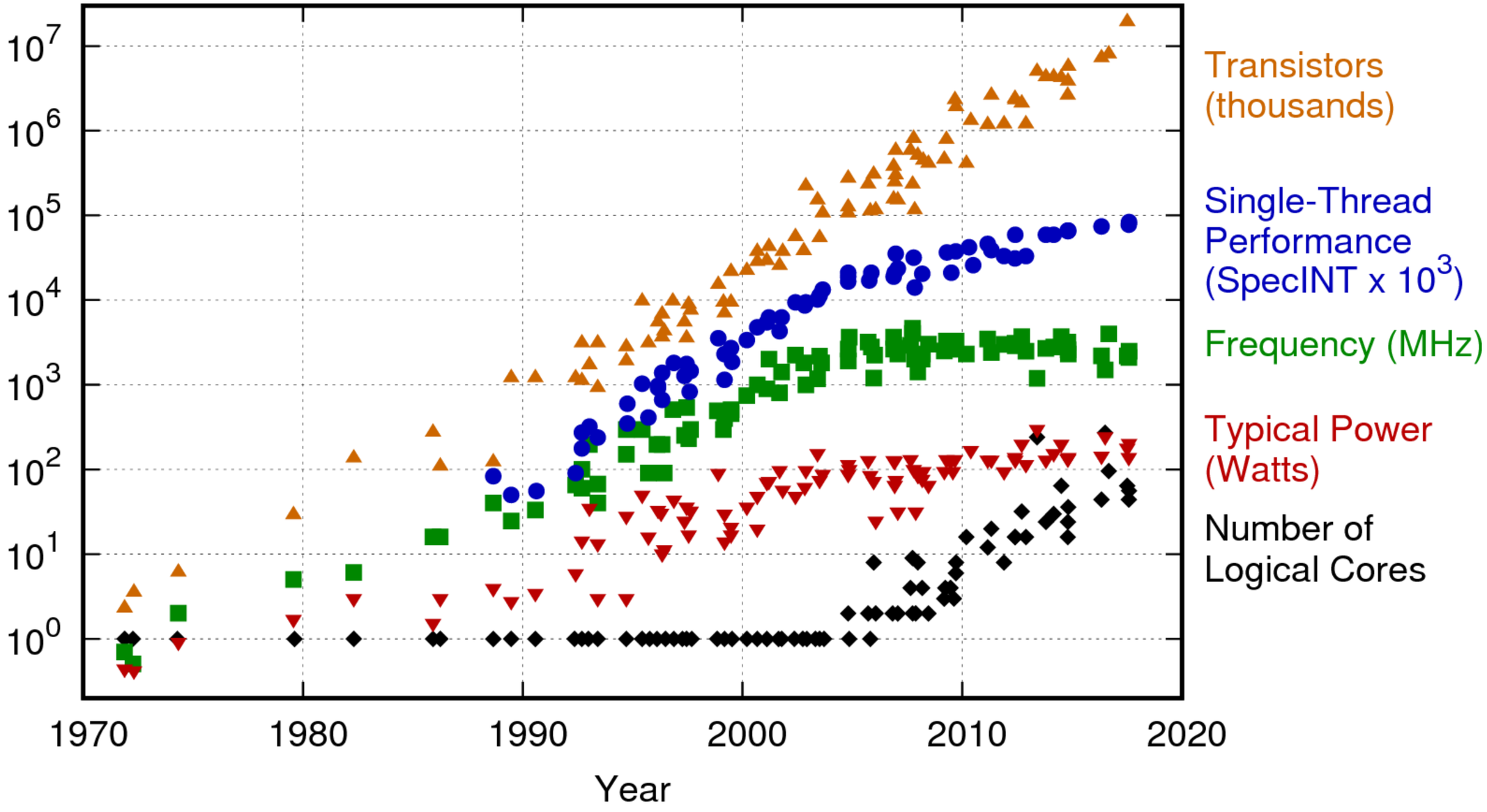
Look ma, no CUDA!

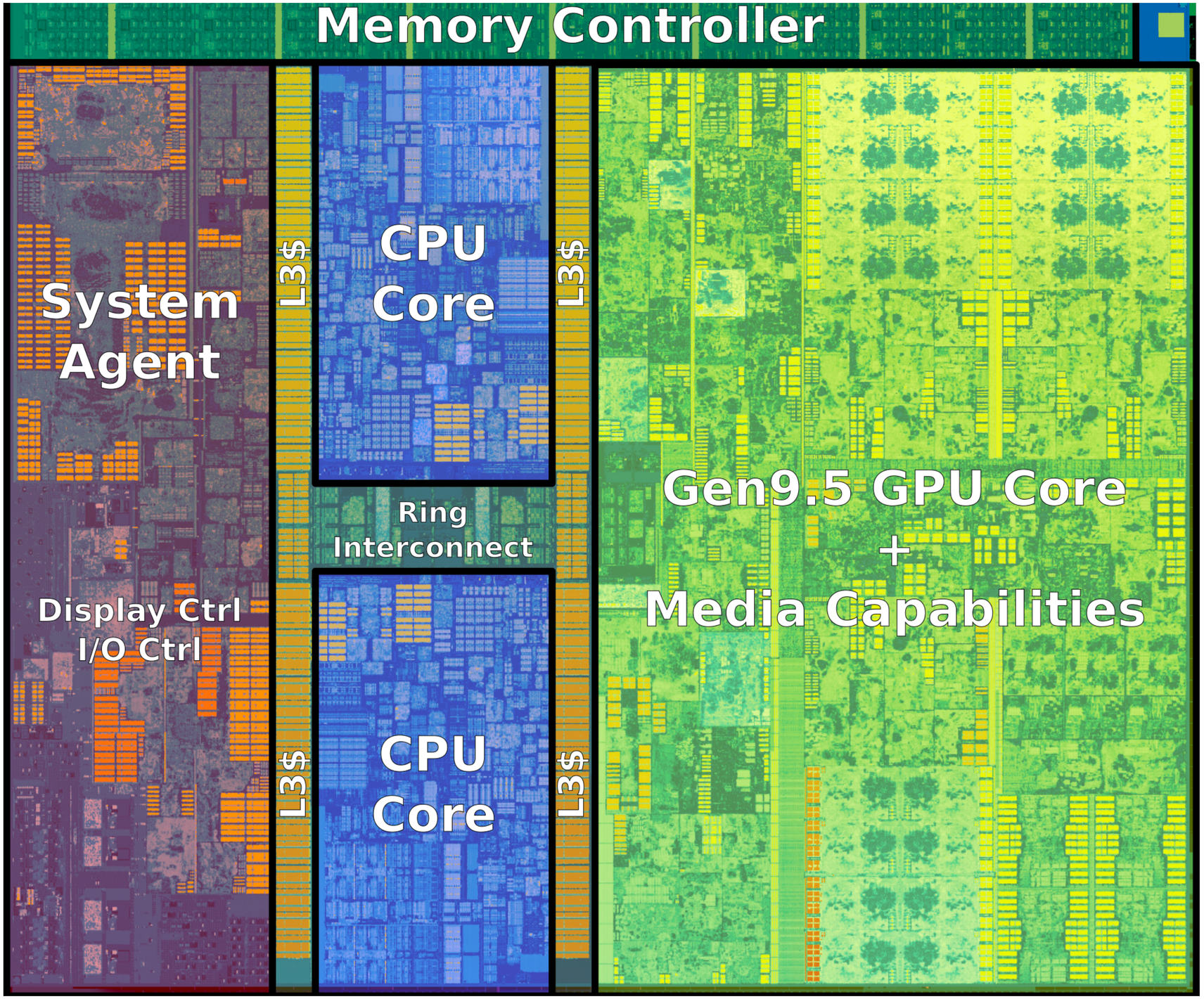
Programming GPUs with modern C++
(finally)

Federico Ficarelli @ Meetup C++ 2019-05-09

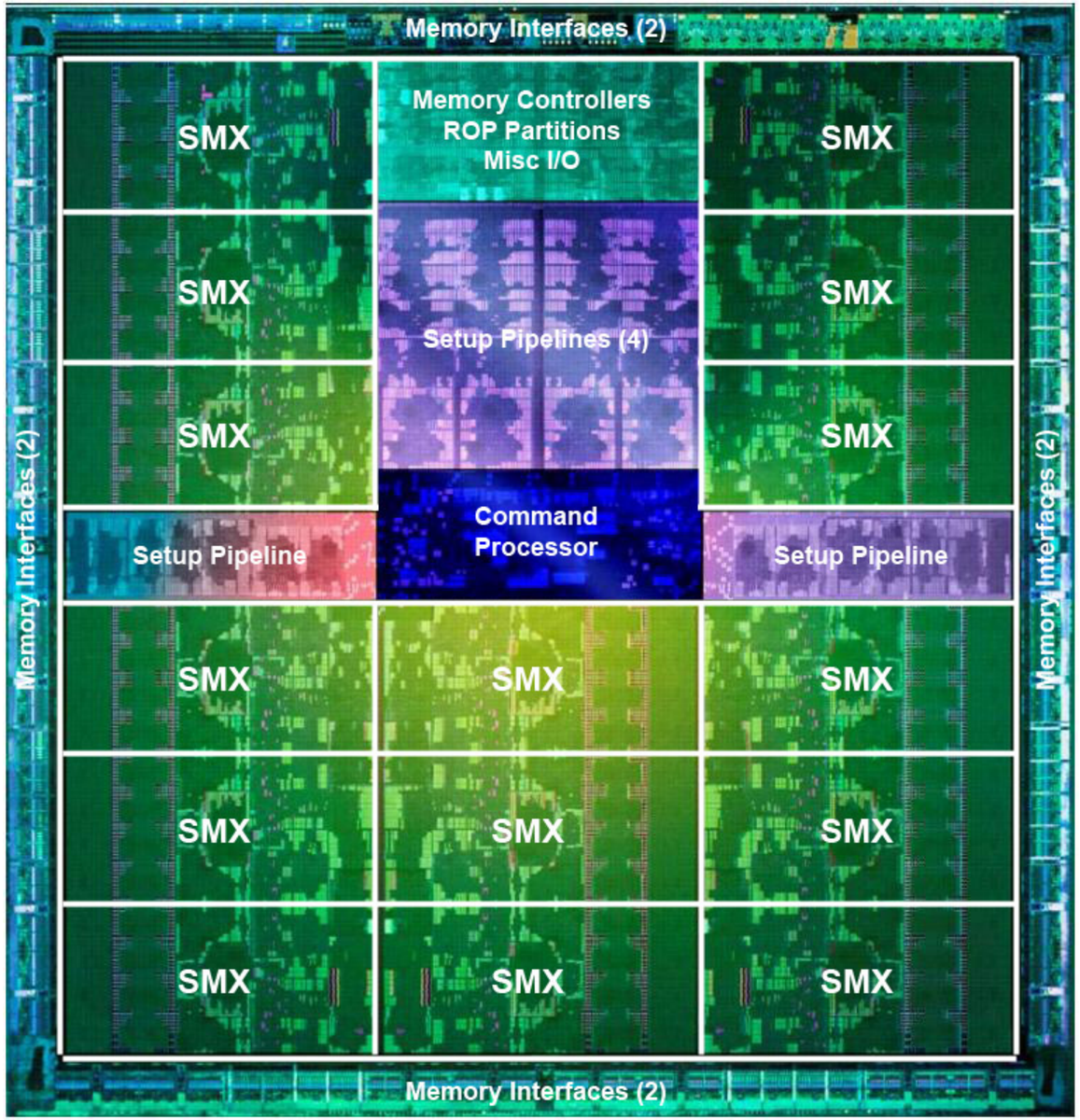
42 Years of Microprocessor Trend Data



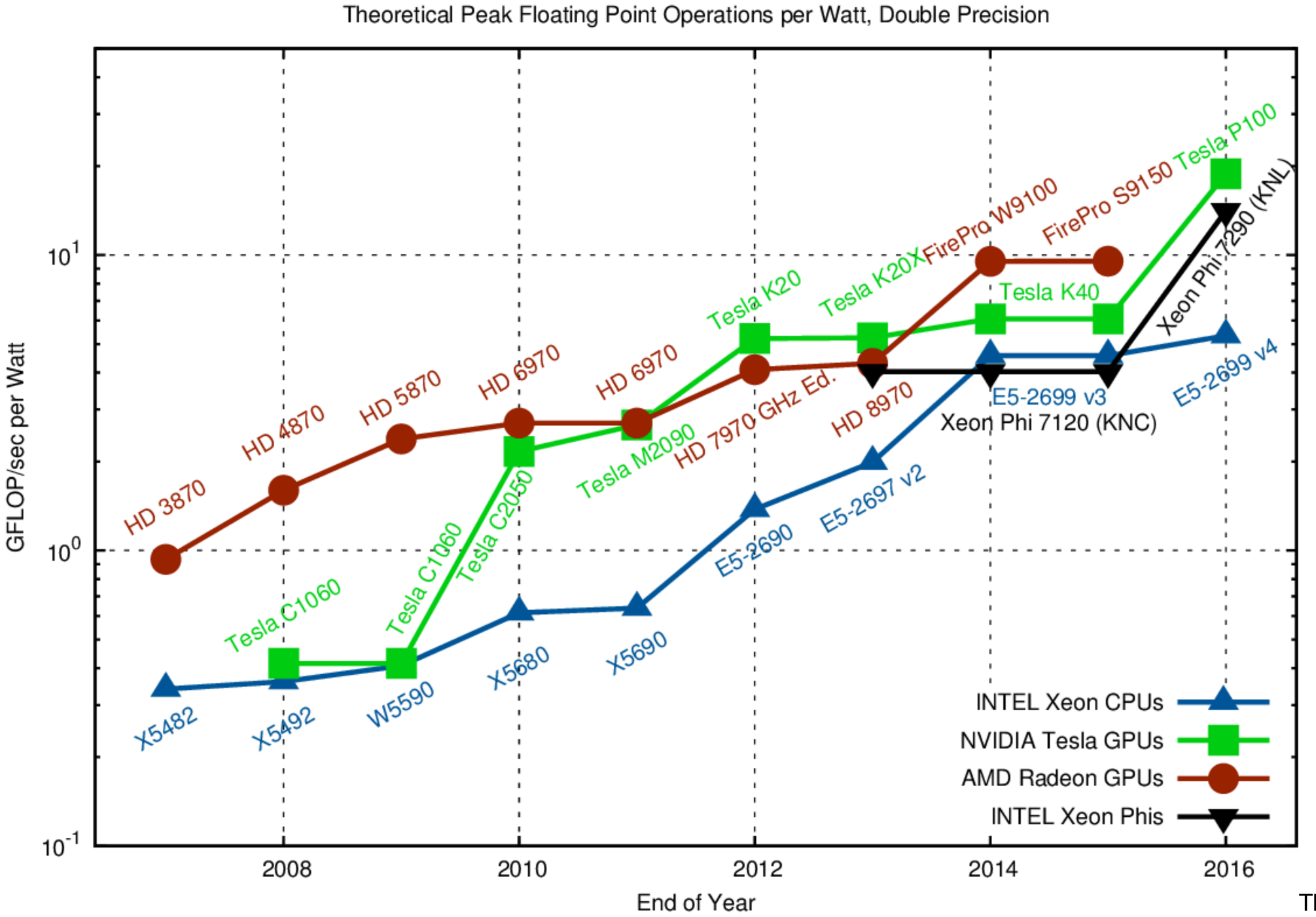
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp



Intel Kaby Lake GT2, gen 9.5 µ-arch, 14 nm



NVidia Kepler GK110-430 (2014), 2880 CUDA cores, 28nm



Thanks to Karl Rupp

CPU vs GPU

- Task-based parallelism
- Small number of large and complex cores
- Each core executes independently
- Lower efficiency
- Random* memory access

- Data-based parallelism
- large number of execution units
- Single instruction on multiple execution units
- High efficiency
- Sequential* memory access

Parallelism Abstractions

Resource	C++ Support
Cores	C++11/14/17 threads, async, OpenMP
Hardware Threads	C++11/14/17 threads, async, OpenMP
Vectors	C++20 std::simd, OpenMP
Parallel Loops	async, `tbb::parallel_invoke`, C++17 parallel algorithms
Heterogeneous Offload	OpenCL, SYCL, OpenMP, OpenACC, Kokkos, Raja, HPX
Distributed	HPX, MPI
Caches	C++17 `std::hardware_constructive_interference_size`, `std::hardware_destructive_interference_size`
NUMA	(C++23?) Executors, Affinity

Programming GPUs

How to program GPUs?

Single Instruction Single Data (SISD)

```
void calc(int *in, int *out) {
  for (int i = 0; i < 1024; i++) {
    out[i] = in[i] * in[i];
  }
}

calc(in, out);
```

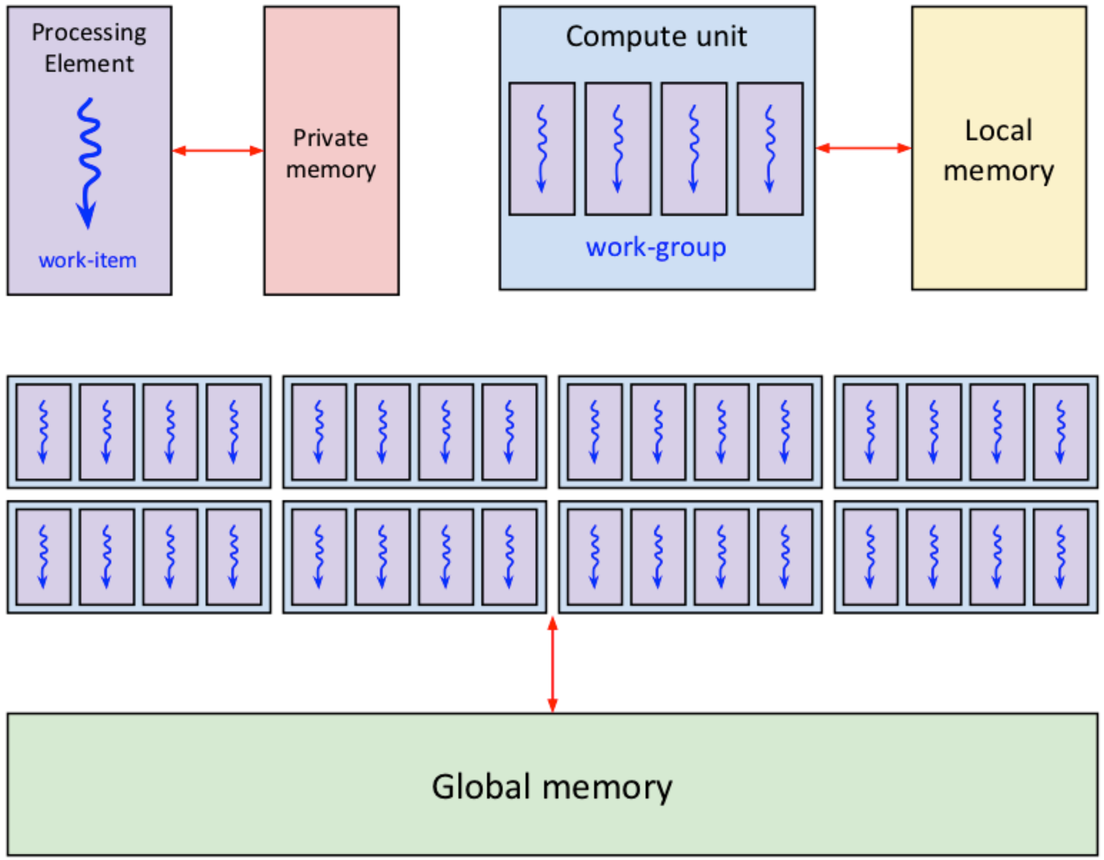
Single Instruction Multiple Data (SIMD)

```
void calc(int *in, int *out) {
  #pragma omp simd
  for (int i = 0; i < 1024; i++) {
    out[i] = in[i] * in[i];
  }
}

calc(in, out);
```

Single Program Multiple Data (SPMD)

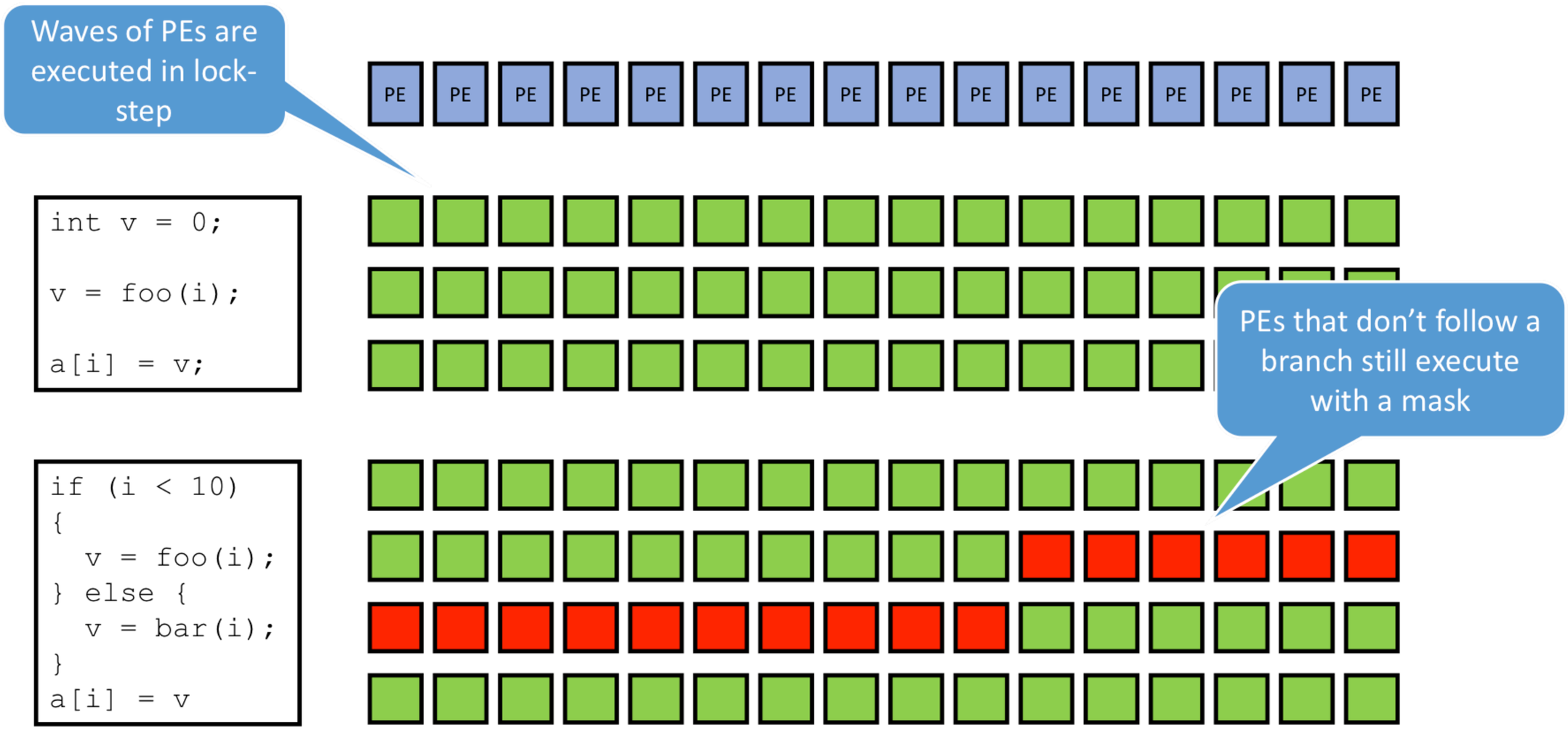
```
void calc(int *in, int *out, int id) {
  out[id] = in[id] * in[id];
}

/* specify degree of parallelism */
parallel_for(calc, in, out, 1024);
```

GPU Programming Model



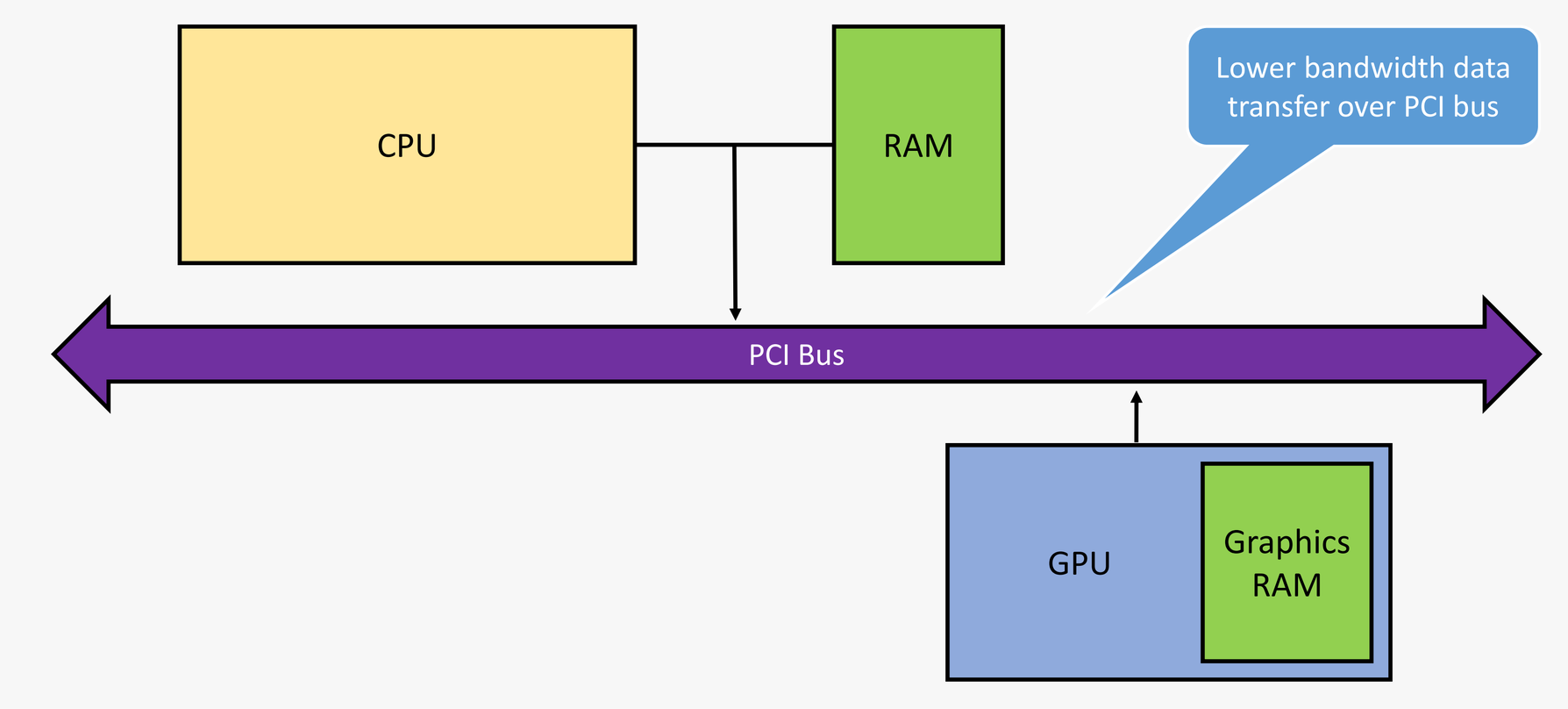
Thanks to Michael Wong/Codeplay

Lockstep execution



Thanks to Michael Wong/Codeplay

Common system architecture



Thanks to Gordon Brown/Codeplay

```cpp
std::vector<int> add(const std::vector<int>& a, const std::vector<int>& b) {
    // We are going to operate on the common indexes subset
    std::vector<int> result(std::min(a.size(), b.size()));
    if (result.size() == 0) return {};
    // No std::size(), CUDA 10 <= C++14
    const auto byte_size = result.size() * sizeof(int);

    int* device_a = nullptr;
    int* device_b = nullptr;
    int* device_result = nullptr;

    // Device allocation
    cudaMalloc(&device_a, byte_size);
    cudaMalloc(&device_b, byte_size);
    cudaMalloc(&device_result, byte_size);

    // Copy host vectors to device
    cudaMemcpy(device_a, a.data(), byte_size, cudaMemcpyHostToDevice);
    cudaMemcpy(device_b, b.data(), byte_size, cudaMemcpyHostToDevice);

    // Number of threads in each thread block
    const auto blockSize = 1024;

    // Number of thread blocks in grid
    const auto gridSize = static_cast<int>(
        std::ceil(static_cast<float>(result.size()) / blockSize)
    );

    // Execute the kernel
    add_kernel<<<gridSize, blockSize>>>(device_a, device_b,
                                        device_result, result.size());

    // Copy result back to host
    cudaMemcpy(result.data(), device_result, result.size() * sizeof(int),
               cudaMemcpyDeviceToHost);

    cudaFree(device_a);
    cudaFree(device_b);
    cudaFree(device_result);

    return result;
}
```

```cpp
__global__ void add_kernel(const int* a, const int* b,
                           int* result, int count) {
    // Get our global thread ID
    int id = blockIdx.x * blockDim.x + threadIdx.x;

    // Check for out of bounds threads
    if (id < count) {
        result[id] = a[id] + b[id];
    }
}
```

Hello CUDA: Vector Add

GPU Programming Tips

- **Ensure the task is suitable**
  - GPUs are most efficient for data parallel tasks
  - Performance gain from performing computation > cost of moving data
- **Avoid branching**
  - Waves of processing elements execute in lock-step
  - Both sides of branches execute with the other masked
- **Avoid non-coalesced memory access**
  - GPUs access memory more efficiently if accessed as contiguous blocks
- **Avoid expensive data movement**
  - The bottleneck in GPU programming is data movement between CPU and GPU memory
  - It’s important to have data as close to the processing as possible

SYCL

SYCL is a **royalty-free standard** from Khronos Group, **cross-platform abstraction layer** that builds on the underlying concepts, **portability** and **efficiency** of OpenCL that enables code for heterogeneous processors to be written in a **single-source** style using completely **standard C++**.

SYCL: Motivations

- **Make heterogeneous programming more accessible**
  - Provide a foundation for efficient and portable template algorithms
- Define an **open portable standard**
- Provide the **performance** of competitors
- Add **portability** across heterogeneous environments
- Based only on **standard C++**
- Provide a high-level **shared source** model
- Provide a **high-level abstraction**
- Allow **type safety** across host and device

What about abstractions?

```
#define __CL_ENABLE_EXCEPTIONS

#include <math.h>

#include <cstdio>
#include <cstdlib>
#include <iostream>

#include "cl.hpp"

// OpenCL kernel. Each work item takes care of one element of c
const char *kernelSource =
    "\n"
    "#pragma OPENCL EXTENSION cl_khr_fp64 : enable                  \n"
    "__kernel void vecAdd(  __global double *a,                     \n"
    "                       __global double *b,                     \n"
    "                       __global double *c,                     \n"
    "                       const unsigned int n)                   \n"
    "{                                                              \n"
    "    //Get our global thread ID                                 \n"
    "    int id = get_global_id(0);                                 \n"
    "                                                               \n"
    "    //Make sure we do not go out of bounds                     \n"
    "    if (id < n)                                                \n"
    "        c[id] = a[id] + b[id];                                 \n"
    "}                                                              \n"
    "\n";

int main(int argc, char *argv[]) {
    // Length of vectors
    unsigned int n = 1000;

    // Host input vectors
    double *h_a;
    double *h_b;
    // Host output vector
    double *h_c;

    // Device input buffers
    cl::Buffer d_a;
    cl::Buffer d_b;
    // Device output buffer
    cl::Buffer d_c;

    // Size, in bytes, of each vector
    size_t bytes = n * sizeof(double);

    // Allocate memory for each vector on host
    h_a = new double[n];
    h_b = new double[n];
    h_c = new double[n];

    // Initialize vectors on host
    for (int i = 0; i < n; i++) {
        h_a[i] = sinf(i) * sinf(i);
        h_b[i] = cosf(i) * cosf(i);
    }

    cl_int err = CL_SUCCESS;
    try {
        // Query platforms
        std::vector<cl::Platform> platforms;
        cl::Platform::get(&platforms);
        if (platforms.size() == 0) {
            std::cout << "Platform size 0\n";
            return -1;
        }

        // Get list of devices on default platform and create context
        cl_context_properties properties[] = {CL_CONTEXT_PLATFORM,
                                              (cl_context_properties)(platforms[0])(), 0};
        cl::Context context(CL_DEVICE_TYPE_GPU, properties);
        std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();

        // Create command queue for first device
        cl::CommandQueue queue(context, devices[0], 0, &err);

        // Create device memory buffers
        d_a = cl::Buffer(context, CL_MEM_READ_ONLY, bytes);
        d_b = cl::Buffer(context, CL_MEM_READ_ONLY, bytes);
        d_c = cl::Buffer(context, CL_MEM_WRITE_ONLY, bytes);

        // Bind memory buffers
        queue.enqueueWriteBuffer(d_a, CL_TRUE, 0, bytes, h_a);
        queue.enqueueWriteBuffer(d_b, CL_TRUE, 0, bytes, h_b);

        // Build kernel from source string
        cl::Program::Sources source(1,
                                    std::make_pair(kernelSource, strlen(kernelSource)));
        cl::Program program_ = cl::Program(context, source);
        program_.build(devices);

        // Create kernel object
        cl::Kernel kernel(program_, "vecAdd", &err);

        // Bind kernel arguments to kernel
        kernel.setArg(0, d_a);
        kernel.setArg(1, d_b);
        kernel.setArg(2, d_c);
        kernel.setArg(3, n);

        // Number of work items in each local work group
        cl::NDRange localSize(64);
        // Number of total work items - localSize must be devisor
        cl::NDRange globalSize((int)(ceil(n / (float)64) * 64));

        // Enqueue kernel
        cl::Event event;
        queue.enqueueNDRangeKernel(kernel, cl::NullRange, globalSize, localSize, NULL,
                                   &event);

        // Block until kernel completion
        event.wait();

        // Read back d_c
        queue.enqueueReadBuffer(d_c, CL_TRUE, 0, bytes, h_c);
    } catch (cl::Error err) {
        std::cerr << "ERROR: " << err.what() << "(" << err.err() << ")" << std::endl;
    }

    // Sum up vector c and print result divided by n, this should equal 1 within error
    double sum = 0;
    for (int i = 0; i < n; i++) sum += h_c[i];
    std::cout << "final result: " << sum / n << std::endl;

    // Release host memory
    delete (h_a);
    delete (h_b);
    delete (h_c);

    return 0;
}
```

OpenCL

```
std::vector<int> add(const std::vector<int>& a, const std::vector<int>& b) {
    // We are going to operate on the common indexes subset
    std::vector<int> result(std::min(std::size(a), std::size(b)));
    if (std::size(result) == 0) return {};

    // Open a new scope to bind the lifetime of the command queue.
    {
        // Queue's destructor will wait for all pending operations to complete.
        sycl::queue queue;

        // Create buffers (views on a, b and result contiguous storage)
        sycl::buffer<int> A{std::data(a), std::size(result)};
        sycl::buffer<int> B{std::data(b), std::size(result)};
        sycl::buffer<int> R{std::data(result), std::size(result)};

        // The command group describing all operations needed for the kernel
        // execution
        queue.submit([&](sycl::handler& cgh) {
            // Get proper accessors to existing buffers by specifying
            // read/write intents
            auto ka = A.get_access<sycl::access::mode::read>(cgh);
            auto kb = B.get_access<sycl::access::mode::read>(cgh);
            auto kr = R.get_access<sycl::access::mode::write>(cgh);

            // Enqueue parallel kernel
            cgh.parallel_for<AddKernel>(
                sycl::range<1>{std::size(result)},
                [=](sycl::id<1> idx) {  // Be sure to capture by value!
                    kr[idx] = ka[idx] + kb[idx];
                });
        });  // End of our commands for this queue
    }        // End scope, so we wait for the queue to complete
    // Queue destructor has completed, results are now available
    // in output storage
    return result;
}
```

SYCL

```
std::vector<int> add(const std::vector<int>& a, const std::vector<int>& b) {
    // We are going to operate on the common indexes subset
    std::vector<int> result(std::min(std::size(a), std::size(b)));
    if (std::size(result) == 0) return {};

    // Open a new scope to bind the lifetime of the command queue.
    {
        // Queue's destructor will wait for all pending operations to complete.
        sycl::queue queue;

        // Create buffers (views on a, b and result contiguous storage)
        sycl::buffer<int> A{std::data(a), std::size(result)};
        sycl::buffer<int> B{std::data(b), std::size(result)};
        sycl::buffer<int> R{std::data(result), std::size(result)};

        // The command group describing all operations needed for the kernel
        // execution
        queue.submit([&](sycl::handler& cgh) {
            // Get proper accessors to existing buffers by specifying
            // read/write intents
            auto ka = A.get_access<sycl::access::mode::read>(cgh);
            auto kb = B.get_access<sycl::access::mode::read>(cgh);
            auto kr = R.get_access<sycl::access::mode::write>(cgh);

            // Enqueue parallel kernel
            cgh.parallel_for<class AddKernel>(
                sycl::range<1>{std::size(result)},
                [=](sycl::id<1> idx) {  // Be sure to capture by value!
                    kr[idx] = ka[idx] + kb[idx];
                });
        });  // End of our commands for this queue
    }        // End scope, so we wait for the queue to complete
    // Queue destructor has completed, results are now available
    // in output storage
    return result;
}
```

Buffers

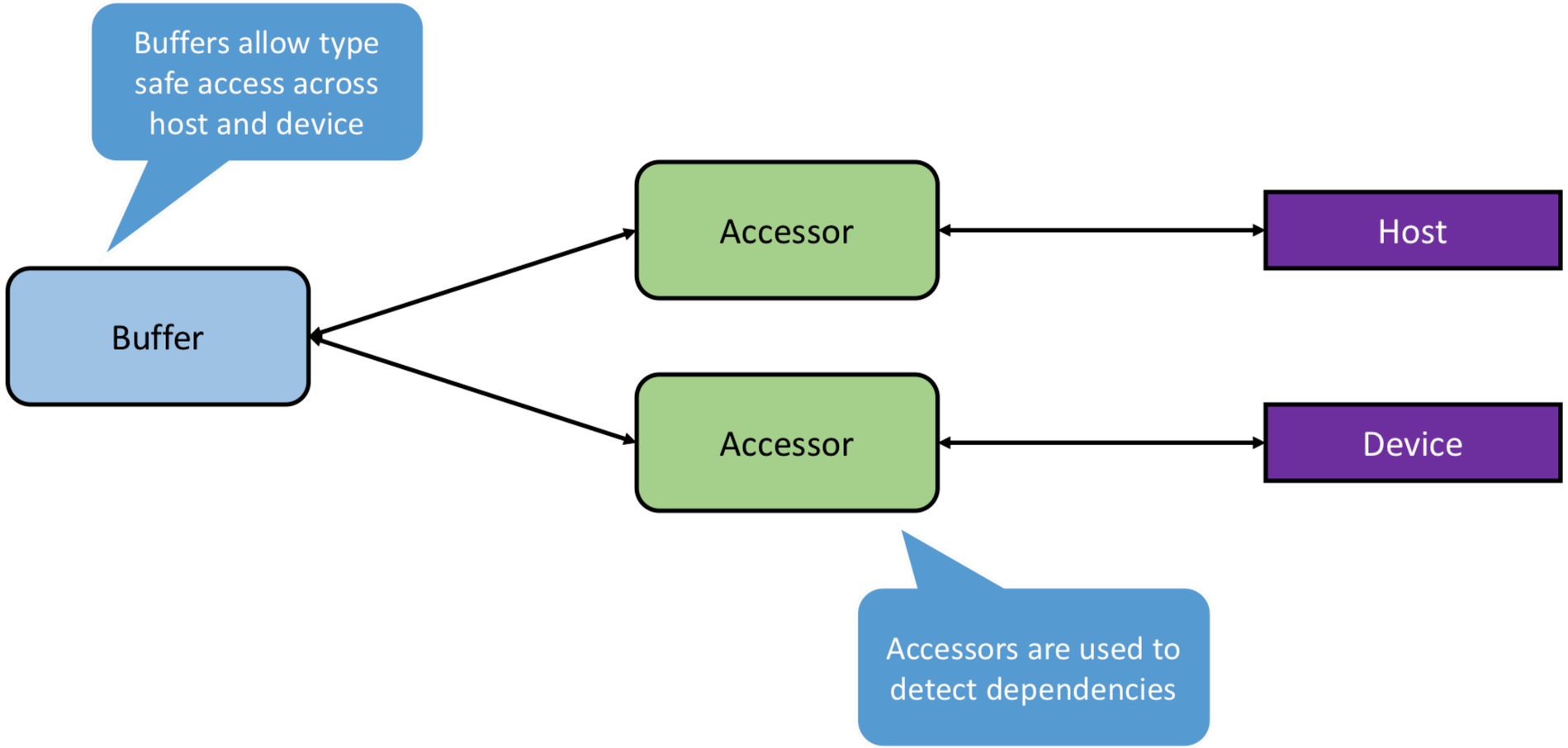
Command group

Accessor

Kernel

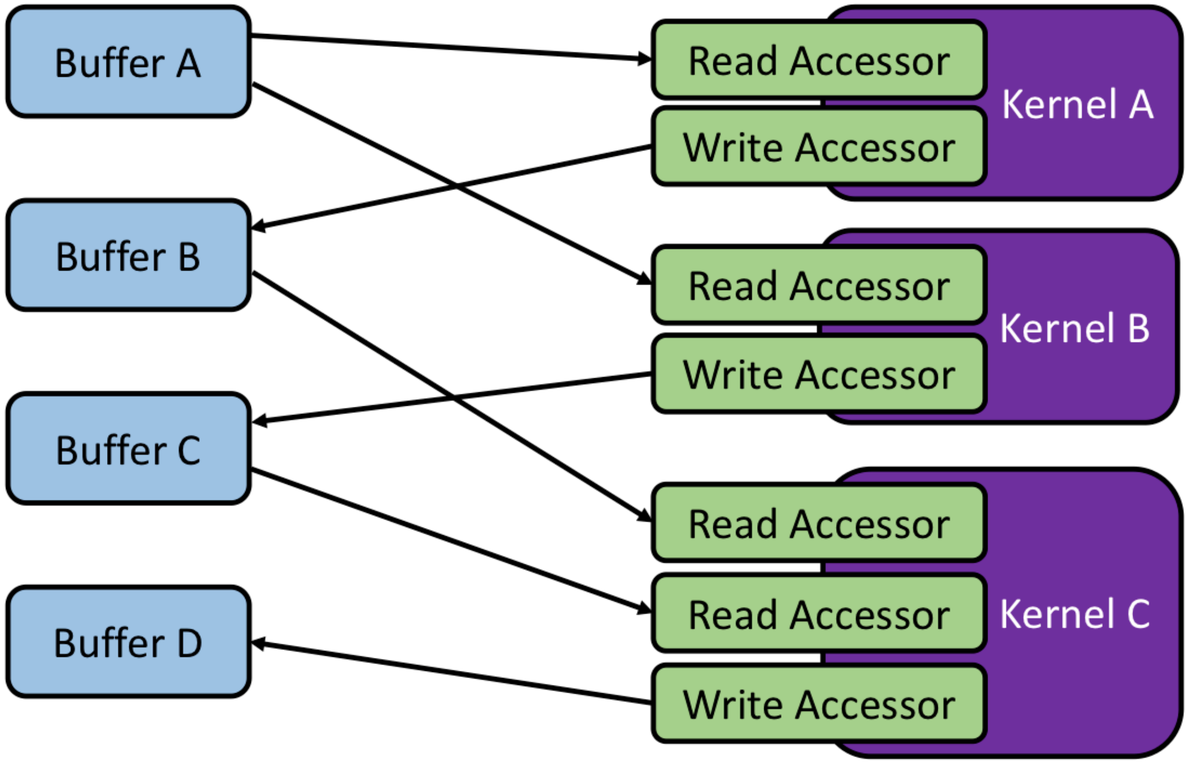
Hello SYCL: Vector Add

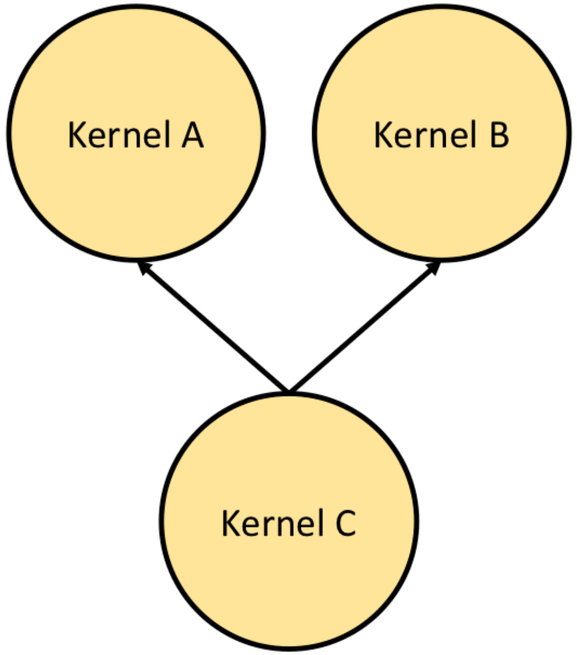
Separating Data & Access



Thanks to Gordon Brown/Codeplay

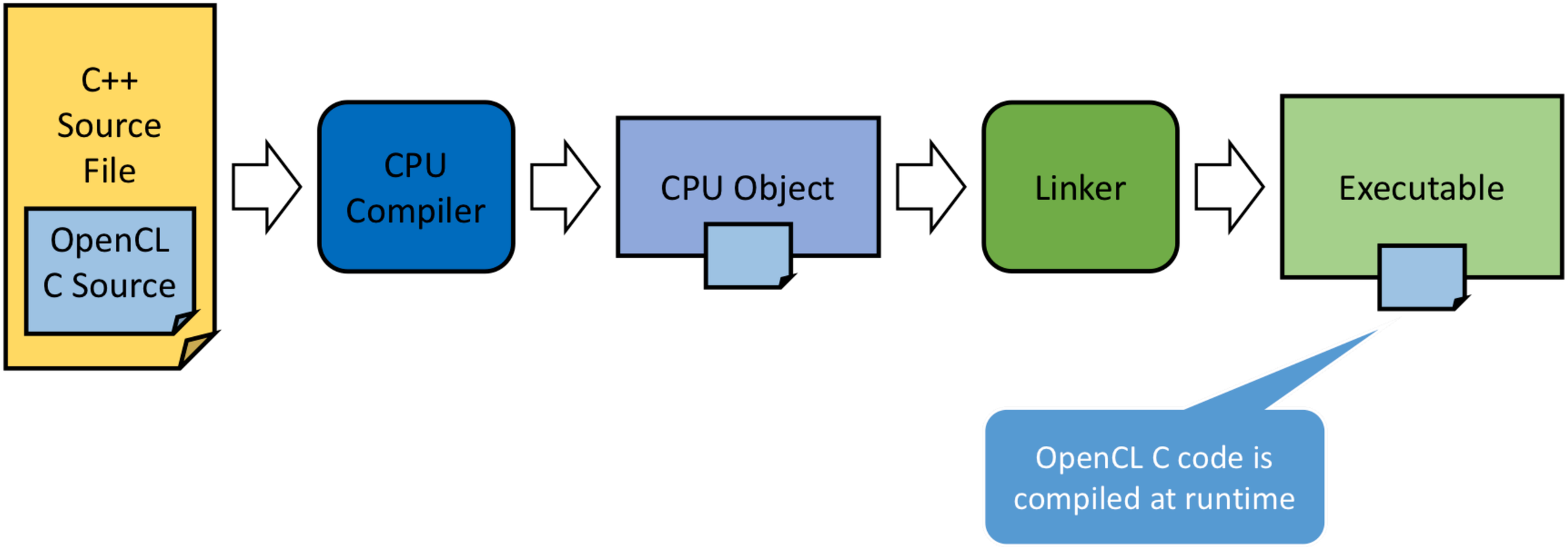
Dependency Task Graph





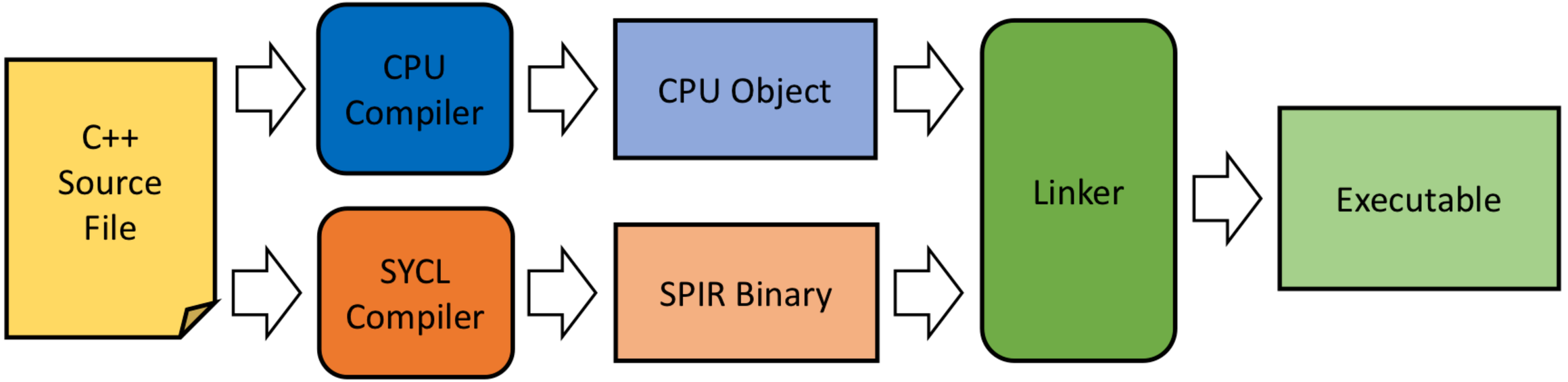
Thanks to Gordon Brown/Codeplay

Compilation Workflow: Separate Source (OpenCL)

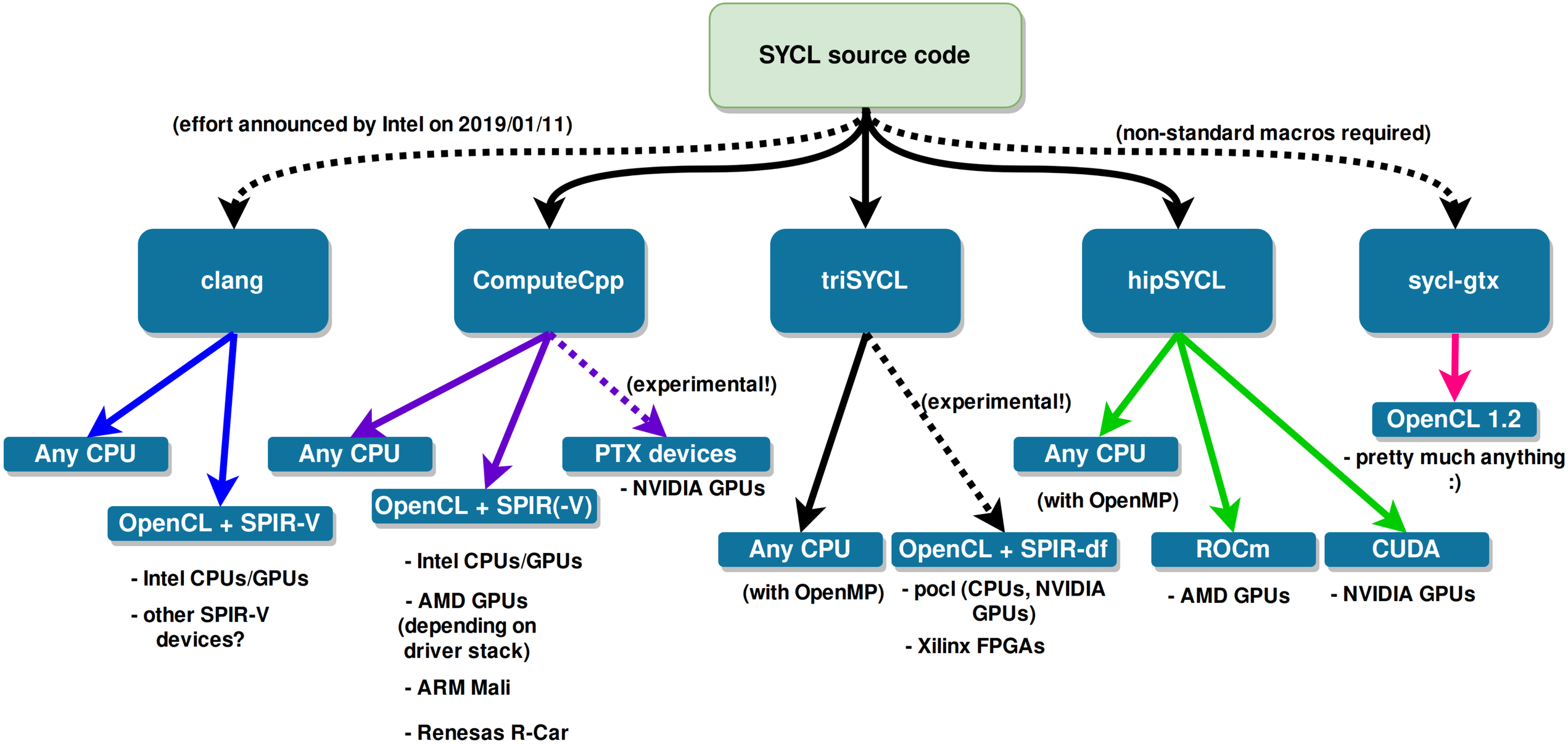


Thanks to Gordon Brown/Codeplay

Compilation Workflow: Shared Source (SYCL)



Thanks to Gordon Brown/Codeplay



Thanks to Aksel Alpay/github.com/illuhad

What about Parallel STL?

```
std::transform(std::begin(a), std::end(a),
               std::begin(b), std::begin(r), std::plus<>{});
```

```
#include <execution>

std::transform(std::execution::par_unseq,
               std::begin(a), std::end(a),
               std::begin(b), std::begin(r), std::plus<>{});
```

```
sycl::buffer<int> ka{std::data(a), std::size(a));
sycl::buffer<int> kb{std::data(b), std::size(b));
sycl::buffer<int> kr{std::data(r), std::size(r));
sycl::queue q;
sycl::sycl_execution_policy<class VectorAdd> sycl_policy{q};

std::transform(sycl_policy, std::begin(ka), std::end(ka),
               std::begin(kb), std::begin(kr), std::plus<>{});
```

Thank you!

nazavode.github.io