특집

GPU Computing on Handheld Devices

Nitin Singhal · 조성대 (삼성전자)

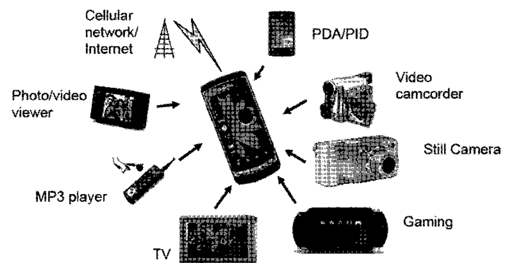
I. 서 론

What would you call a device that has a screen, a keyboard, storage for personal information such as contacts, email, documents, the ability to play audio and video files, games, a spreadsheet program, and a communications capability? Sound like a personal computer? How about a "mobile phone"?

Over the last 30 years, we have seen the power of microprocessor double about every 18 months. An equally rapid increase applied to some other technological parameters such as storage capacity and communication bandwidth. This continuing trend means that computers have become considerably smaller, cheaper, and more abundant. In fact, they are becoming ubiquitous, and are even finding their way into everyday objects.

Mobile phones are forerunner in this technological field. The mobile phone has evolved into a fully functional computer equipped with technology that can be used not only as a phone but also as a music and video player, calender, TV, radio, and still image and video camera, as well as for gaming and surfing the Internet. See <Figure 1>. for an example of mobile ubiquitous environment.

Past few years have seen a tremendous growth in mobile phone users. As of 2007, the number of mobile phone subscribers around



〈Figure 1〉 Mobile phone – multimedia convergence

the world topped 3.3 billion or about 50% of the world's population [1]. Initially used only as telecommunication devices, mobile phones have transformed into information devices. For example, Smartphones (a mobile phone with PC-like functions, equipped with a high performance processor and an open-type operating system that it is used as an Internet, multimedia, and business tool), featuring information device functions such as Internet and multimedia, have been in the spotlight since 2007. In the past, mobile phones small screen hindered the processing of information. However, with the dramatic improvement in display technologies and the arrival of convenient user interfaces (UI) such as touchscreens and 3D widgets, mobile phone usage as an information device has been increasing significantly. Consequently, mobile phones have the potential to deliver graphics to the masses. In addition, the imaging technology has changed significantly over the past few years, and today camera phones are common (and increasing) with 3-5 Mpixel resolution. Modern camera phones are equipped with host of features such as auto focus, panorama, face recognition, stabilizer, etc. In the future, mobile phones are expected to become a daily necessity, combining bio and environmental sensor technologies that can be used for health and personal safety. For example, mobile phones could very well facilitate health-care functions, where exercising, food consumption levels and body temperature can be accurately monitored. The launch of Apple's iPhone [2] has marked a revolution in this mobile application world. Modern mobile phones are competing to provide host of sophisticated applications (graphics and non-graphics) and high-tech games.

Integration of these additional features in mobile phones have some limitations. Compare to the desktop PC, mobile phones are limited by (i) power supply; (ii) computational power; (iii) physical display size; (iv) input modalities. The fundamental problem is that they are battery operated. Whereas many other aspects of computing follow Moore' s law, battery technology develops much more slowly [3]. The display is one of the largest consumers of power, and graphics applications keep the display, often with a backlight, constantly on. Innovation is required at the hardware level for lower power consumption, while diligence is required at the software level for power-aware mobile applications. The devices are small; even if more power were available, that power would turn into heat, which can damage circuits unless the design process considered the thermal aspects early on. Mobile device CPUs also have limited computing power. A related limitation is internal bandwidth for memory accesses, which increases more slowly than

raw computing power and consumes much power. Another limitation is cost: mass-market consumer devices should be cheap, which limits the silicon budget. For example, only the most recent high-end phones support floating-point units. Low computational power of mobile phone CPUs along with other related limitations impedes the integration of additional features in mobile phones.

Specialized hardware known as graphics hardware are introduced as a possible solution to above limitations. On the desktop PC environment, graphics hardware now incorporates a specialized processor called the Graphics Processing Unit, or "GPU". The GPU is a highly parallel architecture for performing the operations of computer graphics fast. As graphics algorithms became more sophisticated, there was a need to develop more flexible hardware and programming environments which led to the development of user-programmable hardware. The new flexibility combined with floating point capability and specialized performance over contemporaneous CPUs, led to work in General Purpose GPU (GPGPU) processing: using the graphics hardware to perform computations for tasks other than graphics [4]. This new approach to GPU programming for non-graphics tasks has been termed GPU Computing. Most mobile phones do not yet have a programmable GPU, but we are in middle of a major change in which more and more phones are being equipped with them. PowerVR SGX [5] and NVIDIA Tegra [6] are few examples of programmable graphics processing units for mobile devices. Samsung Omnia HD i8910 is the first mobile phone equipped with programmable GPU (PowerVR SGX 530) [7].

This article investigates the GPU Computing approach on handheld especially, mobile devices. Section 2 summarize the motivation and essential developments in the hardware and software behind GPGPU. Section 3 describe an overview of GPU hardware architecture. Section 5 serves as an introduction to GPU programming model. Section 6 investigates scope of GPU Computing on handheld devices. Section 7 concludes this article with an outlook on the future on GPU Computing.

II. Why GPGPU?

1. History of GPU

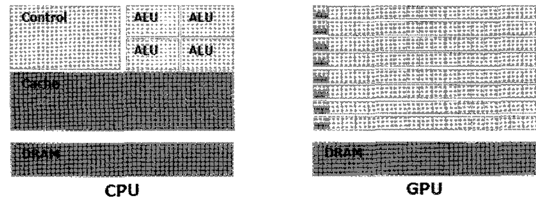
Graphics chips started as fixed function graphics pipeline, that excelled at three-dimensional (3D) graphics but little else. Over the years, these graphics chips became increasingly programmable, with both application programmable interface (APIs) and hardware, increasingly focusing on the programmable

aspect. This led NVIDIA to introduce its GeForce 8800 GTX [6], which can sustain over 330 giga-floating-point per second (GFLOPS) and streaming bandwidth of 80+ GB/s, both significantly higher than high-end CPUs. More details about history of GPU can be found from Wikipedia[8].

The key step in the evolution of GPUs was replacing the fixed-function per-vertex and per-fragment operations with user-specified programs that run on each vertex and fragment. In past few years, these vertex programs and fragment programs have become increasingly more capable, with larger limits on their size and resource consumption, with more fully featured instruction sets, and with more flexible control-flow operations. As the shader model has evolved and become more powerful, and GPU applications of all types have high vertex and fragment program complexity, GPU architectures have increasingly focused on the programmable parts of the graphics pipeline.

2. GPGPU Motivation

Why use GPUs for general-purpose computations? In a matter of just few years, the programmable graphics processor unit has evolved into an absolute computing workhouse. The annual growth rate for GPUs on PC is greater than 2.0x, which is much higher than what Moore's law predicted



〈Figure 2〉 GPU assign more transistors to data processing than CPU (from NVIDIA)

[9]. Due to its ubiquity, the computation power is more reachable than other hardware, and in terms of the computation cost, GPU for per GFLOPS is much lower than CPU.

The main reason behind use of GPU for general purpose computations is that GPUs are designed such that more transistors are devoted to data processing rather than data caching and flow control, as schematically illustrated by <Figure 2>. Another important reason is regarded to the software platform progress. At the initial stage of GPGPU development, researchers had to write assembly instructions to conduct computation on GPU. Then after high level graphics APIs came up, it turns out easier for people to write the hardware code.

3. GPGPU Target Applications

GPU is well-suited to address problems that can be expressed as data-parallel computations i.e., the same program is executed on many data elements in parallel, with high arithmetic intensity (the ratio of

arithmetic operations to memory operations). Data-parallel processing maps data elements to parallel processing threads. Many applications that process large data sets such as arrays can use a data-parallel programming model to speed up the computations. In 3D rendering large sets of pixels and vertices are mapped to parallel threads. Similarly, image and media processing applications such as post-processing of images, video encoding and decoding, image scaling, stereo vision, and pattern recognition can map image blocks and pixels to parallel processing threads. Many applications outside the field of image processing are now driven by data-parallel processing, from general image and video processing to computational finance, computational biology, or information security.

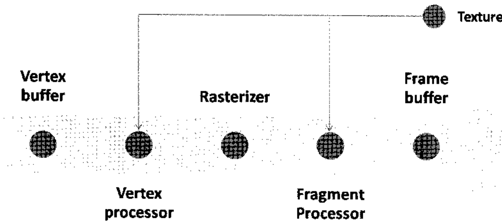
III. GPU Hardware Architecture

In this section, we start with the structure of the graphics pipeline and its evolution as a general-purpose architecture, followed by a look at modern GPU architecture (both on PC and handheld devices).

1. Graphics Pipeline

The input to graphics pipeline is a list of geometric primitives, typically triangles, in a 3D world coordinate system. Through many steps, those primitives are shaded and mapped onto the screen, where they are assembled to create a final picture. <Figure 3>. shows the pipeline stages in current GPUs.

The first stage of the pipeline, the geometry stage, transforms each vertex from object space into screen space, assembles the vertices into triangles. The output of the geometry stage is triangles in screen space. The next stage, rasterization, both determines the screen positions covered by each triangle and interpolates per-vertex parameters across the triangle. The third stage, the fragment stage, computes the color for each fragment, using the interpolated values from the geometry stage. This computation can use values from global memory in the form of textures. In the final stage, composition, fragments are assembled into an image of pixels, usually by choosing the closest fragment to the camera at each pixel location. This pipeline is described in more detail in the OpenGL Programming Guide [10]. Historically, the operations available at the vertex and fragment stages were configurable but not programmable. The key step in the evolution of GPU was replacing the fixed function per vertex and per fragment operations with the user-specified programs that run on each vertex and fragment. Modern GPUs are increasingly focusing on the programmable parts of the



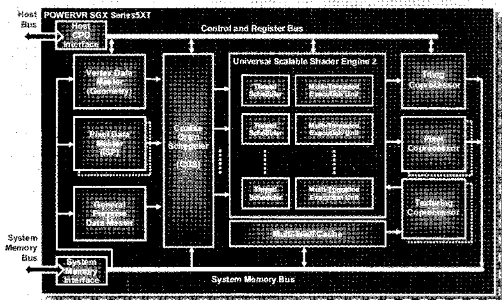
〈Figure 3〉 Graphics hardware pipeline

graphics pipeline, exploiting increased vertex and fragment program capabilities.

2. Architecture of Modern GPUs

The target area for GPU differs greatly from CPUs. GPU is well suited to address problems that can be expressed as data-parallel computations with an emphasis on throughput rather than latency. Consequently, the architecture of the GPU has evolved in a different direction than that of the CPU.

Consider a pipeline of tasks, that must process a large number of input elements. In such a pipeline, the output of each successive task is fed into the input of the next task. To execute such a pipeline, a CPU would take a single element (or group of elements) and process each stage in the pipeline, then the next stage, and so on. The CPU divides the pipeline in time. The GPU divides the resources of the processor among the different stages, such that the pipeline is divided in space, not time. The part of the processor is assigned for each stage, which are executed at the same time (task parallelism); within each stage, more than one element is computed at the same time providing data parallelism. The task and data parallelism across and between stages delivers high throughput as compared to time based division in CPU. <Figures 4, 5> shows architecture of GPU used in handheld devices (PowerVR SGX, [5]) and PC (NVIDIA, [6]), respectively. POWERVR SGX enables highly linear scaling of all aspects of GPU performance, specifically vertex shading, pixel shading, primitive setup and overall GPGPU functionality. The POWERVR SGX543MP family enables upto 16 cores of POWERVR SGX543 programmable GPU. At 200MHz core frequency an SGX543MP4 (four cores) is expected to deliver 133 million polygons per second and fill rates in excess of 4Gpixels/sec.



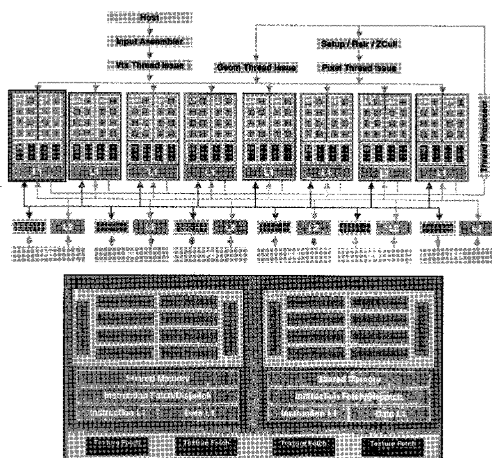
〈Figure 4〉 PowerVR SGX Architecture
[Courtesy Imagination Technologies]

IV. GPU Programming Model

Programmable GPU follow a single program multiple data (SPMD) i.e., it processes many elements (vertices or fragments) in parallel using the same program. Each element is processed independent from the other elements, and in some of the programming model, elements cannot communicate with each other. Elements can read data from a shared global memory and with the newest GPUs, also write back to arbitrary locations in shared global memory.

1. General-Purpose Computing on the GPU

- GPU programming for graphics: The programmer specifies geometry (polygons) that covers a region on the screen. The vertex program and rasterizer generates a fragment at each pixel location. A fragment program then computes the value of the fragment by a combination of arithmetic operations and global memory read. The resulting image is rendered on the display.

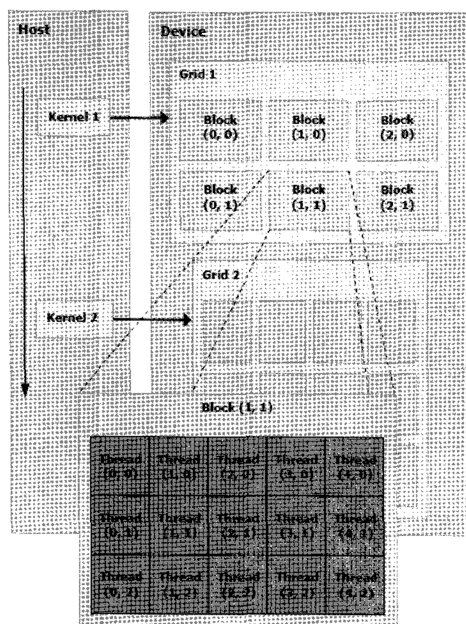


⟨Figure 5⟩ NVIDIA GeForce 8800 Architecture [Courtesy NVIDIA Corporation]

- GPU Programming for General-Purpose Programs (Shader model): The shader model performs general-purpose computation involves the exact same step as programming GPU for graphics. Most researchers use pixel programs to solve general-purpose problems. The reason behind is that often the pixel processors are more powerful than the vertex ones, and it is also easier to fetch textures serving as the memory. After the programmer specifies a geometric primitive, the rasterizer generates a fragment at each pixel location covered by that geometry. Each fragment is shaded by an SPMD general purpose fragment program. The fragment program computes the value of the pixel by a combination of math operations and global memory fetch and store the value in a frame buffer. The buffer is either displayed or used as an input for further processing using different fragment program. Generally, Cg [12], OpenGL Shading Language (GLSL) [10], or High Level Shading Language

(HLSL) [13] is used to write the kernel programs to operate the data on the graphics hardware.

- GPU Programming for General-Purpose Programs (CUDA model): One of the major difficulties in programming GPGPU applications has been that despite their general-purpose tasks, which has nothing to do with graphics, the applications still had to be programmed using graphics APIs. To overcome these problems, NVIDIA unveiled the Compute Unified Device Architecture (CUDA) [14], which allows the use of the C programming language to code algorithms to execute on the GPU. The programmer directly defines the computation domain of interest as a structured grid of threads. An SPMD general-purpose program (or kernel) computes the value of each thread. The value of each thread is computed by a combination of arithmetic operations and both read from and write to global memory. The resulting buffer in global memory can then be used as an input in future computation. <Figure 6>. illustrates the CUDA programming model.



〈Figure 6〉 CUDA programming model (from NVIDIA)

V. GPGPU on Handheld Devices

1. Application Scope

Commercial high performance signal processing applications that use GPUs as accelerators for general-purpose tasks have been implemented using GPU on desktop environment [15,16,17]. Emerging aspects of the architecture of mobile GPUs and their increasing availability make them interesting options for implementing and deploying such applications on handheld devices. Several signal and image processing have been targeted on the GPU, including basic operations (such as the fast Fourier transform [18] and convolution); differential equation based algorithms; and pattern recognition and

vision algorithms (such as those for sequence alignment, optical flow estimation, tracking, and stereo reconstruction [19]).

The GPU is well-suited for convolution and warping algorithms in image processing due to its hardware support for image interpolation and filtering. The GPU is also well-suited for robust edge detection, image segmentation, and other applications that may require differential equation based approaches. GPU can be used to efficiently implement iterative methods such as the conjugate gradient algorithm. The GPU is useful in implementation of both dynamic programming and hidden markov model approaches to sequence alignment and pattern recognition. It will be possible, especially on newer GPUs (multicore GPUs) to build implementations of computationally intensive applications such as video coding and data encryption. Key components of video coding, such as motion compensation and wavelet transformation, have already been implemented on desktop GPUs.

2. GPGPU APIs for Handheld Devices

- OpenGL ES 2.0: OpenGL ES is based on OpenGL, probably the most adopted 3D graphics API ever. OpenGL ES has been desgined as the main 3D API for mobile phones and other handheld devices. The Khronos group [20] took OpenGL as a starting point for a leaner version targeted for embedded system, OpenGL ES. The main design goals were minimizing the use of resources that are scarce on mobile devices. In OpenGL 1.x, the graphics pipeline has fixed functionality. That is, the algorithm of each pipeline stage is fixed. OpenGL 2.0 defines the OpenGL Shading Language, which allows two parts of the graphics pipeline run a program called a shader. The vertex shader replaces the vertex transformation and lighting stages of the fixed-functionality pipeline. Each vertex is processed separately, but by associating other data such as the positions of the neighboring vertices, even non-linear combinations and transformations can be applied to the vertex. Also, an arbitrary lighting model can be applied that may produce non-photorealistic shading models such as cartoon rendering. The vertex shader is followed by a stage that interpolates data associated with vertices, and for each fragment (a sample of a pixel) within a triangle (or line or point) a fragment shader is executed. The fragment shader can access interpolated vertex data, as well as previous values written into that fragment. OpenGL ES 2.0 adopted the OpenGL shading language with few modifications. However, while desktop

OpenGL decided to keep all the old fixed-functionality entry points, OpenGL ES simplified the design by eliminating all the functionality that shaders replace.

- OpenCL: Open computing language [21] is a framework for developing and executing parallel computation across heterogenous processors (CPUs, GPUs, and other embedded/ handheld digital signal processors (DSPs). Originally floated by Apple [2], the OpenCL framework supports both data parallel and task parallel programming models and is designed to operate efficiently with graphics APIs such as OpenGL [22]. The key feature of OpenCL is that it is designed not as a GPU programming platform, but as a parallel platform for programming across a range of platforms. OpenCL is touted as an extension to NVIDIA CUDA [14] language. Its support for heterogenous processors means that it is well suited for programming across desktop and handheld devices.

3. Examples

To better understand some of the characteristics of the available GPGPU programming solutions, we will consider a simple 2D convolution example. An implementation of this example, written in OpenGL ES 2.0 and OpenCL is given in <Figure 7, 8>, respectively. In Figure 7, a 2D convolution filter is expressed using OpenGL ES 2.0 fragment shader. Image and filter coefficient data is loaded into texture maps (Image and Filter). During the rasterization stage, the shader is invoked for each pixel, with the interpolated texture coordinate assigned to TexCoord value. The interpolated texture coordinates give each pixel or fragment a unique part of the input data to process. The uniform keyword identifies values that are same for all pixels, and sampler2D are bound to texture units into which the image and filter data are loaded.

The task of computing 2D convolution in OpenCL is split among threads in the following way:

- [WIDTH X HEIGHT] image space is divided into thread blocks of size [GROUP_DIMX X GROUP_DIMY]
- Each thread with in thread block is responsible for each pixel.
- Image and filter data are copied from CPU to GPU as global float array
- A single kernel program is executed by each thread and output is copied from GPU to CPU.

In these implementations, a large amount of code is not shown. Additional host CPU code is required to copy image and filter data from CPU to GPU.

VI. OUTLOOK

1. Future of GPU Computing

The rising importance of GPU Computing, GPU hardware and software are changing at a remarkable rate. In coming years, we expect to see high-end GPUs available across heterogeneous platforms such as mobile phones and other handheld devices like UMPC's, netbook, etc.

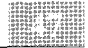
- Future GPUs are expected to support double precision floating point hardware, which will enable adoption of the GPU for scientific computing applications.
- The introduction of OpenCL will provide highly accelerated parallel computation across GPUs and CPUs to many emerging rich consumer applications.
- With OpenCL, the day when we will be able to hold a supercomputer in the palm of our hand is perhaps not to far away.
- With the emergence of multicore GPUs, such as POWERVR SGX543MP, future handheld devices will be computation powerhouses.
- Potential GPGPU applications on handheld devices in near future could include print capability, as well as advanced features like scan and fax a document, Augmented reality, and data encryption.
- High-end mobile GPUs like POWERVR SGX and NVIDIA Tegra will enable interactive 3D user interfaces and advanced multimedia features.
- Most mobile phones do not yet have GPU with GPGPU capabilities. Samsung Omnia HD i8910 is the first mobile phone to be equipped with POWERVR SGX graphics core with OpenGL ES 2.0 support.. We at samsung are devoted at unleashing the power of the GPU to provide users with advanced image and video applications on handheld devices.

```
precision mediump float;
precision highp int;
uniform sampler2D Image;
uniform sampler2D Filter;
uniform float2 offset;
varying vec2 TexCoord;
void main()
{
  const int filter_width =7;
  vec4 c = vec4(0.0,0.0,0.0,1.0);
  for (int x = 0; x < filter_width; x++) {
    for (int y = 0; y < filter_width; y++) {
          float weight = texture2D(Filter, float2(x,y));
          float2 tap = TexCoord - (float2(x,y)+offset);
          c = c + texture2D(Image, tap) * vec4(weight,
weight, weight, 0.0);
    }
  }
  gl_FragColor = c;
}
```

〈Figure 7〉 OpenGL ES 2.0 Convolution Fragment Shader

2. GPU Computing Shortcomings

- The lower bandwidth path between CPU

and GPU is a major bottleneck in many applications.

- One of the prime reason GPU can deliver huge peak performance is that it has some constrained on its architecture, notably supporting simple control flow and memory access patterns. Applications requiring unpredictable branching, looping conditions, and irregular memory access pattern hence suits better to CPU than GPU.
- Handheld devices are mostly powered by battery, which bridles the use of high-end GPUs on present devices.

```
#define GROUP_DIMX          (16)
#define GROUP_DIMY          (16)
#define WIDTH               (256)
#define HEIGHT              (256)
#define KERNEL_WIDTH        (7)
#define KERNEL_RADIUS       (3)

__kernel void Convolution (__global float *in, __global float
*out, __global const float *filter)
{
    int block_x = get_group_id(0);
    int block_y = get_group_id(1);
    int local_x = get_local_id(0);
    int local_y = get_local_id(1);
    int in_x = mad24(block_x,GROUP_DIMX,local_x);
    int  in_y  =  mad24(block_y,GROUP_DIMY,local_y);


    float cc = 0.0;
    for(int jj = 0; jj < KERNEL_WIDTH; jj++) {
        for (int ii = 0; ii < KERNEL_WIDTH; ii++){
                int  kernel_index  =  mad24(jj,
KERNEL_WIDTH, ii);
            int aa = in_y - (ii + KERNEL_RADIUS);
            int bb = in_x - (jj + KERNEL_RADIUS);
            if (aa >=0 & aa <HEIGHT & bb >= 0 &&
bb        <        WIDTH)        {
              int input_index = mad24(aa, WIDTH, bb);
            cc        =        cc        +
in[input_index]*filter[kernel_index];
        }
      }
    }
    int out_index = mad24(in_y,WIDTH,in_x);
    out[out_index] = cc;
}
```

〈Figure 8〉 OpenCL Convolution Kernel

참고문헌

[1] Evolution of the Mobile Phone Industry and Changes in Competition Structure, Korea Economic Trends, 2008.

[2] http://www.apple.com/

[3] K. Takamoro, J. Tabuchi, M. Watanabe, and Y. Hirota, BDevelopment of battery pack for mobile phones, NEC Res. Develop., Vol.44, No.4, pp.315~320, Oct., 2003.

[4] http://gpgpu.org/

[5] http://www.imgtec.com/powervr/

[6] http://www.nvidia.com

[7] http://innovator.samsungmobile.com

[8] http://en.wikipedia.org/wiki/Graphics_processing_unit

[9] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A.E. Lefohn, T.J. Purcell, “A survey of general-purpose

computation on graphics hardware", in Proceedings of Eurographics 2005, State of the Art Reports, Dublin, Ireland, pp.21~51, 2005.

[10] OpenGL, D. Shreiner, M. Woo, J. Neider, and T. Davis, "OpenGL (R) Programming Guide: The Official Guide to Learning OpenGL(R)", version 2 (5th edition), 2005.

[11] D. Blythe, "The Direct3D 10 system," ACM Trans. Graph., Vol.25, No.3, pp.724~734, Aug., 2006.

[12] W. R. Mark, R. S. Glanville, K. Akeley and M. J. Kilgard, "Cg: a system for programming graphics hardware in a C-like language", ACM Transactions on Graphics(Proceedings of SIGGRAPH2003), 22(3), pp.896~907, 2003.

[13] http://en.wikipedia.org/wiki/High_Level_Shader_Language

[14] http://www.nvidia.com/object/cuda_home.html

[15] G. Shen, G.-P. Gao, S. Li, H. Shum, and Y. Zhang, "Accelerate video decoding with generic gpu," IEEE Transactions on Circuits and Systems for Video Technology, Vol.15, No.5, pp.685~693, May, 2005.

[16] R. Yang and M. Pollefeys, "A versatile stereo implementation on commodity graphics hardware," Real-Time Imaging, Vol.11, No.1, pp.7~18, 2005.

[17] J.-M. Frahm, M. Pollefeys, and M. Shah (Co-Chairs), Proc. of CVPR workshop on visual computer vision on GPU"s (CVGPU), 2008.

[18] T. Jansen, B. von Rymon-Lipinski, N. Hanssen, and E. Keeve, "Fourier volume rendering on the GPU using a split-stream-FFT," in Proc. Vision, Modeling and Visualization Workshop, Stanford, Nov., 2004, pp.395~403.

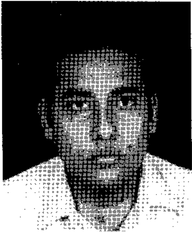
[19] S.M. Seitz, B. Curless, J. Diebel, D. Scharstein, nd R. Szeliski, "A comparison and evaluation of multi-view stereo reconstruction algorithms," in roc. IEEE Computer Society Conf. Computer ision and Pattern Recognition (CVPR), New York, Nov., 2006, Vol.1, pp.519~526.

[20] http://www.khronos.org/

[21] http://www.khronos.org/opencl/

[22] http://www.opengl.org/

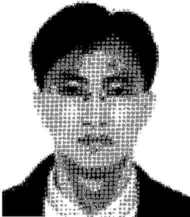
저자소개



Nitin Singhal

2008년 8월 서울대학교 전기컴퓨터공학부 공학석사.
2008년 9월 ~ 현재 삼성전자 DMC 총괄 통신연구소 연구원

주관심 분야 : 영상처리 및 GPGPU



조 성 대

1996년 2월 숭실대학교 컴퓨터학부 공학사
2000년 5월 Rensselaer Polytechnic Institute (R.P.I.) 전자컴퓨터공학부 공학석사
2002년 5월 R.P.I. 전자컴퓨터공학부 공학박사
2002년 6월 ~ 2004년 8월 R.P.I. 영상처리 센터 Postdoc
2004년 9월 ~ 현재 삼성전자 DMC 총괄 통신연구소 책임연구원

주관심 분야 : 멀티미디어 영상처리, Color Processing, GPGPU