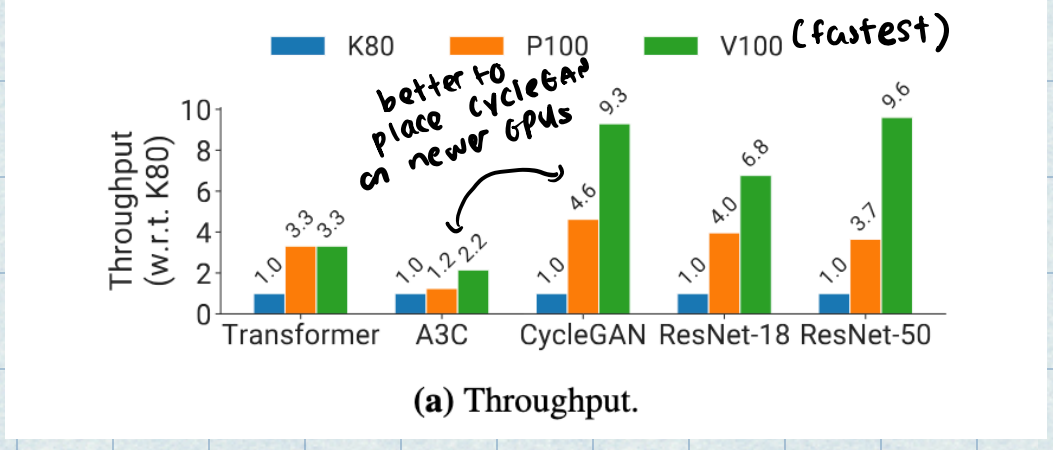
Agenda

- Heterogeneity-Aware cluster scheduling (continuation from last time)
- GPU Architecture / History of Programming GPUs
- Cuda Programming model overview

References: Lecture content taken largely from CS149 at stanford (parallel programming):
https://gfxcourses.stanford.edu/cs149/fall23/lecture/gpucuda/

Heterogeneity Aware cluster scheduling

→ more info in OSDI 20 paper on Gavel: https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak

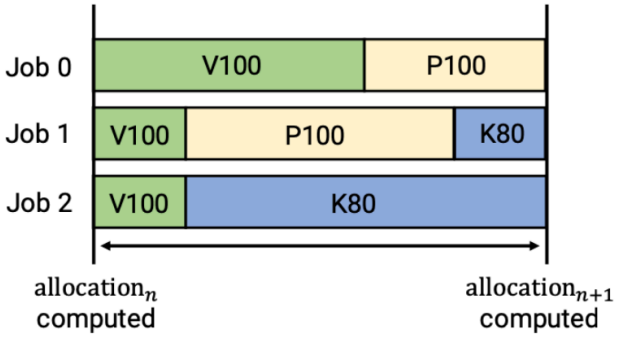


(a) Throughput.

key problem:
- not every job benefits at same proportional rate from better GPUs

- gavel paper lets us reason about resource allocation via optimization problems:

$$X^{\text{example}} = \begin{pmatrix} V100 & P100 & K80 \\ 0.6 & 0.4 & 0.0 \\ 0.2 & 0.6 & 0.2 \\ 0.2 & 0.0 & 0.8 \end{pmatrix} \begin{matrix} \text{job 0} \\ \text{job 1} \\ \text{job 2} \end{matrix}$$



- we have 3 GPU types, 3 Jobs
- allocation matrix: X_{ij} is fraction of time Job i spends on GPU j. → e.g. job 0
- assume for each job and GPU type, we can estimate (from offline mmt):

T_{ij} → throughput of job i on GPU j. (iterations/s)

- given matrix X (allocations) and matrix T (measured tputs) we can calculate:

effective tput: sum of tput achieved on each resource type, weighted by time spent on that type:

$$\text{tput}(i, X) = \sum_{j \in \text{accelerators}} T_{ij} \cdot X_{ij}$$

(job i)

(throughput normalized by allocation)

- now, we have a goal: find some allocation X which satisfies some global perf. objective

$$\text{maximize}(X) \sum_{i \in \text{jobs}} \text{tput}(i, X)$$

(can be represented) this is a linear program that can be solved by existing libraries

s.t.

$0 \leq X_{ij} \leq 1 \quad \forall (i,j)$ → each individual alloc is valid fraction

$\sum_j X_{ij} \leq 1 \quad \forall i$ → total allocation for job

$\sum_i X_{ij} \cdot scale\_factor_i \leq$ #workers $\forall j$ → can't exceed 100%
total alloc of resources cannot be more than # of resources in cluster

- we can now express our previous scheduling policies as objectives:

$$\text{Minimize}_X \min_i \frac{num\_steps_i}{tput(i, X)}$$ → minimize runtime of shortest job (SJF)

$$\text{Minimize}_X \max_i \frac{num\_steps_i}{tput(i, X)}$$ → minimize runtime of longest job (minimizing makespan).

- what about an objective related to fairness?

$$\text{Maximize}_X \min_i \frac{throughput(i, X)}{throughput(i, X^{equal})} \cdot scale\ factor_i$$

↳ isolated allocation where each job has equal time on all resources

- maximizing the *minimum* job throughput normalized by equal share

↳ suppose Job 1 has ratio of 2 → meaning its throughput is 2x what it would have gotten in fair equal share

→ and Job 2 has ratio of .5 → its throughput is HALF of what it would have achieved in fair equal share

- a *different allocation* w/ [1.1, 1.1] (J1, J2) would be favored over [2, .5]

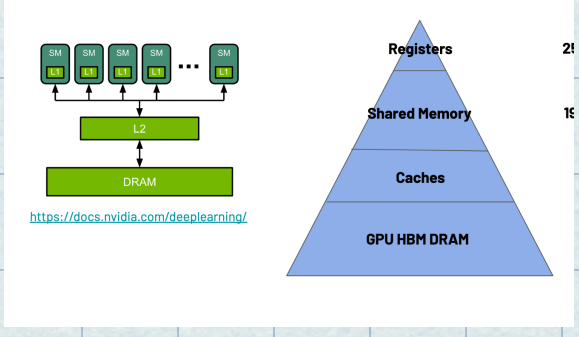
*I encourage you to look at paper for more info on:

- FIFO, cost-aware policies, weighted policies, hierarchical policies, placement sensitive, space-sharing (other policies)

*PART 2 OF LECTURE: History of GPU / Programming GPU

-recall gpu architecture:

- multiple S.M.s (stream processors)
- within an SM, threads divided into warps
  - ↳ warps can use SIMT (perform same instruction for multiple threads) ↳ single instruction, multiple thread
  - ↳ multiple warps executed concurrently
- memory hierarchy



- GPUs were originally designed to do 3d rendering
  - ↳ Input: description of scene:
    - 3d surface geometry, (triangle mesh), surface materials, lights, camera
  - ↳ output: rendered image

*rendering task: compute how each Triangle △ in 3d mesh contributes to appearance of pixel in image.

- given triangle, determine where it lies on

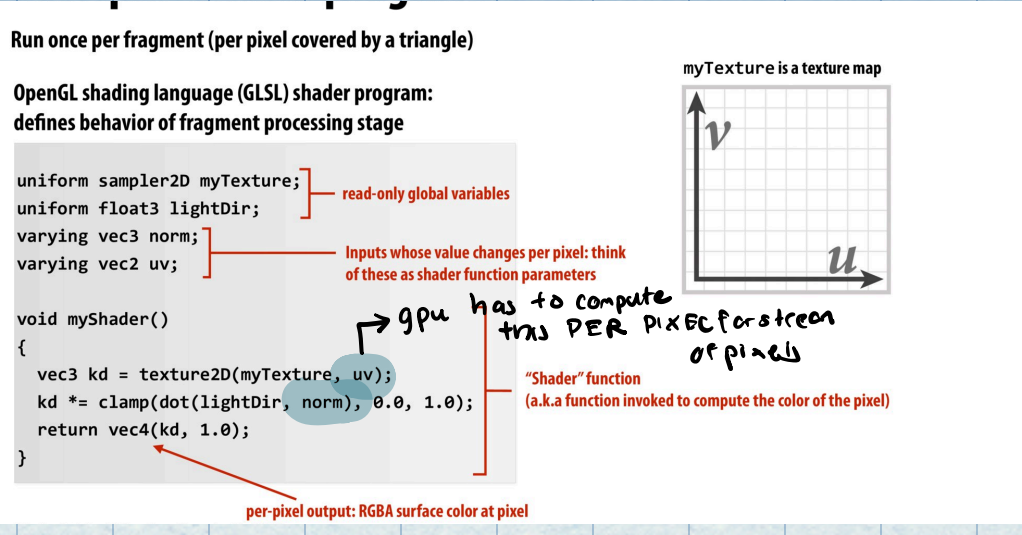
screen, given position of virtual camera

- for all output image pixels covered by triangle, compute color of surface at that pixel

*example GPU programming snippet

- OpenGL shading language: defines behavior of fragment processing stage

- define a function that can op. on a stream of inputs



Credits, Kayvan Fatahalian CS 149 Stanford:https://gfxcourses.stanford.edu/cs149/fall23/lecture/gpucuda/slide_13

* details not important: but myshader() is calculated PER PIXEL - so lots of opportunity for parallelization

* can easily take advantage of SMs, SIMT

- gpu needs to perform SAME computation in parallel on large data collections in graphics(streams of vertices fragments, pixels)

* early gpu-based scientific computation

- input: array of size 512x512
- think of this as output "image" where we have to render 512^2 pixels
  - ↳ set output image to be 512x512
  - ↳ render 2 triangles to cover screen
    - one shader function ran per pixel
    - is now "shader function" per output elt.

- this "hack" was successfully used for
  - sparse matrix solvers
  - ray tracing
  - other simulations

* "compile" data parallel scientific compute to execute on GPU

Credits, Kayvan Fatahalian CS 149 Stanford:https://gfxcourses.stanford.edu/cs149/fall23/lecture/gpucuda/slide_13

```
kernel void scale(float amount, float a<>, out float b<>)
{
   b = amount * a;
}

float scale_amount;
float input_stream<1000>;   // stream declaration
float output_stream<1000>;  // stream declaration

// omitting stream element initialization...

// map kernel function onto streams
scale(scale_amount, input_stream, output_stream);
```

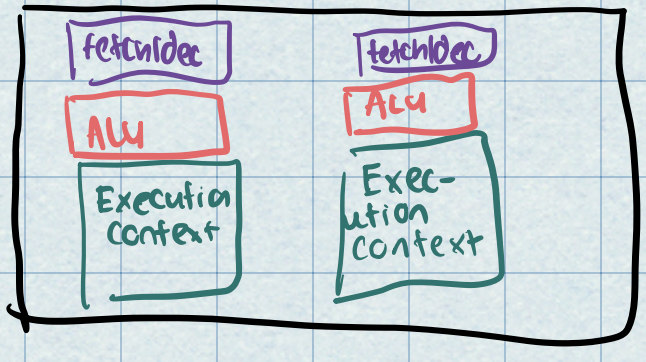
- Brook stream programming language (2004) - Buck et al
- abstract gpu as data parallel processor

- translates function to run into graphics commands (e.g., drawTriangles) - where impls already exist

* Programming GPUs was hard!

- How do we run code on a CPU?

1. OS loads program text into memory
2. OS selects CPU execution context
3. OS interrupts processor, context switches to new program by preparing execution context (setting content of registers, P.C.,)
4. Processor (ALU) can execute instructions from two context:



How do we run code on GPU (2007?

(to draw a picture):

1. App (w/ graphics drivers) provides GPU shader binaries
2. App sets graphic pipeline params (output size)
3. App gives GPU input set of vertices
4. App sends CPU "draw program"

↳ only way to execute code on GPU was via these "draw" commands

w/ NVIDIA Tesla architecture, Nvidia offered new interface

1. App can allocate buffers in GPU memory, copy data to/from buffers
2. App (via driver) provides GPU single kernel program binary
3. App tells GPU to run kernel in SPMD fashion

"run N instances of my kernel on multiple data"

launch(myKernel, N)

CUDA

- 2007
- "C"-like language to run code on GPUs
- pretty low-level: goal is to maintain low abstraction distance from GPU HW

*note: "Cuda thread" has similar abstraction to CPU-threads, but impl is very diff.

1) Cuda programs consist of HIERARCHY of concurrent threads, upto 3D

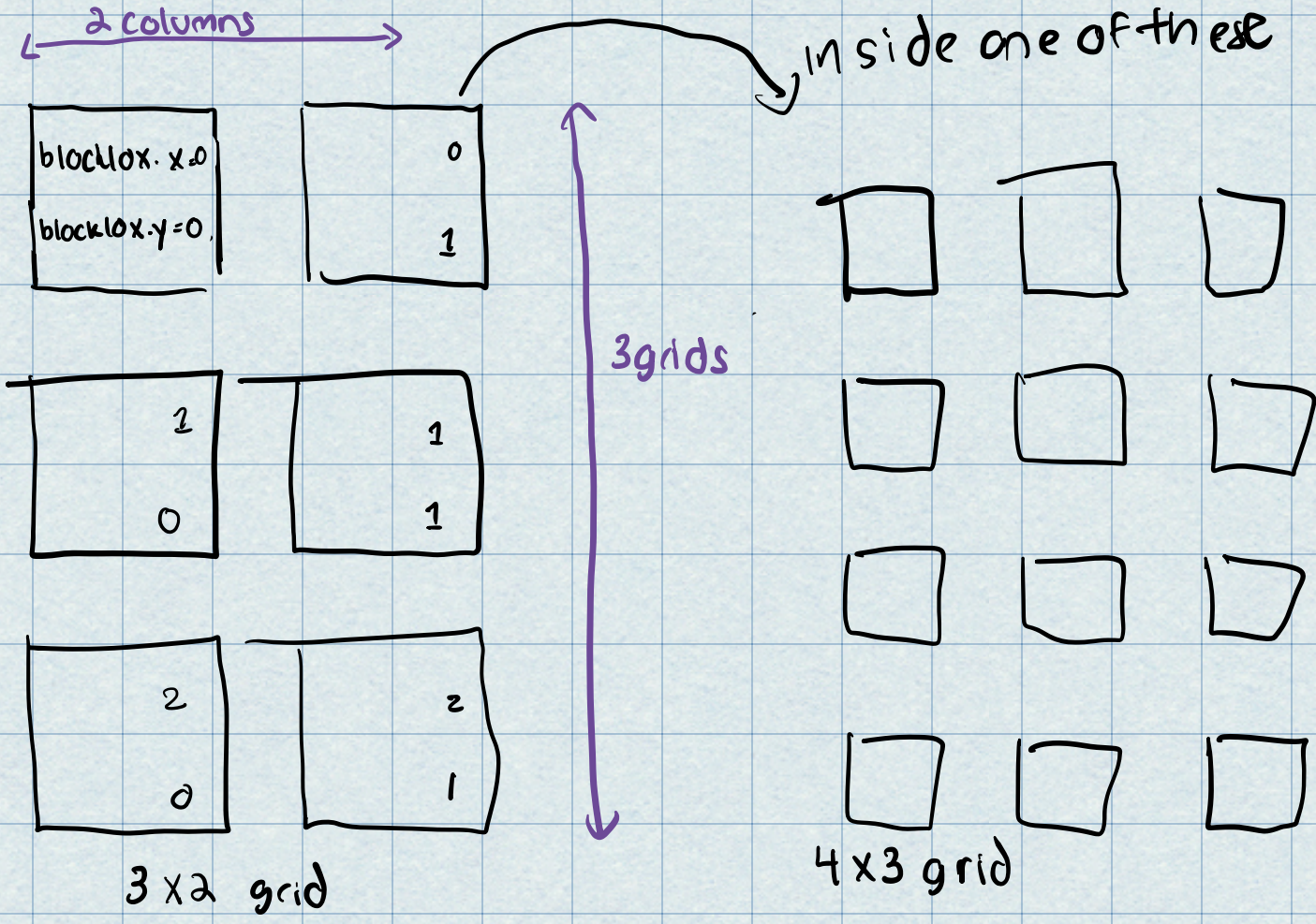
↳ useful for programs that are inherently N-D

-in following example:

- threadblocks arranged into grid (3x2)

-within thread block, threads organized into a grid (4x3)

```
1  const int Nx = 12;
2  const int Ny = 6;
3
4  // dim variable can be up to 3d, has .x, .y, and .z attributes
5
6  // shape inside of a thread block
7  dim3 threadsPerBlock(4, 3);
8  // shape of entire grid (here -- 3 by 2)
9  dim3 numBlocks(Nx/threadsPerBlock.x, Ny/threadsPerBlock.y);
10
11 // A,B,C are allocated Nx x Ny float arrays
12
13 // this call launches 72 CUDA threads:
14 // 6 thread blocks of 12 threads each
15 matrixAdd<<<numBlocks, threadsPerBlock>>>(A,B,C);
16
```



*How does this translate to code?

```
17 // kernel definition of matrix add
18 __global__ void matrixAdd(float A[Ny][Nx],
19                            float B[Ny][Nx],
20                            float C[Ny][Nx])
21 {
22
23     // not showing code here
24     // each thread has access to following variables:
25     // blockIdx -- its block's position in grid
26     // blockDim -- overall dim of blocks
27     // threadIdx -- its position in block
28     // threadDim -- overall dim of threads per block
29     int i = /*TODO*/;
30     int j = /*TODO*/;
31
32     C[j][i] = /*TODO*/
33 }
```

input is still reference to ENTIRE matrix (12x6 in above example)

6 12

- in this code - A, B, C are individual cell values of output/input matrices

- each kernel has access to following state:

1. blockIdx [2D] - position of thread block in overall grid
2. threadIdx [2D] - position of thread in thread block
3. blockDim
   ↳ dimension of blocks in grid

*TODO code:

- given values above for position of thread block in grid, position of thread in thread block
- each matrix can access some PART OF input matrix

*in our example:

- Nx = 12, Ny = 6

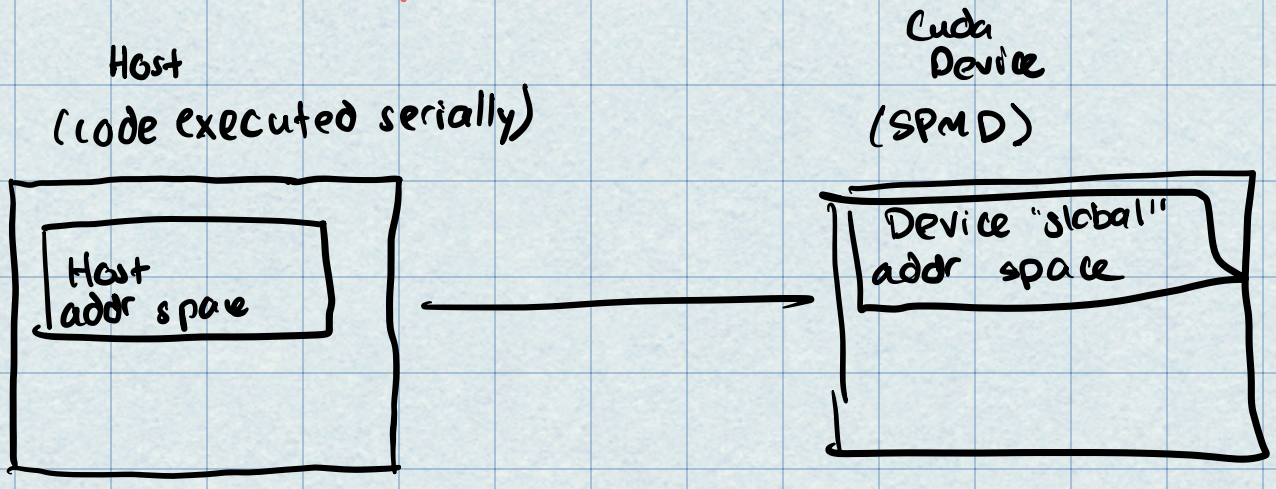
- we have 6 thread blocks (3×2) each w/ 12 threads (4×3)
- how much of output is each thread computing?
- what is missing code? (in class activity)

- cuda has clear separation of host and device code
  - ↳ 1st code snippet executes on host and launches kernel
    2nd snippet is fun SPMD on device

- number of SPMD threads in cuda program is explicit
  - *in previous programming model, #Kernel invocation was dependent on data (in graphics shader programming)
    - ↳ "map (kernel, collection"
  - now, in cuda code snippets, programmer is declaring both the grid size for thread blocks overall, and size of each thread block
    - *this means code has to specifically reference parts of its input w/ indices

of thread blocks and grids.

* CPU / GPU memory model



- how do we move data between address spaces?

use cudaMemcpy!

```
40 float *A = new float[N]; → allocate in host mem
41
42 // populate host address space pointer A
43 for (int i=0; i<N; i++) {
44     A[i] = (float)i;
45 }
46
47 int bytes = sizeof(float) * N;
48 float* deviceA;
49 cudaMalloc(&deviceA, A, bytes, cudaMemcpyHostToDevice);
```

↳ note that we can't directly access deviceA from host mem, as deviceA doesn't pt to memory in host mem

- reminder of memory model:

1) HBM = global
2) per s.m. shared memory (accessible to all threads)
3) per thread memory

- CUDA has synchronization constructs:

  - __syncthreads() → barrier to wait for all threads to arrive at some pt

  - Atomic ops → on global OR per-block shared mem

  - Host/device synchronization

    ↳ implicit barrier ensuring all threads launched finished

- summary of cuda programming model:

  1) Thread hierarchy for execution

     ↳ thread blocks then threads

  2) Distributed addr space

     - built in memcpy to copy from host to device

     - on device:

       · per thread

       · per block ("shared")

       · per program ("global")

  3) Various synchronization primitives