

# Efficient Data Parallel Computing on GPUs

Cliff Woolley  
University of Virginia / NVIDIA



## Overview

- Data Parallel Computing
- Computational Frequency
- Profiling and Load Balancing



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.

## Data Parallel Computing



## Data Parallel Computing

- Instruction-Level Parallelism
- Data-Level Parallelism



While modern CPUs do have SIMD processing extensions such as MMX or SSE, most CPU programmers never attempt to use these capabilities themselves. As a result, it's not uncommon to see new GPU programmers writing code that ineffectively utilizes vector arithmetic – in other words, their code fails to leverage instruction-level parallelism. Alternatively, some problems are inherently scalar in nature and can be more effectively parallelized by operating on multiple domain elements simultaneously. This data-level parallelism is particularly common in GPGPU applications.

## A really naïve shader

```
#pragma SmoothVertFlag 10, uniform samplerRECT Source : texunit0, uniform samplerRECT Operator : texunit1,
uniform samplerRECT Boundary : texunit2, uniform float4 params)

#pragma shader OUT;

float2 center = 20.0*vecCoord.xy;
float2 U = float2RECT(Source, center);
// Calculate Red-Black (odd-even) masks
float2 Intpart;
float2 frac = float(1.0f - mod(round(center.x + float(0.5f, 0.5f)) / 2.0f, intpart));
float2 mask = float(1.0f-frac.x) * (1.0f-frac.y), frac.x * frac.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));
    //
    float4 neighbor = float4(center.x - 1.0f, center.x + 1.0f, center.y - 1.0f, center.y + 1.0f);
    float4 texel = float4(U * 0.25);
    float4 possum = ((params.x*params.x)*U.x + (0.0 * float2RECT(Source, float2(neighbor.x, center.y)) +
    -0.4 * float2RECT(Source, float2(neighbor.y, center.y)) +
    -0.2 * float2RECT(Source, float2(center.x, neighbor.x)) +
    -0.2 * float2RECT(Source, float2(center.x, neighbor.y)))) / 0.4;
    OUT.COLOR = possum;
}
return OUT;
```



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.

## A really naïve shader

```
...
float2 offset = float2(params.x*center.x - 0.5f*(params.x-1.0f), params.x*center.y - 0.5f*(params.x-1.0f));
//
float4 neighbor = float4(center.x - 1.0f, center.x + 1.0f, center.y - 1.0f, center.y + 1.0f);
float4 texel = float4(U * 0.25);
float4 possum = ((params.x*params.x)*U.x + (0.0 * float2RECT(Source, float2(neighbor.x, center.y)) +
-0.4 * float2RECT(Source, float2(neighbor.y, center.y)) +
-0.2 * float2RECT(Source, float2(center.x, neighbor.x)) +
-0.2 * float2RECT(Source, float2(center.x, neighbor.y)))) / 0.4;
OUT.COLOR = possum;
return OUT;
```



Let's look at the highlighted lines first.

## Instruction-Level Parallelism

```
float2 offset = float2(params.x*center.x - 0.5f*(params.x-1.0f),
params.x*center.y - 0.5f*(params.x-1.0f));

float4 neighbor = float4(center.x - 1.0f,
center.x + 1.0f,
center.y - 1.0f,
center.y + 1.0f);
```



The first kind of parallelism you should get used to looking for is instruction-level parallelism. In these two lines of code, you can see that each multiplication and each addition or subtraction 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 multiples 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 multiples 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, and they handle this sort of situation better than they used to. 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.)

## Instruction-Level Parallelism

```
float2 offset = center.xy - 0.5f;
offset = offset * params.xx + 0.5f; // MADR is cool too - one
cycle, two flops

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



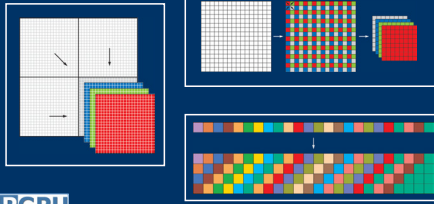
So here's what those same two lines might look like if we were to pack the data so as to extract better instruction-level parallelism.

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 multiples and four subtractions into a single subtraction and a single multiply-and-add.

One thing you might notice is that even though we've gotten better ILP out of 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 newer GPUs might mitigate this problem; for example, the NV40 has "co-issue" capability, which means that two 2-vector operations can run in parallel at the same speed as a single 4-vector operation. But it's still worth keeping this kind of situation in mind, because if you find yourself consistently relying on co-issue within a single shader, you might not have enough arithmetic to fill up all of the available slots. In that situation, you might be better off looking for parallelism at the data level rather than at the instruction level.

## Data-Level Parallelism

- Pack scalar data into RGBA in texture memory



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.

Here we see two common approaches and one custom approach to data packing. The two common approaches simply rearrange the data, either packing domain quadrants into the four color channels of a  $\frac{1}{4}$  size buffer or by stacking adjacent domain elements into the four channels. Which would be more efficient depends on the memory access patterns of your particular application. The custom method was the one used by Goodnight et al. in their EGSR2003 paper on tone mapping on GPUs. They duplicate the data, thus using additional GPU memory, but do so in a way that the data layout matches their computational needs closely and allows them to avoid "unpacking overhead" at the end of processing.

## Computational Frequency

GPGPU

## 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

GPGPU

## Computational Frequency

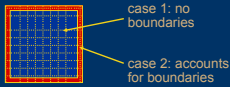
- Branches
  - Avoid these, especially in the inner loop - i.e., the fragment program.

GPGPU

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

- **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



GPGPU

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.

## Computational Frequency

- **Branches**
  - Avoid these, especially in the inner loop - i.e., the fragment program.
  - Mark's talk will give some strategies for branching if you absolutely cannot avoid it.

GPGPU

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

- Precompute
- Precompute
- Precompute

GPGPU

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

- **Precompute texture coordinates**
  - Take advantage of under-utilized hardware
    - vertex processor
    - rasterizer
  - Reduce instruction count at the per-fragment level
  - Avoid lookups being treated as texture indirections

GPGPU

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.)



## Computational Frequency

- Precomputed lookup tables

```
// Calculate Red-Black (odd-even) masks
float2 intpart;
float2 place = floor(1.0f - modf(round(center + 0.5f) / 2.0f,
                               intpart));
float2 mask = float2((1.0f-place.x) * (1.0f-place.y),
                    place.x * place.y);

if (((mask.x + mask.y) && params.y) ||
    !(mask.x + mask.y) && !params.y)
{
    ...
}
```



The other way to use precomputation to your advantage is to build lookup tables and bind them as texture maps. Functions with a constant-size domain and range that are constant across runs of an algorithm – even if they vary in complex ways based on their input – can be precomputed and store in texture maps. Texture maps can be used for storing functions of one, two, or three variables over a finite domain as 1D, 2D, or 3D textures. Textures can store up to four channels, so you can encode as many as four separate functions in the same texture. Texture lookups also provide filtering (interpolation), which you can use to get piecewise-linear approximations to values in between the table entries.

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.

## Computational Frequency

- Precomputed lookup tables

```
half4 mask = f4texRECT(RedBlack, IN.redblack);
/*
 * mask.x and mask.w tell whether IN.center.x and IN.center.y
 * are both odd or both even, respectively. either of these two
 * conditions indicates that the fragment is red. params.x==1
 * selects red; params.y==1 selects black.
 */
if (dot(mask, params.xyxy))
{
    ...
}
```



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



## Profiling and Load Balancing

- Software profiling
- GPU pipeline profiling
- GPU load balancing

GPGPU

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?

## Profiling and Load Balancing

- Run a standard software profiler!
  - Rational Quantify
  - Intel VTune
  - AMD CodeAnalyst

GPGPU

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'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.

## Profiling and Load Balancing

- GPU Pipeline Profiling
  - This is where it gets tricky.
  - Some tools exist to help you:
    - NVPerfHUD  
<http://developer.nvidia.com/docs/10/8343/How-To-Profile.pdf>
    - NVShaderPerf  
[http://developer.nvidia.com/object/nvshaderperf\\_home.html](http://developer.nvidia.com/object/nvshaderperf_home.html)
    - Apple OpenGL Profiler  
[http://developer.apple.com/opengl/profiler\\_image.html](http://developer.apple.com/opengl/profiler_image.html)

GPGPU

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.

## Profiling and Load Balancing

- GPU Load Balancing
  - This is a whole talk in and of itself
    - e.g., <http://developer.nvidia.com/docs/10/8343/Performance-Optimisation.pdf>
  - Be sure to read the NVIDIA GPU Programming Guide
    - [http://developer.nvidia.com/object/gpu\\_programming\\_guide.html](http://developer.nvidia.com/object/gpu_programming_guide.html)
  - Sometimes you can get more hints from third parties than from the vendors themselves
    - [http://www.3dcenter.de/artikel/cinefx/index6\\_e.php](http://www.3dcenter.de/artikel/cinefx/index6_e.php)
    - [http://www.3dcenter.de/artikel/nv40\\_technik/](http://www.3dcenter.de/artikel/nv40_technik/)

GPGPU

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

GPGPU

## Conclusions

- Get used to thinking in terms of parallel 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

GPGPU

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 that mapping, and you'll eek more and more performance out of your system from the most unexpected of places. :)

## Questions?

### • Acknowledgements

- NVIDIA for having given me a job this summer
- Dave Luebke, my advisor
- GPGPU course presenters
- NSF Award #0092793

GPGPU