HPVM2Web
Heterogeneous Compilation for the Web

Aaron Councilman aaronjc4@illinois.edu
Abdul Rafae Noor arnoor2@illinois.edu

CS598 LCE

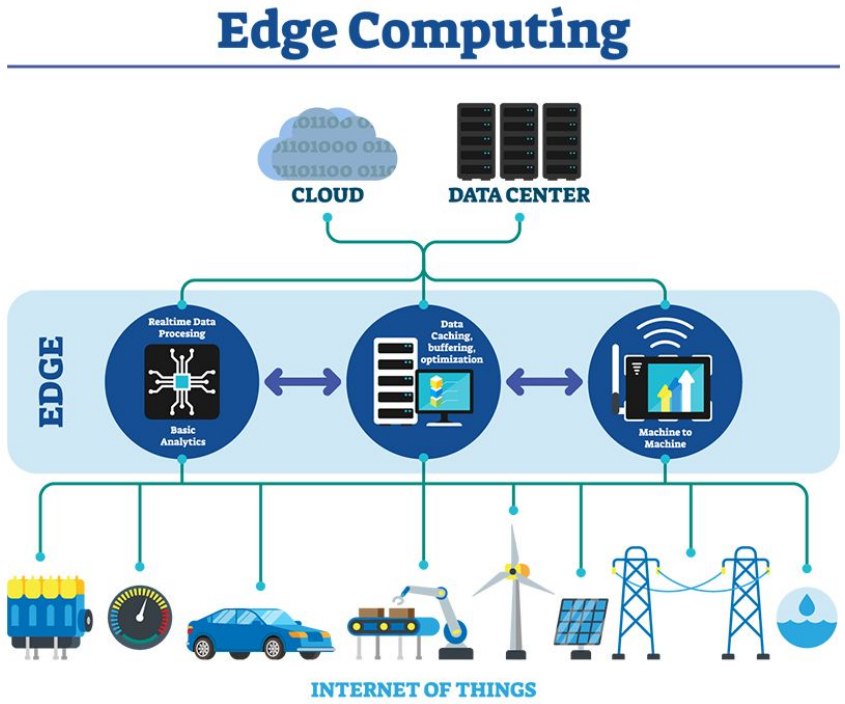
UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Motivation

- Computation at the edge is becoming ubiquitous
  - Types of applications increasingly complex
  - e.g. The Augmented Reality Edge Networking Architecture – ARENA

- In addition to performance requirements, applications must
  - Be as compact as possible (code-size)
  - Provide safe execution environment (e.g. memory safety)
  - Portable across hardware
  - Leverage heterogeneity when available

- Diversity in IOT hardware limit the portability of these applications
  - Require vendor specific programming with low-level API's
    - e.g. Apple's Metal, Vulkan, etc.
  - Code Size vs Portability
  - Number of available hardware configurations is combinatorial



https://innovationatwork.ieee.org/real-life-edge-computing-use-cases/

HPVM2Web

- Just-In-Time compilation for heterogeneous systems
  - Ship code as WebAssembly and JIT for hardware target
  - Particularly for use within web browsers
    - Built-in support for JIT to CPU in modern browsers
    - Support for modern GPU APIs via WebGPU
    - Browser sandboxes provide isolation

- Uses the model of Heterogeneous Parallel Virtual Machine (HPVM)
  - Explicit data-flow graph between leaf/kernel nodes
  - Has shown success in targeting CPU, GPU, and FPGAs from hardware-agnostic LLVM IR
  - Currently, HPVM uses ahead-of-time compilation; distinct binaries based on hardware
  - HPVM2Web can serve as a portable object-code back end for HPVM

- Use of hardware-agnostic binary format allows recompilation to hardware based on availability at run or load-time
  - Developer compiles their code once, never considers hardware aspects

WebAssembly (WASM)

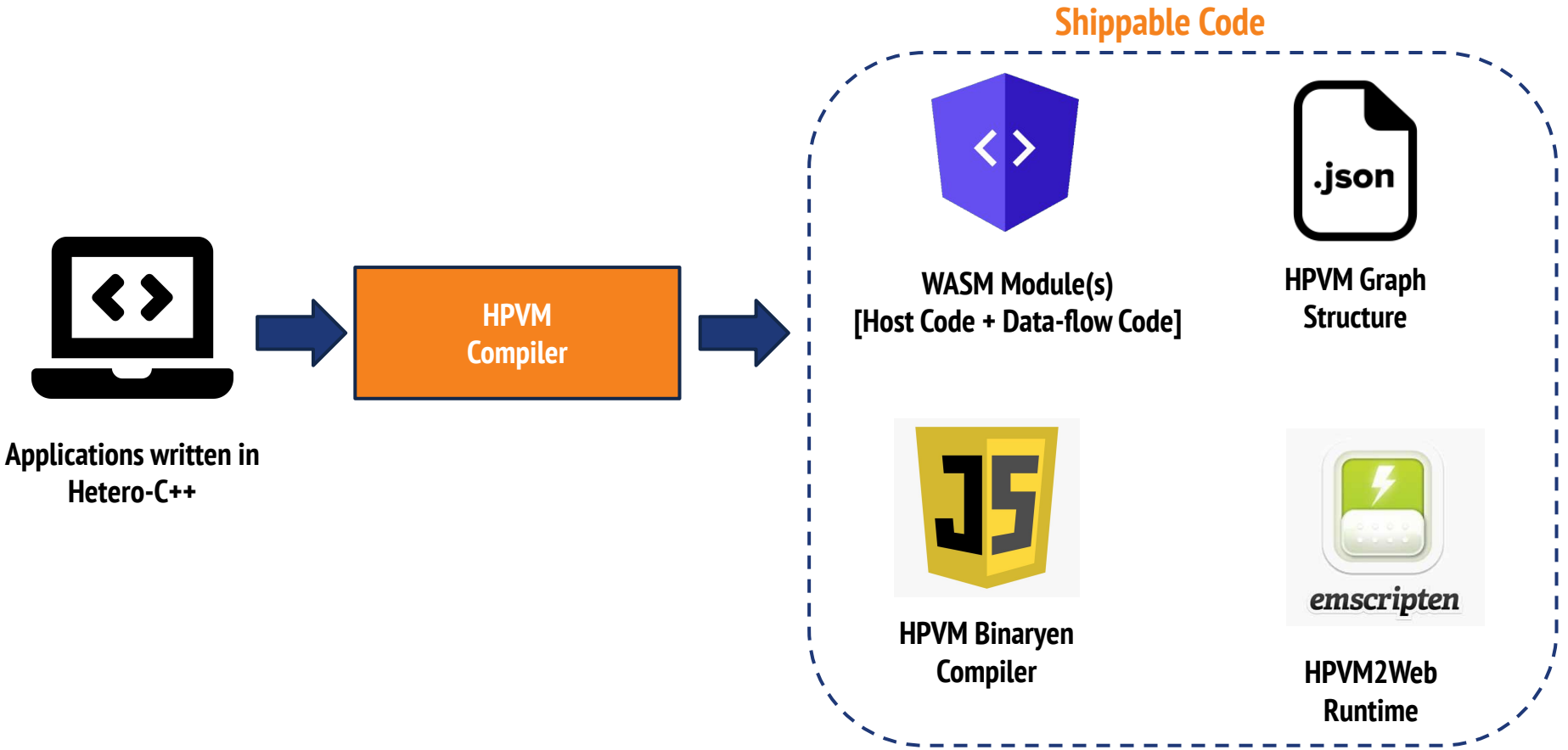
- Binary instruction format for Stack-based Virtual Machine
- Lightweight and Portable by design
- Memory-safe, sandboxed execution environment
- Frontends in 30+ languages including C/C++, JavaScript and Rust [1]
- Primarily designed for web-based applications on browsers

[1] https://github.com/appcypher/awesome-wasm-langs

WebGPU

- API for using GPU rendering and computation on the Web
  - JavaScript API with work on support from C/C++ and Rust
  - Successor of WebGL, designed for more modern GPU APIs (Metal, Vulkan, Direct3D 12)
  - Application communication to GPU is independent of underlying hardware, drivers, and APIs
- Kernels expressed in WebGPU Shading Language (WGSL)
  - Imperative, strong statically typed
  - Structured control flow, no pointer arithmetic
- Currently in development, implementations remain experimental and unstable

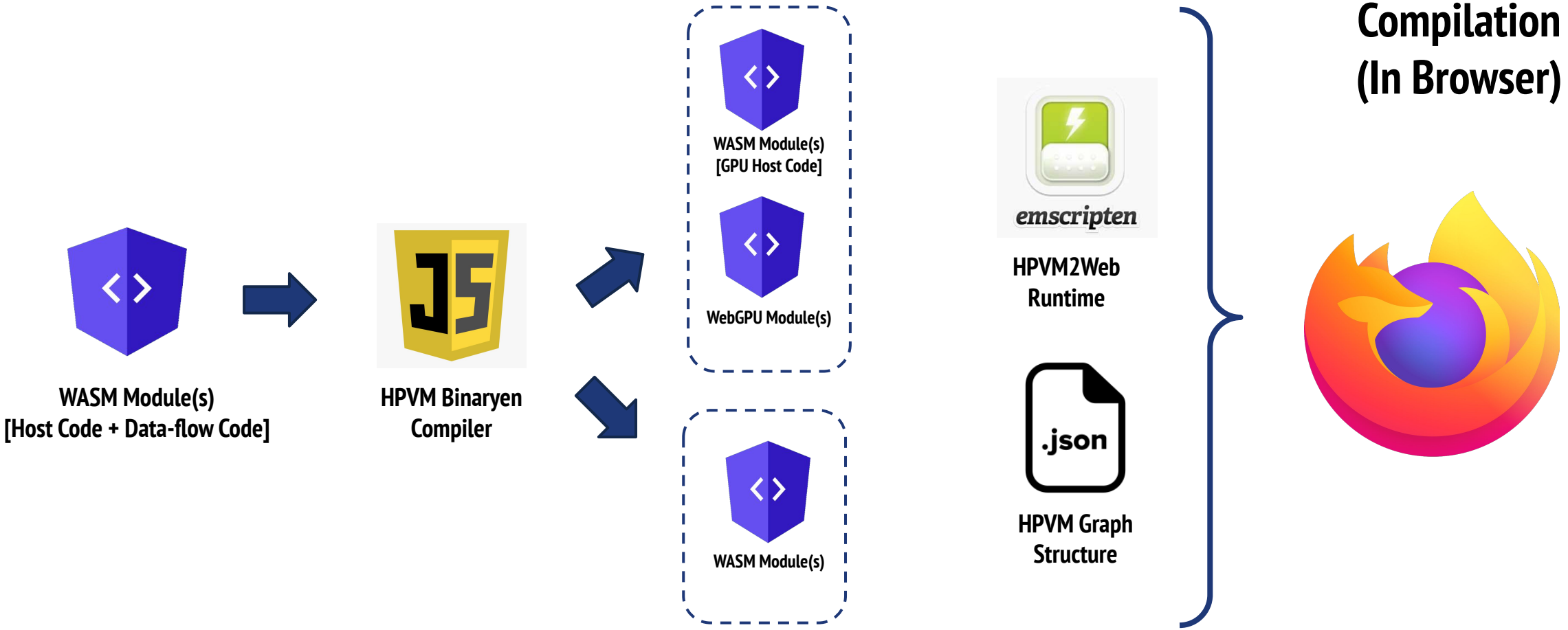
HPVM2Web Workflow



Ahead of Time Compilation

**Integration into HPVM Compiler not yet implemented but included for future vision*

HPVM2Web Workflow



HPVM WASM Runtime

- JavaScript runtime environment for executing HPVM programs on the Web
  - Built on-top of Emscripten runtime for WebAssembly [1]
- Implements subset of the execution semantics of HPVM data-flow graphs
  - Node's fire when their arguments are ready
  - Hierarchical execution of DAG's
  - Dynamic replication of nodes
- Each HPVM leaf node is a separate WASM module
  - JIT compilation of each node when it's ready to execute
    - HPVM Binaryen compiler GPU CodeGen
  - Custom memory tracker and management for copying buffers across address spaces
- Nodes executed asynchronously using JavaScript Promises
- Automatic compilation support for synchronous C/C++ code with Asynchronous JS
  - Emscripten's Asyncify compilation flow [2]
  - Overload HPVM host code API to invoke JavaScript

[1] https://emscripten.org/index.html
[2] https://emscripten.org/docs/porting/asyncify.html

HPVM WASM Graph Description


```
{
  "num_nodes" : 4,
   "nodes" : {
    "node1" :{
      "type" : "leaf",
      "wasm_module": "leaf_nodes/node1.wasm",
      "num_inputs": 3,
      "output_indices": [0,1],
      "replication_factors": ["arg2",1,1],
      "consider_gpu": true,
      "arg_types": "w:i32*,r:u32,r:u32,i:x,i:y,i:z,d:x,d:y,d:z"
    },
        ...
    "wrapper" : {
      "type" : "internal",
      "graph_inputs" : 5,
      "graph_output_indices": [0,1,2,3,4],
      "immediate_children" : ["node1", "node2", "node3"]
    }
   },
   "edges": [
   {"edge_type": "bindin" , "from": "wrapper" , "to" : "node1", "src_idx": 0 , "dst_idx": 0 },
   {"edge_type": "sibling" , "from": "node1" , "to" : "node3", "src_idx": 1 , "dst_idx": 1 },
   ...
   {"edge_type": "bindout" , "from": "node3" , "to" : "wrapper", "src_idx": 3 , "dst_idx": 3 }
   ]
}
```


HPVM Graph Structure JSON

- HPVM Dataflow Graph Structure encoded in JSON format
  - One-to-One mapping from HPVM intrinsics
- Each Leaf node specifies WASM module with inputs and outputs
  - Explicit argument types for WebGPU CodeGen
- Parsed by HPVM WASM RT to manage heterogeneous code compilation and memory management

HPVM WASM Leaf Nodes

```
// HPVM leaf node function, for tone mapping
EMSCRIPTEN_KEEPALIVE
void tone_map_fxp(float *restrict input, size_t bytes_input,
                  float *restrict result, size_t bytes_result,
                  float *restrict tone_map, size_t bytes_tone_map,
                  size_t row_size_, size_t col_size_,
                  size_t idx_x, size_t idx_y, size_t idx_z,
                  size_t dim_x, size_t dim_y, size_t dim_z) {


  long chan = idx_x;            //__hpvm__getNodeInstanceID_x(thisNode);

  long row = idx_y ;            //__hpvm__getNodeInstanceID_y(thisNode);

  long col = idx_z ;            //__hpvm__getNodeInstanceID_z(thisNode);

  long row_size = dim_y;        //__hpvm__getNumNodeInstances_y(thisNode);
  long col_size = dim_z;
//__hpvm__getNumNodeInstances_z(thisNode);
  int index = (chan * row_size + row) * col_size + col;
  uint8_t x = input[index] * 255;
  result[index] = tone_map[x * CHAN_SIZE + chan];
}
```

- Leaf node function's arguments extended to include:
  - Index along 3D grid of parallel execution
  - Replication factors along 3D axes

HPVM WASM Host Code

```
#include "hpvm_wasm.h"

…

// Launches asynchronously executed DAG in graph.json with
// specified arguments
void* DAG = __hpvm__wasm__launch(
                        /* Num Inputs */      2, ptr,  ptr_size, ptr2 , ptr_size,
                        /* Num Scalars */     1 ,num_inputs,
                        /* DAG JSON Path */  "graph.json");


// Blocking call waiting for completion of DAG execution
__hpvm__wasm__wait(DAG);

// Request updated memory contents from HPVM WASM
// runtime to be copied back into current address space.
__hpvm__wasm__request__mem(ptr, ptr_size);
__hpvm__wasm__request__mem(ptr2, ptr_size);

…
```

- Define host-code API to launch Data-flow code
- WASM has no explicit pointer type
  - Treats pointer as integer type
  - Separately list pointers and scalars
- Definition of API entirely in JavaScript
  - Imported into host-code module using WebAssembly

WASM to WGSL

- Implemented as a pass in Binaryen [1]
  - Binaryen's S-expression IR simplifies analyses, handles parsing of WASM code

- Translates single specified function to WebGPU kernel
  - Requires input of argument types and read/write annotations

- Translation process:
  1. Construct CFG
  2. Compute dominance
  3. Identify back-edges and the loops they form
  4. Identify if-then-else
  5. Generate code for each basic block
     a. Array access reconstruction
  6. Glue basic blocks together with appropriate control-flow

[1] https://github.com/WebAssembly/binaryen

HPVM WebGPU Runtime

- JavaScript code to create buffers for each array argument and a buffer of scalar values
- Produces shader model from the kernel code
- Binds the buffers into the shader and executes the shader
- Reads data back from the GPU similar to the HPVM WASM runtime

Matrix Multiplication Example

```
fn main(@builtin(global_invocation_id) global_id : vec3<u32>) {
  /* local initialization, elided */
  if (bool(local1)) {
    let tmp8 : u32 = u32(local9) * u32(local1);
    local16 = tmp8;
    let tmp9 : f32 = 0.000000;
    local15 = tmp9;
    let tmp10 : i32 = 0;
    local17 = tmp10;
    loop {
      /* elided */
      let tmp29 : i32 = 1;
      let tmp30 : i32 = i32(local17) + i32(tmp29);
      local17 = tmp30;
      let tmp31 : bool = u32(local17) != u32(local1);
      if (bool(tmp31)) {
        continue;
      } else {
        break;
      }
    }
  } else {
    let tmp11 : f32 = 0.000000;
    local15 = tmp11;
  }
  let tmp1 : i32 = 0;
  let tmp2 : u32 = u32(local9) * u32(local2);
  let tmp3 : u32 = u32(tmp2) + u32(local10);
  let tmp4 : i32 = 2;
  let tmp5 = u32(tmp3) << u32(tmp4);
  let tmp6 : u32 = u32(tmp1) + u32(tmp5);
  let idx7 = tmp6 / u32(4);
  arg7.value[idx7] = local15;
  return;
}
```

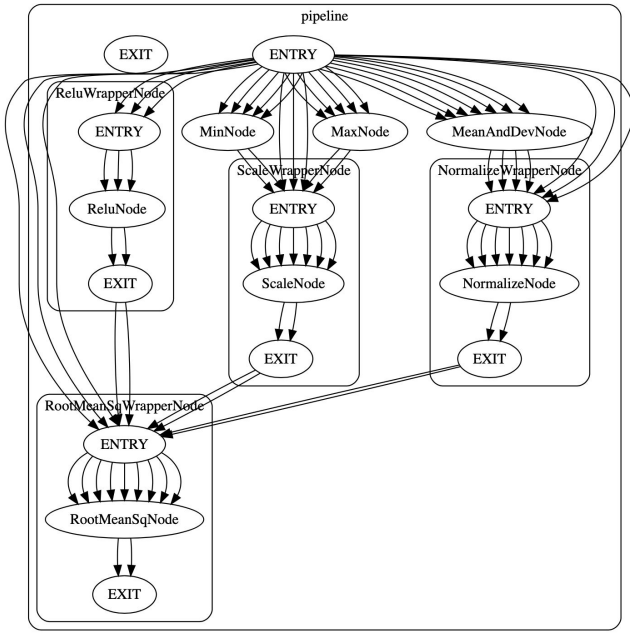
```
signed dimR, float* A, si
, size_t sizeC,
signed idxZ,
signed dimZ) {


idxY];
```

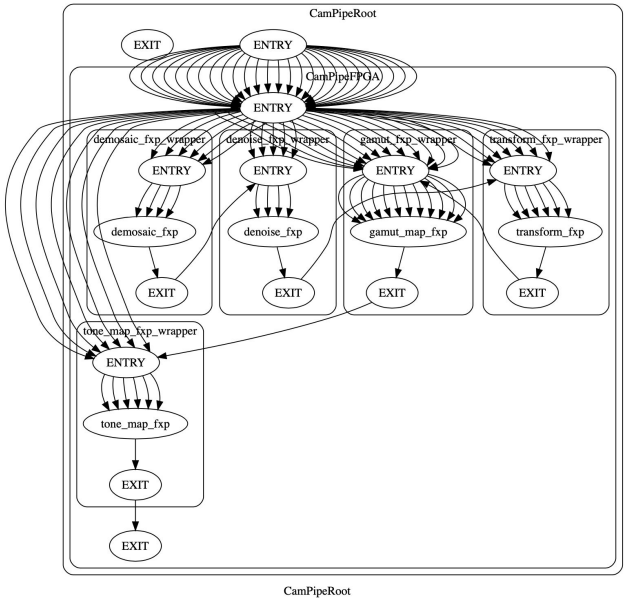
```
(func $1 (param $0 i32) (param $1 i32) (param $2 i32) (param $3 i32)
  (param $4 i32) (param $5 i32) (param $6 i32) (param $7 i32) (param $8 i32)
  (param $9 i32) (param $10 i32) (param $11 i32) (param $12 i32)
  (param $13 i32) (param $14 i32)
  (local $15 f32) (local $16 i32) (local $17 i32)
  (block $label$1
    (block $label$2
      (br_if $label$2 (local.get $1))
      (local.set $15 (f32.const 0))
      (br $label$1))
    (local.set $16 (i32.mul (local.get $9) (local.get $1)))
    (local.set $15 (f32.const 0)) (local.set $17 (i32.const 0))
    (loop $label$3
      (local.set $15 (f32.add <elided> (local.get $15)))
      (br_if $label$3
        (i32.ne
          (local.tee $17 (i32.add (local.get $17) (i32.const 1)))
          (local.get $1)
        )
      )
    )
  )
  (f32.store
    (i32.add
      (local.get $7)
      (i32.shl
        (i32.add (i32.mul (local.get $9) (local.get $2)) (local.get $10))
        (i32.const 2)))
    (local.get $15)
  )
)
```

Evaluation Setup

- Firefox Nightly version 101.a1 (2022-04-11)
  - i7-7700HQ
  - NVIDIA 1050-Ti
- Baseline: HPVM Version 2.0
  - NVIDIA GeForce RTX 2080 Ti
- Emscripten Version 3.1.8
- Benchmarks
  - HPVM Cava Pipeline
  - Hetero-C++ Pipeline
  - Matrix Multiplication Kernel



Hetero-C++ Pipeline



HPVM Cava Pipeline

Evaluation Results

Benchmark	HPVM [CPU] Execution Time / seconds	HPVM [GPU] Execution Time / seconds	HPVM-WASM [CPU] Execution Time / seconds	HPVM-WASM [GPU] Execution Time / seconds
HPVM CAVA Pipeline *(Large Inputs)*	26.3	0.51	-	-
HPVM CAVA Pipeline *(Small Inputs)*	0.73	0.017	5.8	-
Hetero-C++ Pipeline	0.17	0.008	0.56	-
Matrix Multiplication	0.45	0.56	2.5	0.9

Conclusion and Future Work

- Presented HPVM2Web, a Just-in-time compiler for the Web
  - HPVM WASM RT and HPVM WebGPU RT

- **Future Directions**
  - Back ends for additional accelerators
    - Design and integration with browsers to support safely breaking the sandbox
  - Compilation to HPVM2Web from HeteroC++
  - Process Virtualization using WASM
  - Dynamically targeting nodes to devices
  - Other optimizations before CodeGen
    - Graph Transformations depending on target environment

Questions

Division of Work

Sorry, added this after the presentation because we forgot to mention it during the presentation

- Rafae implemented the WASM runtime and handling of host-code, as well as the classes for representing the graph. Rafae also created the examples we evaluated on
- Aaron implemented the WASM to WGSL Binaryen pass and the WebGPU runtime to support it, and then integrated it into the rest of the runtime. Aaron created some examples for testing the WebGPU compilation and runtime