L10: Floating Point Issues and Project

Administrative Issues

- Project proposals
  - Due 5PM, Friday, March 13 (hard deadline)
- Homework (Lab 2)
  - Due 5PM, Wednesday, March 4
  - Where are we?

Outline

- Floating point
  - Mostly single precision
  - Accuracy
  - What's fast and what's not
  - Reading: Programmer's Guide, Appendix B
- Project
  - Ideas on how to approach MPM/GIMP
  - Construct list of questions

Single Precision vs. Double Precision

- Platforms of compute capability 1.2 and below only support single precision floating point
- New systems (GTX, 200 series, Tesla) include double precision, but much slower than single precision
  - A single dp arithmetic unit shared by all SPs in an SM
  - Similarly, a single fused multiply-add unit
- Suggested strategy:
  - Maximize single precision, use double precision only where needed

Summary: Accuracy vs. Performance

- A few operators are IEEE 754-compliant
  - Addition and Multiplication
- … but some give up precision, presumably in favor of speed or hardware simplicity
  - Particularly, division
- Many built in intrinsics perform common complex operations very fast
- Some intrinsics have multiple implementations, to trade off speed and accuracy
  - e.g., intrinsic __sin() (fast but imprecise) versus sin() (much slower)

Deviations from IEEE-754

- Addition and Multiplication are IEEE 754 compliant
  - Maximum 0.5 ulp (units in the least place) error
- However, often combined into multiply-add (FMAD)
  - Intermediate result is truncated
- Division is non-compliant (2 ulp)
- Not all rounding modes are supported
- Denormalized numbers are not supported
- No mechanism to detect floating-point exceptions

Arithmetic Instruction Throughput

- int and float add, shift, min, max and float mul, mad: 4 cycles per warp
  - int multiply (*) is by default 32-bit
    - requires multiple cycles / warp
  - Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply
- Integer divide and modulo are expensive
  - Compiler will convert literal power-of-2 divides to shifts
  - Be explicit in cases where compiler can't tell that divisor is a power of 2!
  - Useful trick: foo % n == foo & (n-1) if n is a power of 2

Arithmetic Instruction Throughput

- Reciprocal, reciprocal square root, sin/cos, log, exp: 16 cycles per warp
  - These are the versions prefixed with "__"
  - Examples:__rcp(), __sin(), __exp()
- Other functions are combinations of the above
  - y / x == rcp(x) * y == 20 cycles per warp
  - sqrt(x) == rcp(rsqrt(x)) == 32 cycles per warp

Runtime Math Library

- There are two types of runtime math operations
  - __func(): direct mapping to hardware ISA
    - Fast but low accuracy (see prog. guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy (5 ulp, units in the least place, or less)
    - Examples: sin(x), exp(x), pow(x,y)
- The -use_fast_math compiler option forces every func() to compile to __func()

Make your program float-safe!

- Future hardware will have double precision support
  - G80 is single-precision only
  - Double precision will have additional performance cost
  - Careless use of double or undeclared types may run more slowly on G80+
- Important to be float-safe (be explicit whenever you want single precision) to avoid using double precision where it is not needed
  - Add 'f' specifier on float literals:
    - `foo = bar * 0.123;    // double assumed`
    - `foo = bar * 0.123f;   // float explicit`
  - Use float version of standard library functions
    - `foo = sin(bar);    // double assumed`
    - `foo = sinf(bar);   // single precision explicit`

Reminder: Content of Proposal, MPM/GIMP as Example

I. Team members: Name and a sentence on expertise for each member

Obvious

II. Problem description

- What is the computation and why is it important?
- Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page

Straightforward adaptation from MPM presentation and/or code

III. Suitability for GPU acceleration

- Amdahl's Law: describe the inherent parallelism. Argue that it is close to 100% of computation. Use measurements from CPU execution of computation if possible

Can measure sequential code

Reminder: Content of Proposal, MPM/GIMP as Example

III. Suitability for GPU acceleration, cont.

- Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.

Some challenges, see remainder of lecture

- Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.

Measure grid and patches to discover data footprint. Consider ways to combine computations to reduce copying overhead.

IV. Intellectual Challenges

- Generally, what makes this computation worthy of a project?

Importance of computation, and challenges in partitioning computation, dealing with scope, managing copying overhead

- Point to any difficulties you anticipate at present in achieving high speedup

See previous

Projects – How to Approach

- Example: MPM/GIMP
- Some questions:
  1. Amdahl's Law: target bulk of computation
     and can profile to obtain key computations…
  2. Strategy for gradually adding GPU execution to CPU code while maintaining correctness
  3. How to partition data & computation to avoid synchronization?
  4. What types of floating point operations and accuracy requirements?
  5. How to manage copy overhead?

1. Amdahl's Law

- Significant fraction of overall computation?
  - Simple test:
    - Time execution of computation to be executed on GPU in sequential program.
    - What is its percentage of program's total execution time?
- Where is sequential code spending most of its time?
  - Use profiling (gprof, pixie, VTUNE, …)

2. Strategy for Gradual GPU…

- Looking at MPM/GIMP
  - Several core functions used repeatedly (integrate, interpolate, gradient, divergence)
  - Can we parallelize these individually as a first step?
  - Consider computations and data structures

2. cont.

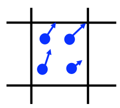
```
void operations<S>::integrate(const patch&pch,const
   vector<double>&pu,vector<double>&gu){
  for(unsigned g=0;g<gu.size();g++)gu[g]=0.;
  for(unsigned p=0;p<pu.size();p++){
    const partContribs&pcon=pch.pCon[p];
    for(int k=0;k<pcon.Npor;k+=1){
      const partContribs::portion&por=pcon[k];
      gu[por.idx]+=pu[p]*por.weight;
    }
  }
}
```

Most data structures are read only!

gu (representing the grid) is updated, but only by nearby particles.

3. Synchronization Recall from MPM Presentation

Blue dots corresponding to particles (pu).
Grid structure corresponds to nodes (gu).



How to parallelize without incurring synchronization overhead?

2. and 3.

- Other common structure in code

```
template<typename S> void operations<S>::interpolate(const
   patch&pch,vector<Vector2>&pu,const vector<Vector2>&gu){
  for(unsigned p=0;p<pu.size();p++)pu[p]=0.;
  for(unsigned p=0;p<pu.size();p++){
    const partContribs&pcon=pch.pCon[p];
    Vector2&puR=pu[p];
    for(int k=0;k<pcon.Npor;k+=1){
      const partContribs::portion&por=pcon[k];
      const Vector2&guR=gu[por.idx];
      puR.x+=guR.x*por.weight;
      puR.y+=guR.y*por.weight;
    }
  }
}
```

puR (representing the particles) is updated, but only by nearby grid points.

4. Floating Point

- MPM/GIMP is a double precision code!
- Phil:
  - Double precision needed for convergence on fine meshes
  - Single precision ok for coarse meshes
- Conclusion:
  - Converting to single precision (float) ok for this assignment, but hybrid single/double more desirable in the future

5. Copy overhead?

- Some example code in MPM/GIMP

```
sh.integrate (pch,pch.pm,pch.gm);
sh.integrate (pch,pch.pfe,pch.gfe);
sh.divergence(pch,pch.pVS,pch.gfi);
for(int i=0;i<pch.Nnode();++i)pch.gm[i]+=machTol;
for(int i=0;i<pch.Nnode();++i)pch.ga[i]=(pch.gfe[i]+pch.gfi[i])/
   pch.gm[i];
…
```

Exploit reuse of gm, gfe, gfi
Defer copy back to host.

Other MPM/GIMP Questions

- Lab machine set up? Python? Gnuplot?
- Hybrid data structure to deal with updates to grid in some cases and particles in other cases

Next Class

- Discussion of tools