Distributed computing and GPUs

Types of parallel computations

Plan

0. Parallelism that does not require programmer intervention
1. SIMD
2. Thread-level concurrency
3. Distributed computing
4. Hardware acceleration

3. Distributed computing

Distributed computing

- In distributed computing, processes do not share memory

- They must communicate by explicitly sending data to each other

  (`send()`, `recv()`, etc.)

  typically over the network

Distributed computing

- Con: Communication is much slower than multithreading

- Pros:
  - Easier to implement and reason about
  - Scales to higher levels of parallelism
    - As of today, off-the-shelf computers can have up to

      2 processors × 192 cores × 2 SMT threads = 768 concurrent software threads

      (AMD Epyc 9965, Zen 5c cores)
    - With distributed computing, networked computers can work together in parallel

- Libraries:
  - Message Passing Interface (MPI)
  - …

4. Hardware acceleration

Graphics processing units (GPUs)

- GPUs were designed to perform the same simple, repetitive operations
  - on many pixels (“fragment shaders”), or
  - on many 3D coordinates (“vertex shaders”)

Simplified fragment shader example

Given:

- the coordinates `(i, j)` of a pixel on screen
- the corresponding coordinates in 3D space `(x, y, z)`
- the normal vector to the surface at that point `(v0, v1, v2)`
- the texture color for that point `(r, g, b)`
- the coordinates `(lx, ly, lz)` and color `(lr, lg, lb)` of a light source

compute

- the apparent color of the pixel.

Simplified vertex shader example

Given:

- the coordinates of an object in 3D space `(x, y, z)`
- the coordinates of the viewpoint camera in 3D space `(vx, vy, vz)`
- the viewpoint camera orientation `(q0, q1, q2, q2)`
- the viewpoint camera's focal length `f`

compute:

- the coordinates `(i, j)` of this object on screen.

- GPUs were designed to perform the same simple, repetitive operations
  - on many pixels (“fragment shaders”), or
  - on many 3D coordinates (“vertex shaders”)
- they generally adopt a SIMT (“single instruction, multiple threads”) model
  - hundreds of threads working on different sets of data
  - but some running the exact same instructions

    (in CUDA, the 32 threads in a “warp” execute in lockstep)
- good fit for long loops performing repetitive operations
- bad fit for if/then/else

Single instruction, multiple threads (SIMT)

```
#                                                    Warp
#                                Thread 0          Thread 1          Thread 2          Thread 3
#                            ----------------- ----------------- ----------------- -----------------
#                             condition = True  condition = True  condition = False condition = False
def function(condition, data):
    if condition:
        r = a(data)
    else:
        r = b(data)
    return r
```

Single instruction, multiple threads (SIMT)

```
#                                                                  Warp
#                                      Thread 0          Thread 1          Thread 2          Thread 3
#                                  ----------------- ----------------- ----------------- -----------------
#                                   condition = True  condition = True  condition = False condition = False
def function(condition, data):
    if condition:                  if True:          if True:          if False:         if False:
        # <-----
        r = a(data)
    else:
        r = b(data)
    return r
```

Single instruction, multiple threads (SIMT)

```
#                                                                  Warp
#                                    Thread 0          Thread 1          Thread 2          Thread 3
#                                ----------------- ----------------- ----------------- -----------------
#                                 condition = True  condition = True  condition = False condition = False
def function(condition, data):
    if condition:                 if True:          if True:          if False:         if False:
        r = a(data)                   r = a(data)       r = a(data)       pass              pass
        # <-----
    else:
        r = b(data)
    return r
```

Single instruction, multiple threads (SIMT)

```
#                                                                  Warp
#                                    Thread 0          Thread 1          Thread 2          Thread 3
#                                ----------------- ----------------- ----------------- -----------------
#                                 condition = True  condition = True  condition = False condition = False
def function(condition, data):
    if condition:                 if True:          if True:          if False:         if False:
        r = a(data)                   r = a(data)       r = a(data)       pass              pass
    else:                         else:             else:             else:             else:
        # <-----
        r = b(data)
    return r
```

Single instruction, multiple threads (SIMT)

```
#                                                    Warp
#                                  Thread 0          Thread 1          Thread 2          Thread 3
#                                ------------------ ------------------ ------------------ ------------------
#                                 condition = True   condition = True   condition = False  condition = False
def function(condition, data):
    if condition:                 if True:           if True:           if False:         if False:
        r = a(data)                   r = a(data)        r = a(data)        pass              pass
    else:                         else:              else:              else:             else:
        r = b(data)                   pass               pass               r = b(data)       r = b(data)
        # <-----
    return r
```

Single instruction, multiple threads (SIMT)

```
#                                                                  Warp
#                                  Thread 0          Thread 1          Thread 2          Thread 3
#                                ------------------ ------------------ ------------------ ------------------
#                                 condition = True   condition = True   condition = False   condition = False
def function(condition, data):
    if condition:                 if True:           if True:           if False:          if False:
        r = a(data)                   r = a(data)        r = a(data)        pass               pass
    else:                         else:              else:              else:              else:
        r = b(data)                   pass               pass               r = b(data)        r = b(data)
    return r                      return r           return r           return r           return r
        # <-----
```

Single instruction, multiple threads (SIMT)

```
#                                                    Warp
#                                  Thread 0          Thread 1          Thread 2          Thread 3
#                                ------------------ ----------------- ----------------- -----------------
#                                 condition = True   condition = True  condition = False  condition = False
def function(condition, data):
    if condition:                 if True:           if True:          if False:         if False:
        r = a(data)                   r = a(data)        r = a(data)       pass              pass
    else:                         else:              else:             else:             else:
        r = b(data)                   pass               pass              r = b(data)       r = b(data)
    return r                      return r           return r          return r          return r
```

How do we use GPUs?

- GPUs are programmed in special-purpose languages

- Typically, all GPU code is compiled
  - during application startup (“shader compliation”),
  - by the device driver
  - for the specific GPU device installed (amount and subdivision of threads, memory, etc.)

- Two dominant players in the GPU market: nVidia and AMD

- Three major general-purpose GPU programming languages:
  - CUDA (nVidia, proprietary)
  - ROCm (AMD, open-source)
  - OpenCL (cross-platform, open-source)

Examples (GLSL)

```
float box(in vec2 st, in vec2 size){
    size = vec2(0.5) - size*0.5;
    vec2 uv = smoothstep(size,
                         size+vec2(0.001),
                         st);
    uv *= smoothstep(size,
                     size+vec2(0.001),
                     vec2(1.0)-st);
    return uv.x*uv.y;
}
```

Examples (GLSL)

```
vec3 rgb2hsb( in vec3 c ){
    vec4 K = vec4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);
    vec4 p = mix(vec4(c.bg, K.wz),
                 vec4(c.gb, K.xy),
                 step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r),
                 vec4(c.r, p.yzx),
                 step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 1.0e-10;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)),
                d / (q.x + e),
                q.x);
}
```

Examples (CUDA)

```
inline __device__ float3 roundAndExpand(float3 v, ushort *w) {
  v.x = rintf(__saturatef(v.x) * 31.0f);
  v.y = rintf(__saturatef(v.y) * 63.0f);
  v.z = rintf(__saturatef(v.z) * 31.0f);

  *w = ((ushort)v.x << 11) | ((ushort)v.y << 5) | (ushort)v.z;
  v.x *= 0.03227752766457f;  // approximate integer bit expansion.
  v.y *= 0.01583151765563f;
  v.z *= 0.03227752766457f;
  return v;
}
```

Single-threaded matrix multiplication

```
2048 x 2048 matrix multiplication
precision: fp32 (“float”) - total 16 MB per matrix
CPU: Ryzen 7900 x3d (released Feb 2023)
```

version	description	seconds	ratio	improvement
1	straightforward implementation	29.3	**1x**	
2	transpose B matrix	4.7	6x	6x
3	block 16x16	1.4	21x	3x
4	`-march=native` (SIMD)	0.21	140x	7x
5	AVX512 intrinsics (SIMD)	0.16	183x	1.4x
6	OpenBLAS (SIMD)	0.15	195x	1.07x

Parallel matrix multiplication

```
65536 x 65536 matrix multiplication
precision: fp32 (“float”) - total 16 GB per matrix
CPU: Ryzen 7900 x3d (released Feb 2023)
GPU: nVidia H100 80GB (released Oct 2022)
GPU: nVidia H200 141GB (released Apr 2024)
```

version	description	seconds	ratio	improvement
6	OpenBLAS	2490.618	**1x**	
7	OpenBLAS, 24 threads	344.961	7x	7x
8	cuBLAS, H100	28.657	87x	12x
8	cuBLAS, H200			