

Deferred Pixel Shading on the PLAYSTATION®3

Alan Heirich and Louis Bavoil

Abstract— This paper studies a deferred pixel shading algorithm implemented on a Cell/B.E.-based computer entertainment system.

The pixel shader runs on the Synergistic Processing Elements (SPEs) of the Cell/B.E. and works concurrently with the GPU to render images. The system's unified memory architecture allows the Cell/B.E. and GPU to exchange data through shared textures. The SPEs use the Cell/B.E. DMA list capability to gather irregular fine-grained fragments of texture data generated by the GPU. They return resultant shadow textures the same way. The shading computation ran at up to 85 Hz at HDTV 720p resolution on 5 SPEs and generated 30.72 gigaops of performance. This is comparable to the performance of the algorithm running on a state of the art high end GPU. These results indicate that the Cell/B.E. can effectively enhance the throughput of a GPU in this hybrid system by alleviating the pixel shading bottleneck.

Index Terms—Computer Graphics, HDTV, Parallel Algorithms, Rendering

I. INTRODUCTION

The current trend toward multi-core microprocessor architectures has led to performance gains that exceed the predictions of Moore's law. Multiple cores first became prevalent as fragment processors in graphics processing units (GPUs). More recently the CPUs for computer entertainment systems and desktop systems have embraced this trend. In particular the Cell/B.E. processor developed jointly by IBM, Sony and Toshiba contains up to nine processor cores with a high concentration of floating point performance per chip unit area.

We have explored the potential of the Cell/B.E. for accelerating graphical operations in the PLAYSTATION®3 computer entertainment system. This system combines the Cell/B.E. with a state of the art GPU in a unified memory architecture. In this architecture both devices share access to system memory and to graphics memory. As a result they can share data and processing tasks.

We explored moving pixel shader computations from the GPU to the Cell/B.E. to create a hybrid real time rendering system.

Alan Heirich is with the Research and Development department of Sony Computer Entertainment America, Foster City, California. Louis Bavoil is with Sony Computer Entertainment America R&D and the University of Utah, School of Computing, Salt Lake City, UT (e-mail: bavoil@sci.utah.edu).

Our initial results are encouraging and we find benefits from the higher clock rate of the Cell/B.E. and the more flexible programming model. We chose an extreme test case that stresses the memory subsystem and generates a significant amount of DMA waiting. Despite this waiting the algorithm scaled efficiently with speedup of 4.33 on 5 SPEs. This indicates the Cell/B.E. can be effective in speeding up this sort of irregular fine-grained shader. These results would carry over to less extreme shaders that have more regular data access patterns.

The next two sections of this paper introduces the graphical problems we are solving and describe related work. We next describe the architecture of the computer entertainment system under study and performance measurements of the pixel shader. We study the performance of that shader on a test image and compare it to the performance of a high-end state of the art desktop GPU, the NVIDIA GeForce 7800 GTX. Our results show the delivered performance of the Cell/B.E. and GPU were similar even though we were only using a subset of the Cell/B.E. SPEs. We finish with some concluding remarks.

II. PIXEL SHADING ALGORITHMS

We study variations of a *Cone Culled Soft Shadow* algorithm [3]. This algorithm belongs to a class of algorithms known as *shadow mapping* algorithms [15]. We first review the basic algorithm then describe some variations.

A. Soft Shadows

Soft shadows are an integral part of computing global illumination solutions. Equation (1) describes an image with soft shadows in which, for every pixel, the irradiance L arriving at a visible surface point from an area light source is

$$L = \int_{\Omega_{light}} E_{light} \left[\frac{\cos \theta_l \cos \theta_i}{\pi r^2} \right] V d\Omega \quad (1)$$

In this equation Ω_{light} is the surface of the area light and $d\Omega$ is the differential of surface area. E_{light} is the light emissivity per unit area, and θ_l, θ_i are the angles of exitance and incidence of a ray of length r that connects the light to the surface point. V is the geometric visibility along this ray, either one or zero. The distance term $1 / \pi r^2$ reflects the reduction in subtended solid angle that occurs with increasing distance. This expression assumes that the material surface is diffuse (Lambertian).

When $V=1$ and Ω_{light} has area $d\Omega$ this equation describes diffuse local illumination from a point light as is typically computed by GPUs using rasterization. When this equation is expanded recursively in E (by treating each surface point as a source of reflected light) the result is a restriction of the *Rendering Equation* of global illumination [12] to diffuse surfaces.

B. Cone Culled Soft Shadows

Equation (1) is traditionally solved by offline methods like ray tracing. Stochastic ray tracing samples the integrand at various points on Ω and accumulates the result into L . The CCSS algorithm takes an analogous approach, rendering from the light and gathering the radiance from the resulting fragments into pixels.

The CCSS algorithm consists of fragment generation steps and a pixel shading step. We have implemented fragment generation on the GPU and pixel shading on the Cell/B.E. The GPU is programmed in OpenGL-ES using Cg version 1.4 for shaders. Fragments are rendered into OpenGL-ES Framebuffer Object texture attachments using one or more render targets. These textures are then detached from the Framebuffer Objects and used as input to the pixel shading step. The pixel shading step returns a shadow texture which is then bound to the GPU for final rendering.

The algorithm is not physically correct and we accept many approximations for the sake of real time performance. Lights are assumed to be spherical which simplifies the gathering step. Light fragments for each pixel are culled against conical frusta rooted at the pixel centroid. These frusta introduce geometric distortions due to their mismatch with the actual light frustum.

The culling step uses one square root and two divisions per pixel. No acceleration structure is used so the algorithm is fully dynamic and requires no preprocessing. The algorithm produces high quality shadows. It renders self-shadowed objects more robustly than conventional shadow mapping without requiring a depth bias or other parameters.

1) Eye Render

The first fragment generation step captures the locations of pixel centroids in world space. This is done by rendering from the eye view using a simple fragment shader that captures transformed x , y and z for each pixel. We capture z rather than obtaining it from the Z buffer in order to avoid imprecision problems that can produce artifacts. We use the depth buffer in the conventional way for fragment visibility determination.

If this is used as a base renderer (in addition to rendering shadows) then the first step also captures a shaded unshadowed color image. This unshadowed image will later be combined with the shadow texture to produce a shadowed

final image. For some shaders, such as approximate indirect illumination, this step can also capture the surface normal vectors at the pixel location.

2) Light Render

The second fragment generation step captures the locations and alpha values (transparency) of fragments seen from the light. For each light, for each shadow frustum, the scene is rendered using the depth buffer to capture the first visible fragments. The positions and alphas of the fragments are generated by letting the rasterizer interpolate the original vertex attributes. For some shaders, including colored shadows and approximate indirect illumination, this step also captures fragment colors.

3) Pixel Shading

In the third step, performed on the Cell/B.E., light fragments are gathered to pixels for shading. Pixels are represented in an HDTV resolution RGBA texture that holds (x,y,z) and a background flag for each pixel. Light fragments are contained in one (or more) square textures.

Pixel shading proceeds in three steps: gathering the kernel of fragments for culling; culling these fragments against a conical frustum; and finally computing a shadow value from those fragments that survived culling.

4) Fragment Gather

For each pixel, for each light, the pixel location (x,y,z) is projected into the light view (x',y',z') . A kernel of fragments surrounding location $(x',y',0)$ in the light texture is gathered for input to the culling step. Figure 1 illustrates this projection and the surrounding kernel.

It is not necessary to sample every location in the kernel, and performance gains can be realized by subsampling strategies. In our present work we are focused on system throughput and so we use a brute-force computation over the entire kernel.

5) Cone Culling

For each pixel, for each light, a conical frustum is constructed tangent to the spherical light with its apex at the pixel centroid as illustrated in figure 2. The gathered fragments are tested for inclusion in the frustum using an efficient point-in-cone test.

The point-in-cone test performs these computations at each pixel:

$$\begin{aligned} \text{axis} &= \text{light.centroid} - \text{pixel.centroid} \\ \text{alength} &= \text{axis} \cdot \text{axis} \\ 2 \\ \cos^2\theta &= \frac{\text{alength}^2}{(\text{light.radius}^2 + \text{alength}^2)} \end{aligned}$$

na = normalize(axis)

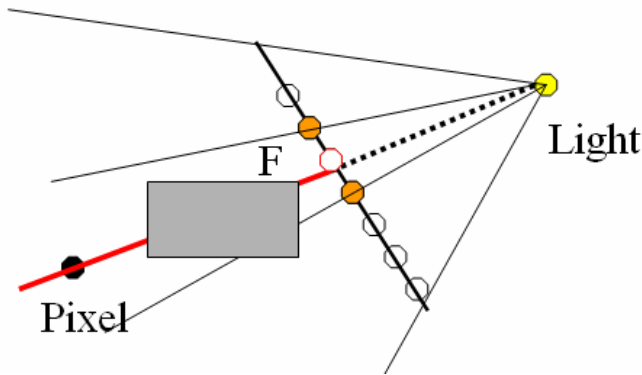


Figure 1 (kernel lookup). The pixel is projected from the world into the light plane, which is equivalent to finding the nearest fragment *F* in the light view to the ray from the pixel to the light center. In this example fragment *F* blocks the ray from the light to the pixel, and we say *F* shadows the pixel.

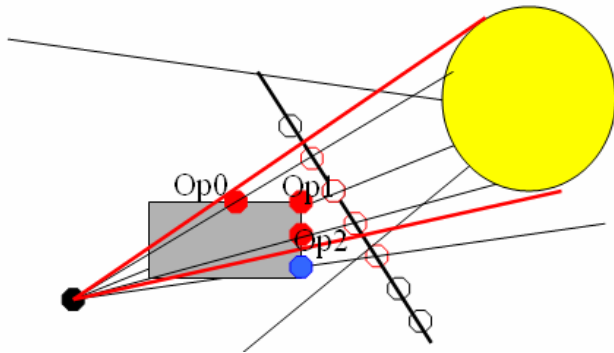


Figure 2 (cone culling). Computing the shadow intensity at a pixel in a cone with the apex at the pixel and tangent to the light sphere. The fragments of the light view are fetched in a kernel centered at the projection of the cone axis over the light plane. Fragments are tested for visibility using an efficient point-in-cone test+.

The point-in-cone then performs these computations for each fragment:

```

fe          = fragment.centroid - pixel.centroid
axisDotFe   = na . fe
direction   = (axisDotFe > 0)
flength2    = fe . fe
inside      = (cos2θ * flength2 ≤ axisDotFe2)
pointInCon e = direction && inside

```

(An expression for $\cos^2\theta$ that more accurately reflects the tangency between the cone and sphere is $(\text{alength}^2 - \text{light.radius}^2) / \text{alength}^2$).

6) Computing new shadow values

The final step is to compute shadow values from the fragments that survived the culling step. Here we describe three such shading computations, and others are possible. We present detailed performance measurements of the monochromatic shader in section 5. We have implemented substantial portions of the other shaders on the Cell/B.E. and GPU to verify proof-of-concept.

a) Monochromatic soft shadows

We can compute monochromatic soft shadows from translucent surfaces by using a generalization of the *Percentage Closer Filtering* algorithm [14]. Among the fragments that survived cone culling we compute the mean alpha (transparency) value. The resulting shadow factor is one minus this mean. At pixels where no fragments survived culling the shadow factor is one. Test images for this shader appear in figure 3.

b) Colored soft shadows

We can obtain colored shadows by including the colors of the translucent fragments and of the light source. In addition to computing the mean alpha value we also compute the mean RGB for the fragments. This requires gathering twice as much fragment data for the shading computation. We multiply these quantities with the light source color to obtain a colored shadow factor. At pixels where no fragments survived culling the shadow factor is one.

c) Approximate indirect illumination

It is worth noting that an approximate indirect illumination component can be computed similarly to Frisvad et. al.'s Direct Radiance Map algorithm [5]. This requires accounting for a transport path from light source to fragment to pixel. This estimate is approximate because it does not account for occluding objects between the fragment and the pixel and also because it only samples a limited kernel of fragments.

Assuming the fragment materials are diffuse (Lambertian), the irradiance at the fragment can be estimated during the light render step proportional to the cosine of the incident angle at the fragment. The subsequent reflected radiance at the pixel is this irradiance times the cosine of the incident angle at the pixel. This radiance can be estimated during the pixel shading step if we have the surface normal at the pixel. This surface normal can be generated during the eye render step.

This computation requires more DMA traffic to accommodate the pixel normals. Since this is not part of the gathered

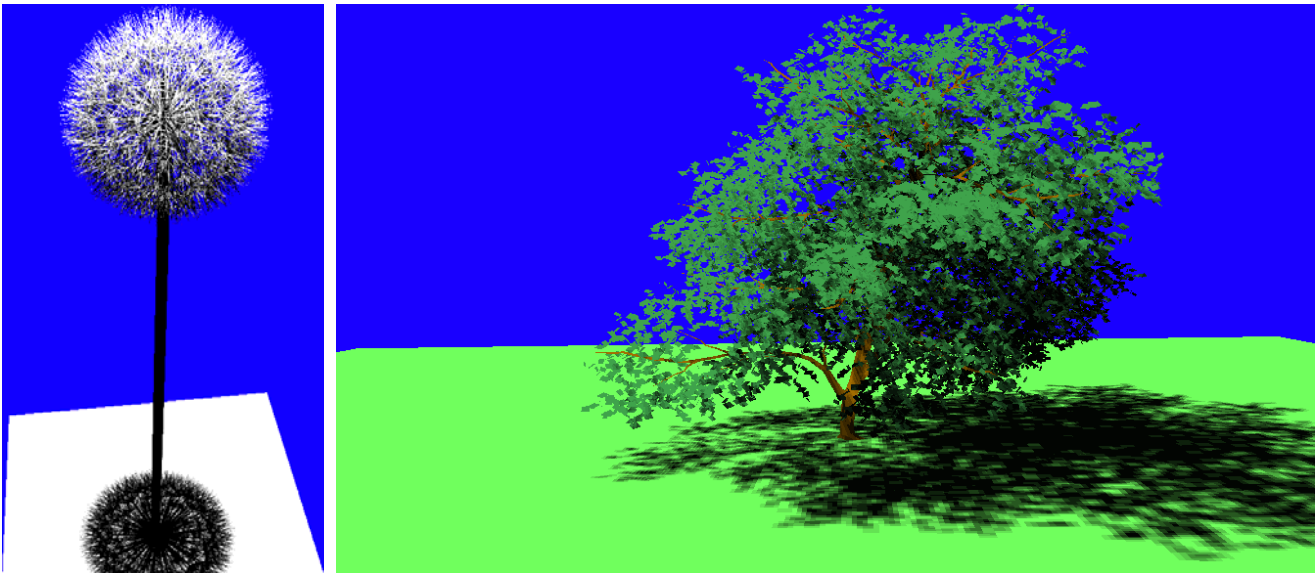


Figure 3: some test images of complex models rendered using the monochromatic shader. (Left) the dandelion is a challenging test for shadow algorithms. The algorithm correctly reproduced the fine detail at the base of the plant as well as the internal self-shadowing within the leaves. (Right) a tree model with over 100,000 polygons rendered above a grass colored surface.

fragment data it can be accommodated efficiently using predetermined transfers of large blocks of data.

III. RELATED WORK

There is an extensive existing literature on shadow algorithms. For a recent survey of real-time soft shadow algorithms see [6]. For a broad review of traditional shadow algorithms see [16].

The most efficient shadow algorithms work in image space to compute the shading for each pixel with respect to a set of point lights. The original image-space algorithm for point lights is *shadow mapping* [15]. In this algorithm the visible surface of each pixel is transformed into the view of the light and then compared against the first visible surface as seen from the light. If the first visible surface lies between the transformed pixel and the light then the transformed pixel is determined to be in shadow.

Traditional shadow mapping produces “hard” shadows that are solid black with jagged edges. They suffer from many artifacts including surface acne (false self-shadowing due to Z imprecision) and aliasing from imprecision in sampling the light view.

The Percentage Closer Filtering algorithm [14] is implemented in current GPUs to reduce jagged shadow edges. This algorithm averages the results of multiple depth tests within a pixel to produce fractional visibility for pixels on shadow boundaries. This has the effect of softening shadow boundaries but since it is a point light algorithm it does not produce the wide penumbrae that characterize shadows from area lights.

Adaptive Shadow Maps [4,13] address the problem of shadow map aliasing by computing the light view at multiple scales of resolution. The multiresolution map is stored in the form of a hierarchical adaptive grid. This approach can be costly because the model must be rendered multiple times from the light view, once for each scale of resolution.

Layered Depth Interval maps [2] combine shadow maps taken from multiple points on the light surface. These are resolved into a single map that represents fractional visibility at multiple depths. In practice four discrete depths were sufficient to produce complex self-shadowing in foliage models. This method produces soft shadows at interactive rates but is costly because it requires multiple renders per light. It does not address translucency.

The irregular Z-buffer [11] has been proposed for hardware realization for real-time rendering. It causes primitives to be rasterized at points specified by a BSP tree rather than on a regular grid. As a result it can eliminate aliasing artifacts due to undersampling. This is similar to Alias-free Shadow Maps [1].

Jensen and Christensen extended photon mapping [10] by prolongating the rays shot from the lights and storing the occluded hit points in a photon map which is typically a *kd*-tree. When rendering a pixel x the algorithm looks up the nearest photons around x and counts the numbers of shadow photons ns and illumination photons ni in the neighborhood. The shadow intensity is then estimated as $V = ni / (ns + ni)$. Our algorithm uses similar concepts to gather fragments and shade pixels, and in addition works with translucent materials.

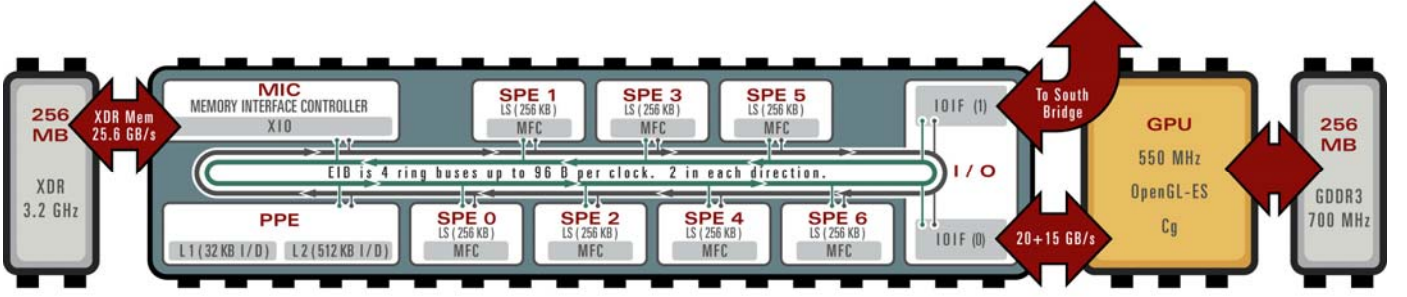


Figure 4: the PLAYSTATION®3 architecture. The 3.2 GHz Cell/B.E. contains a Power Architecture processor (the PPE) and seven Synergistic Processing Elements (SPEs) each consisting of a Synergistic Processing Unit (SPU), 256 KB local store (LS), and a Memory Flow Controller (MFC). These processors are connected to each other and to the memory, GPU and peripherals through a 153.6 GB/s Element Interconnect Bus (EIB). The Cell/B.E. uses Extreme Data Rate (XDR) memory which has a peak bandwidth of 25.6 GB/s. The GPU interface (IOIF) to the EIB provides 20 GB/s in and 15 GB/s out. Memory accesses by the Cell/B.E. to GPU memory pass through the EIB, IOIF and GPU. Access by the GPU to XDR pass through the IOIF, EIB and MIC.

IV. PLAYSTATION®3 SYSTEM

Figure 4 shows a diagram of the PLAYSTATION®3 computer entertainment system and its 3.2 GHz Cell/B.E. multiprocessor CPU. The Cell/B.E. consists of an IBM Power Architecture core called the PPE and seven SPEs. (While the Cell/B.E. architecture specifies eight SPEs our system uses Cell/B.E.s with seven functioning SPEs in order to increase manufacturing yield.) The processors are connected to each other and to system memory through a high speed Element Interconnect Bus (EIB). This bus is also connected to an interface (IOIF) to the GPU and graphics memory. This interface translates memory accesses in both directions, allowing the PPE and SPEs access to graphics memory and providing the GPU with access to system memory. This feature makes the system a unified memory architecture since graphics memory and system memory both are visible to all processors within a single 64-bit address space.

The PPE is a two way in order super-scalar Power Architecture core with a 512 KB level 2 cache. The SPEs are excellent stream processors with a SIMD (single instruction, multiple data) instruction set and with 256 KB local memory each. SIMD instructions operate on 16-byte registers and load from and store to the local memory. The registers may be used as four 32-bit integers or floats, eight halfwords, or sixteen individual bytes. DMA (direct memory access) operations explicitly control data transfer among SPE local memories, the PPE level 2 cache, system memory, and graphics memory. DMA operations can chain up to 2048 individual transfers in size multiples of eight bytes.

The system runs a specialized multitasking operating system. The Cell/B.E. processors are programmed in C++ and C with special extensions for SIMD operations. We used the GNU toolchain g++, gcc and gdb. The GPU is

programmed using the OpenGL-ES graphics API and the Cg shader language.

The Cell/B.E. supports a rich variety of communication and synchronization primitives and programming constructs. Rather than describe these here we refer the interested reader to the publicly available Cell/B.E. documentation [7]-[9].

V. RESULTS

We implemented the CCSS algorithm as described in section 2 using the monochromatic pixel shader described in II.5 and II.6.a. We implemented it in hybrid form on the computer entertainment system using the Cell/B.E. and GPU, and also on a standalone high end GPU for comparison.

On the Cell/B.E. we measured performance in three stages: fragment rendering, shadow generation, and final draw. Times and performance measurements are shown in tables 1 through 4.

Eye render	Light render	1-SPE	5-SPEs	Draw time
10.11	3.29	50.47	11.65	5.6

Table 1: Performance of stages of the algorithm. All times are in milliseconds. The eye and light render stages are performed on the GPU as is the final draw. Pixel shading is performed on the SPEs. We measured the time for pixel shading using from 1 to 5 SPEs. The results showed good parallel speedup. Detailed measurements of pixel shading are given in tables 2 and 3.

A. Cell/B.E. Software Implementation

Eye and light fragments are rendered to OpenGL-ES Framebuffer Object texture attachments. We used 32 bit float RGBA textures for all data. The textures for these attachments may be allocated in linear, swizzled or tiled

formats in either GPU or system memory. We experimented with all combinations of texture format and location in order to find the combination that gave the best performance.

GPU performance is highest rendering to native tiled format in GPU memory. The performance advantage is high enough that it is worth rendering in tiled format and then reformatting the data to linear allocation for processing by the Cell/B.E. In order to minimize the latencies incurred by the SPEs in accessing this data we reformat the data into system memory rather than GPU memory.

The key to running any algorithm on the SPEs is to develop a streaming formulation in which data can be moved through the processor in blocks. We move eye data in scanline order and double buffer the scanline input. While one scanline of pixels is being processed we prefetch the next scanline. As each scanline is completed it is written to the shadow texture. We have measured the DMA waiting for the scanline data and it was negligible.

For every pixel of input we generate a series of DMA transactions to gather the necessary light fragments. The source address for each transaction is a location inside the light fragment buffer. We compute this address by applying a linear transform (matrix multiplication) to the eye data (x, y, z) to obtain a light coordinate (x', y', z') .

These transactions are bundled into long DMA lists. By having multiple DMA lists in flight concurrently we buffer fragment data in order to minimize DMA waiting. We experimented with the number and size of the DMA lists in order to minimize runtime. We found that having four DMA lists was optimal and that larger numbers did not reduce the runtime. We found similarly that fetching 128 pixels per DMA list was optimal and that longer DMA lists did not reduce runtime.

We parallelized the computation across multiple SPEs by distributing scanlines to processors. This is straightforward and provides balanced workloads. We scheduled tasks using an event queue abstraction provided by the operating system that is based on one of the Cell/B.E. synchronization primitives, the mailbox. We measured the cost of this abstraction at less than 100 microseconds per frame. When running in parallel on multiple SPEs the individual processors completed their work within 100 microseconds of each other.

Each SPE computes a set of scanlines for the shadow texture. They deliver their result directly into GPU memory in order to minimize the final render time.

B. Measurements

We validated the correctness of the implementation by rendering a variety of models under different conditions. We then made detailed measurements of performance and scaling of the tree model in figure 3. These measurements

appear in tables 2 and 3. All of our measurements used a single light source. The tree model contains over 100,000 polygons. The performance of the shading computation is independent of the time required to generate the fragments, and thus is independent of the geometric complexity of the model.

	1-SPE	2-SPEs	3-SPEs	4-SPEs	5-SPEs
Full	50.47	28.86	16.78	13.25	11.65
Hz	19	34	59	75	85
Speedup	1	1.75	3.01	3.81	4.33
Scaling	1	0.87	1.00	0.95	0.87
No waiting	41.97	21.05	14.09	10.63	8.56
Speedup	1	1.99	2.98	3.95	4.90
Scaling	1	1.00	0.99	0.99	0.98

Table 2: Parallel performance of the pixel shading computation. All times are in milliseconds. Images were rendered at HDTV 720p resolution (1280x720 pixels). The tree was rendered with data-dependent optimizations disabled in order to obtain worst-case times. The image was rendered using the full algorithm ("full") and with the DMA fragment gather operation disabled ("no waiting"). The computation was exactly the same in both cases, but in the "no waiting" case the shader processed uninitialized fragment data. The speedup and scaling efficiency was evaluated in all cases. These results show that the computation speeds up almost perfectly but that substantial time is lost waiting for the gather operation. Further information about the DMA costs appears in table 3.

	1-SPE	2-SPEs	3-SPEs	4-SPEs	5-SPEs
Wait time	8.50	7.81	2.69	2.62	3.09
% waiting	17	27	16	20	27
DMA GB/s	2.53	4.43	7.62	9.66	10.98
DMA per second	42.47 M	74.27 M	127.73 M	161.76 M	183.97 M

Table 3: DMA costs on different numbers of SPEs. All times are in milliseconds. The algorithm spent considerable time waiting for the results of the DMA fragment gather operation ("wait time"). Expressed as a percentage of the pixel shading computation, the monochromatic shader spent between 17 and 27 percent waiting for fragment DMA. This explains the deviation from ideal scaling in table 2. The Cell/B.E. sustained 10.98 GB/s of DMA traffic using packet sizes that were predominantly 48 bytes in length, and over 183 mega-transactions ($M=1024^2$) per second.

All images were rendered at HDTV 720p resolution, 1280x720 pixels. We used lightmap resolution of 1024x1024 in our experiments and a 3x3 fragment kernel. In order to ensure that we measured worst-case performance we disabled optimizations that skipped background pixels and transparent fragments. We measured performance on one to five SPEs. In our tests the other two SPEs were in use by graphics and operating system services.

C. Data Analysis

Tables 1 and 2 show that the shading calculation can be sped up to meet any realistic performance requirement. The monochromatic shader ran at 85 Hz using 5 SPEs and at 34 Hz using 2 SPEs. Videogames are typically rendered at 30 or 60 frames per second. Shading calculations should generally run at these rates, but for shadow generation it is possible to use lower frame rates without affecting image quality. It would also be possible to use shadows generated at 720p resolution with a base image rendered at a higher 1080p resolution (1920x1080 pixels).

Table 3 analyzes the time spent waiting for DMA transactions to complete. This was as much as 27% of the total time. Note that if we were able to remove all of this DMA waiting the performance on 5 SPEs would reach 116 frames per second as indicated by the "no waiting" data in table 1.

While it is difficult to observe the DMA behavior directly we can reason about the bottlenecks in our computation. Every DMA transaction costs the memory system at least eight cycles of bandwidth no matter how small the transaction. Thus 400 M transactions per second is an upper limit of the system memory performance. The shader generated 183.97 M DMA transactions per second which does not approach the limits of the memory system. Most of these were 48-byte gathers of light view fragments, while the rest were block transfers of entire scanlines 20 KB in size.

We profiled the runtime code to measure the number of SIMD operations that were spent in DMA address calculations. The results appear in table 4. We found that we were spending between 14% and 17% of operations supporting the DMA gather operation.

DMA addressing	Shading	Total	DMA percentage
16,358,400	79,718,400	96,076,800	17

Table 4: Results of run-time profiling. These figures count the number of SIMD instructions executed per frame for both shaders in the inner loop and DMA addressing calculations. It does not include the cost of scalar code that controls the outer loop. The number of operations is four times the number of instructions. The last column shows the percentage of SIMD operations that were spent computing addresses for the DMA gather.

We also measured the time to execute the scalar control logic and perform the DMA for the eye render fragments in order to better estimate the cost of shaders with scanline order data access. These DMA operations are for an entire scanline at a time, 20 K bytes in size. Each frame reads and writes each scanline once for a total of 28.125 megabytes of DMA activity using two transactions. On one SPE this required 2.13 ms of time yielding an effective transfer rate of over 12.89 GB/s. For shaders with scanline order access, it should be possible to read as much as five times as much scanline data without exhausting the overall DMA bandwidth or the number of DMA transactions.

D. Comparison to GeForce 7800 GTX GPU

We implemented the same algorithm on a high end state of the art GPU, the NVIDIA GeForce 7800 GTX running in a Linux workstation. This GPU has 24 fragment shader pipelines running at 430 Mhz and processes 24 fragments in parallel. By comparison the 5 SPEs that we used process 20 pixels in parallel in quad-SIMD form.

The GeForce required 11.1 ms to complete the shading operation. In comparison the Cell/B.E. required 11.65 ms including the DMA waiting time, and would require only 8.56 ms if the DMA waiting were eliminated. The performance of the Cell/B.E. with 5 SPEs was thus comparable to one of the fastest GPUs currently available, even though our implementation spent 27% of its time waiting for DMA. Results would presumably be even better on 7 SPEs, or on fewer SPEs if we could reduce or eliminate the DMA waiting.

VI. REMARKS

We have explored moving pixel shaders from the GPU to the Cell/B.E. processor of the PLAYSTATION®3 computer entertainment system. Our initial results are encouraging as they show it is feasible to attain scalable speedup and high performance even for shaders with irregular fine-grained data access patterns. Removing the computation from the GPU effectively increases the frame rate, or more likely, the geometric complexity of the models that can be rendered in real time.

We can also conclude that the performance of the Cell/B.E. is superior to a current state of the art high end GPU in that we achieved comparable performance despite performance limitations and despite using only part of the available processing power. Our current implementation loses substantial performance due to DMA waiting. This results from the fine-grained irregular access to memory and is specific to the type of shaders we have chosen to implement. We have explored shaders based on shadow mapping [15] which require evaluating GPU fragments generated from multiple viewpoints. These multiple viewpoints are related to each other by a linear viewing transformation. Gathering the data from these multiple viewpoints requires fine-grained irregular memory access.

This represents worst-case behavior for any memory system.

REFERENCES

- [1] Timo Aila and Samuli Laine, "Alias-Free Shadow Maps," in *Proc. Rendering Techniques 2004: 15th Eurographics Workshop on Rendering*, 2004, pp. 161-166.
- [2] Maneesh Agrawala, Ravi Ramamoorthi, Alan Heirich and Laurent Moll, "Efficient Image-Based Methods for Rendering Soft Shadows," in *Proc. ACM SIGGRAPH*, 2000, pp. 375-384.
- [3] Louis Bavoil and Claudio T. Silva, "Real-Time Soft Shadows with Cone Culling," *ACM SIGGRAPH Sketches and Applications*, 2006.
- [4] Randima Fernando, Sebastian Fernandez, Kavita Bala and Donald P. Greenberg, "Adaptive Shadow Maps", in *Proc. ACM SIGGRAPH*, 2001, pp. 387-390.
- [5] J. R. Frisvad and R. R. Frisvad and N. J. Christensen and P. Falster, "Scene independent real-time indirect illumination," in *Proc. Computer Graphics International*, 2005, pp. 185-190.
- [6] Jean-Marc Hasenfratz, Marc Lapiere, Nicolas Holzschuch and Francois Sillion, "A survey of Real-Time Soft Shadows Algorithms," *Computer Graphics Forum*, vol. 22, no. 4, 2003, pp. 753-774.
- [7] IBM, Sony and Toshiba, "Cell Broadband Engine Architecture version 1.0," August 8, 2005.
- [8] IBM, Sony and Toshiba, "SPU Assembly Language Specification version 1.3," October 20, 2005.
- [9] IBM, Sony and Toshiba, "SPU C/C++ Language Extensions version 2.1," October 20, 2005.
- [10] Henrik Wann Jensen and Per H. Christensen, "Efficient Simulation of Light Transport in Scenes with Participating Media Using Photon Maps," in *Proc. ACM SIGGRAPH*, 1998, pp. 311-320.
- [11] Gregory S. Johnson, Juhyun Lee, Christopher A. Burns and William R. Mark, "The irregular Z-buffer: Hardware acceleration for irregular data structures," *ACM Transactions on Graphics*, vol. 24, no. 4, 2005, pp. 1462-1482.
- [12] James T. Kajiya, "The Rendering Equation," in *Proc. ACM SIGGRAPH*, 1986, pp. 143-150.
- [13] Aaron Lefohn, Shubhabrata Sengupta, Joe M. Kniss, Robert Strzodka and John D. Owens, "Dynamic Adaptive Shadow Maps on Graphics Hardware," *ACM SIGGRAPH Conference Abstracts and Applications*, 2005.
- [14] William T. Reeves, David H. Salesin and Robert L. Cook, "Rendering Antialiased Shadows with Depth Maps," in *Proc. ACM SIGGRAPH*, 1987, pp. 283-291.
- [15] Lance Williams, "Casting Curved Shadows on Curved Surfaces," in *Proc. ACM SIGGRAPH*, 1978, pp. 270-274.
- [16] Andrew Woo, Pierre Poulin and Alain Fournier, "A Survey of Shadow Algorithms," *IEEE Computer Graphics & Applications*, vol. 10, no. 6, pp. 13-32.