Practical SPU Programming in God of War III

Jim Tilander, Vassily Filippov Sony Santa Monica



Outline

Motivation - why use the SPUs?

- Helping the simulation
- Helping the scene
- Helping the rendering
- Q&A





- A typical game today contains three sections that feeds data into the next:
 - Simulation of game, joypad input etc.
 - Scene traversal.
 - Render scene.





- A very typical optimization is to make the game render in a double buffered mode.
- This is possible because the GPU and the CPU can do parallel execution!
- This allows us to render a scene while the next one is prepared.
- Hides the cost of the simulation!



Motivation



- Processors are becoming increasingly parallel.
- Let's apply the same technique to the CPU parts.
- Our total frame time is now only bound only by the max of any of the three components, simulation, scene or render.
- This leads to combined processing of all three components in one frame!





Motivation

- In a parallel system we have several types of computing resources.
- Easy to think about in terms of one main CPU.



Main

CPU

- Easy to think about one main GPU.
- Bound by one of them.



Motivation

- If any of the two parts run too slow, offload tasks onto the helper CPUs.
- Continue doing this until the whole system runs within frame.
- When it runs within frame, we are done!





The PS3

- Helper CPUs consists of 6 SPU general purpose processors.
- Have an affinity towards math operations.
- 256kb memory limitation.



SPU is not a co-processor

- The SPU is not a coprocessor.
- Full general purpose processor.
- Operates independently from the other processors.
- You can lift PPU code straight over by bracketing it with DMA calls.
- They are fast enough to make this strategy work.



The SPU is fast.

- Actually, it's super fast.
- With a little help it can run code at unbelievable speeds.
- Manual optimization can use use the potential 48x speedup of the architecture to the fullest (compiler never comes close).
- Memory is nearby.
- Leaves us to worry less about the actual computation on the SPU.

Can still win with slow SPU CPU Bound

- Reduce total time of the frame.
- Frame limited by max(CPU, GPU).
- Move parts to SPU.
- Even a slow SPU job can be a net win.

CPU

GPU

GPU Bound

CPU

GPU

SPU == PPU

- Keep the code compilable on both platforms with minimal changes.
- Limit the memory behavior on the PPU.
- Swap DMA calls for memcpy on PPU.
- Enable on the fly runtime switch between SPU and PPU version.



Our frame

- Normal processing with only PPU and RSX working.
- Processing is shifted, three frames in flight at the same time.



- Processing is fairly lengthy.
- Does not run within frame.

Frame n Frame n + I Frame n + 2

Simulation

Our frame

- Relies on SPUs to accelerate both RSX and PPU.
- Moving parts of all three systems to the SPU shortens the overall time.
- Now runs within frame.



Frame n Frame n + 1 Frame n + 2

Simulation



The On Screen Profiler

- Both the PPU and SPU profilers are in sync.
- Allows for easy identification of parallel tasks.
- We can verify after the fact that something runs in parallel.





Systems on the SPU

Simulation

- Animation
- Cloth
- Collision
- Procedural textures

Scene

- Culling
- Shadows
- Push buffer generation
- Meta tasks

Render

- Geometry conditioning
- Sound



Offloading the Simulation



Titans

- One of our big ticket things in the game are Titans.
- Large scale creatures that move.
 Essentially moving levels.
- Quickly became apparent collision for the Titans were a bottleneck.
- Starting to move tasks onto the SPU.



Titans

- Bracketed PPU code with DMA calls CPU Processing and recompiled for SPU.
- Single buffered implementation.
- One look at the profiler shows us that no more optimizations are necessary.
- Still tons of performance on the table.



One collision job

Titans

- We provide tech to artists and designers.
- Sometimes they run with it to places we never imagined.
- Moving ropes are "titans" from the engine's point of view.



Cloth simulation

- Kratos has a short loin cloth.
- Enemies has various pieces of cloth.
- Independent jobs, naturally parallel.
- Fire and forget jobs, we can figure out early what we need for calculation and don't need the results until render.

Cloth simulation

- One job per cloth simulation.
- Run this wide (5 SPU).
- Job is dominated by processing.
- Data volume is very low.
- Simply lifting over a PPU version with DMA calls begin/end.



Offloading the Scene traversal



Culling

- Simple frustum checks against bound spheres.
- Traverses the list of all potential models.
- Produces visibility bits.
- Processes both frustum and occlusion checks at the same time.
- Highly suited for the SPU.





- Still got PPU parts, only the heavy lifting is on the SPU.
- Occluder selection, visibility bit processing is still on the PPU.



- Generate pushbuffer commands to set vertex buffers, shader constants and textures.
- Pruning of state redundancy.
- A large gather operation with a large amount of pointer to pointer chasing.
- Can easily swamp the PPU with L2 misses.



- Each SPU fetches a small group of model references (one batch) at a time.
- Double buffer DMA, fetch model B while processing model A.
- Masked memory access cost.



- Adapted the PPU version to handle interleaved DMA.
- The SPU version is also the PPU version!
- In debug mode we can switch to the PPU version on the fly.
- PPU version still useful for handling debug-jobs too large for the memory on the SPU (e.g. very large shaders).



- We run this final generation wide on 5 SPUs.
- Allocate a chunk of memory from the pushbuffer.



- We have 5 SPUs all trying to allocate memory from the same pushbuffer.
- Synchronization done through mfc_getllr and mfc_putllr.
- Bypasses regular DMA, goes through the atomic unit instead.
- Should be your staple synchronization mechanism, fast and no OS overhead.

while(true) {

char line[128];

// Pull in one cache line from lineEA
mfc_getllar(line, lineEA);
mfc_read_atomic_status();

// This does our allocation in main memory
do_allocate_mem(line);

// Uhu, didn't work, try it all over again



Offloading the GPU



- Various techniques to offload the GPU (post processing, vertex processing, software rasterizers).
- We've focused on offloading the cost of the opaque pass.
- Majority of this cost comes from vertex processing and lighting.
- Moved both over to the SPU.



- We pass all our vertices through the SPUs to be preconditioned for the GPU.
- A special purpose job handles various tasks to help the GPU.
- Relies heavily on the SDK library EDGE.



• What is EDGE?

- Geometry processing library available to all PS3 developers.
- Highly optimized SPU code.
- Easy integration, you still control main().
- Can greatly improve your performance!



- One job per drawcall.
- Typical frame holds about 3000 geometry jobs.
- Most of our vertex shader is in here.
- Augmented lighting calculations.
- The one place where we've optimized heavily!



Color correction

- Run as a post effects pass to give a certain (cinematic) look to a scene.
- Basically just do a RGB lookup in a cube map for each pixel on the screen.
- For dynamic effects we want to generate the cube map.
- How do we generate the cube map?

Color correction

- Kick a SPU job early on to generate a cube map based on parametric input.
- Algorithm involves a lot of if statements, harder to do efficiently on the GPU.
- Simple lift of code from PPU.
- Job is dominated by processing, single buffered DMA.



In closing



Go parallel

- You must use the parallel nature of the machine.
- Do not special case the SPU, it is a general purpose processor.
- Offload from the currently bound system.



No premature optimizations!

- Focus on user experience.
- Optimize as needed. Really.

Measure speed

- Be scientific, measure before you jump! The on screen profiler is your first tool.
- Start with a simple implementation that might seem non optimal.
- Always keep the PPU version! Invaluable for debugging.
- Remember that the SPU is faster than you think.

