OPENCL COMPILER FOR CPU IN LLVM

Evgeniy Tyurin

evgeniy.tyurin@intel.com

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Objective

In 20 minutes

discover collab opportunities

within OpenCL part of LLVM community

MAPPING TO CPU

OpenCL kernel

Focus on data parallelism!

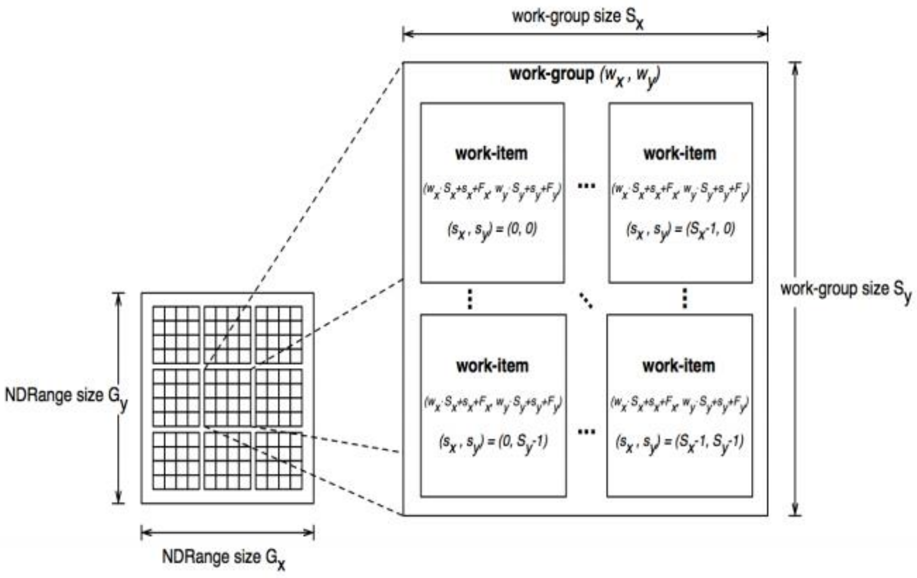
- Developer writes kernel processing a single **work item** within problem space

```
__kernel void
cl_add(__global float *a,
       __global float *b,
       __global float *res) {

  size_t gid = get_global_id(0);
  res[gid] = a[gid] + b[gid];
}
```

OpenCL kernel

Focus on data parallelism!

- Developer writes kernel processing a single **work item** within problem space
- Work-items are organized into **work-groups**
- Work-groups comprise the whole **NDRange** – problem space



OpenCL 1.2 specification, fig. 3.2

OpenCL execution on CPU

Work items in a work group are executed in an implicit loop.

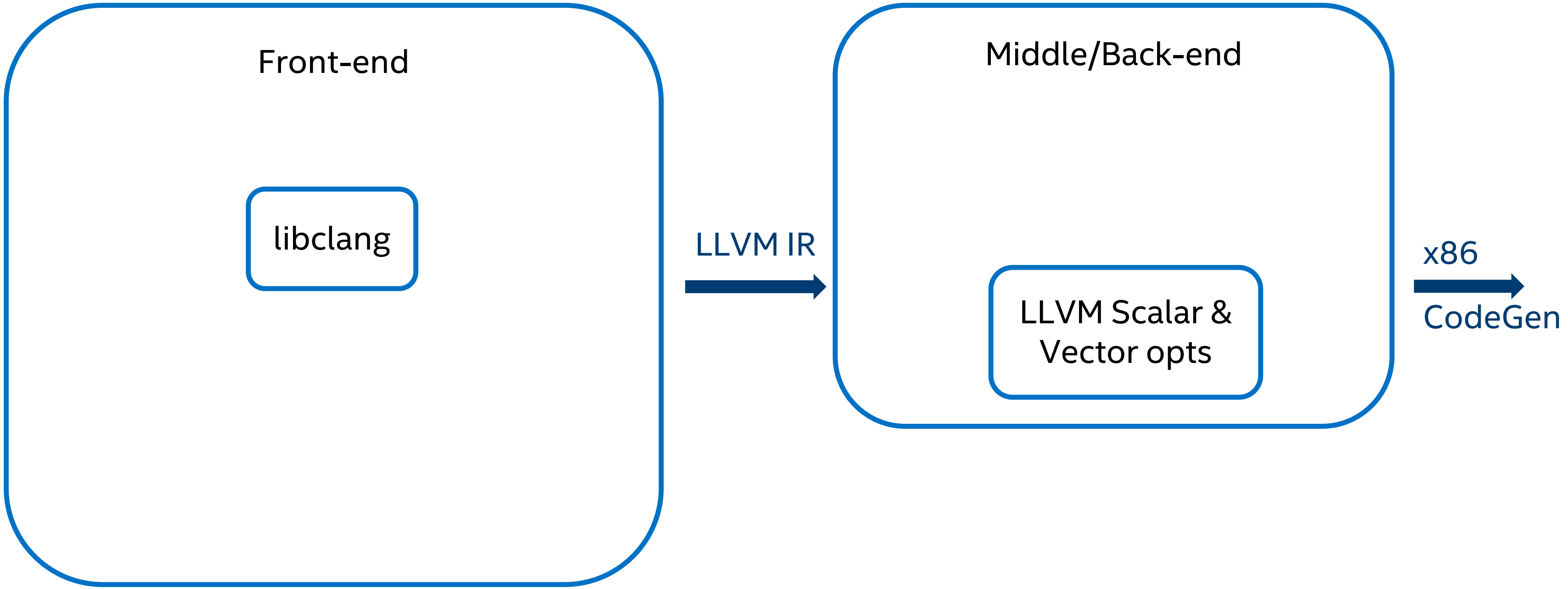
- Work item batch ⇒ **SIMD** lane
- Work group ⇒ CPU **thread**
- NDRange ⇒ CPUs

Execution of work groups is parallelized for CPU units.

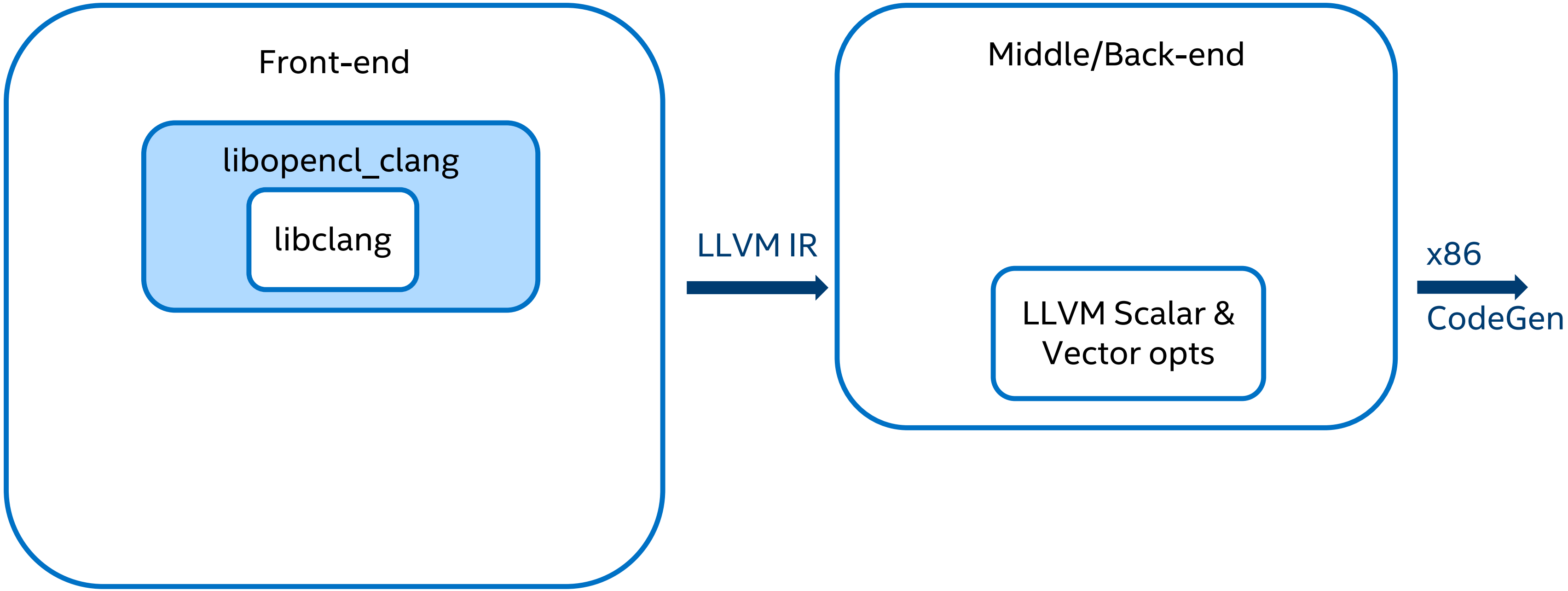
```
__kernel void
cl_mul(__global float *a,
       __global float *b,
       __global float *res) {

  size_t gid = get_global_id(0);
  res[gid] = a[gid] + b[gid];
}
```

COMPILER STACK

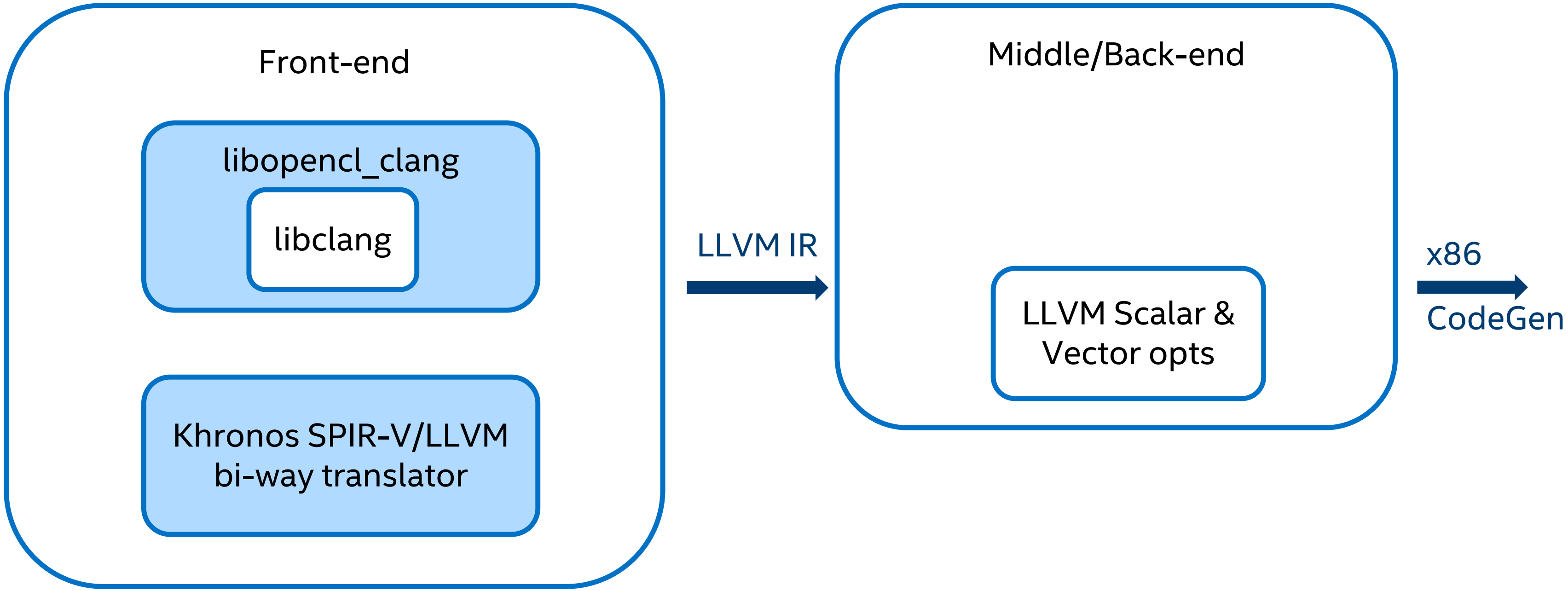
CPU Compiler components



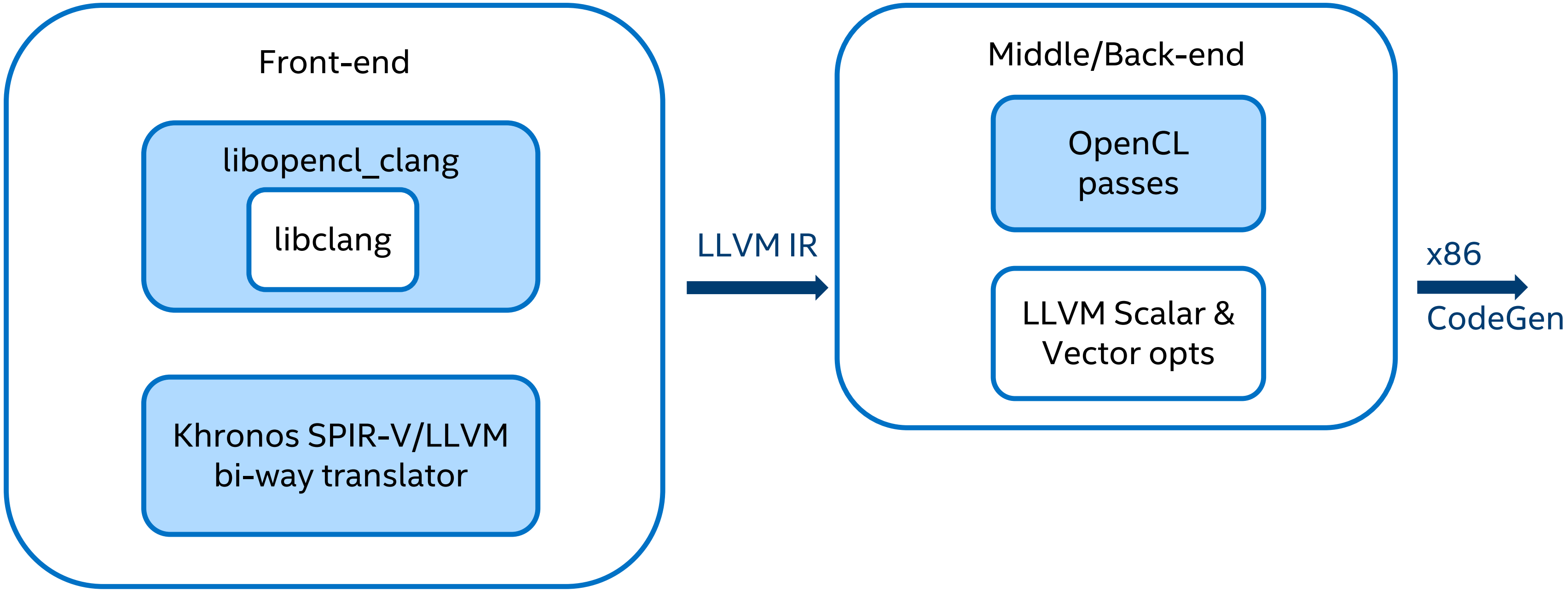
CPU Compiler components



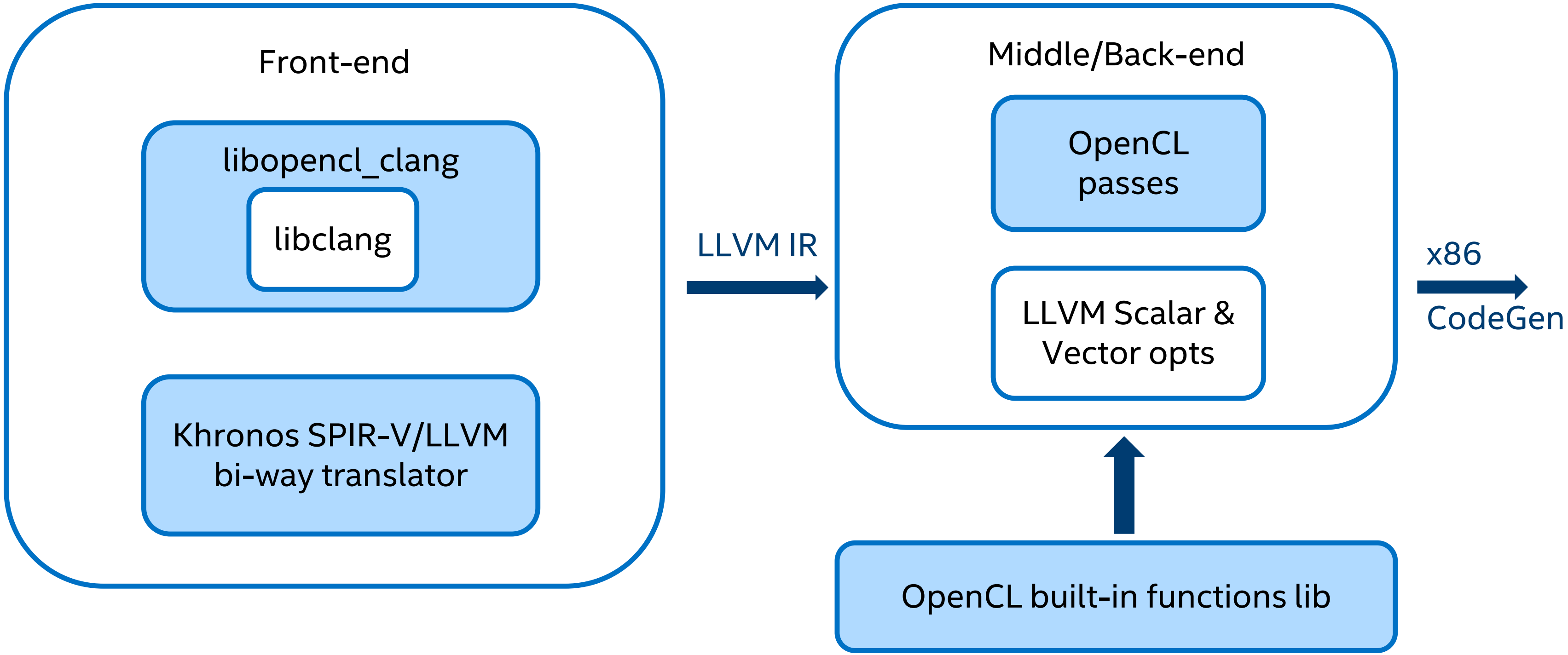
CPU Compiler components



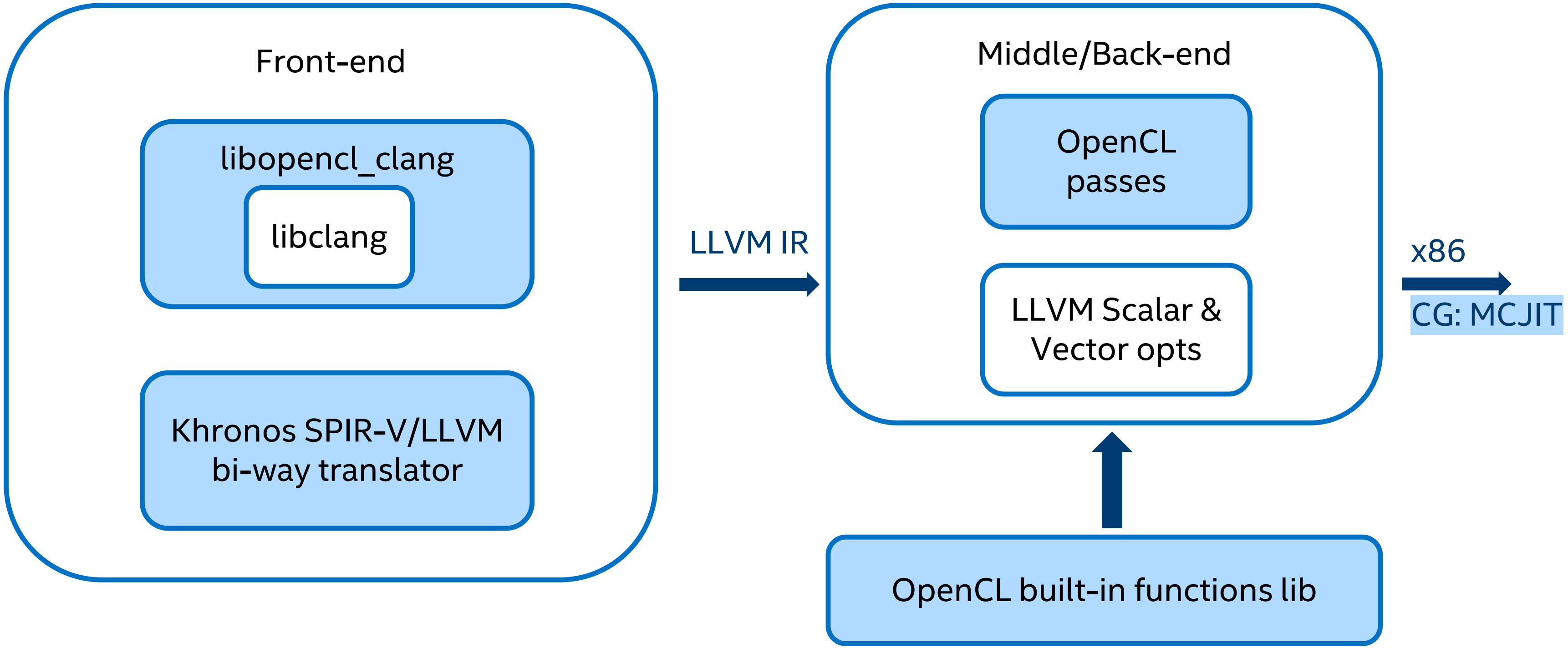
CPU Compiler components



CPU Compiler components



CPU Compiler components



FRONTEND

Frontend challenges

OpenCL C 1.2/2.0

SPIR-V

SPIR 1.2

x86 precompiled binary

- multiple inputs
- multiple targets

Graphics ME/BE

FPGA ME/BE

DSP ME/BE

CPU ME/ BE

x86

Frontend challenges

OpenCL C 1.2/2.0

Graphics ME/BE

FPGA ME/BE

SPIR-V

DSP ME/BE

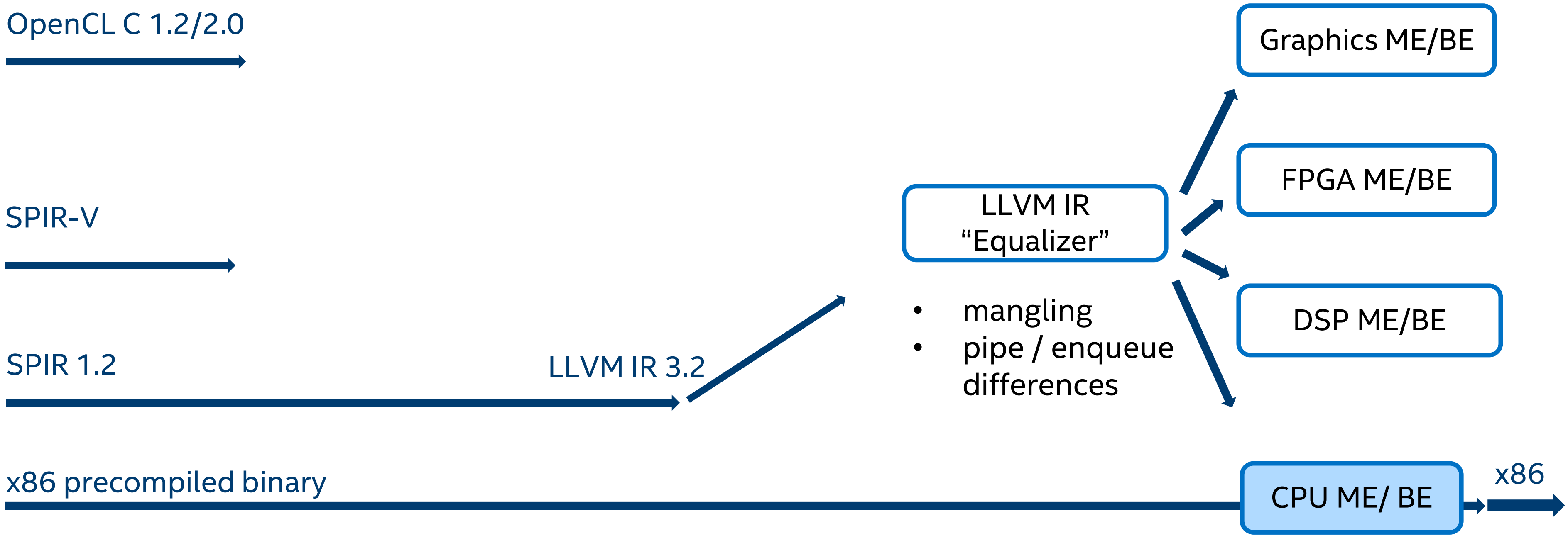
SPIR 1.2

x86 precompiled binary

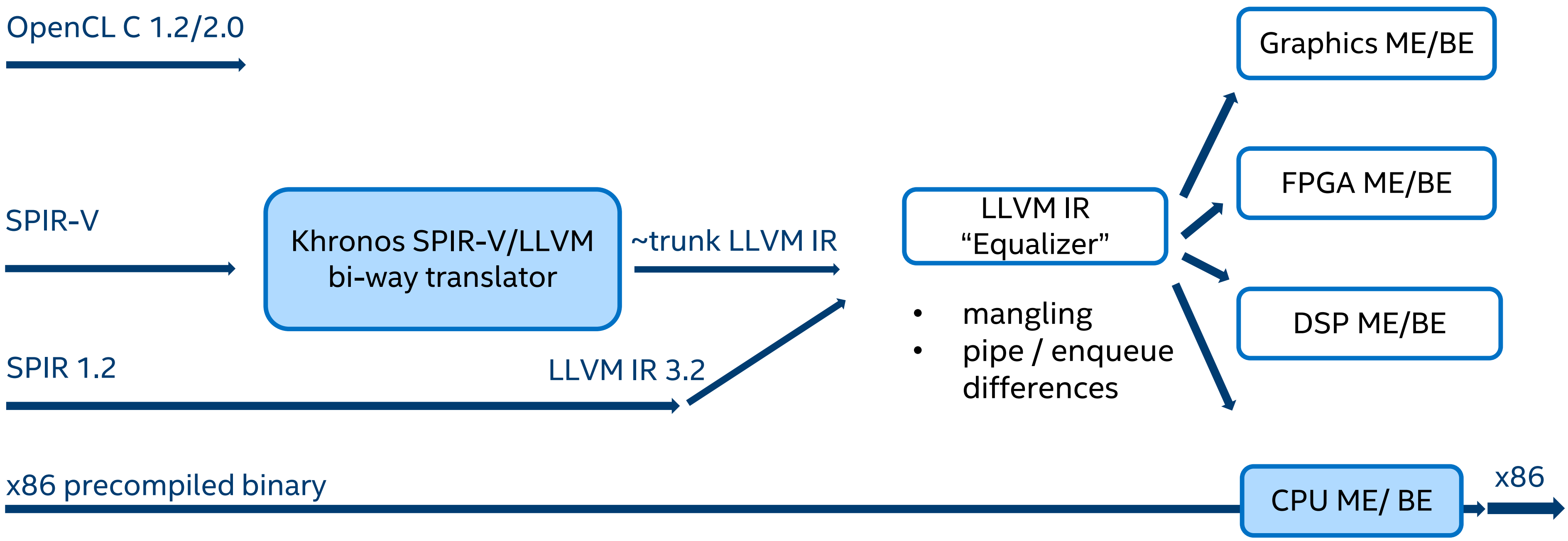
CPU ME/ BE

x86

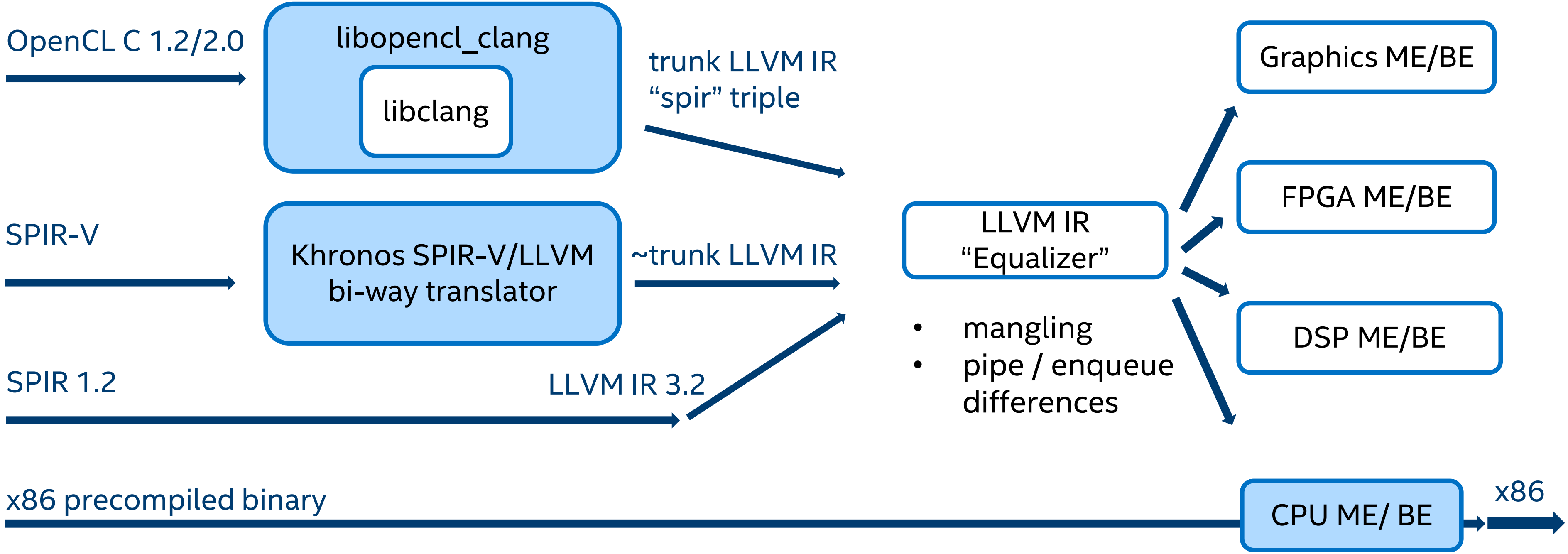
Frontend challenges



Frontend challenges



Frontend challenges



libopencl_clang

OpenCL-oriented libclang extension/wrapper

- In-memory from-source compilation
- Precompiled headers for OpenCL built-ins
- C-style APIs for actions like Compile/Link/GetKernelArgInfo
- Stable API for different device backends

libopencl_clang – example #1

```
extern "C" CC_DLL_EXPORT int Compile(
  // A pointer to main program's source (null terminated string)
  const char *pszProgramSource,
  // array of additional input headers to be passed in memory (each null
  // terminated)
  const char **pInputHeaders,
  // the number of input headers in pInputHeaders
  unsigned int uiNumInputHeaders,
  // array of input headers names corresponding to pInputHeaders
  const char **pInputHeadersNames,
  // optional pointer to the pch buffer
  const char *pPCHBuffer,
  // size of the pch buffer
  size_t uiPCHBufferSize,
  // OpenCL application supplied options
  const char *pszOptions,
  // optional extra options string usually supplied by runtime
  const char *pszOptionsEx,
  // OpenCL version string - "120" for OpenCL 1.2, "200" for OpenCL 2.0, ...
  const char *pszOpenCLVer,
  // optional outbound pointer to the compilation results
  Intel::OpenCL::ClangFE::IOCLFEBinaryResult **pBinaryResult
);
```

libopencl_clang – example #2

```
extern "C" CC_DLL_EXPORT int Link(
  // array of additional input headers to be passed in memory
  const void **pInputBinaries,
  // the number of input binaries
  unsigned int uiNumBinaries,
  // the size in bytes of each binary
  const size_t *puiBinariesSizes,
  // OpenCL application supplied options
  const char *pszOptions,
  // optional outbound pointer to the compilation results
  Intel::OpenCL::ClangFE::IOCLFEBinaryResult **pBinaryResult
 );
```

libopencl_clang

Source is available @ https://github.com/intel/opencl-clang

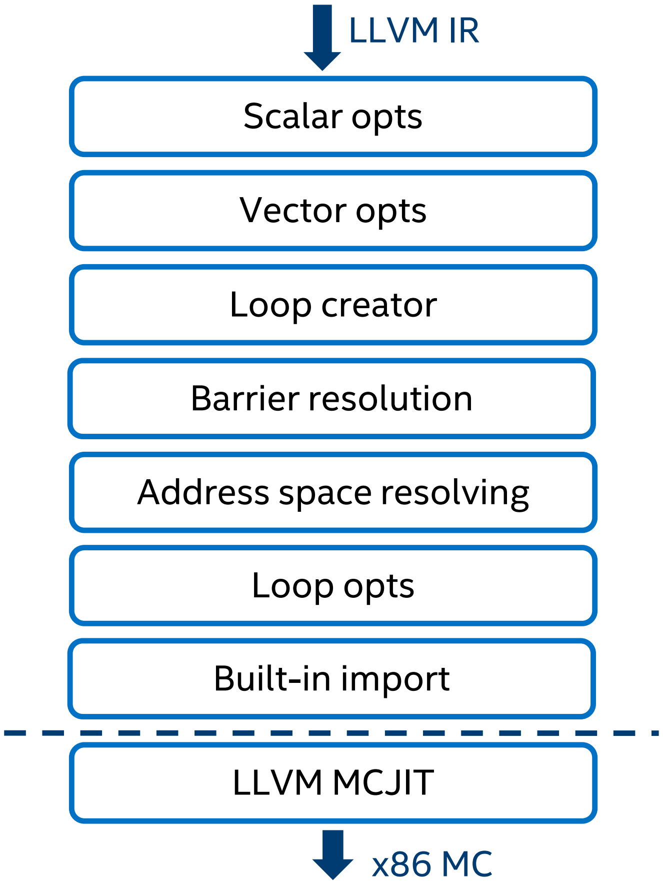
MIDDLE END

CPU middle end challenges

Optimize a hetero language!

CPU-unfriendly OpenCL features:

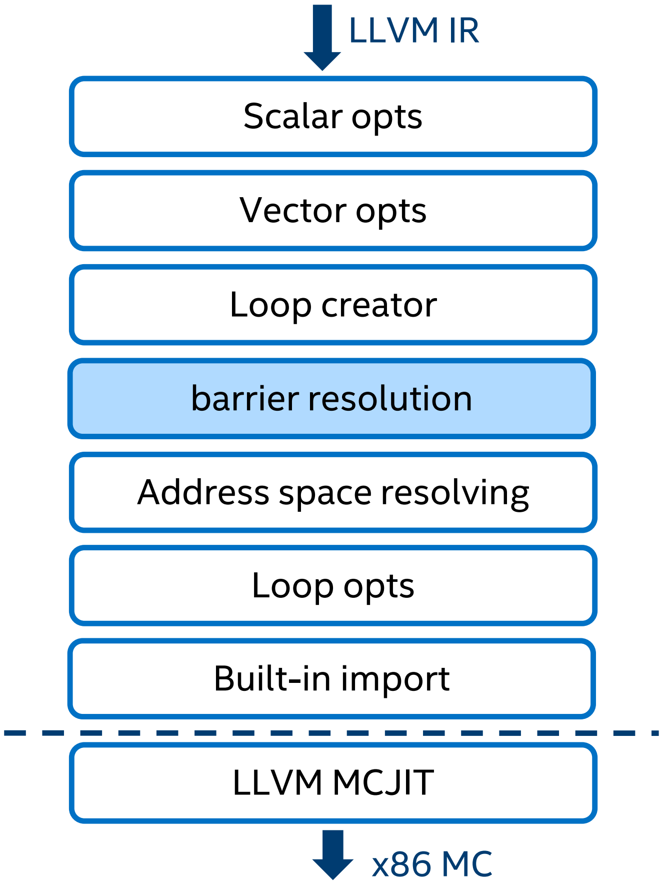
- barrier
- address spaces
- images
- pipes



OpenCL barrier

Handles barrier() built-in function

- All work-items in work-group must hit the barrier before any of them can continue execution
- Pass splits the CFG along barrier calls and creates ‘switch’-driven work-group loops to enforce the barrier



Barrier resolution

Conceptual pseudo code

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
}
```

Barrier resolution

Conceptual pseudo code

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
}
```

```
kernel void test(...)
{
  int currWI = 0;
  int currBarrier = 0;
label_0:
  ...code1
  goto label_barrier_1;
label_barrier_1:
  if (currWI < groupSize) {
    currWI++;
    switch (currBarrier) {
    case 0: goto label_0;
    case 1: goto label_1;
    }
  }
  else {
    currWI = 0;
    currBarrier = 1; //check and exit if finised
  }
label_1:
  ...code2
  goto label_barrier_1;
}
```

Barrier resolution

Let's consider this:

```
kernel void test(...)
{
  int x = b * A[wi_id];
  barrier();
  C[wi_id] = x;
}
```

Barrier resolution

Let's consider this:

```
kernel void test(...)
{
  int x = b * A[wi_id];
  barrier();
  C[wi_id] = x;
}
```

```
kernel void test(...)
{
  int currWI = 0;
  int currBarrier = 0;
label_0:
  int x = b * A[wi_id];
  goto label_barrier_1;
label_barrier_1:
  if (currWI < groupSize) {
    currWI++;
    switch (currBarrier) {
    case 0: goto label_0;
    case 1: goto label_1;
    }
  }
  else {
    currWI = 0;
    currBarrier = 1; //check and exit if finised
  }
label_1:
  C[wi_id] = x;
  goto label_barrier_1;
}
```

Barrier resolution

Let's consider this:

```
kernel void test(...)
{
  int x = b * A[wi_id];
  barrier();
  C[wi_id] = x;
}
```

- values x is different for every work item;

```
kernel void test(...)
{
  int currWI = 0;
  int currBarrier = 0;
label_0:
  int x = b * A[wi_id];
  goto label_barrier_1;
label_barrier_1:
  if (currWI < groupSize) {
    currWI++;
    switch (currBarrier) {
    case 0: goto label_0;
    case 1: goto label_1;
    }
  }
  else {
    currWI = 0;
    currBarrier = 1; //check and exit if finised
  }
label_1:
  C[wi_id] = x;
  goto label_barrier_1;
}
```

Barrier resolution

Let's consider this:

```
kernel void test(...)
{
  int x = b * A[wi_id];
  barrier();
  C[wi_id] = x;
}
```

- values x is different for every work item;
- after barrier all work-items will use same value for x!

```
kernel void test(...)
{
  int currWI = 0;
  int currBarrier = 0;
label_0:
  int x = b * A[wi_id];
  goto label_barrier_1;
label_barrier_1:
  if (currWI < groupSize) {
    currWI++;
    switch (currBarrier) {
    case 0: goto label_0;
    case 1: goto label_1;
    }
  }
  else {
    currWI = 0;
    currBarrier = 1; //check and exit if finised
  }
label_1:
  C[wi_id] = x;
  goto label_barrier_1;
}
```

Barrier resolution

Pseudo code:

```
kernel void test(...)
{
  int x = b * A[wi_id];
  barrier();
  C[wi_id] = x;
}
```

- values crossing the a barrier must be preserved for each work-item;

```
kernel void test(...)
{
  int currWI = 0;
  int currBarrier = 0;
label_0:
  store x into buffer[offset];
  goto label_barrier_1;
label_barrier_1:
  if (currWI < groupSize) {
    currWI++;
    switch (currBarrier) {
    case 0: goto label_0;
    case 1: goto label_1;
    }
  }
  else {
    currWI = 0;
    currBarrier = 1; //check and exit if finised
  }
label_1:
  load x_1 from buffer[offset];
  goto label_barrier_1;
}
```

Barrier: Analysis phase

- both x and y depend on work-item ID.
- scope analysis:
  - x crosses barrier
  - y does not cross
- only x is marked and it’s size 32
- x offset will be 0
- next value’s offset will be 4

```
kernel void test(...)
{
  int x = b * A[wi_id];
  int y = B[wi_id];
  barrier();
  C[wi_id] = x;
}
```

Barrier: Analysis phase - Contd

barrier():

- Give barrier instruction a unique number [1,…,#bariers]
- Find the predecessor barriers for each barrier instruction

IR values:

- We are interested only in values that depend on work-item ID
- Find aliveness scope of such values and mark if they cross the barrier
- Find the total size in bytes of marked LLVM IR values
- Calculate the offset of each marked value with respect to the total size and with alignment consideration

Barrier: Transformation phase

- Add two new alloca variables to the beginning of the kernel
  - “currWI” initialized to 0
  - “currBarrier” initialized to 0
- for every marked LLVM value
  - Store this value to special buffer at offset given by the Analysis pass
  - For each barrier that exists in the scope of the value add a load instruction from the special buffer at same offset
  - Replace all usage of this value to use the new loaded value

Barrier: Transformation phase - Contd

- for each Barrier instruction
  - Replace it with this code:

```
if (currWI < groupSize) {
  currWI++;
  switch (currBarrier) {
    case 0: goto label_0;
    // case i: goto label_i;
    // for all "i" in barrier predecessors
  }
}
else {
  currWI = 0;
  currBarrier = #;
}
label_#: // current barrier number
__mm_mfence();
```

Barrier - Contd

- there's only one barrier
  - it's number is #1
  - barrier #0 is always the prologue of the kernel.
- predecessor of barrier #1 is #0.

```
kernel void test(...)
{
  int x = b * A[wi_id];
  int y = B[wi_id];
  barrier();
  C[wi_id] = x;
}
```

Barrier inside a function

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
  C[wi_id] = foo();
}

int foo()
{
  ...code3
   barrier();
  ...code4
}
```

Barrier inside a function

jump into the insides of a function required

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
  C[wi_id] = foo();
}

int foo()
{
  ...code3
   barrier();
  ...code4
}
```

Barrier inside a function - solution

Inline function?

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
  C[wi_id] = foo();
}

int foo()
{
  ...code3
   barrier();
  ...code4
}
```

Barrier inside a function - solution

Inline function:

- what if we cannot inline?

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
  C[wi_id] = foo();
}

int foo()
{
  ...code3
   barrier();
  ...code4
}
```

Barrier inside a function - solution

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
  C[local_wi_id] = foo();
}

int foo()
{
  ...code3
   barrier();
  ...code4
}
```

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
  barrier(); // extra
  C[wi_id] = foo();
  dummyBarrier();
}

int foo()
{
  dummyBarrier();
  ...code3
   barrier();
  ...code4
  barrier(); // extra
}
```

Barrier inside a function - solution

- For each function with barrier:
  - add dummyBarrier() at its begin
  - add barrier() at it's end.
- For each call to a function with barrier:
  - add barrier() before the function call
  - add dummyBarrier() after the function call
- dummyBarrier()
  - only counts towards barrier predecessors
  - has no barrier semantics

```
kernel void test(...)
{
  ...code1
  barrier();
  ...code2
  barrier(); // extra
  C[wi_id] = foo();
  dummyBarrier();
}

int foo()
{
  dummyBarrier();
  ...code3
   barrier();
  ...code4
  barrier(); // extra
}
```

Takeaway

Let's exchange feedback,

ask questions,

and extend collaboration beyond today's limits

Software