Lec19: WebGPU/WGSL and Triton

WebGPU/WGSL

- WebGPU API enables web apps to use GPU to do high performance **computation** and graphics
- The successor of WebGL/GSGL, which mostly focused on graphics in the same way as OpenGL
- Quite new, still evolving, but already supported on major browser environment on every major OS + GPU combination, including mobile.

Getting context and GPU device

```
async function initWebGPU() {
    if (!navigator.gpu) {
        console.error("WebGPU not supported on this browser.");
        return;
    }

    const adapter = await navigator.gpu.requestAdapter();

    if (!adapter) {
        console.error("No WebGPU adapter found.");
        return;
    }

    const device = await adapter.requestDevice();

    if (!device) {
        console.error("No WebGPU device found.");
        return;
    }

    console.log("WebGPU initialized successfully!");
    // You now have a 'device' object to work with WebGPU commands
    // The next step would typically be to configure the canvas context
    // and start creating buffers, shaders, etc.

    return device;
}
```

Create a shader module

```
const module = device.createShaderModule({
    label: 'doubling compute module',
    code: `
  @group(0) @binding(0) var<storage, read_write> data: array<f32>;

  @compute @workgroup_size(1) fn computeSomething(
    @builtin(global_invocation_id) id: vec3u
  ) {
    let i = id.x;
    data[i] = data[i] * 2.0;
  }
 `,
});
```

Compute Pipeline and Memory Allocation

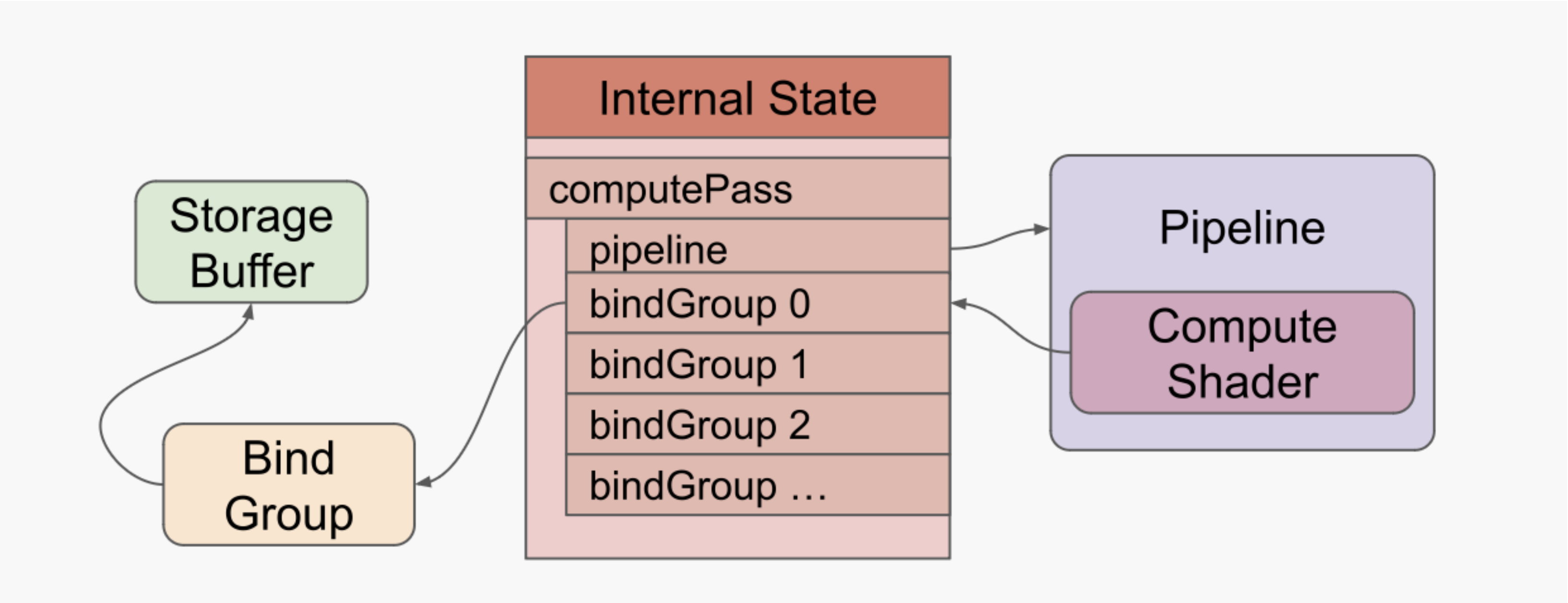
```
// create pipeline
const pipeline = device.createComputePipeline({
    label: 'doubling compute pipeline',
    layout: 'auto',
    compute: {
        module,
    },
});
const input = new Float32Array([1, 3, 5]);
// create a buffer on the GPU to hold our computation
// input and output
const workBuffer = device.createBuffer({
    label: 'work buffer',
    size: input.byteLength,
    usage: GPUBufferUsage.STORAGE | GPUBufferUsage.COPY_SRC | GPUBufferUsage.COPY_DST,
});
// Copy our input data to that buffer
device.queue.writeBuffer(workBuffer, 0, input);
```

Result Buffer & Bind Group

```
// create a buffer on the GPU to get a copy of the results
const resultBuffer = device.createBuffer({
    label: 'result buffer',
    size: input.byteLength,
    usage: GPUBufferUsage.MAP_READ | GPUBufferUsage.COPY_DST
});
// Setup a bindGroup to tell the shader which
// buffer to use for the computation
const bindGroup = device.createBindGroup({
    label: 'bindGroup for work buffer',
    layout: pipeline.getBindGroupLayout(0),
    entries: [
        { binding: 0, resource: { buffer: workBuffer } },
    ],
});
```

Encoding Commands

```
// Encode commands to do the computation
const encoder = device.createCommandEncoder({
    label: 'doubling encoder',
});
const pass = encoder.beginComputePass({
    label: 'doubling compute pass',
});
pass.setPipeline(pipeline);
pass.setBindGroup(0, bindGroup);
pass.dispatchWorkgroups(input.length);
pass.end();
```


Internal State
computePass
pipeline
bindGroup 0
bindGroup 1
bindGroup 2
bindGroup …
Storage Buffer
Bind Group
Pipeline
Compute Shader

Submit command queue and get result

```
// Encode a command to copy the results to a mappable buffer.
encoder.copyBufferToBuffer(workBuffer, 0, resultBuffer, 0, resultBuffer.size);
// Finish encoding and submit the commands
const commandBuffer = encoder.finish();
device.queue.submit([commandBuffer]);
// Read the results
await resultBuffer.mapAsync(GPUMapMode.READ);
const result = new Float32Array(resultBuffer.getMappedRange());
console.log('input', input);
console.log('result', result);
resultBuffer.unmap();
```

WGSL Shader Language

- Strictly typed
- https://google.github.io/tour-of-wgsl/

Links for WebGPU

- https://developer.mozilla.org/en-US/docs/Web/API/WebGPU_API
- Tutorial: https://webgpufundamentals.org/webgpu/lessons/webgpu-fundamentals.html#a-run-computations-on-the-gpu
- https://github.com/robbwu/webgpu-probe https://robbwu.github.io/webgpu-probe/index2.html

Triton

- **Tile-**based intermediate language (IR) and higher level language (such as Python-like) compiler
- Mostly geared towards deep neural network operations on GPU.
- Difference vs CUDA:
  In Triton you load and operate on **tiles** (block arrays) which are automatically mapped to high performance PTX code on NVIDIA
  In CUDA we program each thread in SPMD style.

Triton Programming Model: block oriented

- In Triton you ”prescribe” what to do with regard to each “block” (literally, the threadblock in CUDA), and let compiler decide how to parallelize them across threads, and manage data movement (shared memory, registers, etc) optimally.
- The compiler lowers Triton code into Triton-IR (an extension of LLVM IR) that incorporates tiles, and perform standard optimizations such as control flow, data flow analysis
- So Triton programming model is more “prescriptive” than “declarative” CUDA.
  Pros: Higher level, easier to code (?), less constraint for compiler optimization/auto-tuning, potentially more portable
  Cons: High level, less control.

Hello World example (Vector Add Kernel)

```
@triton.jit
def add_kernel(x_ptr,  # *Pointer* to first input vector.
               y_ptr,  # *Pointer* to second input vector.
               output_ptr,  # *Pointer* to output vector.
               n_elements,  # Size of the vector.
               BLOCK_SIZE: tl.constexpr,  # Number of elements each program should process.
               # NOTE: `constexpr` so it can be used as a shape value.
               ):
    # There are multiple 'programs' processing different data. We identify which program
    # we are here:
    pid = tl.program_id(axis=0)  # We use a 1D launch grid so axis is 0.
    # This program will process inputs that are offset from the initial data.
    # For instance, if you had a vector of length 256 and block_size of 64, the programs
    # would each access the elements [0:64, 64:128, 128:192, 192:256].
    # Note that offsets is a list of pointers:
    block_start = pid * BLOCK_SIZE
    offsets = block_start + tl.arange(0, BLOCK_SIZE)
    # Create a mask to guard memory operations against out-of-bounds accesses.
    mask = offsets < n_elements
    # Load x and y from DRAM, masking out any extra elements in case the input is not a
    # multiple of the block size.
    x = tl.load(x_ptr + offsets, mask=mask)
    y = tl.load(y_ptr + offsets, mask=mask)
    output = x + y
    # Write x + y back to DRAM.
    tl.store(output_ptr + offsets, output, mask=mask)
```

“Launch the kernel”

```python
def add(x: torch.Tensor, y: torch.Tensor):
    # We need to preallocate the output.
    output = torch.empty_like(x)
    assert x.device == DEVICE and y.device == DEVICE and output.device == DEVICE
    n_elements = output.numel()
    # The SPMD launch grid denotes the number of kernel instances that run in parallel.
    # It is analogous to CUDA launch grids. It can be either Tuple[int], or Callable(metaparameters) -> Tuple[int].
    # In this case, we use a 1D grid where the size is the number of blocks:
    grid = lambda meta: (triton.cdiv(n_elements, meta['BLOCK_SIZE']), )
    # NOTE:
    #  - Each torch.tensor object is implicitly converted into a pointer to its first element.
    #  - `triton.jit`'ed functions can be indexed with a launch grid to obtain a callable GPU kernel.
    #  - Don't forget to pass meta-parameters as keywords arguments.
    add_kernel[grid](x, y, output, n_elements, BLOCK_SIZE=1024)
    # We return a handle to z but, since `torch.cuda.synchronize()` hasn't been called, the kernel is still
    # running asynchronously at this point.
    return output
```

Code quality? Example: transpose

- https://gist.github.com/robbwu/f5a6d13ba6f99372af801e60b9fa4163

--- Benchmark Summary ---

Size (MxN)	Provider	Median Time (ms)	Throughput (GB/s)
512x512	triton	0.007	292.57
512x512	pytorch	0.009	223.67
1024x1024	triton	0.016	512.00
1024x1024	pytorch	0.023	372.36
2048x2048	triton	0.056	595.78
2048x2048	pytorch	0.079	425.56
4096x4096	triton	0.275	487.26
4096x4096	pytorch	0.454	295.87
4096x1024	triton	0.067	504.12
4096x1024	pytorch	0.097	344.93
1024x4096	triton	0.054	618.26
1024x4096	pytorch	0.075	448.88
8192x8192	triton	1.198	448.11
8192x8192	pytorch	2.032	264.26

On RTX3080, maximum memory throughput 760GB/s

Example2: Softmax

- https://triton-lang.org/main/getting-started/tutorials/02-fused-softmax.html

Example 3: MatMul

- https://triton-lang.org/main/getting-started/tutorials/03-matrix-multiplication.html

✅ Triton and Torch match

matmul-performance-fp16:

	M	N	K	cuBLAS	Triton
0	256.0	256.0	256.0	4.681143	4.681143
1	384.0	384.0	384.0	12.594107	13.824000
2	512.0	512.0	512.0	18.724571	24.105195
3	640.0	640.0	640.0	23.272727	32.000000
4	768.0	768.0	768.0	27.648000	34.028308
5	896.0	896.0	896.0	39.025776	45.320259
6	1024.0	1024.0	1024.0	43.690665	53.773130
7	1152.0	1152.0	1152.0	44.566925	48.161033
8	1280.0	1280.0	1280.0	39.009524	59.362318
9	1408.0	1408.0	1408.0	47.406747	55.068446
10	1536.0	1536.0	1536.0	43.694879	52.820059
11	1664.0	1664.0	1664.0	51.422356	61.636381
12	1792.0	1792.0	1792.0	59.784168	60.427009
13	1920.0	1920.0	1920.0	52.166036	58.825532
14	2048.0	2048.0	2048.0	58.867427	59.074702
15	2176.0	2176.0	2176.0	44.915723	57.826574
16	2304.0	2304.0	2304.0	58.837122	59.275115
17	2432.0	2432.0	2432.0	54.979383	58.287269
18	2560.0	2560.0	2560.0	59.686704	60.907061
19	2688.0	2688.0	2688.0	56.786010	59.643169
20	2816.0	2816.0	2816.0	55.418308	57.236495
21	2944.0	2944.0	2944.0	58.838291	60.261223
22	3072.0	3072.0	3072.0	57.368903	57.896835
23	3200.0	3200.0	3200.0	56.189637	57.399104
24	3328.0	3328.0	3328.0	59.843141	60.496888
25	3456.0	3456.0	3456.0	58.719278	59.675474
26	3584.0	3584.0	3584.0	59.472770	58.197664
27	3712.0	3712.0	3712.0	53.969391	58.419500
28	3840.0	3840.0	3840.0	57.065016	58.053545
29	3968.0	3968.0	3968.0	56.466423	57.450061
30	4096.0	4096.0	4096.0	57.628908	56.944305

matmul-performance-fp8:

	M	N	K	Triton
0	256.0	256.0	256.0	3.640889
1	384.0	384.0	384.0	10.053818
2	512.0	512.0	512.0	20.164923
3	640.0	640.0	640.0	24.380953
4	768.0	768.0	768.0	26.810182
5	896.0	896.0	896.0	36.971791
6	1024.0	1024.0	1024.0	41.120628
7	1152.0	1152.0	1152.0	42.056111
8	1280.0	1280.0	1280.0	49.951220
9	1408.0	1408.0	1408.0	48.245805
10	1536.0	1536.0	1536.0	45.082089
11	1664.0	1664.0	1664.0	52.319256
12	1792.0	1792.0	1792.0	53.023965
13	1920.0	1920.0	1920.0	51.969927
14	2048.0	2048.0	2048.0	53.946030
15	2176.0	2176.0	2176.0	52.542161
16	2304.0	2304.0	2304.0	54.788697
17	2432.0	2432.0	2432.0	53.721113
18	2560.0	2560.0	2560.0	56.496552
19	2688.0	2688.0	2688.0	55.418440
20	2816.0	2816.0	2816.0	53.383364
21	2944.0	2944.0	2944.0	56.825579
22	3072.0	3072.0	3072.0	55.242052
23	3200.0	3200.0	3200.0	54.283290
24	3328.0	3328.0	3328.0	57.964005
25	3456.0	3456.0	3456.0	57.218997
26	3584.0	3584.0	3584.0	56.550561
27	3712.0	3712.0	3712.0	56.185232
28	3840.0	3840.0	3840.0	55.882771
29	3968.0	3968.0	3968.0	55.800705
30	4096.0	4096.0	4096.0	55.924051

Links for Triton

- Doc & tutorial: https://triton-lang.org/main/index.html
- Github repo: https://github.com/triton-lang/triton
- Puzzles: https://github.com/srush/Triton-Puzzles