codeplay®

Enabling AI & HPC to be open, safe and accessible to all

Performance Portability from Fantasy to Reality

oneAPI DevSummit for AI and HPC

December 2022

Andrew Richards

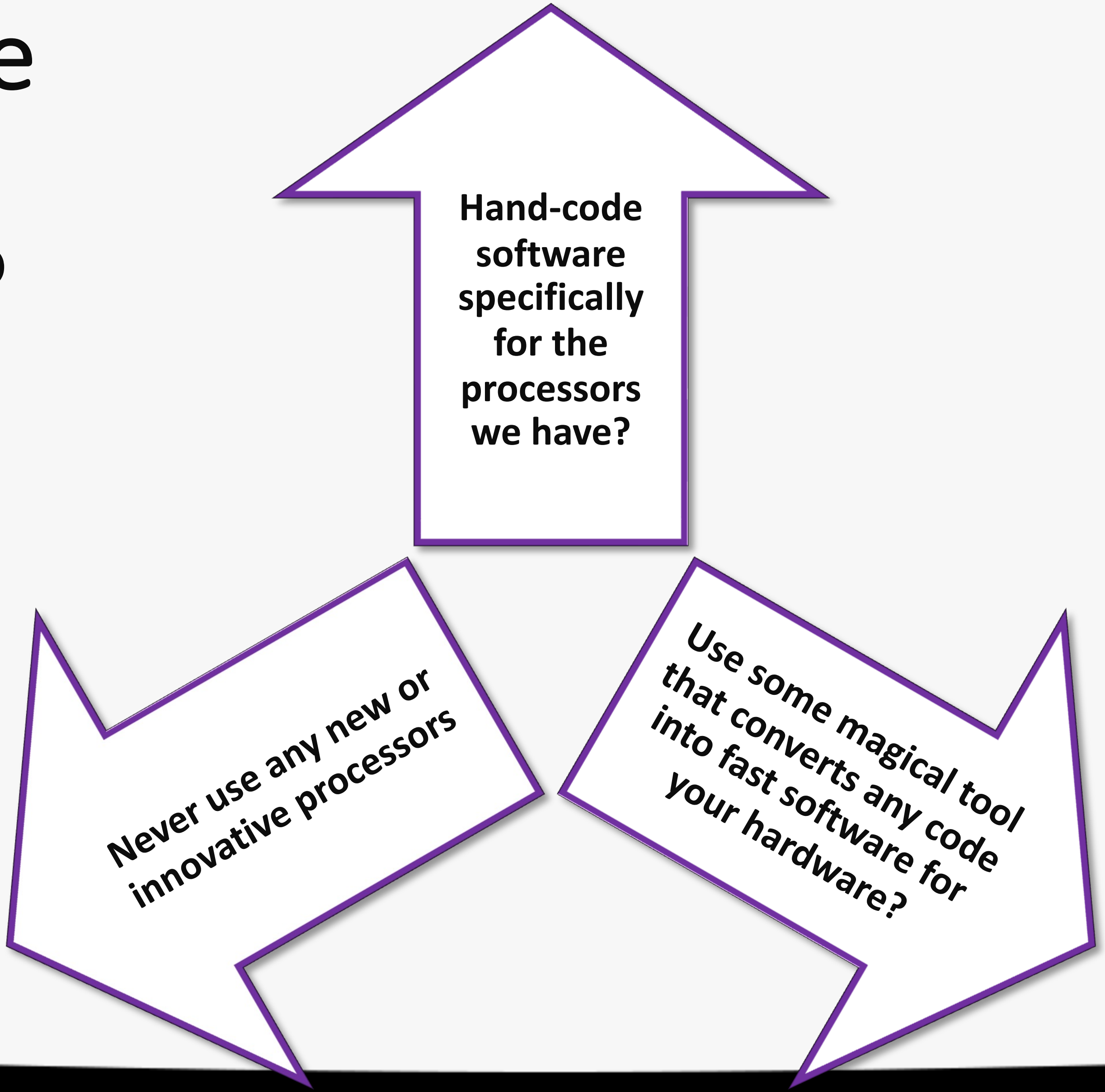
Our Brave New World

Single-core CPUs → Multi-core CPUs → System-on-chips with multiple optimized cores for each class of algorithm and datatype

Great for processor architects, but how do we write the software?

How do we write fast software?

(Your hardware will be obsolete by the time you have optimized it)



(Those days are over)

(Only works if your magical tool has been pre-programmed to understand the software you just invented)

If only there was a proven, practical, solution…

(There is, it's called C++, and it's very widely used)

C++ has 3 key concepts that enable it to support development of very large, very high performance software

1. Zero-cost abstractions
2. Separation of concerns
3. Composability

Starting simple: writing a parallel loop

1. We could write a serial loop & hope the compiler parallelizes it

```
void serial_f (float *out,
               const float *in,
               int size) {
    for (int i=0; i<n; i++) {
        out [i] = f (in [i]);
    }
}
```

2. We could write a serial loop & *tell* the compiler to parallelize it

```
void parallel_f (float *out,
                 const float *in,
                 int size) {
#pragma this_loop_is_parallel
    for (int i=0; i<n; i++) {
        out [i] = f (in [i]);
    }
}
```

```
void explicit_parallel_f (float *out,
                          const float *in,
                          int size) {
    parallel_for (0, n, [=] (int i) {
        out [i] = f (in [i]);
    });
}
```

This is a C++ *zero-cost-abstraction*

3. We could write a parallel loop in C++

Why would we do it like this?

⇒ we told the compiler what we want
⇒ now we have complete control
⇒ we can now parallelize very complex software
⇒ now, when we debug the software, it behaves exactly the way we told it to behave

Writing a parallel loop by hand

```
void parallel_part_f (float *out,
                      const float *in,
                      int start,
                      int end) {
    for (int i=start; i<end; i++) {
        out [i] = f (in [i]);
    }
}
```

4. Or, we could write the whole thing by hand

Why would we do it like this?

⇒ we don’t want to maintain this software on multiple platforms

⇒ we want to learn how multi-threading works

```
void parallel_threads_f (float *out,
                         const float *in,
                         int size) {
    int part = size / num_cores;
    for (int i=0; i<size; i+=part) {
        create_thread (parallel_part_f,
                       out, in, i,
                       min (size, i + part);
    }
    wait_for_threads_to_complete ();
}
```

Which of those 4 methods is faster?

(Answer #1: the serial loop is fastest, because I didn't tell you that *n* is 3)

Lesson: The fastest algorithm varies

Your compiler can’t know any of this

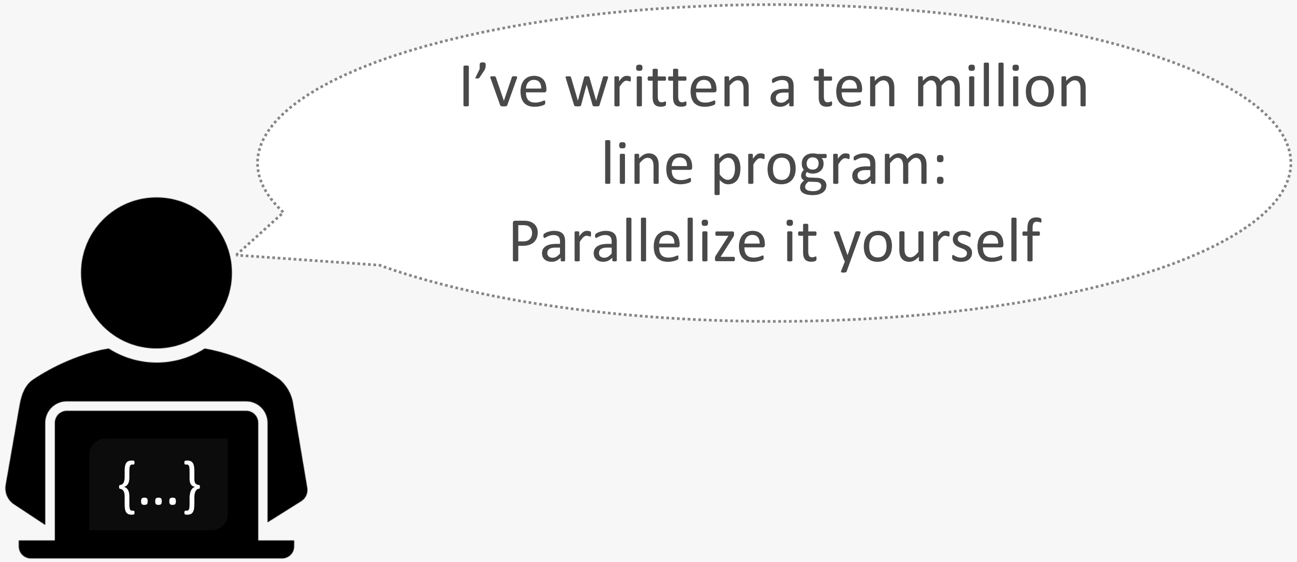
1. Performance varies by the size of the data
2. Performance varies by the underlying hardware
3. Performance varies by where the data is
4. Performance varies by what else is running (or could be running) at the same time in the same system

Your optimized libraries can’t know any of this

Who knows the answers?

The only person who knows: the size of the data; the hardware it's running on; where the data is, and what else is running on the system is:

The user!



How do we provide: programs, libraries & tools that can be parallelized and optimized on different systems?

We *separate the concerns*: This is a key modern C++ concept

What we want the software to do

The algorithm we will use to calculate the answer

Processor-specific optimizations

How the data is stored

We can then *independently choose*: the algorithm, the optimizations, which processor each task runs on, how we store the data

How do we optimize a program where we have *Separated the Concerns?*

Easy: we run it with all the different options and see which runs fastest!

We break down the optimization problem into three stages:

1. Writing optimized algorithms, data structures, kernels, schedules
2. Writing our software in a way where we can switch between the different algorithms, data structures, kernels, schedules
3. Choosing the best options for each for the problem we want to solve

But how do we integrate all the components?

C++ has an answer for this, too: *composability*

- If we write C++ libraries carefully, we can combine them together with user-written code
- If we want to compose across: data formats, different algorithms, different processors, user customization, scheduling, then:
    - ***We need to have the C++ in the same compilation unit, even for different processor cores***
    - ***We call this C++ single-source and it's crucial for making this work on today's heterogeneous multi-core processors***

This seems like an impossibly big task

But we've already done a lot of the work!

There are already several C++ libraries that enable this:
- Kokkos
- Raja
- Eigen
- SYCL-BLAS, SYCL-DNN

There are already C++ single-source compilers & standards to do this:
- ISO C++ Parallel STL
- CUDA, HIP
- SYCL – standard for heterogeneous devices
- C++ with OpenMP/OpenACC
- ComputeCpp, DPC++, triSYCL: implementations of SYCL

There are already applications doing this:
- TensorFlow
- A lot of videogame engines

There are already accelerators supporting this:
- Most CPUs – out of the box C++
- NVIDIA GPUs – CUDA (& SYCL)
- AMD GPUs – HIP (& SYCL)
- Intel GPUs – DPC++/SYCL
- Renesas R-Car - SYCL
- Imagination Technologies GPUs - SYCL
- ARM Mali GPUs – SYCL
- Intel FPGAs – DPC++/SYCL

What is SYCL?

- SYCL is a royalty-free vendor-neutral industry standard C++ for parallel software and accelerator processors
- **SYCL takes proven C++ performance ideas & super-charges them for a heterogeneous processing world**
- Now we can:
  - Build our own C++ SYCL compilers for a variety of new processors
  - We can design our own optimizations
  - We can build C++ libraries that can adapt to the performance requirements of lots of different systems
  - We can integrate native compilation for different processors in one source file

How SYCL handles parallelism

```
cgh.parallel_for<class parallel_demo> (
                cl::sycl::range<1>(n),
                [=](cl::sycl::item<1> i)
{
    out [i] = f (in [i]);
});
```

For more complex parallelism where there are scheduling dependencies, there are a range of options: SYCL requires you to specify where your code *isn't parallel*

- By default, a SYCL `parallel_for` can run entirely parallel
- We define a range to execute in parallel over
- We use a C++ lambda to define the loop body as that's standard now
- It is the job of the programmer to ensure 'f' is safe to run in parallel
- The loop is enqueued and run asynchronously to the CPU thread
- The parallel loop can execute on any SYCL supported core: CPU, GPU, FPGA, DSP, anything programmable

How SYCL handles data access

```
auto in = inp.get_access<cl::sycl::access::mode::read>(cgh);
auto out = outp.get_access<cl::sycl::access::mode::read_write>(cgh);
cgh.parallel_for<class parallel_demo> (
                cl::sycl::range<1>(n),
                [=](cl::sycl::item<1> i)
{
    out [i] = f (in [i]);
});
```

Access mode specified

Performance on accelerators is more about data access than compute:

- GPUs have on-board HBM memory and a small amount of fast on-chip SRAM
- DSPs use DMA to transfer data rapidly to a larger amount of on-chip SRAM
- AI accelerators usually have a lot of fast on-chip SRAM

SYCL requires developer specify how to access data: enables maximum performance

How SYCL handles multiple, different, processors

```
gpu_queue.submit([&](cl::sycl::handler &cgh) {
    auto in = inp.get_access<cl::sycl::access::mode::read>(cgh);
    auto out = outp.get_access<cl::sycl::access::mode::read_write>(cgh);
    cgh.parallel_for<class parallel_demo> (
                  cl::sycl::range<1>(n),
                  [=](cl::sycl::item<1> i)
    {
        out [i] = f (in [i]);
    });
});
```

Compiled for CPU by any normal CPU C++ compiler & runs asynchronously on host CPU to enqueue kernels to accelerator

SYCL Device Compiler extracts this kernel and compiles it natively for accelerator processors

This kernel 'name' allows multiple C++ compilers to be stitched together

- Both host & device code are compiled via C++ native compilers
- When SYCL goes through OpenCL, it can (optionally) use SPIR-V as the compiler IR
    - But it's still C++ source compiled to native device ISA
- SYCL device compilers can have per-device extensions
- More than one device compiler can compile a single source file

Combines the benefits of chosen CPU compiler *and* chosen device compiler

How SYCL handles processor-specific optimizations

- Most vector instructions and memory models map to SYCL 1.2.1 today
- New instructions or memory systems can be mapped to SYCL extensions – there's a clear mechanism for this

- Then, these processor-specific performance features are *integrated into the template libraries* in an appropriate place
  - The aim is to enable processor-specific optimizations in the least disruptive way possible
  - Enables us to run the same software with high performance on lots of different processors

Independent SYCL benchmarking

BabelStream

Memory transfer measurement

https://github.com/UoB-HPC/BabelStream

Parallel Research Kernels

A range of parallel kernel operations

https://github.com/ParRes/Kernels

RSBench

A key computational kernel of the monte carlo neutron transport algorithm

https://github.com/ANL-CESAR/RSBench

SYCLDSlash

Wilson Dslash Stencil Operator implementation

https://github.com/bjoo/SyCLDslash

HeCBench

A range of kernels for heterogenous computing

https://github.com/zjin-lcf/HeCBench

SYCL-Bench

A range of performance benchmarks

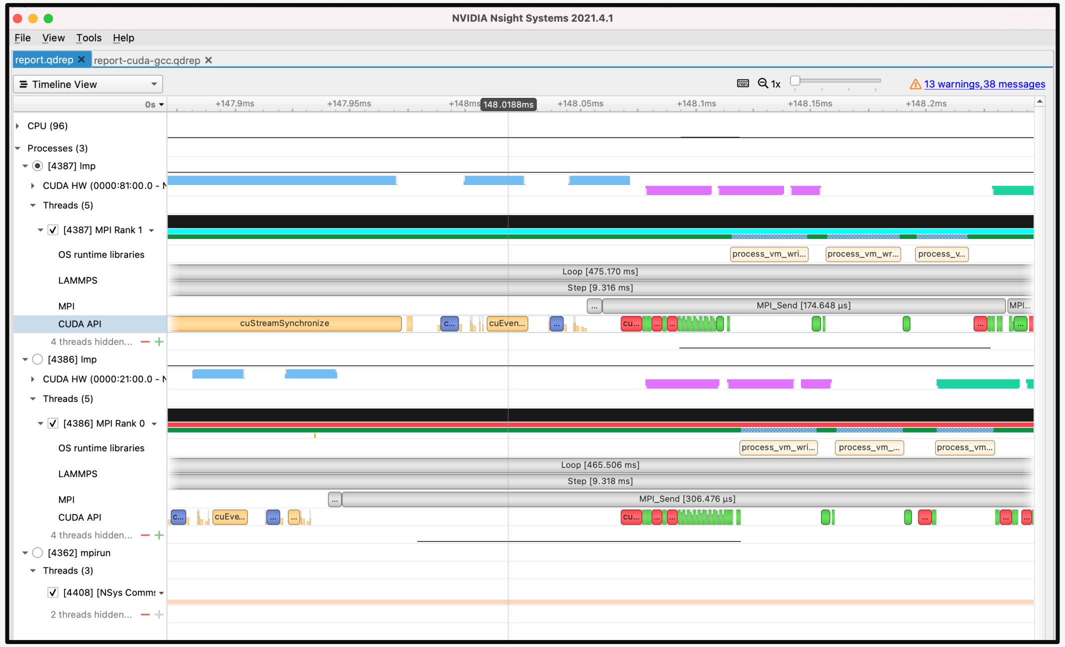
https://github.com/bcosenza/sycl-bench

https://www.cosenza.eu/papers/LalEUROPAR20.pdf

SYCL with Hardware-Specific Profiling Tools

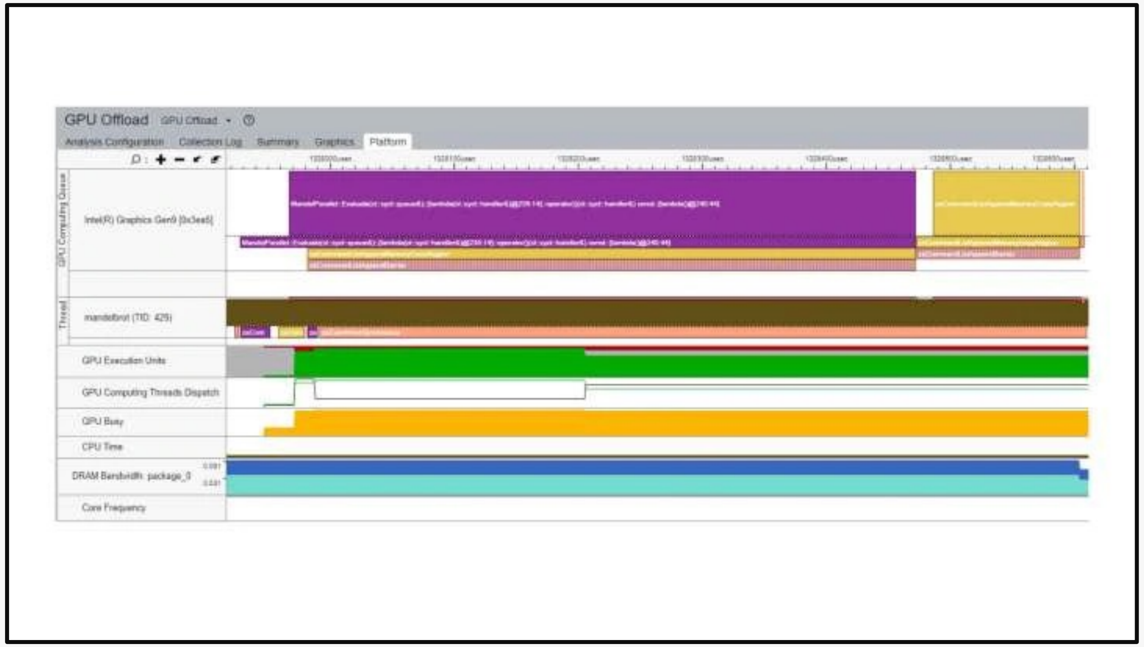
NVIDIA

- Nvprof and Nsight can be used in the same way with Nvidia GPUs



Intel

- Vtune can be used for Intel GPUs and CPUs



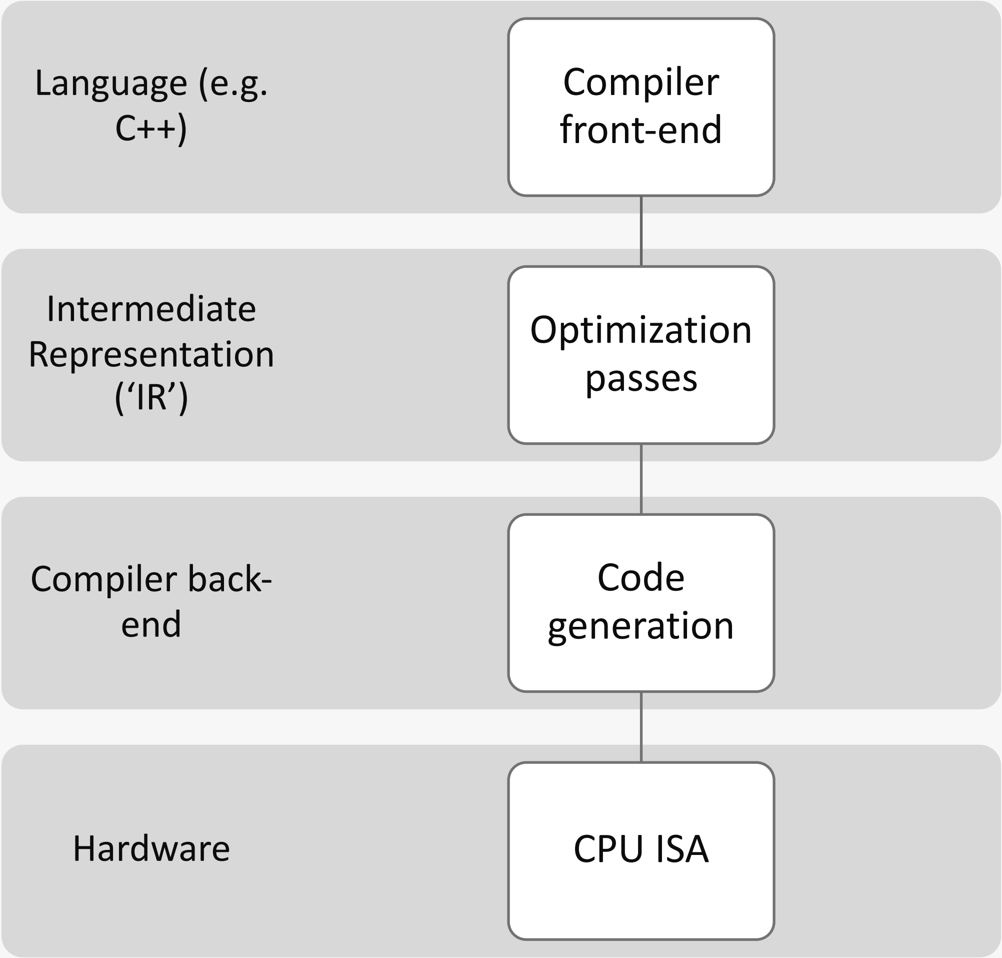
Using SYCL today

- oneAPI/DPC++ - Intel/Codeplay: new open governance
  - Open-source, very active development
  - Intel GPU, NVIDIA GPU, Intel FPGA support released so far
- hipSYCL – Heidelberg University
  - Open-source active development
  - AMD/NVIDIA GPUs: doesn't go through OpenCL
- ComputeCpp - Codeplay
  - Closed-source. Community Edition free. Professional Edition fully-supported
  - Supports OpenCL SPIR-V processors (ARM GPU, Renesas R-Car, PowerVR GPU, Intel GPU, +add your own)
- triSYCL - Xilinx
  - Open-source, less active development now

Check out the growing SYCL ecosystem at sycl.tech & the growing oneAPI ecosystem

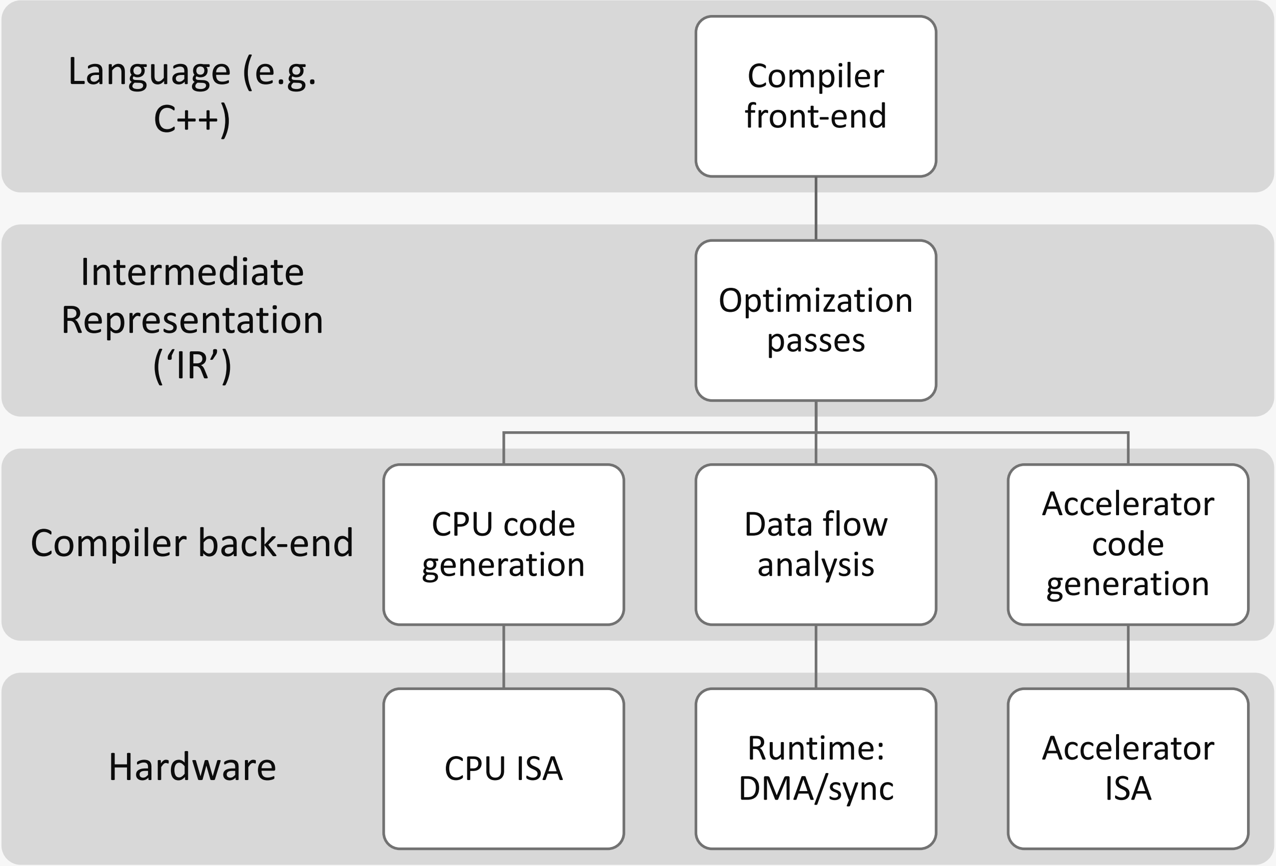
But, you promised a magic compiler that optimizes everything for me!

How compilers work



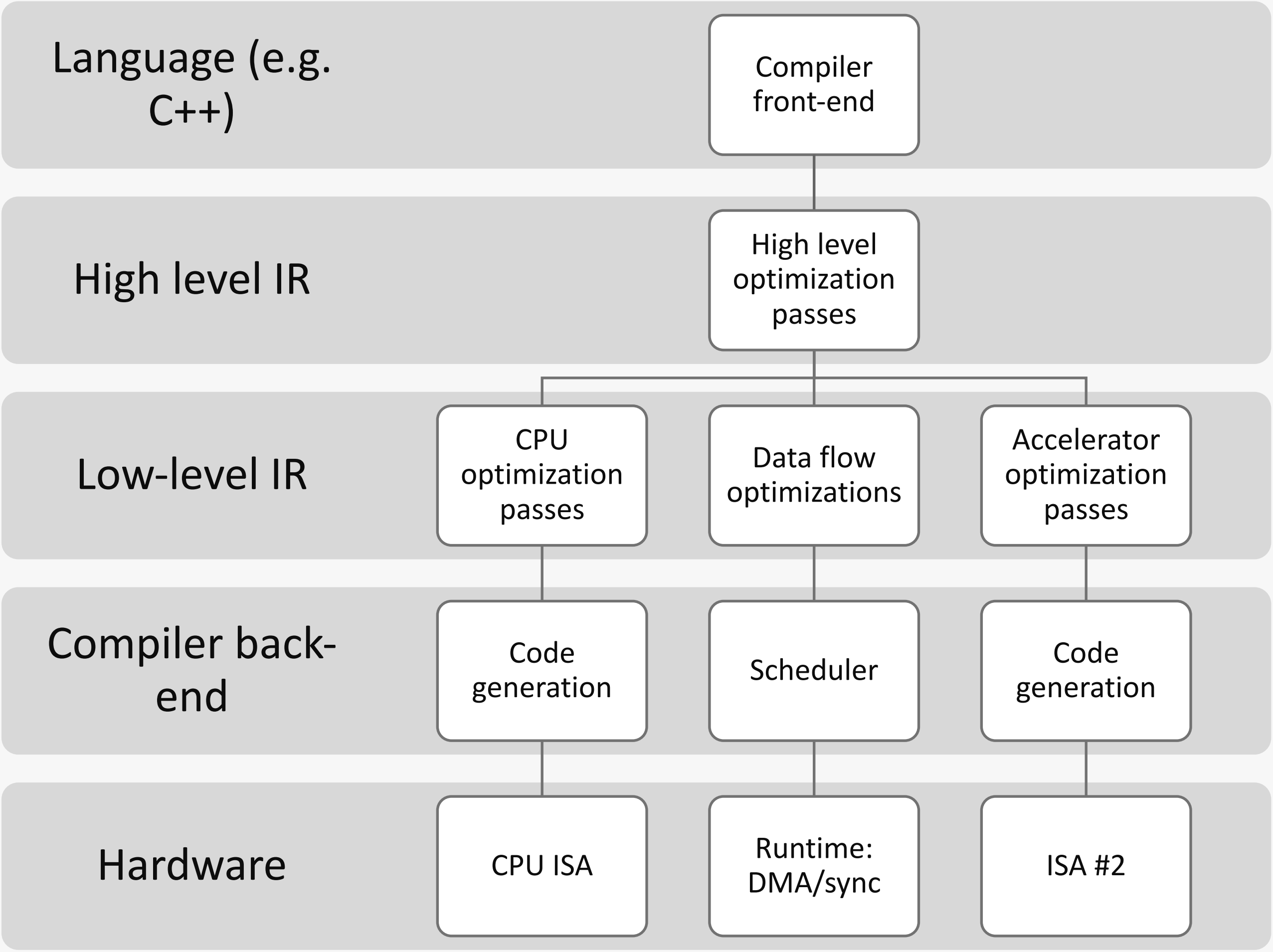
- We transform a language into an intermediate representation which contains a simplified representation of our code
- We do this because it's much easier to *transform* an IR with *passes*

How *heterogeneous* compilers work



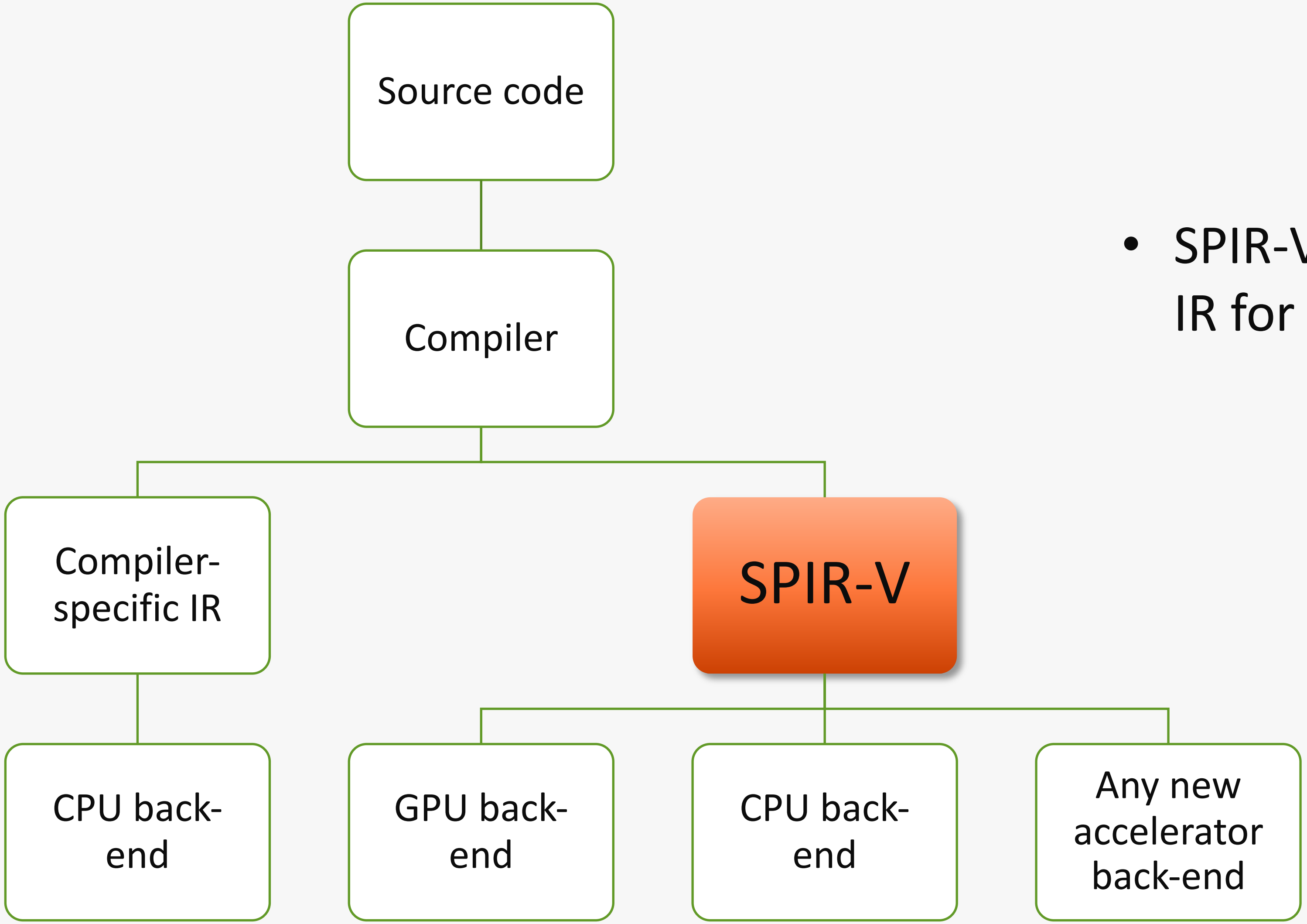
- We now need to create code for 2 (or more) processors
  - 2+ compiler back-ends
- And we also need to transfer data and synchronize
  - We have a *runtime*

Multi-Level Intermediate Representation (MLIR)



- MLIR lets us do different optimizations at different levels
- Enables optimizations for different hardware

SPIR-V



- SPIR-V is a standardized compiler IR for accelerators, such as GPUs

Now you can write your own domain-specific compiler + integrate it with other hardware & software

What now?

What now?

- We’re building out this open ecosystem together
- Join the oneAPI Community Forum to help drive the ecosystem
- Join the Khronos SYCL working group to drive the programming model
- Build performance-portable C++ frameworks
- Use these frameworks & techniques in your projects

codeplay®

Enabling AI to be open, safe and accessible to all