WebAssembly support for clang-repl

Anubhab Ghosh
Mentors: Vassil Vassilev, Alexander Penev
What is Web Assembly?

- Assembly inside a web browser ?!
- Modern Javascript engines rely heavily on JIT compilation of Javascript to native code before execution.
  - There is a trade-off between time to compile vs execution performance improvement.
    - Most engines compile portions of “hot” code after they are executed it several times.
- WebAssembly avoids all of that!
  - Precompiled to an intermediate representation
  - Compact binary representation for fast parsing
  - Simple to execute in an abstract virtual CPU
    - Sandboxed for security and portability
  - Can be easily compiled to native code
How does it actually look like?

- It follows Harvard architecture
  - Instructions and data are strictly separate. Similar to microcontrollers.
  - This has serious consequences for JIT!
- It runs completely sandboxed.
  - By default there is no interface/API for talking to world outside VM.
  - Exports functions that can be called from outside.
  - Imports functions from outside that can be called.
- It is a stack machines.
  - No registers!
  - All computations is performed on the top of the implicit stack.
The architecture

- Linear Data memory is accessible from WASM and JS
- Code is divided into variably sized functions
  - The actual internal representation is on the engine
  - WASM code refers to functions by an index which acts like an opaque handle
  - No way to add new functions from WASM code
- No JIT code generation possible from WASM!
Taking help from JS side

- We can generate a new WASM module and give it to the Javascript “runtime”.
- The runtime can instantiate a new WASM Instance with this module.
- Linear memory can be trivially shared between modules.
- Old module can export its functions that the new module imports. Failing that, JS runtime can provide a transparent RPC service.
  - Cross module calls won’t have the best performance.
  - To achieve all of this, we probably have to take most of clang-repl and libInterpreter functionality from C++/WASM into the Javascript runtime.
The Plan

**Generate WASM**
Produce WASM code in clang-repl. This should be similar to generating CUDA device code. We can use the LLVM WebAssembly target. We skip the execution part and output the generated code.

**Inside WASM**
Compile Clang for WASM. Run the JIT within a Javascript engine. Display the generated code.

**Execute**
Create full WASM modules within the engine and export it. Let the JS runtime execute independent bits of code (no linking/shared memory required).

**Link**
Generate and execute code that depends on previous modules or standard library. This is the trickiest part.

**UI**
Integrate within JupyterLite. Possibly provide some convenience functions to use in a notebook?
Thank you