week 13: GPU's

wbg231

January 2023

1 GPUs

gradient descent

- remember serial gradient descent we more or less do with a basic for loop is $O(i * n)$ where i is the number of points times the number of iterations
- in spark we can run the for loop in parallel, but need to run the outer loop serially so it is $O(i * \frac{n}{k})$ where k is the number of worker nodes
- we could use different algorithms to optimize this more
- or we could use a different type of computer ie a GPU

why GPU's

- speed ups come from constraint so what are we constraining in a GPU?

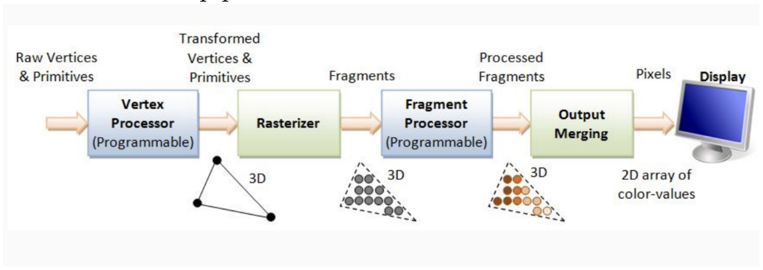
GPU and the rendering pipeline

3d rendering

- to render 3d graphics we can use
- inputs: 3d mesh, textures, light sources, camera positions
- outputs: a 2d array of pixels (that is a rendered scene)
- video games must do this with a constant time constraint
- so computationally there a challenges bases on the complexity of the scene and the output resolution

the pipeline

- here is what the pipeline looks like


Raw Vertices & Primitives
Transformed Vertices & Primitives
Fragments
Processed Fragments
Pixels
Display
Vertex Processor (Programmable)
Rasterizer
Fragment Processor (Programmable)
Output Merging
3D
3D
3D
2D array of color-values

vector processing

- the idea is to perform the coordinate transformations for each model
- also need the camera transformations
- camera lense / field of view
- these are mostly linear transformations (rotations, translations and scales)
- all vertices are mapped to camera coordinates (x,y,z)

rasterizer

- scans the scene for data to render at each (x,y) pixel
- object vertices do not necessarily line up to pixel coordinates so need 3d meshes to be calculated
- outputs pixels containing data to render at each point
- this step is not programable

fragment processing

- texture mapping and lighting
- outputs color values for each fragment
- may include objects that are clouded and thus discarded later

parallelism in graphics

- the vertex processor and fragment processor parts that are programable are called shaders
- linear transformations are by nature independent to each vertex
- texture and lighting is independent across fragments
- specialized hardware can parallelize to meet real time constraints
- to be cost effective each vertex or pixel processor must be simple

shaders

- shaders are short programs that are applied to independently to each vertex or fragment
- they are kinda like mappers
- shaper code tends to have a simple control flow

specialized shader units

- older gpus had separate processors for vertex shading and pixel shading
- this works well if the loads between these two task were ballandce but they are not always
- unbalanced load is idle processors ie key skew

general purpose GPUs

shaders to kernels

- general purpose GPU's remove the distinction between vector and pixel cores
- shaders are replaced by kernel abstractions
- this allows cores to operate at any core
- the CUDA api controls which core get which task

gpu design

- here is what a traditional computer design looks like includegraphics*[width=10cm]images/Screenshot 2023-05-11 at 11.34.03 PM.png
- the ALU (arithmetic logic units) are for basic math operations (ie computation)
- the control i for program flow
- the cache is on PCU memory
- the DRAM is main system memory
- a GPU is organized like this includegraphics*[width=10cm]images/Screenshot 2023-05-11 at 11.34.17 PM.png
- they have the the same components but in different proportions
- overall computing looks something like this includegraphics*[width=10cm]images/Screenshot 2023-05-11 at 11.37.22 PM.png

threads blocks and grids

- cuda arranges kernel execution into threads blocks and grids

thread

- a thread does on execution all threads execute the same kernel program but on different pieces of data
- has local private memory

blocks

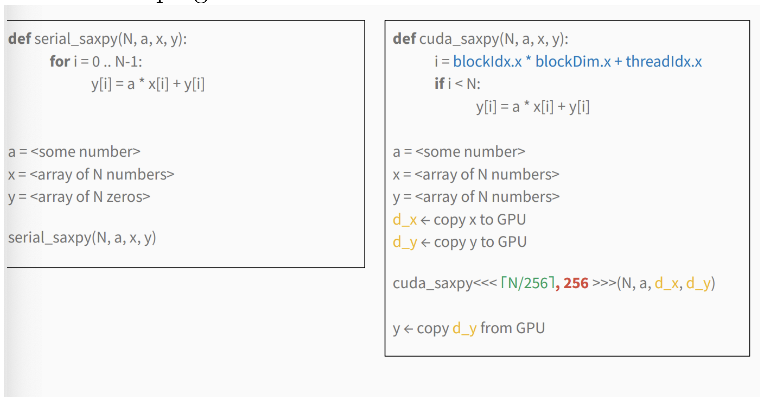
- a group of possibly related threads
- has shared memory accessible by all threads

grid

- a collection of blocks
- all grid cells have access to shared global memory

SAXPY

- here are two programs to calculate an affine translations



```
def serial_saxpy(N, a, x, y):
    for i = 0 .. N-1:
        y[i] = a * x[i] + y[i]


a = <some number>
x = <array of N numbers>
y = <array of N zeros>

serial_saxpy(N, a, x, y)
```

```
def cuda_saxpy(N, a, x, y):
    i = blockIdx.x * blockDim.x + threadIdx.x
    if i < N:
        y[i] = a * x[i] + y[i]

a = <some number>
x = <array of N numbers>
y = <array of N numbers>
d_x ← copy x to GPU
d_y ← copy y to GPU

cuda_saxpy<<< ⌈N/256⌉, 256 >>>(N, a, d_x, d_y)

y ← copy d_y from GPU
```

- the serial takes O(n), but there are no computational dependencies so can be paralleled
- the cuda one shows how cuda runs, more or less run kernel for all block if hte thread id is n output it
- data access is managed by the thread
- have access to blockid.x (current block id), blockdim.x (size of current block) threadid.x (current thread within the block)
- there are 256 threads per block so we want to try to break our computation up into 256 pieces per block
- if $i < n$ avoids out of bound errors from integer math in block derision
- cuda lets you access data out side of your thread and block index

thread execute and warps

- threads within a single block execute in parallel via single instruction multiple thread design
- threads run in groups of 32 called warps
  - thread start at teh same instruction but can follow different execution paths
  - warp finishes when all threads finish
  - threads should follow a common path (that is there should not be a tone of conditionals)
  - threads can be explicitly synchronized

- blocks need not execute all simultaneously (can put them on different codes) but this is why you can not share memory between blocks

threads blocks grids and hardware

- like mappers kernels must be independent from one another to and able to execute in any order (at least at the block level)
- unlike mappers kernels are not pure functions, they do not have return values at all they write output to a pre allocated memory buffer
- to exploit memory within blocks must be careful about organizing data
- different hardware will have different number of cores

complex programs

- cuda kernels are usually simple operations like matrix multiplication
- usually we want to combine these to make a complex program (so multiple independent kernel functions that together make a complex program)
- not dissimilar from in spark doing transformations and actions in multiple parts

pitfalls of gpu usage

- Efficient usage relies on keeping all cores busy at all times
- usually an idle is waiting on data from the cpu
- memory communication between gpu and cpu is usually the biggest bottle neck
- try to keep as much data on the gpu as possible with out transfers
- programs with complex control flows (ie a lot of conditionals) are tough for gpus
- writing custom code on GPU's is really tough so use existing frame works

why gpus for deep learning

- neural nets consist of alternating between simple operations
- these operations have a parallel structure
- further there is almost no control flow or loops

software

what tools to use

- for deep learning use tensor flow or pytorch
- for traditional ml use RAPIDS
- for you own code use numba or raw cuda

RAPIDS

- uses arrows for in memory column oriented storage on the GPU
- this is good for implementing data frames on a gpu
- minimizes overall data transfers
- integrates well with dask

numba

- classically cuda but in python
- is good if you dont want to write code in c