ECE 571 – Advanced Microprocessor-Based Design Lecture 35

Vince Weaver

`http://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

2 December 2020

Project/HW Reminder

- Don't forget Homework #11

3D Graphics Rundown

- Rasterization (traditional 3d cards)
  - Send vertices to card
  - Triangles, normals
  - Project to 2d screen
  - Broken up to pixels and shaded/textured
  - Clipping, depth
- Ray-tracing
  - Light is traced to eye (or the reverse)
- Ray-casting

Older / Traditional GPU Pipeline

- In old days, fixed pipeline (lots of triangles).

- Modern chips much more flexible, but the old pipeline can still be implemented in software via the fancier interface.

Older / Traditional GPU Pipeline

- CPU send list of vertices to GPU.
- Transform (vertex processor) (convert from world space to image space). 3d translation to 2d, calculate lighting. Operate on 4-wide vectors (x,y,z,w in projected space, r,g,b,a color space)
- Rasterizer – transform vertexes/vectors into a grid. Fragments. break up to pixels and anti-alias
- Shader (Fragment processor) compute color for each pixel. Use textures if necessary (texture memory, mostly

read)
- Write out to framebuffer (mostly write)
- Z-buffer for depth/visibility

GPGPUs

- Started when the vertex and fragment processors became generically programmable (originally to allow more advanced shading and lighting calculations)

- By having generic use can adapt to different workloads, some having more vertex operations and some more fragment

Graphics vs Programmable Use

Vertex	Vertex Processing	Data	MIMD processing
Polygon	Polygon Setup	Lists	SIMD Rasterization
Fragment	Per-pixel math	Data	Programmable SIMD
Texture	Data fetch, Blending	Data	Data Fetch
Image	Z-buffer, anti-alias	Data	Predicated Write

Shader Programming

- There are competitions. Also see `shadertoy.com`
- Vertex Shader
  - Vertex transform
  - Object space to clip space
  - Compute colors, normals, texture co-ords
  - Can displace/distort (move vertices: wave flag)
  - Can animate (move vertices: move fish)
- Fragment Shader
  - Compute and color

○ Get data from vorteces and textures
○ Can make better materials. Glossy, reflections, bumpy, shadows

GLSL Shader Programming

- Similar to C code
- Based on OpenGL
- vertex
  - Each time screen drawn main() called once per vertex
  - Massively parallel
  - Have vars. Can get positions
- Fragment
  - Each time screen drawn main() called once per pixel
  - Can get x/y

Example Shader 3.0 (DX9) Capabilities – Vertex Processor

- They are up to Pixel Shader 5.0 now
- 512 static / 65536 dynamic instructions
- Up to 32 temporary registers
- Simple flow control
- Texturing – texture data can be fetched during vertex operations
- Can do a four-wide SIMD MAD (multiply ADD) and a scalar op per cycle:

○ EXP, EXPP, LIT, LOGP (exponential)
○ RCP, RSQ (reciprocal, r-square-root)
○ SIN, COS (trig)

Example Shader 3.0 (DX9) capatbilities– Fragment Processor

- 65536 static / 65536 dynamic instructions (but can time out if takes too long)
- Supports conditional branches and loops
- fp32 and fp16 internal precision
- Can do 4-wide MAD and 4-wide DP4 (dot product)

Program

- Typically textures read-only. Some can render to texture, only way GPU can share RAM w/o going through CPU. In general data not written back until entire chunk is done. Fragment processor can read memory as often as it wants, but not write back until done.

- Only handle fixed-point or floating point values

- Analogies:
  - Textures == arrays

– Kernels == inner loops
– Render-to-texture == feedback
– Geometry-rasterization == computation. Usually done as a simple grid (quadrilateral)
– Texture-coordinates = Domain
– Vertex-coordinates = Range

Flow Control, Branches

- only recently added to GPUs, but at a performance penalty.

- Often a lot like ARM conditional execution

Terminology (CUDA)

- Thread: chunk of code running on GPU.

- Warp: group of thread running at same time in parallel simultaneously

- Block: group of threads that need to run

- Grid: a group of thread blocks that need to finish before next can be started

Terminology (cores)

- Confusing. Nvidia would say GTX285 had 240 stream processors; what they mean is 30 cores, 8 SIMD units per core.

CUDA Programming

- Since 2007
- Use `nvcc` to compile
- *host* vs *device*

  host code runs on CPU

  device code runs on GPU
- Host code compiled by host compiler (gcc), device code by custom NVidia compiler
- `__global__` parameters to function – means pass to CUDA compiler

19

- cudaMalloc() to allocate memory and pointers that can be passed in
- call global function like this `add<<<1,1>>>(args)` where first inside brackets is number of blocks, second is threads per block
- cudaFree() at the end
- Can get block number with blockIdx.x and thread index with threadIdx.x
- Can have 65536 blocks and 512 threads (At least in 2010)
- Why threads vs blocks?

Shared memory, block specific `__shared__` to specify

- `__syncthreads()` is a barrier to make sure all threads finish before continuing

More **CUDA** Programming

- See the NVIDIA "CUDA C Programming Guide"
- Compute Unified Device Architecture
- From CUDA C Programming guide from NVIDIA
- CUDA introduced in 2006
- Heterogeneous programming – there is a host executing a main body of code (a CPU) and it dispatches code to run on a device (a GPU)
- CUDA assumes host and device each have own separate DRAM memory

- CUDA C extends C, define C functions "kernels" that are executed N times in parallel by N CUDA threads

CUDA Coding

- version compliance – can check version number. New versions support more hardware but sometimes drop old
- nvcc – wrapper around gcc. global code compiled into PTX (parallel thread execution) ISA
- can code in PTX code directly which is sort of like assembly language. Won't give out *actual* assembly language. Why?
- CUDA C has mix of host and device code. Compiles the global stuff to PTX, compiles the $<<<\,...\,>>>$ into

code that can launch the GPU code

- PTX code is JIT compiled into native by the device driver
- You can control JIT with environment variables
- Only subset of C/C++ supported in the device code

CUDA Hardware

- GPU is array of Streaming Multiprocessors (SMs)
- Program partitioned into blocks of threads that execute independently from each other.
- Manages/Schedules/Executes threads in groups of 32 parallel threads (warps) (weaving terminology) (no relation)
- Threads have own PC, registers, etc, and can execute independently
- When SM given thread block, partitions to warps and

each warp gets scheduled

- One common instruction at a time. If diverge in control flow, each way executed and thread not taking that path just waits.
- Full context stored with each warp; if warp is not ready (waiting for memory) then it may be stopped and another warp that's ready can be run

CUDA Threads

- kernel defined using `__global__` declaration. When called use `<<<...>>>` to specify number of threads
- each thread that is called is assigned a unique ThreadID Use threadIdx to find what thread you are and act accordingly

```
__global__ void VecAdd(float *A, float *B
        int i = threadIdx.x;
        C[i]=A[i]+B[i];
}
```

```
int main(int argc, char **argv) {
        ....
        /* Invoke N threads */
        VecAdd<<<1,N>>>(A,B,C);
}
```

- threadIdx is 3-component vector, can be seen as 1, 2 or 3 dimensional block of threads (thread block)
- Much like our sobel code, can look as 1D (just x), 2D, (thread iD is ((y*xsize)+x) or (z*xsize*ysize)+y*xsize+x
- Weird syntax for doing 2 or 3d.

```
__global__ void MatAdd( float A[N][N], flo
{
        int i=threadIdx.x;
        int j=threadIdx.y;
        C[i][j]=A[i][j]+B[i][j];
}

int numBlocks=1;
dim3 threadsPerBlock(N,N);
MatAdd<<<numBlocks, threadsPerBlock>>>(A,
```

- Each block made up of the threads. Can have multiple levels of blocks too, can get block number with blockIdx
- Thread blocks operate independently, in any order. That way can be scheduled across arbitrary number of cores (depends how fancy your GPU is)

CUDA Memory

- Per-thread private local memory
- Shared memory visible to whole block (lifetime of block)
- Global memory
- also constant and texture spaces. Have special rules. Texture can do some filtering and stuff
- Global, constant, and texture persistent across kernel launches by same app.

More Coding

- No explicit initialization, done automatically first time you do something (keep in mind if timing)
- Global Memory: linear or arrays.
  - Arrays are textures
  - Linear arrays are allocated with cudaMalloc(), cudaFree()
  - To transfer use cudaMemcpy()
  - Also can be allocated cudaMallocPitch() cudaMalloc3D() for allignment reasons

○ Access by symbol (?)
- Shared memory, `__shared__`. Faster than Global also `__device__`
  Manually break your problem into smaller sizes

Misc

- Can lock host memory with cudaHostAlloc(). Pinned, can't be paged out. Can load store while kernel running if case. Only so much available. Can be marked writecombining. Not cached. So slow for host to read (should only write) but speeds up PCI transaction.

Async Concurrent Execution

- Instead of serial/parallel/serial/parallel model
- Want to have CUDA running and host at same time, or with mem transfers at same time
  - Concurrent host/device: calls are async and return to host before device done
  - Concurrent kernel execution: newer devices can run multiple kernels at once. Problem if use lots of memory
  - Overlap of Data Transfer and Kernel execution
  - Streams: sequence of commands that execute in order,

but can be interleaved with other streams
complicated way to set them up. Synchronization and
callbacks

Events

- Can create performance events to monitor timing
- PAPI can read out performance counters on some boards
- Often it's for a full synchronous stream, can't get values mid-operation
- NVML can measure power and temp on some boards?

Multi-device system

- Can switch between active device
- More advanced systems can access each others device memory

Other features

- Unified virtual address space (64 bit machines)
- Interprocess communication
- Error checking

Texture Memory

- Complex

3D Interop

- Can make results go to an OpenGL or Direct3D buffer
- Can then use CUDA results in your graphics program

Code Example

```c
#include <stdio.h>

#define N 10

__global__ void add (int *a, int *b, int *c
        int tid=blockIdx.x;

        if (tid<N) {
                c[tid]=a[tid]+b[tid];
```

```c
            }
}


int main(int arc, char **argv) {

        int a[N],b[N],c[N];
        int *dev_a,*dev_b,*dev_c;
        int i;

        /* Allocate memory on GPU */
```

```
cudaMalloc ( ( void  ** )& dev_a , N* sizeof
cudaMalloc ( ( void  ** )& dev_b , N* sizeof
cudaMalloc ( ( void  ** )& dev_c , N* sizeof

/*  Fill  the  host  arrays  with  values
for ( i =0; i <N; i++)  {
        a [ i ]=− i ;
        b [ i ]= i * i ;
}

cudaMemcpy ( dev_a , a , N* sizeof ( int ) , cu
```

```
cudaMemcpy( dev_b , b , N∗sizeof( int ) , cu

add<<<N,1>>>(dev_a , dev_b , dev_c );

cudaMemcpy( c , dev_c , N∗sizeof( int ) , cu

/∗ results ∗/
for ( i =0; i <N; i ++) {
        printf("%d+%d=%d\n" , a [ i ] , b
}
```

```
        cudaFree(dev_a);
        cudaFree(dev_b);
        cudaFree(dev_c);

        return 0;
}
```