An Approach on Hardware Design for Computationally Intensive Image Processing Applications based on Light Field Refocusing Algorithm

Jiayuan Meng, Dee A.B. Weikle, Greg Humphreys, Kevin Skadron
Dept. of Computer Science, University of Virginia
{jm6dg, dweikle, humper, skadron}@cs.virginia.edu
UVA Technical Report CS-2007-15

Abstract

This paper describes the performance analysis of the light field refocusing algorithm running on different hardware specifications, including the Intel Pentium 4, SSE2(Streaming SIMD Extensions), GPU, and also Cell Broadband Engine. The hardware chosen has unique features, making it interesting to compare their performance on such an application with each other, and how much advantage or disadvantage each one has over others. By doing so, we attempt to clarify the pros and cons of each hardware design in its capability of handling computational intensive image processing applications. The execution time is used as the main metric.

1 Introduction

Light Field Photography is a new technology which trades spatial resolution for directional resolution. It is able to capture an image which, after computation, reveals the object from different view angles and also at different depth.

With the raw image captured by the camera, the refocusing algorithm [2] computes a new image focused at a user-specified focal depth. For a 4096*4096 light field image with 256*256 spatial resolution and 16*16 directional resolution, computation can barely reach real-time with traditional spatial algorithms [2]. Moreover, the size of the image will grow in the future. In fact, many computationally intensive image processing applications are calling for realtime hardware design.

Most of these image processing applications involve a large amount of memory accesses, and the operations are often repetitive. We chose the light field refocusing algorithm as a typical and booming image processing application. The repetitive behavior denotes a high potential for parallelism. Meanwhile, a variety of hardware has been designed for parallelism, including SSE2, GPU, and multi-core chips. Although not originally designed for image processing applications, they each contribute some inspiration for future designs.

The rest part of this paper is organized as follows: Section 2 gives an overview of the light field refocusing algorithm. Section 3 presents how we implemented the refocusing algorithm on different hardware. Performance is compared and analyzed in Section 4. Conclusions are drawn in Section 5.

2 Refocusing Algorithm of Light Field Photography

2.1 A Basic Algorithm

The concept of light field photography was originally developed by Ren Ng and Marc Levoy [3]. The image captured can be regarded as a 4D light field array, with 2 dimensions for spatial resolution and another 2 for directional resolution. For n light field image of size 4096*4096 captured by a plenoptic camera with a 256*256 microlens array, the spatial resolution is 256*256 and the directional resolution is 16*16. Behind each microlens, a small 16*16 version of the original image is formed, which actually correspond to 16*16 sub–apertures. If we extract those pixels corresponding to the same sub–aperture, then we get a sub-aperture image. In our example, the sub–aperture image is 256*256, and there are 16*16 sub–aperture images. Figure 1 shows a sample light field image [2].

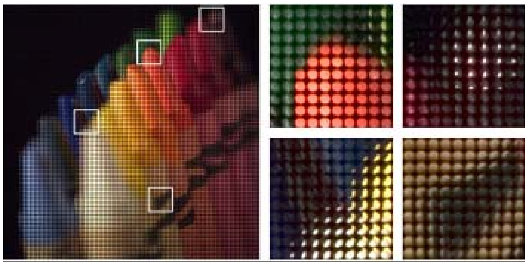


Figure 1: A sample light field image

The basic task of the refocusing algorithm is illustrated in Figure 2. Rays focusing at certain depth will be scattered and captured with different angles and different microlenses, and thus will be mapped onto a set of pixels. The refocusing algorithm computes an inverse mapping and adds up all the pixels corresponding to the same position on the refocused plane.The refocusing algorithm is described in detail in [2].

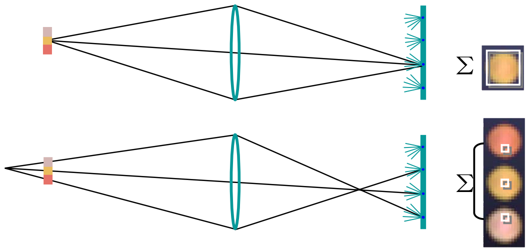


Figure 2: Refocusing Principle: lights focused at certain depth is scattered and captured by different microlenses. All we need to do is gather them and summing up!

This refocusing process is referred to by a **Photography Operator**: For a position(x, y) on the refocused plane, integration is computed for rays traveling from all the positions (u, v) on the main lens plane to (x, y) on the refocused plane. It is illustrated in Figure 3.

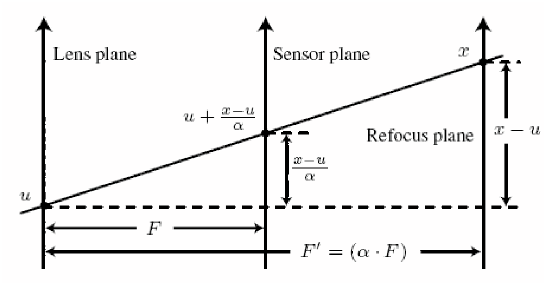


Figure 3: Illustration of the refocusing algorithm

The Photography Operator can be expressed by:

$$P\alpha = \frac{1}{\alpha^2 F^2} \int \int L_F(u(1 - \frac{1}{\alpha}) + \frac{x}{\alpha}, v(1 - \frac{1}{\alpha}) + \frac{y}{\alpha}, u, v) du dv \tag{1}$$

Where L_F is the 4D light field captured by the camera, x, y are defined on the refocused plane, u, v are defined on the main lens' plane. F is main lens' focal length. α is a scaling factor which determines the refocusing depth. α*F is the distance between the refocused plane and the main lens' principle point. A ray is defined with a vector(x,y,u,v) This operator then computes which position on the focal plane that the ray corresponds to, since it's the focal plane where we have a image formed. We then extract color information from the light field image captured at the focal plane.

2.2 A Derived Algorithm

Assume the spatial resolution is $N * N$. After computation with the basic algorithm, the refocused image has the size of $\alpha N * \alpha N$. This is physically correct. Imagine moving the refocus plane in Figure 3 forward and backward, the area which forms image will vary by a factor of α. If we sample the refocused image at the same rate for different refocusing depth, the effective image size will vary according to α. However, in [2], Ren showed that the Nyquist resolution is $N * N$ with exact refocusing and less with inexact refocusing. Thus, we may want to adaptively vary the sampling rate according to α to achieve the best possible resolution.

The new Photography Operator then becomes:

$$P\alpha = \frac{1}{\alpha^2 F^2} \int \int L_F(u(\alpha - 1) + x, v(\alpha - 1) + y, u, v) du dv \tag{2}$$

This Photography Operator generate refocused image with a fixed size of $N * N$.

Note that a sub–aperture image is defined as

$$S_{(u_s, v_s)}(x, y) = L_F(x, y, u_s, v_s) \tag{3}$$

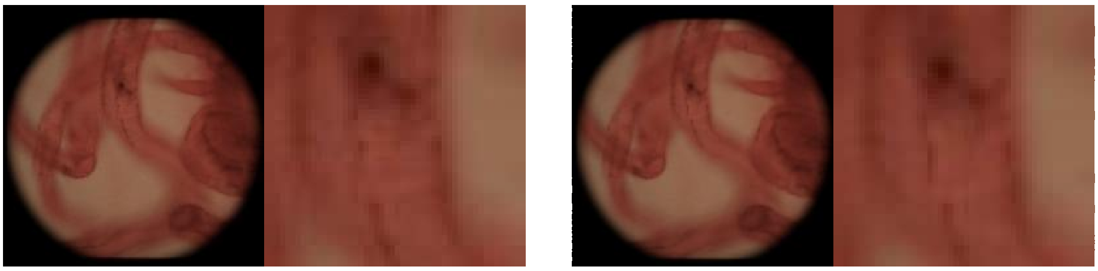
This is very similar to the inner part of Equation 2. The only difference is that in Equation 2, x has an offset of $u(\alpha - 1)$, and y has an offset of $v(\alpha - 1)$. Thus, we can treat the derived refocusing algorithm as the summing of all the sub-aperture images with different offsets according to α.

Intuitively, we can think of sub–aperture images as a group of images taken from slightly different angles. Thus, objects at different depth will be shifted with different amount. The refocusing algorithm computes how much offset each sub-aperture image has for a certain refocusing depth, and then align the sub–aperture images so that objects at the refocusing depth overlap in all the images and appear focused, while objects elsewhere do not overlap and appear blurry.

The derived algorithm also improves the performance of the refocusing algorithm, since the offset is fixed for a sub–aperture image, we don't have to compute them for every element. It also exploit locality in memory access. Instead of fetching data ad-hoc, we can process the sub-aperture images in sequence.

This has a tremendous improvement in performance (see Table 1 in Section 4)Since we are targeting at an optimized algorithm as our base for further optimization on different hardware, the derived algorithm is chosen for further development.

Moreover, we used only nearest point interpolation. Note that the offset is a floating point, bilinear interpolation or more complex interpolation scheme can be applied. Figure 4 compares the image quality between nearest point interpolation and bilinear interpolation. The raw image is from light field microscopy[1] The results doesn't show much difference. Since complex interpolation significantly elongate the execution time(see Table 1 in Section 4), we simply used nearest point interpolation, which just round the floating point to the nearest integer.



(a) Nearest Point Interpolation

(b) Bilinear Point Interpolation

Figure 4: Comparison of Nearest Point Interpolation and Bilinear Point Interpolation

A more detailed comparison of the implementation is shown in Figure 5.

3 Implementation on Different Hardware

We implemented the refocusing algorithm with regular C code, SSE2 instructions, GPU fragment shader, and Cell Broadband Engine. A brief introduction of GPU and CBE can be found in [4] and [5]. The plain C code has been shown as the

Derived Refocusing Algorithm in Section 2, and is run on Intel Pentium 4 as a standard reference. Let's now focus on the other 3 implementations.

3.1 Implementation with SSE2

SSE2 is specialized in vectorized stream data. It has multiple 128 bit registers. In our implementation, data is added column by column. Each line is processed with SSE2 instructions. Since the offset varies for different sub–aperture image, it is not possible to align the data once and for all. Therefore, we use those instructions for unaligned vectors. The code and annotation is shown in Figure 6

3.2 Implementation with Cell Broadband Engine

With CBE [5], we use the master-slave programming model. The refocused image is divided into 8 portion, each portion is computed by an SPU. There are two pieces of codes, one for PPU, which divides the work and synthesize it at last. The main code for refocusing algorithm is on SPU. Since CBE processes data with 128 bit aligned vectors, we have to shuffle the unaligned data. To ease the task, we reorganize each color data as 4 shorts. 3 of them representing RGB channels and the last is used just for padding. A short in SPU is 16 bits long, the same as char16. With this modification, a 128bit vector contains 2 pixels' information. We are using the CBE simulator for testing the performance. We count the cycles and then convert it to time according to the clock rate. Code is provided in Figure 7 and Figure 8

3.3 Implementation with GPU Fragment Shader

We use CG[4] to program the fragment shader on GPU. The performance of GPU depends heavily on the hardware. Since the CG compiler has to unroll the loops, our fragment shader is limited by not only texture size, but also the code size. This limitation depends on specific GPUs. For GPUs with small capacity, we have to do multipass, which increases complexity and has a significant overhead in performance. We will demonstrate the performance in Section 4.

As a sample code, we demonstrate an implementation with 64 passes, in each pass, $2 * 2$ sub-aperture images are processed. Implementation with less passes only requires a few modifications on the global constants.

The initialization prior to the refocusing is straightforward. The program reads in the light field image and extract sub-aperture images as before. The sub–apertures images are then tiled into groups to form textures for GPU processing. In our example, a texture is a square image composed of $2 * 2$ sub–aperture images.

We also used render-to-framebuffer technique to store the result from former passes. Figure 9 describes the code in detail.

4 Performance and Analysis

Table 1 compares the performance of the Basic Algorithm and the Derived Algorithm. We also compare their performance with gcc's sse2 compiler option[-msse2]. In addition, bilinear interpolation is also compared.

Spatial&Directional	*B*	*Bs*	*D*	*D(-msse2)*	*D(bi)*	*D(bi,-msse2)*
(200*200),(20*20)	1.219	1.047	0.266	0.266	3.36	4.422
(256*256),(16*16)	1.187	0.984	0.297	0.281	3.562	4.672

Table 1: Performance Comparison of the Basic Algorithm(B) and Derived Algorithm(D), as well as the Derived Algorithm with bilinear interpolation(bi). Execution time in seconds are used as metrics.

We see that the gcc's [-msse2] compiler option does improve the performance for the Basic Algorithm. However, it hardly improves the performance of the optimized Derived Algorithm. It even produces significant overheads in some cases, e.g. when doing bilinear interpolation.

Indeed, using SSE2 is not always beneficial. SSE2 is based on vectors for stream data. However, many image processing applications are not stream data. Many of the applications require windowing and involves a large amount of computation in locality. Implementing this kind of operation in SSE2 requires additional complex operation. In many case a vector has to be split or permuted before computation. SSE3 and SSE4 have added instructions to add and subtract the multiple values stored within a single register.

Table 2 compares the performance of the algorithm with different hardware specifications described in Section 3. Since GPU can only hold textures with a size of a power of 2, we have to pad the image with spatial resolution (200*200) to (256*256). The directional resolution remain the same.

Spatial&Directional	*SSE2*	$GPU_{64passes}$	$GPU_{16passes}$	$GPU_{4passes}$	*CBE*
(200*200),(20*20)	0.047	0.026	0.026	0.026	0.011
(256*256),(16*16)	0.047	0.013	0.013	0.026	0.00434

Table 2: Performance Comparison of the Basic Algorithm(B) and Derived Algorithm(D), as well as the Derived Algorithm with bilinear interpolation(bi). Execution time in seconds are used as metrics.

Compare the performance of the manually tuned SSE2 instructions with the compiler generated SSE2 instructions in Table 1, we can see that the manually tuned SSE2 instructions can still improve the performance significantly.

Cell Broadband Engine is a multi-core exploitation of parallelism. Similar to SSE2, it is also based on vectors. The 8 SPUs compute data in parallel, so it is even faster than SSE2, and is comparable with GPU. However, with 8 SPUs, the performance is not 8 times that of SSE2. This difference may due to branch miss penalty as well as the Direct Memory Access overhead. CBE tries to alleviate these negative impacts by introducing branch hint and double buffering. To process unaligned data, we have to shuffle the bits in different vectors. However, this mechanism also requires many branches, which introduce a branch penalty which is hard to be alleviated by branch hint.

GPU provides options for a multitude of vector sizes. It is also optimized for texture access in locality. Moreover, it supports interpolation in hardware level, so the GPU implementation is automatically interpolated. However, in general, GPU is not as flexible as CBE in many aspects. First, GPU has a fixed programming model — the graphics pipeline. Parallelism can only be exploited within certain stages, and the resource is not fully utilized for many GPGPU applications. In our example, only fragment processors are utilized. Moreover, in a single pass, memory writes is limited to vertex program, and memory reads is limited to fragment program. Multipass is usually implemented for more complex memory access demands. In addition, GPU performance depends heavily on specific graphics card. Code has to be tuned for capacity of the particular card with respect of texture size and instruction size. Our experiment is based on NVidia GeForce 7600GS model. We have tried to tile the sub-aperture images to textures with sizes of 512*512, 1024*1024 and 2048*2048, and for each size of texture, we need to do 64, 16, and 4 passes. According to Table 2, the performance of 64 passes and 16 passes are identical. However, with a texture of 2048*2048, the performance drops even if only 4 passes are needed. This is due to the overhead of image transferring between the main memory to the memory on GPU.

5 Conclusion and Future Work

With respect to image processing applications, we list the strengths and weaknesses of SSE2, CBE and GPU as follows:

SSE2:

- *Strengths:*
  1. *Vector level parallelism, work best for stream data*
  2. *Simplicity, integrated well into the single chip, thus less memory transfer and branch miss penalty*
- *Weaknesses:*
  1. *No chip-level parallelism*
  2. *Difficult to process small vectors(like those with 32 bits or less)*

CBE:

- Strengths:
  1. Exploited parallelism in both vector level and chip level

2. Enable the user to control memory transfer as well as branch hint, give more opportunity to optimize applications
3. Provide flexible programming model for more complex applications. In addition, user can utilize the resources as much as possible

- Weaknesses:
  1. Vectors are most suitable for stream data, operating smaller vectors requires much more effort. Flexibility has increased complexity.
  2. Although efforts has been made to accelerate memory access, the communication between the PPU and SPUs still accounts for a long time.

GPU:

- Strengths:
  1. Exploit parallelism both in multi-chips and in vectors with 16,24,and 32 bits.
  2. Locality in memory is handled nicely
  3. More basic functions are hardware supported, such as sin/cos, and bilinear interpolation.
  4. accessorial processors are closely integrated into the pipeline, there is less memory transfer.
- Weaknesses:
  1. Pipeline is fixed, resources cannot be fully utilized. Additional efforts such as multi-pass is required when more complex computation is involved.
  2. Limited memory access in different processing units

Thus, we conclude that an ideal hardware model should have the following features:

- *Scalable Parallelism*: Vector size varies from 16 bit to 128 bit or more. Chip-level parallelism also benefits given a good integration
- *Simplicity with Speciality*: With simplicity comes improvement in speed. A simple unit can achieve the best performance for its specialized task. However, simple units alone has limited capability, as described in SSE2 analysis.
- *Flexibility*: Flexible combination of operations provide possibilities for more complex applications. However, with flexibility comes complexity. As for the example of CBE, the SPU provide a variety of functions, however, this also slows down its performance. We often have to trade off simplicity with flexibility.
- *Locality-aware Memory Units*: A special memory units which can tile a texture into blocks for the ease of computation in locality.

In the future, a possible method to achieve simplicity, speciality and flexibility at the same time may be that of a pool of simple but specialized accessorial functional units with programmable connections. The units can be simple vector registers and ALUs as in SSE2, or can be small processors as vertex and fragment processors in GPU. Programmable connection between the units can result in better resource utilization as well as flexible pipeline for complex applications. Small units also alleviate the burden of memory transferring.

6 Appendix: Code Annotation

References

[1] M. Levoy, R. Ng, A. Adams, M. Footer, and M. Horowitz. Light field microscopy. *ACM SIGGRAPH*, 2006.

[2] R. Ng. Fourier slice photography. *ACM SIGGRAPH*, 2005.

[3] R. Ng, M. Levoy, M. Brdif, G. Duval, M. Horowitz, and P. Hanrahan. Light field photography with a hand-held plenoptic camera. *Stanford University Computer Science Tech Report CSTR 2005-02*.

[4] J. Owens and D. Luebke. A survey of general-purpose computation on graphics hardware. 2005.

[5] I. Systems and T. Group. Cell broad band engine programming tutorial. Oct. 2005.

```
float beta = (1.0f-1.0/alpha)*alpha;             coefficient in refocusing equation
int weight = nu*nv;                              value used for nomalizing the color
for(int x=0; x<nx; ++x){                         for each pixel on refocused image
for(int y=0; y<ny; ++y){                         r, g, b: unsigned int for color
                                                 components
   r=g=b=0;
   for(u=0; u<nu; ++u){                          for each sub-aperture
      kux = u*beta;                              kux, kvy:   float temporaries
      for(v=0; v<nv; ++v){
         kvy = v*beta;                           compute where on the focal plane
         kx = (int) (x+kux);                     does the ray hits (equation 2)
         ky = (int) (y+kvy);
         ku = (int) u;                           kx, ky, ku, kv: indices to the
         kv = (int) v;                           4D lightfield
         if(kx<0 || ky<0 || kx>=nx || ky>=ny)    make sure it is in a valid range
               continue;
         lf_pixel=GetColor(&lightfield, kx, ky, ku, kv);   get color from lightfield's 4D array
         r += (unsigned int) lf_pixel[0];        lf_pixel: a pointer to unsigned char
         g += (unsigned int) lf_pixel[1];
         b += (unsigned int) lf_pixel[2];        add them up
      }
   }
   r = (float)r/weight;                          normalize the color
   g = (float)g/weight;
   b = (float)b/weight;
   CLAMP(r);                                     clamp to 0~255
   CLAMP(g);
   CLAMP(b);
   pixel = GetPixel(&refoc_img, x, y);           assign the color to the pixel in the
   pixel[0] = (unsigned char) r;                 refocused image
   pixel[1] = (unsigned char) g;
   pixel[2] = (unsigned char) b;
}
}
```

(a) Code of Basic Refocusing Algorithm

```
beta = (1.0f-1.0/alpha)*alpha;                   Coefficient in refocusing equation
for(u=0; u<nu; ++u){                             For each sub-aperture image
for(v=0;v<nv;++v){
   offset_x = ((float)u)*beta;                   Compute the offsets
   offset_y = ((float)v)*beta;                   Compute where to start and end for each
   beginx = (offset_x<0)?-offset_x : 0;          line and row
   endx = (offset_x<0)?nx : nx-offset_x;
   beginy   = (offset_y<0)?-offset_y : 0;
   endy= (offset_y<0)?ny : ny-offset_y;          For each pixel on the sub-aperture image
   for(x=beginx; x<endx; ++x){                   sub_images: unsigned char*. the merged
      src = &sub_images[3*(beginy+offset_y+ \    sub-aperture images
            ny*(x+offset_x+nx*(v+nv*u)))   ];    src: unsigned char*
      dst = &refocus_dst[3*(beginy+ny*x)];       dst: float*
      for(y=beginy; y<endy; ++y){
         dst[0]+=(float)src[0];                  Add their value to the corresponding
         dst[1]+=(float)src[1];                  position on the refocused image
         dst[2]+=(float)src[2];
         dst+=3;
         src+=3;
      }
   }
}
}                                                Since each pixel on the refocused image
                                                 is added multiple times,
JSAMPLE* pixel, * lf_pixel;                      w is used to normalize the value
float w = F/nu/nv;                               JSAMPLE: unsigned char
for(x=0; x<nx; ++x){
for(y=0; y<ny; ++y){
   dst = &refocus_dst[3*(y+ny*x)];
   src = GetPixel(&refoc_img, x, y);
   src[0] = (JSAMPLE)CLAMP(dst[0]*w);
   src[1] = (JSAMPLE)CLAMP(dst[1]*w);
   src[2] = (JSAMPLE)CLAMP(dst[2]*w);            Clamp the value between 0 and 255
}
}
```

(b) Code of Derived Refocusing Algorithm

Figure 5: Comparison of the Basic Refocusing Algorithm and the Derived Refocusing Algorithm

```
int offset_x, offset_y;
char16* dst;
char16* src;
char16* sub_image;
int beginx, endx, beginy, endy, startrow, endline;
int step_row = ny*3;
```

Color components are defined to be 16 bits, which is enough for computing colors based on RGB values in [0,255], and can utilize the vector instruction as much as possible

```
for(int u=0; u<nu; ++u){
for(int v=0;v<nv;++v){
    sub_image = \
            &sub_images[3*(ny*nx*(v+nv*u))];
```

For each sub-aperture image

sub_images is a 1D array representing the sub-aperture images extracted from the 4D lightfield with a dimension of (nu, nv, nx, ny). Each element in the array is a triple char16 consisting RGB values. To index data at position (x,y) in sub-aperture image(u, v), the indexing method is sub_images[3*(y+ny*(x+nx*(v+nv*u)))]

```
    offset_x = ((float)u)*beta;
    offset_y = ((float)v)*beta;
    beginx = (offset_x<0)?-offset_x : 0;
    endx = (offset_x<0)?nx : nx-offset_x;
    beginy    = (offset_y<0)?-offset_y : 0;
    endy = (offset_y<0)?ny : ny-offset_y;
    startrow= (offset_y<0)?0 : offset_y;
    endline = (endy-beginy)*3;
    src = &sub_image \
          [3*(startrow+ny*(beginx+offset_x))];
    dst = \
        &refocus_dst[3*(beginy+ny*beginx)];
```

Compute the offsets, as well as where to begin and end for each line and row

```
    for(x=beginx; x<endx; ++x){
        int i;
        int divider=8;
    asm(".intel_syntax noprefix\n");
    asm(" mov ecx, %0\n": : "r"(divider) : "%ecx");
    asm(" mov edx, 0\n" : : : "%edx", "%ecx");
    asm(" mov eax, %0\n": :"r"(endline) : "%eax",\
          "%ecx", "%edx");
    asm(" div ecx\n");
    asm(" mov ecx, eax\n");
    asm(" mov ebx, %0\n": : "r"(dst) : "%eax", \
          "%ebx", "%ecx", "%edx");
    asm(" mov eax, %0\n": : "r"(src) : "%eax",\
        "%ebx", "%ecx", "%edx");
```

Process each column with SSE2 instructions

SSE2 instruction can processing 8 bytes at a time

Finally, ecx stores how many times we can use SSE2 per line, since the bytes in a line isn't necessarily divided by 8, there are some leftovers and we need to do it in plain assembly.

```
    asm(" loop:\n");
    asm(" movdqu xmm0, [eax]\n");
    asm(" movdqu xmm1, [ebx]\n");
    asm(" paddd xmm0, xmm1\n");
    asm(" movdqu [ebx], xmm0\n");
    asm(" add eax, 16\n");
    asm(" add ebx, 16\n");
    asm(" sub ecx, 1\n");
    asm(" jnz loop\n");
    asm(" mov ecx, edx\n");
    asm(" shr ecx\n");
```

Start SSE2 processing
Load destination address
Load source address
Adding
Store value to destination adress

```
    asm("leftover:\n");
    asm(" jz endloop\n");
    asm(" mov edx, [ebx]\n");
    asm(" add edx, [eax]\n");
    asm(" mov [ebx], edx\n");
    asm(" add eax, 4\n");
    asm(" add ebx, 4\n");
    asm(" sub ecx, 1\n");
    asm(" jmp leftover\n");
    asm("endloop:\n");
    src += step_row;
    dst += step_row;
    }
}
}
```

Processing the leftover with plain assembly code

```
JSAMPLE* pixel, * lf_pixel;
RawImage refoc_img;
allocImg(&refoc_img, nx, ny, 3);
float w = F/nu/nv;
JSAMPLE* pdst;
for(x=0; x<nx; ++x){
for(y=0; y<ny; ++y){
    src = &refocus_dst[3*(y+ny*x)];
    pdst = GetPixel(&refoc_img, x, y);
    pdst[0] = (JSAMPLE)CLAMP(src[0]*w);
    pdst[1] = (JSAMPLE)CLAMP(src[1]*w);
    pdst[2] = (JSAMPLE)CLAMP(src[2]*w);
}
}
```

JSAMPLE: unsigned char
Normalize the image with the number of additions

Figure 6: SSE2 Implementation of the Derived Algorithm

```
// PPU code

float beta = (1.0f-1.0/alpha)*alpha;
for (i = 0; i < 8; i++) {
   cb[i].u_resolution    = nu;
   cb[i].v_resolution    = nv;
   cb[i].addrDst      = (unsigned int) refocus_dst;
   cb[i].width          = nx;
   cb[i].height          = ny;
   cb[i].portion_width = width_spe;
   cb[i].addrSrc      = (unsigned int) sub_images;
   cb[i].startx        = width_spe*i;
   cb[i].beta            = beta;
}
cb[7].portion_width = nx-width_spe*7;

for (i = 0; i < 8; i++) {
   speids[i] = spe_create_thread ( \
                  gid, &refocus_spu, \
                  (unsigned long long *) &cb[i],\
                  0, -1, 0);
   if (speids[i] == NULL) {
      fprintf (stderr, "FAILED: \
            spe_create_thread(num=%d, \
            errno=%d)\n", i, errno);
      exit (3+i);
   }
}

for (i=0; i<8; ++i) spe_wait(speids[i], &status[i], 0);
__asm__ __volatile__ ("sync" : : : "memory");
```

On the PPU, data is initialized and control blocks (cb) with parameters are transferred to each SPU. The refocused image is divided evenly across the eight SPUs.

u_resolution: directional resolution

v_resolution: directional resolutiooin

addrDst: address of the refocused image

width: width of the refocused image

height: height of the refocused image

portion_width: columns to be processed by each spu

addrSrc: address of the sub-aperture images

startx: the address of the column in refocused image to start refocusing

beta: coefficient in refocusing equation

Initialize the identical thread on each SPU for different part of data. This is a typical SIMD programming model.

Wait for all the spu threads to finish refocusing

```
// SPU code
for(u=0; u<nu; ++u){
for(v=0;v<nv;++v){
   offset_x = ((float)u)*beta;
   offset_y = ((float)v)*beta;
   src_startx = offset_x+start_x;
   src_endx = src_startx+nx-1;
   src_starty = offset_y;
   src_endy = src_starty+ny-1;
   dst_startx = 0;
   dst_endx     = nx-1;
   dst_starty = 0;
   dst_endy     = ny-1;
   if(src_startx<0) {
         dst_startx = -src_startx;
         src_startx = 0;
         if(dst_startx >= nx)
             continue;
   }else if(src_endx >= original_width){
         dst_endx -= src_endx-original_width+1;
         src_endx = original_width-1;
         if(dst_endx<0)
             continue;
   }
   if(src_starty<0) {
         dst_starty = -src_starty;
         src_starty = 0;
         if(dst_startx >= ny)
             continue;
   }else if(src_endy >= ny){
         dst_endy -= src_endy-ny+1;
         src_endy = ny-1;
         if(dst_endx < 0)
             continue;
   }
   stridey = (dst_endy-dst_starty+1)*8;
   stridex = dst_endx-dst_startx;
   x=0;
   mfc_get(((unsigned int)data[0])+16, \
         (unsigned int) sub_images+ \
         8*(ny*(src_startx+x+original_width*(v+nv*u))),\
         linestride, 20, 0, 0);
```

Here we are implementing Direct Memory Access with double buffering

Compute the start and end offset, as always

Here, source is the data in the sub-aperture image, destination is the portion of the refocused image The computation is processed column by column in the refocused image.

```
   mfc_get(dst[0], (unsigned int)refocus_dst+ \
         8*(ny*(dst_startx+x)), linestride, 30, 0, 0);
   for (x=1; x<=stridex; ++x) {
         mfc_get(((unsigned int)data[x&1])+16, \
               (unsigned int) sub_images+ \
               8*(ny*(src_startx+x+ \
               original_width*(v+nv*u))), linestride, \
               20+(x&1), 0, 0);
         mfc_write_tag_mask(1<<(21-(x&1)));
         mfc_read_tag_status_all();
         mfc_get(dst[x&1], (unsigned int) refocus_dst+\
            8*(ny*(dst_startx+x)), linestride, \
            30+(x&1), 0, 0);
         mfc_write_tag_mask(1<<(31-(x&1)));
         mfc_read_tag_status_all();
         add_data_SIMD_misaligned(dst[(x-1)&1], \
                     data[(x-1)&1], src_starty, \
                     dst_starty, stridey);
         mfc_put(dst[(x-1)&1], \
                     (unsigned int)refocus_dst+\
                     8*(ny*(dst_startx+(x-1))), \
                     linestride, 31-(x&1), 0, 0);
      }
      mfc_write_tag_mask(1<<(21-(x&1)));
      mfc_read_tag_status_all();
      mfc_write_tag_mask(1<<(31-(x&1)));
      mfc_read_tag_status_all();
      add_data_SIMD_misaligned(dst[(x-1)&1], \
         data[(x-1)&1],src_starty, dst_starty, stridey);
      mfc_put(dst[(x-1)&1], (unsigned int)refocus_dst+ \
         8*(ny*(dst_startx+(x-1))), \
         linestride, 31-(x&1), 0, 0);
      mfc_write_tag_mask(1<<(31-(x&1)));
      mfc_read_tag_status_all();
   }
}
```

Start double buffering

Fetch the first block of data from both the source and destination

Fetching the next block of data while doing computation on the current block of data

Process the current data. The function is described below

Save to the destination address

Process the last block of data without fetching more blocks

Save the result from the last block

```
void add_data_SIMD_misaligned
(unsigned int* pdst, unsigned int* psrc, int src_start, int dst_start, int stride)
{   // add 'stride' bytes from src[src_start] to dst[dst_start]
    vec_ushort8 *vdst, *vsrc;
    int i,loop;
    vec_ushort8 temp;
    unsigned int dst = (unsigned int) pdst;
    unsigned int src = (unsigned int) psrc;
    if(stride%16){
            // stride is odd
            loop = (int)(stride/16)+1;
            vec_uchar16 mask =
            (vec_uchar16)(0x08,0x09,0x0a,0x0b,\
            0x0c,0x0d,0x0e,0x0f,\
            0x10,0x11,0x12,0x13,\
            0x14,0x15,0x16,0x17);
            if(src_start%2){
                    // src_start is odd, dst_start is even
                    vsrc = (vec_ushort8*)(src+src_start*8+8);
                    vdst = (vec_ushort8*)(dst+dst_start*8);
            }else{
                    // src_start is even, dst_start is odd
                    vsrc = (vec_ushort8*)(src+src_start*8);
                    vdst = (vec_ushort8*)(dst+dst_start*8-8);
            }
            for(i=0; i<loop;++i){
                    temp = spu_shuffle(vsrc[i],vsrc[i+1],mask);
                    vdst[i]=spu_add(vdst[i],temp);
            }
    }else{
            loop = (int)(stride/16);
            vsrc = (vec_ushort8*)(src+src_start*8+16);
            vdst = (vec_ushort8*)(dst+dst_start*8);
            for(i=0; i<loop; ++i)
                vdst[i]=spu_add(vdst[i],vsrc[i]);
    }
}
```

Here we do the main computation
pdst: destination address in refocused image
psrc: source address in sub-aperture image
src_start: the place in psrc to start for each column
dst_start: the place in pdst to start for each column
stride: number of bytes to be processed per column
SPU is vector based. It can process 16 bytes as a vector
Care must be taken for data that are not aligned or not a multiple of 16 chars

mask: a mask to shuffle the unaligned data. Using the mask, spu_shuffle produces a permuted new vector from two input vectors

In this example, a vector contains exactly 2 pixels's color information, therefore, at most one of the source or destination array is not aligned
If source is not aligned
Adjust the beginning of the source and destination address for shuffling

If destination is not aligned
Adjust the beginning of the source and destination address for shuffling

Shuffle the unaligned data and add up

if everything is aligned, it is straightforward to compute the sum

Figure 8: continued: SPU core function of the CBE Implementation

```
void Refocus(){
int canvas_id, fbo_id ,t=0;
for(int u=0; u<UV_RES; ++u){
        for(int v=0;v<UV_RES;++v){
                g_offsets[t++] = ((float)u)*beta;
                g_offsets[t++] = ((float)v)*beta;
        }
}
```

Compute the offset for each sub-aperture image

```
glClearColor(1.f,1.f,0.f,1);
cgGLBindProgram( vertexProgram );
cgGLEnableProfile( vertexProfile );
cgGLSetStateMatrixParameter(g_CGparam_ModelViewProj,
                CG_GL_MODELVIEW_PROJECTION_MATRIX,
                                CG_GL_MATRIX_IDENTITY);
cgGLBindProgram( fragmentProgram );
cgGLEnableProfile( fragmentProfile );
```

Preparation for multi-pass rendering. According to GPU capacity, a maximum number of sub-aperture images are tiled into one texture, In each pass, pixels are extracted from the texture and added with the canvas. The canvas is a texture associated with frame buffer, which stores aggregating results during the multi-pass. We use 2 frame buffer textures to achieve this.

```
cgGLSetParameter2f(g_CGfrag_sub_size, RENDERBUFFER_WIDTH,\
                        RENDERBUFFER_HEIGHT);
cgGLSetParameter1f(g_CGfrag_total_subs, UV_RES*UV_RES);
```

Bind CG programs and pass parameters to it.

```
glViewport(0,0,RENDERBUFFER_WIDTH, \
                RENDERBUFFER_HEIGHT);
glMatrixMode( GL_MODELVIEW );
glLoadIdentity();
glTranslatef( 0.0f, 0.0f, -500.0f );
```

Set Viewport to be the same size of the refocused image

```
glScalef(RENDERBUFFER_WIDTH/2,
RENDERBUFFER_HEIGHT/2,1.f);
canvas_id=0;
glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, g_frameBuffer );
glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT,
                                GL_COLOR_ATTACHMENT0_EXT,
                                GL_TEXTURE_RECTANGLE_EXT,
                                g_fbo_texid[1], 0 );
```

We essentially render a square with the refocused texture on it. To do that, we need some transformation so that the square fits well within the viewport

```
glFramebufferRenderbufferEXT( GL_FRAMEBUFFER_EXT,
                                GL_DEPTH_ATTACHMENT_EXT,
                                GL_RENDERBUFFER_EXT,
                                g_depthRenderBuffer );
glClearColor(0,0,0,1);
glClear(GL_COLOR_BUFFER_BIT);
```

Initialize the frame buffer and clear it with 0 (black color)

```
glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, 0 );
for(int pass=0; pass<g_tex_number;pass++){
        fbo_id = canvas_id%2;
        canvas_id++;
        canvas_id %=2;
        glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, \
                                g_frameBuffer );
        glFramebufferTexture2DEXT( GL_FRAMEBUFFER_EXT, \
                                GL_COLOR_ATTACHMENT0_EXT, \
                                GL_TEXTURE_RECTANGLE_EXT, \
                                g_fbo_texid[fbo_id], 0 );
        glFramebufferRenderbufferEXT( GL_FRAMEBUFFER_EXT,
                                GL_DEPTH_ATTACHMENT_EXT, \
                                GL_RENDERBUFFER_EXT, \
                                g_depthRenderBuffer );
```

Start the multipass

The result of the current pass is stored in the frame buffer texture, in the next pass, this texture serves as a canvas, then it is added with lightfield data stored in more sub-aperture images, the result is stored in the other frame buffer texture, which becomes the canvas in the next pass.

```
        cgGLSetTextureParameter(g_CGfrag_canvas, \
                                g_fbo_texid[canvas_id]);
        cgGLEnableTextureParameter(g_CGfrag_canvas);
        cgGLSetTextureParameter(g_CGfrag_lf_tex, g_tile_texids[pass]);
        cgGLEnableTextureParameter(g_CGfrag_lf_tex);
        cgGLSetParameterArray2f(g_CGfrag_offsets, 0, SHADER_LOOP,
                                &g_offsets[pass*SHADER_LOOP*2]);
```

Bind CG program (the fragment shader)

Pass the parameters

```
        CheckCgError();
        drawImage();
        glBindFramebufferEXT( GL_FRAMEBUFFER_EXT, 0 );
}
cgGLDisableTextureParameter(g_CGfrag_canvas);
cgGLDisableTextureParameter(g_CGfrag_lf_tex);
cgGLDisableProfile( vertexProfile );
cgGLDisableProfile( fragmentProfile );
}
```

Draw the square with the CG fragment shader

```
// Fragment Shader
void main(in float2 canvas_coord: TEXCOORD0,
   uniform samplerRECT canvas_tex: TEXTUNIT0,
   uniform samplerRECT lf_tex: TEXTUNIT1,
   uniform float2 offsets[4],
   uniform float2 sub_size,
   uniform float total_subs,
              out float3 OUT : COLOR)
{
float3 temp=float3(0,0,0);
float u,v;
float2 coord;
float n=0;
```

A CG fragment shader implements the refocusing algorithm with 2*2 sub-aperture images

canvas_coord: rasterized coordination of canvas

canvas_tex: The texture for recorded refocusing process by far

lf_tex: The texture for the new set of sub-aperture images

offsets: The offsets of the 4 sub-aperture images

sub_size: The size of each sub-aperture image

total_subs: Number of all the sub-aperture images, used for normalization

```
for(u=0; u<2; ++u){
for(v=0; v<2; ++v){
coord = float2(v*sub_size.y,    \
u*sub_size.x)+canvas_coord+float2(offsets[n].y, offsets[n].x);
temp += texRECT(lf_tex, coord);
n++;
}
}
```

Compute the coordinates for each pixel and add them up

```
OUT = temp/total_subs+texRECT(canvas_tex, canvas_coord);
}
```

Normalize the color data and add it to the canvas color

Figure 9: CG fragment shader for the Derived Algorithm