THE UNIVERSITY *of* EDINBURGH
informatics

icsa | Institute for Computing Systems Architecture

Introducing a Heterogeneous Execution Engine for LLVM

Chris Margiolas
chrmargiolas@gmail.com

What is this presentation about?

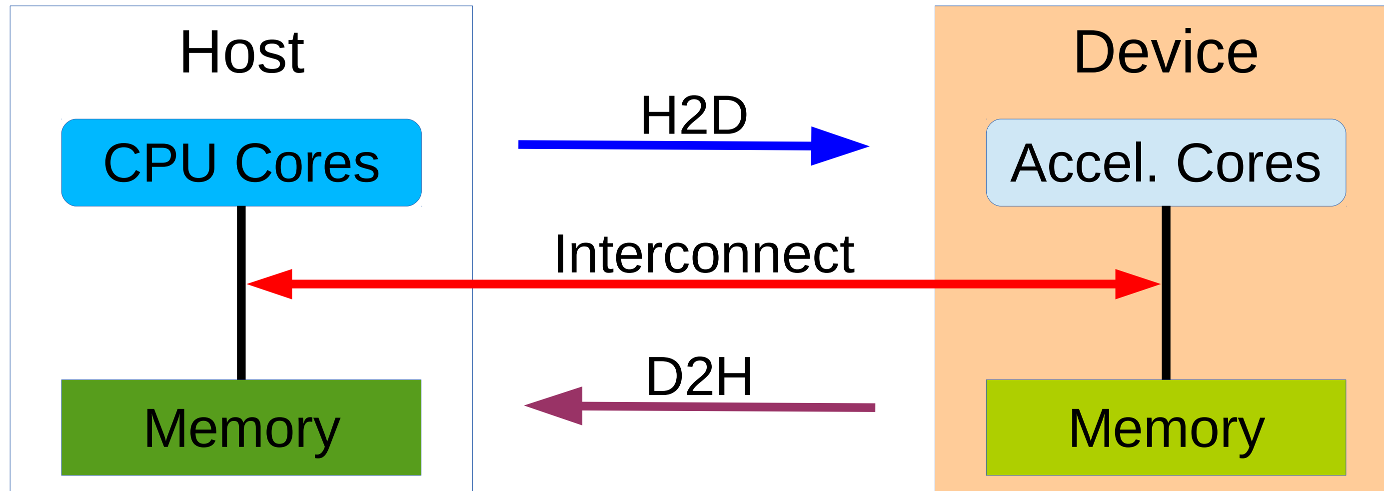
Hexe: A compiler and runtime infrastructure targeting transparent software execution on heterogeneous platforms.

- Hexe stands for **H**eterogeneous **EX**ecution **E**ngine

Key features:

- Compiler Passes for Workload Analysis and Extraction.
- Runtime Environment (Scheduling, Data Sharing and Coherency etc).
- Modular Design. Core functionality independent of the accelerator type. Specialization via Plugins both on the compiler and runtime.
- Extending the LLVM infrastructure.

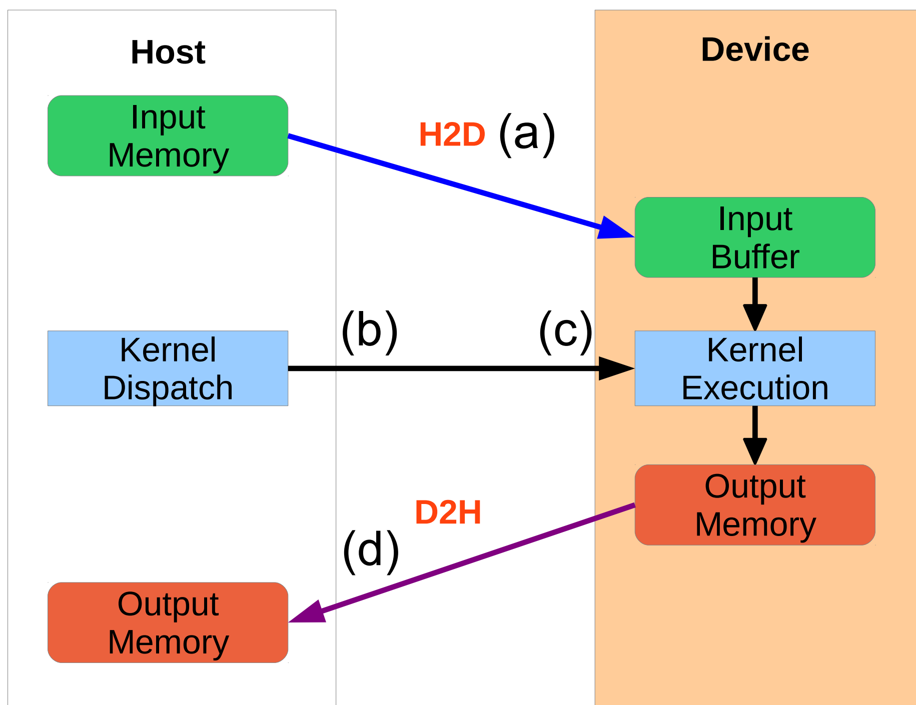
A Reference Heterogeneous System



- Two platform components, named **Host** and **Device**.
- The **Host** is the main architecture where the OS and core applications run.
- The **Device** is a co-processor that computes workloads dispatched by **Host**.
- **H2D**: Host to Device Communication/Coherency operations.
- **D2H**: Device to Host Communication/Coherency operations.

This is only a high level abstraction, actual hardware varies.

Workload Offloading Concept



This scheme is followed by:

- OpenCL
- CUDA
- DSP SDKs
- OpenGL
- Cell BE in the past etc..

➢ Depending on the hardware and software capabilities these operations may vary significantly.

Offloading operations:

a) Enforce Data Sharing & Coherency (From Host to Device).

b) Kernel Dispatch (From Host to Device)

c) Kernel Execution (On Device)

d) Enforce Data Sharing & Coherency (From Device to Host).

Existing Solutions for Workload Offloading

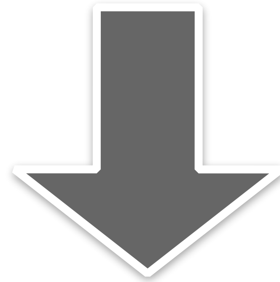
- Programming languages for explicit accelerator programming such as OpenCL and CUDA.
- Language extensions such as OpenMP 4.0, OpenACC.
- Domain Specific Languages.
- Source to Source compilers.

Upcoming Issues:

- Adoption of a new programming model is required.
- Significant development effort.
- No actual compiler integration.
- No integration with JIT technologies.
- Current solutions are language, platform and processor specific.

What is missing? (1)

```
void funnycopy(char *dst, char *src, unsigned size)
{
  for(; size; --size, ++dst, ++src) *dst=*src;
}
```

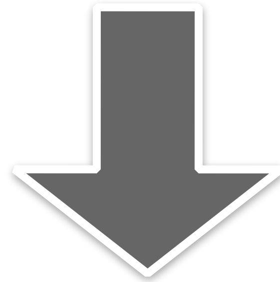


CPU Cores

- Targeting CPUs is trivial.
- Minimal development effort.
- Well defined programming model and conventions (been in use for decades).

What is missing? (2)

```c
void funnycopy(char *dst, char *src, unsigned size)
{
  for(; size; --size, ++dst, ++src) *dst=*src;
}
```



Accelerator Cores

- Targeting accelerators is complex.
- Significant development effort.
- Multiple and diverse programming environments.
- The programming models and conventions vary across accelerator types and vendors.

Chris Margiolas chrmargiolas@gmail.com

What is missing? (3)

```
void funnycopy(char *dst, char *src, unsigned size)
{
  for(; size; --size, ++dst, ++src) *dst=*src;
}
```

- How do we target multiple processor types at the same time?
- How do we remain portable and transparent?
- How do we support diverse processors and platform types?

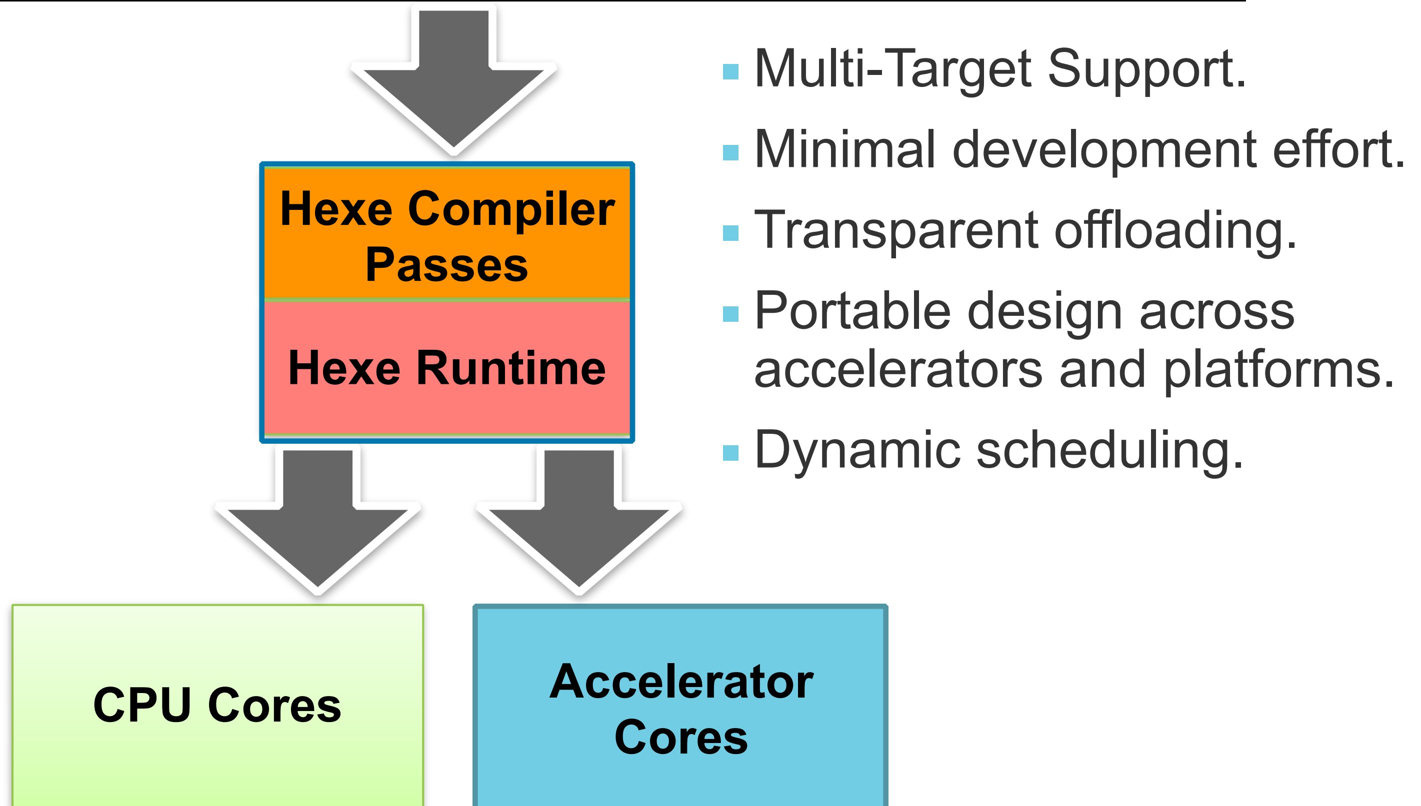
CPU Cores

Accelerator Cores

Chris Margiolas chrmargiolas@gmail.com

What is missing? (4)

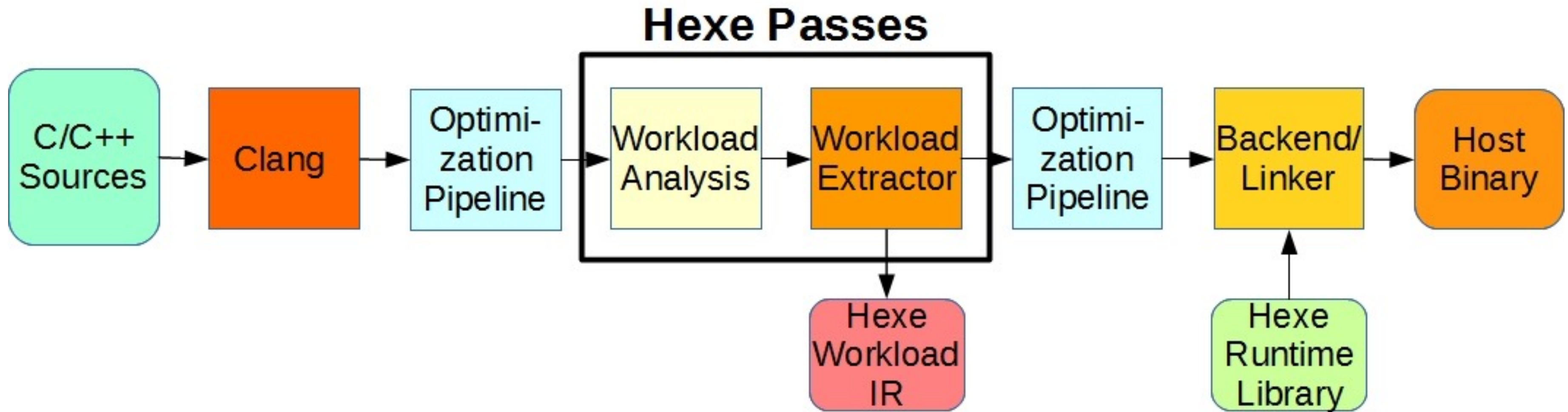
```c
void funnycopy(char *dst, char *src, unsigned size)
{
  for(; size; --size, ++dst, ++src) *dst=*src;
}
```



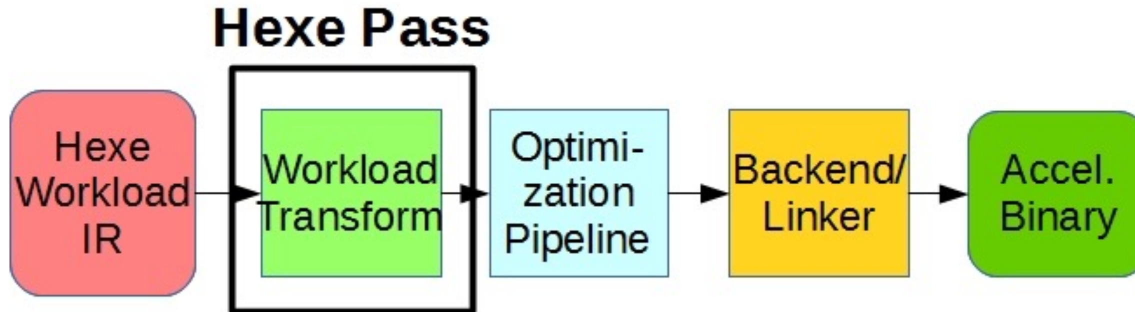
Chris Margiolas chrmargiolas@gmail.com

Hexe Compilation Overview

Step 1: Compilation Targeting the Host



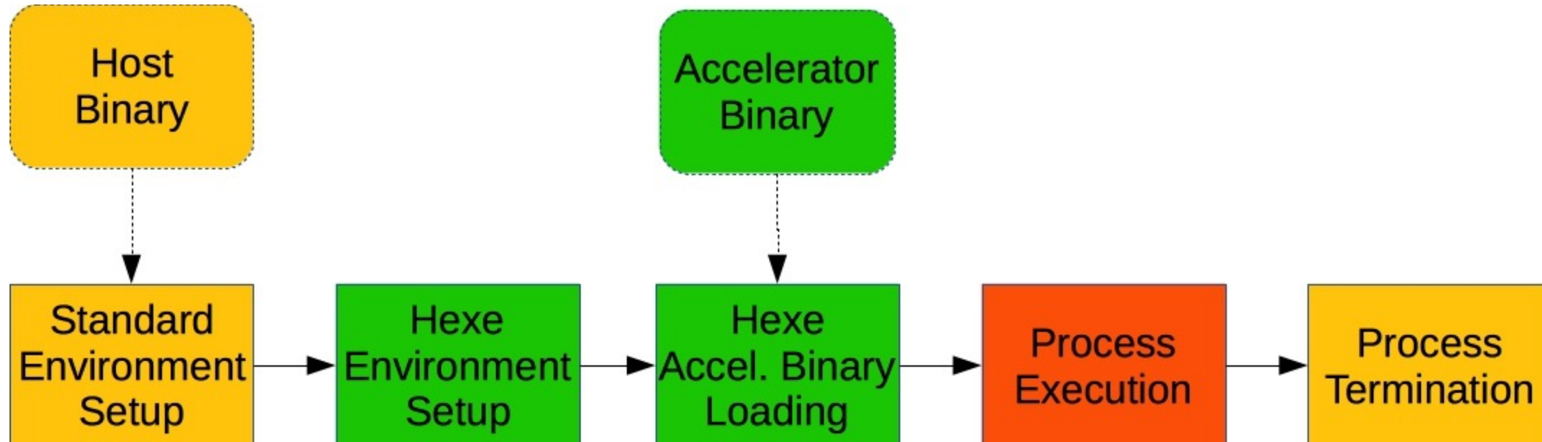
Step 2: Compilation Targeting the Device



Chris Margiolas chrmargiolas@gmail.com

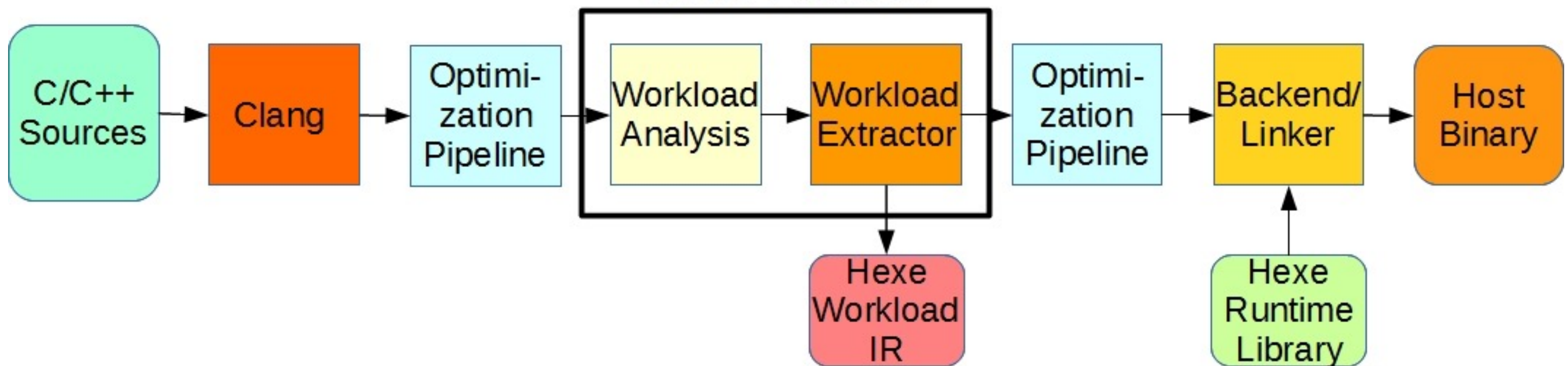
Hexe Execution Overview

Hexe Process Lifecycle:



- **Hexe runtime** handles the Host-Accelerator interaction.
- **Hexe runtime** manages the accelerator environment and loads the accelerator binary.
- **Hexe compiler transformations** inject calls to Hexe runtime library. These calls handle scheduling, data sharing and coherency.
- **Executable types** and their **Loading Procedure** is target dependent. They are handled by the appropriate runtime plugin.

Compilation For The Host



- Two new compiler passes, **Workload Analysis** and **Workload Extractor**.
- The application code is transformed to IR and optimized as usual.
- **Workload Analysis** detects Loops and Functions that can be offloaded.
- **Workload Extractor** extracts Loops and Functions for offloading (Hexe Workload IR), transforms the host code and injects Hexe Runtime calls.
- The IR is optimized again, compiled for the host architecture and linked against the Hexe Runtime Library.

Workload Analysis

- A Module Analysis Pass, Target Independent.
- It investigates the eligibility of **Workloads** for offloading.
- We consider as a **Workload** either (a) **a call to a function** or (b) **a loop**.
- Analysis assumptions:
  - Different Host and Accelerator architectures.
  - Different types of memory coherency may be available.
  - The origin of the input LLVM IR may be C/C++ (via clang), other high level languages or a Virtual Machine.

Analysis steps (for Loops and Functions):

1. Code Eligibility
2. Memory Reference Eligibility

Workload Analysis – Code Eligibility

Instruction Inspection:

- Host and Accelerator architectures can vary significantly in:
  - Atomic Operation Support.
  - Special instructions (a.k.a. LLVM Intrinsics).
  - Exception handling.

We **Do Not Support** the offloading of code containing:

- Atomics.
- Intrinsics. However, we relax this to support the core Memory Intrinsics of LLVM which are generated by front-ends or LLVM transformations.
- Function Calls. This could be supported in the future at some extent.
- Exceptions.

Workload Analysis – Memory Reference Eligibility (1)

Why to analyze memory references?

- We need to extract code to a new module. We need to make sure that this code still access valid memory.

We require a **Function** to **only access memory** via:

A. Its **Function Interface (pointer arguments).**

B. **Global Variables.**

We require a **Loop** to **only access memory** via:

A. Its **Host Function Interface (pointer arguments).**

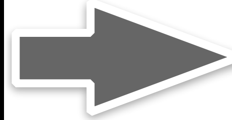
B. **Global Variables.**

We keep track of the Global Variables and Function Pointer Arguments for each Workload. This information is later used by the Workload Extractor.

Workload Analysis – Memory Reference Eligibility (2)

Example 1:

```
int funnycomp(int *array, unsigned index)
{
  return array[index/2] + 10;
}
```



Function Interface: array

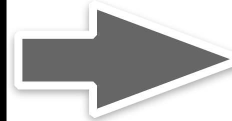
Global Vars: -

Valid Code to Offload

Example 2:

```
int GV=10;

int funnycomp(int *array, unsigned index)
{
  return array[index/2] + GV;
}
```



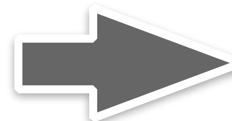
Function Interface: array

Global Vars: GV

Valid Code to Offload

Example 3:

```
int funnycomp(int *array, unsigned index)
{
  int *ip = (int *) 0xfffffff;
  return array[index/2] + 10;
}
```



Function Interface: array

Invalid: reference to 0xfffffff

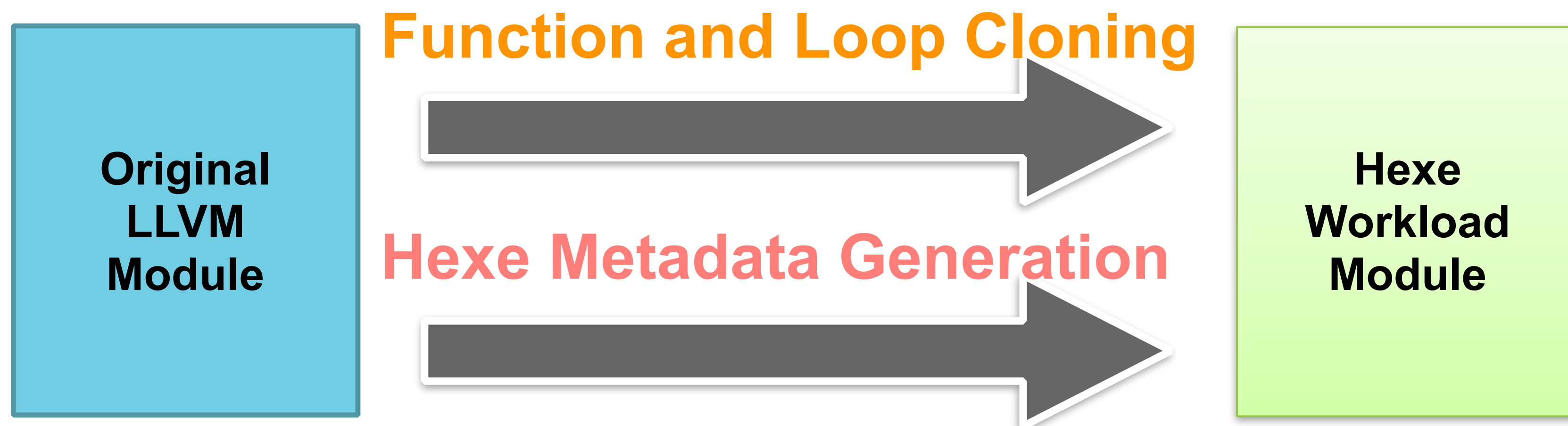
Invalid Code to Offload

Workload Extractor

- **Workload Extractor** is a Module Transformation Pass, which is Target Independent.
- We provide a set of Utility Classes that perform the following:
  - ➢Code Extraction and Cloning (for Loops and Functions).
  - ➢Host Code Transformation (To support workload offloading).
  - ➢Injection of Hexe runtime calls; they manage scheduling, offloading and data sharing. Their interface is platform independent.

- The **Workload Extractor** pass is built on the top of these utilities.
- The pass can be easily specialized to support specific use cases.
- Compiler flags control Workload Extraction.

Workload Extractor – Code Extraction and Cloning



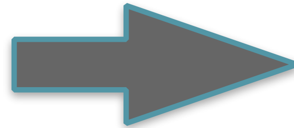
- We extract eligible Workloads (Loops and Functions) by cloning them to a separate Module named **Hexe Workload**.
- We preserve the original Workload code on the main module. The runtime scheduling may either offload a Workload or compute it on the CPU.

- A Loop is cloned to **Hexe Workload** in two steps:
  1. The Loop is extracted to a Function.
  2. The Function is then cloned to the **Hexe Workload**.

Workload Extractor – Host Code Transformation

Original BB

Instruction 1
Instruction 2
....
CallInst @F
....
Instruction N



Instruction 1
Instruction 2
....
Hexe_sched
Sched branch

Offloading BB

Enforce Coherency
Call Data Marshaling
Dispatch Workload
Wait for Completion
Enforce Coherency
Read Return Value

Host BB

CallInst @F

Merge BB

PhiNode (Ret. Value)
....
Instruction N-1
Instruction N

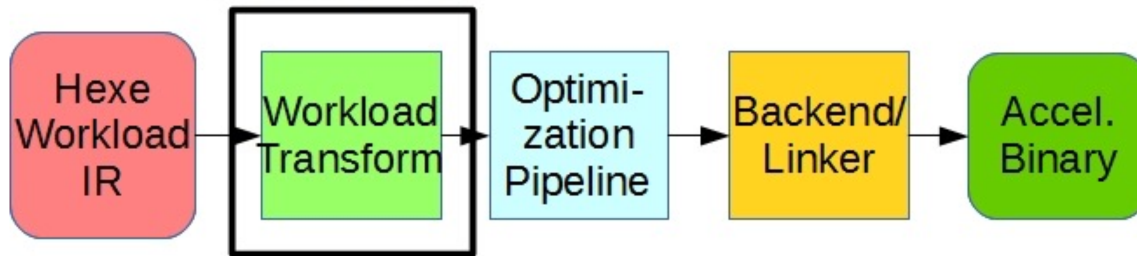
Function Call Offloading:

At this point, all the workloads are functions. We enable offloading at their call points.

We support automatic offloading by modifying the control flow and injecting calls to the runtime library.

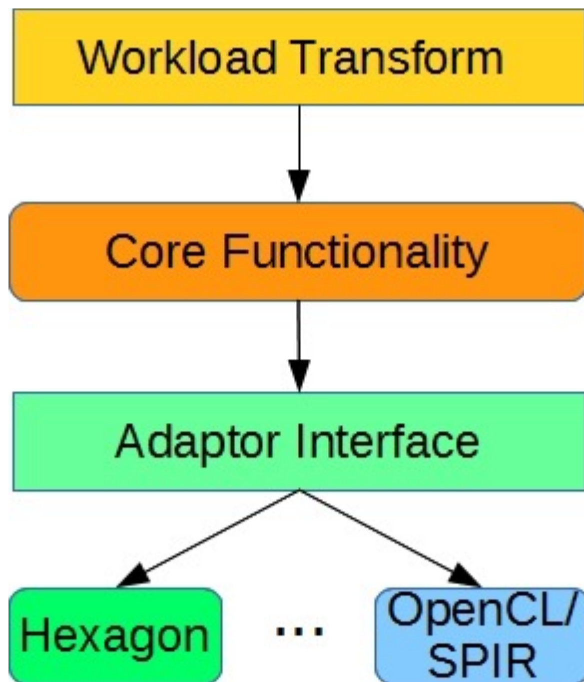
The runtime decides on the fly if the CPU or the accelerator will compute the workload.

Compilation For The Accelerator



- **Workload Transform**, a Module Transformation pass
- It transforms the code to guarantee compatibility with the target accelerator architecture. **Reminder:** The host and accelerator architectures may be quite different (e.g. 32 bit vs 64 bit, stack alignment, endianness, ABI etc).
- The IR is transformed to comply to a set of conventions defined by the accelerator toolchain (e.g. function interface, accelerator runtime calls).
- The IR is then optimized and an accelerator binary is generated. The binary type (e.g. elf executable, shared library etc) is accelerator specific.

Workload Transform Pass Adaptors



Key Design:

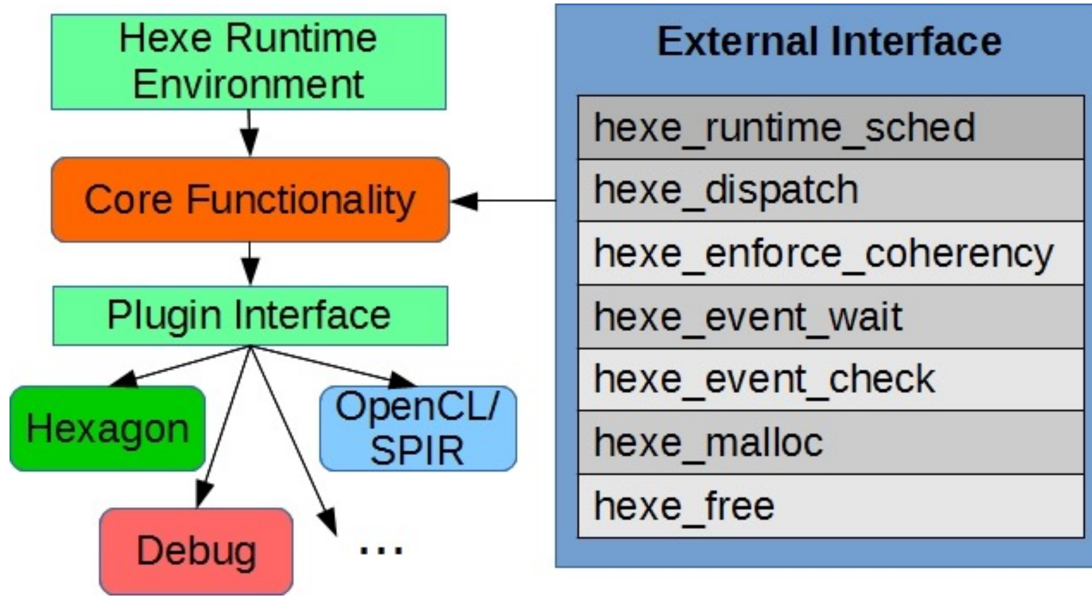
- Core design remains accelerator and platform independent.
- An **Adaptor** performs the required accelerator specific transformations. **Remainder**: These transformations handle compatibility transformations and toolchain conventions.

Available Adaptors:

- **Debug Adaptor.** It supports debugging and correctness runs. It transforms the Workload code for offloading on the host architecture.
- **OpenCL Adaptor.** It transforms Workload code for execution on accelerators that support OpenCL.
- **Hexagon Adaptor.** It transforms Workload code for execution on Qualcomm DSPs.

Chris Margiolas chrmargiolas@gmail.com

Hexe Runtime Environment



Key Design:

- Core design remains accelerator and platform independent.
- The runtime exposes a standard interface which is called by the application.
- A plugin interface is defined.
- Plugins support individual accelerators and platforms.

Features:

- Low overhead runtime, written exclusively in C.
- Minimalist design. This component may run on embedded systems.
- Plugin interface built with virtual tables (Linux kernel style).
- It handles scheduling, memory management, data sharing and coherency.
- The design considers asynchronous offloading (not supported yet).

Hexe Runtime Environment - Data Sharing Management

Platforms with varying data sharing capabilities:
- No memory sharing or coherency (explicit data copies are required).
- Hardware/Driver Support for Data Sharing (Virtual Shared Memory).
- Special Memory Pools (for data sharing or high performance).

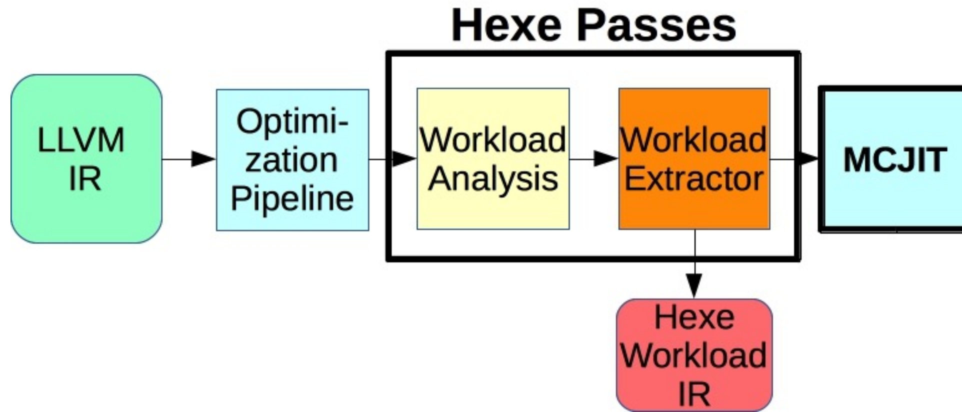
Based on the platform capabilities, Hexe may:
- Track host program memory allocations.
- Handle explicitly data transfers.
- Utilize special memory pools.
- Rely on hardware/driver assisted data sharing (HSA, CUDA Unified Addressing, OpenCL Shared Memory).

These action are managed by the runtime and its plugins. Every plugin should specialize its behavior based on the platform capabilities.

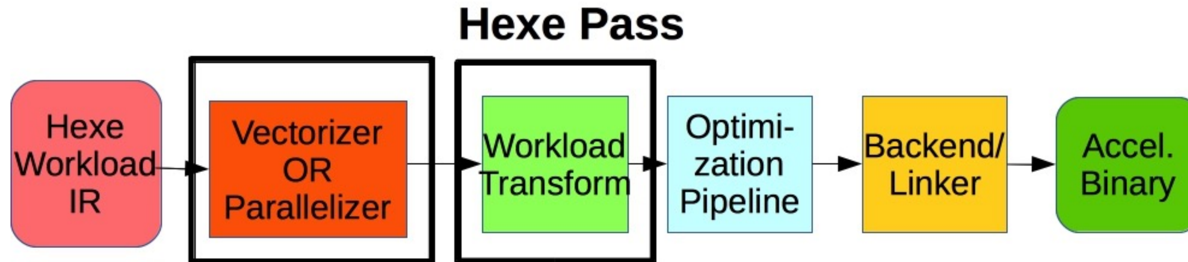
Current paradigms are the **OpenCL, Debug** and **Hexagon** plugins.

Hexe Integration with Third Parties

Hexe and Just in Time Compilation (Host):



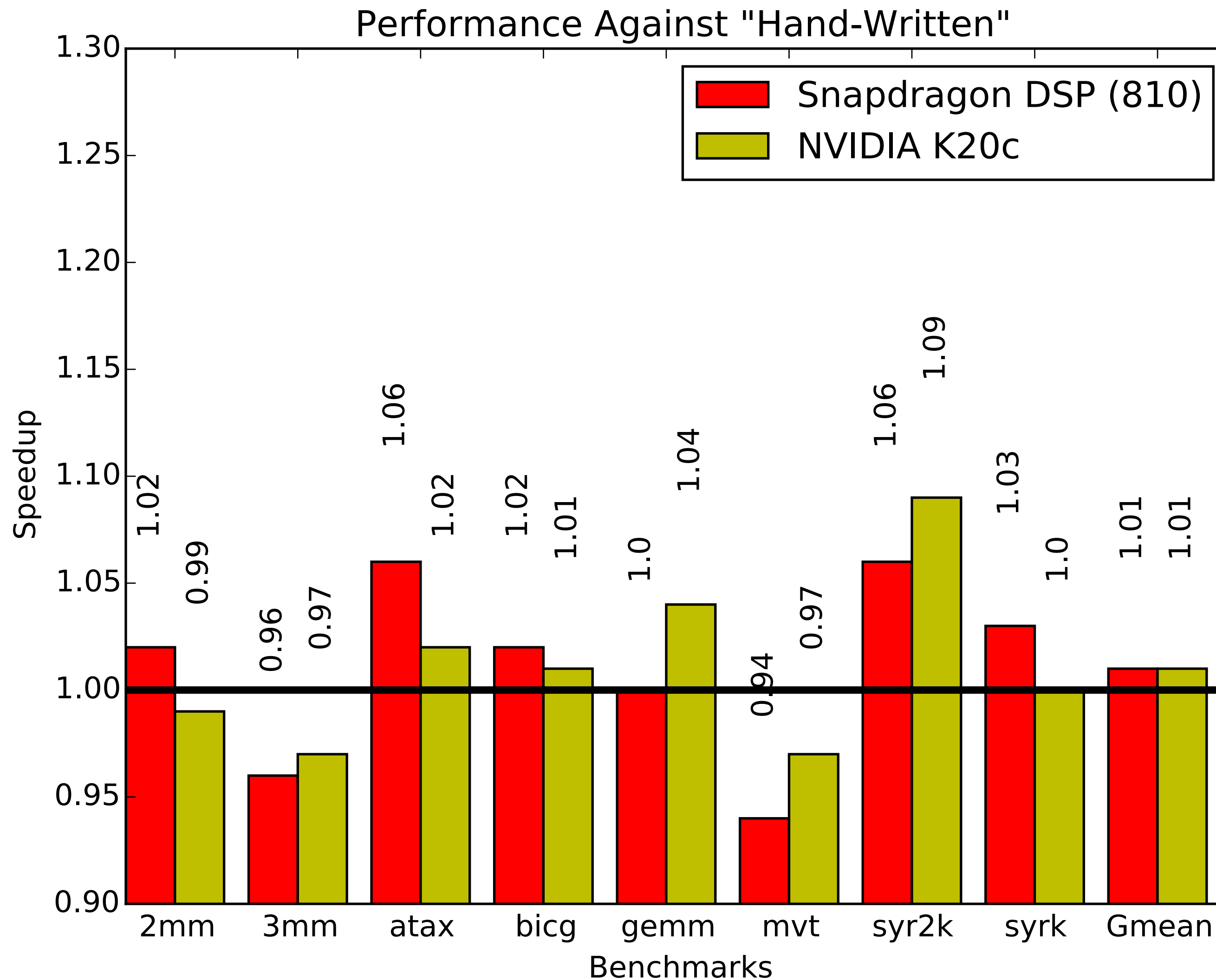
Vectorization and Auto-parallelization (Accelerator):



Combining Hexe with MCJIT is easy. However, picking the right vectorization and parallelization infrastructure and strategy per accelerator is tricky.

Chris Margiolas chrmargiolas@gmail.com

Comparing against the Developer



Baseline: Developer's port to the accelerator environment

Chris Margiolas chrmargiolas@gmail.com

Conclusion

This presentation is about Hexe, a heterogeneous execution engine.

It comprises the following:

- Compiler Passes for Workload Analysis and Extraction.
- Runtime Environment (Scheduling, Data Sharing and Coherency etc).
- Modular Design. Core functionality independent of the accelerator type. Specialization via Plugins both on the compiler and runtime.
- Extends the LLVM infrastructure.

Chris Margiolas chrmargiolas@gmail.com