Efficient Data Parallel Computing on GPUs

Cliff Woolley
University of Virginia / NVIDIA
In this part of the course, we'll look at some tricks and traps for programming the GPU for general purpose computation more effectively. This first part of this section will attempt to get you, the CPU programmer, to start to think in "GPU terms." There are some mistakes that every beginning GPU programmer (even those who are experienced CPU programmers) seem to make; those are the things I'm aiming to address in this part of the talk. After this introduction to "GPU thinking," Aaron and Ian will delve into more details about exactly how you can sculpt your algorithms so that they make the best use of what the GPU has to offer.
The first thing you'll have to get used to as a GPU programmer, if you've not played at all with vector processing before (eg, MMX or SSE), is that the GPU is inherently a vector processor; this provides you with a certain amount of work that can be done for free, if you just stop to think about how to take advantage of it. You can also think of the GPU as a stream processor – this allows another level of data parallelism to be leveraged. Which of these two "scales" of parallelism you should focus on depends on your problem, but in both cases, to spot the possibilities, you'll have to start to think a bit differently than you might be accustomed to.
The easiest way to explain what I mean by most of the following is to give you an example to go from. Ignore the fact for now that this fragment program is more or less unreadable... we’re going to factor it out to have it make more sense (and, more importantly, be more efficient), as we go along.

This example was originally written for the paper "A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware", Goodnight et al., Graphics Hardware 2003. The complete original shader is in the course notes. My task in working on the research that went into that paper was to take shaders like this one and optimize them so that they’d perform more reasonably. I’ll walk you through some of those transformations in the following slides.
Data Parallel Computing: Vector Processing

```cpp
fragframe smoothVert2Frag IN, uniform samplerRECT Source : texunit0, uniform samplerRECT Operator : texunit1, uniform samplerRECT Boundary : texunit2, uniform float4 params)
{
  fragframe OUT;
  float2 center = IN.TexCoord0.xy;
  float4 U = floatRECT(Source, center);
  // Calculate Red-Black (odd-even) masks
  float2 mask;
  float2 place = floor(1.0f - modf(round(center + float2(0.5f, 0.5f)) / 2.0f, intpart));
  float2 mask = float2(1.0f-place.x, 1.0f-place.y); place.a = place.y;
  if (((mask.x & mask.y) || params.y) || !(mask.x & mask.y) && !params.y)
  {
    float2 offset = float2(params.x-center.x - 0.5f*params.x-1.0f, params.x-center.y - 0.5f*params.x-1.0f);
    float central = floatRECT(Source, float2(center.x - 1.0f, center.x + 1.0f, center.y - 1.0f, center.y + 1.0f));
    float central = -2.0f*(0.5f + 0.5f);
    float poison = (params.x*params.x)*x + {-0.5f * floatRECT(Source, float2(neighbors, center.x)) + 
      -0.5f * floatRECT(Source, float2(neighbors, center.y)) + 
      -0.5f * floatRECT(Source, float2(neighbors, center.w)) + 
      -0.5f * floatRECT(Source, float2(neighbors, center.w)) / 0.0f;}
    OUT.COL.a = poison;
  }
  return OUT;
}
```

The first kind of data parallelism you should get used to looking for is "local" data parallelism in the vector processing sense. In these two lines of code, you can see that each multiple and each add or subtract is operating on a scalar value. That's a waste of computational resources! If you can manage to get all of the data packed into a 4-vector, you can get all of your multiplies done in a single instruction, for example. The trick is to find a way to do it such that the packing of data amounts to fewer instructions than the original, separate multiplies would have been. In the second case listed above, that's particularly easy... swizzling is your friend! For the first line, a little simple algebraic manipulation turns out to work wonders...

Granted, high-level compilers for GPU code and the internal run-time optimizers contained within the drivers are getting better all the time. In practice, however, it still seems to be the case that the more you can exploit the vector nature of the hardware yourself in your fragment and vertex programs, the better off you’ll be. (Of course, make sure you run a benchmark both before and after each optimization you apply to make sure that it really does give you a speedup.)
float2 offset = center.xy - 0.5f;
offset = offset * params.xx + 0.5f; // MADR is cool too - one cycle, two flops

float4 neighbor = center.xxyy + float4(-1.0f,1.0f,-1.0f,1.0f);

So here's what those same two lines might look like if we were to pack the data so as to take advantage of the vector processor.

A particular combination of operations to watch out for is multiply-and-add, since that's a single instruction on both NVIDIA and ATI GPU's. We can thus turn our nasty offset computation, which had four multiplies and four subtractions into a single subtract and a single multiply-and-add.

One thing you might notice is that even though we've taken better advantage of the vector operations in the first example, we're still operating on 2-vectors rather than 4-vectors. This means half of our computational power for those two instructions is wasted. Superscalar capabilities of new and upcoming GPUs might mitigate this problem; but it's still worth keeping in mind, because if there are enough of these cases, we should consider looking at a more broad scope for other work we could have done at the same time (for free). Maybe if we can rearrange the operations in our shader, we could get some other subtractions and/or MADR's done at the same time, for example.
But there's an even broader scope we should consider when looking for parallelism. Frequently in GPGPU-land, the geometry we end up drawing is really big quads, so that we can have our fragment programs run on a big 2D array of data that happens to be represented as a texture map of some kind. Sometimes, the most effective parallelism you can get turns out to be to pack the data itself more efficiently. This gives you opportunities to have each instruction do four times as much work more readily, of course, but it also gives the additional important benefit that it cuts down on your memory bandwidth usage. That's not to say that this approach will always be applicable; maybe you need some of those extra channels to store other data in, for example. But it's worth considering whether moving that extra data to a separate texture and stacking the bulk of your data (also called "domain decomposition") could give you a speedup. Hint: if you're memory-bandwidth bound, it very well might.

Figure courtesy Aaron Lefohn
Computational Frequency
Computational Frequency

- Think of your CPU program and your vertex and fragment programs as different levels of nested looping.
  - CPU Program
    - Vertex Program
    - Fragment Program
An interesting aspect of "GPU thinking" is that it's both possible and likely (at least for GPGPU applications) that subsequent stages of the GPU pipeline will have to do increasing amounts of work. For every big quad you draw, only four vertices get processed, but many many more fragments result. So the next part of wrapping your brain around efficient use of the GPU is deciding exactly how frequently each value you use needs to be computed. To put it in more familiar terms, this is more or less identical to loop invariant code from nested loops in a CPU program, except that the different levels of loop nesting in this case translate into entirely separate programs *that run in parallel when possible*. 
Computational Frequency: Avoid inner-loop branching

- Static branch resolution
  - write several variants of each fragment program to handle boundary cases
  - eliminates conditionals in the fragment program
  - equivalent to avoiding CPU inner-loop branching

There's one other aspect of computational frequency that I've yet to mention. In this case it's not so much a matter of precomputation as it is optimizing the "fast path" if you have a big block of code in your fragment program that will only get executed on some fragments and not on others. It might well be advantageous to split such a case up into multiple fragment programs; one that uses the block and one that omits it. Then just draw more than one primitive, using each shader in turn. In the case shown here, our sample "smooth" shader is much more complex on the boundaries of our domain than on the interior of the domain. So we split up the shader into two; one fast-path shader that doesn't handle any boundary conditions, and a slower one that does handle them. Again this is equivalent to an optimization you might do in equivalent CPU code; if you have a conditional in your inner loop, it might be advantageous to split it into two loops, one of which handles all the "true" cases for the condition and one of which handles all the "false" cases.

Following are the final versions of the two smooth shaders after all optimizations have been applied.
An interesting aspect of "GPU thinking" is that it's both possible and likely (at least for GPGPU applications) that subsequent stages of the GPU pipeline will have to do increasing amounts of work. For every big quad you draw, only four vertices get processed, but many many more fragments result. So the next part of wrapping your brain around efficient use of the GPU is deciding exactly how frequently each value you use needs to be computed. To put it in more familiar terms, this is more or less identical to loop invariant code from nested loops in a CPU program, except that the different levels of loop nesting in this case translate into entirely separate programs \textit{that run in parallel when possible}.
An interesting aspect of "GPU thinking" is that it's both possible and likely (at least for GPGPU applications) that subsequent stages of the GPU pipeline will have to do increasing amounts of work. For every big quad you draw, only four vertices get processed, but many many more fragments result. So the next part of wrapping your brain around efficient use of the GPU is deciding exactly how frequently each value you use needs to be computed. To put it in more familiar terms, this is more or less identical to loop invariant code from nested loops in a CPU program, except that the different levels of loop nesting in this case translate into entirely separate programs *that run in parallel when possible.*
The first mistake you’re likely to make along these lines is to compute texture coordinates inside your fragment program when you don’t really need to. If the texture coordinates (or any other value, for that matter) vary linearly across your domain, let the rasterizer do the work for you! This means you’ll need to split your big nasty fragment program into a fragment program and a vertex program. Not only will you avoid doing redundant computation, but you’ll make a bit more use of the vertex processor (which was probably sitting idle anyway if you use the fragment processor as heavily as most of us GPGPU types do), and you might even get a slight boost in your texture read performance (computing texture coordinates inside the fragment program causes those texture coordinates to effectively be treated as a texture indirection, possibly defeating prefetching.)
Here we have a vertex program that was created by stripping all of the texture coordinate computation out of the shader above. You end up with parts of your fragment program looking like this:

```
OUT.COL.x = params.w*O.z +
  (params.z*U.z - (O.x *(fltexRECT(Source, IN.hneighbor.xz)+
                fltexRECT(Source, IN.hneighbor.yz))) +
  O.y * fltexRECT(Source, IN.vneighbor.xy) +
  O.y * fltexRECT(Source, IN.vneighbor.xz))) / O.w;
```

Keep in mind that, once again, swizzling is our friend; the rasterizer will interpolate 4-vectors for us as cheaply as 2-vectors, so we might as well get the most bang for our buck. Using the swizzle to select which channels of a varying parameter contain your texture coordinates is efficient for several reasons; first, as I mentioned before, it aids in prefetching and reduces computations done in the fragment shader. Perhaps less obviously, though, fetching the channels you want out of a single pre-existing 4-vector by swizzling is free, whereas if you built a 2-vector on the fly, it would turn into several MOV instructions. So, for example,

```
fltexRECT(Source, IN.hneighbor.xz);
```

would be better than:

```
fltexRECT(Source, float2(IN.foo.x, IN.bar.y));
```
Computational Frequency: Precomputing texcoords

fragFrame smooth(rect2frag IN, uniform samplerRECT Source : texunit0, uniform samplerRECT Operator : texunit1, uniform samplerRECT Boundary : texunit2, uniform float4 params)
{
  fragFrame OUT;
  float2 center = INTexCoord0.xy;
  float4 U = fractRECT(Source, center);
  // Calculate odd-even (black-red) masks
  float3 impmask = fract(RAND(0.5f - modf(round(center + float2(0.5f, 0.5f)) / 2.0f, intpart));
  float2 mask = fract(0.5f-place(0) + 0.5f-place(1));
  if ((mask.x & mask.y) || (!mask.x & !mask.y))
  {
    float2 offset = float2(params.x*center.x - 0.5f*params.x - 0.5f, params.x*center.y - 0.5f*params.x - 0.5f);
    float4 neighbor = fract(float2(center.x + 0.5f, center.y - 1.0f), center.x - 1.0f, center.y - 1.0f);
    float centerz = -2.0f*IN.x + 0.5f;
    float poisson = (params.x*params.x)*IN.x + 0.5f + fractRECT(Source, float3(neighbor.x, center.y) +
                  -0.5f + fractRECT(Source, float3(neighbor.y, center.y) +
                  -0.5f + fractRECT(Source, float3(neighbor.z, center.w)) +
                  -0.5f + fractRECT(Source, float3(neighbor.w, center.w))) / 2.0f;
    OUT.COL.x = poisson;
  }
  return OUT;
}
Computational Frequency:
Precomputing other values

- Same deal! Factor other computations out:
  - Anything that varies linearly across the geometry
  - Anything that has a complex value computed per-vertex
  - Anything that is uniform across the geometry

This same concept applies even to values that aren't texture coordinates; if it varies linearly across the geometry, turn it into a texture coordinate (who cares if it really is one) and compute it in the vertex program. So once you've learned the texture coordinate lesson, don't forget to apply it elsewhere, as well! Sounds simple, but it's easy to forget or overlook.
You can sometimes even take what we did in the previous slide a step further and do some of the work on the CPU... use your `glMultiTexCoord4f`'s and other per-vertex data passed from the CPU creatively. Furthermore, don't neglect your uniform parameters; values that are invariant both per-fragment AND per-vertex can definitely be done on the CPU once rather than multiple times either per-vertex or per-fragment. Again, this is just like removing loop-invariant code from nested loops, even though we've now got three levels of code we're dealing with -- things computed on the CPU, things computed on a per-vertex basis (and then interpolated), and things computed per-fragment; it's still the same principle. But let's stop and look more closely at this: you might actually get slightly better-performing code if you think of the CPU program/vertex program/fragment program trio as a set of nested loops rather than as function calls. For example, if you have a value "size" that is uniform across your domain, it would make sense from a function call argument-passing perspective to pass size as a uniform parameter to your vertex or fragment program. But let's say you never actually use size – only size*size+100. That expression is also uniform across the domain. If you think of the different levels of programming as nested loops, passing size*size+100 as the uniform value makes sense; it's just a matter of removing loop-invariant code from the inner loops. From the function call perspective, however, passing size*size+100 rather than size makes no sense at all, as it has little semantic meaning. That's why I say it's better to think in terms of loop invariants rather than function call arguments.
The other way to use precomputation to your advantage is to build lookup tables and bind them as texture maps. The cases where this will be potentially advantageous are the ones where you have values that vary non-linearly across your geometry but are nevertheless predictable; maybe the value of this one variable at a given pixel (x,y) location is always fixed regardless of the other data at that location, for example. Or maybe it’s simply some function f(x), where you can have a 1D-texture (or rather, a 1-by-N 2D texture, since 2D textures are better optimized in the driver than 1D textures) that provides the value of f(x) in one texture lookup rather than by computing it in the fragment processor.

In this slide we see an example from our smooth shader. The original shader would take the window coordinate of the fragment being processed and would determine whether it was red or black (in the checkerboard sense). It turns out to be more efficient to build a texture map as a preprocess that replaces all of this computation with a value that specifies whether each cell is red or black.
Once you’ve factored out the computation into a lookup table, the code that executes per-fragment is reduced to something much simpler. In my case, what was left is shown here.

It’s important to note that, unlike the other types of precomputation you can do, this one is less guaranteed to provide a speedup. If you’re memory bandwidth limited, for example, adding another memory fetch might slow things down rather than speeding them up. Whether it does or not will probably depend on just how much computation you’re removing in favor of the table lookup. Benchmarking is the best way to know in this case -- try it and see.
 Computational Frequency: Precomputed lookup tables

- Be careful with texture lookups – cache coherence is crucial
- Use the smallest data types you can get away with to reduce bandwidth consumption

- “Computation is cheap; memory accesses are not.” ...if you’re memory access limited.
Profiling and Load Balancing

- Software profiling
- GPU pipeline profiling
- GPU load balancing

While I singled out table lookups above as being particularly needy of benchmarking, any optimization you apply will need to be benchmarked to be certain you got a speedup as you expected you would. Sometimes counterintuitive things enter the picture, and a code transformation that you were *sure* would give a speedup actually slows your program down.

Once you've gone as far as you can go by just inspecting your fragment program for things you can factor out from it, though, how do you get additional speedups?

At this point, you have to start trying to figure out exactly where in your overall system the bottleneck is. As I mentioned earlier, if you can shift work from busy parts of the GPU to idle parts, such as from the fragment processor to the vertex processor, that's all to the good. But if you're not sure which is the busy part, how do you know how to balance the load?
The first step in profiling, even on a GPU application, is always to run a standard software profiler. As important as this is for optimizing CPU applications, it's no less important for GPU applications. Even if it it's not the case that the CPU end of your application does a lot of work, a software profiler will help pinpoint which of your shaders needs the most attention if you have many (and will tell you what the relative time spent on each is). Furthermore, it might be the case that your real bottleneck isn’t on the GPU at all, but rather is due to driver overhead (such as in context switching) or other CPU-side effects, even if your app seems relatively simple on the CPU side. A software profiler will sniff these kinds of problems out early before you waste a lot of time. Keep running your profiler repeatedly as you make improvements; new bottlenecks may emerge at unexpected times.
Once you start trying to find (and remove) the bottleneck in your use of the GPU itself, however, things start to get a bit hairy. Listed here are a couple of tools (and in the case of NVPerfHUD, a talk that NVIDIA presented at Game Developers Conference Europe 2003 that talks about NVPerfHUD and other profiling topics) that will help you to track down the bottleneck.

If neither of these tools does the job for you (for example, NVPerfHUD as of this writing only works on D3D applications, not OpenGL), you can do a bit of pseudo-profiling yourself. Basically you just start reducing the work on each part of the pipeline systematically; if reducing the work of the fragment processor (by temporarily reducing the instruction count in your fragment program) causes a speedup, then you’re probably compute bound in the fragment processor; if reducing the number of texture reads causes a speedup (or increasing it causes a slowdown), you could be texture-bandwidth bound; etc.
Once you know where your GPU pipeline bottleneck is and understand it, there are any number of techniques you can use to try to mitigate it. Ultimately the goal is to shift work around (even if such shifts are counter-intuitive) to exploit the inherent parallelism of the GPU. These range from manipulations of what kind of graphics API calls you use in your CPU program (see the above-referenced talk by John Spitzer from NVIDIA, also from Game Developers Conference Europe 2003) to rearranging your entire data layout to try to balance the load.

*Understanding* the bottleneck is usually key to knowing how to get it to go away. If you have contacts at NVIDIA or ATI, use them – they’re quite helpful with tracking down application bottlenecks. If you don’t, why not? :) But there’s also something to be said for having more a more fundamental understanding of the pipeline as a whole, and sometimes you can get more details on this from third parties who have studied the architectures than you can from the vendors themselves. For example, listed here are a couple of articles from 3DCenter that give some good insights into the NV3x and NV4x pipelines. They may be educated guesses rather than solid facts, but having more information at your disposal rather than less is probably still a good thing.
Conclusions
Conclusions

- Get used to thinking in terms of vector computation
- Understand how frequently each computation will run, and reduce that frequency wherever possible
- Track down bottlenecks in your application, and shift work to other parts of the system that are idle

Once you get used to thinking like a GPU, you’ll start to realize that most of the things you needed to know to make your GPU apps run fast are things you already knew for making your CPU apps run fast – it just probably wasn’t immediately obvious when you started how the concepts mapped from CPU-land to GPU-land. Hopefully through this talk you’ve gained some insight into how to make that mental leap, and you’ll eek more and more performance out of your system from the most unexpected of places. :)
Questions?

- Acknowledgements
  - Pat Brown and Nick Triantos at NVIDIA
  - NVIDIA for having given me a job this summer
  - Dave Luebke, my advisor
  - GPGPU course presenters
  - NSF Award #0092793