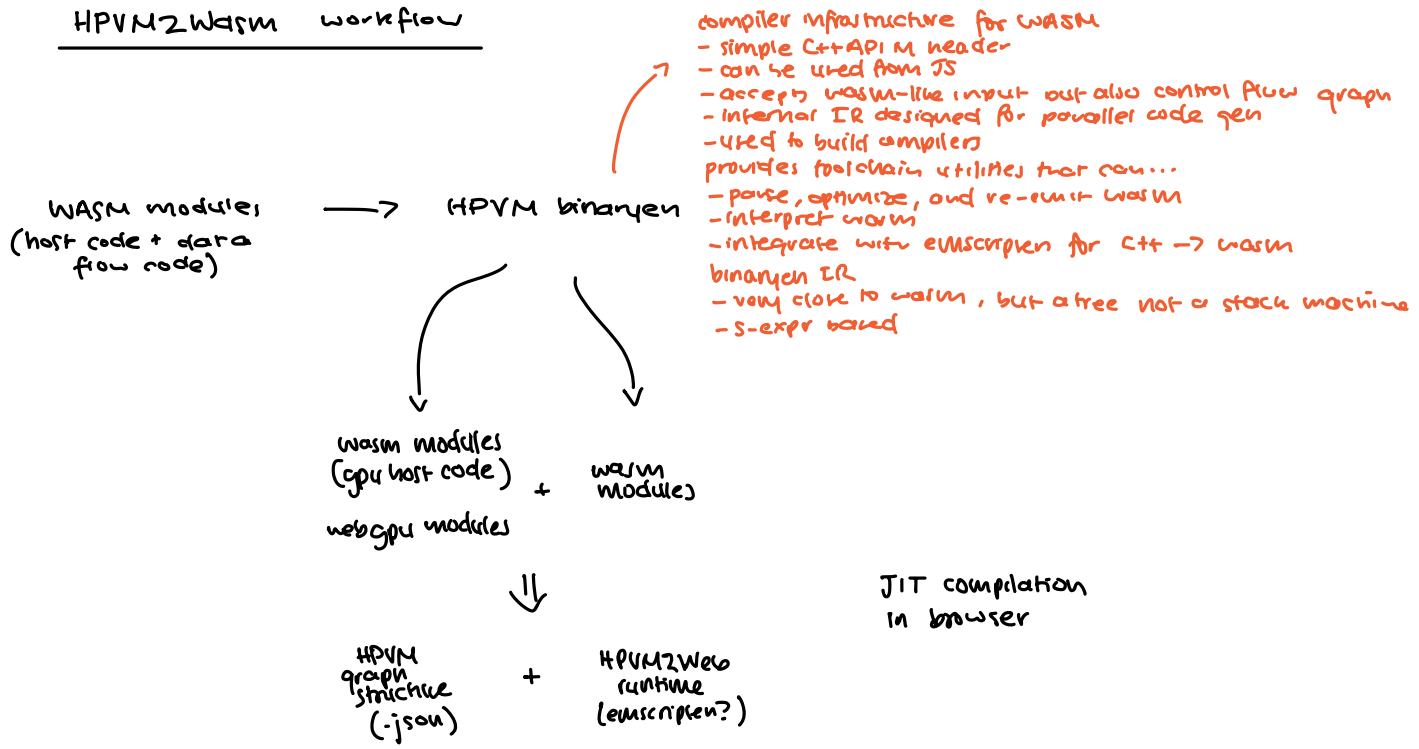
HPVM2Wasm workflow



RUNTIME

amcc_rt.js included in all the example dirs

- built on top of emscripten runtime for web assembly
  - JS runtime environment
- implements subset of execution semantics of HPVM data flow graph
  - nodes fire when ready
  - hierarchical DAG execution
  - dynamic node replication
- each HPVM leaf node is a separate wasm module
  - node JIT compiled when ready (using HPVM binaryen compiler GPU code gen)
  - custom memory tracker and management for copying memory
- nodes executed asynchronously using JS promises
- automatic compilation support for synchronous C/C++ code with async JS (don't understand this)
  - emscripten asyncify compilation flow
  - overload HPVM host code API to invoke JS

I/O
- uses SDL for browser environment
- applications that use SDL require no change

File systems
- browser prevents direct host file access
- emscripten provides own libc and libcxx
- creates own virtual file system

Browser main loop
- each event runs a turn and then returns control (cooperative multitasking)

Execution lifecycle
- preloading phase:
- run phase

Memory Representation
- pointers as offsets

allows sync C++ code to interact w/ async JS
- transforms compiled code into code that can be paused and resumed
- handles the pausing + resuming
- implemented as a binaryen pass
- alternatives: coroutines (not supported by wasm) and threading (not supported by all browsers)
- sleep actually returns control (for example)

Graph Description

- dataflow graph encoded in json ← RN the user writes but ideally generated HPVM could generate easily
- each leaf node specifies WASM module
- json parsed by runtime, which manages heterogeneous compilation + memory

Host code API

- defined in hpvm2wasmRT.js
- defined entirely in JS and imported into host code module with Wasm

Wasm to WGSL ← WebGPU shading language

Aaron implemented pass that operates on sexpr
- takes wasm → cfg
- tries to identify patterns
- to blocks for WebGPU
- block structured?
- pointer access

- implemented as Binaryen pass
  - Aaron implemented Binaryen pass
  - operates on the s-expr format of Binaryen IR
  - takes wasm to a CFG
  - identifies patterns to build block structure for WebGPU (big task #1)
  - WebGPU doesn't support pointer arith, re-write pointer access (big task #2)
  - this is compiled to binaryen.js
- translates a single specified function to a WebGPU kernel

in other words...

  - translation process:
    1. construct control flow graph
    2. compute dominance (?)
    3. identify back edges + loops
    4. identify if-then-else
    5. generate code for each basic block
       - array access reconstruction (?)
    6. glue basic blocks together w/ appropriate control flow

happens w/in binaryen pass

WebGPU runtime ← nodes can be marked for gpu — wasm runtime considers this marking + may send node here (in execute_node())

- JS code
- creates buffers for each array argument and buffer of scalar args
- produces shader module
- binds buffer into the shader (copies memory over?)
- executes the shader
- copies data back from GPU

Future work

- additional accelerator backends ← have gpu + cpu support but not for other accel. (like ML accel)
- integration with browsers?
- compilation from hetero C++ to hpvm2wasm
- process virtualization
  (same physical machine but isolated envs)
- dynamically targeting nodes to devices (at runtime?)
- other optimizations before code gen

Translating hetero-clt to hpvm2web

Big Action Items:

1) generate .json from HPVM
2) kernels in separate files... is this necessary?
3) lower HPVM intrinsics

Project for the rest of semester:

- hetero-clt to web assembly for CPU only
- involves creating the wasm modules and integrating the JS runtime
- probably need to do some work in the runtime

What does the hpvm2web matmul look like?

1) matmul.json
   - describes the graph (should be easy to generate from HPVM)