OpenCL™ as an Infrastructure

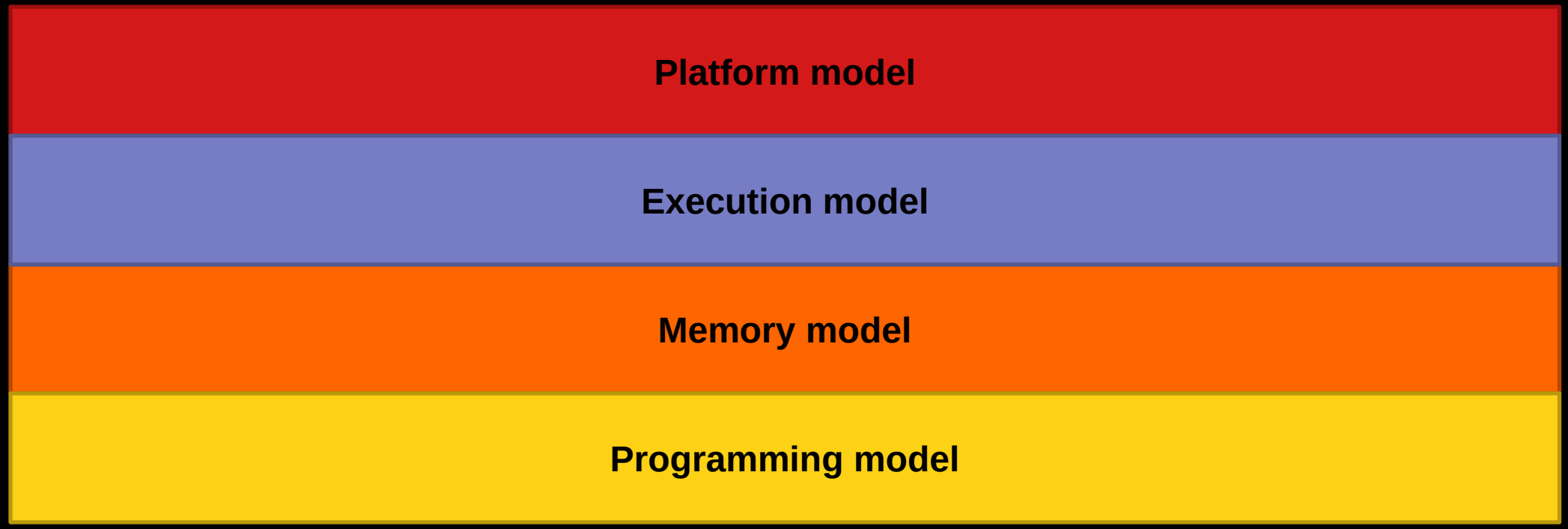
LEE HOWES
MAY 2013

AGENDA OR CALL-OUTS

AMD

- **OpenCL, the state of the art**
- **Building on OpenCL**
- **Current limitations**
- **Creating a portable intermediate language**
- **Hiding OpenCL away**
- **The future of supporting a variety of runtimes**

WHAT IS OPENCL?

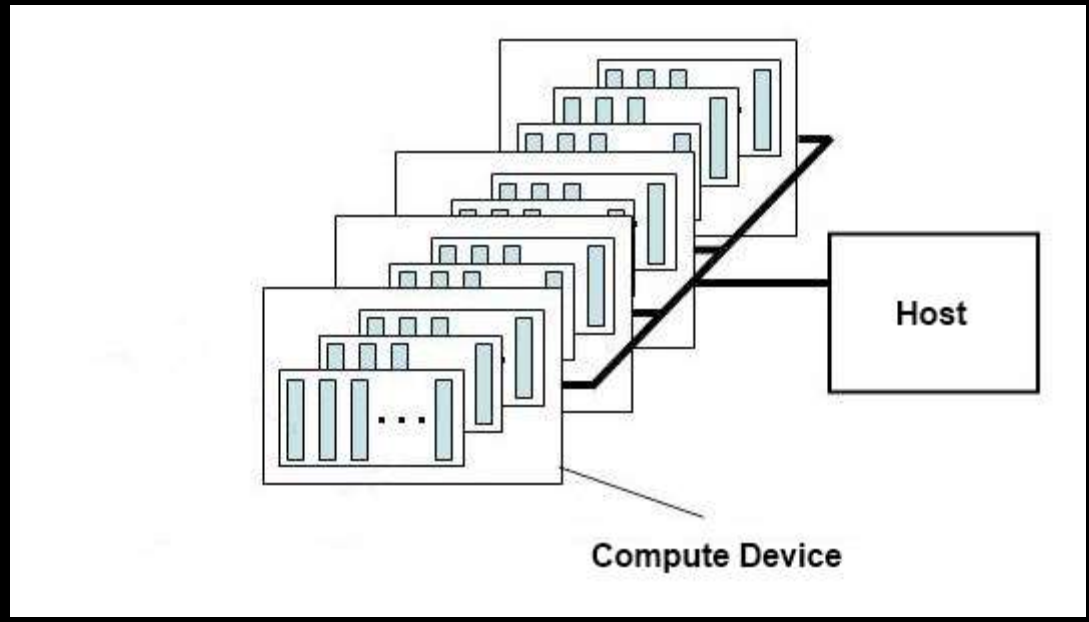


WHAT IS OPENCL?

AMD

Platform model

- The basic separation of a host from a set of devices
  - Multiple devices individually versioned
  - Multiple vendor runtimes accessible
- Designed to ensure a degree of backward compatibility
  - Future proofing of applications built on top of OpenCL
  - As long as the right queries are used to perform version checks!

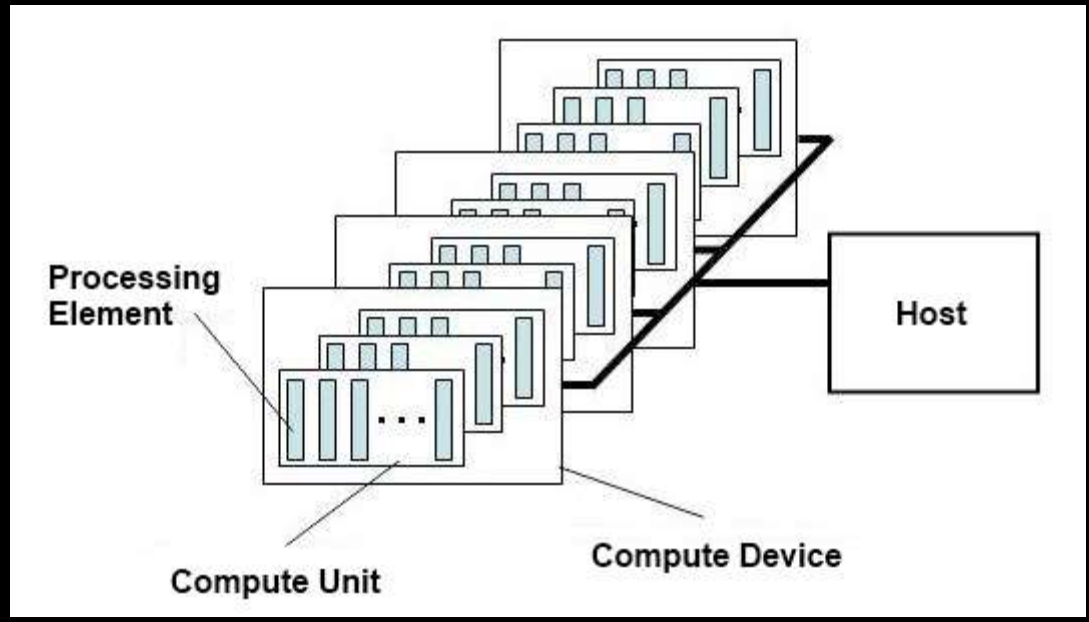


WHAT IS OPENCL?

AMD

Platform model

- Specifies the interpretation we must apply to a device
- Devices consist of abstract computational elements
  - Compute units
  - Processing elements
- Compute units generally map to larger structural entities with caches
  - Cores, really
- Processing elements often map to SIMD lanes
  - The standard specifies that SIMD execution on processing elements is valid

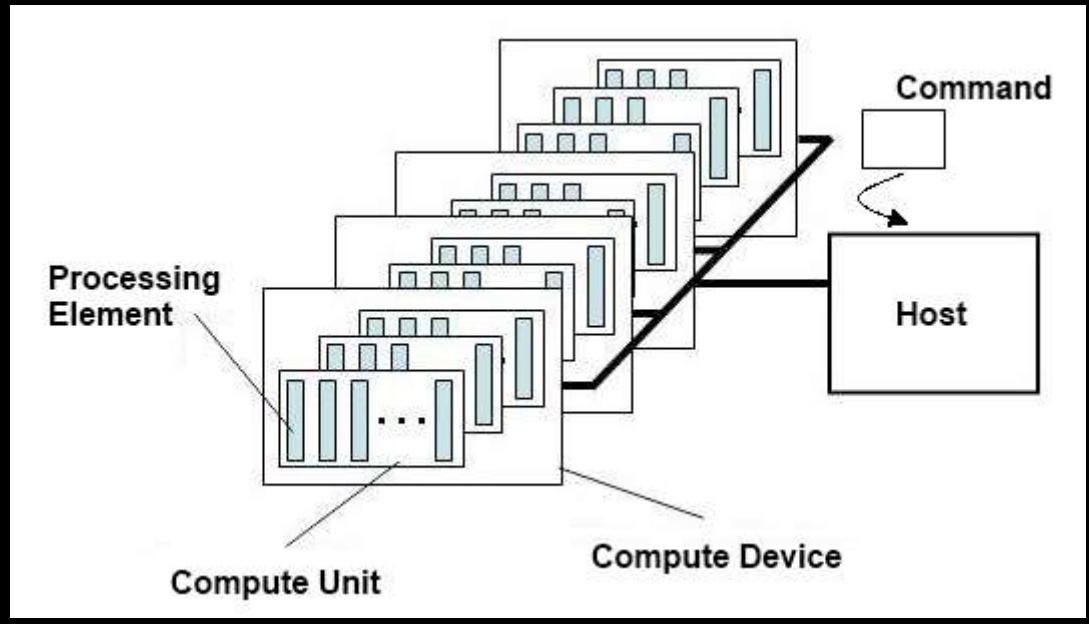


WHAT IS OPENCL?

AMD

Platform model

- Work is abstracted as a command
  - Commands follow a specific execution model
  - Communication between commands follow a specific memory model
- Commands as issued by the host
  - To a specific queue
  - The queue is associated with a given device
  - There is no explicit control over compute units

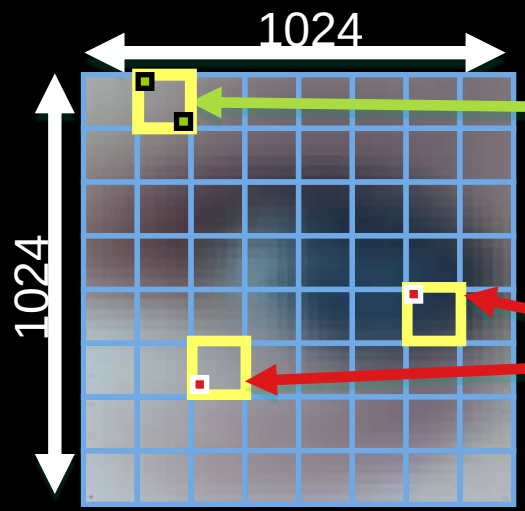


WHAT IS OPENCL?

AMD

Execution model – device side

- Slightly hierarchical data-parallel execution
  - Work-items in work-groups
- Work groups execute completely independently
- Within a work-group work-items may communicate and synchronize
  - Using barrier synchronization primitives
- Work-items within a work-group must thus, in the presence of barriers:
  - execution concurrently
  - make forward progress
- Note that work-items need ONLY be concurrent in the presence of barriers



Synchronization between work-items possible only within workgroups:
barriers and memory fences

Cannot synchronize outside of a workgroup

WHAT IS OPENCL?

AMD

Execution model – host side

Command queues

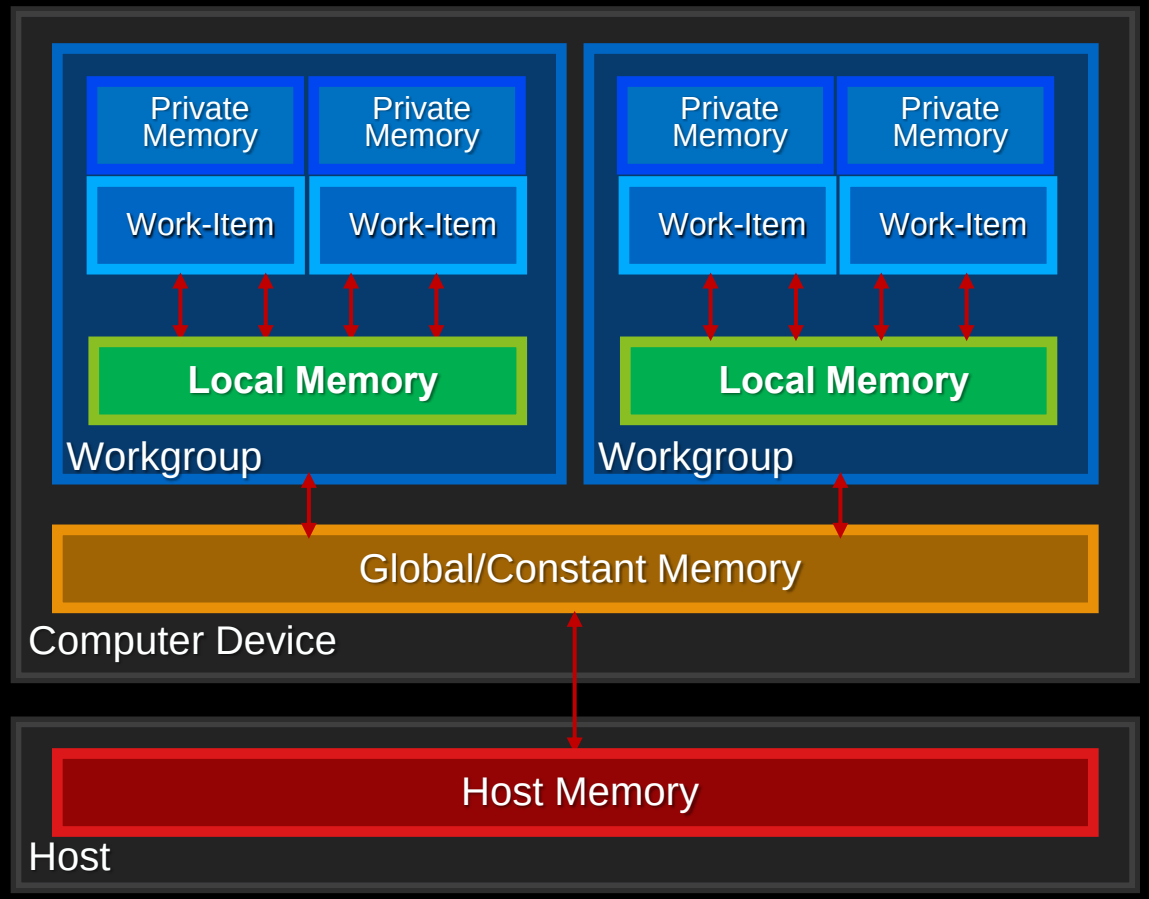
- Command queues allow us to manage data and perform computations on a given device
- Can be in-order or out-of-order (within certain limits)
- Can only be filled from the host side
- While out-of-order queues may allow for concurrent command execution, they do not require it

WHAT IS OPENCL?

AMD

Memory model

- Structured
  - Data is contained both in host-side buffers and in device-side address spaces
  - Implementations can use more efficient access modes the more information they have about the data
- Intentionally weak
  - Provides maximum portability to a range of architectures
  - Allows architectures to optimise location of data
- Memory consistency is defined ONLY at work-group level and at synchronization points



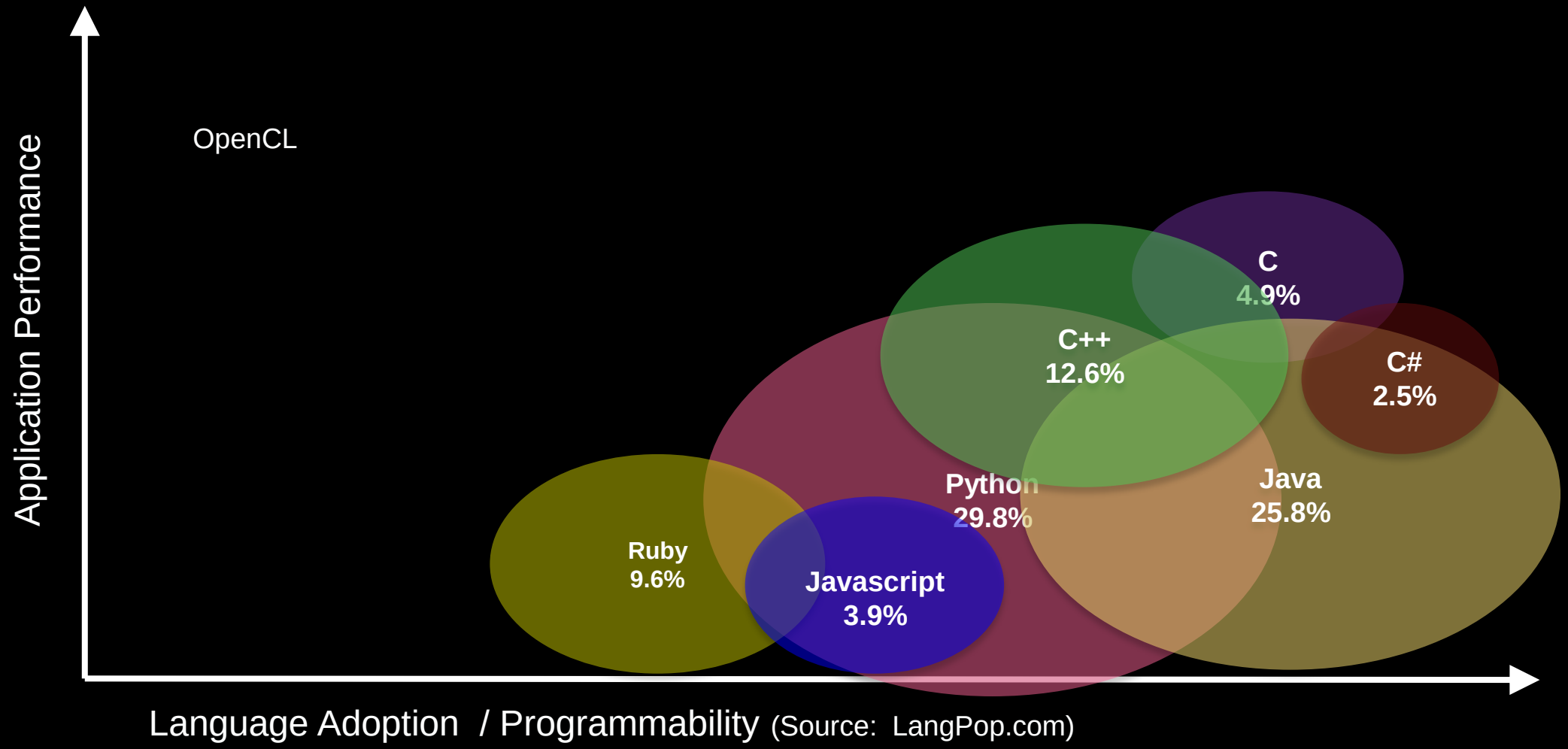
WHAT IS OPENCL?

AMD

Programming model

- OpenCL's programming model comprises two sides:
  - The host API
  - The OpenCL C device programming language
- The host API is a standard C API that exposes the required functionality of the OpenCL standard
- The OpenCL C programming language is a C99-derived language with embedded C-style address spaces
  - Run-time compiled from a string
  - Represents one instance in an SPMD execution that may be mapped to SIMD
- Abstractly OpenCL defines two models:
  - Data-parallel and task-parallel
  - Data-parallel is the execution model described earlier. Task-parallel is the launch of a single work-item domain

HOWEVER…



Bubble size represents 2013 new project starts
Source: CodeEval.com

AGENDA OR CALL-OUTS

AMD

OpenCL, the state of the art

Building on OpenCL

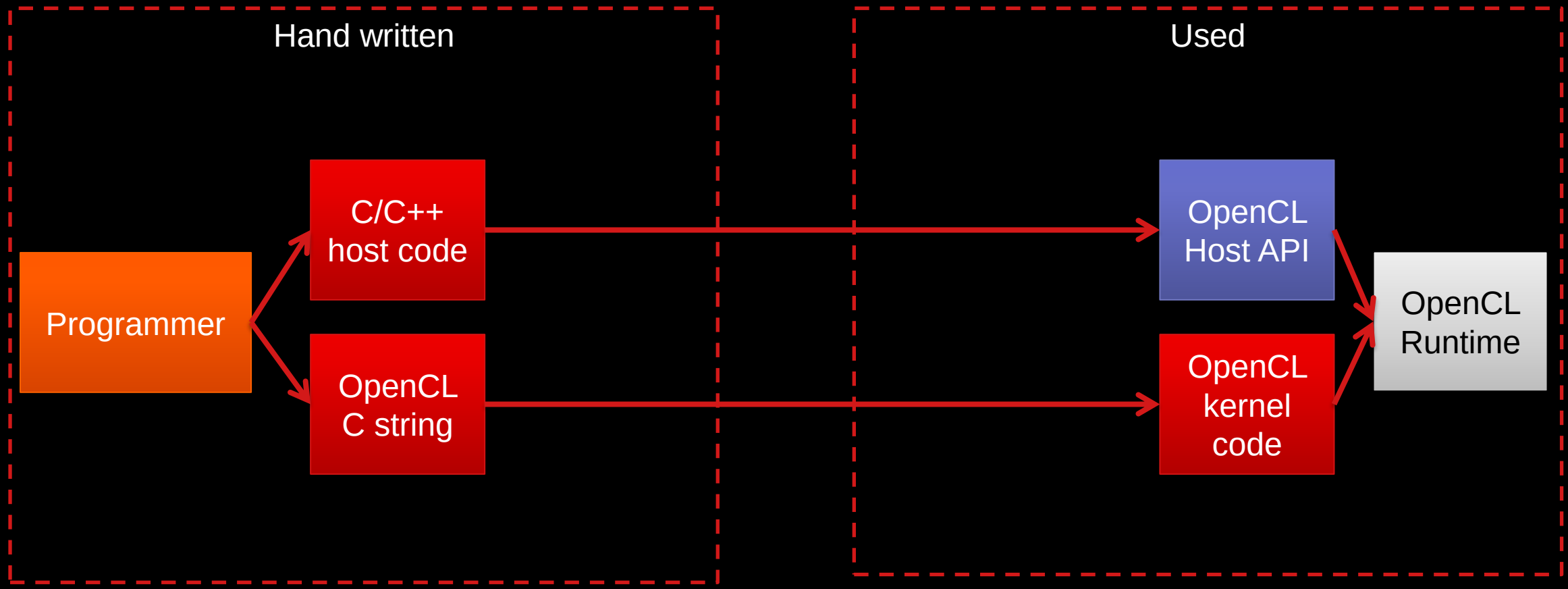
Current limitations

Creating a portable intermediate language

Hiding OpenCL away

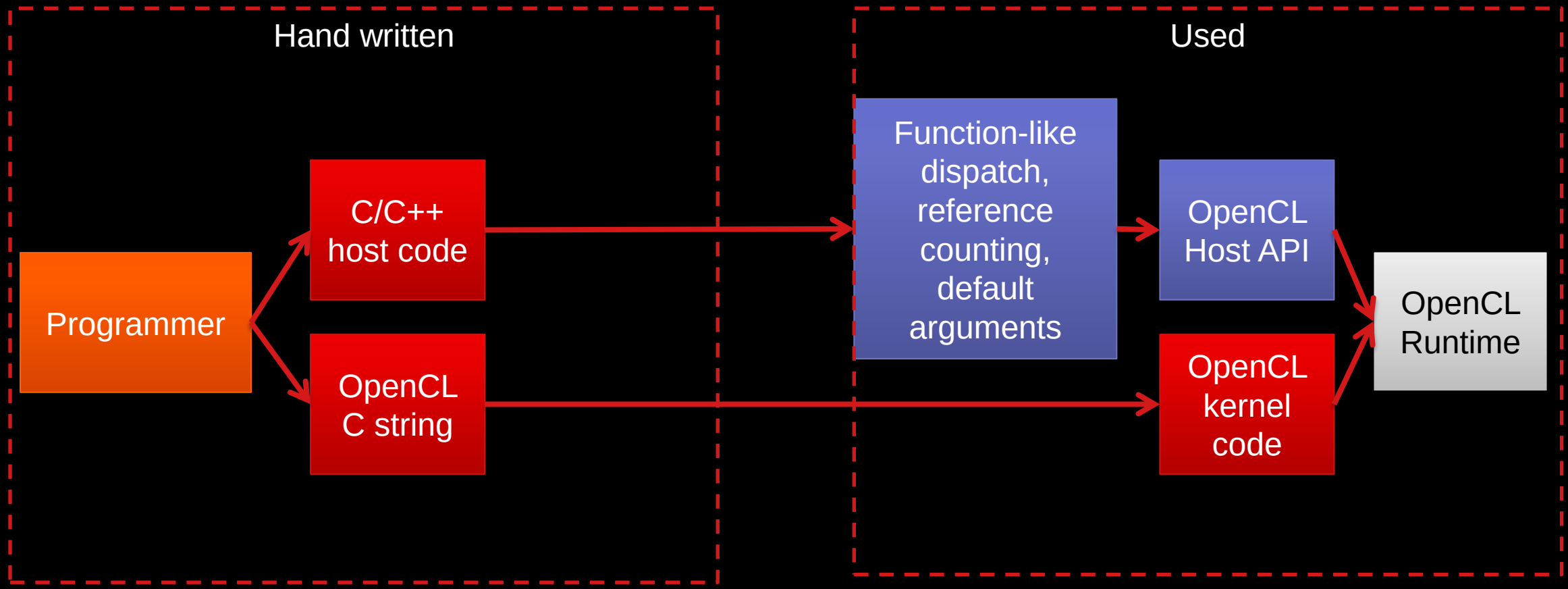
The future of supporting a variety of runtimes

STANDARD LOW-LEVEL OPENCL USE



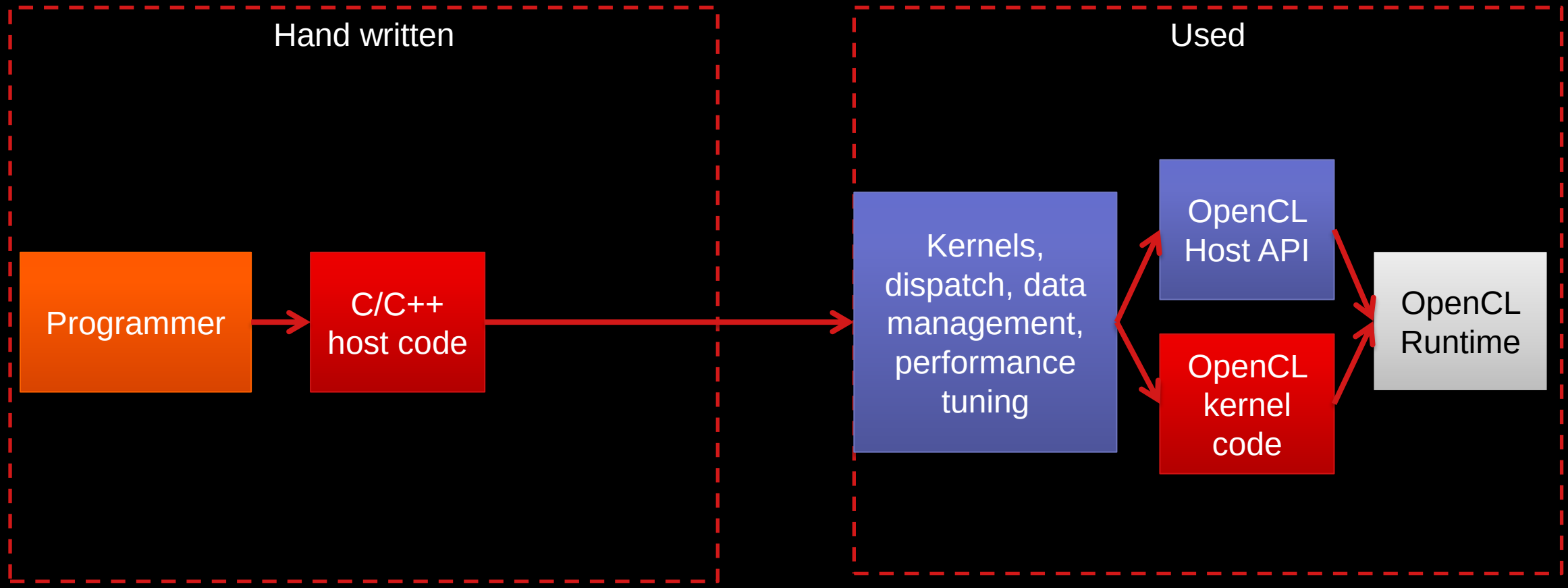
C++ BINDINGS AND RELATED APIS

- Official OpenCL C++ bindings. Various other C++ wrapper interfaces



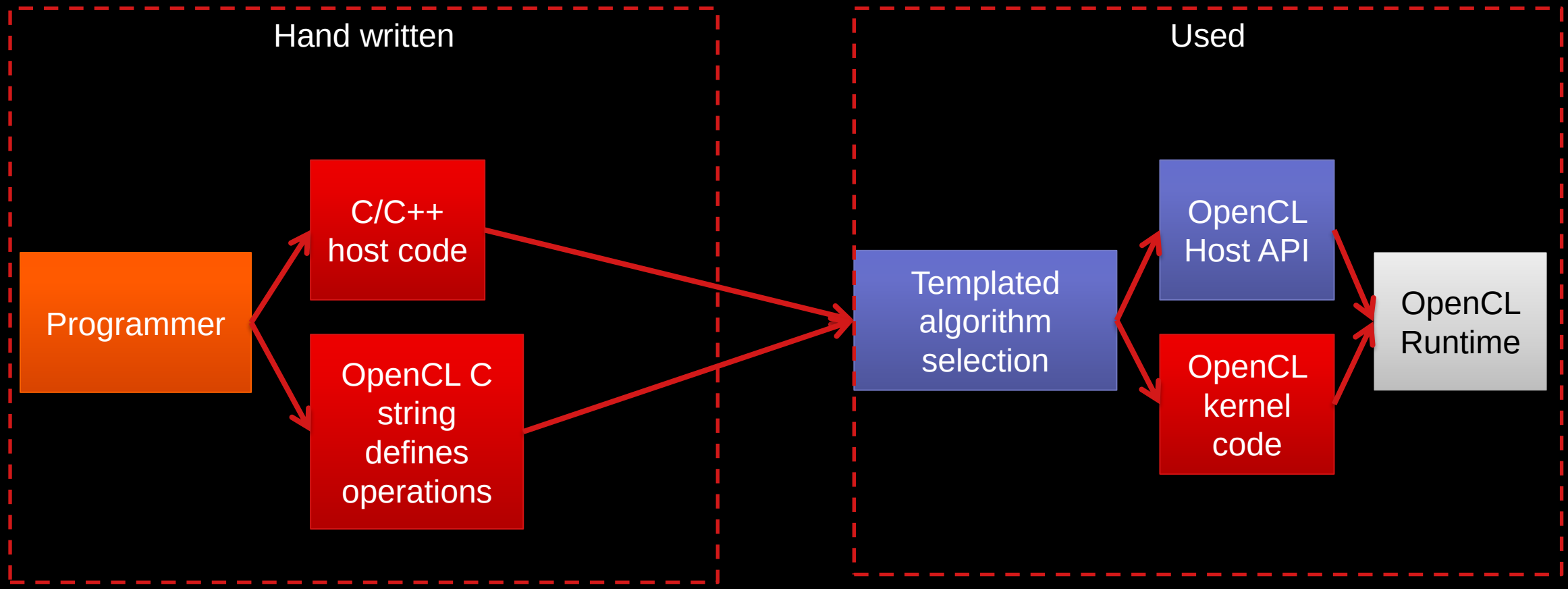
LIBRARIES

- AMD APP BLAS/FFT, ViennaCL etc



BOLT

- C++ template library for OpenCL



BOLT – CLARIFICATION EXAMPLE

AMD

```
BOLT_FUNCTOR(Functor,
  struct Functor {
    float _a;
    Functor(float a) : _a(a) {};
    float operator() (const float &xx, const float &yy) {
      return _a * xx + log(yy) + sqrt(xx);
    };
  };
);
…

Functor func(10.0)
std::transform(A.begin(), A.end(), B.begin(), Z0.begin(), func);
bolt::cl::transform(A.begin(), A.end(), B.begin(), Z1.begin(), func);
```

AGENDA OR CALL-OUTS

AMD

- **OpenCL, the state of the art**
- **Building on OpenCL**
- **Current limitations**
- **Creating a portable intermediate language**
- **Hiding OpenCL away**
- **The future of supporting a variety of runtimes**

LACK OF C++

AMD

- Static C++ features
  - Generic kernels/templated operations
  - A sophisticated type system
  - Overloading
  - Subclassing
- Dynamic C++ features
  - Virtual functions/function pointers
  - Exceptions
- It’s true that AMD has a static C++ kernel language
  - In the long term, is a runtime-compiled C++ kernel language the right way to go?
  - What do people really want from C++?

LACK OF SINGLE-SOURCE

AMD

- The real use for C++?
  - Passing types across kernel boundaries
  - Templating device code from host code
- Cleaner, integrated models for new programmers
  - Simple dispatch APIs
  - Wrapper libraries can provide a lot of this
- Type safety of the kernel dispatch mechanism
  - CLU is an approach for fixing this
- Is this something that is worth standardising, or is something that is worth supporting?
  - Or something in between.

HOST-DRIVEN EXECUTION MODEL

- Host->Device->Host turnaround time can be slow
  - When the device needs to generate more work this cycle can kill performance
- The fixed execution hierarchy cleanly abstracts a wide range of machines
  - It matches none well
  - Consider that many of the devices use vector execution, but this is poorly abstracted
- No well-defined concurrency between workgroups

WEAK MEMORY MODEL

- Like C before C11, OpenCL C has a very weakly defined memory model
  - Weak in terms of consistency of operations
  - Weak in terms of consistency of implementations
- The points at which memory is synchronized vary from one implementation to another
  - Atomic operations may flush the cache or only ensure that the specific atomic operation is visible
  - Acquiring a lock may fences memory operations between the acquire and release of the lock but other implementations may allow considerable flexibility in memory reorderings
- In essence, OpenCL is not designed for communication between workgroups
  - Therefore any such behaviour is implementation-defined
  - This makes it hard to build on

ISSUES OF VECTOR EXECUTION AND FORWARD PROGRESS

AMD

- There is a lack of guarantees about forward progress in GPU schedulers
- Worse, still, the treatment of ‘SIMT’ has serious correctness issues in the presence of synchronization
  - Even if we fix the memory model
- As a trivial example:

```
while(!acquire_lock(&l)) {}
// Do some work
release_lock(l);
```

- If two work-items are passing through the block simultaneously
  - One acquires the lock
  - The others are spinning in the while loop
  - The acquirer can not progress, do the work and release
- Deadlock through SIMD! (This is why SIMT is not threads: it’s a software abstraction on top of threads)

OPENCL C AS THE ONLY ENTRY POINT DOESN'T SCALE

AMD

- Outputting C is not a perfect compiler solution
  - Therefore using OpenCL C as an intermediate representation is imperfect
- Many developers don't want to runtime compile OpenCL C
  - Delays error reporting until very late in the process
  - Requires storing of source code in the final application – poor IP protection
- That's not to say OpenCL C doesn't have its place
  - Runtime compilation has great value under some circumstances
  - A low-level entry point to the system is vital or experimentation and tuning

AGENDA OR CALL-OUTS

AMD

OpenCL, the state of the art

Building on OpenCL

Current limitations

Creating a portable intermediate language

Hiding OpenCL away

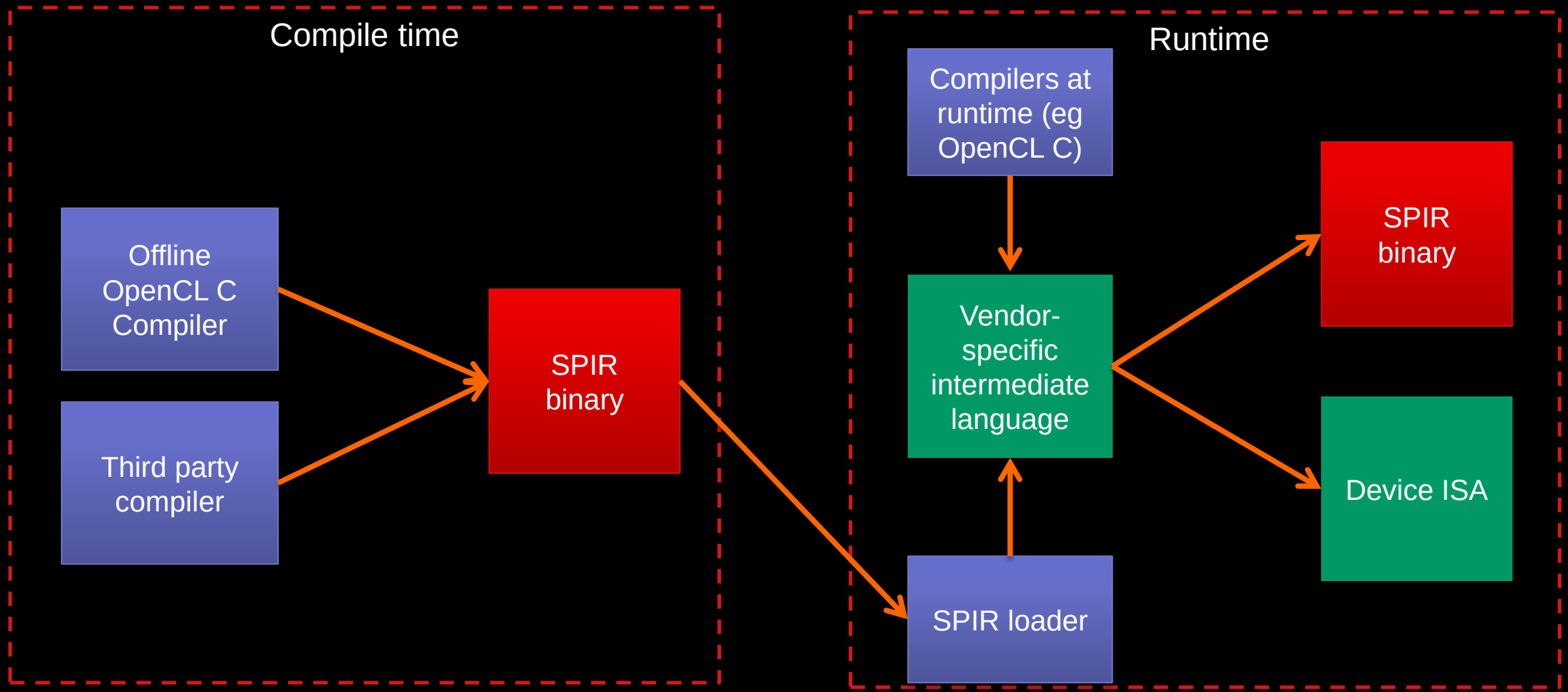
The future of supporting a variety of runtimes

ENTRY POINTS: STANDARD PORTABLE INTERMEDIATE REPRESENTATION

AMD

- A proposed binary representation for OpenCL C programs
- Designed to be portable across OpenCL vendors
  - One OpenCL vendor's toolchain might be used to generate the intermediate representation
  - Another vendor's toolchain would consume it
- Possibly more importantly allows third party tool chains to generate a well-defined target
  - Innovation on top of the OpenCL framework
  - New programming models may target SPIR
  - The generated SPIR and host API calls may then execute on the underlying OpenCL runtime
  - Possible efficiency gains by removing the IR->C->IR transformation
- Defines the IR and rules for consumption
  - Generation is undefined: if it were not, innovation would be stifled

STANDARD PORTABLE INTERMEDIATE REPRESENTATION



PURPOSE

AMD

Enable third party compilers

- Expand the set of languages that target OpenCL runtimes
- Improve the ease of use of the standard
- Enable innovation

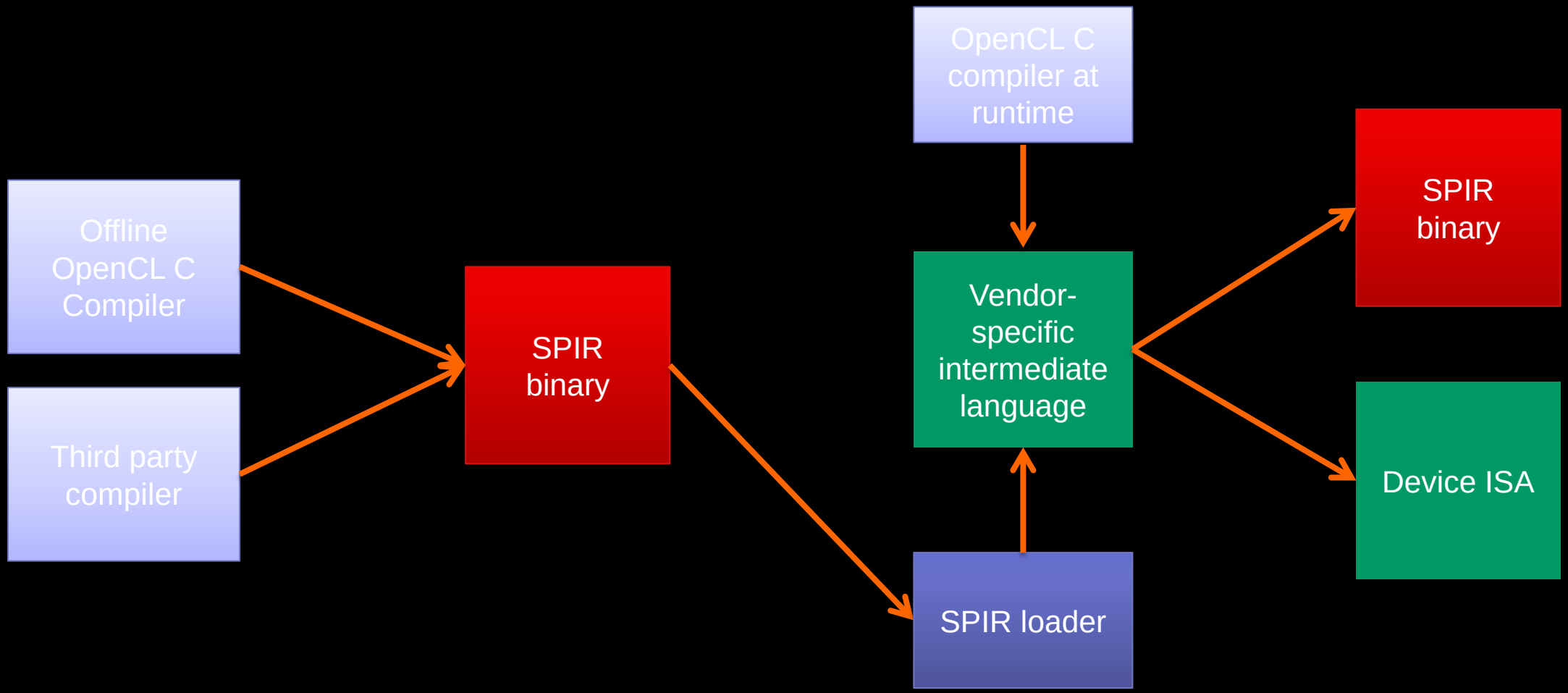
Remove the need to release plain text kernel code

- OpenCL C's plain text kernels must be stored somewhere
  - A common ISV concern
  - Mitigated only by compiling binaries for multiple targets offline: no scaling
- SPIR is a binary representation of the kernel
  - Minimal optimisation so the runtime can still optimise for the target
  - Some degree of obfuscation

ONLY DEFINE CONSUMPTION

Define the intermediate representation and its mapping to the concepts in the OpenCL standard.

Leave SPIR production to innovative ISVs (and we as OpenCL runtime vendors can support it too)



BASED ON LLVM-IR

AMD

- SPIR is based on LLVM-IR
  - The current public provisional specification defines SPIR in terms of LLVM 3.1
- LLVM-IR is already used by tool chains, so was close to being suitable
  - The goal was to make as small a set of modifications as possible to the standard
  - Achieve portability without major change
  - Maintain an easy path to future compatibility
  - Track LLVM's built-in upgrade path to future versions of the infrastructure
- To be portable SPIR has to define the mapping from LLVM-IR types to OpenCL C types
- Add calling conventions and a new build target

STATUS OF SPIR

- Provisional specification released in November 2012
  - Aim to collect feedback from the community
- There is an open source SPIR producer under development as part of Clang.
- The development of SPIR consumption is underway by AMD and other OpenCL vendors.
- However… it is still limited by OpenCL's underlying feature set and models
- As OpenCL evolves, SPIR will evolve

AGENDA OR CALL-OUTS

AMD

OpenCL, the state of the art

Building on OpenCL

Current limitations

Creating a portable intermediate language

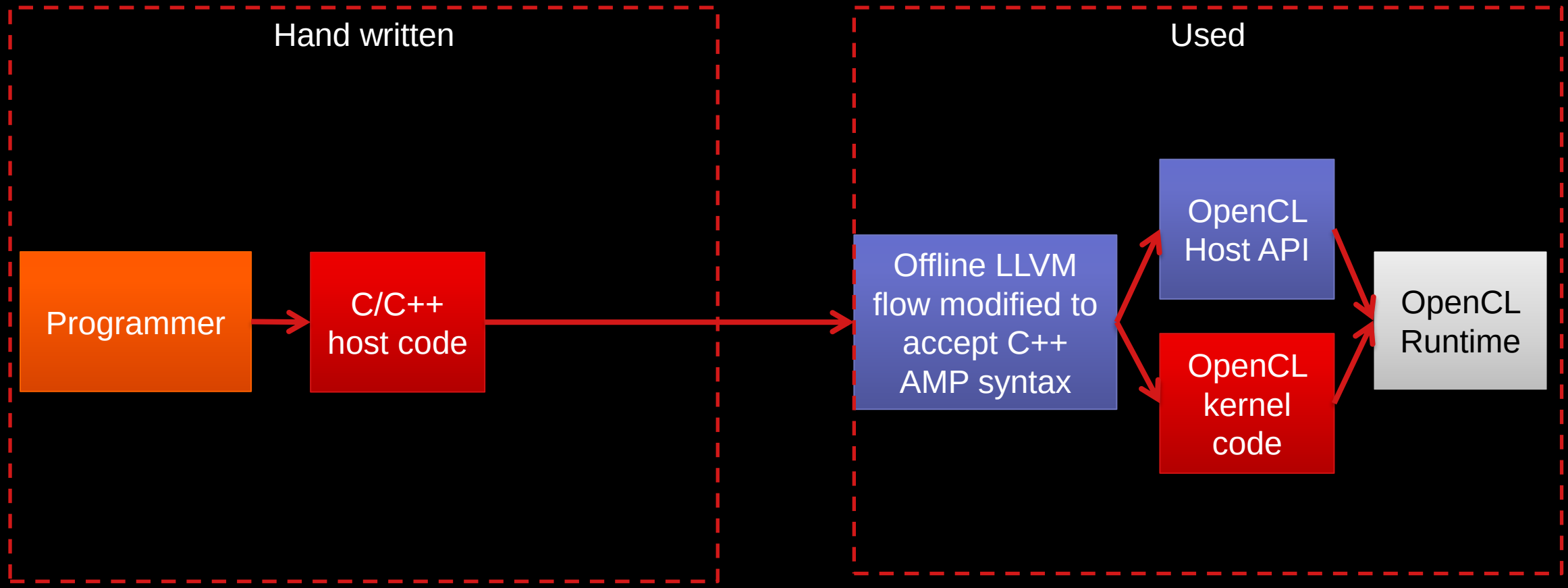
Hiding OpenCL away

The future of supporting a variety of runtimes

INTEL®'S SHEVLIN PARK PROJECT

AMD

- C++AMP-on-OpenCL: a single source compiler



http://llvm.org/devmtg/2012-11/Sharlet-ShevlinPark.pdf

APARAPI

AMD

- Runtime offload of Java to the GPU
  - Currently compiles down to OpenCL C
- Uses standard Java types
- Reflection-based compilation
  - Class files trigger generation of GPU code by analysing their own Java bytecode
  - No language syntax additions but similar benefits
- Some scope limitations
  - The compiler cannot see code before and after the dispatch call
  - Extra copy overhead as a result of the lack of global visibility
- Capabilities are restricted by its OpenCL target and by the goal of creating a simple path for data-parallel loop offload for Java

APARAPI

AMD

- To run the following computation on the GPU:

```
final float inA[] = .... // get a float array of data from somewhere
final float inB[] = .... // get a float array of data from somewhere (inA.length==inB.length)
final float result = new float[inA.length];

for (int i=0; i<array.length; i++){
    result[i]=intA[i]+inB[i];
}
```

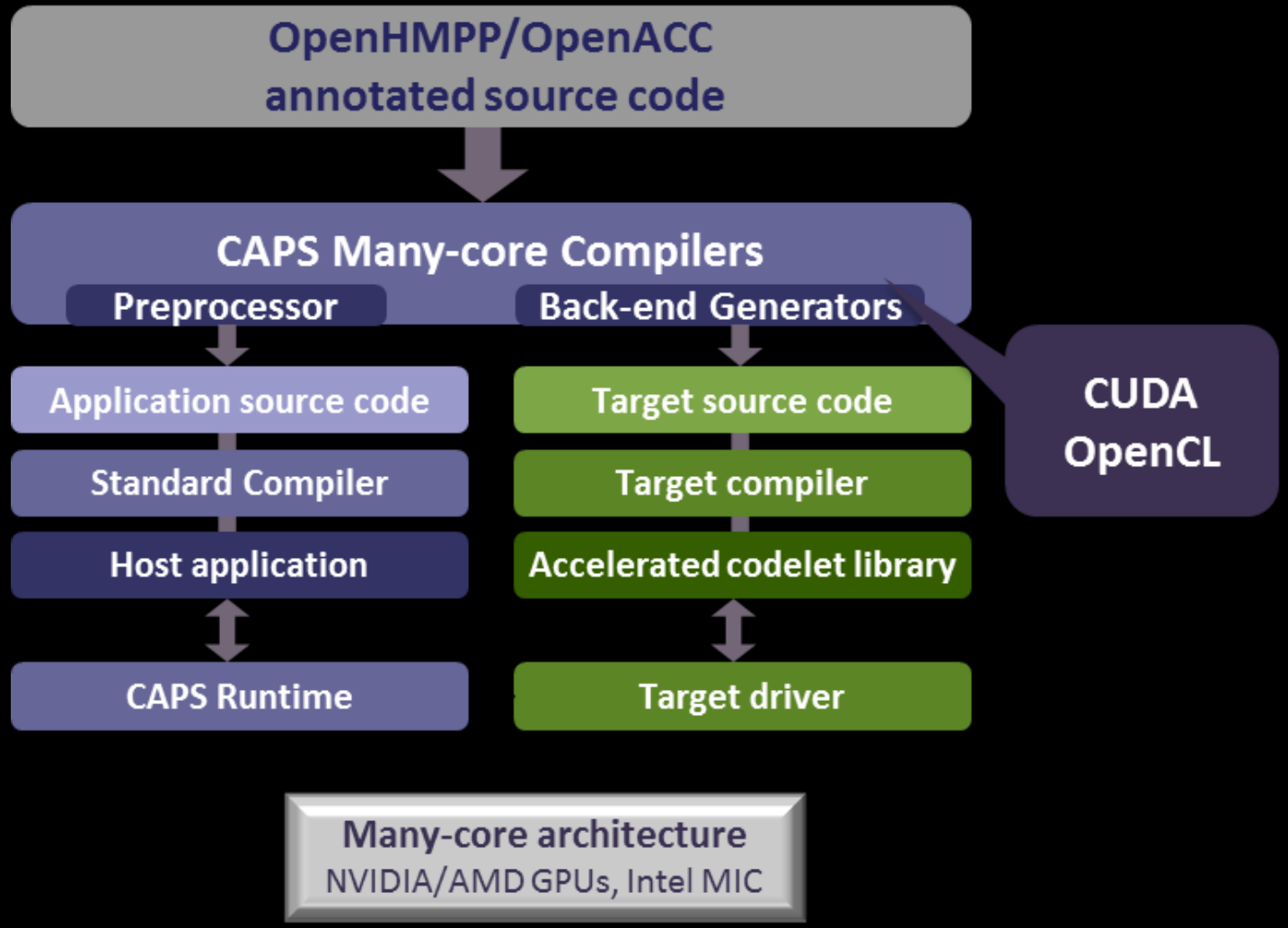
- We can refactor the sequential loop to the following form:

```
Kernel kernel = new Kernel(){
   @Override public void run(){
      int i= getGlobalId();
      result[i]=intA[i]+inB[i];
   }
};
Range range = Range.create(result.length);
kernel.execute(range);
```

Kernel compilation will be triggered at this point

- The API will become simpler with the addition of lambdas in the near future

CAPS OPENACC/OPENHMPP MODELS




```
#pragma acc loop gang(64)
for (int i = 1; i < M - 1; ++i) {
   #pragma acc loop worker(128)
   for (int j = 1; j < N - 1; ++j) {
      B[i][j] =
         c11 * A[i - 1][j - 1] +
         c12 * A[i + 0][j - 1] +
         c13 * A[i + 1][j - 1] +
         c21 * A[i - 1][j + 0] +
         c22 * A[i + 0][j + 0] +
         c23 * A[i + 1][j + 0] +
         c31 * A[i - 1][j + 1] +
         c32 * A[i + 0][j + 1] +
         c33 * A[i + 1][j + 1];
   }
}
```

Diagram and code example from www.caps-enterprise.com

IS THIS NOT ENOUGH?

AMD

- All of the models basically follow the same OpenCL execution model
  - Big map operations
  - No explicit concurrency
  - No meaningful communication
- The question is what we need to add to support a wider set of models
  - Programming models that CPU programmers are used to using
  - Programming models that scale

AGENDA OR CALL-OUTS

AMD

OpenCL, the state of the art

Building on OpenCL

Current limitations

Creating a portable intermediate language

Hiding OpenCL away

The future of supporting a variety of runtimes

THE CURRENT STATE OF THE ART

AMD

- I can't talk about the next version of OpenCL
  - There are many other people in this room who also can't talk about the next version of OpenCL…
- However, I can talk a little about HSA, on top of which AMD will be implementing OpenCL
  - Although this is also a standard in development
- HSA is an architected layer specifying:
  - A memory model for interacting components in the platform with full shared virtual memory capabilities
  - A queue specification in user-space such that any device in the platform can write to another device's queues in a standardised way
  - A portable intermediate language (HSAIL) that is intended to be at a lower level than SPIR such that most optimisations have been performed in the high level compiler and the runtime compilation time is small.
- The definition acts at a different layer from OpenCL
  - It is lower level
  - It uses architected shared-memory interfaces

HSA INTERMEDIATE LAYER - HSAIL

AMD

- Explicitly parallel
  - Designed for data parallel programming
- Support for exceptions, virtual functions, and other high level language features
- Syscall methods
  - GPU code can call directly to system services, IO, printf, etc

HSA MEMORY MODEL

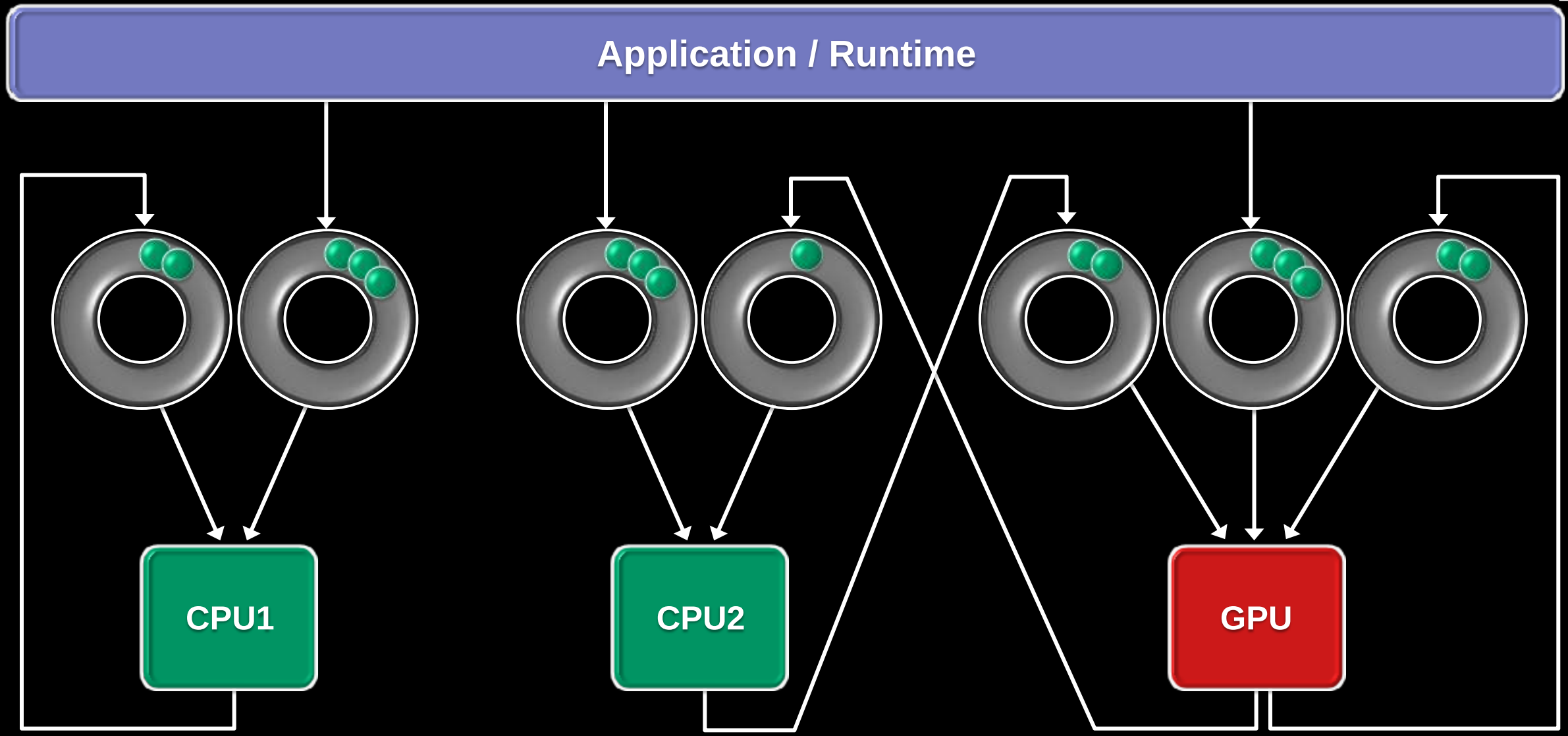
- Designed to be compatible with C++11, Java and .NET Memory Models
- Relaxed consistency memory model for parallel compute performance
- Loads and stores can be re-ordered by the finalizer
- Visibility controlled by:
  - Load.Acquire, Load.Dep, Store.Release
  - Barriers
- A strict memory model allows us to reason about correctness of communicating processes
- It also provides a stronger basis for academic research
  - A stronger underlying model offers more scope for innovation on top of the underlying model.

HSA ENABLES DEVICE TO DEVICE ENQUEUE

AMD

- HSA queues sit in user memory so both host and device can write new commands to them
- Popular pattern for task- and data-parallel programming on SMP systems today
- Characterized by:
  - A work queue per core
  - Runtime library that divides large loops into tasks and distributes to queues
  - A work stealing runtime that keeps the system balanced
- HSA is designed to extend this pattern to run on heterogeneous systems

FUTURE COMMAND AND DISPATCH CPU <-> GPU



DO WE NEED CONTEXT SWITCHING?

AMD

- Yes and no…

- For Quality of service, we need context switching
  - If we don't have firm control of software running on the machine
  - If we need to guarantee that software doesn't harm other software – similar arguments to those used for VM

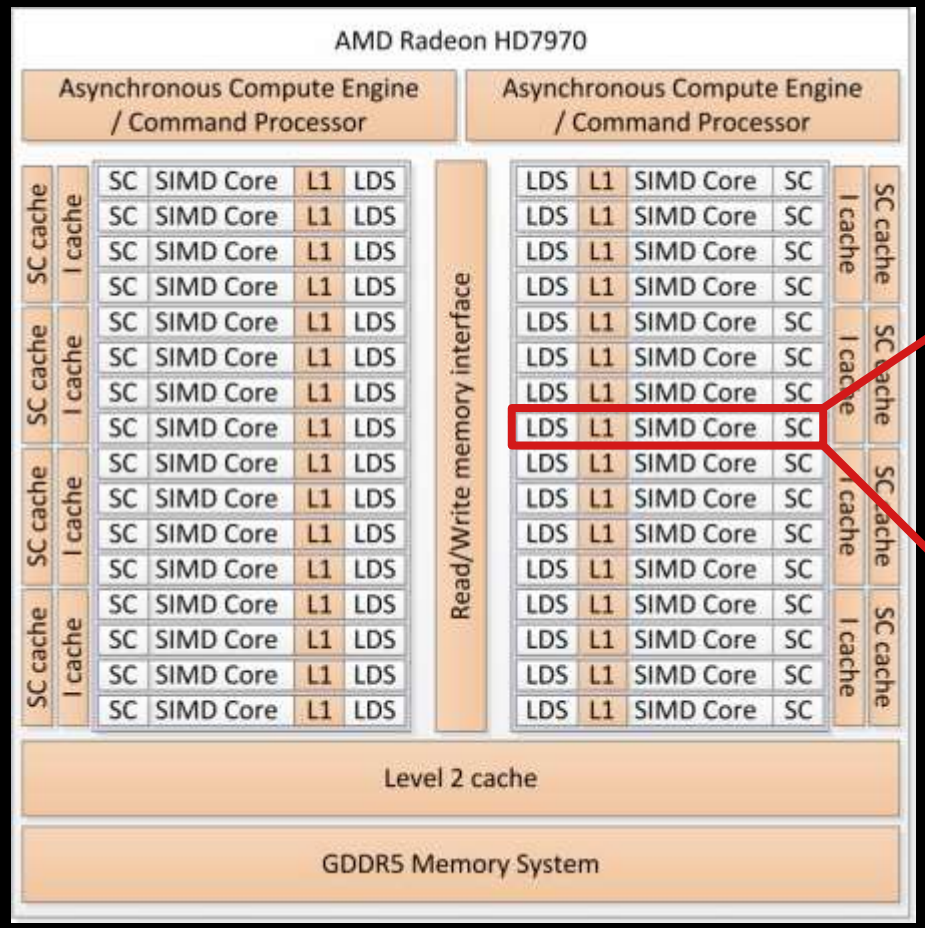
- For other, simpler tasks, for fine grained switching, maybe context switching isn't the right way go go
  - How much of either true context switching or support for other forms do we need to add to OpenCL?

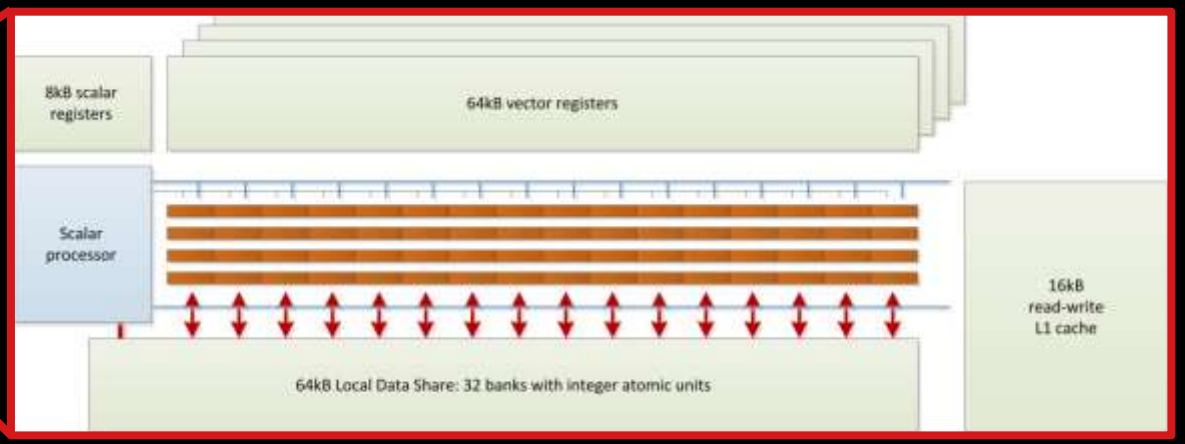
AMD'S HSA-DRIVEN FEATURE ROADMAP

AMD

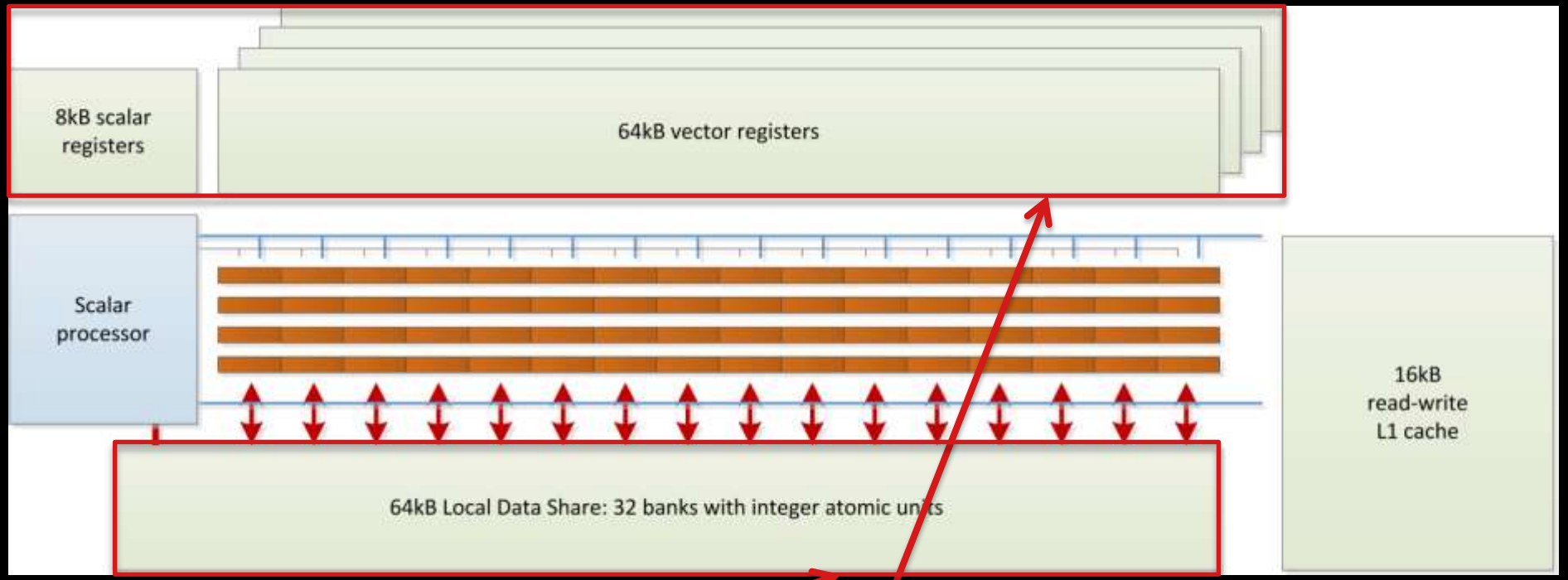
Physical Integration	Optimized Platforms	Architectural Integration	System Integration
Integrate CPU & GPU in silicon	GPU Compute C++ support	Unified Address Space for CPU and GPU	GPU compute context switch
Unified Memory Controller	User mode schedulng	GPU uses pageable system memory via CPU pointers	GPU graphics pre-emption
Common Manufacturing Technology	Bi-Directional Power Mgmt between CPU and GPU	Fully coherent memory between CPU & GPU	Quality of Service
			Extend to Discrete GPU

TAKE A SPECIFIC GPU – THE AMD RADEON™ HD7970



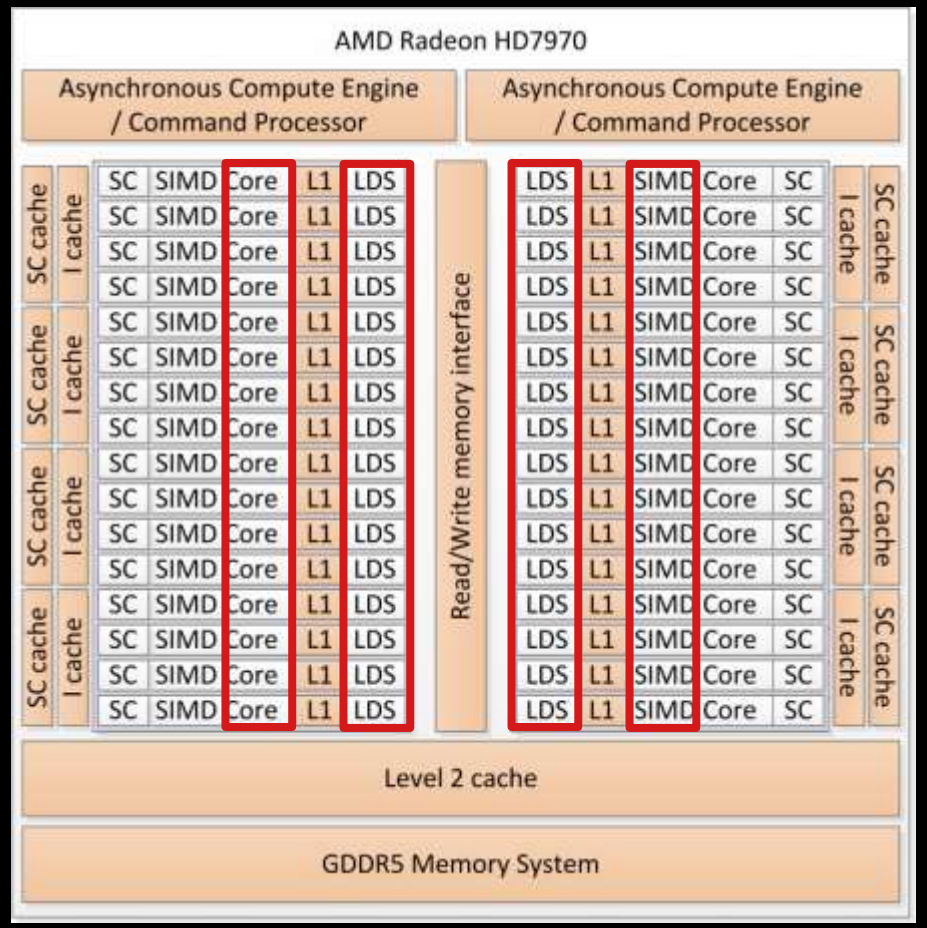


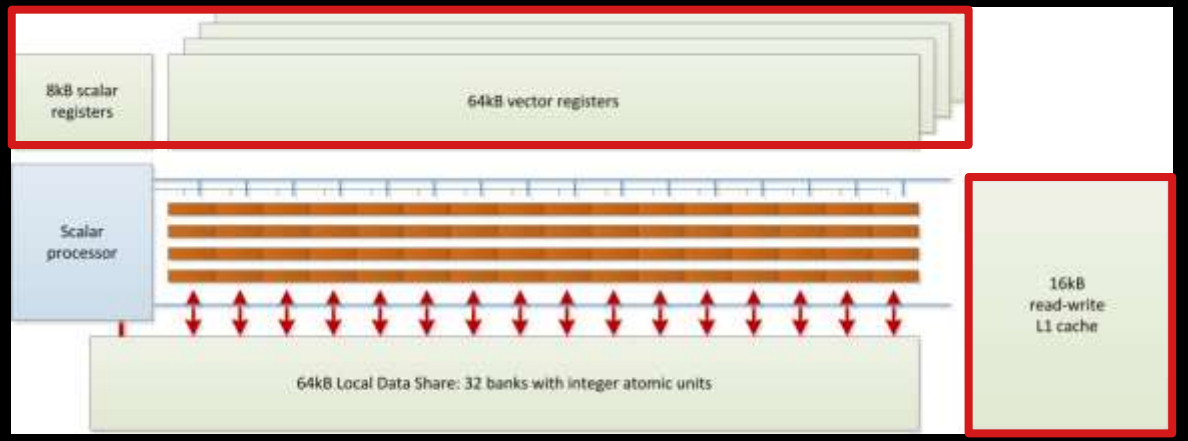
THE SIMD CORE



- The scalar core manages a large number of threads
  - Each thread requires its set of vector registers
  - Significant register state for both scalar and vector storage
  - 10 waves per SIMD, 40 waves per CU (core), 2560 work items per CU, 81920 work items on the AMD HD7970
- 64kB of LDS, OpenCL's "`__local`"

TAKE A SPECIFIC GPU – THE AMD RADEON™ HD7970



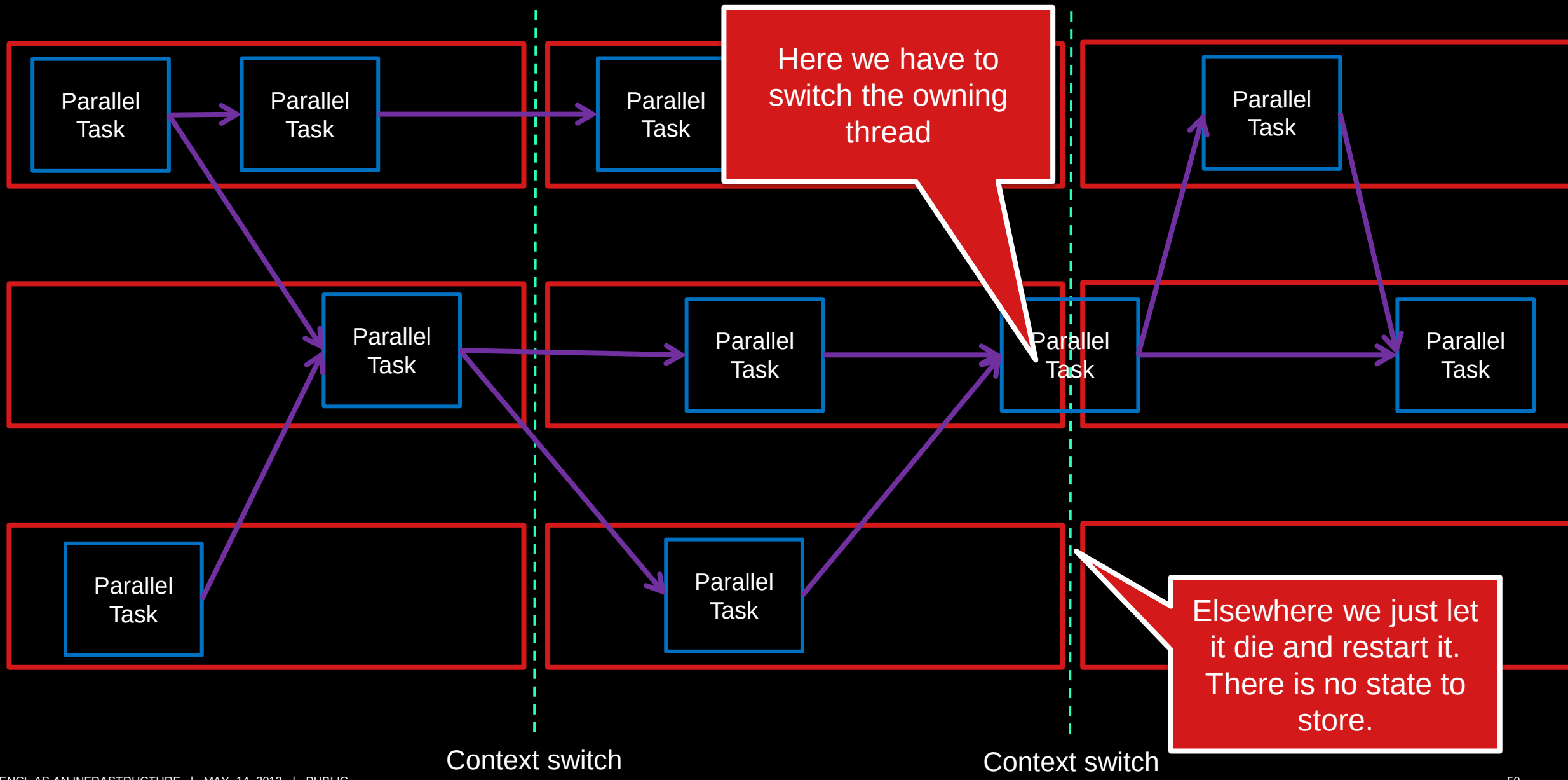


TOO SIMPLE?

AMD

- Look at fine-grained tasking systems on the CPU
  - Cilk, ConcRT, TBB
- Various subtly different execution model:
  - Cooperatively switches between tasks
  - Supports continuations, cooperative locks, passing of exceptions
  - Context switching underlying fabric
  - We can’t implement them all directly in OpenCL
- We overlay those fine-grained tasks on long running threads
  - Those threads would end up context switching!
  - CPU-like architectures without hardware dispatch already implement OpenCL this way
- Then use the same underlying task architecture to portably support efficient reductions
  - We don’t actually want to launch a huge number of workgroups
  - We want to launch as few workgroups as possible to keep the machine occupied and carry a reduction variable

CONTEXT SWITCHING TASK MANAGEMENT THREADS



CAN WE ADAPT OPENCL'S EXECUTION MODEL IN THIS DIRECTION?

AMD

- We want to support a wide range of models sitting on top of OpenCL
- We do not know how all those models work
  - How can we? We have to allow runtime and compiler developers to innovate.
- So can we adapt the OpenCL execution model to support such techniques?

THE PATH FORWARD

AMD

Adapt OpenCL's execution model

Offer as many underlying models as possible

Allow programmable control of the execution patterns

Expose OpenCL scheduling in a more flexible manner

Minimise OpenCL's execution model

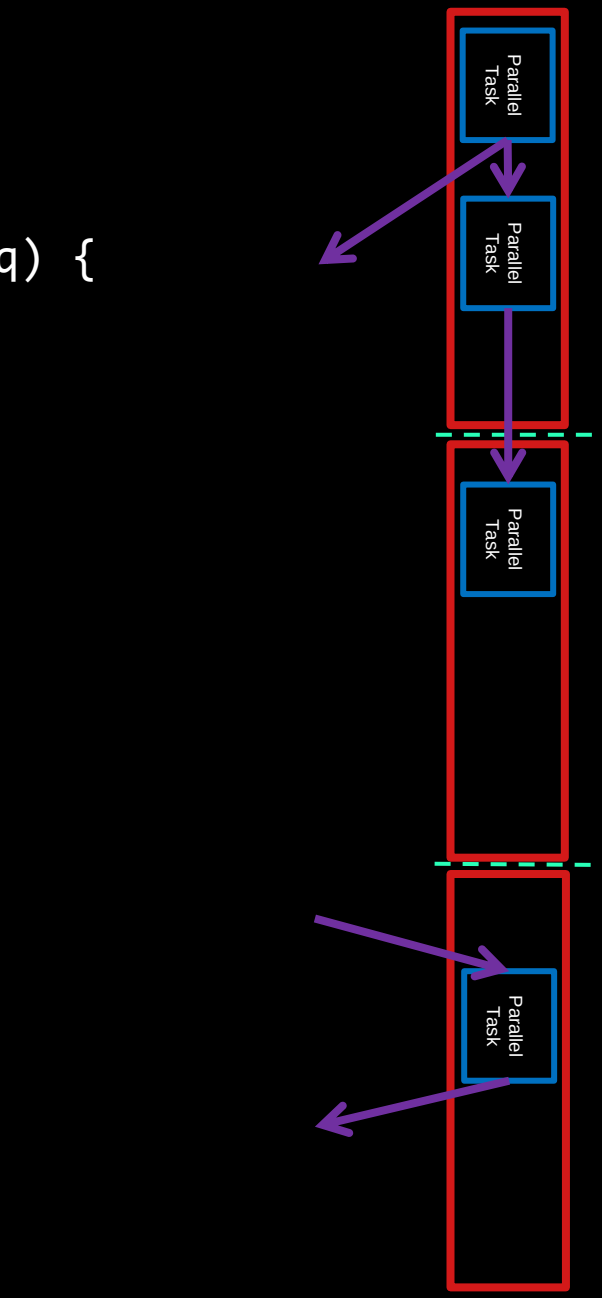
Explicitly support constructs to enable software scheduling on top of the underlying OpenCL schedule

Allow a software scheduler to communicate with the execution model for efficient execution

Enable software to avoid context switching, but offer the benefits context switching would provide

INFRASTRUCTURE SUPPORT

```
kernel managementTask(WorkQueue &wq, Yielder &y, TaskQueue &tq) {
    while(!told to finish) {
        Task t = tq.pop();
        t.run();
        if( y.yield() && get_global_id(0) == 0 ) {
            wq.push(self, NDRange(fill));
            exit();
        }
    }
}
```

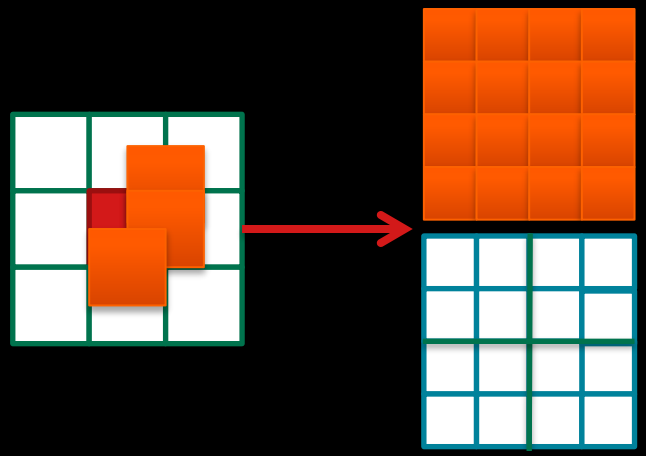


FLATTENING NESTED DATA PARALLEL CODE CAN BE MESSY

AMD

- What do we do when we manually write that in OpenCL?
  - We launch a workgroup for each block
  - Some work items in the workgroup that may or may not match the number of iterations we need to do
  - Each work item will contain the same serial loop
    - Will the compiler realize that this is the same loop across the group?

- We've flattened a clean nested algorithm
  - From a 2D loop + a serial loop + a 2D loop
  - Into a 4D blocked iteration space
  - Spread the serial loop around the blocks
  - It's a messy projection... surely not the best way?



- How far do we want to take the work-group execution model?

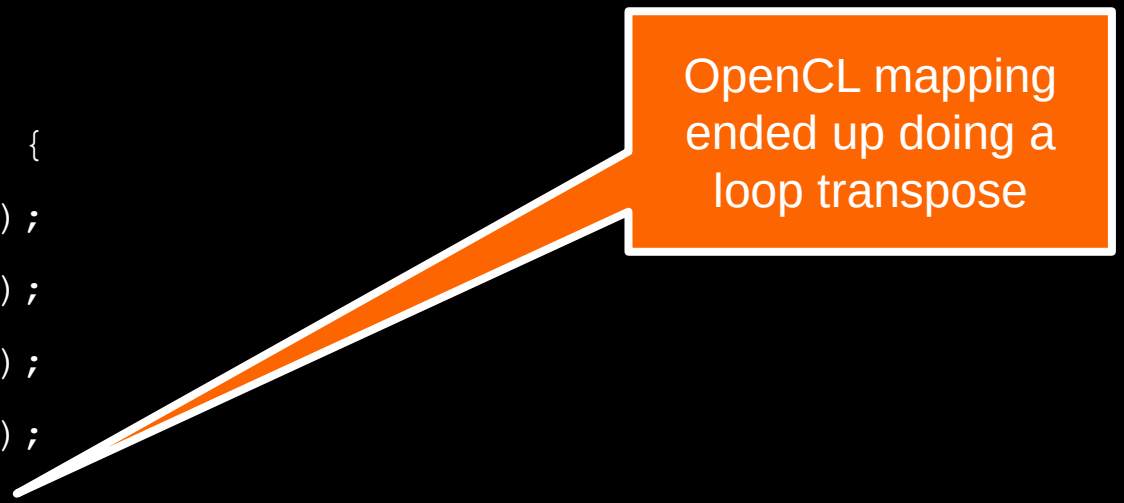
SOMETHING LIKE…

AMD

```
for( int y = 0; y < YMax; ++y ){ // for each macroblock in Y
  for( int x = 0; x < XMax; ++x ) { // for each macroblock in X

    while( not found optimal match ) {

      for( int y2 = 0; y2 < 16; ++y2 ) {
        for( int x2 = 0; x2 < 16; ++x2 ) {
          diff = block(x2 + 16*x, y2 + 16*y) - target(x2 + tx, y2 + ty);
          // use diff… maybe sum of squared differences
        }
      }
    }
  }
}
```

SOMETHING LIKE… KERNELIZATION

```
kernel void motionVector(…) {
    int x = get_global_id(0);
    int y = get_global_id(1);
    int x2 = get_local_id(0);
    int y2 = get_local_id(1);

    while( not found optimal match ) {
        diff = block(x2 + 16*x, y2 + 16*y) – target(x2 + tx, y2 + ty);
        // use diff… maybe sum of squared differences
    }
}
```



ABSTRACT THE WORKGROUP SIZE FROM THE COMPUTATION?

AMD

```
scalar kernel void motionVector(…) {
    int x = get_group_id(0);
    int y = get_group_id(1);
    while( not found optimal match ) {

      parallel_for( int y2 = 0; y2 < 16; ++y2 ) {
        parallel_for( int x2 = 0; x2 < 16; ++x2 ) {
          diff = block(x2 + 16*x, y2 + 16*y) - target(x2 + tx, y2 + ty);
          // use diff… maybe sum of squared differences
        }
      }
    }
  }
}
```

Leave the loop transpose for later in the tool chain.
More readable code.

YIELD AND STRUCTURED PARALLELISM IN THE IR AND RUNTIME

AMD

- Operations such as these can be integrated into SPIR or, more likely, called from SPIR as standard library operations

- The runtime would need functionality to
  - Describe yieldable dispatches
  - Enable dispatch-to-fill; ie to issue workgroups while there is space to issue them, and do so repeatedly until the task terminates
  - Launch in terms of threads (or small groups of threads) rather than work-items

- The IR might also carry abstractions of the parallelism of a workgroup
  - Allow the tool chain to map to vector units as necessary
  - Carry as much information as late in the tool chain as possible
  - Better performance portability

OVERALL GOALS

AMD

Think of OpenCL as an infrastructure

Make decisions based on OpenCL as a compiler/runtime target, not a programmer target

Make OpenCL a simple, efficient target for implementing complicated functionality rather than embedding complicated functionality inside OpenCL

We need to provide concurrency guarantees, quality of service, reasons for a runtime vendor to believe that task graphs they implement will behave in a manageable way when running on the OpenCL infrastructure

Investigate runtimes that developers use

What functionality do these runtimes support?

What would we need to add to OpenCL to support them?

What should we not do to OpenCL that would complicate the mapping

Ensure that SPIR can represent the upstream models while maintaining enough information to allow portable mapping

Questions?

DISCLAIMER & ATTRIBUTION

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION