GPGPU Performance Tuning – An illustrated example

Dominik Göddeke, University of Dortmund, Germany
dominik.goeddeke@mathematik.uni-dortmund.de
www.mathematik.uni-dortmund.de/~goeddeke

This tutorial describes some common techniques to improve performance of GPU-based implementations in linear algebra applications. The example presented here is a Jacobi iteration (commonly used as a smoother in multigrid scenarios) on a sparse matrix arising from Finite Element discretizations of standard operators. However, none of that advanced background is neccessary to understand the GPU-specific examples given here.

Full sources (in OpenGL and Cg) are provided on the web page. Most of this work is based on [BVP] and several discussions at the online forums on [GPGPU].

CPU implementation

We consider a nine-banded sparse matrix **A** that defines a linear system we want to "solve" by Jacobi iteration. The nine bands are stored in separate vectors each of length N, with some zero padding at the beginning or end of the off-diagonal vectors. The bands are labelled **DD**, **DU**, **DL** for center tridiagonal submatrix, **UD**, **UU**, **UL** for the upper tridiagonal and **LD**, **LU**, **LL** for the lower respectively. Given a RHS vector **B** and a solution vector **X**, one step of the Jacobi iteration can be written as

X[i] = **X**[i] + (**B**[i] – (**A*****X**)[i])/**DD**[i], i=1..N

Taking the band structure of the matrix into account, the matrix-vector-multiplication inside this iteration can be rewritten using a single loop over all bands:

Y[i] = **DD**[i]***X**[i] + **DL**[i]***X**[i-1] + **DU**[i]***X**[i+1] +
LD[i]***X**[i-SQRT(N)] + **LL**[i]***X**[i-SQRT(N)-1] +
LU[i]***X**[i-SQRT(N)+1] + **UD**[i]***X**[i+SQRT(N)] +
UL[i]***X**[i+SQRT(N)-1] + **UU**[i]***X**[i+SQRT(N)+1]

Alternatively, each band can be looped over separately with a *DAXPY*-like operation, yielding better performance when linked against the *BLAS* library.

Note that for a given length N, the number of arithmetic operations for the a single step of the Jacobi iteration is 20*N.

Mapping the CPU implementation to the GPU

The first attempt in porting the application to the GPU roughly follows the steps outlined in the PingPong tutorial [PINGPONG]. The iteration vector **X** is stored in a double-buffered pBuffer (using only a single channel), alternately serving as render target and input texture as described in the above tutorial. The right-hand-side and the various bands are stored in different channels of three other square floating point textures with some appropriate zero padding to avoid ugly index shifts. Alternatively, each vector can of course be stored in a single-channeled texture alone, increasing the number of additional textures, but making better use of the texture caches. All vectors are packed into the 2D textures in a row-major manner, each texture is of size SQRT(N) by SQRT(N). A single step of the Jacobi iteration can be coded into a single fragment program running entirely on the GPU. To make sure the program gets all the neccessary data, an orthographic projection with a one to one mapping from viewport to screen is chosen, and a multitextured screen-filling quad is rendered. This ensures that for each entry in our vector, the fragment program is executed exactly once. After this pass, the CPU takes care of swapping the surfaces of the offscreen buffer, the newly calculated values on the formerly write-only surface of the buffer are now bound as a read-only input texture for the next iteration. At no time, any user data is transferred back to the CPU. Instead of implementing a convergence control, a fixed number of iterations is performed.

Because of the mapping from vector to texture, for a given fragment at coordinates *(i,j)*, the following positions need to be accessed in the input textures: Of the nine bands and the right hand side vector, data at location *(i,j)* is read in, and in the texture representing the vector **X**, data from pixel location *(i,j)* and the eight directly surrounding neighbours is required, i.e. from pixel *(i-1,j-1)* to pixel *(i+1,j+1)*. In total, 12 texture lookups (9 into the **X** texture and 3 into the other data textures) are required to calculate a single new value of the iteration vector **X**. The arithmetic intensity of this Jacobi implementation is rather low.

One detail needs additional explanation: When looking up values from the **X** texture at the "boundary layer" of the texture, data has to be wrapped around to the next or previous "row" of texels as a consequence of the vector to texture mapping chosen. In this first version, the respective coordinates are wrapped around using some straight forward algebra, presented here directly as a Cg snippet. Note that *"ocoords"* is the actual screen position as passed to the fragment program through the *WPOS* semantics. The subtraction/addition of 0,5 is implied by the use of *texRECT*-style texture coordinates.

```
float2 coords_right = ocoords-0.5;
float z = (coords_right.y)*texwidth + (coords_right.x+1);
coords_right.y = floor(z/texwidth);
coords_right.x = z-texwidth*coords_right.y;
coords_right += 0.5;
float2 coords_left = ocoords-0.5;
z = (coords_left.y)*texwidth + (coords_left.x-1);
coords_left.y = floor(z/texwidth);
coords_left.x = z-texwidth*coords_left.y;
coords_left += 0.5;
```

Substancial performance improvement by exploiting the SIMD model

The first step in achieving a good GFLOP/s rate is to exploit the SIMD *(single instruction multiple data)* view of the graphics processor. When viewing the GPU as a streaming processor, a computational kernel is executed on each fragment. For the programmer, it is safe to abstractly view this process as if "on each fragment, the same operations are performed simultaneously", we do not have to worry about the number of parallel pipes the card provides, and especially we are not able to influence the parallel execution.

But there is a second level of SIMD nature: Instead of operating on a single floating point value, modern GPUs can perform the same operation on a quadrupel (RGBA) of values at the same time, almost at the same speed.

With respect to the Jacobi iteration discussed here, the initial task to solve one system is extended to solving four systems at the same time. Each of these must be of same size and format (must have the same 9-banded structure as presented above), but the values in the different band vectors and the right hand side may of course be different. This may sound like a "cheat", in practical situations however where systems with the given matrix structure arise, there are usually many systems that need solving. Another possibility would be to map a single vector into a 2D-RGBA structure, but this is tedious and is beyond the scope of this tutorial. **Warning: Different cards behave differently, this trick might even result in a slowdown!** To check this for yourself, two versions of the code are available on my web page, one using only single-channeled textures, and the other one using the idea to "solve four systems simultaneously".

The neccessary changes to the implementation are straight-forward and need not be documented here. Just note that for each fragment (for each quadrupel of new **X** values), a total of 19 texture lookups (9 times into the X texture, nine times into the various band textures and once into the RHS texture) are required, improving the arithmetic intensity slightly.

On an ideal SIMD machine, one would expect a speedup by a factor of four since the systems are completely independent of each other. However, due to the fact that less texels fit in the texture cache (one texel comsumes 128 Bits instead of 32 Bits now), the number of cache hits decreases. In this example, a speedup by a factor of two was measured on certain cards, on other hardware, this resulted in a factor of 2 slowdown.

Another trick is worth an additional 100 MFLOP/s of speedup: Instead of actually dividing by the main diagonal in each fragment, its inverse is precomputed on the CPU, passed as an additional texture to the GPU where a multiplication is performed. Note that this approach does not only yield better performance, but also better accuracy, as pointed out by [PARANOIA].

Multipass rendering to reduce shader complexity

A closer examination of the fragment program and some experiments show the next major bottleneck: For each fragment, a lot of possibly unneccessary algebra is performed, including two rather expensive calls to the Cg library function *floor()*. For the vast majority of the pixels, wrapping to get the correct offsets in the texture representing the vector **X** is not neccessary at all. Additionally, for texels on the left boundary, all texels will wrap to the left and no texels will wrap to the right and vice versa.

To get around this problem, instead of running one generic shader that works correctly for all pixel locations, three rendering passes – each using a specialized shader - are performed to implement a single iteration of the Jacobi algorithm:

In the first pass, a single quad is rendered with coordinates *(1.0, 0.0)*, *(texWidth-1.0, 0.0)*, *(texWidth-1.0, texHeight)* and *(1.0, texHeight)*. This covers all the "interiour texels", in the corresponding fragment program, the neccessary offsets can be directly calculated without any wrapping.

The second pass renders a single line from *(0.0, 0.0)* to *(0.0, texHeight)*. All texels activated by this line will wrap their left neighbours to the next line. By passing the *texWidth* value as uniform parameter to the shader, this can be implemented without any call to the expensive *floor()* library function as well.

The last pass uses a line from *(texWidth-1.0, 0.0)* to *(texWidth-1.0, texHeight)* to perform an analogous operation for the texels on the right boundary.

After these three passes, the surfaces are swapped, and one Jacobi iteration is completed. Note that despite the fact that the CPU overhead (per pass, three programs need to be bound/unbound with their respective parameter handles) is bigger, this multipass approach yields a total speedup of 400 MFLOP/s.

Including the vertex shader

The graphics card provides powerful interpolation units in the fixed-function part of the hardware. With the approach as described before, this feature has been used only sparsely: The texel coordinates as queried with the *WPOS* semantics have been interpolated from the corner coordinates of the screen-sized quad that was rendered. To get additional efficiency, the other texture coordinates (up, down, left, right) can be calculated explicitely in the vertex shader and be interpolated automatically: They depend linearly on the texture coordinate values assigned to the corners of the quads. This is implemented by a small vertex program that calculates the eight offset coordinates for each of the four vertices of our screen-sized quad, and passes them through the pipeline (correctly interpolated) on to the fragment program using the *TEXCOORD0* to *TEXCOORD7* binding semantics.

In total, this trick yields another 300 MFLOP/s: The pipeline is still dominated by the workload of the fragment shader, but the fragment shader only performs the 20*N essential operations of the Jacobi iteration, and no additional coordinate calculations.

Results

The Jacobi solver has been applied to a couple of test cases of various vector lengths. The following table summarizes averaged results for two common cards.

Problem size	MFLOPs 5900U RGBA	MFLOPs 5900U RED	MFLOPs 6800 RGBA	MFLOPs 6800 RED
4096	807	451	283	474
16384	1595	594	669	1233
65536	1925	639	979	1894
262144	1504	643	1083	2238
1048576	1363	632	1002	2358
4194304	1280		984	2240

Note that a problem size of 4096 means N=4096 for the single channeled (RED) layout, and 4 systems each of size N=1024 for the layout labelled RGBA: Indeed the data layout makes a huge difference, and different cards perform differently. It goes beyond the scope of this tutorial to explain why, pragmatic programmers should just run tests with both data layouts (provided independently on my homepage) and pick their layout based on the results of this benchmark.

References

[BVP] Goodnight, Woolley, Lewin, Luebke, Humphreys: A multigrid solver for boundary-value problems using programmable graphics hardware

[GPGPU] General Purpose Computations on Graphics Hardware, http://www.gpgpu.org

[GPUBENCH] http://graphics.stanford.edu/projects/gpubench/

[PARANOIA] Hillesland, Lastra: GPU floating-point paranoia, http://www.cs.unc.edu/~ibr/projects/paranoia/

[PINGPONG] Göddeke: Playing Ping Pong with Render-To-Rexture and Cg, http::www.mathematik.uni-dortmund.de/~goeddeke

[RT] RenderTexture-2.0.3 by Mark J. Harris, http://www.sourceforge.net/projects/gpgpu