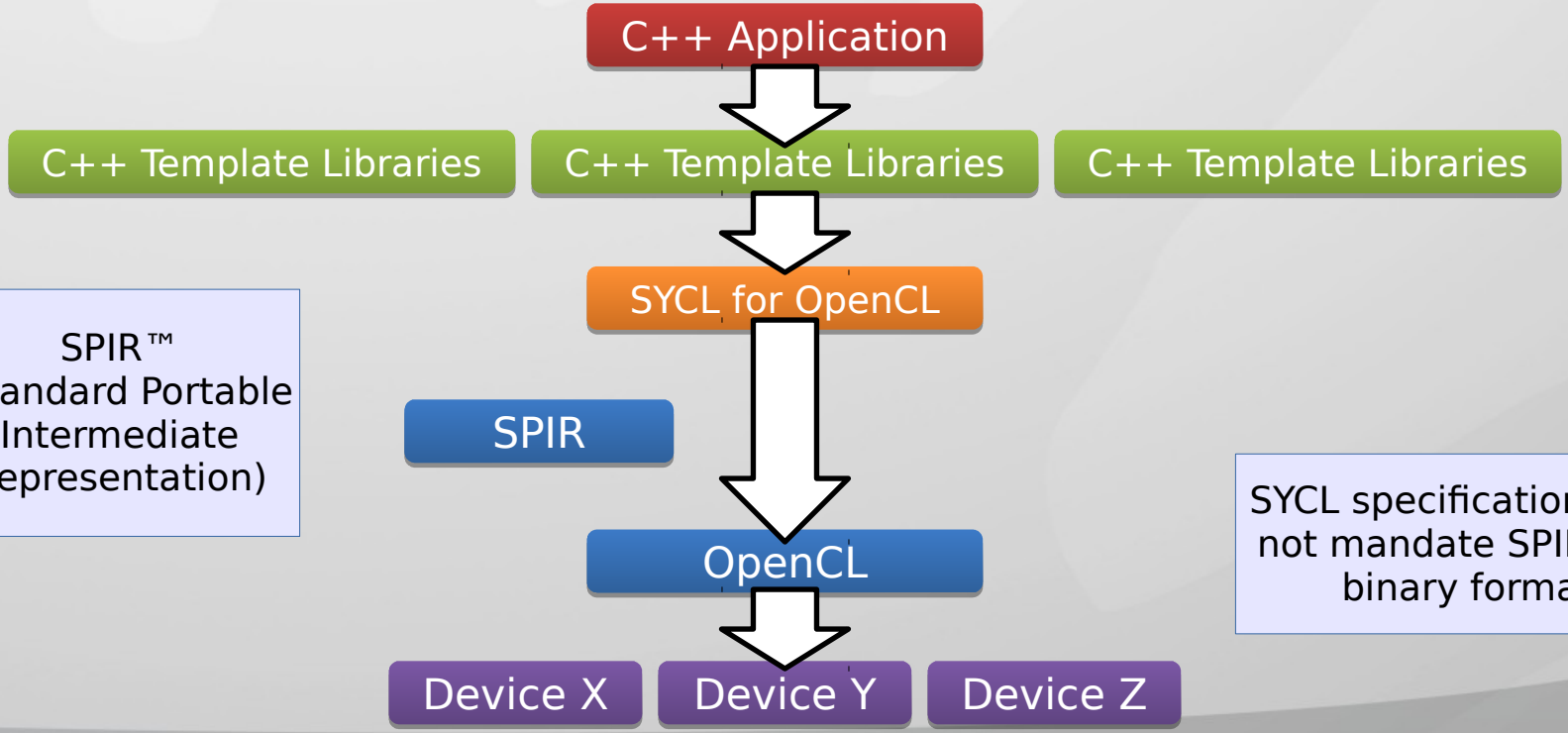
# Motivation Behind SYCL

- To make GPGPU simpler and more accessible.
- To create a C++ for OpenCL™ ecosystem.
  - Combine the ease of use and flexibility of C++ and the portability and efficiency of OpenCL.
- To provide a foundation for constructing complex and reusable template algorithms:
  - parallel_reduce(), parallel_map(), parallel_sort()
- To define an open and portable standard.

# The SYCL Ecosystem



# The SYCL Ecosystem

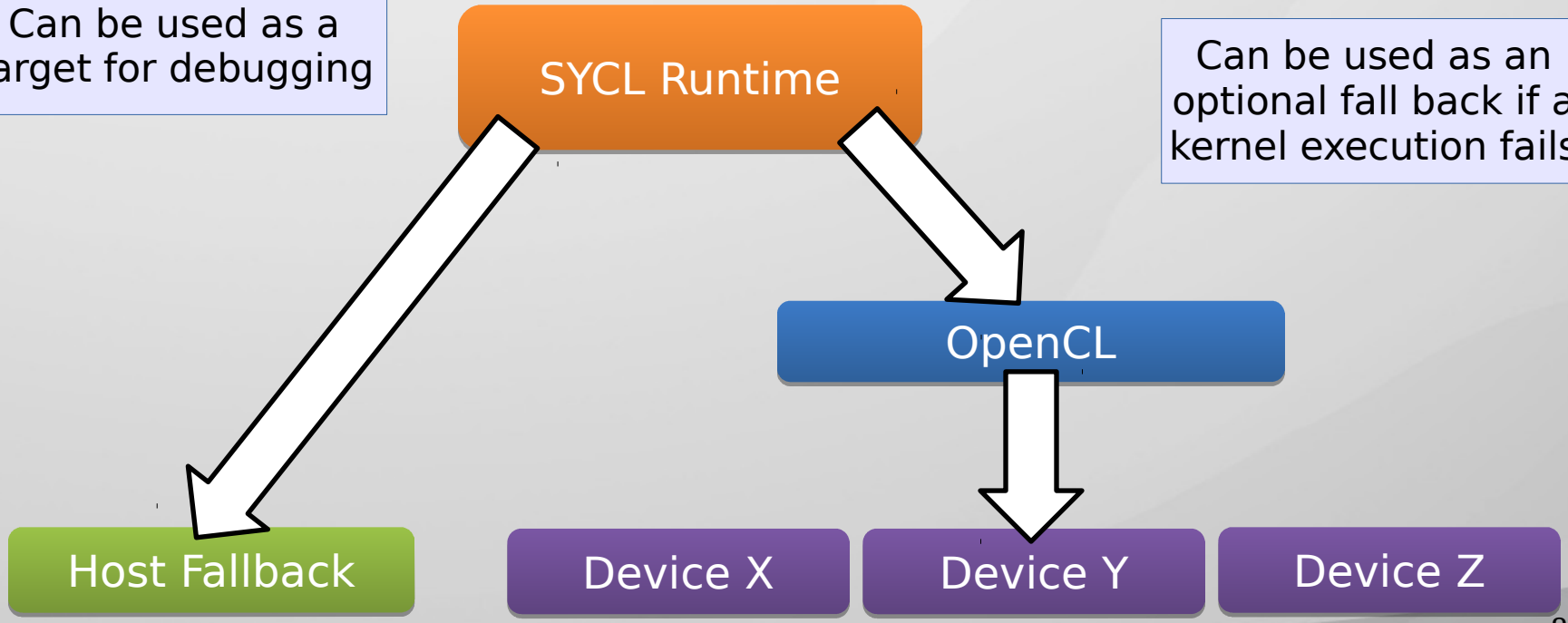



SPIR™ (Standard Portable Intermediate Representation)

SYCL specification does not mandate SPIR as a binary format

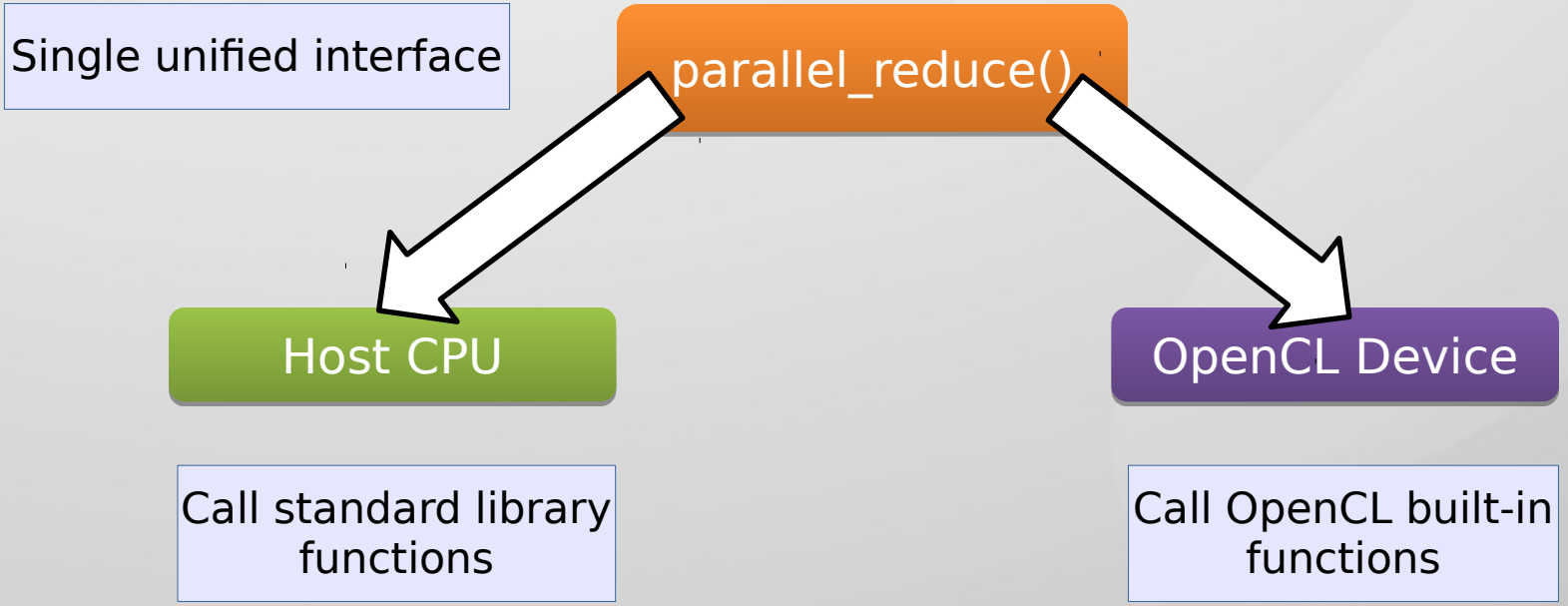
# Host Device

Can be used as a target for debugging




Can be used as an optional fall back if a kernel execution fails

# Use Common Libraries



# What is SYCL for?

- Modern C++ lets us separate the **what** from the **how**.
- Separate **what** users want to do:
  science, computer vision, AI, …
- From **how** to run fast on an OpenCL device.

# What can SYCL do?

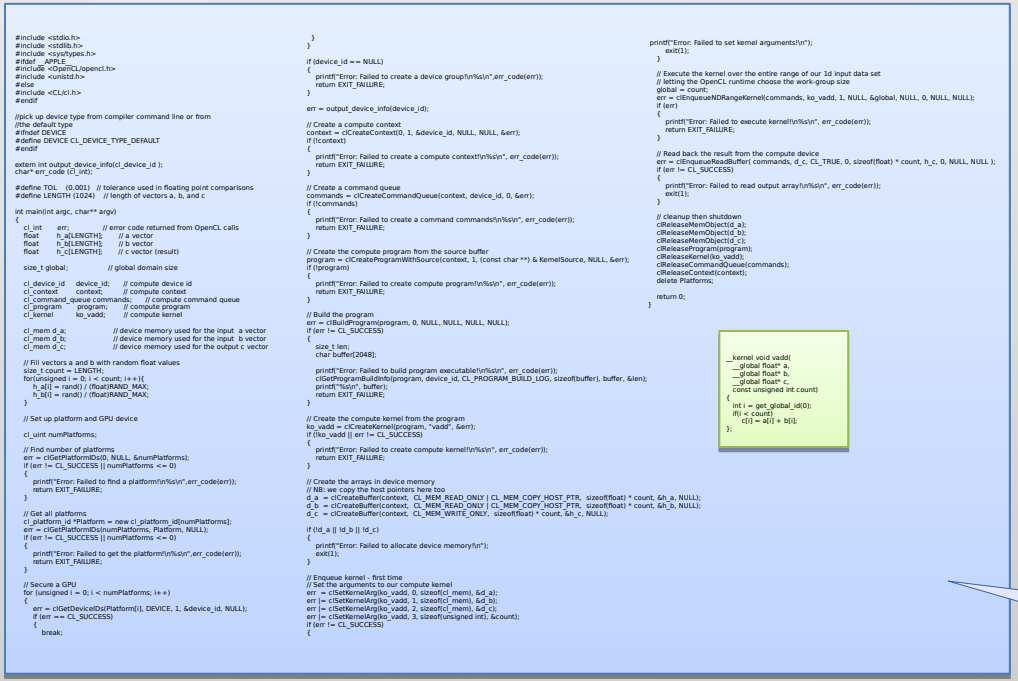
- Applications.
  - Example: Vector addition.
- Template libraries:
  - Example: Parallel STL.
- Embed DSLs:
  - Example: Image processing DSL.

# SYCL Example: Vector Addition

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

template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data(), range<1>(inputA.size()));
  buffer<T, 1> inputBBuf(inputB.data(), range<1>(inputB.size()));
  buffer<T, 1> outputBuf(output.data(), range<1>(output.size()));
  queue defaultQueue;
  defaultQueue.submit([&] (handler &cgh) {
    auto inputAPtr = inputABuf.template get_access<access::mode::read>(cgh);
    auto inputBPtr = inputBBuf.template get_access<access::mode::read>(cgh);
    auto outputPtr = outputBuf.template get_access<access::mode::write>(cgh);
    cgh.parallel_for< class vadd<T> >(range<1>(output.size()), [=](id<1> idx)  {
      outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    });
  });
}
```

This is the bit that runs in parallel.

# Comparison with OpenCL



```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#include <unistd.h>
#else
#include <CL/cl.h>
#endif

//pick up device type from compiler command line or from
//the default type
#ifndef DEVICE
#define DEVICE CL_DEVICE_TYPE_DEFAULT
#endif

extern int output_device_info(cl_device_id );
char* err_code (cl_int);

#define TOL    (0.001)   // tolerance used in floating point comparisons
#define LENGTH (1024)    // length of vectors a, b, and c

int main(int argc, char** argv)
{
    cl_int      err;               // error code returned from OpenCL calls
    float       h_a[LENGTH];       // a vector
    float       h_b[LENGTH];       // b vector
    float       h_c[LENGTH];       // c vector (result)

    size_t global;                 // global domain size

    cl_device_id    device_id;     // compute device id
    cl_context      context;       // compute context
    cl_command_queue commands;     // compute command queue
    cl_program      program;       // compute program
    cl_kernel       ko_vadd;       // compute kernel

    cl_mem d_a;                    // device memory used for the input  a vector
    cl_mem d_b;                    // device memory used for the input  b vector
    cl_mem d_c;                    // device memory used for the output c vector

    // Fill vectors a and b with random float values
    size_t count = LENGTH;
    for(unsigned i = 0; i < count; i++){
        h_a[i] = rand() / (float)RAND_MAX;
        h_b[i] = rand() / (float)RAND_MAX;
    }

    // Set up platform and GPU device

    cl_uint numPlatforms;

    // Find number of platforms
    err = clGetPlatformIDs(0, NULL, &numPlatforms);
    if (err != CL_SUCCESS || numPlatforms <= 0)
    {
        printf("Error: Failed to find a platform!\n%s\n",err_code(err));
        return EXIT_FAILURE;
    }

    // Get all platforms
    cl_platform_id *Platform = new cl_platform_id[numPlatforms];
    err = clGetPlatformIDs(numPlatforms, Platform, NULL);
    if (err != CL_SUCCESS || numPlatforms <= 0)
    {
        printf("Error: Failed to get the platform!\n%s\n",err_code(err));
        return EXIT_FAILURE;
    }

    // Secure a GPU
    for (unsigned i = 0; i < numPlatforms; i++)
    {
        err = clGetDeviceIDs(Platform[i], DEVICE, 1, &device_id, NULL);
        if (err == CL_SUCCESS)
        {
            break;
        }
    }

    if (device_id == NULL)
    {
        printf("Error: Failed to create a device group!\n%s\n",err_code(err));
        return EXIT_FAILURE;
    }

    err = output_device_info(device_id);

    // Create a compute context
    context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
    if (!context)
    {
        printf("Error: Failed to create a compute context!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Create a command queue
    commands = clCreateCommandQueue(context, device_id, 0, &err);
    if (!commands)
    {
        printf("Error: Failed to create a command commands!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Create the compute program from the source buffer
    program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL, &err);
    if (!program)
    {
        printf("Error: Failed to create compute program!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Build the program
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    if (err != CL_SUCCESS)
    {
        size_t len;
        char buffer[2048];

        printf("Error: Failed to build program executable!\n%s\n", err_code(err));
        clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
        printf("%s\n", buffer);
        return EXIT_FAILURE;
    }

    // Create the compute kernel from the program
    ko_vadd = clCreateKernel(program, "vadd", &err);
    if (!ko_vadd || err != CL_SUCCESS)
    {
        printf("Error: Failed to create compute kernel!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Create the arrays in device memory
    // NB: we copy the host pointers here too
    d_a  = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  sizeof(float) * count, &h_a, NULL);
    d_b  = clCreateBuffer(context,  CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  sizeof(float) * count, &h_b, NULL);
    d_c  = clCreateBuffer(context,  CL_MEM_WRITE_ONLY,  sizeof(float) * count, &h_c, NULL);

    if (!d_a || !d_b || !d_c)
    {
        printf("Error: Failed to allocate device memory!\n");
        exit(1);
    }

    // Enqueue kernel - first time
    // Set the arguments to our compute kernel
    err  = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
    err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
    err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
    err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to set kernel arguments!\n");
        exit(1);
    }

    // Execute the kernel over the entire range of our 1d input data set
    // letting the OpenCL runtime choose the work-group size
    global = count;
    err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
    if (err)
    {
        printf("Error: Failed to execute kernel!\n%s\n", err_code(err));
        return EXIT_FAILURE;
    }

    // Read back the result from the compute device
    err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0, sizeof(float) * count, h_c, 0, NULL, NULL );
    if (err != CL_SUCCESS)
    {
        printf("Error: Failed to read output array!\n%s\n", err_code(err));
        exit(1);
    }

    // cleanup then shutdown
    clReleaseMemObject(d_a);
    clReleaseMemObject(d_b);
    clReleaseMemObject(d_c);
    clReleaseProgram(program);
    clReleaseKernel(ko_vadd);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);
    delete Platforms;

    return 0;
}
```

```
__kernel void vadd(
  __global float* a,
  __global float* b,
  __global float* c,
  const unsigned int count)
{
  int i = get_global_id(0);
  if(i < count)
    c[i] = a[i] + b[i];
};
```

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

int main()
{
  int count = 1024;

  std::vector<float> vecA(count) = { /* input */ };
  std::vector<float> vecB(count) = { /* input */ };
  std::vector<float> vecC(count) = { /* output */ };

  {
    buffer<float, 1> bufA(vecA.data(), vecA.size());
    buffer<float, 1> bufB(vecB.data(), vecB.size());
    buffer<float, 1> bufC(vecC.data(), vecC.size());

    queue myQueue;

    command_group(myQueue, [&] ()
    {
      auto ptrA = bufA.get_access<access::read>();
      auto ptrB = bufB.get_access<access::read>();
      auto ptrC = bufC.get_access<access::write>();

      parallel_for(range<1>(count), kernel_functor<class vector_add>([=](id<1> idx)
      {
        ptrC[idx] = ptrA[idx] + ptrB[idx];
      }));
    });
  }

  return 0;
}
```

SYCL for OpenCL

Traditional OpenCL

```
template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
```

```
}
```

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

template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T>
```

The SYCL runtime is in sycl.hpp and within the cl::sycl namespace

```
}
```

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

template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data(), range<1>(inputA.size()));
  buffer<T, 1> inputBBuf(inputB.data(), range<1>(inputB.size()));
  buffer<T, 1> outputBuf(output.data(), range<1>(output.size()));
```

Construct three SYCL buffers and initialise them with the data from the std::vectors.

Data is synchronised by RAII

```
}
```

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

template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data(), range<1>(inputA.size()));
  buffer<T, 1> inputBBuf(inputB.data(), range<1>(inputB.size()));
  buffer<T, 1> outputBuf(output.data(), range<1>(output.size()));
  queue defaultQueue;
```

Construct a SYCL queue to execute work on a device.

```
}
```

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

template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data(), range<1>(inputA.size()));
  buffer<T, 1> inputBBuf(inputB.data(), range<1>(inputB.size()));
  buffer<T, 1> outputBuf(output.data(), range<1>(output.size()));
  queue defaultQueue;
  defaultQueue.submit([&] (handler &cgh) {
  });
}
```

The command_group is en-queued asynchronously and is thread safe.

Construct a SYCL command_group to define the work to be en-queued on a device.

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

template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data(), range<1>(inputA.size()));
  buffer<T, 1> inputBBuf(inputB.data(), range<1>(inputB.size()));
  buffer<T, 1> outputBuf(output.data(), range<1>(output.size()));
  queue defaultQueue;
  defaultQueue.submit([&] (handler &cgh) {
    auto inputAPtr = inputABuf.template get_access<access::mode::read>(cgh);
    auto inputBPtr = inputBBuf.template get_access<access::mode::read>(cgh);
    auto outputPtr = outputBuf.template get_access<access::mode::write>(cgh);



  });
}
```

The SYCL runtime uses accessors to track dependencies across command_groups.

Construct three SYCL accessors with access modes, to give the device access to the data.

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

template <typename T>
void parallel_vadd(std::vector<T> &in       std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data     nge<1>(inputA.size()));
  buffer<T, 1> inputBBuf(inputB.data     range<1>(inputB.size()));
  buffer<T, 1> outputBuf(output.dat      range<1>(output.size()));
  queue defaultQueue;
  defaultQueue.submit([&] (handle     &cgh) {
    auto inputAPtr = inputABuf.t    plate get_access<access::mode::rea      );
    auto inputBPtr = inputBBuf.     mplate get_access<access::mode::r      (cgh);
    auto outputPtr = outputBuf      template get_access<access::mode:      write>(cgh);
    cgh.parallel_for< class vadd<T> >(range<1>(output.size()), [=](id<1> idx)  {

    });
  });
}
```

The typename 'vadd' is used to name the lambda.

Call parallel_for() to execute a kernel function.

The range provided to the parallel_for() should match the size of the data buffers.

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

template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data(), range<1>(inputA.size()));
  buffer<T, 1> inputBBuf(inputB.data(), range<1>(inputB.size()));
  buffer<T, 1> outputBuf(output.data(), range<1>(output.size()));
  queue defaultQueue;
  defaultQueue.submit([&] (handler &cgh) {
    auto inputAPtr = inputABuf.template get_access<access::mo
    auto inputBPtr = inputBBuf.template get_access<access::mode:          (cgh);
    auto outputPtr = outputBuf.template get_access<access::m      .write>(cgh);
    cgh.parallel_for< class vadd<T> >(range<1>(output.siz   ()), [=](id<1> idx)  {
      outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    });
  });
}
```

The body of the lambda expression is what is compiled into an OpenCL kernel by the SYCL device compiler.

Use the subscript operator on the accessors to read and write the data.

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

template <typename T>
void parallel_vadd(std::vector<T> &inputA, std::vector<T> &inputB, std::vector<T> &output) {
  buffer<T, 1> inputABuf(inputA.data(), range<1>(inputA.size()));
  buffer<T, 1> inputBBuf(inputB.data(), range<1>(inputB.size()));
  buffer<T, 1> outputBuf(output.data(), range<1>(output.size()));
  queue defaultQueue;
  defaultQueue.submit([&] (handler &cgh) {
    auto inputAPtr = inputABuf.template get_access<access::mode::read>(cgh);
    auto inputBPtr = inputBBuf.template get_access<access::mode::read>(cgh);
    auto outputPtr = outputBuf.template get_access<access::mode::write>(cgh);
    cgh.parallel_for< class vadd<T> >(range<1>(output.size()), [=](id<1> idx)  {
      outputPtr[idx] = inputAPtr[idx] + inputBPtr[idx];
    });
  });
}
```

Template the function

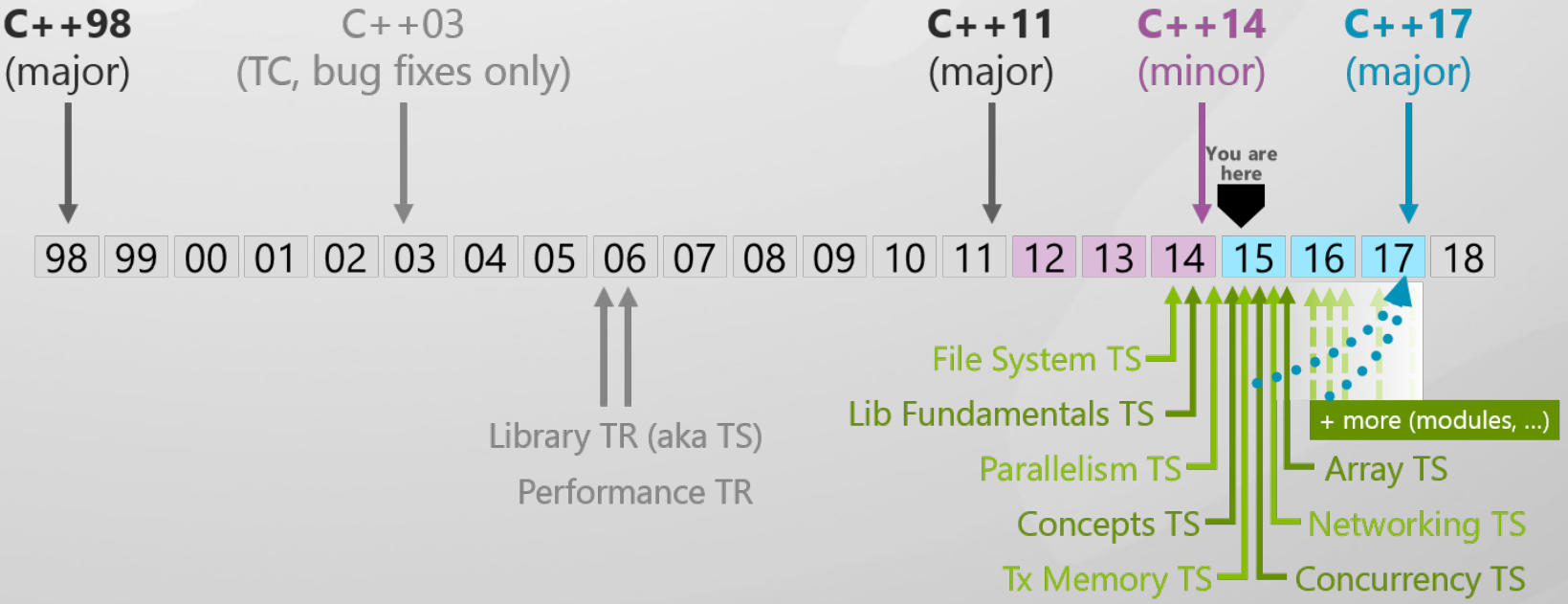
# Hierarchical Parallelism

```
void hierarchical(/* … */) {
  // … Setup buffers here …

  queue defaultQueue;
  defaultQueue.submit([&] (handler &cgh) {
    // … Setup accessors here …

    cgh.parallel_for_work_group< class k >(range<2>(/* … */), range<2>(/* … */),
                                           [=](group<2> groupID) {
      parallel_for_work_item(groupID, [&](item<2> itemID) {
        // … Do something …
      });




    });
  });
}
```

```
void hierarchical(/* … */) {
  // … Setup buffers here …

  queue defaultQueue;
  defaultQueue.submit([&] (handler &cgh) {
    // … Setup accessors here …

    cgh.parallel_for_work_group< class k >(range<2>(/* … */), range<2>(/* … */),
                                           [=](group<2> groupID) {
      parallel_for_work_item(groupID, [&](item<2> itemID) {
        // … Do something …
      });

      parallel_for_work_item(groupID, [&](item<2> itemID) {
        // … Do something else …
      });
    });
  });
}
```

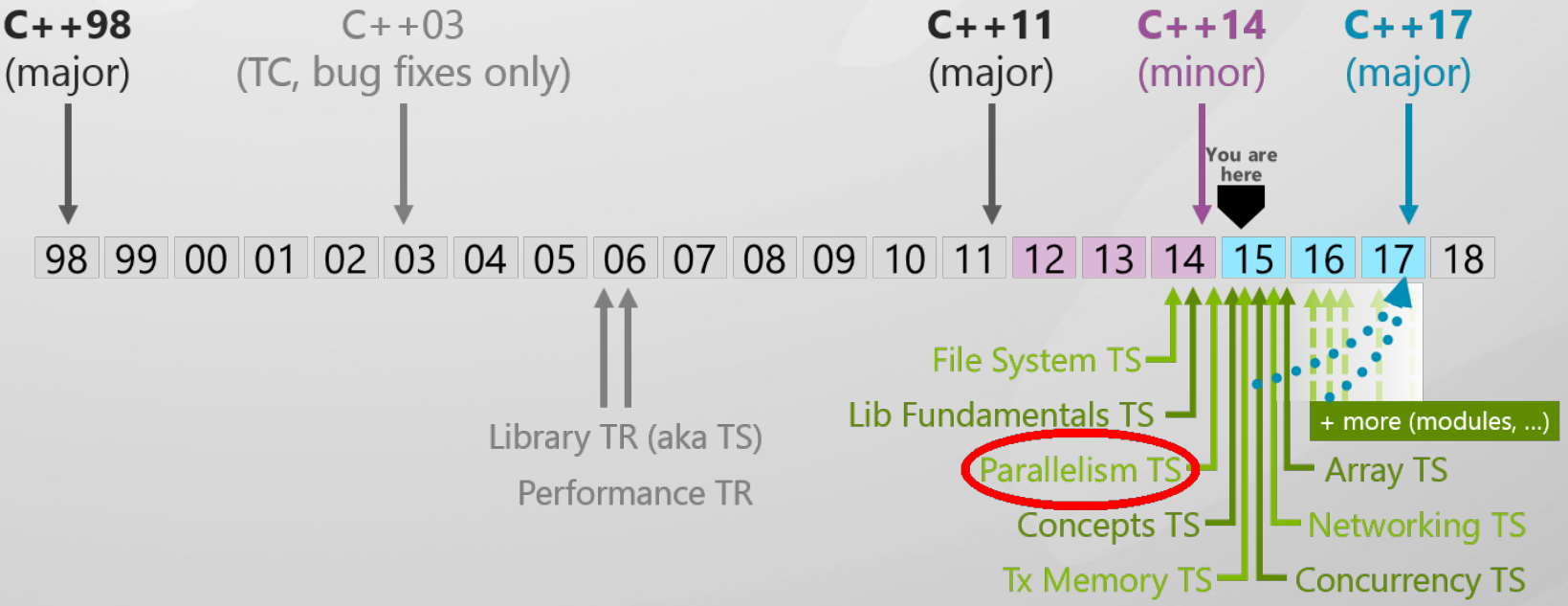
# SYCL Libraries: Parallel STL

# C++ Roadmap




https://isocpp.org/std/status

# C++ Roadmap

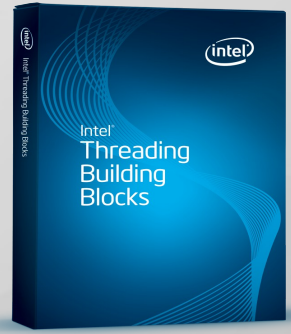



https://isocpp.org/std/status

# Existing Parallel Libraries

- Each vendor has its own parallel library.
- Interfaces resemble STL, but different.
- Results in platform specific code!

BOLT




Thrust

# N4409 Proposal: Parallel STL

- Written by engineers from Intel, Microsoft, and NVIDIA.
  - Based on TBB (Intel), C++AMP/PPL (Microsoft), and Thrust (NVIDIA).
- Describes an interface to:
  - Algorithms with parallel implementations.
  - Perform parallel operations on generic containers.
- Extends the current STL interface with parallel execution policies.

# Sorting in Parallel STL

```
// Current C++11 sequential sort.
std::sort(data.begin(), data.end());

// C++17 explicitly sequential sort.
using namespace std::experimental::parallel;
std::sort(seq, data.begin(), data.end());

// C++17 parallel sort.
// Still using namespace std::experimental::parallel;
std::sort(par, data.begin(), data.end());
```

# Sorting in Parallel STL

Sequential execution policy.

```
// Current C++11 sequential sort.
std::sort(data.begin(), data.end());

// C++17 explicitly sequential sort.
using namespace std::experimental::parallel;
std::sort(seq, data.begin(), data.end());

// C++17 parallel sort.
// Still using namespace std::experimental::parallel;
std::sort(par, data.begin(), data.end());
```

# Sorting in Parallel STL

```
// Current C++11 sequential sort.
std::sort(data.begin(), data.end());

// C++17 explicitly sequential sort.
using namespace std::experimental::parallel;
std::sort(seq, data.begin(), data.end());

// C++17 parallel sort.
// Still using namespace std::experimental::parallel;
std::sort(par, data.begin(), data.end());
```

Parallel execution policy.

# Generic Algorithms in Parallel STL

```
// C++17 parallel loop.
using namespace std::experimental::parallel;
std::for_each(par, data.begin(), data.end(),
    [](float &i) { /* Do something. */ });

// C++17 parallel transform.
// Still using namespace std::experimental::parallel;
std::transform(par, data.begin(), data.end(),
    [](float &i) { /* Modify i. */ });
```

# Codeplay SYCL Parallel STL

- Khronos Open Source License.
- Available on Github:
  - https://github.com/KhronosGroup/SyclParallelSTL
- Current basic implementation:
  - Execution policy mechanism in place.
  - **`sort`** (*bitonic* if size is power of 2, *sequential* on gpu otherwise).
  - Parallel **`transform`**.
  - Parallel **`for_each`**.

# Codeplay SYCL Parallel STL

```
// SYCL STL sort.
std::vector<int> data { /* Setup your data. */ };
sycl::sort(sycl_policy, data.begin(), data.end());
```

# Codeplay SYCL Parallel STL

```
// SYCL STL sort.
std::vector<int> data { /* Setup your data. */ };
sycl::sort(sycl_policy, data.begin(), data.end());
```

```
template<typename KernelName = DefaultKernelName>
class sycl_execution_policy {
  public:
    using kernelName = KernelName;

    sycl_execution_policy() = default;
    sycl_execution_policy(cl::sycl::queue q);
    cl::sycl::queue get_queue() const;
};
```

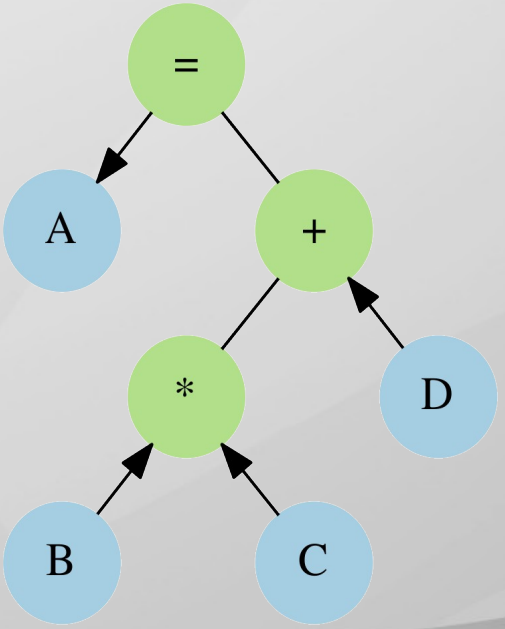
# Embedding DSLs: Image Processing

# Image Processing DSL

- Is SYCL **expressive** enough to allow us to build higher level programming models?
- Can we define a way to **efficiently** compose kernels from simple primitives?

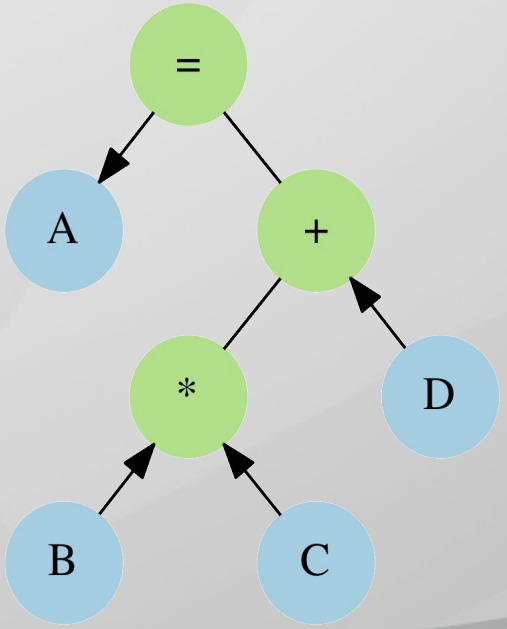
# FMA

$$A = B \times C + D$$



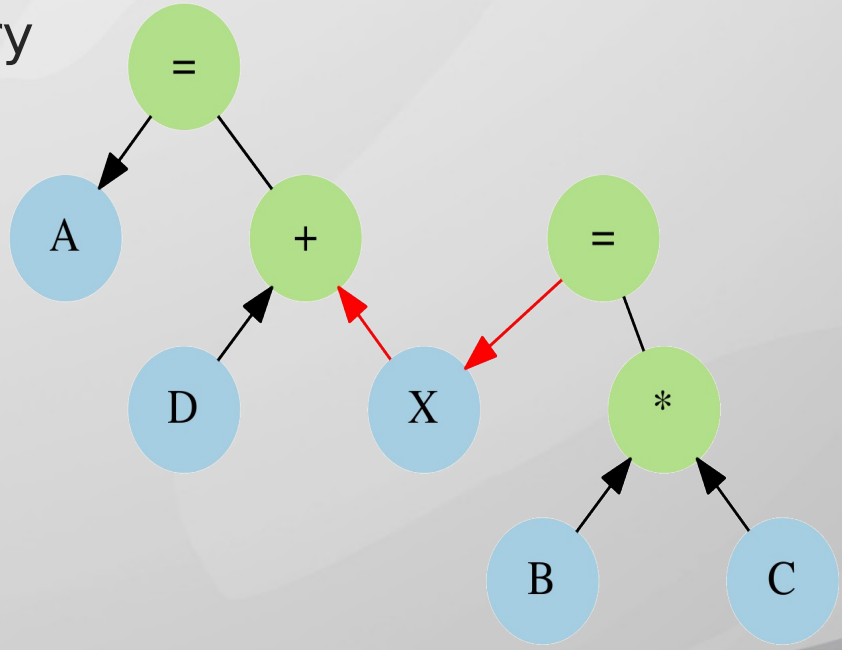
# FMA: OpenCL

```
__kernel
void fma (__global float* A,
          __global float* B,
          __global float* C,
          __global float* D) {
  int i = get_global_id(0);
  A[i] = B[i] * C[i] + D[i];
}
```



# FMA: OpenCLiPP

- OpenCLiPP: Parallel primitive library for image processing in OpenCL.
- No FMA primitive.
- Requires an extra store/load to global memory.




```
Image A, B, C, D;
TmpImage X;
arith->Mul(B, C, X);
arith->Add(D, X, A);
```

# FMA: Runtime Construction

- Fuse FMA through runtime kernel construction.
  - Halide.
  - ArrayFire.
- Requires significant compiler machinery to be implemented by library author.

# FMA: DSEL in SYCL

- Use C++ templates to define an embedded DSL.
- Use C++ type system to capture expression syntax at compile time.
  - Boost Proto.
- Implement DSEL using SYCL.
  - SYCL already has a device compiler.

# FMA: DSEL in SYCL

```
// Declare some SYCL images.
cl::sycl::image A, B, C, D;

// Declare our expression.
auto expr = ( _1 = _2 * _3 + _4 );

// Substitute arguments and evaluate.
dsl::eval(queue, range, expr, A, B, C, D);
```

# FMA: DS

```
expr<tag::assign,
    expr<tag::terminal, term<placeholder<0>>,
    expr<tag::addition,
        expr<tag::multiply,
            expr<tag::terminal, term<placeholder<1>>,
            expr<tag::terminal, term<placeholder<2>>
        >,
        expr<tag::terminal, term<placeholder<3>>
    >
>
```

```
// Declare some SYCL images
cl::sycl::image A, B, C

// Declare our expression.
auto expr = ( _1 = _2 * _3 + _4 );

// Substitute arguments and evaluate.
dsl::eval(queue, range, expr, A, B, C, D);
```

# FMA: DSEL in SYCL

```
// Declare some SYCL images.
cl::sycl::image A, B, C, D;

// Declare our expression.
auto expr = ( _1 = _2 * _3 +

// Substitute arguments and evaluate.
dsl::eval(queue, range, expr, A, B, C, D);
```

```
cgh.parallel_for< decltype(expr) >(range,
    [=](cl::sycl::item<2> item) {
    // Construct a context per thread.
    // Provides implementations of operators.
    dsl::context ctx{item};

    // Evaluate expression tree per work item.
    boost::proto::eval(expr, ctx);
});
```

# Unsharp Mask

http://commons.wikimedia.org/wiki/File:Accutance_example.png

# Unsharp Mask

$$I_{out} = I_{\text{in}} + (I_{\text{in}} - G(I_{\text{in}})) \times w$$

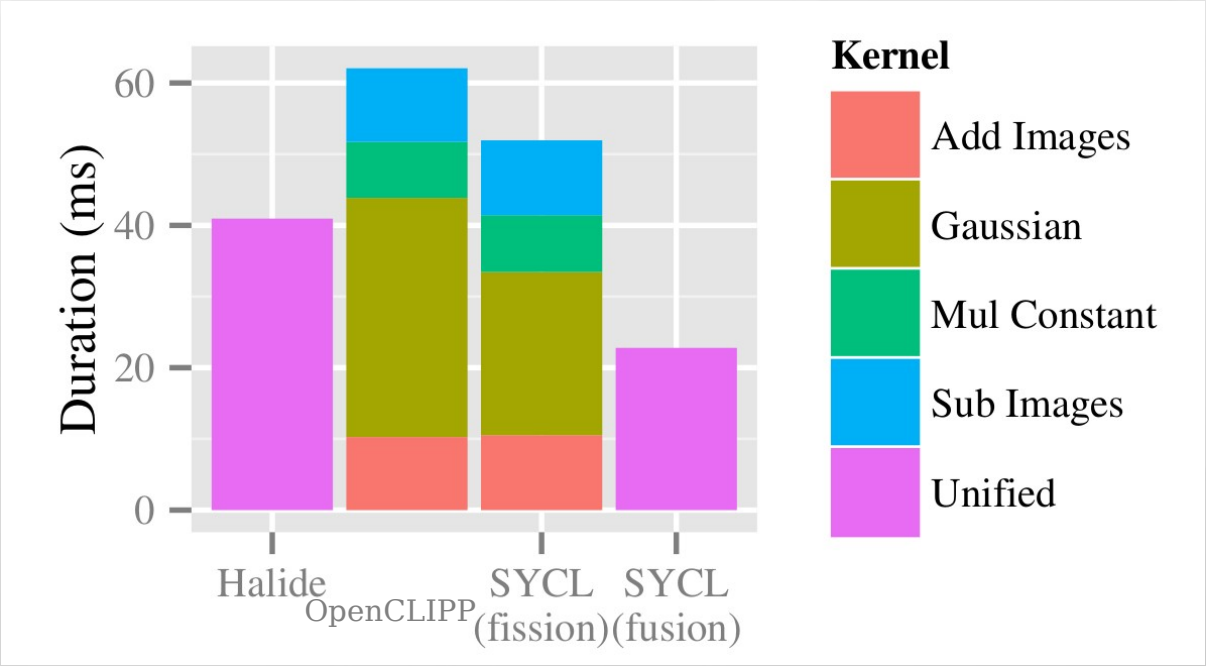
- Where:
  - $I_{\text{in}}$ is the input image.
  - $G(I_{\text{in}})$ is a Gaussian blur on input image.
  - $w$ is the scalar weight of the mask.

# Unsharp Mask

$$I_{out} = I_{\text{in}} + \left(I_{\text{in}} - G\left(I_{\text{in}}\right)\right) \times w$$

```
// Declare operands.
cl::sycl::image<2> in = ...;
cl::sycl::image<2> out;
float w = ...;

// Declare unsharp mask expression.
auto expr = (_1 = _2 + (_2 - gaussian(_2)) * w);

// Evaluate unsharp mask.
dsl::eval(queue, range,
          expr,
          out, in);
```

# Unsharp Mask




Codeplay's SYCL implementation, on top of AMD's OpenCL implementation. Hardware: GPU within an AMD A10-7850K.

# Summary

# What to know more?

- Specification:
  - https://www.khronos.org/opencl/sycl
- Exercises:
  - https://github.com/codeplaysoftware/sycl-exercises
- SYCL Parallel STL:
  - https://github.com/KhronosGroup/SyclParallelSTL

# Conclusion

- SYCL brings:
  - Modern C++.
  - OpenCL performance.
  - Standardised specification.
- Reduces the difficultly of implementing complex parallel software.

# Thank you