Game Developers Conference® | March 9-13, 2010 | Moscone Center | San Francisco, CA

THINK SERVICES
A DIVISION OF UNITED BUSINESS MEDIA LLC

GDC 10

Learn. Network. Inspire.

www.GDConf.com

Game Developers Conference®
March 9-13, 2010
Moscone Center
San Francisco, CA
www.GDConf.com

DirectCompute Performance on DX11 Hardware

Nicolas Thibieroz, AMD
Cem Cebenoyan, NVIDIA

Why DirectCompute?

- Allow arbitrary programming of GPU
  - General-purpose programming
  - Post-process operations
  - Etc.
- Not always a win against PS though
- Well-balanced PS is unlikely to get beaten by CS
  - Better to target PS with heavy TEX or ALU bottlenecks
  - Use CS threads to divide the work and balance the shader out

Feeding the Machine

- GPUs are throughput oriented processors
  - Latencies are covered with work
- Need to provide enough work to gain efficiency
- Look for fine-grained parallelism in your problem
- Trivial mapping works best
  - Pixels on the screen
  - Particles in a simulation

Feeding the Machine (2)

- Still can be advantageous to run a small computation on the GPU if it helps avoid a round trip to host
  - Latency benefit
  - Example: massaging parameters for subsequent kernel launches or draw calls
- Combine with DispatchIndirect() to get more work done without CPU intervention

Scalar vs Vector

- NVIDIA GPUs are *scalar*
  - Explicit vectorization unnecessary
    - Won't hurt in most cases, but there are exceptions
  - Map threads to scalar data elements
- AMD GPUs are *vector*
  - Vectorization *critical* to performance
  - Avoid dependant scalar instructions
- Use IHV tools to check ALU usage

CS5.0 >> CS4.0

- CS5.0 is just better than CS4.0
- More of everything
  - Threads
  - Thread Group Shared Memory
  - Atomics
  - Flexibility
  - Etc.
- Will typically run faster
  - If taking advantage of CS5.0 features
- Prefer CS5.0 over CS4.0 if D3D_FEATURE_LEVEL_11_0 supported

Thread Group Declaration

- Declaring a suitable number of thread groups is *essential* to performance

```
numthreads(NUM_THREADS_X, NUM_THREADS_Y, 1)
void MyCSShader(...)
```

- Total thread group size should be above hardware's wavefront size
  - Size varies depending on GPUs!
  - ATI HW is 64 at max. NV HW is 32.
- Avoid sizes below wavefront size
  - `numthreads(1,1,1)` is a *bad idea*!
- Larger values will generally work well across a wide range of GPUs
  - Better scaling with lower-end GPUs

Thread Group Usage

- Try to divide work evenly among all threads in a group
- Dynamic Flow Control will create divergent workflows for threads
  - This means threads doing less work will sit idle while others are still busy

```
[numthreads(groupthreads,1,1)]
void CSMain(uint3 Gid : SV_GroupID,
            uint3 Gtid: SV_GroupThreadID)
{
    ...
    if (Gtid.x == 0)
    {
      // Code here is only executed for one thread
    }
}
```

!

Mixing Compute and Raster

- Reduce number of transitions between Compute and Draw calls
- Those transitions can be expensive!

Compute A		Compute A
Compute B		Draw X
Compute C	>>	Compute B
Draw X		Draw Y
Draw Y		Compute C
Draw Z		Draw Z

Unordered Access Views

- UAV not strictly a DirectCompute resource
  - Can be used with PS too
- Unordered Access support scattered R/W
  - Scattered access = cache trashing
  - Prefer grouped reads/writes (bursting)
  - E.g. Read/write from/to float4 instead of float
  - NVIDIA scalar arch will not benefit from this
- Contiguous writes to UAVs
- Do not create a buffer or texture with UAV flag if not required
  - May require synchronization after render ops
  - D3D11_BIND_UNORDERED_ACCESS only if needed!
- Avoid using UAVs as a scratch pad!
  - Better use TGSM for this

Buffer UAV with Counter

- Shader Model 5.0 supports a counter on *Buffer* UAVs
  - Not supported on textures
  - `D3D11_BUFFER_UAV_FLAG_COUNTER` flag in `CreateUnorderedAccessView()`
- Accessible via:
  - `uint IncrementCounter();`
  - `uint DecrementCounter();`
- Faster method than implementing manual counter with UINT32-sized R/W UAV
  - Avoids need for atomic operation on UAV
- See Linked List presentation for an example of this
- On NVIDIA HW, prefer Append buffers

Append/Consume buffers

- Useful for serializing output of a data-parallel kernel into an array
  - Can be used in graphics, too!
  - E.g. deferred fragment processing
- Use with care, can be costly
  - Introduce serialization point in the API
  - Large record sizes can hide the cost of append operation

Atomic Operations

- "Operation that cannot be interrupted by other threads until it has completed"
  - Typically used with UAVs
- Atomic operations cost performance
  - Due to synchronization needs
- Use them only when needed
  - Many problems can be recast as more efficient parallel reduce or scan
- Atomic ops with feedback cost even more

E.g.
```
Buf->InterlockedAdd(uAddress, 1,
                    Previous);
```

Thread Group Shared Memory

- Fast memory shared across threads *within a group*
  - *Not* shared across thread groups!
  - `groupshared float2 MyArray[16][32];`
  - Not persistent between Dispatch() calls
- Used to reduce computation
  - Use neighboring calculations by storing them in TGSM
  - E.g. Post-processing texture instructions

TGSM Performance (1)

- Access patterns matter!
  - Limited number of I/O banks
  - 32 banks on ATI and NVIDIA HW
- Bank conflicts will reduce performance

TGSM Performance (2)

- 32 banks example
  - Each address is 32 bits
- Banks are arranged linearly with addresses:

Address:	0	1	2	3	4	...	31	32	33	34	35	...
Bank:	0	1	2	3	4	...	31	0	1	2	3	...

- TGSM addresses that are 32 DWORD apart use the same bank
- Accessing those addresses from multiple threads will create a *bank conflict*
- Declare TGSM 2D arrays as MyArray[Y][X], and increment X first, then Y
  - Essential if X is a multiple of 32!
- Padding arrays/structures to avoid bank conflicts can help
  - E.g. MyArray[16][33] instead of [16][32]

TGSM Performance (3)

- Reduce access whenever possible
  - E.g. Pack data into uint instead of float4
  - But watch out for increased ALUs!
- Basically try to read/write once per TGSM address
  - Copy to temp array can help if it avoids duplicate accesses!
- Unroll loops accessing shared mem
  - Helps compiler hide latency

Barriers

- Barriers add a synchronization point for all threads within a group
  - GroupMemoryBarrier()
  - GroupMemoryBarrierWithGroupSync()
- Too many barriers will affect performance
  - Especially true if work is not divided evenly among threads
- Watch out for algorithms using many barriers

Maximizing HW Occupancy

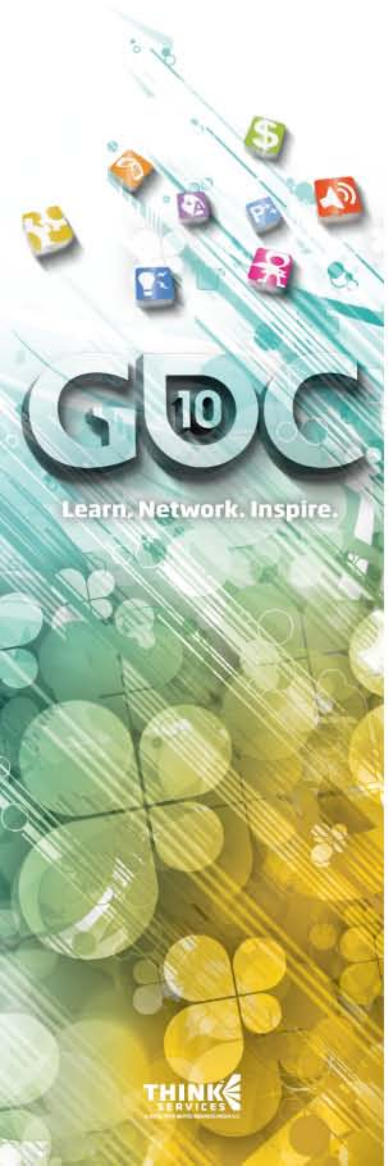
- A thread group cannot be split across multiple shader units
  - Either in or out
  - Unlike pixel work, which can be arbitrarily fragmented
- Occupancy affected by:
  - Thread group size declaration
  - TGSM size declared
  - Number of GPRs used
- Those numbers affect the level of parallelism that can be achieved

Maximizing HW Occupancy (2)

- **Example:** HW shader unit:
  - **8** thread groups max
  - **32KB** total shared memory
  - **1024** threads max
- With thread group size of 128 threads requiring 24KB of shared memory can only run **1** thread group per shader unit (128 threads) **BAD**
- Ask your IHVs about GPU Computing documentation

Maximizing HW Occupancy (3)

- Register pressure will also affect occupancy
  - You have little control over this
  - Rely on drivers to do the right thing ☺
- Tuning and experimentations are required to find the ideal balance
  - But this balance varies from HW to HW!
  - Store different presets for best performance across a variety of GPUs



Conclusion

- Threadgroup size declaration essential to performance
- I/O can be a bottleneck
- TGSM tuning is important
- Minimize PS->CS->PS transitions
- HW occupancy is GPU-dependent
- DXSDK DirectCompute samples not necessarily using best practices atm!
  - E.g. HDRToneMapping, OIT11

Questions?

Nicolas.Thibieroz@amd.com
cem@nvidia.com