Designing a GPU architecture: Waves

Git revision #5a9dfdd7[1]
Modified at 26. August 2025 21:13

Written by alex_s168[2]

Introduction

In this article, we'll be looking into the hardware of GPUs, and then designing our own. Specifically GPUs with unified shader architecture.

Comparison with CPUs

GPUs focus on operating on a lot of data at once (triangles, vertices, pixels, …), while CPUs focus on high performance on a single core, and low compute delay.

GPU Architecture

GPUs consists of multiple (these days at least 32) compute units (= CU).

Each compute unit has multiple SIMD units, also called "wave", "wavefront" or "warp". Compute units also have some fast local memory (tens of kilobytes), main memory access queues, texture units, a scalar unit, and other features. Subscribe to the Atom feed[3] to get notified of future articles.

The main memory (graphics memory) is typically outside of the GPU, and is slow, but high-bandwidth memory.

[1] https://github.com/alex-s168/website/tree/5a9dfdd720bcba1e5d1562279e09c674a30a174b
[2] https://alex.vxcc.dev
[3] atom.xml

Waves

A wave is a SIMD processing unit consisting of typically 32 “lanes” (sometimes called threads).

Each wave in a CU has separate control flow, and doesn’t have to be related.

Instructions that waves support:
- arithmetic operations
- cross-lane data movement
- CU local and global memory access: each SIMD lane can access a completely different address. similar to CPU gather / scatter.
- synchronization with other CUs in the work group (see future article)

Since only the whole wave can do control flow, and not each lane, all operations can be masked so that they only apply to specific lanes.

=> waves are really similar to SIMD on modern CPUs

In modern GPUs, instruction execution in waves is superscalar, so there are multiple different execution units for executing different kinds of instructions, and multiple instructions can be executed at once, if there are free execution units, and they don’t depend on each other.

We’ll be exploring that in a future article.

Local memory

The local memory inside GPUs is banked, typically into 32 banks. The memory word size is typically 32 bits.

The addresses are interlaved, so for two banks:
- addr 0 => bank 0
- addr 1 => bank 1
- addr 2 => bank 0
- addr 3 => bank 1
- …

Each bank has an dedicated access port, so for 32 banks, you get 32 access ports.

The lanes of the waves inside a CU get routed to the local memory banks magically.

Why are the banks interlaved?

When the whole wave wants to read a contiguous array of `f32`, so when each wave performs `some_f32_array[lane_id()]`, all 32 banks can be used at the same time.

Why multiple waves share the same local memory

A wave doesn't do memory accesses every instruction, but also does computations. This means that there are cycles where the memory isn't doing anything.

By making multiple waves share the same local memory and access ports, you save resources.

Global memory

Since global memory reads/writes are really slow, they happen asynchronosly.

This means that a wave requests an access, then can continue executing, and then eventually waits for that access to finish.

Because of this, modern compilers automagically start the access before the data is needed, and then wait for the data later on.

Scalar unit

Most newer GPUs also have a scalar unit for saving energy when performing simple operations.

When the controller sees a scalar instruction in the code running on a wave, it automatically makes the code run on the scalar unit.

The scalar unit can be used for:
- address calculation
- partial reductions
- execution of expensive operations not implemented on SIMD because of costs

GPU Programming Terminology

- "work item": typically maps to a SIMD lane
- "kernel": the code for a work item
- "work group": consists of multiple work items. typically maps to an CU. the `__local` memory in OpenCL applies to this.
- "compute task": a set of work groups

OpenCL and other APIs let you specify both the number of work groups and work items.

Since a program might specify a higher number of work items per work group than we have available, the compiler needs to be able to put multiple work items onto one SIMD lane.

Our own architecture

We'll go with these specs for now:

- N compute units
- 2 waves per CU
- 32 lanes per wave.
- 1KiB local memory per lane => 64 KiB
- 48 vector registers of 16x32b per wave
- one scalar unit per CU
- 128 global memory ports
- 16 async task completion "signal" slots per wave
- no fancy out of order or superscalar execution
- support standard 32 bit floating point, without exceptions.

Note that we won't specify the exact instruction encoding.

Predefined Constants

We will pre-define 16 constants (as virtual vector registers):

- `zero`
- `one`
- `sid`: 0,1,2,3,4,5,6
- `wave`: the ID of the wave in the compute task, broadcasted to all elements.
- `u8_max`: 255,255,…
- `n2nd`: 1,2,1,2,…
- `n3rd`: 1,2,4,1,…
- `n4th`: 1,2,4,8,1,…
- `lo16`: 1,1,1,… (x16) 0,0,0,… (x16)
- `ch2`: 1,1,0,0,1,1,…
- `ch4`: 1,1,1,1,0,0,0,0,1,…
- `alo8`: 1 (x8) 0 (x8) 1 (x8) 0 (x8)
- a few reserved ones

Operands

We define the following instruction operands:

- `Vreg`: vector register
- `M`: (read only) vector gp reg as mask (1b). only first 32 registers can be used as mask. the operand consists of two masks and-ed together, each of which can conditionally be inverted first. this means that this operand takes up 12 bits
- `Vany`: `Vreg` or `M`
- `Simm`: immediate scalar value
- `Sreg`: the first element of a vector register, as scalar
- `Sany`: a `Simm` or an `Sreg`
- `dist`: `Vany`, or a `Sany` broadcasted to each element
- `sig`: one of the 16 completion signal slots

Instructions

We will add more instructions in future articles.

Data Movement

- `fn mov(out out: Vreg, in wrmask: M, in val: dist)`
- `fn select(out out: Vreg, in select: M, in false: dist, in true: dist)`
- `fn first_where_true(out out: Sreg, in where: M, in values: dist)`: if none of the elements are true, it doesn't overwrite the previous value in out.
- cross-lane operations: not important for this article

Mathematics

- simple (unmasked) `u32`, `i32`, and `f32` elementwise arithmetic and logic operations: `fn add<u32>(out out: Vreg, in left: Vany, in right: dist)`
- scalar arithmetic and logic operations: `fn add<u32>(out out: Sreg, in left: Sany, in right: Sany)`
- partial reduction operations: "chunks" the input with a size of 8, reduces each chunk, and stores it in the first element of the chunk. this means that every 8th element will contain a partial result.
- and operations to finish that reduction into the first element of the vector

Local memory

- load 32 bit value at each elem where mask is true:
  `fn local_load32(out out: Vreg, in mask: M, in addr: Vreg)`
- store 32 bit value at each elem where mask is true:
  `fn local_store32(in addr: Vreg, in mask: M, in val: Vany)`

Global (async) memory

- start an async global load, and make the given signal correspond to the completion of the access: load 32 bit value at each elem where mask is true: `fn global_load32(out sig: sig, out out: Vreg, in mask: M, in addr: Vreg)`
- see above and `local_store32`
  `fn global_store32(out sig: sig, in addr: Vreg, in mask: M, in val: Vany)`
- `fn sig_done1(out r: Sreg, in sig: sig)`
- `fn sig_done2(out r: Sreg, in sig1: sig, in sig2: sig)`
- `fn sig_wait(out r: Sreg, in sig: sig)`
- `fn sig_waitall2(out r: Sreg, in sig1: sig, in sig2: sig)`
- `fn sig_waitall3(out r: Sreg, in sig1: sig, in sig2: sig, in sig3: sig)`
- `fn sig_waitall4(out r: Sreg, in sig1: sig, in sig2: sig, in sig3: sig, in sig4: sig)`

As a future extension, we could add a instruction that waits for any of the given signals to complete, and then jump to a specific location, depending on which of those completed.

Control flow (whole wave)

- branch if scalar is zero: `fn brz(in dest: Simm, in val: Sany)`
- branch if scalar is not zero: `fn brnz(in dest: Simm, in val: Sany)`
- branch on the whole wave if each element has a true value for the mask:
  `fn br_all(in dest: Simm, in cond: M)`
- branch on the whole wave if any element has a true value for the mask:
  `fn br_any(in dest: Simm, in cond: M)`

Hand-compiling code

Now that we decided on a simple compute-only GPU architecture, we can try hand-compiling an OpenCL program.

I asked an LLM to produce a N*N matmul example (comments written manually):

```
// convenient number for our specifc hardware
#define TILE_SIZE 8

// this kernel will be launched with dimensions:
//   global[2] = { 128,128 } = { N, N };
//   local[2]  = { 8,8 } = { TILE_SIZE, TILE_SIZE };
__kernel void matmul_tiled(
  __global float* A,
  __global float* B,
  __global float* C,
  const int N)
{
  int row = get_global_id(1); // y
  int col = get_global_id(0); // x
  int local_row = get_local_id(1); // y
  int local_col = get_local_id(0); // x

  __local float Asub[TILE_SIZE][TILE_SIZE];
  __local float Bsub[TILE_SIZE][TILE_SIZE];

  float sum = 0.0f;

  for (int t = 0; t < N / TILE_SIZE; ++t) {
    // load tiles into local
    int tiledRow = row;
    int tiledCol = t * TILE_SIZE + local_col;
    float av;
    if (tiledRow < N && tiledCol < N)
      av = A[tiledRow * N + tiledCol];
    else
      av = 0.0f;
    Asub[local_row][local_col] = av;

    tiledRow = t * TILE_SIZE + local_row;
    tiledCol = col;
    float bv;
    if (tiledRow < N && tiledCol < N)
        bv; = B[tiledRow * N + tiledCol];
    else
        bv = 0.0f;
    Bsub[local_row][local_col]= bv;

    // sync local access across local grp
    barrier(CLK_LOCAL_MEM_FENCE);

    for (int k = 0; k < TILE_SIZE; ++k)
        sum += Asub[local_row][k] * Bsub[k][local_col];

    // sync local access across local grp
    barrier(CLK_LOCAL_MEM_FENCE);
  }

  if (row < N && col < N)
    C[row * N + col] = sum;
}
```

First, we have to decide on how we want to map the kernel to the hardware.

Since the local dimension of the kernel is 8*8, which is 64, we can map each local group to one CU, by mapping 32 kernels to one wave, and using both waves available on one CU for the local group.

Our global dimension is 128*128, which means that we would need 256 compute units. But since we probably don't have 256 compute units, GPUs, including ours, will have a on-hardware task scheduler, for scheduing tasks onto compute units.

Outro

Modern GPUs are really complex, but designing a simple GPU is not that hard either.

Subscribe to the Atom feed[4] to get notified of future articles.

[4] atom.xml