codeplay

SYCL™ for OpenCL™

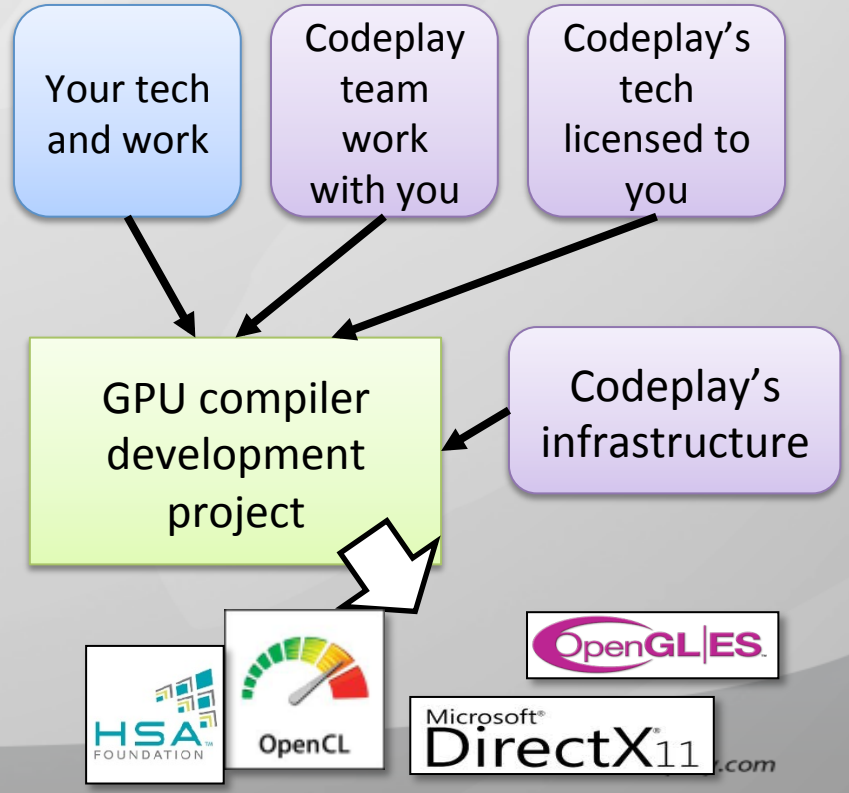
Andrew Richards, CEO Codeplay
& Chair SYCL Working group

Visit us at
www.codeplay.com

45 York Place
Edinburgh
EH1 3HP
United Kingdom

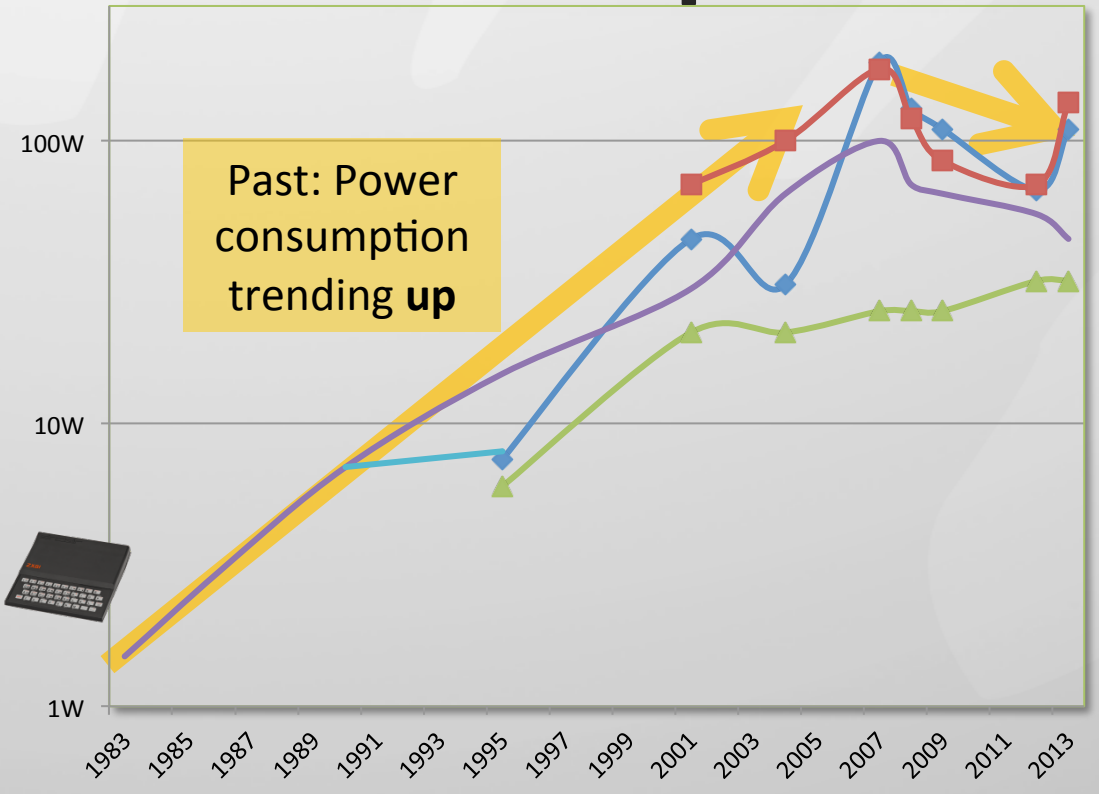
Codeplay: How we help

- Codeplay work as part of a project to deliver a new technology
- We work in partnership with our customers
  - Part of our customers' team
  - Bringing in a mix of: existing technology, contracting, testing tools, experience and optimization techniques



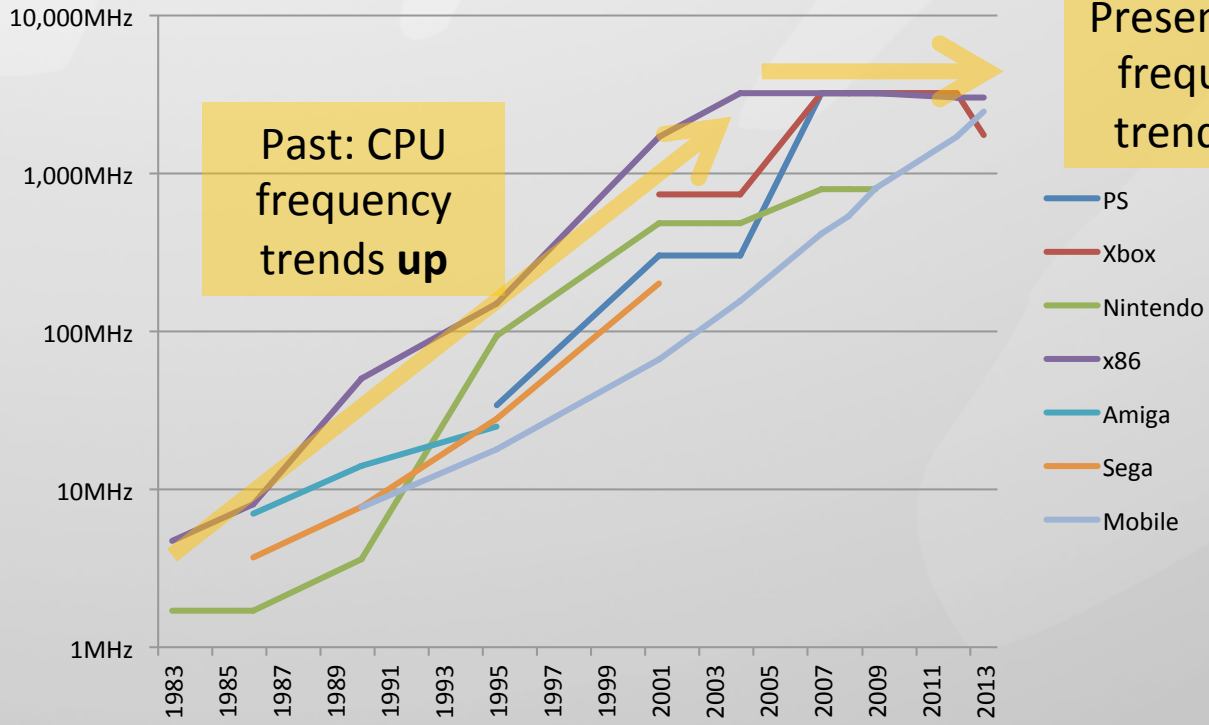
Power consumption over time

(Logarithmic scale)



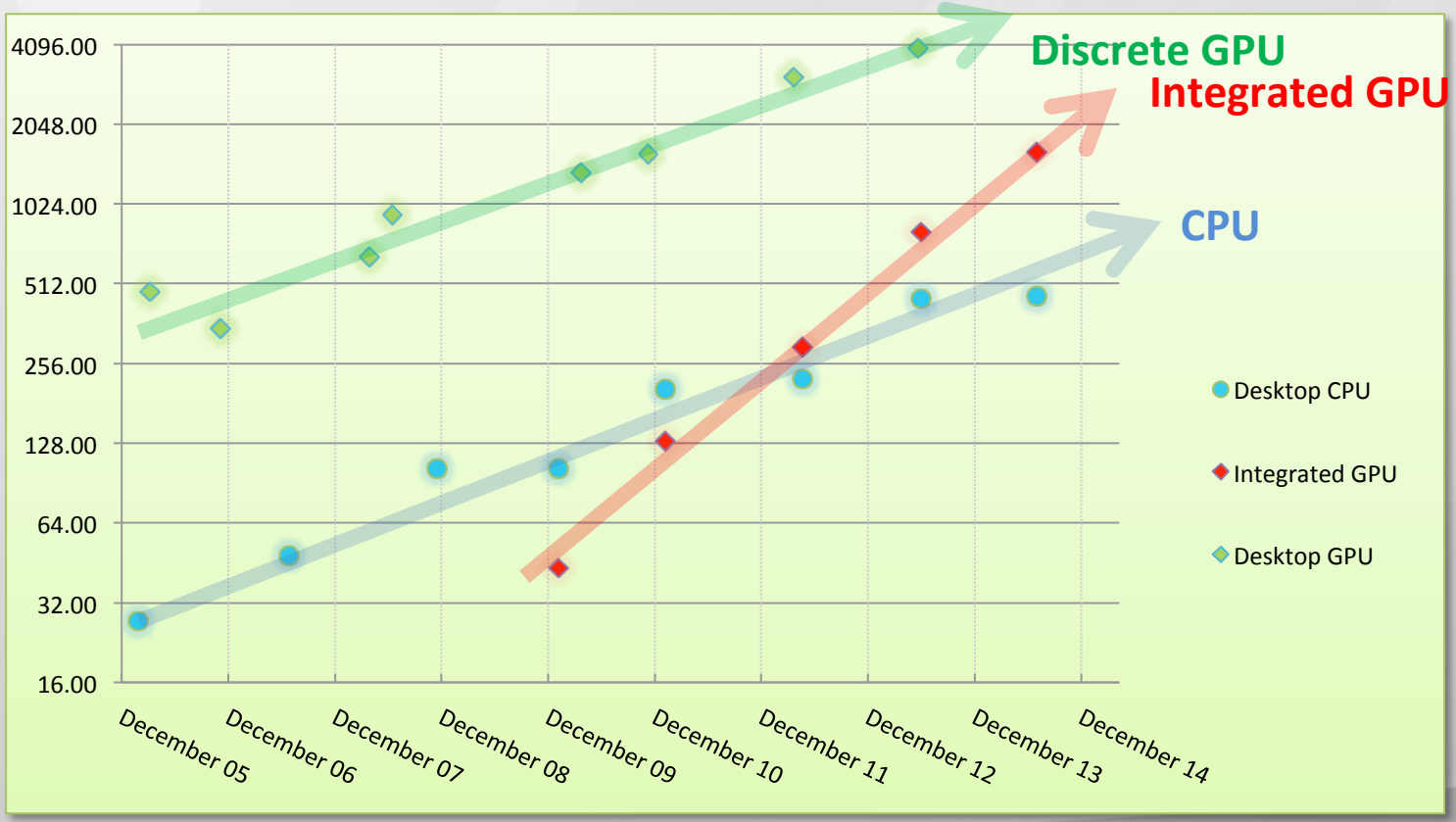
CPU Clock Frequency Over Time

(Logarithmic scale)



Raw performance Trends: Desktop

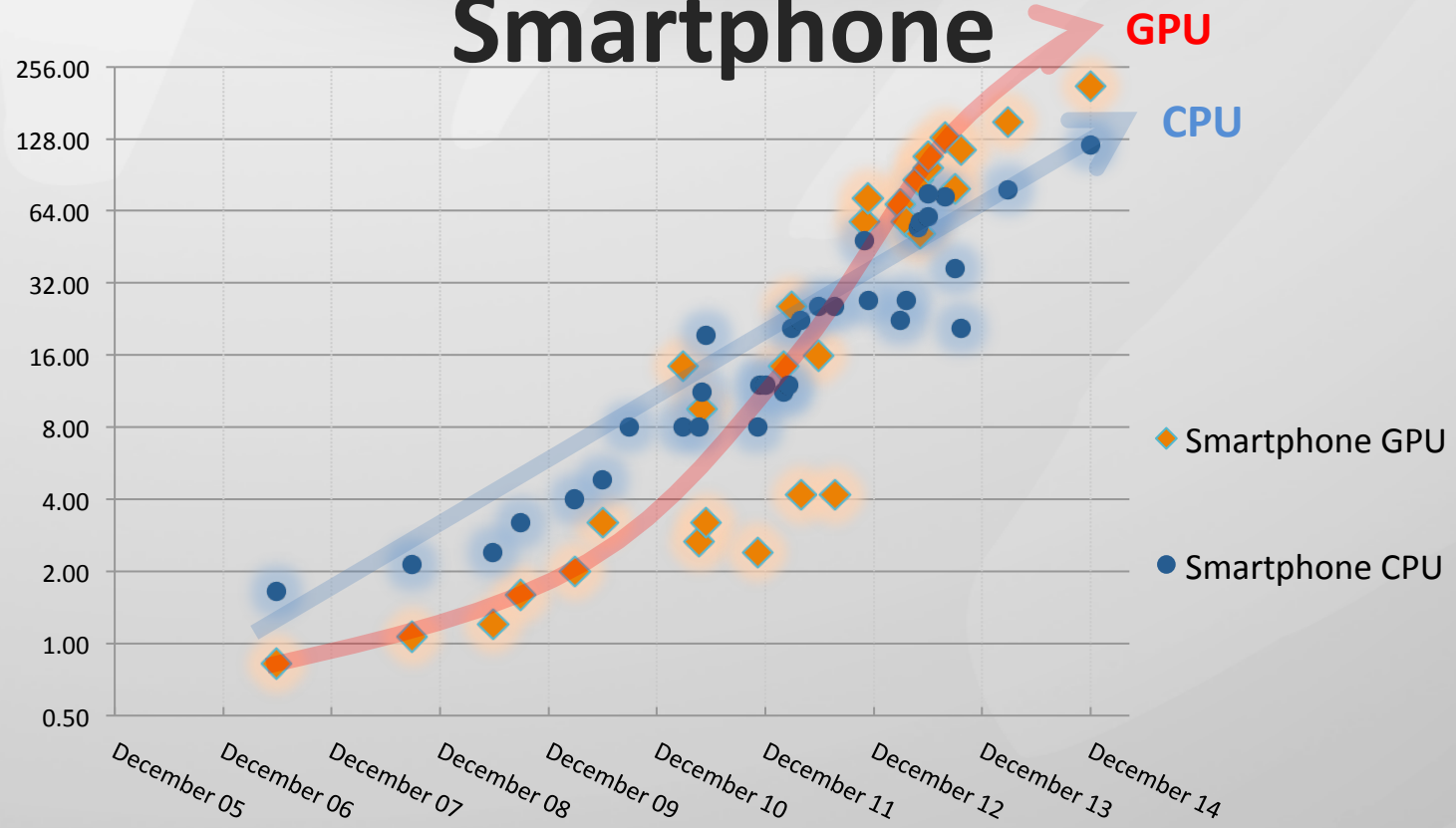
(Logarithmic scale)



Source: lots of different places, these actually have very wide variation. Very rough figures!

Raw performance Trends: Smartphone

codeplay

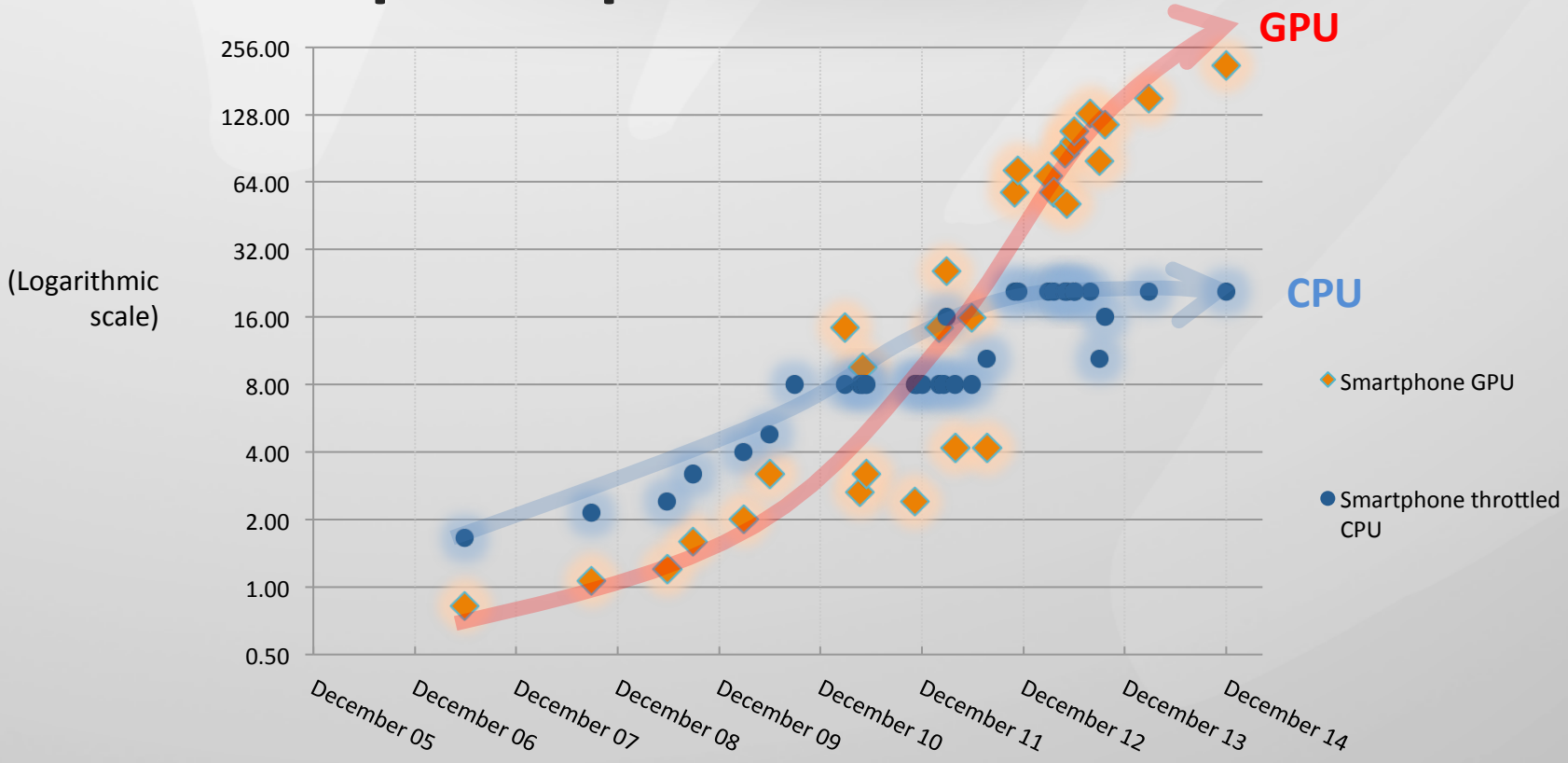


Source: lots of different places, these actually have very wide variation. Very rough figures!

Reality: throttling

- What happens if you run all 4 smartphone CPU cores at 2.7GHz for more than a few seconds?

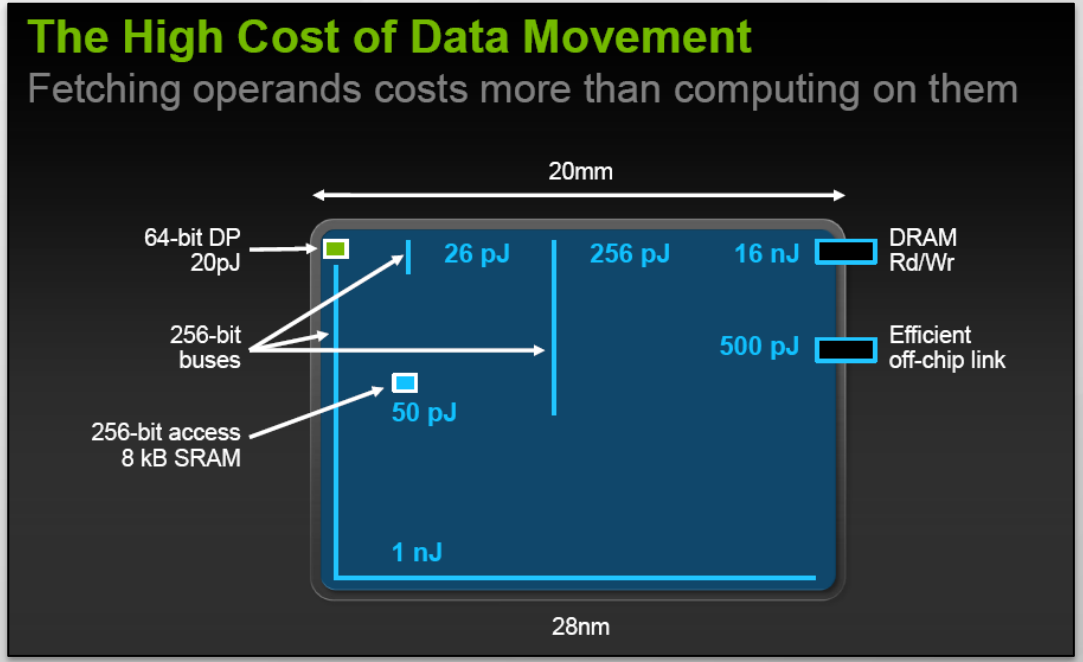
Smartphone performance trends: throttled



Source: lots of different places, these actually have very wide variation. Very rough figures!

Power and data movement

- Power consumption is more about data movement than processing
- (Also about frequency)



Source: NVIDIA: Bill Dally's presentation at SC10

Performance and Power

- Power cannot be increased
  - Manufacturers won't build machines which require more power
  - (unless we have fire-proof pockets)
- So:

power efficiency = performance

Why GPGPU and not other forms of parallelism?

- GPUs manage data-movement cost
  - Different memory spaces
  - Data-parallel means only one instruction fetch for tens of “threads”
  - Large register sets
  - Abstract instruction sets means innovation in processor cores

Why C++?

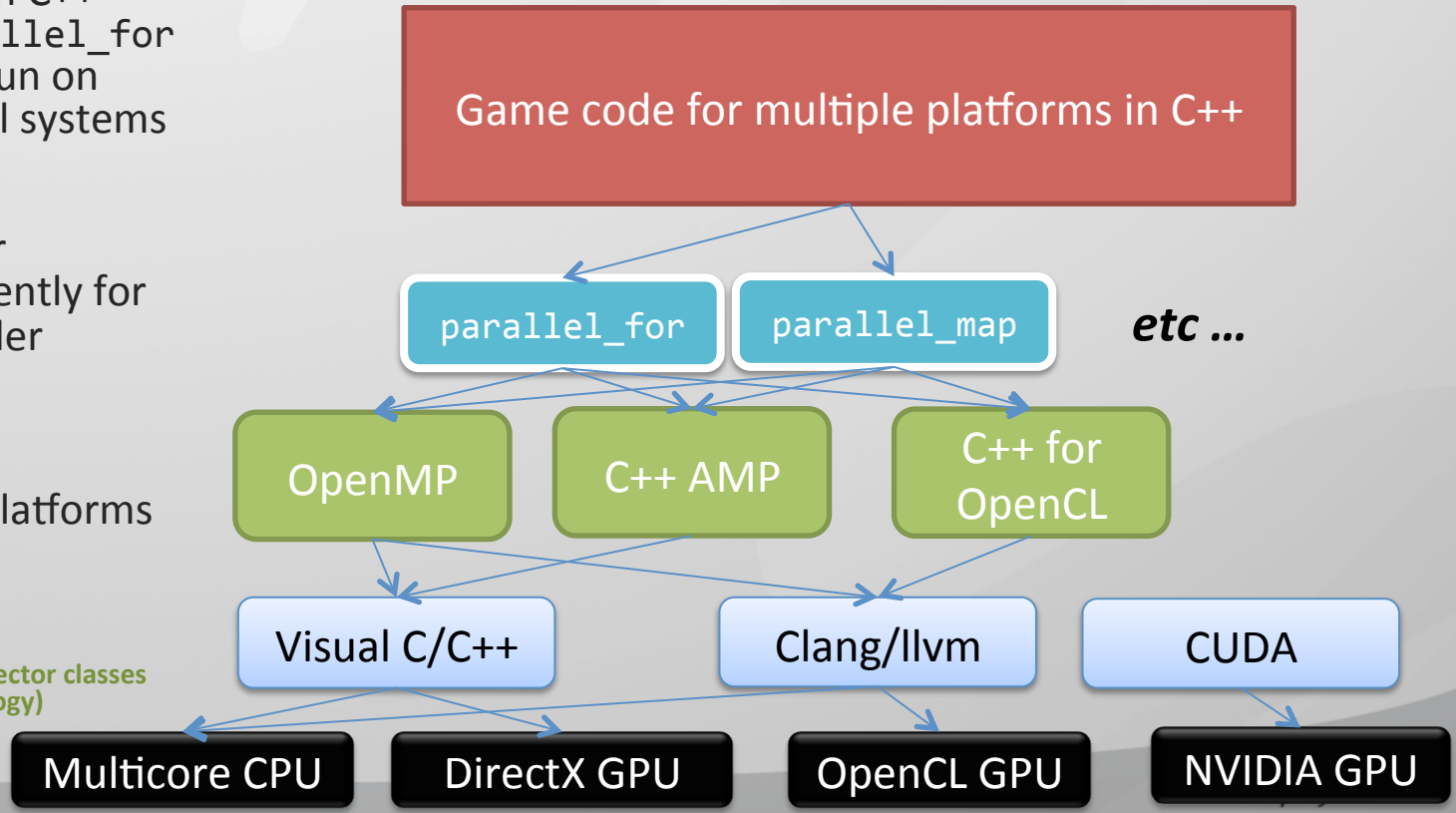
- C++ gives lots of opportunities to abstract platform differences

codeplay

C++ for multiple platforms

- Create your own C++ templated `parallel_for` functions that run on multiple parallel systems
- Implement your functions differently for different compiler technologies
- Run on lots of platforms and devices

(You probably already have vector classes that do this for SIMD technology)



codeplay

SYCL for OpenCL™

- Pronounced ‘sickle’
- Royalty-free, cross-platform C++ programming layer
  - Builds on concepts portability & efficiency of OpenCL
  - Ease of use and flexibility of C++
- Single-source C++ development
  - C++ template functions can contain host & device code
    - e.g. `parallel_sort<MyType> (myData);`
  - Construct complex reusable algorithm templates that use OpenCL for acceleration
- Provisional spec released at GDC in March 2014

SYCL Roadmap

- GDC, March 2014
  - Released a provisional specification to enable feedback
  - Developers can provide input into standardization process
  - Feedback via Khronos forums
- Next steps
  - Full specification, based on feedback
  - Conformance testsuite to ensure compatibility between implementations
  - Release of implementations

What we want to achieve

- We want to enable a C++ on OpenCL ecosystem
  - With C++ libraries supported on OpenCL
  - C++ tools supported on OpenCL
  - Aim to achieve long-term support for OpenCL features with C++
  - Good performance of C++ software on OpenCL
  - Multiple sources of implementations
  - Enable future innovation

Simple example

Does everything* expected of an OpenCL program: compilation, startup, shutdown, host fall-back, queue-based parallelism, efficient data movement.

* (this sample doesn't catch exceptions)

```
#include <CL/sycl.hpp>

int main ()
{
    int result; // this is where we will write our result

    {  // by sticking all the SYCL work in a {} block, we ensure
       // all SYCL tasks must complete before exiting the block

                        //  create a queue to work on
        cl::sycl::queue myQueue;

         // wrap our result variable in a buffer
        cl::sycl::buffer<int> resultBuf (&result, 1);

         // create some 'commands' for our 'queue'
        cl::sycl::command_group (myQueue, [&] ()
        {
            // request access to our buffer
            auto writeResult = resultBuf.access<cl::sycl::access:write_only> ();

            // enqueue a single, simple task
            single_task(kernel_lambda<class simple_test>([=] ()
            {
                writeResult [0] = 1234;
            }
        }); // end of our commands for this queue

    } // end scope, so we wait for the queue to complete

    printf ("Result = %d\n", result);
}
```

Templated matrix multiply example

A user can instantiate this templated class for a given type:

```
sycl::command_group(myQueue,
                    matrix_mul<double> (
                              Mdim, Ndim, Pdim, d_a, d_b, d_c)
                    );
```

```
template<typename TYPE>
class matrix_mul_row_priv : public matrix_mul
{

    void do_mxm()
    {
        int _Ndim = this->_Ndim;
        int _Mdim = this->_Mdim;
        int _Pdim = this->_Pdim;
        auto ptrA = _ma.get_access<access::read> ();
        auto ptrB = _mb.get_access<access::read> ();
        auto ptrC = _mc.get_access<access::write> ();

        parallel_for(nd_range<1>(range<1>(_Ndim), range<1>(ORDER/16)),
                kernel_functor<class mxm_kernel_row_priv> ([=] (
                                cl::hlm::item_id item, void*)
                {
                    int k, j;
                    int i = item.get_global(0);
                    TYPE Awrk[ORDER];
                    TYPE tmp;

                    if (i < _Ndim)
                    {
                       for (k = 0; k < _Pdim; k++)
                           Awrk[k] = A[i*Ndim+k];

                       for (j = 0; j < _Mdim; j++)
                       {
                            tmp = 0.0;
                            for(k = 0; k < _Pdim; k++)
                            {
                                tmp += Awrk[k]
                                     * ptrB[k*_Pdim+j];
                            }
                            ptrC[i*Ndim+j] = tmp;
                        }
                    }

                })
        );
    }
};
```

Hierarchical parallelism

This example shows the special syntax that SYCL provides to help write OpenCL-style workgroups

This `parallel_for` executes in parallel across the workgroups

This code just executes once per workgroup

```
parallel_for_workgroup(nd_range<>(
                                        range<>(GLOBAL_ITEMS_1D,
                                                GLOBAL_ITEMS_1D,
                                                GLOBAL_ITEMS_1D),
                                        range<>(LOCAL_ITEMS_1D,
                                                LOCAL_ITEMS_1D,
                                                LOCAL_ITEMS_1D)
                                        ),
kernel_functor<class hierarchical_reduce> ([=] (group group)
{
    T local_sums[LOCAL_ITEMS_TOTAL];

    /* Process items in each work item */
    parallel_for_workitem(group, [=, &local_sums] (item item)
    {
        int local_id = item.get_global ();

        /* Split the array into work-group-size different arrays */
        int values_per_item = (INPUT_SIZE/NUM_GROUPS)/LOCAL_ITEMS_TOTAL;
        int id_start = values_per_item * local_id;
        int id_end = values_per_item * (local_id + 1);

        /* Handle the case where the number of input values is not
           divisible by the number of items. */
        if (id_end > INPUT_SIZE - 1)
            id_end = INPUT_SIZE - 1;

        for (int i = id_start; i < id_end; i++)
            local_sums[i].increment(inputPtr[i]);
    });

    /* Sum items in each work group */
    for (int i = 0; i < LOCAL_ITEMS_TOTAL; i++)
        groupSumsPtr[group.get_id(0)].increment(local_sums[i]);
}));
```

Easy-to-use C++ can target powerful heterogeneous systems

SYCL standard enables:

- C++ compilers
- C++ debuggers
- C++ template libraries
- Open-source & commercial C++ compilers & runtimes

SYCL can target:

- GPUs
- FPGAs
- CPUs
- DSPs
- Custom processor cores

So, let's build them!