LLVM Interpreter

a key component in validation of OpenCL™ compilers

Oleg Maslov, Alexey Bader, Yuri Veselov
Intel
April 30, 2013, The 3rd LLVM European meeting

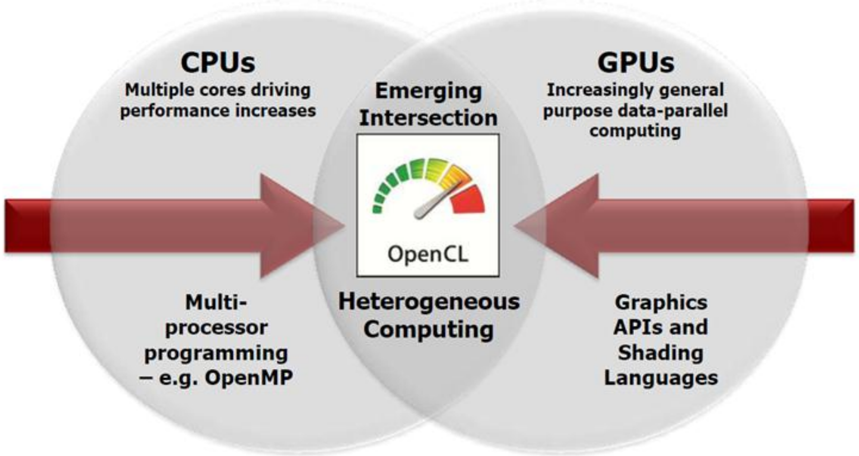
Agenda

- OpenCL™ standard overview
- OpenCL compiler standalone validation tool
- LLVM Interpreter
- Contribution to the LLVM community

Optimization Notice

OpenCL™ Standard

OpenCL™ - Open Standard for Parallel Computing



- Open standard driven by Khronos*
- Royalty-free
- First spec in 2009
- Cross-platform

Diagram based on PDF OpenCL overview available at http://www.khronos.org/opencl/

OpenCL™ Standard Overview

- Portable C code for all architectures
- Derived from ISO C99
  - Few restrictions, e.g. recursion, function pointers
  - Short vector types e.g., float4, short2, int16
  - Built-in functions: math (e.g., sin), geometric, common (e.g., min, clamp)
- Configurable N-dimensional computation domain
- Barriers and memory fences within workgroups
- Extensive list of optimized built-in functions

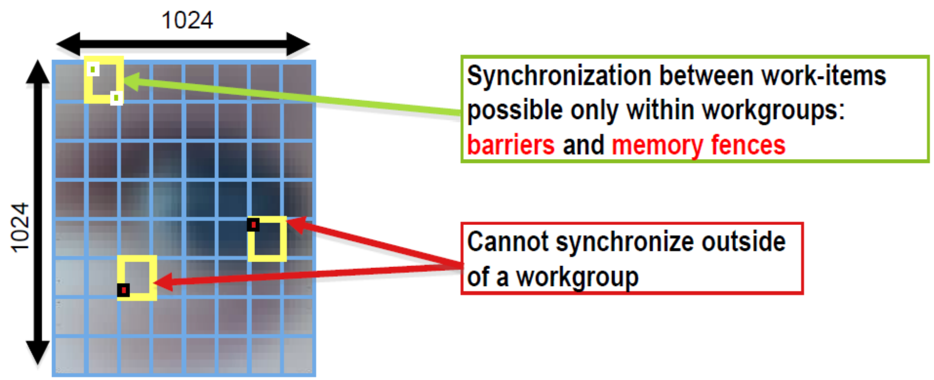


Diagram based on the PDF OpenCL™ overview available at http://www.khronos.org/opencl/

Intel® SDK for OpenCL™ Applications 2013

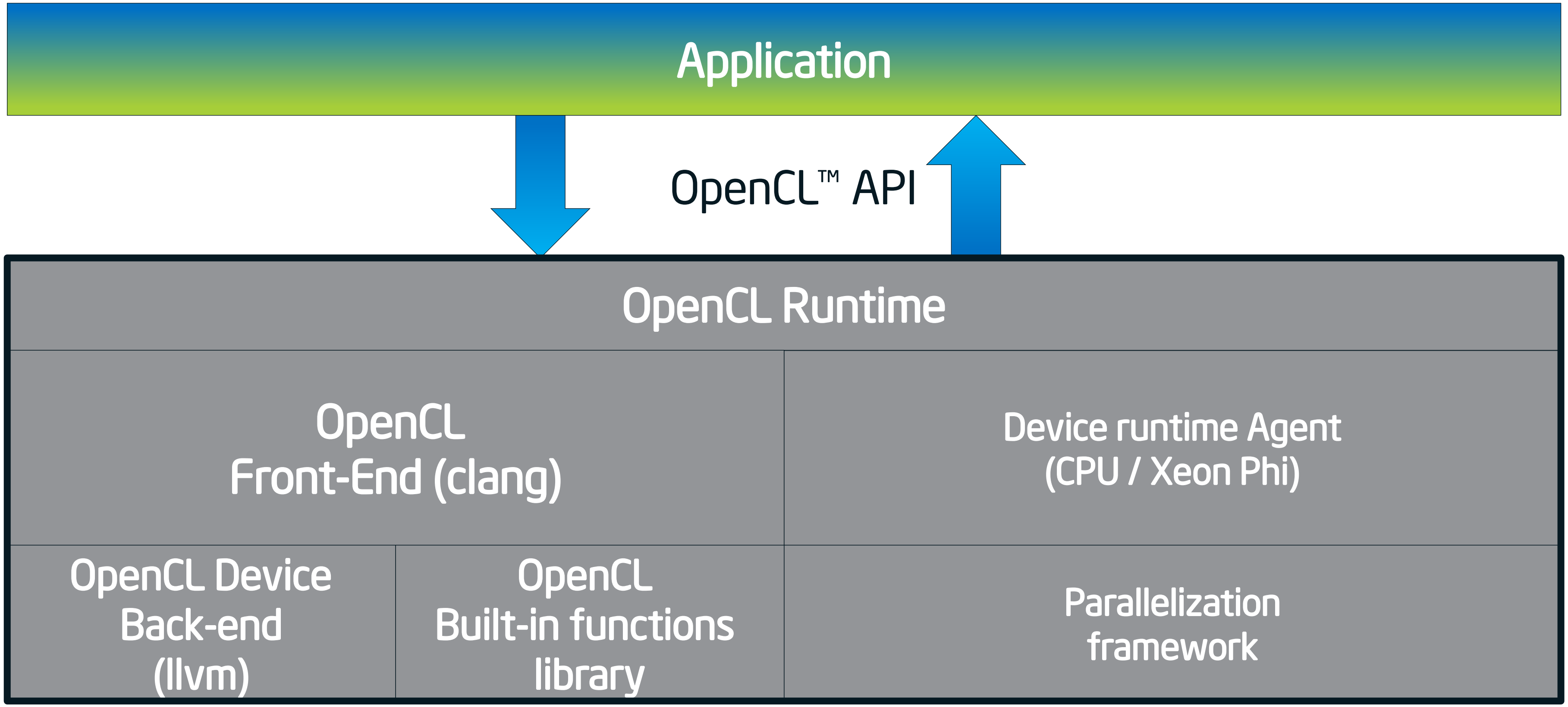
A Comprehensive Software Development Environment for OpenCL™ Applications

	Target Processors	Target Operating System	OpenCL Spec Version	Target SDK	Developer Benefits
Visual Computing Domain		Windows 7 Windows 8	OpenCL 1.2	(Version 2013)	Develop and deploy visual computing applications for 3rd and 4th Generation Intel® Core™ Processors Get ready for next generations
Data Center Domain		(Red Hat*, SUSE*)	OpenCL 1.2	(Version XE 2013)	Preserve your investment when developing high performance compute applications

OpenCL compiler validation Standalone Tool

Intel OpenCL™ implementation architecture

Running on Intel® CPU and Intel Xeon Phi™ Coprocessor



OpenCL is multicomponent system challenging to validate

Conformance Tests
OpenCL™ Implementation validation

- Provided and maintained by Khronos*
- A compliance test for OpenCL™ specification
- System level tests
  - Tests OpenCL implementation as whole entity
  - All components should be functional

Validation on component level is not trivial

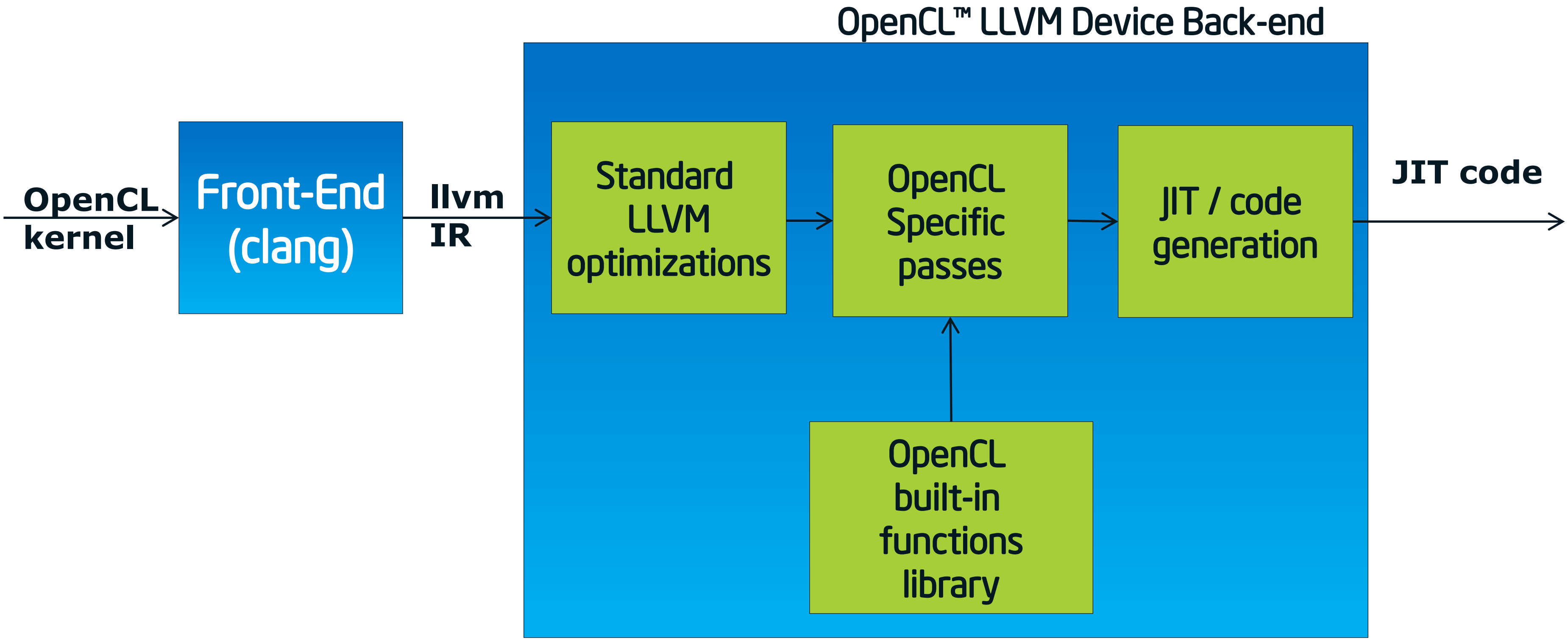
Optimization Notice

OpenCL™ Standard
Supporting Intel® Many Integrated Core Architecture

- Simultaneous development of multiple components
  - Runtime
  - Device agent
  - Device Back-end
  - OpenCL™ built-in functions library
  - …
- Conformance tests
  - Not functional until all components are ready
- Early testing is highly needed
  - Boosts development speed

How to validate device backend?

OpenCL™ Compiler for Intel® CPU and Intel Xeon Phi™ Coprocessors
Zoom in

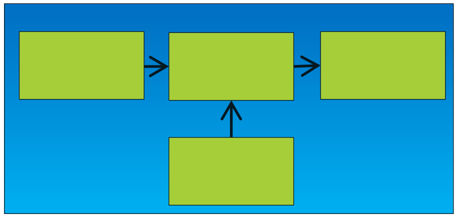


How to validate device backend?

Optimization Notice

Standalone compiler validation tool

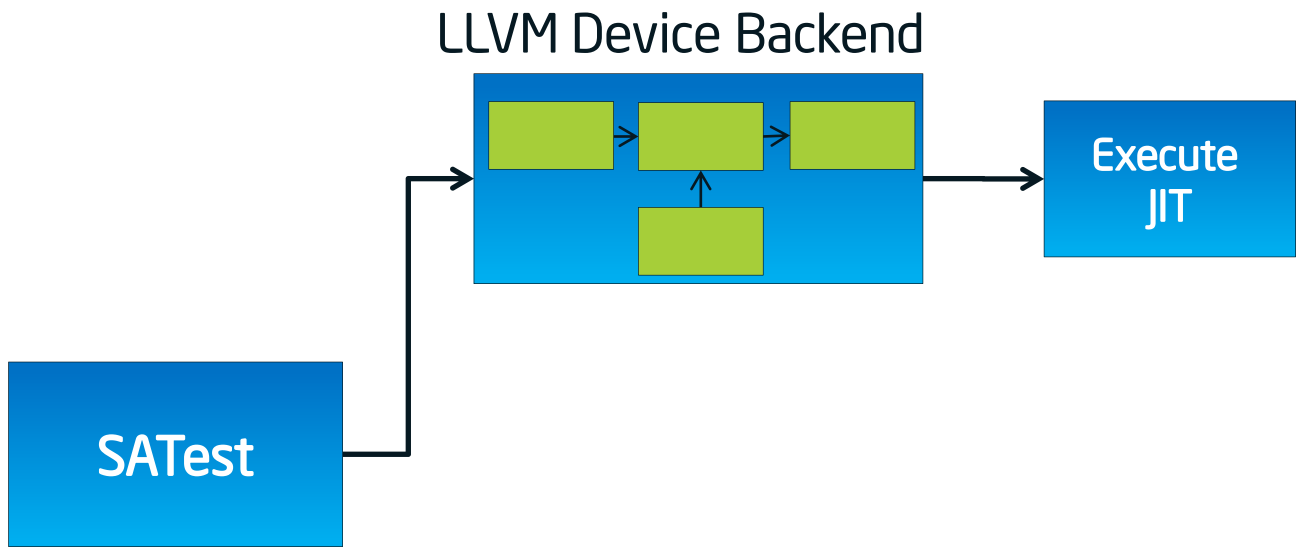
LLVM Device Backend



IDEA!!!

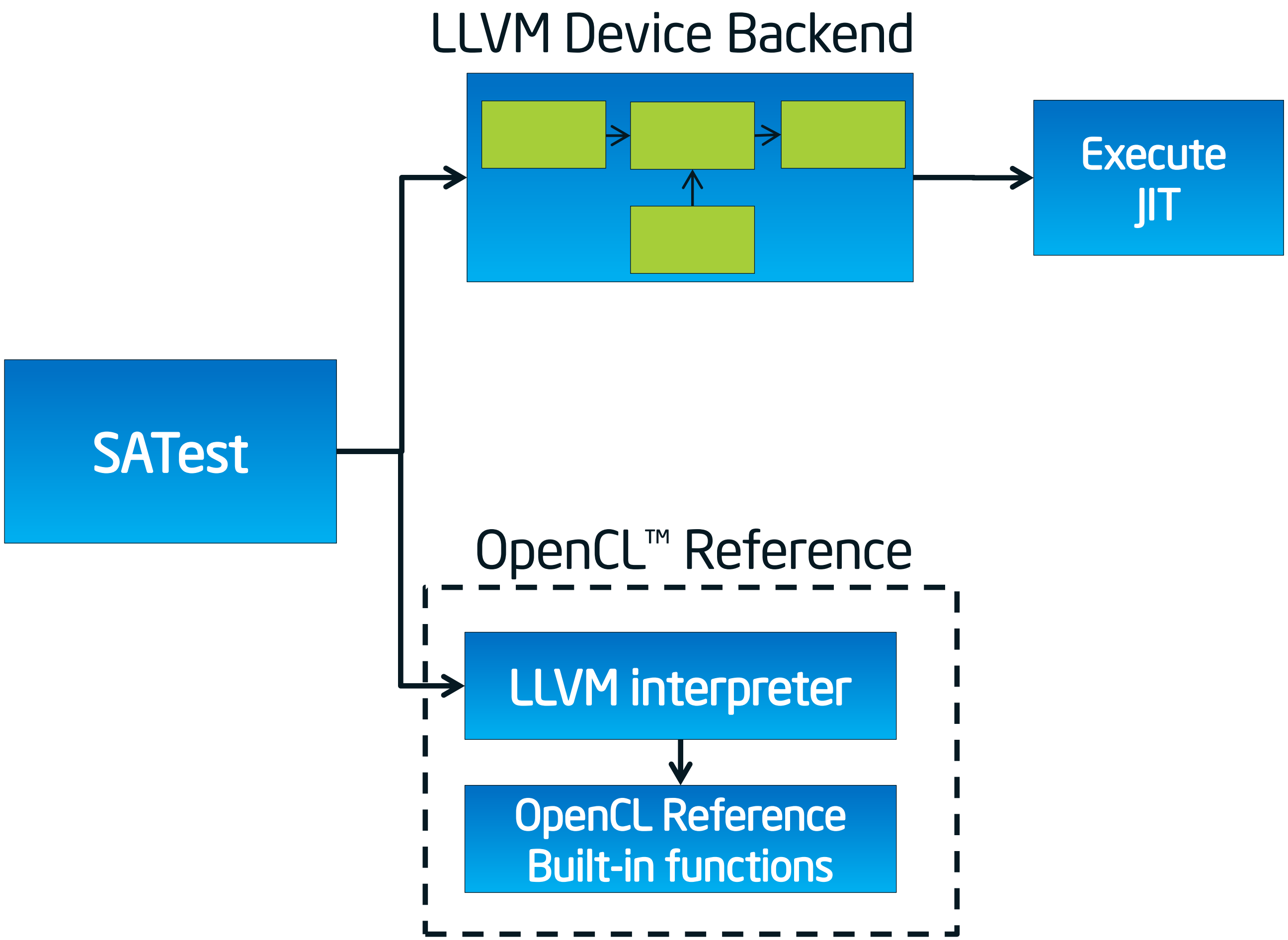
Isolate testing device backend from other components

Standalone compiler validation tool



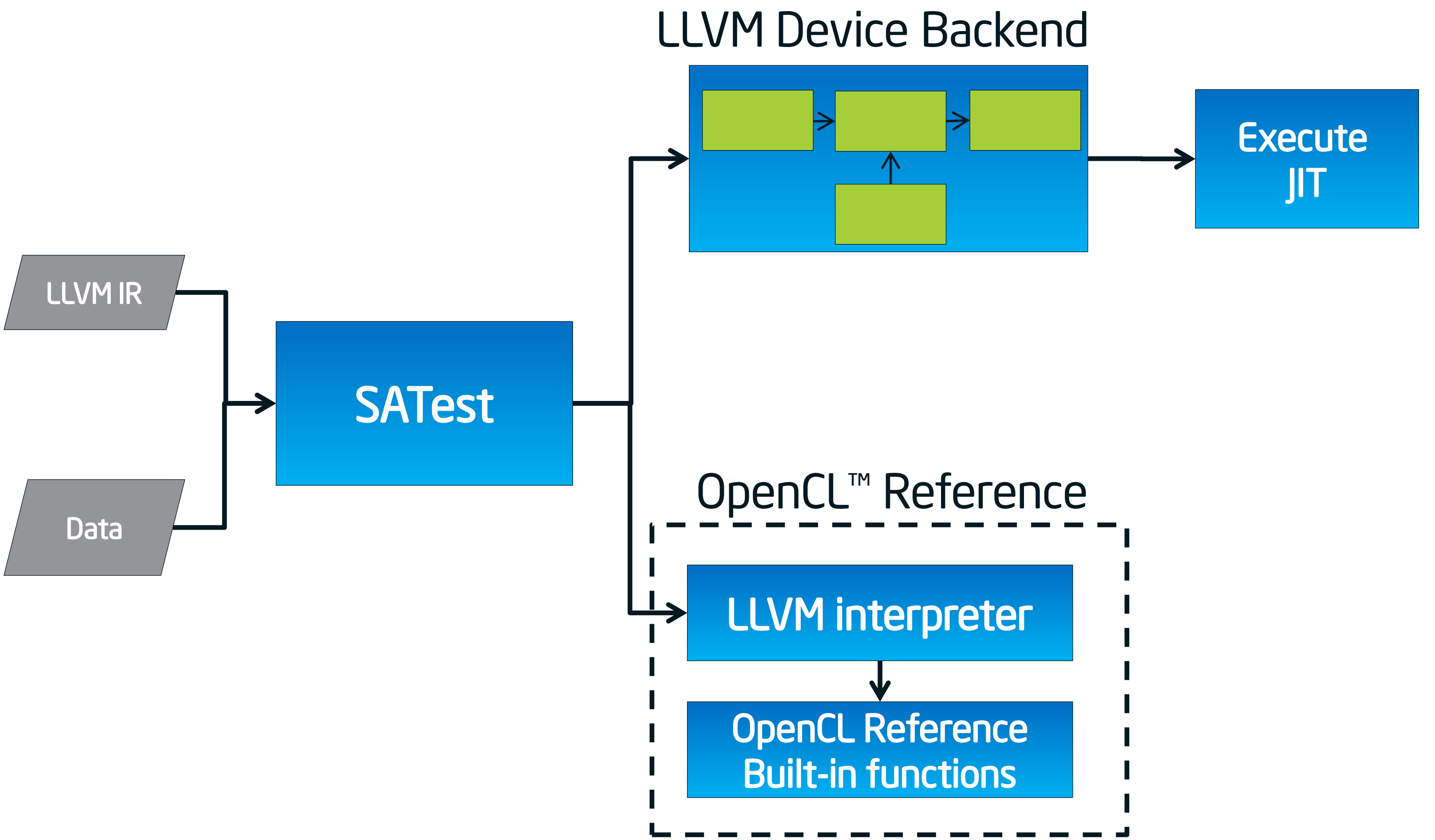
Implement standalone tool to work with device back-end

Standalone compiler validation tool



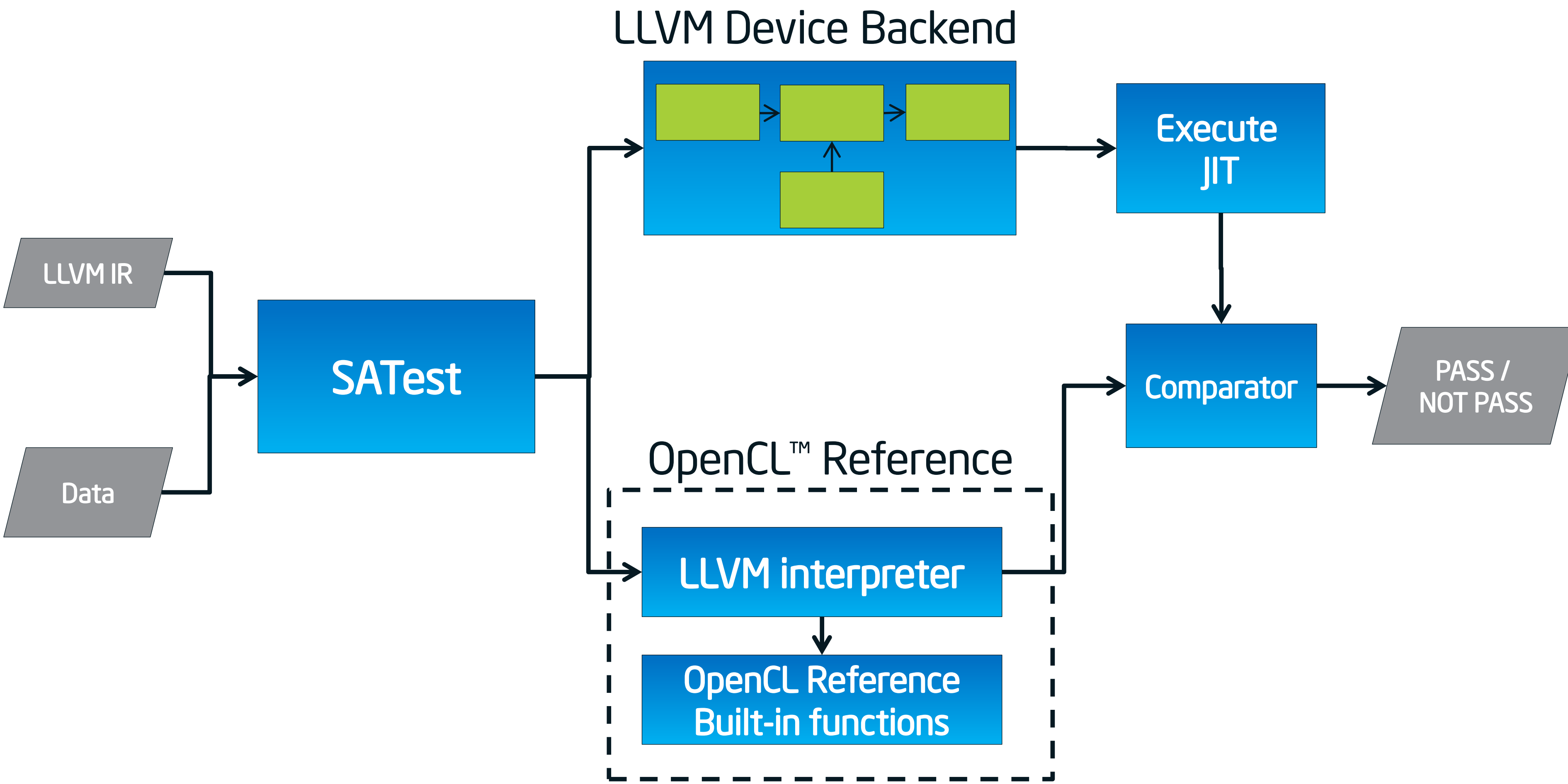
Implement OpenCL Reference to obtain reference results
Share the same build/execute interface with back-end

Standalone compiler validation tool



Import LLVM kernel compiled from OpenCL
Import data recorded from running OpenCL kernel

Standalone compiler validation tool



Compare Reference and Back-end outputs
Report PASS/NOT PASS

Standalone compiler validation tool
Features

Several modes of operation

- Functional Validation
  - Check device back-end produces OpenCL™ 1.2 conformant results
  - Real life kernels and data
- Performance
  - Compile time
  - Kernel execution time
  - Single thread execution
  - Isolate and Detect issues coming from backend
- Build
  - Dump LLVM IR and JIT code on specified pass
  - Directly track/debug optimizations

Tool for backend developers

LLVM Interpreter

LLVM Interpreter

- MUST have component for OpenCL™ Reference
- Executes LLVM bytecode directly
- Produces bitwise accurate results
- Slow
- No optimizations
- LLVM trunk version lacks vector and aggregate types support. Need to implement them

Produces reference results of running OpenCL kernel

Missing pieces in LLVM interpreter

OpenCL™

```
kernel void cl_exp(global const float4 *a,
                   global float4 *result) {

    int id = get_global_id(0);
    result[id] = exp(a[id]);
}
```

LLVM IR

```
define void @cl_exp(<4 x float> addrspace(1)* %a, <4 x float>
addrspace(1)* %result) nounwind {

  %1 = call i64 @get_global_id(i32 0) nounwind readnone
  %sext = shl i64 %1, 32
  %2 = ashr exact i64 %sext, 32
  %3 = getelementptr inbounds <4 x float> addrspace(1)* %a,
i64 %2
  %4 = load <4 x float> addrspace(1)* %3, align 16
  %5 = call <4 x float> @_Z3expDv4_f(<4 x float> %4)
nounwind readnone
  %sext1 = shl i64 %1, 32
  %6 = ashr exact i64 %sext1, 32
  %7 = getelementptr inbounds <4 x float>
addrspace(1)* %result, i64 %6
  store <4 x float> %5, <4 x float> addrspace(1)* %7, align 16
  ret void
}
```

Not supported in LLVM interperter

LLVM Interpreter: Current State
Essential Methods of Interpreter

```
class Interpreter : public ExecutionEngine, public
InstVisitor<Interpreter> {

  …

  // The runtime stack of executing code.
  // current function record.
  std::vector<ExecutionContext> ECStack;

  …
}
```

All interpreter logic encapsulated into 'Interpreter' class

LLVM Interpreter: Current State
Essential Methods of Interpreter

```
class Interpreter : public ExecutionEngine, public
InstVisitor<Interpreter> {

 …

  /// run - Start execution with the specified function and
arguments.

 virtual GenericValue runFunction(Function *F, const
                          std::vector<GenericValue> &ArgValues);


 /// Opcode Implementations(e.g. void visitLoadInst(LoadInst &I))

 void visit*();

 …

}
```

visit*() methods execute instructions

LLVM Interpreter: Current State
Execution Context

```
struct ExecutionContext {
  …
  /// LLVM values used in this invocation
  std::map<Value *, GenericValue> Values;
  /// Values passed …
  std::vector<GenericValue>  VarArgs;
  …
};
```

Local variables and arguments container

Stores current basic block, function and next instruction

Optimization Notice

GenericValue Structure

```
struct GenericValue {
  union {
    double          DoubleVal;
    float           FloatVal;
    PointerTy       PointerVal;
    struct { unsigned int first; unsigned int second; } UIntPairVal;
    unsigned char   Untyped[8];
  };
  APInt IntVal;   // also used for long doubles
    ...

};
```

Universal container for variable values

GenericValue Structure

```
struct GenericValue {
  union {
    double          DoubleVal;
    float           FloatVal;
    PointerTy       PointerVal;
    struct { unsigned int first; unsigned int second; } UIntPairVal;
    unsigned char   Untyped[8];
  };
  APInt IntVal;   // also used for long doubles
    ...
  std::vector<GenericValue> AggregateVal;

...

};
```

Stores missing vector and aggregate type

Adding new single field covers vector and aggregate types

Optimization Notice

Example of New visit* Method

Typical visit* method

```
void visit*(Instruction& I)
{
  ExecutionContext &SF = ECStack.back();
  const Type *Ty    = I.getType();
  GenericValue Src1 = getOperandValue(I.getOperand(0), SF);
  GenericValue Src2 = getOperandValue(I.getOperand(1), SF);
  GenericValue R;   // Result
  switch (Ty->getTypeID())
  {
    case: Type::IntegerTyID: // R.IntVal = Src1.IntVal OP Src2.IntVal;
    case: Type::FloatTyID:   // R.FloatVal = Src1.FloatVal OP Src2.FloatVal;
    …
}
```

Example of New visit* Method

Typical visit* method

```
void visit*(Instruction& I)
{
  ExecutionContext &SF = ECStack.back();
  const Type *Ty    = I.getType();
  GenericValue Src1 = getOperandValue(I.getOperand(0), SF);
  GenericValue Src2 = getOperandValue(I.getOperand(1), SF);
  GenericValue R;   // Result
  switch (Ty->getTypeID())
  {
    case: Type::IntegerTyID: // R.IntVal = Src1.IntVal OP Src2.IntVal;
    case: Type::FloatTyID:   // R.FloatVal = Src1.FloatVal OP Src2.FloatVal;
    …
    case: Type::VectorTyID:
       // set vector size
       R.AggregateVal.resize(Src1.AggregateVal.size());

       if (cast<VectorType>(Ty)->getElementType()->isFloatTy())
       // process R.FloatVecVal
       …
       if (cast<VectorType>(Ty)->getElementType()->isIntegerTy())
       // process R.IntVecVal

  }
```

Processing vectors

Interpreter
Operations implemented

Group	Number of operations
Vector Memory Access and Addressing	4
Vector binary	12
In-vector	3
Vector comparison	3
Vector bitwise binary	6
Vector conversion	10
Aggregate	2

OpenCL™ Reference Tool
Built-in Functions Implemented

- High-precision reference implementation
- Full support of OpenCL™ 1.2 standard
- Over 3000 functions
  - Math
  - Geometric
  - Common
  - Image
  - Conversions between types (~2000)
  - etc

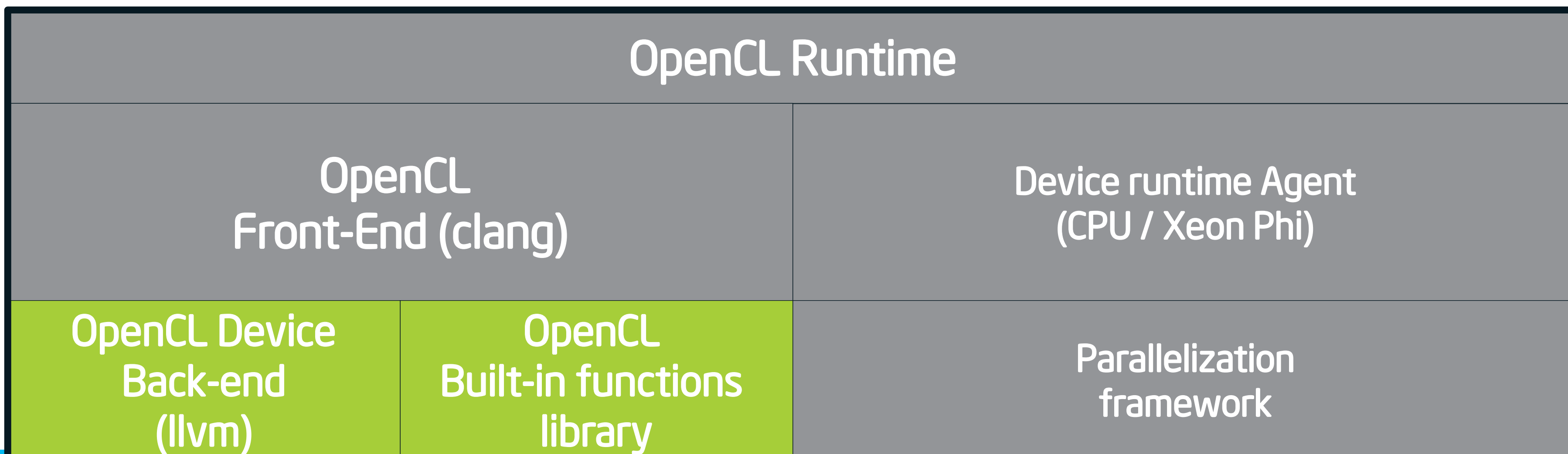
Optimization Notice

Demo

Optimization Notice

OpenCL™ Back-end Validation for Intel® Xeon Phi™ Coprocessor Summary

- Enabled validation at early stage
  - At 1st stages on functional simulator
  - Other components of OpenCL™ infrastructure not yet ready
  - Before hardware was ready
- Saved time on isolated backend development



Contribution to the LLVM Community

Interpreter
Summary of our Modifications

- Added Vector and aggregate types to Execution Context
- Added Vector operations
  - Memory access and addressing
  - Binary
  - In-vector
  - Comparison
  - Constant
- Added Aggregate types operations
  - 'extractvalue'
  - 'insertvalue'

Optimization Notice

Contribution to the LLVM Community

- Uploading interpreter changes to LLVM repository
- Incremental commits adding new features to interpreter
- Vectors and aggregate types will be supported in interpreter

Wish list for the Interpreter

- Add standard C LLVM intrinsics support
- Enhance processing of constant expressions

- Multiple components are developed simultaneously
- Existing tests work only after all components are ready
- **Lack of Early validation**
- We provide early validation infrastructure using LLVM interpreter

Optimization Notice

Thank you

Legal Disclaimer & Optimization Notice