3-2005

Accelerating Missile Threat Engagement Simulations Using Personal Computer Graphics Cards

Sean E. Jeffers

ACCELERATING MISSILE THREAT ENGAGEMENT SIMULATIONS USING PERSONAL COMPUTER GRAPHICS CARDS

THESIS

Sean E. Jeffers, Major, USAF

AFIT/GE/ENG/05-08

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

ACCELERATING MISSILE THREAT ENGAGEMENT SIMULATIONS USING PERSONAL COMPUTER GRAPHICS CARDS

THESIS

Sean E. Jeffers, Major, USAF
AFIT/GE/ENG/05-08

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/05-08

ACCELERATING MISSILE THREAT ENGAGEMENT SIMULATIONS USING PERSONAL COMPUTER GRAPHICS CARDS

Sean E. Jeffers, BS
Major, USAF

Approved:

/signed/
______________________________ ____________
Rusty O. Baldwin, Ph.D. (Chairman) date

/signed/
______________________________ ____________
Stephen C. Cain, Ph.D. (Member) date

/signed/
______________________________ ____________
Barry E. Mullins, Ph.D. (Member) date

AFIT/GE/ENG/05-08

ACCELERATING MISSILE THREAT ENGAGEMENT SIMULATIONS USING PERSONAL COMPUTER GRAPHICS CARDS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science in Electrical Engineering

Sean E. Jeffers, BS

Major, USAF

March 2005

Table of Contents

Page

List of Figures

List of Tables

AFIT/GE/ENG/05-08

Abstract

The 453d Electronic Warfare Squadron supports on-going military operations by providing battlefield commanders with aircraft ingress and egress routes that minimize the risk of shoulder or ground-fired missile attacks on our aircraft. To determine these routes, the 453d simulates engagements between ground-to-air missiles and allied aircraft to determine the probability of a successful attack. The simulations are computationally expensive, often requiring two-hours for a single 10-second missile engagement. Hundreds of simulations are needed to perform a complete risk assessment which includes evaluating the effectiveness of countermeasures such as flares, chaff, jammers, and missile warning systems. Thus, the need for faster simulations is acute.

This research speeds up these mission critical simulations by using inexpensive commodity PC graphics cards to perform intensive image processing computations used to simulate a heat seeking missile's tracking system. The innovative techniques developed in this research reduce execution time by 33% and incorporate a user-selectable fidelity feature to perform high-fidelity simulations when required. Furthermore, these image processing computations use only 5% of the available computational capacity of the graphics cards, providing a ready source of additional computational power for future simulation enhancements.

Analysts can now meet shorter suspenses with more accurate products, ultimately enhancing the safety of Air Force pilots and their weapon systems. With ongoing operations in Iraq and Afghanistan, and a growing threat at home and abroad posed by the proliferation of man-portable missiles, the speed of these simulations play an important role in protecting forces and saving lives.

This page intentionally left blank.

ACCELERATING MISSILE THREAT ENGAGEMENT SIMULATIONS USING PERSONAL COMPUTER GRAPHICS CARDS

I. Introduction

Motivation for this research comes from two fronts. First, a review of the literature reveals that commodity graphics accelerator cards, found in almost every personal computer on the market today, have reached a level of power and programmability that enables them to be used as high performance stream computers, adaptable to a variety of general purpose computing tasks [MoA03][Mor03][RuS01][KrW03][LaM01][LWK03]. Further, these devices, commonly referred to as Graphics Processing Units (GPU), can actually outperform the modern CPU in a range of computationally intensive applications [TrS01][KrW03][BFH04][LWK03]. The GPU therefore represents a powerful, untapped resource with the potential to provide a sizeable performance boost for little to no extra cost[1].

The second motivation for this research stems from a mission requirement. The 453d Electronic Warfare Squadron, part of the Air Force Information Warfare Center (AFIWC), is exploring ways to speed up the execution of computer-based simulations, specifically those used to evaluate the effectiveness of the countermeasures, such as flares, chaff, jammers, and missile warning systems, used by USAF aircraft against missile threats. AFIWC uses the Joint Modeling and Simulation System (JMASS) Threat Engagement Analysis Model (TEAM) software to run simulated engagements between missile threats and friendly aircraft, under various maneuver and environmental conditions, evaluating scenarios for the warfighter that would be cost prohibitive or logistically impossible to obtain otherwise. The results of AFIWC threat analyses

[1] Mainstream graphics cards range in price from about $60 or less to about $500.

determine the adequacy of existing countermeasures, tactics, techniques and procedures, and are used in the development of new ones. With ongoing operations in Iraq and Afghanistan, and a growing threat abroad posed by the proliferation of man-portable missiles, AFIWC simulations play an important role in protecting forces and saving lives.

Unfortunately, JMASS simulations take a long time to execute: up to two hours to simulate a 10-second engagement. This is a problem for several reasons. To provide the best possible analysis, hundreds of simulations must often be done to cover the many variations of position, maneuver, and environment for a given scenario. The quality of analysis is therefore constrained both by the amount of time available for conducting simulations and the JMASS execution time. When operating under a short suspense, quality can suffer. Further, the missiles are becoming smarter, able to identify target features at ever increasing levels of detail. Correspondingly, there is an increasing need for higher-fidelity simulations, which of course requires more time to execute due to the increased amount of computation required. JMASS is generally run on high-end personal computers and multiprocessor workstations. Though the speed of these machines continues to increase, it has not been sufficient to match the demand for faster and more detailed simulations.

To address these concerns, AFIWC initiated a collaborative effort with the Air Force Research Laboratory, Naval Sea Systems Command and the Air Force Institute of Technology to develop a hardware-based means for accelerating the image processing calculations thought to present the greatest computational load during JMASS simulations. Since this requirement emphasizes performance in the processing of graphical information, it seemed worthwhile to apply today's flexible and powerful GPUs toward providing a low-cost, potentially high-payoff solution. The remainder of this chapter provides an overview of the JMASS simulation process and a detailed characterization of the problem posed by AFIWC.

JMASS Background and Characterization of AFIWC Requirement

JMASS simulations execute, as do most simulations, in discrete steps that model the state of the system at regular intervals of simulated time. This interval is called the *model time step*, and can be thought of as either the simulation's time resolution, or the rate at which the simulation "samples" the simulated world [Air04]. In JMASS, the model time step is usually set to update the simulated environment in 1/250 second (equivalently, 0.004 second) intervals. During each time step, the JMASS simulator generates a digital image to simulate the missile's current infrared (IR) field of view, essentially mimicking the way the world would appear to a missile during flight. The image is submitted to a mathematical model representative of a particular missile's electro-optical sensor (a.k.a. seeker) and control system path, and the missile's response (i.e., maneuver or change in direction) is fed back to the JMASS simulator for generating the next scene. This iterative and interactive process of scene generation and missile optics response occurs about 2,500 times to simulate a 10-second engagement.

Of specific interest are the image processing calculations for modeling the optical path of the seeker, since this is where JMASS appears to spend most of its runtime. A typical infrared seeker is positioned, not surprisingly, at the front of the missile and consists of an IR-transparent dome followed by a set of optics not unlike a telescope. The optics focus incoming light, presumably emanating from the missile's prospective target, through a rapidly spinning, partly transparent disc, called a *reticle, w*hich modulates the light and passes it to an IR detector. The reticle is specially designed to modulate the light in such a way that the position of the target relative to the center of the missile's field of view can be determined from the modulated signal. The missile's control system uses this signal to guide the missile to the target [MaV83].

JMASS simulates the seeker system described above by modeling the interaction between the spinning reticle and the incoming IR scene. It accepts IR scene images as input, and produces a reticle-modulated signal as output. The calculations associated with this step in the simulation process, described in the following paragraphs, are the subject of AFIWC's hardware acceleration initiative, and likewise, the candidate for potential GPU acceleration.

Prior to beginning a JMASS simulation, a data structure is initialized to model the reticle. The reticle image is represented as a static 480 x 480 element array, with each element (or pixel) containing a floating point number whose value is between zero and one, indicating the degree to which each point on the reticle permits light to pass through it. The reticle image for the chosen missile is loaded into CPU memory from a data file prior to the start of the simulation.

For each model time step, JMASS determines an appropriate angular displacement for the reticle (recall the reticle is spinning), then creates a rotated copy by performing a linear coordinate transformation on the original. The rotated reticle image may be resized to match the resolution of the IR scene produced by the simulator, then interpolated by one of four selectable algorithms to smooth any artifacts that may have been caused by the rotation and resizing transformations. JMASS performs an element-by-element multiplication of the rotated, smoothed reticle image with the current IR scene to produce a new image, one that represents the IR scene filtered (or attenuated) by the reticle. Finally, the values of all the pixels of this resultant image are summed to produce a single radiance value. This value represents the light intensity that would be incident on the missile's IR detector given the input scene and reticle orientation at a particular instant in simulated time.

Recall the field of view is updated (i.e., a new IR scene is produced by the JMASS simulator) 250 times per simulated second. However, because the spinning reticle results in a modulated detector signal with frequency on the order of 1-2 kHz, sampling theory requires a minimum sampling rate of 4,000 samples per simulated second. AFIWC has specified a higher, 10 kHz sampling rate to protect against aliasing. Since the JMASS simulator's 250 Hz simulation step falls well short of this, each scene must be multiplied by *forty* reticle images (each requiring a different amount of rotation, followed by resizing and interpolation), and forty sums produced, to provide the 10,000 samples per simulated second to replicate the detector signal. Figure 1-1 below presents a simplified view of this process.

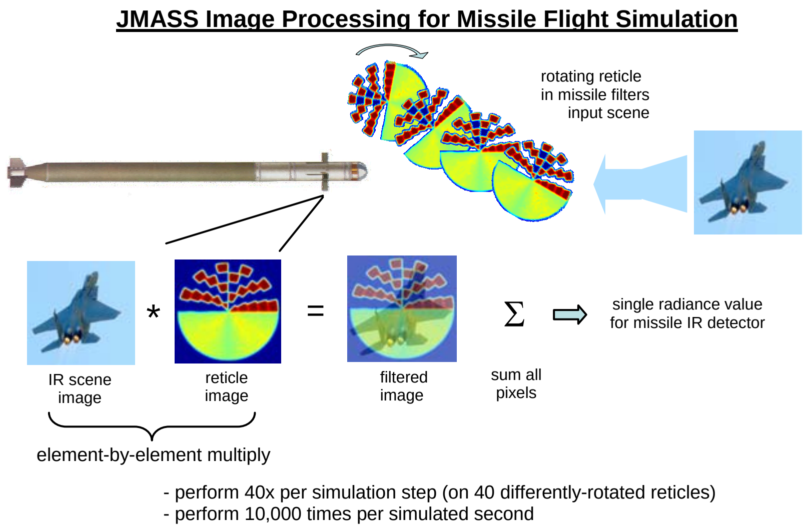

JMASS Image Processing for Missile Flight Simulation
rotating reticle in missile filters input scene
*
=
Σ
single radiance value for missile IR detector
IR scene image
reticle image
filtered image
sum all pixels
element-by-element multiply
- perform 40x per simulation step (on 40 differently-rotated reticles)
- perform 10,000 times per simulated second

Figure 1-1. How JMASS models missile optics to produce simulated IR detector signal.

For each model time step, JMASS performs the rotate, interpolate, multiply, and add operations described above as a series of separate $O(N^2)$ computations where N is the width (and height for square images), in pixels, of the images being operated on. Depending on the size of the images used, this could require on the order of 75 million double precision floating point calculations per model time step, or 19 billion calculations per simulated second[2]. The JMASS software is written in C++, for the most part, and executes on a Windows or Unix-based platform. To provide a concrete example, it takes about two hours for JMASS, running on a 2.8 GHz Pentium 4, using 512^2-sized images, to simulate a 10-second engagement.

The optics calculations described above model the behavior of a *spin scan* seeker. Generally, missiles employ one of two types of seekers, spin scan or *conical scan.* JMASS can simulate both types. Conical scan is similar to spin scan except that the IR scene is larger (generally twice the height and width of the reticle image), and prior to performing the reticle-scene multiply-add operation, the reticle image is shifted with respect to the scene by a set of specified x-y offset values, in pixels. The offset can be different for each of the forty reticle images used during a model time step. To be of greatest use to AFIWC, a GPU implementation should support both spin scan and conical scan seekers.

In addition to the GPU-based effort that is the subject of this research, AFIWC is investigating Field Programmable Gate Array (FPGA) technology to accelerate both software-based (like JMASS) and real-time, so-called "hardware-in-the-loop"

[2] Assuming a 256x256 size image and bilinear interpolation. This accounts for floating point addition, multiplication, and *sin* and *cos* operations, but does not include instructions for performing loops, lookups or array index calculations. Interpolation requires 10 floating point operations per image pixel, rotation requires 16, and the multiply-add about 2.

simulations, which interface with real missile hardware. Since software-based simulations are not performed in real-time, they stand to benefit from any amount of speedup that can be provided. However, this is not the case for real-time simulations which must either sustain a throughput of 19 GFLOPS or fail. Whether or not this kind of performance is within the capabilities of FPGAs remains to be determined, and is beyond the scope of this research. However, as will be shown later in this thesis, such performance is almost certainly beyond the current capabilities of graphics cards. Therefore, any performance gains be realized through a GPU will likely only benefit software-based simulations.

As indicated throughout this section, the AFIWC hardware acceleration initiative is predicated on the assumption that image processing calculations are the source of the performance bottleneck, and should therefore be the prime target for optimization efforts. Indeed, an analysis of the JMASS C++ code supports this assumption, since the bulk of the calculations reside in the $O(N^2)$ code structure which performs the image processing calculations [Joi04]. However, if this is not the case, optimizing the image processing calculations may not be enough. According to Amdahl's Law [HeP96], if other bottlenecks exist, they could reduce the effectiveness of even the most spectacular image processing performance gains provided by a GPU or FPGA. This does not diminish the importance of these hardware acceleration efforts. However, it suggests adopting a system-wide approach in addressing the JMASS performance issue.

This page intentionally left blank.

II. Literature Review

The General Purpose GPU

Almost every personal computer available today comes equipped with dedicated graphics acceleration hardware, either built-in to the motherboard or provided as an add-in circuit card. Though graphics co-processors, or Graphics Processing Units (GPU) as the industry refers to them, have become commodity items in personal computers, what is not generally recognized is these devices have become formidable computing machines in their own right, exceeding the modern desktop CPU in terms of raw computational power.

For example, Macedonia [Mac03] reported a 20 GFLOPS peak performance of the Nvidia GeForce FX 5900, a mainstream GPU in 2003, to be equivalent to a 10 GHz Intel Pentium. It is interesting that GPUs achieve such performance running at much slower clock rates than CPUs, the result of a highly parallelized architecture. Typical GPU clock rates range from 233 to 400 MHz, while current CPU clock rates are on the order of a few GHz. Current models of GPU contain 220 million transistors, the bulk of which are dedicated to parallel processing of input streams, whereas Intel's Xeon CPU has only 108 million transistors, 60 percent of which are devoted to cache memory [Mac03]. Equally impressive, the growth of GPU performance has exceeded Moore's Law [MoA03], increasing at a rate of 2.8 times per year since 1993, and is expected to continue at this rate for another five years, perhaps achieving tera-FLOP performance by 2005 [Mac03].

While the main, market-driven purpose of the GPU continues to be providing increased resolution, dynamic range, frame rates and programmability to keep pace with the demand for ever more realistic games and multimedia applications, these same advances have resulted in an important, perhaps revolutionary side benefit: the architecture of the modern programmable GPU has become so flexible it is possible to exploit its inherent computational power for many general-purpose computing tasks faster than they can be done on a CPU [TrS01][KrW03][BFH04][LWK03]. These developments have not been lost on a number of researchers who have, especially over the past four years, successfully used a GPU to accelerate a myriad of general-purpose computing tasks. Just a few of the diverse examples include linear algebra [Mor03][KrW03][LaM01], finite element analysis [RuS01], lattice Boltzmann computation [LWK03] and Fast Fourier Transform calculations [MoA03].

GPU Architecture

The ability to use the GPU for general purpose computing results from its evolution over the past decade from a fixed-function pipeline architecture, to a fully programmable Single Instruction Multiple Data (SIMD) parallel, or streaming, processor [MoA03][BFH04]. This section describes the GPU architecture.

A stream is simply a collection of data operated on in parallel [BFH04]. The GPU is optimized for rendering images, a task that involves performing fast, parallel operations on large streams of data. As such, most GPUs include their own high-bandwidth memory subsystems for storing and manipulating graphical data. For example, the current top-of-the-line mainstream GPU from nVidia, the GeForce 6800, has 256 MB of memory accessible via a 256-bit bus with an advertised bandwidth of 35.2 GB per second [Nvi04]. In late 2004, 3DLabs is expected to make available its high-end Wildcat Realizm 800 GPU with 640 MB memory, 512-bit bus, and an advertised memory bandwidth of 64

GB/second [Pci04a]. By way of comparison, the Intel 875 chipset that supports the Pentium IV only provides 6.4 GB/second CPU-to-main memory bandwidth [Int04a].

In general, the GPU processes two kinds of data: vertices and textures [Mor03]. Vertices represent points in space and are used to build graphical primitives, such as polygons, which can be assembled to form complex 3-dimensional objects. Vertices possess attributes such as color, position vector and texture coordinates, which are stored in registers and can be operated on by various functions [THO02]. Textures, on the other hand, are 1, 2, or 3-dimensional images applied to polygons, much like wallpaper or shrink-wrap, to impart the look of a realistic surface. Textures are stored in GPU memory as arrays of pixels, and each texture pixel is represented by a four-component vector, holding the intensity values for red, green, blue and alpha (RGBA) color channels.

To render an image, a user application must provide the GPU a set of vertices and/or textures. Some or all of the data may already be in GPU memory, left over from previous operations; otherwise, data must be uploaded to the GPU. The GPU can retrieve large blocks of data from CPU main memory via DMA. To prevent fast GPUs from becoming data-starved, modern PC busses include a dedicated interface for the GPU, the Advanced Graphics Port (AGP), which provides a 2 GB per second path between the GPU and main memory. This figure will increase to 4 GB per second when computers based on the next-generation PCI Express bus standard become available within the next year [Int04b][Pci04b]. Unfortunately, DMA hardware is not provided for transferring data quickly in the opposite direction. Such a capability is important since any significant use of the GPU for general-purpose computing requires transferring GPU-computed results into CPU main memory for further processing [THO02].

Once the appropriate data has been loaded into the GPU, it proceeds through the GPU pipeline in the following general sequence. First, the GPU generates geometry using the vertex information provided by the user application. The GPU transforms the geometry into a chosen coordinate frame, clips it to fit within a specified viewport, or drawing rectangle, if need be, and applies lighting and color calculations [THO02]. Next, the GPU applies textures to the geometry, and passes everything to the *rasterizer* which converts the vector-based geometry data into a pixel-based representation for rendering [THO02]. These pixels, as they exist prior to rendering, are referred to as *fragments*. Finally, the pixels are rendered into a section of GPU memory, called the *frame buffer* [LaM01], for display on the screen.

The functions of the GPU are accessible via an Application Programmer Interface (API) such as OpenGL, created by Silicon Graphics, or Microsoft's DirectX. These provide standardized interfaces, data types and functions to access the features of many GPUs. The extent the API feature set is supported or extended depends on the GPU manufacturer.

What remains to be explained is how the GPU architecture can be applied to solving general-purpose computing problems. The following from [TrS01] addresses this and nicely captures the motivation behind using the GPU for general-purpose computation:

> Modern raster graphics implementations typically have a number of buffers with a depth of 32 bits per pixel or more. In the most general setting, each pixel can be considered to be a data element upon which the graphics hardware operates. This allows a single graphics language instruction to operate on multiple data as in a SIMD machine.

Since the bits associated with each pixel can be allocated to one of four components, a raster image can be interpreted as a scalar or vector valued function defined on a discrete rectangular domain in the *xy* plane. The luminance value of a pixel can represent the value of the function while the position of the pixel in the image represents the position in the *xy* plane. Alternatively, an RGB or RGBA image can represent a three or four dimensional vector field defined over a subset of the plane. The beauty of this kind of interpretation is that operations on an image are highly parallelized and calculations on entire functions or vector fields can be performed very quickly in graphics hardware.

Further, typical scientific computing applications perform at about 1% of peak (CPU) processor performance. Recall a CPU cache hierarchy excels when it performs repeated operations on a block of data, but suffers when the block of data exceeds the cache size. The GPU, however, generally has much more memory capacity than a CPU cache, and is capable of performing operations in parallel [RuS01].

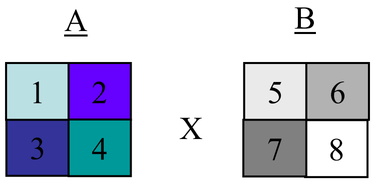
Using the GPU Fixed-function Pipeline

An early attempt to use the GPU for general-purpose numerical computation used the fixed-function pipeline of the GPU to perform matrix multiplication. 2D textures stored the matrices, with matrix element values stored as individual pixels within the textures. For reasons to be discussed later, the technique of using textures versus vertices to represent data in the GPU is widespread in the literature. The matrix multiplication algorithm referred to above exploits the spatial parallelism of GPU computation, performing a series of element-by-element multiplications of texture pairs, with element-by-element additions performed in between to accumulate results [LaM01].

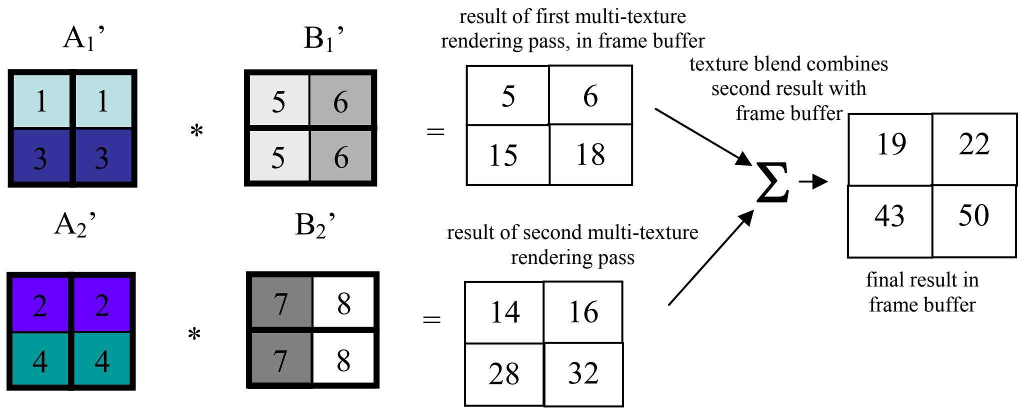
To implement the algorithm, a pair of order-*n* square matrix multiplicands A and B are preprocessed using the CPU to create two new sets of textures, A' and B', each containing *n*, *n x n* textures, such that the *i* th texture in A' contains the *i* th column from A copied across its columns, and the *i* th texture of B' contains the *i* th row from B copied across its rows. Figure 2-1 shows an example using 2 x 2 matrices. As if dealing corresponding cards from two decks, the *i* th textures from A' and B' are transferred to the GPU in pairs and multiplied element-by-element in what is called a *multi-texturing* operation. Multi-texturing takes two textures as operands and combines them in one of several user-selectable ways to produce an output texture. In this example, each pair of textures is multiplied using the "modulate" multi-texturing mode, applied to a single quadrilateral fragment in the rasterization stage of the GPU pipeline, then rendered to the frame buffer. To accumulate results, the output of each texture multiply is rendered to the frame buffer using the "sum" texture blending mode. In this mode, rendering causes the contents of the rasterizer to be added, pixel-by-pixel, with the existing contents of the frame buffer, thereby allowing the accumulation of results in the frame buffer [LaM01].

Using this technique two order-1024 square matrices were multiplied in 0.546 seconds on the nVidia GeForce3 [LaM01]. This time includes converting matrices to texture maps, transfering the textures to GPU memory, performing the calculations, copying the frame buffer back to CPU main memory, and converting back to matrix format. GPU performance is compared to a CPU-based benchmark, Automatically Tuned Linear Algebra Software (ATLAS) running on a Pentium IV. However, direct comparison is not possible because then-current GPUs were only capable of 8-bit fixed point arithmetic, and ATLAS performed its calculations in 32-bit floating point. To acknowledge this difference, GPU performance is stated in terms of byte operations per second (BOPS), and compared with ATLAS's FLOPS.

For the order-1024 matrix multiply, the GPU achieved 4.4 GBOPS and ATLAS yielded 4.0 GFLOPS. Though no execution time metric is provided for ATLAS



(a) matrices to be multiplied


A1'
B1'
A2'
B2'
result of first multi-texture rendering pass, in frame buffer
texture blend combines second result with frame buffer
result of second multi-texture rendering pass
final result in frame buffer
1 1 3 3 * 5 6 5 6 = 5 6 15 18
2 2 4 4 * 7 8 7 8 = 14 16 28 32
Σ
19 22 43 50

(b) Matrix A columns and matrix B rows copied into texture sets A' and B'. Corresponding textures multiplied element-by-element using GPU multi-texturing. Final result is computed in GPU by adding results in texture blending operation.

Figure 2-1. A technique for multiplying matrices using GPU fixed-function pipeline and textures [LaM01].

[LaM01], ATLAS running on a Pentium IV can multiply two order-1000 matrices in about 0.5 seconds [Mor03], which, precision issues aside, is comparable to the 0.546 GPU time achieved in [LaM01].

For large operations, such as multiplying twenty order-1024 matrices, the time spent transferring data to and from the GPU is negligible compared to the time spent performing multiplication and accumulation calculations. Further, calculation time is dominated by memory accesses within the GPU because the GPU architecture requires frame buffer memory accesses for both accumulation operations and for copying results from the frame buffer back into a texture [LaM01].

Though the results of [LaM01] are not entirely compelling from a performance or practical standpoint (recall the GPU's 8-bit limitation), it represents a starting point for discussion because its techniques, observations and recommendations are recurring themes in subsequent research.

First, to be useful in most scientific or engineering computing applications, the GPU should be capable of handling at least 32-bit floating point numbers [LaM01]. This limitation has in fact been overcome by recent generations of GPU, which now support 32-bit processing throughout the entire pipeline [MoA03][KrW03][Nvi04].

Second, accumulating results between rendering passes requires multiple memory accesses within the GPU, whereas a CPU can store intermediate results in fast registers. So, future GPU architectures should include persistent registers for this purpose [LaM01]. Unfortunately, current GPU hardware still does not provide this capability [BFH04]. Further, though the memory bandwidth of current GPUs is almost five times faster than those of three years ago, the integration of 32-bit floating point support offsets this bandwidth improvement because more memory accesses per pixel must be made. This is confirmed in [Mor03], where a GPU with 32-bit functionality multiplied two order-1000 floating point matrices in just over 0.5 seconds, almost exactly the same time required by the older-generation GPU operating on 8-bit data.

In addition to the above, there are other ways to increase GPU performance [LaM01]: up to four numbers may be packed into a single pixel by setting the red, green, blue and alpha channels to different values; lowering the refresh rate of the monitor could yield a 10% performance improvement; running full screen versus in a window increases performance; and using ABGR_EXT versus RGBA texture formatting in OpenGL can improve performance by 40%, since it eliminates time-consuming re-reformatting within the GPU.

The technique of texture blending in the fixed-function GPU pipeline has been used to do finite element [RuS01], and Lattice Boltzmann [LWK03] computations on GPU hardware.

The GPU Programmable Pipeline

The three years following the work of [LaM01] brought significant improvements to GPU architecture. 8-bit fixed point has been replaced with IEEE 32-bit floating point representation for each of the four color components in each pixel [KrW03]. GPU internal memory bandwidth increased by a factor of four, and clock speed increased by a factor of two. But the most significant advance with respect to GPU general purpose computing is the move toward a programmable architecture. GPUs now contain programmable vertex and fragment processors. Each processor respectively executes a user-specified assembly-level vertex or pixel *shader* program consisting of 4-way SIMD instructions that perform standard math operations, such as 3- and 4-component dot product, addition and multiplication on large, parallel streams of data. Instructions for texture fetching and other special-purpose instructions are also available. Each vertex or pixel fragment to be processed is placed in a set of read-only input registers. The shader program is executed next and the results written to a set of output registers. The shader program performs an implicit loop, executing over all the elements of a stream [THO02][BFH04].

Pixel Shaders versus Vertex Shaders

Pixel shaders have been used for matrix-vector, vector-vector and matrix-matrix multiplication, and for 2D Fast Fourier Transforms [Mor03] [KrW03] [MoA03]. Matrices are represented as a set of diagonal vectors inside a 2-dimensional texture to facilitate efficient processing of banded diagonal matrices [KrW03]. A more straightforward approach breaks column vectors into smaller, four-element sub-columns, and stores each sub-column as a texture pixel, placing the four individual elements into the R, G, B and A components of the pixel [Mor03]. Despite differing methods for packing data into textures, all exploit the 4-tuple parallelism of texture pixels to achieve four 32-bit calculations per pixel for each SIMD shader instruction. Below, is justification for using texture fragments versus vertices as the GPU data format of choice [Mor03]:

> Textured geometry is preferable because of the more compact representation when compared with highly tessellated geometry with vertex colors. Also, unlike geometry, textures can also be output by the GPU in the form of render target surfaces. If we store a matrix as a texture, and then perform a matrix operation such as matrix addition by rendering two textures with additive blending into a third render target surface, the storage format of the resulting matrix can be identical to the input format. This is a desirable property because this way we can immediately reuse the resulting texture as an input to another operation without having to perform format conversion.

A notable exception to the above approach develops a framework for general-purpose GPU computing based on vertex shader programs, as opposed to pixel (texture- or fragment-based) shaders [THO02]. The reasoning behind this choice is primarily motivated by the state of GPU technology, which at the time offered higher, 16-bit precision for vertex operations versus only 10 bits for texture operations, and a more robust, 21-opcode instruction set for vertex shaders. The framework itself is discussed later; however, there are several weaknesses in using vertex shaders, some of which have since been addressed by later GPU designs [THO02].

First, the results of vertex programs cannot be stored directly into a GPU memory buffer without first passing through the GPU pipeline and being converted to pixels. Then-current GPUs represented pixels with only 8-bit precision. Though internal vertex computations are carried out with 16-bit precision, a significant precision loss is realized when the result is retrieved as 8-bit pixels.

Second, program size is limited to 128 instructions, and branching and logical Boolean operations are not supported. Such restrictions required awkward hand-coded programming. For example, loops had to be "unrolled", and the number of loops is limited by the maximum instruction count. This limitation applies to both vertex and pixel shaders [THO02].

Lastly, there is no way to share data between multiple vertex program invocations. Though vertex programs provide at least 96 registers for holding intermediate results within a program, all registers are zeroed upon program termination [THO02].

As has been discussed previously, precision is no longer an big issue, since 32-bit floating point is supported by some models of GPU. Also, published specifications for the nVidia GeForce 6800 advertise hardware support for pixel and vertex shader programs of "unlimited" length, plus support for branching within pixel shader programs, with the caveat that the operating system and API may impose limits on program length, even though the hardware does not [Nvi04]. Further, Microsoft's High-Level Shading Language (HLSL) now supports branching and looping in pixel and vertex shader programs [Msd04]. Despite these advances, GPU hardware still does not provide persistent registers for vertex programs or a means to store the results of vertex operations without rendering to pixels. Theremfore, most recent GPU-based implementations use pixel shaders which operate on data stored as textures, and maintain state between rendering passes by saving results to off-screen texture memory buffers (a.k.a. render target textures) [MoA03][Mor03][KrW03][BFH04]. Older versions of pixel shader were subject to *clamping*, whereby color intensities were restricted to values between zero and one. Much effort has been devoted find a way to convert between real values, represented as floats, to numbers that fit within the required [0,1] range, such as in GPU-based finite element analysis [RuS01] and refractive caustics [TrS01]. It is less complicated now since subsequent versions of HLSL, with Pixel Shader version 2.0 or later, support the full floating point range [Mor03]. The abundance of applications based on pixel shaders seems to indicate the pixel shader instruction set has caught up with the vertex shader in terms of flexibility, leaving little incentive to use vertices for computation. However, vertices are still used in most of these applications for setting up the shape and size of the area to be rendered.

Frameworks, Models and Compilers for General Purpose GPU Computing

In the examples described thus far, getting a GPU to perform general purpose computing required extensive knowledge of graphics hardware and graphics programming, down to the assembly language level in many cases, on the part of the programmer. Such programming is tedious and error-prone, and best managed by a compiler [THO02]. In fact, several languages now exist that allow shader programs to be written in a high-level, C-like programming language [BFH04], including Microsoft's High-level Shading Language (HLSL), nVidia's Cg, and the OpenGL Shading Language [BFH04]. While a step in the right direction, these languages are still graphics-oriented, and require a programmer to express algorithms and data structures in terms of graphics primitives, such as textures and triangles [BFH04]. Therefore they fall short of providing an environment for generalized stream computing on the GPU.

There has, however, been some research devoted to this. One example, alluded to previously, presents a framework with abstractions for expressing vectors, and functions for operating on vectors, on a GPU. This framework defined a *DFunction* class which allows unary, binary and ternary functions, with vector operands and scalar or vector outputs, to be defined. The *DVector* class works behind the scenes to allocate an OpenGL *p-buffer* in GPU memory to accumulate results, thus shielding the programmer from the intricacies of graphics programming [THO02].

Similarly, [KrW03] devised a stream model for operating on vectors and matrices and it defined *clVec* and *clMat* container classes for expressing vectors and matrices respectively. Upon initialization, vectors, originally stored as C++ arrays, are converted to textures in the GPU and bound to texture handles. The class instance keeps track of the texture handles and sizes associated with its respective matrix or vector, and makes that information available through public functions. Arithmetic is performed via the *clVecOp* function, with an enumerator *op* to select addition, multiplication, or subtraction operations. The setting of *op* selects a corresponding pixel shader program to perform the operation on the two input textures.

An important operation in graphics processing is *reduction* [KrW03]. Reduction is an operation that condenses or evaluates all data in a stream to produce a smaller subset or a single value. Examples include summing all elements of a matrix to produce a single scalar, or finding the element with the minimum or maximum value. GPU hardware does not yet provide efficient means for accomplishing reduction operations [KrW03]

[BFH04]. Reduction, therefore, requires multiple rendering passes to accomplish. To sum all of the elements in a matrix, for example, a pixel shader program could render a quadrilateral with dimensions half those of the original matrix, placing into the elements of the new matrix the sum of four adjacent pixels from the original. The pixel shader executes recursively, operating on previous results, producing a quarter-sized texture each iteration. The final result is a single pixel containing the desired sum. Figure 2-2 illustrates this concept. This reduction algorithm operates in *O(log(n))* time, where *n* is the dimension of the original matrix [KrW03]. Of course, the number of reduction passes required can be reduced if the number of neighboring pixels summed on each pass is increased [BFH04].

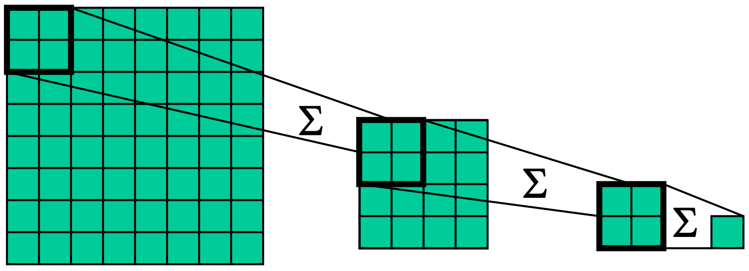


Figure 2-2. Reduction operation achieved with GPU in successive rendering passes, summing groups of four adjacent pixels in a texture and rendering to a quarter-sized render target texture in each pass.

Researchers at Stanford University went a step further than the examples above by creating a language and compiler for stream computing on graphics hardware, called *Brook*. Brook manages memory via *streams*, data objects containing collections of records. Parallel functions, called *kernels*, invoke parallel operations on streams in the GPU. Reduction functions similar to those described above are also provided. The Brook system consists of two parts, *brcc* a source-to-source compiler, and the Brook Runtime (BRT), a library of runtime support routines for kernel execution. The compiler maps Brook kernels into Cg shaders, which are subsequently compiled into GPU assembly by commonly available vendor-provided compilers. *brcc* also produces C++ code which uses BRT to invoke the kernels. Originally developed as a language for streaming processors such as Stanford's Merrimac streaming supercomputer and the Imagine processor, Brook has been adapted for use on the GPU, supports both OpenGL and DirectX, and is freely available [BFH04].

GPU Performance

The GPU does not generally operate in the same address space as the host CPU, therefore, an analysis of GPU performance must not only consider computation time, but also the time spent transferring data into and out of the GPU. This concept is captured in the metric *computational intensity*, the ratio of the total cost of executing an algorithm on a device versus the cost of transferring the data into and out of the device [BFH04]. For an application to effectively use the GPU, it must possess the following two key properties [BFH04]:

First, in order to outperform the CPU, the amount of work performed must overcome the transfer costs which is a function of the computational intensity of the algorithm and the speedup of the hardware. Second, the amount of work done per kernel call should be large enough to hide the setup cost required to issue the kernel.

In [LaM01], two order-1024 matrices were multiplied in 0.54 seconds, including data transfer time, but there was a one-time 0.2 second set-up cost. Such an application would obviously not be a suitable candidate for GPU acceleration unless many more matrices are to be multiplied.

Setting up kernels or shader programs on a GPU requires a fixed amount of CPU time. If multiple kernel calls are executed back-to-back, the setup time can overlap with the kernel execution. If the streams are large, the GPU will be the limiting factor, but if streams are small, it may not be possible to issue kernel calls fast enough to keep the GPU busy [BFH04].

The performance of Brook-compiled GPU applications has been compared against optimized and well-known CPU benchmarks, as well as hand-coded or GPU vendor-provided versions of the applications optimized for a particular GPU. In addition, both DirectX and OpenGL configurations have been tested on the most capable GPUs available in early 2004, the ATI X800XT and nVidia GeForce 6800, providing a fairly complete and current evaluation of general-purpose GPU computing capability. The test applications are linear algebra, FFT, and ray tracing [BFH04].

For linear algebra, two low-level subroutines from the ATLAS Basic Linear Algebra Subprograms (BLAS) library are emulated, SAXPY and SGEMV. SAXPY performs a vector scale and sum operation, $y = \alpha x + y$, and SGEMV performs a matrix-vector product followed by a scaled vector add, $y = \alpha Ax + \beta y$ where x and y are vectors, A is a matrix and α and β are scalars. Vector length is 1024 and matrices 1024^2 single-precision floating point. For the CPU benchmarks, the commercial Intel Math Kernel Library is used for SAXPY, BLAS for SGEMV, and FFTW-3 for the FFT [BFH04].

In most of the trials the hand-coded, optimized GPU reference applications ran slightly faster than the Brook-compiled versions. Generally, the ATI card outperformed the nVidia card by a wide margin, almost by a factor of four in the worst case. This is possibly due to higher floating point texture bandwidth on the ATI card, about 4.5 Gfloats/second, versus nVidia's 1.2 Gfloats/sec. Peak compute performance of ATI and nVidia was 40 billion and 33 billion multiplies per second respectively. Generally, DirectX outperformed OpenGL since DirectX can render directly to a texture, whereas with OpenGL, an additional copy operation is required to transfer the contents of the p-buffer into a texture [BFH04].

The ATI card running the reference GPU application under DirectX executed SAXPY about eight times faster than the CPU version, achieving about 4.9 GFLOPS. Brook running under the same circumstances achieved a 7x improvement over the CPU version. In contrast, the nVidia card's best performance for this application was only 1.5 GFLOPS, about 2.4-times improvement over the CPU version. For the reason noted earlier, OpenGL versions generally achieved only half the performance of the DirectX versions. For SGEMV, the ATI card under DirectX provided about a 1.7x increase in performance over the CPU, and the nVidia card actually ran slower than the CPU. For the FFT application, the ATI card performance matched that of the CPU, and the nVidia card achieved about 0.7 the performance of the CPU [BFH04].

In the case of the ATI card under DirectX, the GPU either exceeded or matched the CPU-based applications. Even more encouraging is that the CPU benchmarks were optimized to make very efficient use of the CPU cache structure [Mor03][BFH04], which means that the GPU would most likely provide even greater performance gains versus non-optimized C++ applications. For instance, without its cache optimization, the effective performance of the CPU-based FFT application FFTW would be cut by over 80 percent, making the GPU version a full six times faster by comparison [BFH04]. It would certainly be beneficial if cache optimizations could be applied in the programming of GPUs. Unfortunately, the order pixels are processed within the GPU is an undocumented implementation detail, which makes it difficult to exploit data locality in the same manner as is routinely done in CPU programming [Mor03].

For the SGEMV application, the GPU beat the CPU, but not by as wide a margin as in other applications. This is most likely because SGEMV involves a vector-matrix multiplication, requiring a multi-pass reduction step. If GPU hardware is equipped with the persistent registers necessary to facilitate single-pass reductions, performance could be significantly enhanced. For instance, computing the sum of 2^{20} 32-bit floats took approximately 0.79 milliseconds on an ATI/DirectX platform, compared to 14.6 milliseconds on an optimized CPU implementation. While this is good, it is estimated that such an operation would only take 0.18 milliseconds were the GPU hardware to provide support for such reductions [BFH04].

Conclusion

Some have envisioned supercomputing may one day be conducted on clusters of inexpensive PCs equipped with multiple high-performance graphics cards versus multiple CPUs [THO02][Mor03]. The power of the modern GPU is indeed impressive, and it is becoming increasingly easier to harness that power for general-purpose computing. With respect to the JMASS requirement, some of the examples in the literature are directly applicable. For example, time-domain convolution has been accomplished more efficiently by the common technique of first performing an FFT on two images, multiplying them element-by-element in the frequency domain, then performing an inverse FFT on the result [MoA03]. JMASS similarly requires an element-by-element multiplication of two matrices, an operation that can be trivially accomplished with a pixel shader program [MoA03]. After multiplying the rotated reticle image with the IR scene, JMASS requires that all elements be summed to produce a single luminance value. Such reduction operations were considered in [KrW03] and [BFH04], and it has been shown that they can be accomplished faster on a GPU. Of the operations required by JMASS, only the rotation operation seems to have no direct parallel in the literature. The GPU does provide built-in means for mapping textures, via indexed lookups, to transformed (including rotated) polygons [THO02], making it likely that the GPU can be used for accelerating the JMASS rotation operation. However, to do so the GPU must implicitly perform an interpolation on the original data. How best to implement these operations on a GPU, and whether the GPU can deliver acceptable levels of accuracy will certainly be subjects of this research.

This page intentionally left blank.

III. GPU Implementation and JMASS Integration

Integration With JMASS

For the GPU to be of any help to JMASS, JMASS must change the way it processes reticle images. Recall from Chapter I baseline JMASS generates properly oriented reticle images to multiply with the IR scene by rotating and interpolating a static reticle image template. These image rotation and interpolation operations are performed forty times per model time step, for a total of 100,000 times each during the simulation of a 10-second engagement. A more efficient approach, proposed herein, is to store a set of pre-rotated and interpolated reticle images of the required size in memory (either in the GPU or in CPU main memory), and to look them up when needed versus generating them repeatedly through costly transformation operations throughout the execution of the simulation. Integrating GPU processing into JMASS essentially *requires* this sort of approach to capitalize on the GPU's fast texture memory and to limit costly data transfers between CPU and GPU. Even if the GPU is not used, such a lookup-based approach is much more efficient because it effectively eliminates *hundreds of thousands* of $O(N^2)$ image rotation and interpolation operations.

AFIWC accepted this proposal and produced a modified version of JMASS which implements a lookup-based approach for reticle images. A set of 100 incrementally rotated images, spanning a complete rotation of a reticle, is sufficient to replace the continuously variable rotations of the baseline approach. Prior to simulation start, modified JMASS generates this set of 100 pre-processed reticle images, then, depending on whether or not the GPU is being used, either uploads them to the GPU, or stores them in CPU main memory for later use in the simulation.

The only image processing operation remaining to be performed by the GPU, therefore, is the reticle-scene multiply-add operation. This consists of performing an element-by-element multiplication of two arrays (the scene and reticle images), and a summation of the result to produce a single output value. This operation is performed on forty reticle-scene image pairs during each simulation time step after each IR scene update. Upon each scene update, the new scene image is uploaded to the GPU. For spin scan simulations, the scene is multiplied by forty consecutive reticle images (out of the 100), each with a slightly greater rotation than the next, and the forty results are returned to JMASS. For conical scan, the reticle images are called for in a random-access fashion, such that the forty that are used may not be consecutive, or may even repeat. The conical scan approach additionally requires the reticle and scene images be shifted with respect to each other by specified amounts prior to the multiply-add operation, and the shift can be different for each of the forty reticle images used in the time step.

GPU Implementation

Before attempting to implement the JMASS multiply-add operation on a GPU, several design choices had to be made, starting with the graphics cards. First and foremost, the graphics cards need to support the IEEE-754 floating point format. At the time of this writing only two graphics cards meet this requirement, the ATI X800XT and the nVidia 6800 Ultra. Though it is possible that other exotic and far more expensive graphics cards exist with similar or better features, these cards were chosen because they represent the top of the line available to consumers, and because their GPU clock speed and feature sets are directly comparable. A second important requirement is the graphics cards have sufficient on-board memory to support the storage of the 100 reticle images, plus the input scene and several textures for storing intermediate results between rendering passes. The 256MB capacity of these cards was adequate in most cases. A final necessity with respect to the graphics cards is they must support Pixel and Vertex Shader version 2.0 or better because the reduction operations require dependent texture addressing[3], which is not fully supported in previous versions. For the graphics API, DirectX was chosen over OpenGL because it provides quicker mechanisms for retrieving data from the GPU [BFH04]. Shader programs were written in Microsoft's High-Level Shader Language (HLSL) versus assembly language for the sake of simplicity. The code for controlling the GPU and interfacing it with JMASS was written in C++ to facilitate easier integration with JMASS, which is also written in C++. This code is included in Appendix B.

Theory of Operation

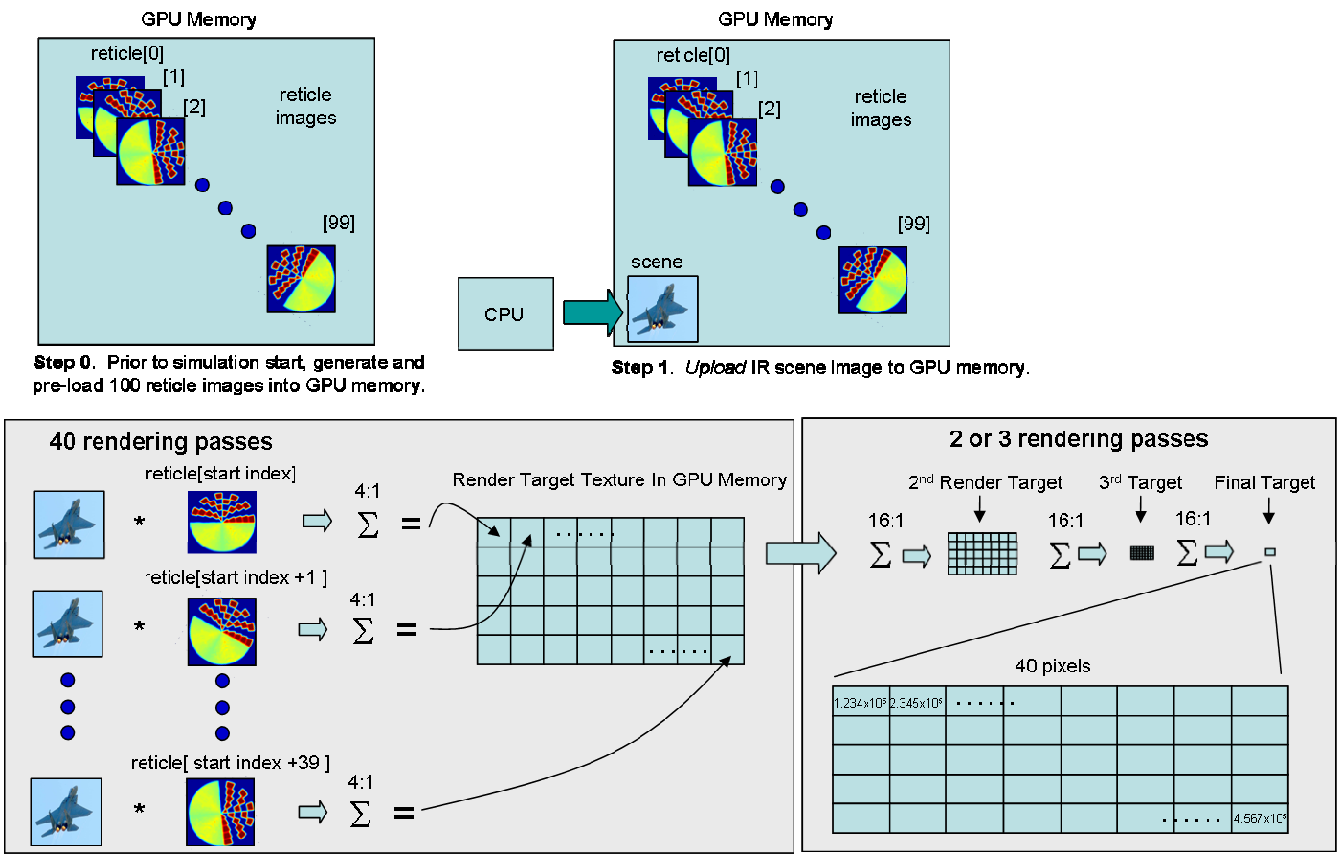
The GPU interface is instantiated as an object, with methods for uploading reticle images and for processing scene images. For spin scan, JMASS calls the *GPU.process* method, sending as parameters references to both the scene array and an array for storing the forty returned results, plus the starting reticle image index for the consecutive sequence of reticles to multiply with the scene. For the conical scan implementation, JMASS identifies the indices of the forty reticle images to use, and provides forty sets of x-y offsets for shifting them with respect to the scene.

Due to the high degree of programmability and rich feature set offered by the graphics cards and DirectX API, there are many ways to implement the multiply-add operation on a GPU. For this research, two approaches were explored for organizing the computations within the GPU.

[3] Dependent texture addressing allows texture coordinates which address one texture pixel to be used to derive the coordinates for another.

Sequential Approach

The first is called the "Sequential" approach. Figure 3-1 shows a step-by-step progression of this algorithm. "Step 0" consists of uploading the 100 pre-processed reticle images into GPU memory and storing each reticle image as a separate texture. When 512^2 images are used, this requires about 100MB (1MB = 2^{20} bytes) of GPU RAM for storing the reticle images. This step occurs before the start of the actual JMASS simulation. Once the simulation is started, JMASS calls the *GPU.process* method during each simulation time step after a new IR scene is generated. As shown in Figure 3-1, GPU processing takes place in three steps. During the first step, the new IR scene is uploaded into GPU memory. In the second, *multiply and add* step, the scene is multiplied (element-by-element) with forty consecutive reticle textures, producing a sequence of forty new result images. Further, blocks of four adjacent pixels are summed, producing result images that are a quarter the size of the original scene and reticle images. The forty result images are rendered one at a time into a single, large texture in GPU memory, arranged so as to fill five rows of eight images. At this point, the reticle and scene images have been multiplied, but their elements have only been partially summed. These intermediate results are stored in a single large texture. Step three, called the *reduce* step, completes the summation operation by successively rendering from one intermediate result texture into another sixteenth-size texture, summing blocks of 16 adjacent pixels. After two or three such reduction operations, depending on the initial size of the reticle and scene images, the final result texture contains forty pixels, with each pixel containing the result of a corresponding reticle-scene multiply-add operation. The forty numbers are



Step 0. Prior to simulation start, generate and pre-load 100 reticle images into GPU memory.

Step 1. *Upload* IR scene image to GPU memory.

Step 2. *Multiply and add.* Using GPU programmable pipeline, multiply scene by a reticle image element-by-element, then sum blocks of four adjacent pixels, producing a quarter-sized result image (4:1 reduction). Store result in render target texture. Do for 40 consecutive reticles. Render target contains the 40 resulting images arranged in a grid. Note: though not indicated above, reticle index values are mod 100.

Step 3. *Reduce.* Sum blocks of 16 adjacent pixels, producing a 1/16 th-sized result image (16:1 reduction). Repeat up to three times (depending on original scene/reticle dimensions). End result is 40 pixels, each a floating point number that is the desired matrix "dot product" of the scene with a reticle. These are returned to JMASS. Return to Step 1 to process next scene.

Figure 3-1. "Sequential" approach for processing JMASS multiply-add operation in the GPU.

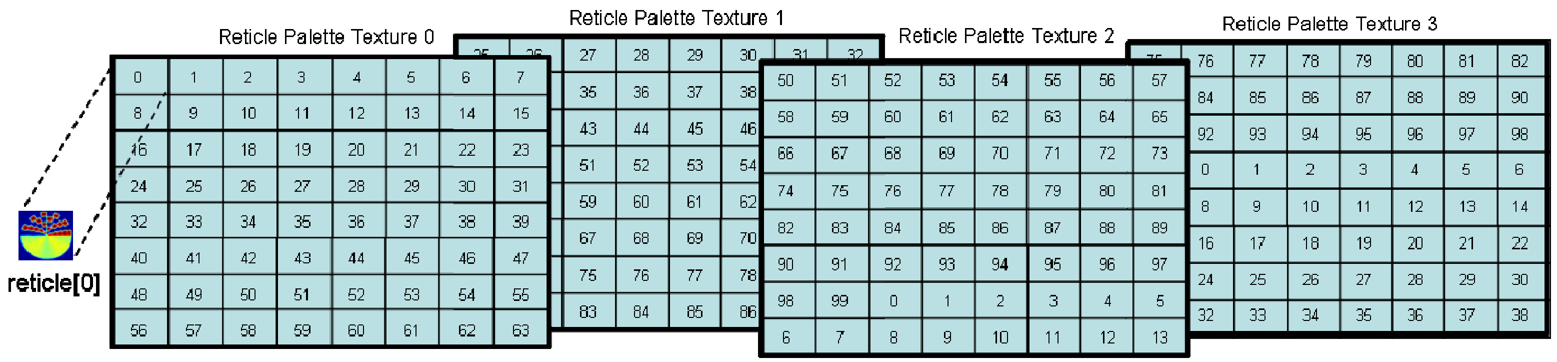
retrieved from the GPU and returned to JMASS, and the GPU waits for the next scene to be uploaded (i.e., returns to "Step 1" in Figure 3-1). This "Sequential" approach requires up to 43 rendering passes: 40 reticle-scene multiply and add operations, followed by up to three reduction (summation) operations.

Palette Approach

A second method, called the "Palette" approach, achieves the same results, but gets there by taking a different path. Figure 3-2 provides a step-by-step pictorial representation of this algorithm. For clarity, the general approach is first described, followed by specific details.

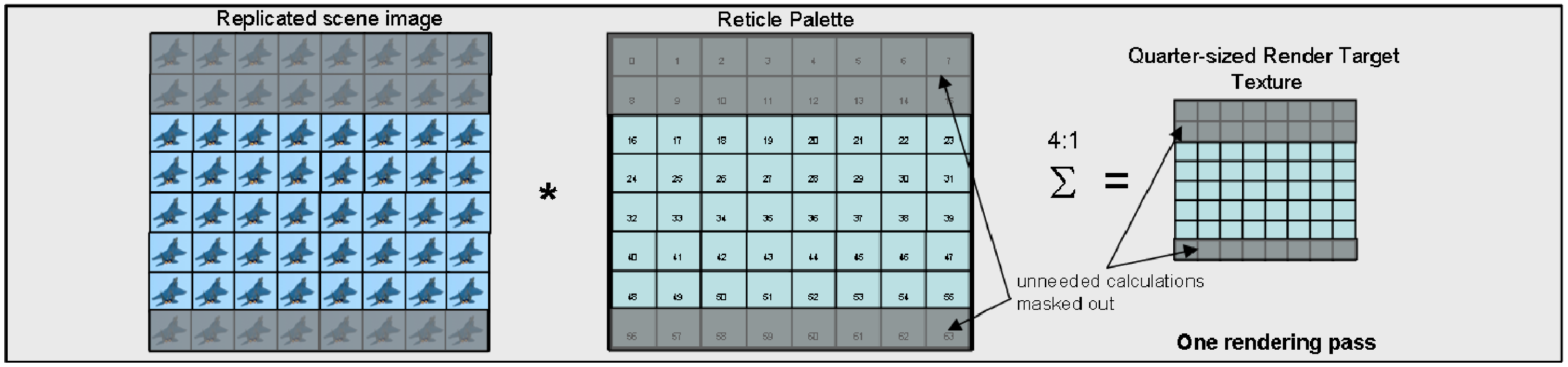
To begin with, this algorithm stores the reticle images in the GPU differently than the previously described approach. Instead of storing the 100 reticle images as separate textures, they are arranged by rows and columns, like tile, into one large "palette" texture. After the scene image is uploaded to the GPU, it is multiplied with the larger palette texture, taking advantage of a GPU addressing mode which effectively replicates the scene image across the palette texture so many copies of the scene image line up to be multiplied with the many reticle images contained in the palette texture. This is shown in Figure 3-2, in the diagram for Step 2. In this manner, the scene can be multiplied by many reticle images in a *single* rendering pass. After multiplying, blocks of four adjacent pixels are summed such that the resulting texture is a quarter the size of the original reticle palette texture. Thus, this first rendering pass produces a quarter-sized texture holding the results of many reticle-scene multiplications, but the pixels have only been partially summed. As in the "Sequential" approach, the summation operation is completed by performing up to three 16:1 reduction operations, resulting in a final result texture containing forty pixels, from which the values are retrieved and returned to JMASS (see Figure 3-2, Step 3). This approach requires a maximum of four rendering passes to complete.

Now that the basic approach has been presented, some of the important details left out of the above discussion can be addressed. First, GPUs impose limitations on the maximum allowable size for textures. Shader programs further impose that textures be square and they have a power-of-two dimension to use dependent texture addressing.

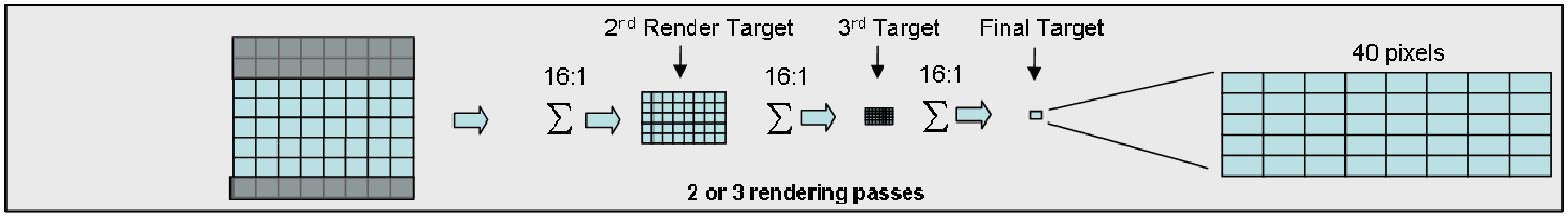

Reticle Palette Texture 0
Reticle Palette Texture 1
Reticle Palette Texture 2
Reticle Palette Texture 3
reticle[0]

Step 0. The 100 pre-rotated reticle images are pre-loaded into GPU memory, distributed among four large texture grids, or palettes, as shown above. Each palette contains an assortment of 64 reticle images. The number in each box indicates the index {0..99} of the reticle image placed there. The above distribution ensures that, regardless of the choice of starting reticle index, there is always one palette that contains all 40 needed reticle images consecutively. This method was chosen as an efficient way to perform the algorithm, while keeping the textures square and power-of-two sized to conform to pixel shader constraints, and small enough so as not to exceed maximum allowable texture sizes on GPUs.

Step 1. *Upload* IR scene image to GPU.


Replicated scene image
*
Reticle Palette
4:1
Σ
=
Quarter-sized Render Target Texture
unneeded calculations masked out
One rendering pass

Step 2. *Multiply and add*. Through repeated addressing, the input scene is effectively replicated into an 8 x 8 grid of images (above left) equal in size to a reticle palette, then multiplied, using the GPU programmable pipeline, by the appropriate reticle palette, as determined by the starting reticle index. Because only 40 image multiplications are necessary, the GPU drawing rectangle is adjusted to mask out unneeded images, and eliminate unneeded calculations. The above example depicts the case where the starting reticle index is 16. After the multiplication, groups of four adjacent pixels are summed, producing a quarter-sized result image (4:1 reduction). The multiplication and summation described occurs in a single rendering pass.



Step 3. *Reduce*. Same as Step 3 in "Sequential" approach. The result from Step 2 is reduced up to three times via 16:1 summation operations to yield 40 floating point pixels containing the matrix "dot product" of the scene with 40 reticle images. These results are returned to JMASS. Return to Step 1 to process next scene.

Figure 3-2. "Palette" approach for processing JMASS multiply-add operation in the GPU.

Since this addressing mode is vital to efficient accomplishment of this algorithm, the textures are subject to all of the above constraints. The effect of these constraints is it is impossible to fit the complete set of 100 reticle images into a single palette texture for all but the smallest supported image size (128^2). To solve this dilemma, four different palettes are loaded into GPU memory, each containing a 64-image subset of the 100 reticle images. The 100 reticle images are distributed among the four palettes such that, given any starting reticle index, there is always at least one palette which contains the next 39 required reticles in a contiguous block. All that is required is some simple range checking in the GPU interface to ensure the correct palette is chosen for the multiply-add operation based on the starting index provided by JMASS. Storing the four palette textures requires 64MB of GPU memory if the 256^2 image size is being used. For the 512^2 image size, however, the 256MB GPU memory capacity is not large enough to store four reticle palettes. To solve this problem, a more complicated three-palette method was devised, which consists of multiplying the scene with up to two different palettes, essentially performing this algorithm twice, and combining the results at the end. The three palette images require 192MB of GPU memory.

Though the fixes described above meant the “Palette” approach overcame texture size constraints, the approach itself proved to be very inefficient; it always performed 64 (or more) reticle-scene multiply-add operations, when only 40 are actually needed. However, DirectX provides a way to narrow the size of the drawing rectangle so rendering can be restricted to a desired rectangular subset of a render target texture. The palette approach was therefore modified to automatically set the drawing rectangle so as to exclude as much of the unneeded portions of the textures as possible from being processed. Doing so reduced execution time for this algorithm by almost 20% compared to its original incarnation. Some inefficiency still remains, however, because this method can still allow up to eight extraneous images to be processed. Figure 3-3 shows this remaining inefficiency.

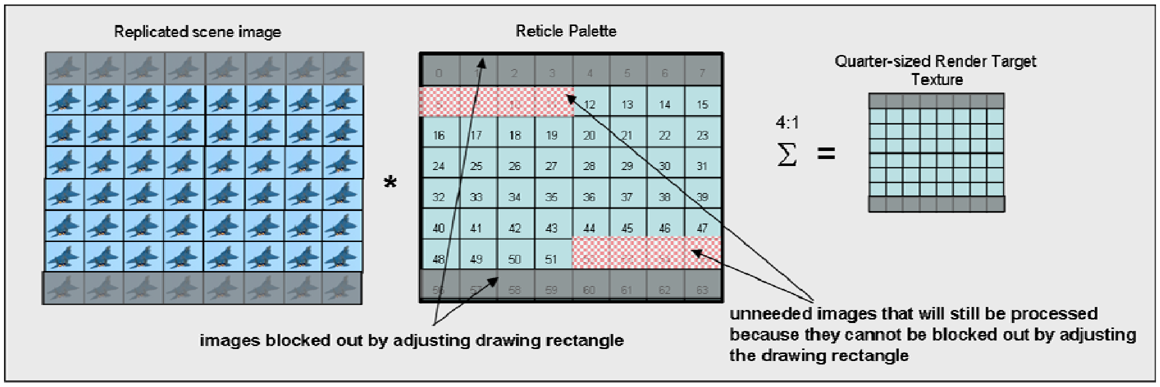

Replicated scene image
Reticle Palette
Quarter-sized Render Target Texture
4:1
Σ =
*
images blocked out by adjusting drawing rectangle
unneeded images that will still be processed because they cannot be blocked out by adjusting the drawing rectangle

Figure 3-3: Inefficiency can result in "Palette" GPU algorithm implementation. In this example, the forty needed reticle images are indices 12–51 in the Reticle Palette. Though adjusting the drawing rectangle can block out some of the unneeded reticle images (grayed-out strips at top and bottom), it cannot block out those which are highlighted in the checkered pattern. This results in the GPU processing eight extra images that are not needed. The GPU therefore takes longer to process certain ranges of reticle images than others, depending on the range given, and how many of the unnecessary images can be masked by the drawing rectangle.

Preliminary tests show that the "Palette" approach works well on the smaller image sizes (128^2 and 256^2), but the "Sequential" approach may be the better of the two for the 512^2 image size. Interesting to note, despite the fact that the "Sequential" approach requires up to 43 rendering passes, and the "Palette" approach requires only four, the two algorithms are comparable in performance. Further, though the "Palette" approach works at all three image sizes on the PCI-express platform, it does not support the 512^2 image size on the AGP platform. For some reason, perhaps due to DirectX or graphics card AGP drivers, the AGP machine will not allow more than one reticle texture (which is a full 64MB in this case) to be loaded into GPU memory. Instead, the remaining palette textures are forced into AGP aperture memory (off the graphics card), causing GPU processing to take minutes instead of seconds to accomplish. The "Sequential" approach is most likely immune to this limitation because it does not require so many large textures.

In both the “Palette” and “Sequential” approaches, the reticle and scene images are stored in the GPU such that four image pixel values are packed into each texture pixel, using the texture pixel’s four (R, G, B and A) color channels as a vector, thereby fully exploiting the four-way parallelism of the GPU. However, through experimentation it was found that maintaining such packing in the reduction stage slows the GPU down, and it is better to transition to a single-channel texture format (having a single 32-bit floating point R channel versus the full 4 x 32-bit RGBA) for the reduction passes. Doing so results in as much as 1.21x speedup for these implementations.

Conical Scan

To support conical scan, a separate GPU algorithm was created, based on the “Sequential” algorithm described above. The “Palette” approach could not be used because it cannot process reticle images out of order, and the reticle images, being part of a single, static texture that is accessed in one rendering pass, cannot be shifted by differing amounts with respect to the scene. Because conical scan requires shifting images by arbitrary amounts prior to multiplying them, the reticle and scene images cannot be packed four-to-one into texture pixels as they are for the spin scan approaches. Instead, a one-to-one correspondence has to be maintained between image and texture pixels, so spatial integrity is retained after shifting. Although this reduces efficiency somewhat, resulting in slightly higher GPU execution times, the performance is still competitive with other approaches (see Chapter V). To implement this algorithm, the shader programs were modified to add horizontal and vertical offset values to the texture coordinates for the reticle images as shown in Fig 3-4.

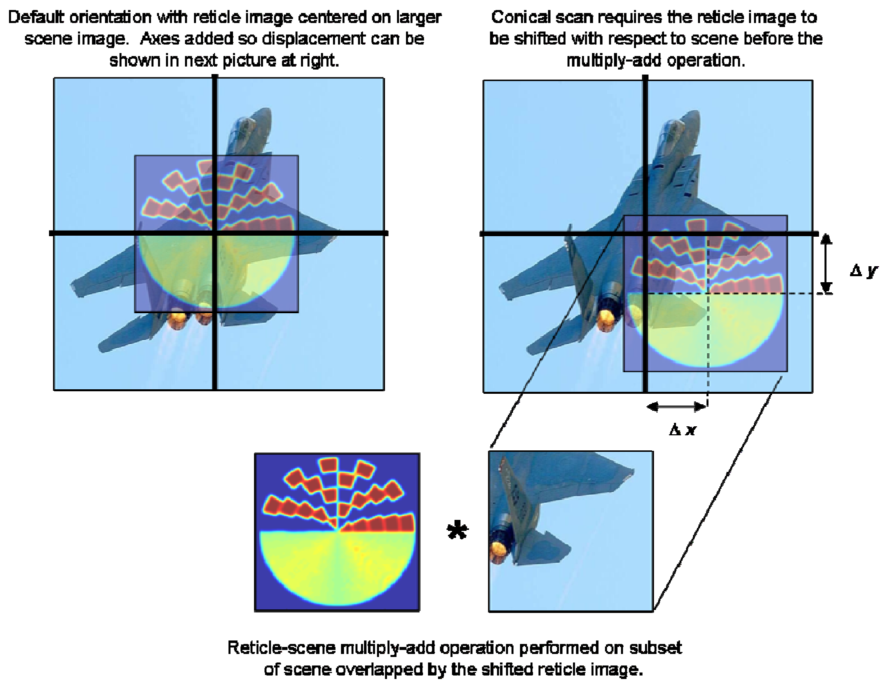

Default orientation with reticle image centered on larger scene image. Axes added so displacement can be shown in next picture at right.
Conical scan requires the reticle image to be shifted with respect to scene before the multiply-add operation.
Δ y
Δ x
*
Reticle-scene multiply-add operation performed on subset of scene overlapped by the shifted reticle image.

Figure 3-4. Conical scan variation of the JMASS optics processing. Same as "Sequential" implementation, except reticle image must first be shifted with respect to the larger scene image, by specified x-y offsets, before performing the multiply-add operation. The shift is performed in the GPU, using a vertex shader.

A special feature was also added to the code to permit scene sizes of arbitrary dimension, versus the power-of-two and square shape constraints of the spin scan versions. Reticle images, however, remain bound by those constraints, but this is not a detriment since reticles are circular and hence symmetrical in shape. One final observation with respect to shifting the images: some shift amounts can result in 5-10% longer execution times for the GPU. This is likely due to cache misses or address translation within the GPU. For the conical scan experiments, whose results are discussed in Chapter V, the shift amounts are randomized to provide reasonably accurate estimates of performance that can be expected.

This page intentionally left blank.

IV. Methodology

Problem Definition

Goals, hypotheses and approach. The primary goal of this research is to determine whether, and to what extent, a GPU can speed up JMASS simulations. The image processing calculations currently carried out in the JMASS software have been presented as the main system performance bottleneck. The general approach, therefore, is to replace the JMASS image processing software routines with GPU hardware processing, and compare the performance of the GPU-assisted JMASS with that of the baseline JMASS system. It is expected that the GPU will provide some degree of acceleration since its inherent parallelism, enhanced memory bandwidth and stream processing characteristics make it better suited to these tasks than traditional pipelined CPU processing.

The second goal of this research is to determine the performance gains achievable using a GPU. To do so requires testing GPU and alternative processing methods apart from JMASS in a controlled environment. The resulting experiments represent a control group to be used as a basis for comparison and for interpreting the results of the experiments which involve JMASS. To accomplish this goal, GPU performance is compared with that of two CPU-based (i.e., software-based) alternatives for accomplishing the reticle-scene multiply-add operation: a basic C++ software implementation, and another which makes use of a widely available cache-optimized linear algebra library. The first implements the reticle-scene multiply-add operation using basic C++ loop structures, much like baseline JMASS does; the second uses the cache-optimized Intel Math Kernel Library (MKL) *sdot* command to perform the operation. Throughout the rest of this document, the three implementations are referred to as GPU, C++ and MKL.

Using the experimental methodology defined herein, it is determined whether currently available GPUs will reduce JMASS execution time, and whether they provide any advantage over CPU-based implementations. It is anticipated the GPU will do both. In addition, the results will quantify JMASS speedup due to the GPU and the maximum overall speedup achievable by optimizing the image processing operations. Ultimately, the results will guide future hardware and software designs.

System Boundaries

For the primary goal of determining whether GPU hardware processing can improve JMASS performance, the system under test includes: the JMASS software; a high-performance mainstream PC host with minimal I/O (only keyboard, monitor, mouse, and disk drive); MS Windows XP operating system and the latest version of the DirectX API; a top-of-the-line mainstream graphics card; and a custom-designed C++ module to control GPU operations and provide an interface for exchanging data between JMASS and the GPU. The component under test in this case is the combination of the graphics card and the custom interface software.

For the second goal of comparing GPU performance to that of CPU-based software alternatives, three system configurations are used. Each configuration consists of a stand-alone PC as defined above, a simple application to generate reticle images and scene images (workloads), plus one of the following processing methods, described earlier, as the component under test: GPU, C++, or MKL.

An additional initial phase of experiments is conducted prior to those indicated above to select the best-performing GPU for use in subsequent experiments. See the Experimental Design Section, Table 4-1 and Figure 4-1 for precise details on system configurations and what is tested in each experimental phase.

System Services

The JMASS system simulates the flight of an IR-seeking missile from launch to contact with the target. It simulates both the external environment and the missile's responses to that environment with the behavior of the simulated missile recorded for later analysis. The overall system service provided by JMASS, therefore, is to generate behavioral data for various simulated missiles and environments.

This research focuses on optimizing the subset of JMASS that performs the image processing calculations which simulate the optical path of the missile's IR seeker. The image processing service receives an IR scene from the JMASS environment simulator, multiplies it element-by-element with rotated versions of a template reticle image, reduces each of the resulting images to a single pixel by summing all its pixels, and returns the computed values back to the JMASS simulator. JMASS supports various image resolutions. The following image sizes, in pixels, are representative of those routinely used in JMASS simulations: 128^2, 256^2, 512^2 and 1024^2 (for conical scan).

Possible outcomes are either a correct or incorrect computation of results, or complete failure to produce results. Incorrect results would result if the GPU algorithm were in some way flawed. Possible causes include improper texture lookup or interpolation of texture values. Floating point truncation is another possible source of error. In baseline JMASS, calculations are carried out in double-precision floating point format. The graphics cards, however, are limited to single-precision IEEE-754 standard 32-bit floating point format. Additionally, although the ATI card supports IEEE-754 format, it uses only 24 bits to represent each float (16 bits mantissa, 7 exponent), making the ATI implementation more susceptible to truncation error. It can therefore be expected the different implementations will produce different, if not incorrect, results. Due to the graphics cards' decreased precision, it is also possible numeric overflow may result. Complete failure to produce results would be indicative of a system or subsystem failure.

Workload

For those experiments involving the JMASS system, the workload consists of running an unclassified AFIWC-provided test scenario at each of three scene/reticle image sizes: 128^2, 256^2 and 512^2 pixels. The specific scenario used is the unclassified Generic Man-Portable Air Defense System (MANPADS) Threat Model, set for a 10-second engagement. This scenario is representative of the types of workload used in JMASS simulations.

For the remaining experiments which do not involve JMASS, the workload consists of test images representing the IR scene, submitted repeatedly to the system for processing. Since the GPU executes a deterministic mathematical operation on known input data, the accuracy of the output is easily verified, and the test data for these experiments need not originate from JMASS. The workload is varied by changing the size and content of the test images, which are the only aspects of the workload that can be changed.

With respect to the content of the workload, three scene update schemes are used: non-changing, fully changing, and moving point source. The non-changing scheme sends the same image to the processor every time. It is expected that this scheme will provide the most accurate, "best case" measurement of execution time, since it introduces no delay between calls to the processing method (i.e., the GPU, C++ or MKL method under test). The moving point source scheme causes a single, unit-valued pixel to trace out a square path within the scene over time, emulating a JMASS point source simulation. This is considered a "middle of the road" scheme in that it only changes two pixels of the scene upon each update. Since the point source "moves" in a non-sequential way through array memory, it may induce cache-specific behavior. The fully-changing scheme is intended to more closely resemble JMASS since it changes every pixel in the scene image between calls to the optics processing routine. The fully-changing scheme is accomplished by adding the value of 1.0 to each pixel upon a scene update. Since updating the scene in this manner requires some processing time, it is expected that observed execution times will at least increase by some uniform amount. A change that is disproportionate may indicate an unexpected interaction of factors.

The workload uses image sizes of 128^2, 256^2 and 512^2 pixels. Conical scan experiments, however, use scene image dimensions twice those of the reticle image. For those experiments, the reticle image sizes (in pixels) are 128^2, 256^2 and 512^2, and the corresponding scene image sizes are 256^2, 512^2 and 1024^2.

Image size is the most important factor of the workload, since it directly affects execution time. Image content is important from the standpoint of verifying calculations have been performed correctly, and that values do not exceed the range of 32-bit floating point numbers. The scene update scheme, which periodically alters the contents of the workload, might also impact performance.

Performance Metrics

Execution time is the natural choice for a performance metric since this research is motivated by a requirement to reduce JMASS execution time. An additional metric is to measure the differences, if any, between the results computed by baseline JMASS and those produced by the GPU and software-based image processing implementations developed for this research. Such deviations are expected to be relatively small, resulting from floating point truncation. They are nevertheless reported because it is unknown how such differences, however small, will affect simulation outcomes.

Parameters

Parameters are those aspects of the system or the workload which could affect system performance if changed. The following is a comprehensive list of parameters, and their associated levels where applicable. Note that only a subset of these are actually varied during the experiments (see Factors below).

- System parameters:
    - PC platform.
        - Processor type and speed, cache size
        - Memory and I/O configuration
        - I/O Bus architecture
        - Operating system
        - DirectX version
    - GPU (graphics card)
    - Software
        - JMASS version
        - JMASS configuration
        - GPU algorithm implementation
        - Pixel shader version
        - Image processing implementation
- Workload parameters
    - Image size in pixels
    - Scene update scheme

Factors

Factors are those parameters which are expected to have the greatest impact on system performance and so will be varied singly or in combination with other factors during the experiments. The chosen factors, and their associated levels, are listed below.

- System factors:
  - PC platform.
    - Bus architecture: AGP or PCI-express
  - GPU (graphics card): ATI Radeon X800XT or NVidia GeForce 6800
  - Software
    - JMASS configuration: Baseline, Modifed JMASS (Software), Modified JMASS (GPU-assisted)
    - Image processing implementation: GPU, non-optimized software (C++), or cache-optimized linear algebra library (MKL)
    - GPU algorithm implementation: Palette versus Sequential
- Workload factors
  - Image size: 128^2, 256^2 and 512^2 pixels IR scene and reticle images
  - Scene update scheme: non-changing, fully-changing, moving point source

The factor expected to cause the greatest performance variation is the image size (workload) factor. This is because increasing image dimensions exponentially increases the required number of multiplication and summation operations. Further, based on review of the literature, there may be large performance differences between cache-optimized and non-optimized software implementations. The PCI-express bus architecture doubles the bandwidth for data transfers between host and GPU memories, and so may also be an important factor.

GPU algorithm implementation (Palette or Sequential) represents two different ways to organize the rendering operations performed by the GPU. Preliminary tests show the best (i.e., fastest) method to use depends on the GPU and image size being used. Details of the GPU algorithm implementation options are discussed in Chapter III.

Parameters related to the PC platform (with the exception of bus architecture) are not varied because it is expected that AFIWC will simply run JMASS on the highest-performance mainstream PC available, equipped with minimal I/O and a large RAM complement. While such parameters can certainly affect overall system performance, any changes attributable to them would be the same regardless of whether GPU acceleration was being used, so they are held constant during these experiments. Further, because neither the JMASS nor GPU processes require frequent disk access, the disk subsystem is not seen as an important parameter with respect to JMASS or GPU performance.

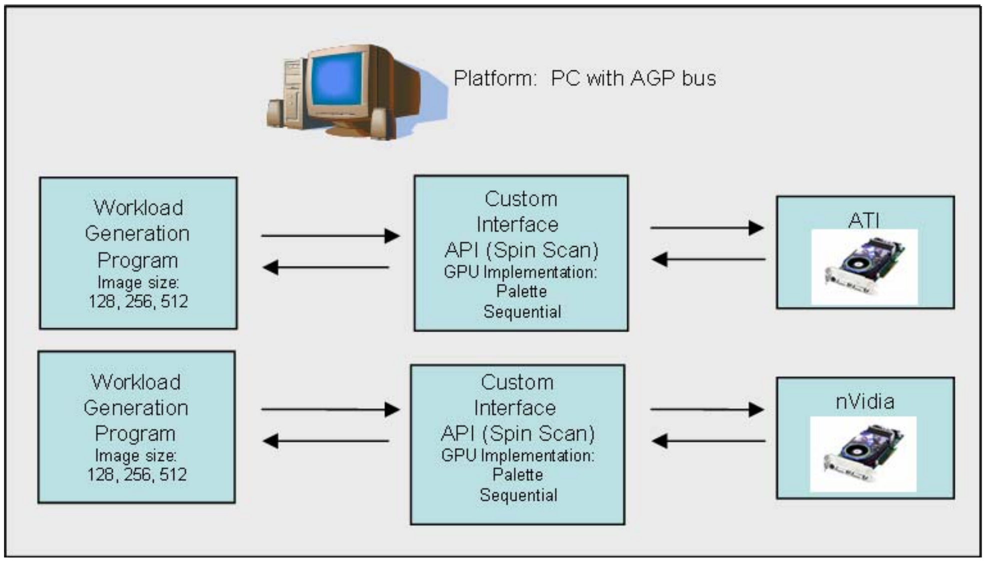
Evaluation Technique

The evaluation technique is primarily direct measurement since all resources are readily available for experimentation, and execution time is easily measured. Further, since graphics cards are proprietary devices, they defy simulation using standard software tools or analytical methods. While strictly speaking simulation is not used, the first three phases of experiments, described in detail in the following section, can be considered an *emulation* which predicts to some degree how the GPU would perform if it were integrated into JMASS. The results of such stand-alone subsystem testing can be validated by comparing them with the results of the fourth phase of experiments, which integrate the GPU with JMASS. This is discussed further in the Analysis of Results section.

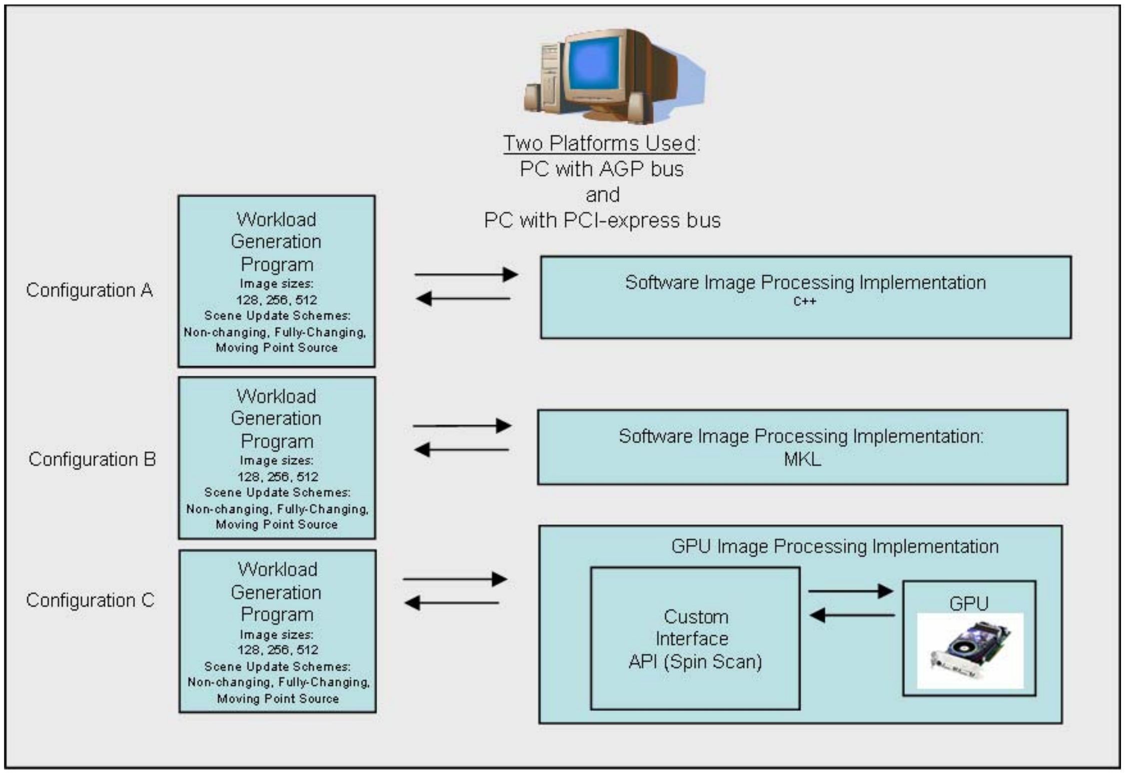
Experimental Design

Experiments are organized into four phases, each with its own specific purpose and experimental design. The first phase is intended to compare the stand-alone (separate from JMASS) spin scan image processing performance of the two graphics cards under various workloads, using two different GPU algorithm implementations, and to select the configuration which yields the best performance for use in subsequent experiments. In this first phase of experiments, JMASS is not used. Instead, the two graphics cards, the ATI X800XT and the nVidia 6800 Ultra, are treated as stand-alone subsystems which emulate the JMASS image processing function. Since the nVidia card was not available in a PCI-express version, only the AGP versions of the cards are compared. Each replication of an experiment consists of submitting a test image (IR scene) to the graphics card for processing 1,000 times and measuring the total execution time. Execution times were measured using calls to the Windows C++ *timeGetTime()* command, which returns the value of the system clock with one-millisecond resolution. Running 1,000 iterations of the GPU algorithm ensures execution time results well above 1 millisecond for all experiments. Running the GPU algorithm 1,000 times is roughly equivalent to the amount of optics processing performed by JMASS to simulate 4 seconds of a missile's flight. The factors (levels) varied in this set of experiments are: graphics card (NVidia, ATI); image size (128^2, 256^2, 512^2); GPU algorithm implementation (Palette, Sequential); and scene update scheme (non-changing, fully-changing). However, because the "Palette" GPU implementation does not run correctly on the AGP platform at the 512^2 resolution, the subset of experiments involving the 512^2 image size are analyzed separately to prevent skewing the results.

Two experimental designs are used in this phase of experiments. The first, involving the 128^2 and 256^2 image sizes, is a $2^k r$ full-factorial experimental design using the $k = 4$ factors listed above and $r = 30$ replications. In this phase, and in Phases Two and Three

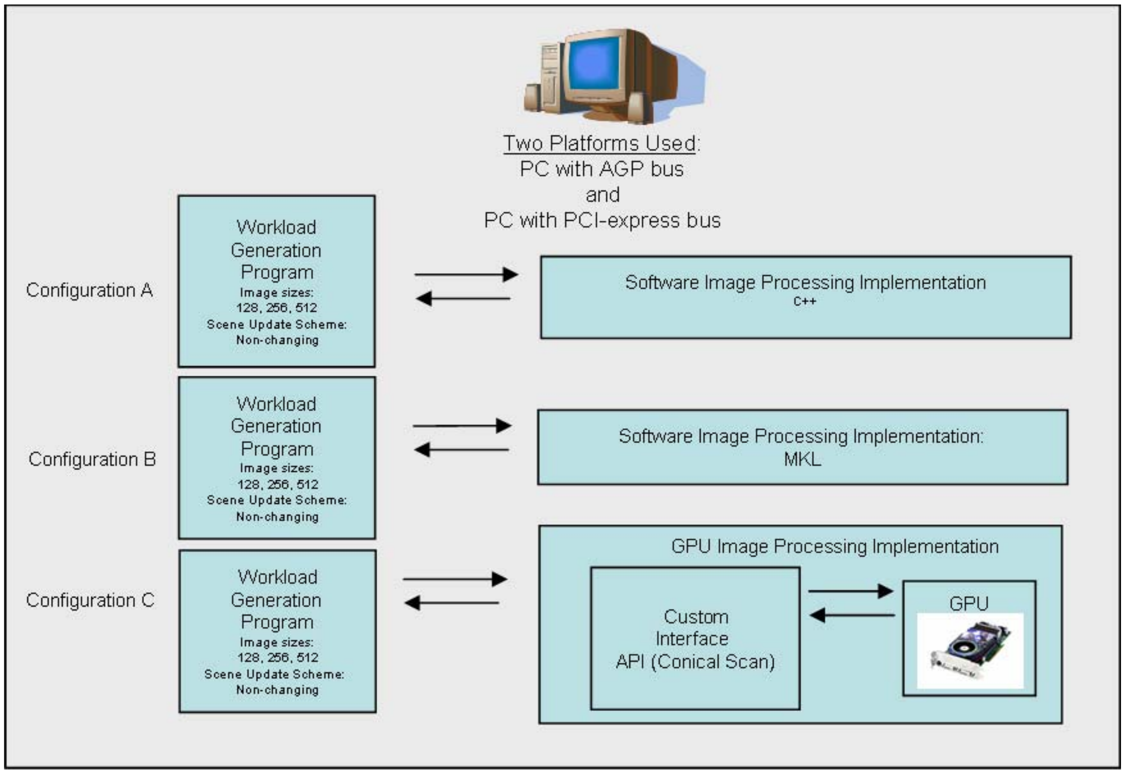

Platform: PC with AGP bus
Workload Generation Program Image size: 128, 256, 512
Custom Interface API (Spin Scan) GPU Implementation: Palette Sequential
ATI
Workload Generation Program Image size: 128, 256, 512
Custom Interface API (Spin Scan) GPU Implementation: Palette Sequential
nVidia

(a) System configurations for Phase 1 experiments. Spin scan implementation. Compares performance of ATI and nVidia GPUs under various workloads, scene update schemes and GPU algorithm implementations.

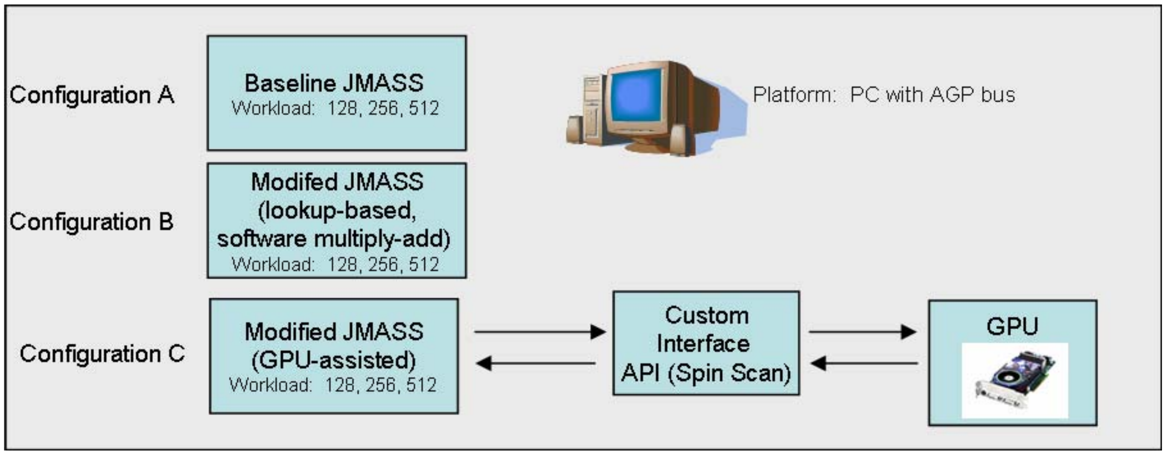

Two Platforms Used: PC with AGP bus and PC with PCI-express bus
Configuration A
Workload Generation Program Image sizes: 128, 256, 512 Scene Update Schemes: Non-changing, Fully-Changing, Moving Point Source
Software Image Processing Implementation C++
Configuration B
Workload Generation Program Image sizes: 128, 256, 512 Scene Update Schemes: Non-changing, Fully-Changing, Moving Point Source
Software Image Processing Implementation: MKL
Configuration C
Workload Generation Program Image sizes: 128, 256, 512 Scene Update Schemes: Non-changing, Fully-Changing, Moving Point Source
GPU Image Processing Implementation
Custom Interface API (Spin Scan)
GPU

(b) Three system configurations for the Phase 2 experiments. Spin scan implementation. GPU image processing performance compared to software-based implementations under different workloads and scene update schemes, on both AGP and PCI-express platforms.

Figure 4-1. System configurations for each phase of experiments. Figure continues on next page.


Two Platforms Used:
PC with AGP bus
and
PC with PCI-express bus
Configuration A
Workload Generation Program
Image sizes: 128, 256, 512
Scene Update Scheme: Non-changing
Software Image Processing Implementation
C++
Configuration B
Workload Generation Program
Image sizes: 128, 256, 512
Scene Update Scheme: Non-changing
Software Image Processing Implementation: MKL
Configuration C
Workload Generation Program
Image sizes: 128, 256, 512
Scene Update Scheme: Non-changing
GPU Image Processing Implementation
Custom Interface API (Conical Scan)
GPU

(c) Three system configurations for the Phase 3 experiments. Conical scan implementation. GPU image processing performance compared to software-based implementations under different workloads and scene update schemes, on both AGP and PCI-express platforms.


Configuration A
Baseline JMASS
Workload: 128, 256, 512
Platform: PC with AGP bus
Configuration B
Modifed JMASS (lookup-based, software multiply-add)
Workload: 128, 256, 512
Configuration C
Modified JMASS (GPU-assisted)
Workload: 128, 256, 512
Custom Interface API (Spin Scan)
GPU

(d) System configurations for Phase 4 experiments. JMASS baseline performance compared to Modified JMASS and GPU-assisted JMASS performance.

Figure 4-1 (continued). System configurations for each phase of experiments.

Table 4-1. Experimental designs for all phases of experiments.

Scene Update Scheme = Non-Changing				
	GPU: ATI		GPU: nVidia	
	Algorithm		Algorithm	
Size	Palette	Sequential	Palette	Sequential
128	exp1	exp3	exp5	exp7
256	exp2	exp4	exp6	exp8

Scene Update Scheme = Fully-changing				
	GPU: ATI		GPU: nVidia	
	Algorithm		Algorithm	
Size	Palette	Sequential	Palette	Sequential
128	exp9	exp11	exp13	exp15
256	exp10	exp12	exp14	exp16

(a) Phase 1a experimental design. Spin scan. ATI vs. nVidia

	GPU	
Update Scheme	ATI	nVidia
Non-Changing	exp1	exp3
Fully-Changing	exp2	exp4

(b) Phase 1b experimental design. Spin scan. ATI vs. nVidia. Image size = 512. Algorithm = Sequential.

Bus/Platform = AGP

Update Scheme = Non-Changing			
	Processing Method		
Size	GPU	C++	MKL
128	exp1	exp4	exp7
256	exp2	exp5	exp8
512	exp3	exp6	exp9
Update Scheme = Fully-Changing			
	Processing Method		
Size	GPU	C++	MKL
128	exp10	exp13	exp16
256	exp11	exp14	exp17
512	exp12	exp15	exp18
Update Scheme = Moving Point Source			
	Processing Method		
Size	GPU	C++	MKL
128	exp19	exp22	exp25
256	exp20	exp23	exp26
512	exp21	exp24	exp27

Bus/Platform = PCI-express

Update Scheme = Non-Changing			
	Processing Method		
Size	GPU	C++	MKL
128	exp28	exp31	exp34
256	exp29	exp32	exp35
512	exp30	exp33	exp36
Update Scheme = Fully-Changing			
	Processing Method		
Size	GPU	C++	MKL
128	exp37	exp40	exp43
256	exp38	exp41	exp44
512	exp39	exp42	exp45
Update Scheme = Moving Point Source			
	Processing Method		
Size	GPU	C++	MKL
128	exp46	exp49	exp52
256	exp47	exp50	exp53
512	exp48	exp51	exp54

(c) Phase 2 experimental design. Spin scan. GPU vs. Software-based Processing Methods.

Reticle Image Size = 128				Reticle Image Size = 256				Reticle Image Size = 512			
Bus/	Processing Method			Bus/	Processing Method			Bus/	Processing Method		
Platform	GPU	C++	MKL	Platform	GPU	C++	MKL	Platform	GPU	C++	MKL
AGP	exp1	exp3	exp5	AGP	exp7	exp9	exp11	AGP	exp13	exp15	exp17
PCI-e	exp2	exp4	exp6	PCI-e	exp8	exp10	exp12	PCI-e	exp14	exp16	exp18

(d) Phase 3 experimental design. Conical scan. GPU vs. Software-based Processing Methods. Non-changing scene update scheme.

	JMASS Version			
		Modified JMASS		
		Software	GPU-assisted version	
Size	Baseline	Version	ATI	nVidia
128	exp1	exp4	exp7	exp10
256	exp2	exp5	exp8	exp11
512	exp3	exp6	exp9	exp12

(e) Phase 4 experimental design. Baseline vs. Modified JMASS Software and GPU-assisted versions.

as well, each experiment was repeated 30 times, back to back. The second experimental design separately tests the 512^2 image size case. Though also full-factorial, there are only two factors that can be varied: GPU and scene update scheme.

The full-factorial design tests all possible combinations of the factors, and identifies the configuration that yields the best performance. Replication provides more samples than single trials and allows the estimation of experimental error. Knowing experimental error is advantageous because it isolates the error attributable to unknown sources from the error produced by the factors under test. Therefore, confidence intervals can be calculated for the effects and provide a qualitative indicator of the validity of the experimental design. There are $2^k(r - 1) = 464$ degrees of freedom in the mean squared error calculations for the first design, and 116 for the second design. Since there are greater than 30 degrees of freedom, confidence intervals for the effects are determined using quantiles of the unit normal distribution. There are 20 total experiments in this phase. Table 4-1(a and b) provides templates for this experimental design, Figure 4-1(a) shows the system configurations used in the experiments.

In the second phase of experiments, the best-performing GPU from the first phase is retained to be tested against both C++ and MKL software implementations of the JMASS spin scan reticle-scene multiply-add operation. The intent of this phase is to compare GPU hardware-accelerated image processing performance with that achievable using software. In these experiments, JMASS is not used. Instead, the three implementations are treated as stand-alone subsystems which emulate the JMASS image processing function. A test workload is submitted for processing 1,000 times, and the total execution time measured. The experimental design is four-factor, full factorial with replication. The factors (levels) used in these experiments are: processing method (GPU, C++, MKL); bus/platform (AGP, PCI-express); scene update scheme (non-changing, fully-changing, moving point source); and image size (128^2, 256^2, 512^2). Each experiment is conducted 30 times, providing 1566 degrees of freedom in the mean squared error calculations. Since there are more than 30 degrees of freedom, confidence intervals for the effects are determined using quantiles from the unit normal distribution. There are 54 total experiments in this phase. Table 4-1(c) provides a template for this experimental design, Figure 4-1(b) shows the system configurations used in the experiments.

A third phase of experiments compares the performance of the GPU against CPU-based approaches for performing the conical scan variation of the JMASS image processing calculations on both AGP and PCI-express platforms at three reticle image sizes: 128^2, 256^2, and 512^2. For these experiments the non-changing scene update scheme is used, and the experiments are broken into three subsets according to reticle image size, and analyzed as three separate designs (for rationale, see Chapter V). Each design is two-factor, full-factorial, with the following factors (levels): processing method (GPU, C++, MKL) and bus/platform (PCI-express, AGP).

As in the previous phases, each experiment consists of running the algorithm 1,000 times and measuring the total execution time, and 30 replications were accomplished for each experiment. Conical scan, however, requires more input parameters: a list of 40 reticle indices and shift offsets. These were chosen at random prior to each experiment using the C++ *rand* command. A different seed was used for each experiment, drawn from a uniform distribution between zero and 4,294,967,295. The seeds were generated using the Matlab *rand* command. Conical scan also requires the scene image be larger than the reticle image. For these experiments, the scene image dimensions were chosen to be twice those of the reticle image, resulting in the scene having four times the number of pixels as the reticle. As in the previous phases, confidence intervals for the effects are based on the unit normal distribution. There are 18 total experiments in this phase. Table 4-1(d) provides a template for this experimental design, Figure 4-1(c) shows the system configurations used in the experiments.

In the fourth and final phase of experiments, the two graphics cards are integrated with JMASS, and the performance of GPU-assisted JMASS is compared to that of baseline JMASS. The intent of this set of experiments is to determine whether, and to what extent GPU hardware acceleration can speed up JMASS simulations. Experimental design in this case is a two factor, full factorial experiment without replication (i.e., each experiment was conducted once).

The first factor is the JMASS software version, consisting of the following four levels: baseline JMASS, Modified JMASS (Software), Modified JMASS using the nVidia card for acceleration, and Modified JMASS using the ATI card. The last two levels are also referred to as "GPU-assisted JMASS" throughout this document. Modified JMASS (Software) is an improved version of baseline JMASS, implementing a lookup based approach for the reticle images that eliminates the rotation, resizing and interpolation operations (refer to the beginning of this chapter, and in Chapter V, for more details). Modified JMASS (Software) processes the reticle-scene multiply-add operations in software, analogous to the "C++" implementation in the Phase Two experiments. The GPU-assisted version of Modifed JMASS is the same as Modified JMASS (Software), except the reticle-scene multiply-add operation is performed in GPU hardware. The second factor is image size, with the same three levels used in other experiments: 128^2, 256^2, 512^2.

Configuring JMASS, integrating the GPU code and interpreting the JMASS results required the assistance of JMASS subject matter experts. Only one replication was performed for each experiment because the experiments take so long to run (almost two hours for the 512^2 case), and because running them required travel to an out-of-state contractor facility, so the time for conducting the experiments was limited. Though this single-replication experimental design precludes conducting an analysis of variance, it should be sufficient for the purposes of estimating likely speedup resulting from GPU acceleration. There are 12 total experiments in this phase. Table 4-1(e) provides a template for this experimental design, Figure 4-1(d) shows the system configurations used in the experiments.

Analysis of Results

The full-factorial designs described above permit a comprehensive analysis of results. Because the effects of processors and workloads interact in a multiplicative fashion [Jai91], a multiplicative model, using a log-transform of the execution time results is used for the analyses. The analysis model assumes that errors in the experimental results are normally distributed, and there is no trend in variance with respect to mean responses. Normal quantile-quantile plots of the errors, and plots of the errors with respect to the mean responses are therefore used to validate use of the multiplicative model where applicable. The mean effects of all factors and their associated levels are computed, and a 90% confidence interval given for each. The same applies for all possible combinations (i.e., interactions) of factors/levels. The mean effects for each level of a factor are used to determine the average relative speedup (or slowdown) that results when one level is chosen over another. Effects that are statistically significant have confidence intervals that do not include zero. An effect whose confidence interval contains the mean of another effect indicates statistically identical performance. In such cases, increasing the number of replications or decreasing the confidence level (narrowing the confidence interval) may permit the effects to be distinguished.

In addition to determining the confidence intervals for all effects, each factor, and interaction of factors, is examined to determine its contribution to the total variation of results (a.k.a. "allocation of variation"). Those factors which contribute the most to the total variation generally have the greatest practical impact on performance. The statistical significance of each factor can be further verified by performing an analysis of variance, or "F-test", using a 90% confidence level. For a factor to be considered significant, its contribution to the variance of results must exceed that of the estimated experimental error for the respective degrees of freedom.

For the first phase of experiments, which compares the performance of the two graphics cards under various workloads using the two GPU algorithm implementations, the analysis techniques described above are used to determine which factors/levels have the greatest impact on performance. In addition, the two graphics cards are contrasted to determine which provides the best average performance.

For the second and third phase of experiments, which compare GPU performance to software-based alternatives for accomplishing the JMASS spin scan and conical scan procedures, a similar analysis is performed to determine significant factors, and to contrast the performance of the three implementations.

For the fourth phase, which tests GPU-assisted JMASS (using both ATI and nVidia cards) against baseline JMASS and Modified JMASS (Software), the performance of the four implementations is contrasted to determine whether, and to what extent, GPU acceleration improves JMASS performance. Additionally, using insight from the previous phases of experiments and Amdahl's performance equation, it is possible to determine the maximum JMASS speedup achievable with GPU acceleration.

Summary

The experiments described herein are designed to definitively address the research goal, which is to determine whether, and to what extent GPU hardware acceleration can be used to improve JMASS execution time. In addition, the results of this research provide valuable insight as to how GPU algorithm implementations, scene update schemes and bus technologies affect GPU performance in the accomplishment of certain general-purpose computing tasks. Finally, the comparison of GPU-accelerated image processing performance with that of non-optimized code and code using a cache-optimized linear algebra library will likely provide AFIWC new alternatives for optimizing the existing JMASS software, even if the GPU fails to be a good option.

V. Results and Analysis

Introduction

Experiments are conducted in four phases. In all but the fourth phase, where tests were actually run using JMASS, experiments consist of calling the optics processing algorithm 1,000 times and recording the total execution time. Recall that a single iteration performs forty reticle-scene multiply-add operations (equivalent to performing a dot product on forty pairs of vectors, with each vector containing the same number of elements as there are pixels in the scene or reticle image), and returns the forty results in an array back to the calling application. It follows that for 1,000 iterations, each experiment results in 40,000 reticle-scene multiply-add operations. For reference, this is the amount of optics processing that occurs in JMASS during a simulated 4-second engagement. In those experiments involving a GPU, the execution time includes both the GPU processing time, plus the time spent transferring data into and out of the GPU. Each experiment was repeated 30 times. Results, analysis of variance and allocation of variation for each phase are included in Appendix A. Analysis was performed using a log-transform of the execution time results (cf., Chapter IV). An analysis of each phase follows.

Phase One Experiments: ATI Versus nVidia

This phase compares the performance of the nVidia and ATI graphics cards executing the JMASS spin scan optics calculations. Since a PCI-version of the nVidia card was not available, only the AGP versions of the cards are compared. The test platform for these experiments was a 3.0 GHz P4 (HT) with 875P chipset and 1GB RAM, running Windows XP Professional (SP2) and DirectX 9.0c. The factors varied were: GPU (nVidia, ATI), image size (128^2, 256^2, 512^2), GPU algorithm implementation ("Palette" and "Sequential"), and scene update scheme (non-changing, completely-changing). However, because the "Palette" GPU implementation does not run correctly on the AGP platform at the 512^2 resolution, the subset of experiments involving the 512^2 image size are analyzed separately to prevent skewing the results. For this subset, there are only two factors: GPU and scene update scheme.

For the subset of experiments involving 128^2 and 256^2 image sizes, analysis of variation (Table A-1a) indicates that all of the effects and interactions, except for the interaction between algorithm and scene update scheme, are statistically significant. This is due to the small amount of variance in the experimental results—the graphics cards seem to be very consistent in their execution times—resulting in 90% confidence intervals that are orders of magnitude smaller than the mean effects in most cases. Analysis of variance for the 512^2 image size experiments (Table A-1b) yields similar results. Except for a few outliers, normal quantile-quantile plots of the errors (Part 1 of Figures A-1a and b) are reasonably linear, satisfying the analysis model constraint that errors be normally distributed. Plots of errors versus mean response indicate no trend in variance with respect to response, satisfying the remaining model constraint (Part 2 of Figures A-1a and b). F-test results for $2^k r$ designs indicated statistically significant results with respect to experimental error [Jai91] Although the statistical F-test is not discussed further in this research, it was in fact passed in all cases of interest: the ratio of the mean-square value of any given effect to the mean-squared error is generally greater than 1,000 (mean-squared error is on the order of 10^{-6} in all experiments), which is much greater than any F-distribution percentile for the ratio, given the relatively large degrees of freedom of the error compared to the effects.

Though most of the effects and interactions are statistically significant, only a few turned out to be of practical importance. For the 128^2 and 256^2 experiments, allocation of variation (Table A-1a) indicates that over 61% of the variation is attributable to the choice of GPU, 35% to image size, and nearly 2% to an interaction between GPU algorithm and image size. Each of the remaining effects and interactions, including GPU algorithm and scene update scheme, account for less than 1% to the total variation, and are unimportant for practical purposes. For the 512^2 subset of experiments, almost 100% of the variation is due to choice of GPU. The effects of the scene update scheme factor and its interaction with the GPU account for much less than 1% of the total variation, and so are unimportant. Since varying the scene update scheme made little difference in these experiments, the examples and discussion that follow only address the non-changing scene update scheme case. The non-changing scene update scheme carries out no processing between calls to the GPU, resulting in execution times that more purely reflect the actual GPU processing time.

Effect of GPU

Performance in these experiments for all the image sizes was most dramatically affected by the choice of GPU. On average, the ATI card performs 4.8 times faster than the nVidia card when processing 128^2 and 256^2 image sizes. This can be derived from the analysis results for these experiments shown in Table A-1a, where the mean effect of the GPU is shown to be -0.3424. Since a multiplicative model using a log-transformation of the data is used, this figure means the ATI card performs the experiment in $10^{-0.3424}$ =

0.45 the time of the average GPU, given an average image size, scene update scheme and GPU algorithm implementation. Similarly, the nVidia GPU requires $10^{0.3424} = 2.2$ times the execution time of the average GPU under average conditions to accomplish the same calculations. Since the execution time of the mean GPU is 1 / 0.45 = 2.2 times that of the ATI card, and the nVidia execution time is 2.2 times that of the mean GPU, the nVidia execution time is therefore $2.2^2 = 4.8$ times that of the ATI card, on average.

A similar approach can be used to compare the two graphics cards for the 512^2 image size case. Per Table A-2a, the mean effect with respect to choice of GPU indicates the ATI card provides a full 5.0 times speedup over the nVidia card for the JMASS optics calculations at the 512^2 resolution.

One may intuitively validate the above GPU comparisons by simply using the (non-transformed) mean execution times to do a case-by-case comparison of the GPUs. Table 5-1 shows the mean execution times for the Phase One experiments, and Figure 5-1 shows the same information graphically. Dividing the nVidia execution time by the ATI execution time for a given image size and algorithm implementation yields speedup (ATI over nVidia) in the range of 3.84 – 6.98. Speedup figures for all applicable combinations of image size and GPU algorithm appear in Table 5-2.

In these experiments, the ATI card was consistently and significantly faster than the nVidia card. Such a performance disparity between these particular cards was noted by [BFH04] and is most likely attributable to the fact that the nVidia card carries out floating point operations at full IEEE-754 floating point precision, while ATI card does not. Though the ATI card supports the IEEE-754 format, it only implements 24 of the required 32 bits per float (16 bits matissa, 7 for exponent). ATI therefore likely trades

Table 5-1. GPU execution time, in seconds, for performing 1,000 iterations of the JMASS spin scan optics processing calculations, or equivalently, performing a dot product on 40,000 pairs of vectors whose dimension is indicated in the Image Size column. Times shown are for all applicable combinations of GPU, image size, and GPU algorithm combinations, using non-changing scene update scheme. Accompanying figure shows the same information graphically. The ATI card is faster (up to 7x) than the nVidia card in all cases. For the ATI card, the "Palette" approach provides slightly improved times over the "Sequential" approach at 128^2 and 256^2 image sizes. For the nVidia card, the "Palette" approach was best for the 128^2 size, and the "Sequential" approach was best for the 256^2 image size. For the 512^2 image size, only the "Sequential" approach is used because the "Palette" approach does not work correctly on the AGP platform.

GPU Execution Time (seconds) Spin Scan Procedure

	ATI		NVIDIA	
	GPU algorithm		GPU Algorithm	
Image Size	Palette	Sequential	Palette	Sequential
128^2	0.640	0.870	2.456	4.216
256^2	2.059	2.199	14.377	10.124
512^2	NA	7.226	NA	37.427

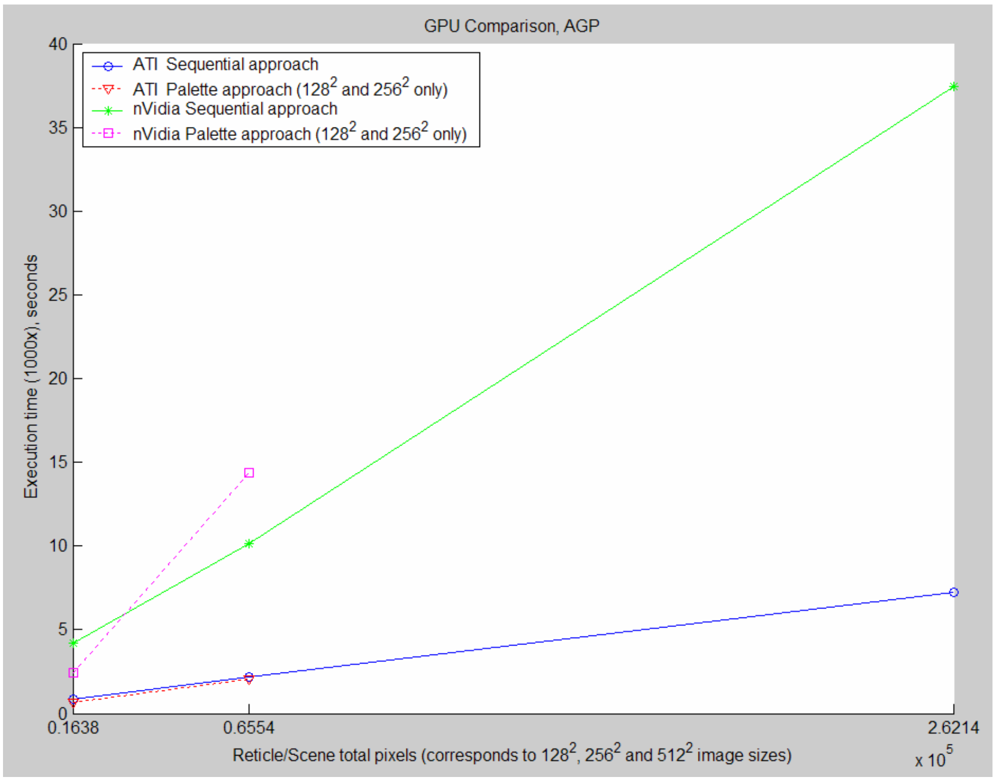


Figure 5-1. Graphical depiction of the data in Table 5-1, comparing nVidia and ATI GPU execution times for the three image sizes, using Palette and Sequential GPU algorithm implementations. The Palette approach provides slightly better performance over the Sequential approach for the ATI card at the 128^2 and 256^2 image sizes.

Table 5-2. Comparison of ATI and nVidia graphics cards showing relative speedup provided by ATI over nVidia for executing the JMASS spin scan image processing calculations. ATI is faster than nVidia in all cases.

ATI Speedup over nVidia

	GPU algorithm	
Image Size	Palette	Sequential
128^2	3.84	4.85
256^2	6.98	4.60
512^2	NA	5.18

speed for precision, and this is reflected in the accuracy of computed results. While testing the JMASS algorithm on the ATI GPU, a result (of an element-by-element multiplication of two images, then summation) on the order of 10^{12} can fall short of the correct answer by as much as 0.016% due to floating point truncation. The nVidia card is more accurate, yielding error about one-twentieth that of ATI. The impact of this error on JMASS simulations is discussed later in this chapter.

Effect of image size

For the subset of experiments involving the 128^2 and 256^2 image sizes, the 256^2 case took, on average, 3.3 times more time to execute than the 128^2 case. Note that although the workload increases by a factor of four when moving from the 128^2 to the 256^2 image size, the execution time increases by a lesser factor, indicating the GPU performs better when the larger image size is used, on average.

Effect of GPU algorithm implementation

For the subset of experiments involving the 128^2 and 256^2 image sizes, the GPU algorithm interacts with the image size factor to contribute about 2% of the total variation. Though not very significant in terms of overall performance, the mean effects for this interaction (Table A-1a) indicate that, on average, the “Palette” approach performs better with 128^2 images, and the “Sequential” approach performs better with 256^2 images. For the ATI card specifically, the “Palette” approach performs slightly better than the “Sequential” approach for both the 128^2 and 256^2 image sizes. For the nVidia card, results are mixed, with the “Palette” approach being best for the 128^2 image size, providing a 1.7x speedup over the “Sequential” approach, and the “Sequential” approach being better for the 256^2 image size, providing a 1.4x speedup over the “Palette” approach. The bottom line is the best choice for the algorithm depends on which graphics card and image size one intends to use. The fact the two graphics cards respond differently to the two approaches is most likely attributable to their differing internal architectures—the details of which are proprietary.

The single configuration that maximizes performance of the average case is the “Palette” approach for the 128^2 images, and the “Sequential” approach for all others. The “Palette” approach provides a 1.15x speedup over the “Sequential” approach for both 128^2 and 256^2 image sizes, while using the “Palette” approach in conjunction with the 128^2 image size (or using the “Sequential” approach in conjunction with the 256^2 image size) provides an additional 1.32x speedup over other combinations.

Useful work performed

When comparing the two GPUs, the concept of “useful work” supplements this analysis. Useful work is a measure of a processor’s effective rate for performing the floating point calculations required by the user, independent of implementation. Consider that each reticle-scene multiply-add operation requires $size^2$ floating point multiplications, plus $size^2-1$ additions, with *size* being the image width in pixels (128, 256 or 512). Thus, the amount of useful work performed in each experiment is:

$$\underset{(FLOPS)}{\textit{useful work}} = \frac{\textit{1,000 iterations x 40 reticle-scene operations/iteration x (2 size}^2\textit{-1) FP operations}}{\textit{execution time}}$$

This metric is specific to this application, and provides insight into the efficiency and suitability of the processing method under consideration.

Table 5-3 compares the useful work performed by the two GPUs for the Phase One experiments. The entries in the table correspond to the execution times listed in Table 5-1. In viewing the useful work figures, keep in mind the ATI card does not process at full

Table 5-3. Useful work, in GFLOPS, performed by the ATI and nVidia graphics cards at various image size and GPU algorithm combinations, using non-changing scene update scheme. Figures represent the number of useful floating point calculations performed per second in accomplishing 1,000 iterations of the JMASS spin scan optics processing calculations, or equivalently, performing a dot product of 40,000 pairs of vectors whose dimension is indicated in the Image Size column.

Useful Work Performed by GPU (GFLOPS)

	ATI		NVIDIA	
	GPU algorithm		GPU Algorithm	
Image Size	Palette	Sequential	Palette	Sequential
128^2	2.0	1.5	0.53	0.31
256^2	2.5	2.4	0.36	0.52
512^2	NA	2.9	NA	0.56

floating point precision, so comparing useful GFLOPS between the two cards is only valid if one accepts ATI's floating point limitations.

The ATI card's best times for performing the experiments were 0.640, 2.059 and 7.226 seconds for the 128^2, 256^2 and 512^2 image sizes, respectively, achieving rates of useful work between 2.0 and 2.9 GFLOPS. In contrast, the nVidia card's best times for these experiments were 2.456, 10.124 and 37.427 seconds for the 128^2, 256^2 and 512^2 image sizes, with corresponding useful work rates between 0.52 and 0.56 GFLOPS. Note that in all but one case, for any given GPU and algorithm combination, the rate of useful work increases with image size, which is consistent with the interpretation of experimental results presented earlier in this chapter. Such behavior is to be expected because larger image sizes result in a higher proportion of the total execution time being spent in actual GPU processing, versus transferring data into and out of the graphics card.

Applying terminology from the literature, processing larger image sizes increases *computational intensity*, enabling the GPU to be used more efficiently.

Phase Two Experiments: GPU Versus CPU-based Implementations

The second phase of experiments compares GPU performance with that of two alternative, CPU-based, processing methods for accomplishing the JMASS spin scan optics calculations: a C++ software implementation, and C++ code using the cache-optimized Intel Math Kernel Library (MKL). The factors are: processing method (GPU, C++, MKL), bus/platform (AGP, PCI-express), scene update scheme (non-changing, fully-changing, moving point source), and image size (128^2, 256^2, 512^2). Though the scene update scheme had little effect in the first phase of experiments, which involved only the GPUs, the factor is retained for this phase because of its potential to affect the CPU-based implementations. For these experiments, the ATI card is used as the representative GPU since it proved to be consistently faster than the nVidia card. For the GPU algorithm, the "Palette" approach was used for the 128^2 and 256^2 image sizes because it yields slightly better performance than the "Sequential" approach does on the ATI card. The "Sequential" approach was used for the 512^2 image sizes because it is the only approach that works on both AGP and PCI-express platforms at the 512^2 resolution. Tests are conducted on AGP and PCI-express platforms, using the AGP and PCI-express versions of the ATI X800XT graphics card. The 3.0 GHz P4 machine from the first phase of experiments serves as the platform for the AGP experiments, and a 3.6 GHz P4 (HT) with 925X chipset and 4GB RAM, running Windows XP Professional (SP2) and DirectX 9.0c is used for the PCI-express experiments. Though the two machines do not exactly facilitate an “apples-to-apples” comparison, it is shown that the differences between these two platforms had little effect on the execution times of these experiments.

Consistent with the previous phase of experiments, experimental error was very small, yielding 90% confidence intervals orders of magnitude smaller than the mean effects in most cases (see Table A-2a). The effects of the factors and their interactions are statistically significant, but only a few are of practical importance. Processing method (GPU, C++ or MKL) accounts for about 9% of the total variation; image size accounts for 89%; and the interaction between image size and processing method accounts for 1.2%. All other factors and interactions contribute less than 1% of the total variation, so may be considered unimportant for practical purposes. Except for a few outliers, normal quantile-quantile plots of the errors (Figure A-2a, Part 1) are reasonably linear, satisfying the analysis model constraint that errors be normally distributed. Plots of errors versus mean response indicate no trend in errors with respect to response, satisfying the remaining model constraint (Part 2 of Figure A-2a).

Effect of scene update scheme

As in the first phase of experiments, scene update scheme is not a significant factor. Perhaps this is to be expected since the Pentium 4 level two cache (512K on the AGP platform, 1MB on the PCI-express machine) can easily hold one or more 128^2 or 256^2 images, and perhaps one 512^2 image (PCI-machine only), allowing the fully-changing scene update scheme to occur very quickly. From examining the execution times (Table A-2a), a fully-changing scene update scheme does little more than add about 3-4% more time to each experiment, and does not appear to affect any method, including the cache-optimized MKL implementation, any more than the others. The moving point source update scheme, which changes two scene pixels per update, produced almost identical results to the non-changing update method. Because scene update scheme has little effect on performance, the analyses and examples that follow only address the non-changing case.

Effect of processing method

In all cases, the GPU implementation ran faster than the MKL and C++ implementations, providing 1.7x and 2.5x speedup over the two, respectively, on average. These figures come from interpreting the mean effects computed in Table A-2a in the same fashion as was done in the previous phase. For the smallest image size tested (128^2), GPU speedup is less than this average, providing only 1.2x speedup over MKL, and about 2x speedup over C++. This is the closest the CPU-based approaches come to matching the speed of the GPU. As image size is increased the gap widens and the GPU provides an increasing performance advantage over the CPU-based approaches. This is shown in Table 5-4, which lists the relative speedup provided by the GPU on the two platforms at the three image sizes. Note the GPU generally has less of an advantage on

Table 5-4. Speedup provided by GPU over CPU-based methods.

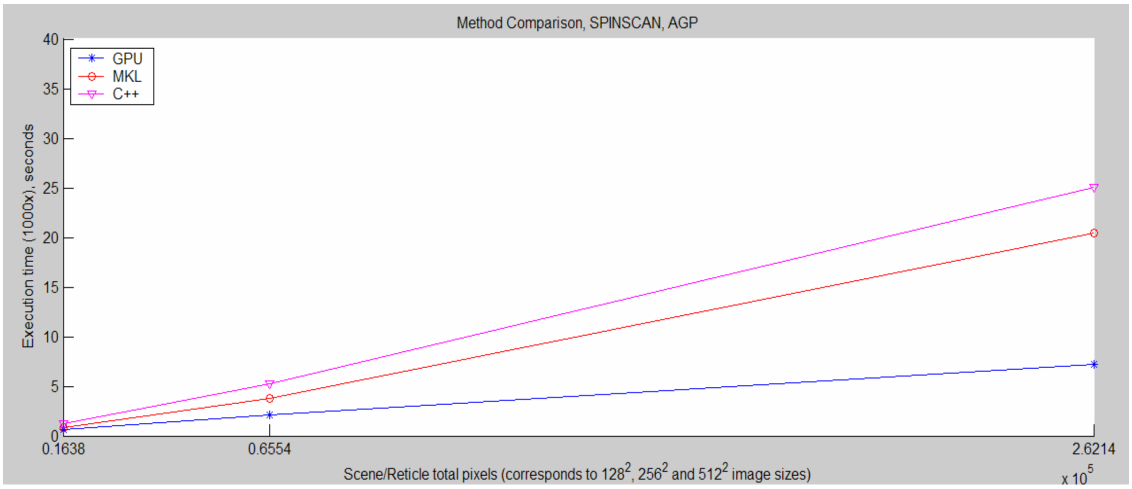
	AGP platform		PCI-express Platform	
Image Size	C++	MKL	C++	MKL
128^2	2.0	1.4	2.1	1.2
256^2	2.5	1.8	2.4	1.3
512^2	3.5	2.8	3.0	2.7

the PCI-express machine because the CPU-based approaches run their fastest on this machine, almost certainly due to its faster CPU. The speedup figures are computed by dividing the execution time of the CPU-based method by that of the GPU method for a given image size and platform. The complete list of execution times for this phase of

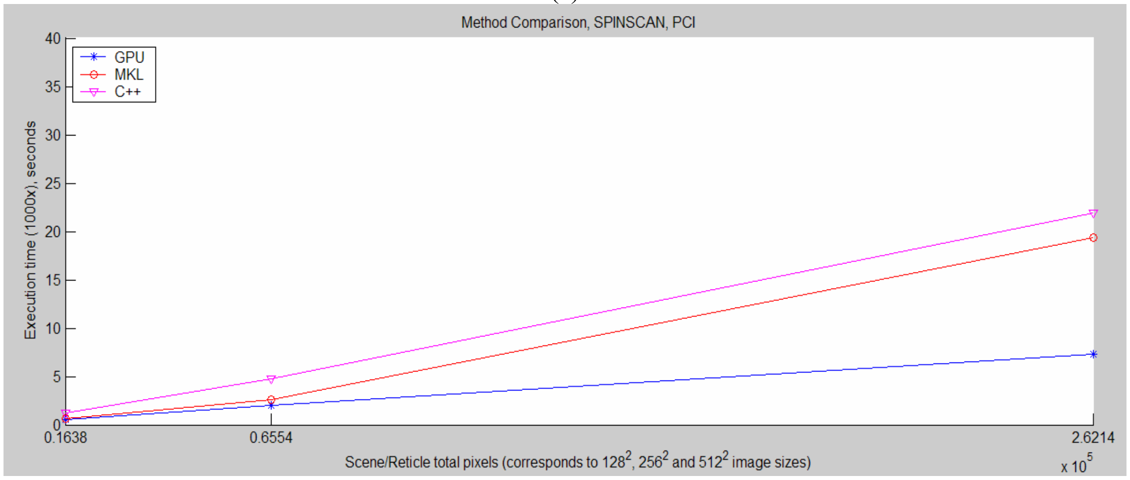
Table 5-5. Execution times, in seconds, compared for the GPU and CPU-based processing methods, at the three image sizes, and on both AGP and PCI-express machines. Times are for completing 1,000 iterations of the JMASS spin scan image processing calculations. Figure 5-2 shows the same information graphically.

Comparison of GPU and CPU-based Approaches
Execution Times (seconds) Spin Scan Procedure

	AGP platform			PCI-express Platform		
Image Size	GPU	C++	MKL	GPU	C++	MKL
128^2	0.640	1.267	0.876	0.576	1.199	0.664
256^2	2.059	5.234	3.776	1.980	4.787	2.645
512^2	7.226	25.032	20.448	7.283	21.885	19.409



(a)



(b)

Figure 5-2. Comparison of GPU and CPU-based processing methods in executing the JMASS spin scan procedure on the (a) AGP platform, and (b) PCI platform.

experiments is listed in Table A-2a. For convenience, the execution times also appear in Table 5-5, and are shown graphically in Figure 5-2.

Effect of platform

In this phase of experiments, the effect of graphics bus (AGP versus PCI-express) is confounded with the difference in CPU speeds and cache sizes of the two machines used. However, the analysis shown in Table A-2a indicates that the bus/platform factor accounts for barely 0.3% of the total variation, making the choice of platform almost irrelevant in these experiments. The mean effect for the platform/bus factor indicates that the PCI-express platform, with its faster graphics bus, CPU, and larger cache, provided a 1.14x speedup over the AGP platform, on average. Though the PCI-express bus provides a data path between the CPU main memory and GPU that is two times faster than AGP, only relatively small improvements in GPU execution time are observed on the PCI-express machine: 1.11x, 1.03x, 0.99x at the 128^2, 256^2 and 512^2 image sizes respectively. For the 512^2 image size, the AGP card yielded *better* performance than the PCI-express version, but only by a fraction of a percent. Note that the difference in GPU performance between AGP and PCI-express platforms diminishes as image size is increased. This is further evidence that larger image sizes allow the GPU to operate at higher levels of computational intensity, thereby reducing platform-specific impacts on the GPU processing time. At the 512^2 image size, the AGP and PCI-express graphics cards perform almost identically, despite the difference in CPU speed between the two platforms. This demonstrates that the GPU acts as an equalizer, allowing machines with slower CPU's to perform as fast (or faster) than machines with faster CPU's.

Interaction between processing method and image size

This effect accounts for little (about 1.2%) of the total variation, but illustrates the fact that the GPU is best used for the larger image sizes, compared to the CPU-based alternatives. From Table A-2a the effects of various combinations of method and image size result in slight penalties for using the GPU at the smaller image sizes, compared to the other methods, and slight gains for using the GPU at the 512^2 image size.

Effect of image size

The image size factor accounts for the greatest amount of variation (89%) in this phase of experiments. Unfortunately, this information is not particularly useful, since it is known that successive increases in image size represent fourfold increases in workload, and execution times vary widely with changing image size in these experiments. Since image size seems to overshadow the other factors in this set of experiments, some subsets of the experiments are analyzed to discover any trends that might otherwise have remained hidden.

The first subset to be analyzed considers only those experiments involving the non-changing scene update scheme. The analysis appears at Table A-2b, and is almost identical to that of the larger set of experiments, further confirming that the various scene update schemes have little effect on execution time.

If the above subset is further broken down, and a separate analysis performed for each image size case (Table A-2c), a trend with respect to the bus/platform factor appears. At the 128^2 image size, bus/platform accounts for about 6% of the total variation. As image size increases, this figure drops: to 4% at 256^2, and to less than 1% at the 512^2 image size. This would seem to indicate that as image size increases, the bus/platform factor becomes less significant, on average. This may make sense for the GPU case, but does not make sense for the C++ and MKL cases, whose performance is completely dictated by platform. A better interpretation of this trend arises if one considers that, for the 512^2 image size, the mean is most influenced by the MKL and C++ methods, whose execution times are both on the order of 20 seconds, compared to the GPU, whose execution time is on the order of seven seconds. In this case, the GPU execution time represents the greatest deviation from the mean, and so should be expected to dominate the analysis. Since GPU performance depends least on the bus/platform, it makes sense that the bus/platform factor would have less impact as image size increases and the GPU becomes more dominant.

For these subsets of experiments, particularly those involving the 128^2 and 256^2 image sizes, the interaction between bus/platform and processing method accounts for a greater share (2-3%) of the total variation than previously observed. This effect provides the greatest performance reward when MKL is combined with the faster platform, about a 1.1x speedup (best case) over the "average" combination of platform and processing method. This is simply because MKL methods run faster on the faster CPU, while the GPU performs almost the same, regardless of platform.

Phase Three Experiments: Conical Scan

This phase of experiments compares the performance of the GPU against CPU-based approaches for performing the conical scan variation of the JMASS image processing calculations on both AGP and PCI-express platforms. As in the previous phases, each experiment consists of running the algorithm 1,000 times and measuring the total execution time. Each iteration results in 40 reticle-scene multiply-add operations, for a total of 40,000 per experiment. Conical scan, however, requires more input parameters: a list of 40 reticle indices and shift offsets. These were chosen at random prior to each experiment using the C++ *rand* command. A different seed was used for each experiment, drawn from a uniform distribution between zero and 4,294,967,295. The seeds were generated using the Matlab *rand* command. Conical scan also requires that the scene image be larger than the reticle image. For these experiments, the scene image dimensions were twice those of the reticle image, so the scene contained four times the number of pixels as the reticle image. Taking a cue from the results of the previous phases, only the non-changing scene update scheme was used, and separate sets of experiments were conducted for each image size.

For this phase of experiments, there are two factors: processing method (GPU, MKL, C++) and bus/platform (AGP, PCI-express). Per the analysis shown in Table A-3, these two factors, and their interaction, are statistically significant for all image sizes. Except for a few outliers, normal quantile-quantile plots of the errors (Figure A-3, Part 1) are reasonably linear, satisfying the analysis model constraint that errors be normally distributed. Plots of errors versus mean response indicate no trend in error with respect to response, satisfying the remaining model constraint (Figure A-3, Part 2).

Allocation of variation and GPU speedup figures for the three sets of experiments in this phase are summarized in Table 5-6. Note that for the experiments involving the 128^2 and 256^2 reticle image sizes, the bus/platform factor accounts for a much larger percentage of the total variation than observed in previous phases of experiments. This is explained by the fact that, although the GPU remains faster on average than the CPU-based approaches, it does so by a smaller margin than was observed in the spin scan experiments (in fact, MKL on the PCI-express bus is faster than the GPU at the 128^2 image size). Since the GPU times are closer to those of the CPU-based methods, the effect of the platform is more apparent in these experiments.

Recall from Chapter III the GPU takes longer to execute the conical scan procedure for several reasons. First, shifting the images requires that a less efficient method be used for storing textures in GPU memory. Second, conical scan requires random access to the reticle images, forcing the use of the "Sequential" algorithm implementation, which, on the ATI card, is slower for the 128^2 and 256^2 image sizes. Third, extra time is needed in the vertex shader to add offsets to texture coordinates. Lastly, since the scene dimensions are twice those of the reticle image in these experiments, four times more scene data has to be uploaded to the GPU per iteration than with spin scan. With all that extra data being uploaded to the GPU, one might expect to observe improved GPU performance with the PCI-express bus. However, this is not the case. From Table 5-6, under "Speedup of PCI GPU vs. AGP GPU", it can be seen that the PCI-bus provides little more speedup for the GPU than it did in previous phases of experiments.

Table 5-6. Summary of GPU performance versus that of the CPU-based methods for the conical scan procedure. Allocation of variation for the effects of method and platform/bus are shown to indicate the relative importance of each factor as image size is increased.

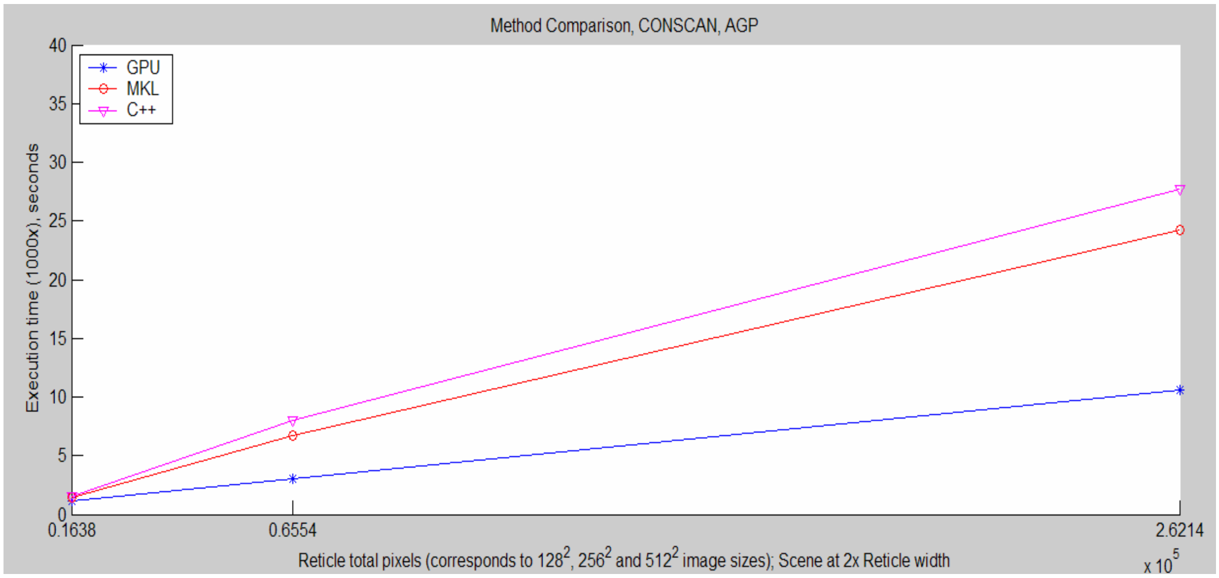
	GPU Speedup Over		Average Speedup of PCI Platform	Speedup of of PCI GPU	Allocation of Variation (%)		
Reticle Size	C++	MKL	over AGP	vs. AGP GPU	Method	Platform	Interaction of Platform & Method
128^2	1.3x	0.9x	1.3x	1.13x	34	53	13
256^2	2.3x	1.9x	1.3x	1.05x	86	11	3
512^2	2.5x	2.2x	1.1x	1.02x	99	1	0

The disadvantages described above seem to apply most to the two smaller image sizes. However, consistent with previous experiments, the relative speedup provided by the GPU increases as image size is increased, such that at the 512^2 image size, the GPU

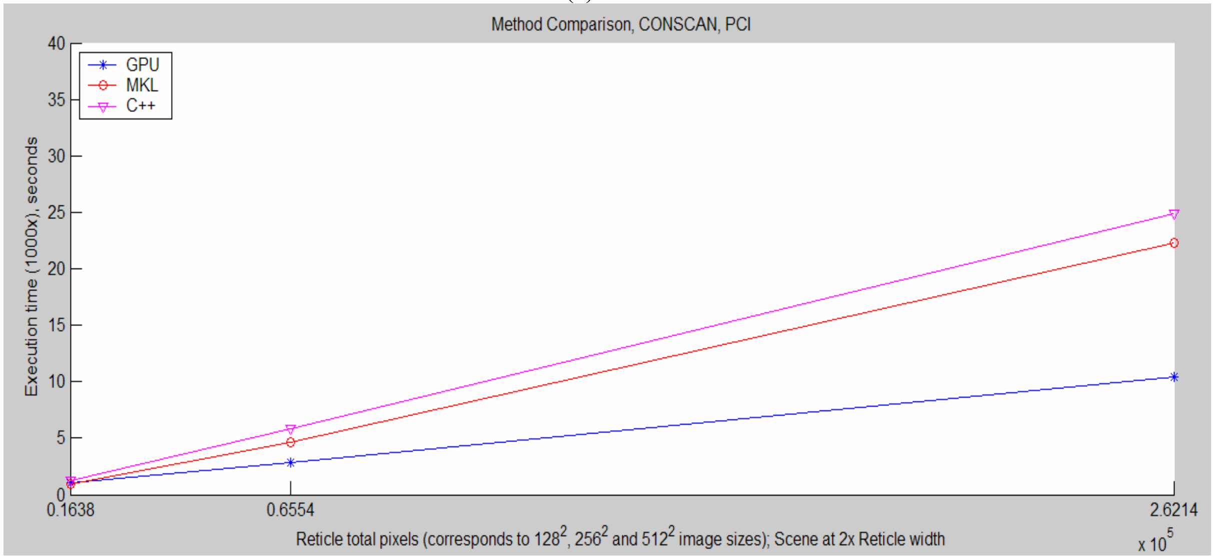
Table 5-7. Execution times, in seconds, compared for the GPU and CPU-based Processing Methods, at the three image sizes, and on both AGP and PCI-express machines. Times are for completing 1,000 iterations of the JMASS conical scan image processing calculations. For these experiments, the scene contains four times the number pixels as the reticle image. Accompanying figures show the same information graphically.

Execution Times (seconds) Conical Scan Procedure

	AGP platform			PCI-express Platform		
Reticle Size	GPU	C++	MKL	GPU	C++	MKL
128^2	1.138	1.505	1.414	1.012	1.221	0.942
256^2	3.037	7.971	6.688	2.889	5.796	4.650
512^2	10.639	27.759	24.235	10.430	24.920	22.256



(a)



(b)

Figure 5-3. Conical scan execution times compared for the GPU and CPU-based Processing Methods, at the three image sizes, and on both (a) AGP, and (b) PCI-express platforms. Plots show same information contained in Table 5-7 above.

provides a sizeable 2.2x speedup over MKL, and 2.5x speedup over basic C++ for the conical scan procedure.

Generally, *all* the methods were slower with conical scan than they were with spin scan. Compare the execution times for these experiments, shown in Table 5-7, with those of the Phase Two experiments. For the C++ approach, extra calculations are needed per pixel to index into the subset of the scene array overlapped by the reticle. MKL does not appear to provide an efficient means for performing the required operations on subsets of matrices, so instead of performing a single MKL *sdot* operation on the two images, the MKL routine computes starting indices for each row accessed in the scene array, performs a dot product on each row of the reticle and scene subset, and accumulates the results. This approach was still faster than basic C++, but by a smaller margin than observed in previous experiments.

For the 128^2 and 256^2 image sizes, the interaction between method and bus/platform accounts for about 13% and 3% of the total variation respectively, diminishing to below 1% for the 512^2 case. As in the Phase Two experiments, the effects of the various combinations of method and platform are explained by the fact that the GPU provides a higher margin of performance gain over the CPU-based methods when combined with the slower platform, and the opposite generally holds true when the CPU-based methods are combined with the faster processor. The impact of this interaction diminishes with increasing image size because total variation becomes dominated by the GPU, and GPU performance is little-affected by platform.

Phase Four Experiments: GPU Performance With JMASS

This phase of experiments compares the performance of baseline JMASS to that of GPU-assisted JMASS in running an actual JMASS simulation, specifically the JMASS generic Man-Portable Air Defense System (MANPADS) threat model, set for a 10 second engagement, at the three image sizes. Only the spin scan case was tested because integrating the GPU code for conical scan required extensive modifications to JMASS.

Two versions of JMASS were used, baseline JMASS and modified JMASS. Baseline JMASS is the version currently used by AFIWC, and the target of this, and other, hardware acceleration efforts. During each simulation time step, it performs costly image processing computations in software to simulate the missile's optical path: reticle image rotation and interpolation, and a reticle-scene multiply-add operation. Modified JMASS improves upon baseline JMASS by switching to a lookup-based approach for the reticle images, effectively eliminating thousands of repetitive rotation and interpolation operations, leaving only the reticle-scene multiply-add operation to be done on a repeated basis during the simulation. As a result of this research, it was mutually agreed upon with AFIWC that they should transition JMASS to this lookup-based approach, not only to support integration of GPU processing, but because it could improve JMASS performance even if GPU acceleration were not used. Hence, modified JMASS can run in either "GPU-assisted" or "Software" modes. The GPU-assisted version performs the reticle-scene multiply-add operation in GPU hardware, using the same GPU code implementation that was used in the Phase Two experiments. The Software version accomplishes the calculations in software, analogous to the "C++" processing method used in the Phase Two experiments. GPU-assisted JMASS was tested with both ATI and nVidia graphics cards. The platform was a 2.8 GHz Pentium 4 (HT) with 512MB RAM, running Windows XP Professional (SP1) and DirectX 9.0b. Only one replication of each experiment was conducted.

The results for these experiments appear in Table 5-8. From the table it can be seen that Modified JMASS, both the Software and GPU-assisted versions, outperform baseline JMASS in every case. From the analysis at Table A-4, the Software version provides about 1.4x speedup over baseline JMASS, and the GPU-assisted version provides about 1.5x speedup over baseline JMASS, on average.

Table 5-8. Execution times, in seconds, compared for original JMASS, modified JMASS and GPU-assisted JMASS, at the three image sizes. Each experiment consisted of running the JMASS generic MANPADS threat model, set for a 10 second engagement.

		Modified JMASS		
	Baseline	Multiply-add	GPU-assisted	
Image Size	JMASS	In Software	ATI	nVidia
128^2	579	407	360	359
256^2	2141	1574	1393	1411
512^2	8200	6289	5530	5525

Modified JMASS (Software) can be viewed as the first of two incremental improvements over baseline JMASS: it implements the more efficient lookup-based approach described above, providing 1.4x speedup over baseline JMASS, on average. The GPU-assisted version provides a second incremental improvement, enhancing the Software version by performing the reticle-scene multiply-add operation in GPU hardware. The GPU speeds up the Software version by about 1.1x, providing an absolute speedup over baseline JMASS of 1.5x, on average. Viewing the successive improvements in this manner reveals that the biggest performance gain for JMASS comes from transitioning to the lookup-based approach, and using the GPU to further optimize the reticle-scene calculations provides only a small additional benefit. This information is summarized in Table 5-9.

As indicated above, the GPU does not provide much of a performance boost to JMASS. The reason for this lies in the fact that Modified JMASS (Software version), by going to a lookup-based approach, does away with most of the time-consuming optics processing, namely the reticle rotation and interpolation operations. In so doing, the optics calculations become a much smaller contributor to the total JMASS execution time. This is shown in Table 5-9, which gives the estimated[4] proportion of JMASS

Table 5-9. Percentages of JMASS execution time spent performing optics versus other processing for the three versions of JMASS, and the speedup provided by these successive improvements. Optics processing includes reticle rotation and interpolation, and the reticle-scene multiply-add operations. Modified JMASS improves Original JMASS by essentially eliminating the rotation and interpolation operations via preprocessing and look-up, but continues to perform the recticle-scene multiply-add operation in software. GPU-assisted JMASS improves Modified JMASS by performing the multiply-add operation in GPU hardware. It can be seen that Modified JMASS, in switching to a look-up based approach, makes significant improvement to the optics processing. GPU-assisted JMASS provides further optimization to the optics processing, virtually eliminating it as a factor in the total JMASS execution time. These figures show that modifying JMASS to pre-process and look-up reticle images results in the largest improvement. Using the GPU to speed up the remaining reticle-scene multiply-add operation adds a further small improvement.

			Modified JMASS			
	Baseline JMASS		Software		GPU-Assisted	
	Other	Optics	Other	Optics	Other	Optics
	65%	35%	89%	11%	>99%	≤1%
Incremental Speedup over previous version		→	1.4x	→	1.1x	
Absolute Speedup over Baseline JMASS		→	1.4x	→	1.5x	

execution time attributable to optics processing versus other activities for the three tested JMASS versions. In baseline JMASS, optics processing accounts for about 35% of the total execution time, whereas in Modified JMASS (Software), it only accounts for 11%.

[4] Estimates derived using known GPU execution times and the JMASS execution times from Table 5-8. Example: for the 128^2 image size, the ATI GPU takes no more than 2 seconds to process the 2,425 spin scan iterations required during the simulation of a 10 second engagement. Subtracting 2 seconds from the 360 second GPU-assisted (ATI) JMASS execution time yields 358 seconds for all "other" processing. Dividing this number by the execution times of the JMASS versions at the 128^2 image size gives the fraction of the total time spent in this activity for each. Percentages shown in Table 5-9 are averages. Actual percentages for each image size case vary by +/- 3 percentage points. These estimates agree with results provided by profiler software, which indicate that the optics processing carried out in Modified JMASS (Software) accounts for about 10% of the execution time for that JMASS version.

Using Amdahl's famous equation, Modified JMASS (Software) provides about 3.8x speedup for the optics processing, compared to baseline JMASS. At this point, the best speedup attainable by further optimizing the optics processing is 1.12x, the speedup that would be gained by eliminating the optics calculations altogether. The GPU therefore performs admirably in these experiments, because it almost accomplishes this, with GPU-assisted JMASS reducing the optics processing time to 1% or less of the total JMASS execution time. Equivalently, the GPU provides speedup on the order of 10-40x (depending on the GPU and image size used) for the optics processing compared to Modified JMASS (Software). The end result of all the improvements is the elimination of about 35% of the baseline JMASS execution time, which is a significant improvement. Unfortunately, the majority of this improvement is due to the efficiency of the lookup-based approach, and not the GPU. The reticle-scene multiply-add operation does not account for enough of the total JMASS execution time for the GPU to make a big difference overall.

Somewhat puzzling in these results is the 10-40x speedup indicated for the GPU-assisted versus non-GPU versions of Modified JMASS. Recall that the only difference between the two versions is the method used for processing the reticle-scene multiply-add operation: GPU hardware, or software. The GPU speedup observed in these experiments is not in line with the results of the Phase Two experiments, in which the GPU yielded a maximum of about 7x speedup over the software-based implementation. Using profiler software, it was verified that the reticle-scene multiply-add operation in Modified JMASS (Software) accounted for about 10% of the total execution time, meaning that for some reason, it runs considerably slower than the functionally equivalent routine used in the

Phase Two experiments. One possible explanation for this difference is that JMASS represents the scene as a C++ object, containing an assortment of attributes and methods, versus using a simple array. AFIWC is investigating the cause of the apparent inefficiency. If the inefficiency can be overcome, and the Modified JMASS multiply-add routine can be made to run as fast as the one used in the Phase Two experiments, it is expected that the already small advantage provided by the GPU-assisted version will become even less significant, especially at the smaller two image sizes.

An inconsistency seems to exist in the results due to the small difference between the ATI and nVidia cases (see Table 5-8). Using known GPU times for executing the approximately 2,500 iterations required for simulating a 10-second engagement, the ATI and nVidia cards should be expected to differ in their execution times by approximately 5, 30, and 70 seconds at the 128^2, 256^2 and 512^2 image sizes respectively. However, the actual differences observed in JMASS execution time when using the different graphics cards were only 1, 17 and 5 seconds for the respective image sizes.

The simplest explanation for this disparity between expected and observed differences in execution time is that JMASS execution times can vary from run to run enough to mask the differences in GPU performance. In this case, a variation of 1-2% would be enough. However, the existence of such variance cannot be confirmed because only one replication of each experiment was performed.

Another possibility that was investigated is whether the graphics cards behave differently when there is significant time delay between calls to the GPU. In the first three phases of experiments, the GPU was tested by calling its processing algorithm 1,000 times, back to back, with almost no delay between calls. However, with JMASS, there can be more than two seconds between calls to the GPU. To see if this was a factor, some experiments were run with similar delays inserted between calls to the GPU. After running the experiment using different image sizes and delay times ranging from 0.1 to about two seconds, no differences were observed in GPU execution time. However, large fluctuations, sometimes over 10%, were observed in the delay times themselves, even when the GPU was completely removed from the experiment. Further investigation revealed that the variance in execution time of the delay loop generally increased when the size of the dummy array was increased, and most dramatically when it was increased so as to exceed the capacity of the CPU's level two cache. Though by no means conclusive, such variation in the execution of a simple loop makes it conceivable that similar variation could exist in the execution of a large and complex program like JMASS.

One other possible explanation exists for the above-noted inconsistency, having to do with the difference in the floating point precision of the two cards. Analysis of the JMASS output reveals that the simulated IR detector signals produced by the two graphics cards during JMASS simulation differ from each other, and from that produced by baseline JMASS. This comes as no surprise, since baseline JMASS uses double-precision while the GPU is limited to single-precision, or a subset thereof in the ATI case. Per an AFIWC subject matter expert, it is possible that such differences could cause the simulated missile to take longer to acquire or reacquire lock on the target, or to lose lock more often, resulting in longer simulations. Another example of the graphics cards producing different results lies in the "miss distance" displayed by JMASS at the end of the simulation, indicating the missile's final proximity to the target. Given the same simulation parameters, baseline JMASS produces miss distances of just over half a meter, nVidia just over a meter, and ATI about 3 meters. Because nVidia produces miss distances that are closer to those generated by baseline JMASS, it is considered more accurate. It is a possible concern that ATI's considerably larger miss distance could lead to falsely predicting a miss when a more accurate simulation would predict a hit. At this point, however, it is only known that these differences exist. The impact, if any, such differences might have on the outcome and validity of JMASS simulations remains to be established.

Summary

These experiments accomplished the research goals identified in Chapter IV. The first phase of experiments was designed to compare the candidate graphics cards, and a clear winner emerged. The ATI processor outperformed the nVidia GPU in all cases, providing an average 5x speedup over its rival. This advantage is somewhat unfair, however, because the ATI GPU cuts corners with respect to floating point precision, resulting in faster processing, but less accurate results. Though it is too early to tell, these inaccuracies may make this card unsuitable for the JMASS application. The ATI and nVidia GPUs sustained useful work rates of up to 2.9 and 0.56 GFLOPS respectively in these experiments, displaying formidable processing power—especially considering that these figures include the time spent transferring data into and out of the graphics cards.

The second phase of experiments pitted GPU hardware acceleration against software-based alternatives for implementing the JMASS spin scan reticle-scene multiply-add operation. Using the faster ATI graphics card as the representative GPU, GPU hardware consistently outperformed C++ and Intel Math Kernel Library software implementations, providing 1.4x to 3.5x speedup, with the GPU achieving its greatest advantage when processing the largest 512^2 image size.

The third phase of experiments compared GPU performance against the same software-based alternatives for executing the conical scan variation of the JMASS multiply-add operation. In all but one case, the GPU outperformed C++ and Intel Math Kernel Library implementations, providing 0.9x to 2.5x speedup.

The results of these experiments demonstrate that the GPU can indeed provide significant speedup over software-based alternatives for performing both the spin scan and conical scan variations of the JMASS reticle-scene multiply-add operation. However, as was forewarned in Chapter I, even the most spectacular GPU speedup could be expected to have little effect on JMASS system performance if the multiply-add operation were not to account for a significant amount of the total JMASS execution time. The fourth phase of experiments, which integrated GPU processing into JMASS, revealed exactly that. The full suite of optics calculations performed by baseline JMASS (rotate, interpolate and multiply-add) only accounted for about 35% of the total baseline JMASS execution time, which is much less than originally expected. Further, in order to integrate GPU processing into JMASS, JMASS was modified to use a lookup-based approach which eliminated the bulk of the optics computations. In this modified version of JMASS, only the reticle-scene multiply-add operation remained to be optimized, accounting for only 11% of the execution time. As described earlier in this chapter, the GPU provided excellent acceleration, reducing the time spent in the multiply-add operation so as to account for less than 1% of the total JMASS execution time, yielding close to the theoretical maximum achievable acceleration of 1.1x. The bottom line, with respect to JMASS, is that the GPU provided the best possible speedup given its frequency of use. The results of the first three phases of experiments indicate that the GPU could have a much greater impact, providing up to 3.5x speedup, in applications where the multiply-add operation accounts for the bulk of the execution time.

On a very positive note, though the original intent of transitioning JMASS to the lookup-based approach was to enable the integration of GPU processing, it resulted in a 1.4x speedup over baseline JMASS. With the inclusion of GPU processing, the overall speedup is increased to 1.5x. This equates to eliminating 40 minutes of a two-hour simulation.

VI. Discussion

Summary of Findings

This research demonstrates GPU hardware can support JMASS spin scan and conical scan simulations, performing the reticle-scene multiply-add operation up to 3.5x faster than software-based solutions including those that have been cache-optimized. The GPU advantage is greatest when processing larger image sizes, due to increased computational intensity, achieving useful work rates as high as 2.9 GFLOPS for this application. Two top-of-the-line consumer graphics cards, the ATI X800XT and nVidia 6800 Ultra, were tested, and the ATI card was five times faster on average than its nVidia counterpart in executing the JMASS multiply-add operation. However, the ATI card is also less accurate due to its reduced floating point precision, which may or may not impact the validity of JMASS simulation results.

This research resulted in a 1.5x speedup for JMASS by fostering its transition to a lookup-based approach for processing the reticle images which eliminates hundreds of thousands of unnecessary image transformation operations. This speedup is equivalent to eliminating 40 minutes of every 2-hour simulation, and therefore delivers a significant, immediate benefit to AFIWC.

Nevertheless, despite the speed increases afforded by the graphics cards for performing the JMASS image processing computations, GPU acceleration impact on overall JMASS performance does not reflect the speedup achieved by the GPU. This is not due to any problem with the GPU--the GPU executed the multiply-add operation up to 40 times faster than the JMASS program--but rather the multiply-add operation accounts for just a small portion of the total JMASS execution time, so optimizing it has a correspondingly small effect. The results of the first three phases of experiments indicate that the GPU could have a much greater impact, providing up to 3.5x speedup, in applications where the multiply-add operation accounts for the bulk of the total execution time.

Final Observations and Recommendations

Since JMASS only uses the GPU about 1% of the time for the multiply-add operation, the GPU can perform other JMASS processing as well. One such use is for IR scene generation. Graphics cards excel at rendering complex and dynamic 3D scenes, and so will be faster than the procedural methods currently used by JMASS to generate the scene images. Combining scene generation and multiply-add operations in the GPU is very efficient because the scene would reside natively in GPU memory, and would not have to be uploaded via costly data transfers to the GPU after every scene update.

Efforts to accelerate JMASS more using hardware should be focused on the portions of JMASS which have not been optimized (e.g., IR scene generation). Further effort and expense devoted to optimizing the JMASS optics calculations, including reticle image rotation and interpolation, and reticle-scene multiply-add (a.k.a. "convolution"), is not recommended since these now only account for 1% of the JMASS execution time when using the GPU (11% otherwise) and further optimization will yield no noticeable performance gain for JMASS simulations.

The GPU implementations developed in this research can be further optimized. The "Palette" approach used for spin scan can be made more efficient (cf., Chapter III) by not processing unneeded images at the top and bottom rows of the palette. In hindsight this inefficiency could be eliminated altogether by modifying the algorithm to take advantage of the fact that sequencing through consecutive groups of 40 reticle images, mod 100, returns to the initial group every five iterations. Thus, all needed reticle image orderings can be stored in five smaller palette textures, each containing 40 reticle images instead of 64. Since each palette is used in its entirety, there is no need to resize the drawing rectangle, and no processing of unwanted images. The smaller palette textures could also support the 512^2 image size within GPU memory constraints, whereas the current algorithm uses a more complicated and inefficient procedure to deal with the memory limitation for this image size. The proposed approach would therefore support all three image sizes with a more efficient, common algorithm.

Though designed specifically to support the JMASS image processing requirement, the GPU implementations developed for this research could, with some modifications, support any application that requires an abundance of image processing operations involving shifting and multiplying images, and reducing the results. However, the GPU hardware imposes some restrictions on expandability. In designing the GPU-based algorithms, the chief limitations were GPU memory capacity, maximum supported texture size, and the texture dimension and shape constraints imposed by shader programs.

Given the 256MB memory capacity of the GPUs, 512^2 is the largest reticle image size that can be supported if all 100 reticle images are to be stored in GPU memory. The next larger (power-of-two) image size, 1024^2, cannot be supported because 100 images of that size would require 400MB of GPU memory, and few, if any, graphics cards are so equipped.

All the implementations rely on large-sized textures for storing collections of images, such as those used for the reticle palettes and for storing intermediate results between rendering passes. This seems to be a GPU-efficient approach. However, once again 512^2 is the largest image size supported if a texture containing 64 images is desired. nVidia allows very large 4096^2 texture sizes, but actually creating a floating point texture of that size would use up the entire 256MB of available memory! Therefore, if future GPUs are improved to support larger textures, GPU memory size must also be increased for it to benefit this application.

Per Chapter III, pixel shader programs impose power-of-two dimension and square shape limitations under certain circumstances. These restrictions can force using larger textures than necessary, resulting in wasted GPU processing. Another limitation with respect to pixel shaders is the limited depth of dependent texture addressing supported. Dependent texture addressing allows texture coordinates which address one pixel to be used to derive the coordinates for another. Limiting this practice decreases the number of adjacent pixels that can be summed or multiplied during a rendering pass, and restricts the creativity of the programmer. Removing these restrictions could allow programmers to create more efficient algorithms.

This research has demonstrated that graphics cards can provide an impressive performance boost for a general computing application, provided the application lends itself to SIMD processing and can maintain high enough rates of computational intensity. It has further been shown that GPU acceleration can enable slower computers to meet or exceed the performance of faster and otherwise better-equipped machines. If GPU technology continues to improve as it has (and given the current state of the PC gaming industry there is no reason to expect otherwise), the limitations described above are not likely to exist for long, and the GPU could indeed become the processor of choice for many applications. In the meantime, the latest graphics cards, which support floating point operations and can be flexibly programmed via rich APIs and shader programming languages, are better prepared than ever to meet the demands of scientific, engineering and modeling and simulation applications.

This page intentionally left blank.

Appendix A. Analysis Tables and Figures

List of Tables

List of Figures

TABLE A-1a

Results and analysis of Phase One experiments. Compares nVidia to ATI GPU at 128 and 256 image sizes, using Palette and Sequential GPU algorithm implementations and non-changing and fully-changing scene update schemes.

Results, Mean Response, Execution Time (seconds)

Update Scheme		GPU A		GPU B		Update Scheme		GPU A		GPU B	
Non-chg		Algorithm		Algorithm		Fully-Chg		Algorithm		Algorithm	
	SIZE	PALETTE	SEQ	PALETTE	SEQ		SIZE	PALETTE	SEQ	PALETTE	SEQ
	128	0.640	0.870	2.456	4.216		128	0.667	0.896	2.457	4.248
	256	2.059	2.199	14.377	10.124		256	2.198	2.342	14.495	10.241

log-transform of data

Update Scheme		GPU A		GPU B		Update Scheme		GPU A		GPU B	
Non-chg		Algorithm		Algorithm		Fully-Chg		Algorithm		Algorithm	
	SIZE	PALETTE	SEQ	PALETTE	SEQ		SIZE	PALETTE	SEQ	PALETTE	SEQ
	128	-0.194	-0.060	0.390	0.625		128	-0.176	-0.048	0.390	0.628
	256	0.314	0.342	1.158	1.005		256	0.342	0.370	1.161	1.010

sum of squares

	GPU A		GPU B		GPU A		GPU B	
	Algorithm		Algorithm		Algorithm		Algorithm	
SIZE	PALETTE	SEQ	PALETTE	SEQ	PALETTE	SEQ	PALETTE	SEQ
128	1.131	0.110	4.567	11.714	0.928	0.069	4.572	11.838
256	2.953	3.513	40.205	30.322	3.509	4.096	40.452	30.624

Computation of Effects

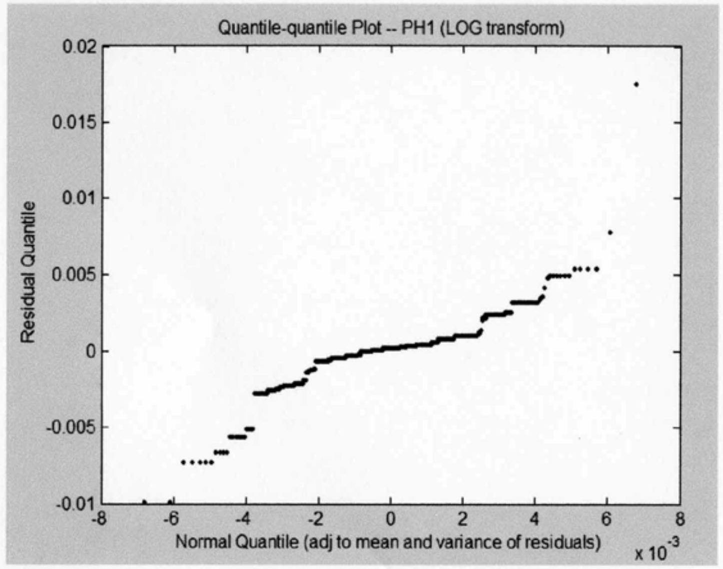
		GPU	ALG	Update Sch.	SIZE												
Exp	I	A	B	C	D	AB	AC	AD	BC	BD	CD	ABC	ABD	ACD	BCD	ABCD	mean
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-0.194
2	1	1	1	1	-1	1	1	-1	1	-1	-1	1	-1	-1	-1	-1	0.314
10	1	1	-1	1	1	-1	1	1	-1	-1	1	-1	-1	1	-1	-1	-0.060
11	1	1	-1	1	-1	-1	1	-1	-1	1	-1	-1	1	-1	1	1	0.342
4	1	1	1	-1	1	1	-1	1	-1	1	-1	-1	1	-1	-1	-1	-0.176
5	1	1	1	-1	-1	1	-1	-1	-1	-1	1	-1	-1	1	1	1	0.342
12	1	1	-1	-1	1	-1	-1	1	1	-1	-1	1	-1	-1	1	1	-0.048
13	1	1	-1	-1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1	0.370
23	1	-1	1	1	1	-1	-1	-1	1	1	1	-1	-1	-1	1	-1	0.390
24	1	-1	1	1	-1	-1	-1	1	1	-1	-1	-1	1	1	-1	1	1.158
29	1	-1	-1	1	1	1	-1	-1	-1	-1	1	1	1	-1	-1	1	0.625
30	1	-1	-1	1	-1	1	-1	1	-1	1	-1	1	-1	1	1	-1	1.005
26	1	-1	1	-1	1	-1	1	-1	-1	1	-1	1	-1	1	-1	1	0.390
27	1	-1	1	-1	-1	-1	1	1	-1	-1	1	1	1	-1	1	-1	1.161
31	1	-1	-1	-1	1	1	1	-1	1	-1	-1	-1	1	1	1	-1	0.628
32	1	-1	-1	-1	-1	1	1	1	1	1	1	-1	-1	-1	-1	1	1.010
tot	7.258	-5.4787	-0.4871	-0.0985	-4.1465	-0.1484	-0.0744	0.4555	-0.0021	-0.9814	0.0297	-0.0113	0.5699	0.0196	-0.0031	-0.0065	7.258
tot/16	**0.454**	**-0.3424**	**-0.0304**	**-0.0062**	**-0.2592**	**-0.0093**	**-0.0047**	**0.0285**	**-0.00013**	**-0.0613**	**0.0019**	**-0.0007**	**0.0356**	**0.0012**	**-0.00019**	**-0.0004**	**EFFECTS**
squared	0.206	**0.1173**	0.0009	0.0000	**0.0672**	0.0001	0.0000	**0.0008**	0.0000	**0.0038**	0.0000	0.0000	0.0013	0.0000	0.0000	0.0000	

TABLE A-1a (cont)

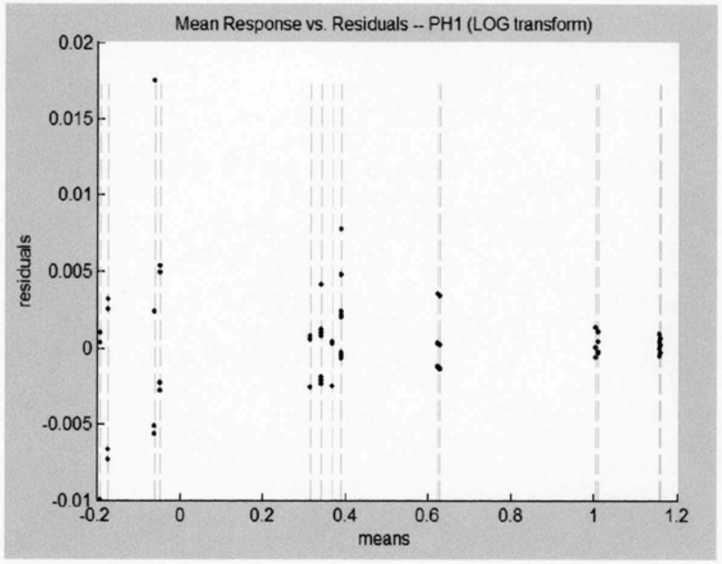
SSY =	**190.603**
MEAN=	**0.454**
SS0=	**98.761**
SST = SSY-SS0 =	**91.843**

Allocation of Variation					
	2^k*r*q^2	%variation		2^k*r*q^2	%variation
SSA=	**56.281**	**61.280**	SSBD=	**1.806**	**1.966**
SSB=	0.445	0.484	SSCD=	0.002	0.002
SSC=	0.018	0.020	SSABC=	0.000	0.000
SSD=	**32.238**	**35.102**	SSABD=	0.609	0.663
SSAB=	0.041	0.045	SSACD=	0.000	0.000
SSAC=	0.010	0.011	SSBCD=	0.000	0.000
SSAD=	0.389	0.423	SSABCD=	0.000	0.000
SSBC=	0.000	0.000			
tot =	91.840	99.997		SSE(est)	%var
				2.77E-03	0.003

ANOVA & SSE calculated from experimental errors				
variance (sum of Sqared Error/(r-1))				
7.49E-06	2.54E-05		2.66E-06	9.32E-07
1.78E-06	1.94E-06		5.20E-08	1.20E-07
2.19E-05	1.36E-05		1.91E-06	8.18E-07
2.92E-06	1.03E-06		6.42E-08	1.44E-07
		SSE	**2.40E-03**	sum of above * (r-1)
		MSE	**5.17E-06**	SSE/2^k(r-1)
st dev of errors		s_e	**2.27E-03**	MSE$^{1/2}$
st dev of effects		s_q	**1.04E-04**	$s_e/(2^k r)^{1/2}$
90% c.i.		z[.95]	1.65	
ci for effects		q+/- z*s_q	q+/-	**1.71E-04**



Part 1



Part 2

Figure A-1a. Normal quantile-quantile and errors-versus-responses plots for Phase 1a (ATI vs. nVidia, spin scan, 128 & 256 image sizes) experiments. Quantile-quantile plot is reasonably linear, except for a few outliers. Errors-versus-responses plot shows no trend. Analysis model is therefore valid.

TABLE A-1b

Results and analysis of Phase One experiments, 512 image size case.
Compares nVidia to ATI GPU at non-changing and changing scene update schemes.
SIZE=512, ALG= SEQUENTIAL

Results, mean responses, execution time (seconds)		
Scene Update Scheme	ATI	NVIDIA
non-chg	7.226	37.427
fully-chgng	7.842	38.010

log-transform of data		
Scene Update Scheme	ATI	NVIDIA
non-chg	0.859	1.573
fully-chgng	0.894	1.580

sum of squares		
	ATI	NVIDIA
WL1	22.131	74.247
WL2	24.001	74.883

Computation of effects					
	GPU	Update Scheme			
I	A	B	AB	mean	
1	1	1	1	0.859	
1	1	-1	-1	0.894	
1	-1	1	-1	1.573	
1	-1	-1	1	1.580	
4.906	-1.400	-0.042	-0.029	4.906	
1.227	**-0.350**	**-0.011**	**-0.007**	1.227	**EFFECTS**
1.505	0.122	0.000	0.000	1.505	squared

SSY	**195.261**
MEAN	**1.227**
SST	**14.714**

Allocation of Variation				
SS0	**SSA**	**SSB**	**SSAB**	**SSE**
180.547	**14.695**	**0.013**	**0.006**	**9.14E-06**

% variation explained				
A	**B**	**AB**	**Error**	**total**
99.867	**0.091**	**0.042**	**0.000**	100.000

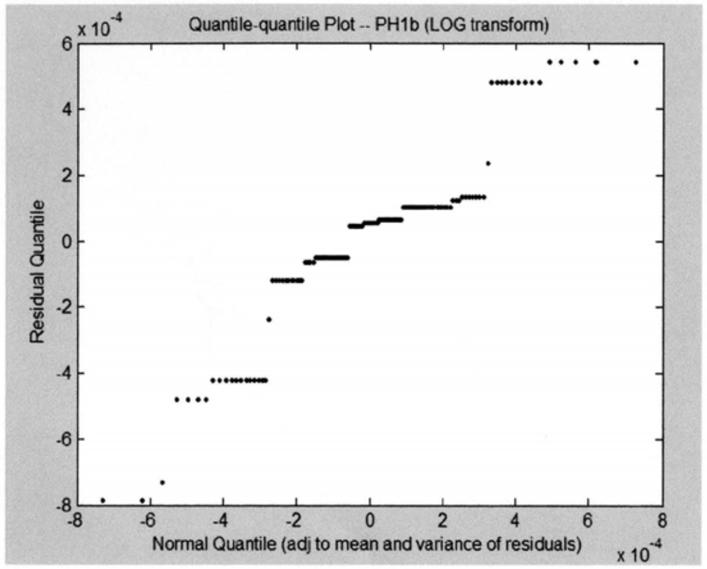
ANOVA & SSE calculated from experimental errors

variances		
	ATI	NVIDIA
WL1	6.59E-06	2.83E-07
WL2	1.99E-06	2.75E-07

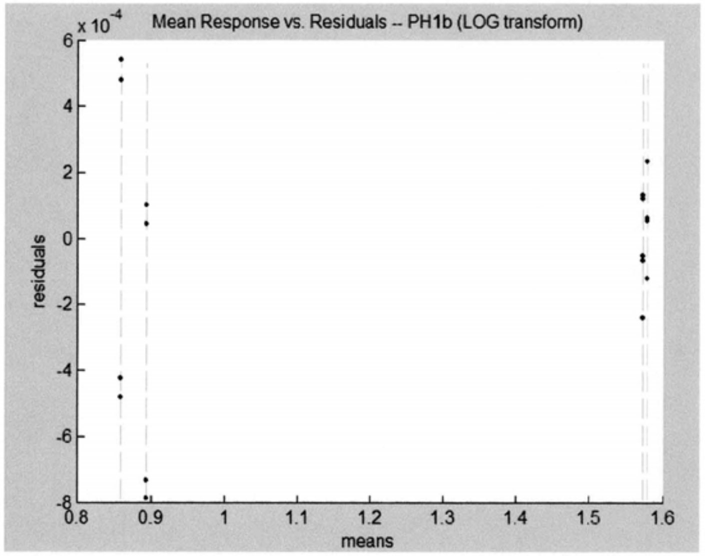
SSE	**9.14E-06**	
MSE	7.88E-08	
se	**2.81E-04**	**st dev of errors**
sq	**2.56E-05**	**st dev of effects**
90% c.i.	z[.95]	1.645

c.i. for effects and mean = q +/- sq * z =

q +/-	**4.21E-05**



Part 1



Part 2

Figure A-1b. Normal quantile-quantile and errors-versus-responses plots for Phase 1b (ATI vs. nVidia, spin scan, 512 size) experiments. Quantile-quantile plot is reasonably linear, except for a few outliers. Errors-versus-responses plot shows no trend. Analysis model is therefore valid.

TABLE A-2a

Results and analysis of Phase Two experiments, comparing GPU to CPU-based approaches, at three image sizes and three scene update schemes, on PCI-e and AGP platforms.

Results (Mean Responses of Execution Times, in seconds)

Scene Update Scheme:		Bus/Platform = AGP			Bus/Platform = PCI-e		
		Method			Method		
non-changing	Size	GPU	C++	MKL	GPU	C++	MKL
	128	0.640	1.267	0.876	0.676	1.199	0.664
	256	2.059	5.234	3.776	1.980	4.787	2.645
	512	7.226	25.032	20.448	7.263	21.885	19.409
fully-changing							
	Size						
	128	0.667	1.286	0.894	0.596	1.213	0.675
	256	2.198	5.404	4.046	2.059	4.843	2.709
	512	7.842	25.902	21.267	7.728	22.325	20.165
point source							
	Size						
	128	0.643	1.268	0.877	0.677	1.197	0.664
	256	2.060	5.232	3.758	1.978	4.786	2.646
	512	7.225	25.051	20.593	7.282	21.888	19.612

log-transform of data

		Bus/Platform = AGP			Bus/Platform = PCI-e		
Scene Update Scheme:		Method			Method		
non-changing	Size	GPU	C++	MKL	GPU	C++	MKL
	128	-0.194	0.103	-0.058	-0.240	0.079	-0.178
	256	0.314	0.719	0.577	0.297	0.680	0.423
	512	0.859	1.399	1.311	0.862	1.340	1.288
fully-changing	Size						
	128	-0.176	0.109	-0.049	-0.225	0.084	-0.171
	256	0.342	0.733	0.607	0.314	0.685	0.433
	512	0.894	1.413	1.328	0.888	1.349	1.305
point source	Size						
	128	-0.192	0.103	-0.067	-0.239	0.078	-0.178
	256	0.314	0.719	0.575	0.296	0.680	0.423
	512	0.859	1.399	1.314	0.862	1.340	1.293

Sum of Squares

Bus/Platform = AGP			Bus/Platform = PCI-e		
Method			Method		
GPU	C++	MKL	GPU	C++	MKL
1.131	0.318	0.100	1.721	0.186	0.952
2.953	15.501	9.989	2.638	13.874	5.356
22.131	58.674	51.534	22.309	53.880	49.769
0.928	0.357	0.071	1.514	0.212	0.875
3.509	16.105	11.065	2.952	14.082	5.620
24.001	59.925	52.884	23.659	54.576	51.059
1.106	0.319	0.098	1.716	0.183	0.951
2.955	15.496	9.916	2.631	13.870	5.359
22.128	58.702	51.776	22.304	53.884	50.118

Computation of Main Effects

Allocation of Variation

ANOVA

						sum of sq	% variation	dof	effect st dev	c.i = mean effect +/-
					SSY	**949.911**		1620		
					MEAN	**0.538**		1	3.98E-05	0.000
					SS0	**468.197**				
Effects			mean effect		**SST**	**481.714**				
Method		mean y	effect y-MEAN	sq						
A	GPU	0.324	**-0.213**	0.046		**43.547**	**9.040**	2	5.63E-05	0.000
	C++	0.723	**0.185**	0.034					5.63E-05	
	MKL	0.566	**0.028**	0.001					5.63E-05	
Bus/Platform										
B	AGP	0.565	**0.028**	0.001		**1.245**	**0.258**	1	3.98E-05	0.000
	PCI-e	0.510	**-0.028**	0.001					3.98E-05	
Scene Update										
C	non	0.532	**-0.005**	0.000		**0.087**	**0.018**	2	5.63E-05	0.000
	fully	0.548	**0.010**	0.000					5.63E-05	
	pt src	0.533	**-0.005**	0.000					5.63E-05	
Size										
D	128	-0.078	**-0.615**	0.379		430.221	**89.311**	2	5.63E-05	0.000
	256	0.507	**-0.030**	0.001					5.63E-05	
	512	1.183	**0.646**	0.417					5.63E-05	

sum of squared errors

	GPU	C++	MKL	GPU	C++	MKL
non-chg						
128	2.17E-04	7.82E-05	2.86E-04	1.85E-05	6.18E-06	2.10E-05
256	5.15E-05	6.89E-05	1.08E-04	2.38E-06	5.84E-06	5.73E-06
512	6.59E-06	3.33E-06	2.25E-06	7.63E-08	4.73E-05	3.82E-07
fully-chg						
128	6.35E-04	1.70E-04	4.30E-04	1.10E-05	2.65E-06	1.35E-05
256	8.48E-05	5.58E-05	8.77E-05	8.06E-06	8.77E-06	4.58E-06
512	1.99E-06	3.96E-06	1.38E-06	5.05E-07	1.52E-06	6.11E-07
pt src						
128	3.83E-04	1.03E-04	7.91E-04	2.44E-05	3.78E-06	1.57E-05
256	4.63E-05	3.89E-05	9.39E-05	8.35E-07	8.65E-06	1.60E-06
512	6.36E-06	6.00E-06	2.31E-06	1.33E-07	4.87E-05	4.32E-07
	SSE	**4.03E-03**	sum of above			
	MSE	**2.571E-06**	SSE/54*(r-1)	r=30		
	s_e	**1.60E-03**	$MSE^{1/2}$			
90% c.i.	z[.95]	1.645				

TABLE A-2a (continued)

Computation of Interaction Effects

ABCD Effects

		Bus/Platform						interactions squared Bus/Platform					
		AGP			PCI-e			AGP			PCI-e		
Scene Update		GPU	C++	MKL	GPU	C++	MKL	GPU	C++	MKL	GPU	C++	MKL
non-chg	128	-2.37E-05	2.99E-05	-6.23E-06	2.37E-05	-2.99E-05	6.23E-06	5.60E-10	8.94E-10	3.89E-11	5.60E-10	8.94E-10	3.89E-11
	256	4.87E-04	4.17E-04	-9.05E-04	-4.87E-04	-4.17E-04	9.05E-04	2.38E-07	1.74E-07	8.19E-07	2.38E-07	1.74E-07	8.19E-07
	512	-4.64E-04	-4.47E-04	9.11E-04	4.64E-04	4.47E-04	-9.11E-04	2.15E-07	2.00E-07	8.30E-07	2.15E-07	2.00E-07	8.30E-07
fully-chg	128	7.77E-05	2.42E-04	-3.20E-04	-7.77E-05	-2.42E-04	3.20E-04	6.03E-09	5.86E-08	1.02E-07	6.03E-09	5.86E-08	1.02E-07
	256	-1.17E-03	-9.65E-04	2.13E-03	1.17E-03	9.65E-04	-2.13E-03	1.36E-06	9.31E-07	4.55E-06	1.36E-06	9.31E-07	4.55E-06
	512	1.09E-03	7.23E-04	-1.81E-03	-1.09E-03	-7.23E-04	1.81E-03	1.19E-06	5.22E-07	3.29E-06	1.19E-06	5.22E-07	3.29E-06
pt src	128	-5.40E-05	-2.72E-04	3.26E-04	5.40E-05	2.72E-04	-3.26E-04	2.92E-09	7.40E-08	1.06E-07	2.92E-09	7.40E-08	1.06E-07
	256	6.80E-04	5.48E-04	-1.23E-03	-6.80E-04	-5.48E-04	1.23E-03	4.62E-07	3.00E-07	1.51E-06	4.62E-07	3.00E-07	1.51E-06
	512	-6.26E-04	-2.76E-04	9.02E-04	6.26E-04	2.76E-04	-9.02E-04	3.92E-07	7.59E-08	8.13E-07	3.92E-07	7.59E-08	8.13E-07

				sum of sq	**% variation**	**dof**	**st dev**	**+/-**
ABCD	sum of sq	0.000	30	0.001	0.000	8	1.13E-04	0.000

AB Effects

	AGP	PCI-e	squares	
GPU	-0.016	0.016	2.68E-04	2.68E-04
C++	-0.007	0.007	4.30E-05	4.30E-05
MKL	0.023	-0.023	5.26E-04	5.26E-04

				sum of sq	**% variation**	**dof**	**st dev**	**+/-**
AB	sum of sq	0.002	270	0.452	0.094	2	5.63E-05	0.000

AC Effects

	non	fully	pt src	squares		
GPU	-0.002	0.005	-0.003	6.07E-06	2.55E-05	6.70E-06
C++	0.002	-0.004	0.002	5.95E-06	1.91E-05	3.73E-06
MKL	0.000	-0.001	0.001	6.62E-10	4.66E-07	4.31E-07

				sum of sq	**% variation**	**dof**	**st dev**	**+/-**
AC	sum of sq	0.000	180	0.012	0.003	4	7.97E-05	0.000

AD Effects

	128	256	512	squares		
GPU	0.080	0.019	-0.099	6.45E-03	3.59E-04	9.85E-03
C++	-0.015	0.010	0.005	2.20E-04	1.04E-04	2.15E-05
MKL	-0.065	-0.029	0.095	4.28E-03	8.49E-04	8.95E-03

				sum of sq	**% variation**	**dof**	**st dev**	**+/-**
AD	sum of sq	0.031	180	5.594	1.161	4	7.97E-05	0.000

BC Effects

	non	fully	pt src	squares		
AGP	-0.001	0.002	-0.001	1.35E-06	5.30E-06	1.30E-06
PCI-e	0.001	-0.002	0.001	1.35E-06	5.30E-06	1.30E-06

				sum of sq	**% variation**	**dof**	**st dev**	**+/-**
BC	sum of sq	0.000	270	0.004	0.001	2	5.63E-05	0.000

BD Effects

	128	256	512	squares		
AGP	0.004	0.009	-0.014	2.00E-05	8.96E-05	1.94E-04
PCI-e	-0.004	-0.009	0.014	2.00E-05	8.96E-05	1.94E-04

				sum of sq	**% variation**	**dof**	**st dev**	**+/-**
BD	sum of sq	0.001	270	0.164	0.034	2	5.63E-05	0.000

CD Effects

	128	256	512	squares		
non	0.002	0.000	-0.002	3.44E-06	6.84E-08	2.54E-06
fully	-0.004	0.001	0.002	1.40E-05	1.96E-06	5.49E-06
pt src	0.002	-0.001	-0.001	3.55E-06	1.29E-06	5.60E-07

				sum of sq	**% variation**	**dof**	**st dev**	**+/-**
CD	sum of sq	0.000	270	0.009	0.002	4	7.97E-05	0.000

TABLE A-2a (continued)

ABC — ABC Effects

	non			fully			pt src		squares					
	AGP	PCI-e		AGP	PCI-e		AGP	PCI-e						
GPU	**-3.25E-04**	**3.25E-04**	GPU	**2.84E-04**	**-2.84E-04**	GPU	**4.10E-05**	**-4.10E-05**	1.05E-07	1.05E-07	8.05E-08	8.05E-08	1.68E-09	1.68E-09
C++	**2.23E-04**	**-2.23E-04**	C++	**-6.00E-04**	**6.00E-04**	C++	**3.77E-04**	**-3.77E-04**	4.96E-08	4.96E-08	3.59E-07	3.59E-07	1.42E-07	1.42E-07
MKL	**1.02E-04**	**-1.02E-04**	MKL	**3.16E-04**	**-3.16E-04**	MKL	**-4.18E-04**	**4.18E-04**	1.04E-08	1.04E-08	9.97E-08	9.97E-08	1.75E-07	1.75E-07

			<u>**sum of sq**</u>	<u>**% variation**</u>	<u>**dof**</u>	<u>**st dev**</u>	<u>**+/-**</u>
sum of sq	2.05E-06	90	**1.84E-04**	**0.000**	4	7.97E-05	1.31E-04

ACD — ACD Effects

	128				256				512		
	non	fully	pt src		non	fully	pt src		non	fully	pt src
GPU	**8.78E-05**	**-1.05E-03**	**9.58E-04**	GPU	**6.22E-04**	**-1.63E-03**	**1.01E-03**	GPU	**-7.10E-04**	**2.68E-03**	**-1.97E-03**
C++	**-7.20E-04**	**1.64E-03**	**-9.19E-04**	C++	**1.25E-04**	**-1.05E-03**	**9.25E-04**	C++	**5.95E-04**	**-5.89E-04**	**-6.45E-06**
MKL	**6.32E-04**	**-5.93E-04**	**-3.96E-05**	MKL	**-7.47E-04**	**2.68E-03**	**-1.93E-03**	MKL	**1.14E-04**	**-2.09E-03**	**1.97E-03**
					squares						
	7.71E-09	1.09E-06	9.19E-07		3.87E-07	2.66E-06	1.02E-06		5.04E-07	7.16E-06	3.87E-06
	5.19E-07	2.69E-06	8.44E-07		1.56E-08	1.10E-06	8.56E-07		3.54E-07	3.47E-07	4.15E-11
	4.00E-07	3.52E-07	1.57E-09		5.58E-07	7.19E-06	3.74E-06		1.31E-08	4.36E-06	3.89E-06

			<u>**sum of sq**</u>	<u>**% variation**</u>	<u>**dof**</u>	<u>**st dev**</u>	<u>**+/-**</u>
sum of sq	4.48E-05	60	**2.69E-03**	**0.001**	8	1.13E-04	1.85E-04

ABD — ABD Effects

	128			256			512		squares					
	AGP	PCI-e		AGP	PCI-e		AGP	PCI-e						
GPU	**0.008**	**-0.008**	GPU	**-0.010**	**0.010**	GPU	**0.003**	**-0.003**	6.01E-05	6.01E-05	1.06E-04	1.06E-04	6.36E-06	6.36E-06
C++	**-0.013**	**0.013**	C++	**-0.010**	**0.010**	C++	**0.023**	**-0.023**	1.76E-04	1.76E-04	9.56E-05	9.56E-05	5.31E-04	5.31E-04
MKL	**0.006**	**-0.006**	MKL	**0.020**	**-0.020**	MKL	**-0.026**	**0.026**	3.03E-05	3.03E-05	4.02E-04	4.02E-04	6.53E-04	6.53E-04

			<u>**sum of sq**</u>	<u>**% variation**</u>	<u>**dof**</u>	<u>**st dev**</u>	<u>**+/-**</u>
sum of sq	0.004	90.000	**0.371**	**0.077**	4	7.97E-05	1.31E-04

BCD — BCD Effects

	128				256				512		
	non	fully	pt src		non	fully	pt src		non	fully	pt src
AGP	**0.001**	**-0.002**	**0.001**	AGP	**-0.001**	**0.002**	**-0.001**	AGP	**0.000**	**0.000**	**0.000**
PCI-e	**-0.001**	**0.002**	**-0.001**	PCI-e	**0.001**	**-0.002**	**0.001**	PCI-e	**0.000**	**0.000**	**0.000**
					squares						
	3.99E-07	3.20E-06	1.34E-06		8.88E-07	4.83E-06	1.57E-06		9.63E-08	1.67E-07	9.74E-09
	3.99E-07	3.20E-06	1.34E-06		8.88E-07	4.83E-06	1.57E-06		9.63E-08	1.67E-07	9.74E-09

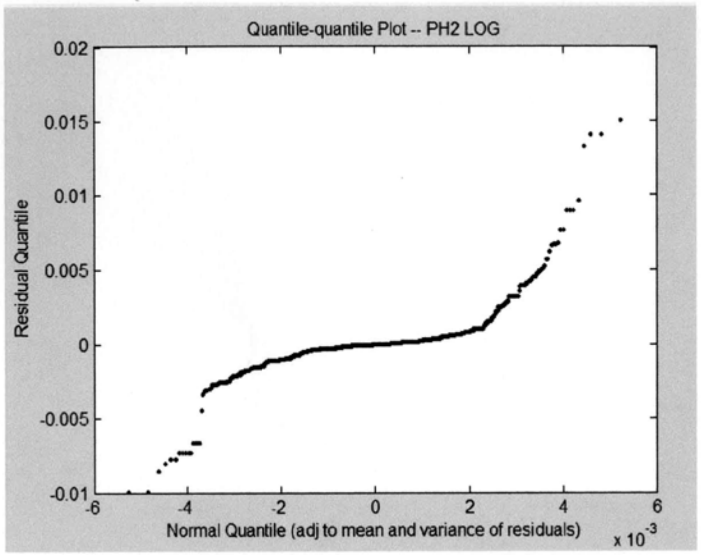
			<u>**sum of sq**</u>	<u>**% variation**</u>	<u>**dof**</u>	<u>**st dev**</u>	<u>**+/-**</u>
sum of sq	0.000	90.000	**0.002**	**0.000**	4	0.000	0.000

subtotal	**481.713**	**100.000**
SSE	**1.07E-03**	**0.000**
TOTAL	**481.714**	**100.000**

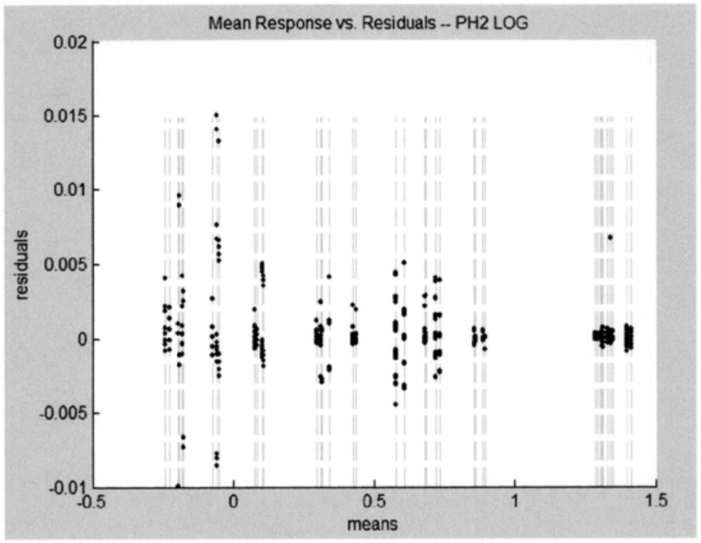
Representative F-test

SSA	DOF A	MSA	DOF Error	MSE	MSA/MSE	F[0.90;1,∞]	F[0.90;2,∞]
43.547	2	21.773	1566	2.57E-06	8469361	2.710	2.300

MSA/MSE >> F[0.90;x,y] for large denominator DOF and small (1 or 2) numerator DOF



Part 1



Part 2

Figure A-2. Normal quantile-quantile and errors-versus-responses plots for Phase 2 (spin scan, GPU, C++, MKL) experiments. Quantile-quantile plot is reasonably linear, except for a few outliers. Errors-versus-responses plot shows no trend. Analysis model is therefore valid.

TABLE A-2b

Subset of Phase Two experiments. Compares GPU to CPU-based processing methods. Non-changing scene update scheme only.

Scene Update Scheme=	log-transform of data					
non-changing	Bus/Platform = AGP			Bus/Platform = PCI-e		
	Method			Method		
Size	GPU	C++	MKL	GPU	C++	MKL
128	-0.194	0.103	-0.058	-0.240	0.079	-0.178
256	0.314	0.719	0.577	0.297	0.680	0.423
512	0.859	1.399	1.311	0.862	1.340	1.288

Sum of Squares					
Bus/Platform = AGP			Bus/Platform = PCI-e		
Method			Method		
GPU	C++	MKL	GPU	C++	MKL
1.131	0.318	0.100	1.721	0.186	0.952
2.953	15.501	9.989	2.638	13.874	5.355
22.131	58.674	51.534	22.309	53.880	49.769

			Allocation of Variation			ANOVA		
							effect	c.i = mean effect
	Computation of Effects							
	main effects			sum of square	% variation	dof	st dev	+/-
MEAN	**0.532**					1	5.74E-05	0.000
SSY	**313.015**					540		
SS0	**152.937**							
SST	**160.078**							
Method								
SSA		**0.083**	**180**	**14.870**	**9.289**	2	8.12E-05	0.000
GPU	**-0.216**	0.047						
C++	**0.188**	0.035						
MKL	**0.028**	0.001						
Bus/Platform								
SSB		**0.001**	**270**	**0.381**	**0.238**	1	5.74E-05	0.000
AGP	**0.027**	0.001						
PCI-e	**-0.027**	0.001						
Size								
SSC		**0.792**	**180**	**142.629**	**89.100**	2	8.12E-05	0.000
128	**-0.614**	0.376						
256	**-0.031**	0.001						
512	**0.644**	0.415						

sum of squared error					
2.17E-04	7.82E-05	2.86E-04	1.85E-05	6.18E-06	2.10E-05
5.15E-05	6.89E-05	1.08E-04	2.38E-06	5.84E-06	5.73E-06
6.59E-06	3.33E-06	2.25E-06	7.63E-08	4.73E-05	3.82E-07
SSE	**9.29E-04**				
MSE	**1.78E-06**				
se	**1.33E-03**				
90% c.i.	z[.95]	1.645			

TABLE A-2b (continued)

Effects of Interactions

					AB					
		AGP	PCI-e		squares					
GPU		**-0.017**	**0.017**		0.000	0.000				
C++		**-0.006**	**0.006**		0.000	0.000				
MKL		**0.023**	**-0.023**		0.001	0.001				
							dof	**st dev**	**+/-**	
SSAB	ssq		**0.002**	**90**	**0.153**	**0.096**	2	8.12E-05	0.000	
					AC					
		128	256	512			squares			
GPU		**0.080**	**0.020**	**-0.100**		0.006	0.000	0.010		
C++		**-0.016**	**0.010**	**0.005**		0.000	0.000	0.000		
MKL		**-0.065**	**-0.030**	**0.095**		0.004	0.001	0.009		
							dof	**st dev**	**+/-**	
SSAC	**ssq**		**0.031**	**60**	**1.876**	**1.172**	4	1.15E-04	0.000	
					BC					
		128	256	512			squares			
AGP		**0.005**	**0.009**	**-0.014**		2.61E-05	7.26E-05	1.86E-04		
PCI-e		**-0.005**	**-0.009**	**0.014**		2.61E-05	7.26E-05	1.86E-04		
							dof	**st dev**	**+/-**	
SSBC	**ssq**		**0.001**	**90**	**0.051**	**0.032**	2	8.12E-05	0.000	
					ABC					
			AGP				PCI-e			
		GPU	C++	MKL		GPU	C++	MKL		
	128	**0.008**	**-0.013**	**0.005**		**-0.008**	**0.013**	**-0.005**		
	256	**-0.010**	**-0.009**	**0.019**		**0.010**	**0.009**	**-0.019**		
	512	**0.002**	**0.023**	**-0.025**		**-0.002**	**-0.023**	**0.025**		
					squares					
		5.98E-05	1.75E-04	3.02E-05		5.98E-05	1.75E-04	3.02E-05		
		9.58E-05	8.76E-05	3.67E-04		9.58E-05	8.76E-05	3.67E-04		
		4.23E-06	5.10E-04	6.07E-04		4.23E-06	5.10E-04	6.07E-04		
							dof	**st dev**	**+/-**	
SSABC			**0.004**	**30**	**0.116**	**0.073**	4	1.15E-04	0.000	
					160.077	**99.999**	**Total percent**			
SSE					**9.29E-04**	**0.001**				

TABLE A-2c

Subset of Phase Two Experiments. Compares GPU versus CPU-based methods on PCI-e and AGP platforms. Separate analysis performed for each image size case. Non-changing scene update scheme.

			log-transform of data					sum of squares			
Size=			Method					Method			
128	Bus/Pltfm	GPU	C++	MKL		Bus/Pltfm	GPU	C++	MKL		
	AGP	-0.194	0.103	-0.058		AGP	1.131	0.318	0.100		
	PCI-e	-0.240	0.079	-0.178		PCI-e	1.721	0.186	0.952		
							ANOVA				
							effect	**c.i = mean effect**			
						dof	**st dev**	**+/-**			
MEAN	**-0.0813279**					1	1.41E-04	0.000			
SSY	**4.407**					180			sum of squared error		
SS0	**1.19056083**										
SST	**3.217**			**Allocation of Variation**							
	main effects			**SSx**	**%var**				2.17E-04	7.82E-05	2.86E-04
A - Bus/Platform		**0.002**	**90.000**	**0.180**	**5.609**	1	1.41E-04	0.000	1.85E-05	6.18E-06	2.10E-05
AGP	**0.032**	0.001									
PCI-e	**-0.032**	0.001							**SSE**	**6.27E-04**	
									MSE	**3.60E-06**	
B- Method		**0.049**	**60.000**	**2.959**	**91.989**	2	2.00E-04	0.000	**s_e**	**1.90E-03**	$MSE^{1/2}$
GPU	**-0.135**	0.018							90% c.i.	z[.95]	1.645
C++	**0.172**	0.030									
MKL	**-0.037**	0.001									
	GPU	C++	MKL								
AGP	**-0.009**	**-0.020**	**0.029**		0.000	0.000	0.001				
PCI-e	**0.009**	**0.020**	**-0.029**		0.000	0.000	0.001				
AB	**ssq**	**0.003**	**30.000**	**0.077**	**2.382**	2	2.00E-04	0.000			
Total percent explained				**3.216**	**99.981**						
SSE				**6.27E-04**	**0.019**	174					

			log-transform of data					sum of squares			
			Method					Method			
Size=	Bus/Pltfm	GPU	C++	MKL		Bus/Pltfm	GPU	C++	MKL		
256	AGP	0.314	0.719	0.577		AGP	2.953	15.501	9.989		
	PCI-e	0.297	0.680	0.423		PCI-e	2.638	13.874	5.355		
							ANOVA				
							effect	**c.i = mean effect**			
						dof	**st dev**	**+/-**			
MEAN	**0.50144904**					1	8.788E-05	0.000			
SSY	**50.311**					180			sum of squared error		
SS0	**45.261205**										
SST	**5.050**			**Allocation of Variation**					Variance	x1 SSE	x1 SSE
	main effects			**SSx**	**%var**				1.78E-06	6.89E-05	1.08E-04
A - Bus/Platform		**0.002**	**90.000**	**0.221**	**4.386**	1	8.788E-05	0.000	8.20E-08	5.84E-06	5.73E-06
AGP	**0.035**	0.001							5.39E-05	7.47E-05	1.13E-04
PCI-e	**-0.035**	0.001									
									SSE	**2.42E-04**	
B- Method		**0.078**	**60.000**	**4.664**	**92.368**	2	1.24E-04	0.000	**MSE**	**1.39E-06**	
GPU	**-0.196**	0.039							**s_e**	**1.18E-03**	$MSE^{1/2}$
C++	**0.198**	0.039							90% c.i.	z[.95]	1.645
MKL	**-0.002**	0.000									
	GPU	C++	MKL								
AGP	**-0.026**	**-0.016**	**0.042**		0.001	0.000	0.002				
PCI-e	**0.026**	**0.016**	**-0.042**		0.001	0.000	0.002				
AB	**ssq**	**0.005**	**30.000**	**0.164**	**3.241**	2	1.24E-04	0.000			
Total percent explained				**5.049**	**99.995**						
SSE				**2.42E-04**	**0.005**	174					

TABLE A-2c (continued)

			log-transform of data					sum of squares			
			Method					Method			
Size=	Bus/Pltfm	GPU	C++	MKL		Bus/Pltfm	GPU	C++	MKL		
512	AGP	0.859	1.399	1.311		AGP	22.131	58.674	51.534		
	PCI-e	0.862	1.340	1.288		PCI-e	22.309	53.880	49.769		
							ANOVA				
							effect	**c.i = mean effect**			
						dof	**st dev**	**+/-**			
MEAN	**1.17642353**					1	4.374E-05	0.000			
SSY	**258.297**					180				sum of squared error	
SS0	**249.115018**										
SST	**9.182**			**Allocation of Variation**					6.59E-06	3.33E-06	2.25E-06
	main effects			**SSx**	**%var**				7.63E-08	4.73E-05	3.82E-07
A - Bus/Platform		**0.000**	**90.000**	**0.030**	**0.328**	1	4.374E-05	0.000			
AGP	**0.013**	0.000									
PCI-e	**-0.013**	0.000							**SSE**	**5.99E-05**	
									MSE	**3.44E-07**	
B- Method		**0.152**	**60.000**	**9.123**	**99.357**	2	6.19E-05	0.000	**s_e**	**5.87E-04**	MSE$^{1/2}$
GPU	**-0.316**	0.100							90% c.i.	z[.95]	1.645
C++	**0.193**	0.037									
MKL	**0.123**	0.015									
	GPU	C++	MKL								
AGP	**-0.015**	**0.016**	**-0.002**		0.000	0.000	0.000				
PCI-e	**0.015**	**-0.016**	**0.002**		0.000	0.000	0.000				
AB	**ssq**	**0.001**	**30.000**	**0.029**	**0.314**	2	6.19E-05	0.000			
Total percent explained				**9.182**	**99.999**						
SSE				**5.99E-05**	**0.001**	174					

TABLE A-3

Results and analysis of Phase Three experiments. Compares GPU versus CPU-based methods for the JMASS conical scan procedure.

SIZE=	results (mean responses)		
128		method	
platform	GPU	C++	MKL
AGP	1.138	1.505	1.414
PCI-e	1.012	1.221	0.942

	log-transform of data		
		method	
platform	GPU	C++	MKL
AGP	5.61E-02	1.78E-01	1.50E-01
PCI-e	5.01E-03	8.69E-02	-2.60E-02

	sum of squares		
	method		
platform	GPU	C++	MKL
AGP	0.095	0.946	0.678
PCI-e	0.001	0.226	0.020

ANOVA

					Allocation of Variation			effect	c.i = mean effect
					SSx	**% variation**	dof	st dev	+/-
MEAN	**0.075**						1	2.20E-04	0.000
SSY	**1.967**						180		
SS0	**1.012**								
SST	**0.955**								
main effects									
A-bus/platform		0.006		90	**0.506**	**52.969**	1	2.20E-04	0.000
AGP	**0.053**	0.003							
PCI-e	**-0.053**	0.003							
B-method		0.005		60	**0.325**	**34.023**	2	3.11E-04	0.001
GPU	**-0.044**	0.002							
C++	**0.057**	0.003							
MKL	**-0.013**	0.000							

	GPU	C++	MKL	effects squared		
AGP	**-0.027**	**-0.008**	**0.035**	7.54E-04	5.86E-05	1.23E-03
PCI-e	**0.027**	**0.008**	**-0.035**	7.54E-04	5.86E-05	1.23E-03

				SSx	% variation	dof	effect st dev	+/-
AB	ssq	0.004	30	**0.123**	**12.849**	2	3.11E-04	0.001
SSE				**0.002**	**0.159**	174		
TOTAL				**0.955**	**100.000**			

sum of squared error		
8.27E-04	1.74E-04	4.45E-04
4.18E-05	3.22E-06	2.37E-05

SSE	**0.002**	
MSE	**8.71E-06**	
s_e	**2.95E-03**	$MSE^{1/2}$
90% c.i.	z[.95]	1.645

TABLE A-3 (continued)

SIZE= 256

results (mean responses)

platform	GPU	C++	MKL
		method	
AGP	3.037	7.971	6.688
PCI-e	2.889	5.796	4.650

log-transform of data

platform	GPU	C++	MKL
		method	
AGP	4.82E-01	9.02E-01	8.25E-01
PCI-e	4.61E-01	7.63E-01	6.67E-01

sum of squares

platform	GPU	C++	MKL
		method	
AGP	6.982	24.383	20.434
PCI-e	6.370	17.472	13.364

ANOVA

			dof	effect st dev	c.i = mean effect +/-
MEAN	**0.683**		1	5.20E-05	0.000
SSY	**89.004**		180		
SS0	**84.076**				
SST	**4.928**				

Allocation of Variation

main effects				**SSx**	**% variation**			
A-bus/platform		0.006	90	**0.505**	**10.252**	1	5.20E-05	0.000
AGP	**0.053**	0.003						
PCI-e	**-0.053**	0.003						
B-method		0.071	60	**4.260**	**86.442**	2	7.35E-05	0.000
GPU	**-0.212**	0.045						
C++	**0.149**	0.022						
MKL	**0.063**	0.004						

	GPU	C++	MKL	effects squared		
AGP	**-0.042**	**0.016**	**0.026**	1.78E-03	2.63E-04	6.73E-04
PCI-e	**0.042**	**-0.016**	**-0.026**	1.78E-03	2.63E-04	6.73E-04

AB	ssq	0.005	30	**0.163**	**3.304**	2	7.35E-05	0.000
SSE				**0.000**	**0.002**			
TOTAL				**4.928**	**100.000**			

sum of squared error

3.41E-05	6.09E-06	3.30E-06
1.14E-06	1.13E-05	2.86E-05

SSE	**0.000**	
MSE	**4.86E-07**	
s_e	**6.97E-04**	$MSE^{1/2}$
90% c.i.	z[.95]	1.645

TABLE A-3 (continued)

SIZE=
512

results (mean responses)

	method		
platform	GPU	C++	MKL
AGP	10.639	27.759	24.235
PCI-e	10.430	24.920	22.256

log-transform of data

	method		
platform	GPU	C++	MKL
AGP	1.027	1.443	1.384
PCI-e	1.018	1.397	1.347

sum of squares

	method		
platform	GPU	C++	MKL
AGP	31.635	62.503	57.501
PCI-e	31.108	58.511	54.469

ANOVA

dof	effect st dev	c.i = mean effect +/-
1	4.94E-05	0.000
180		

MEAN	**1.270**
SSY	**295.726**
SS0	**290.096**
SST	**5.629**

Allocation of Variation

main effects				SSx	% variation			
A-bus/platform		0.000	90	**0.043**	**0.759**	1	4.94E-05	0.000
AGP	**0.015**	0.000						
PCI-e	**-0.015**	0.000						
B-method		0.093	60	**5.575**	**99.029**	2	6.99E-05	0.000
GPU	**-0.247**	0.061						
C++	**0.150**	0.023						
MKL	**0.096**	0.009						

	GPU	C++	MKL	effects squared		
AGP	**-0.011**	**0.008**	**0.003**	1.24E-04	6.44E-05	9.55E-06
PCI-e	**0.011**	**-0.008**	**-0.003**	1.24E-04	6.44E-05	9.55E-06

AB	ssq	0.000	30	**0.012**	**0.210**	2	6.99E-05	0.000
SSE				**0.000**	**0.001**			
TOTAL				**5.629**	**100.000**			

sum of squared error

1.41E-06	8.02E-07	7.09E-05
4.41E-07	1.32E-06	1.56E-06

SSE	**0.000**	
MSE	**4.39E-07**	
s_e	**6.63E-04**	MSE$^{1/2}$
90% c.i.	z[.95]	1.645

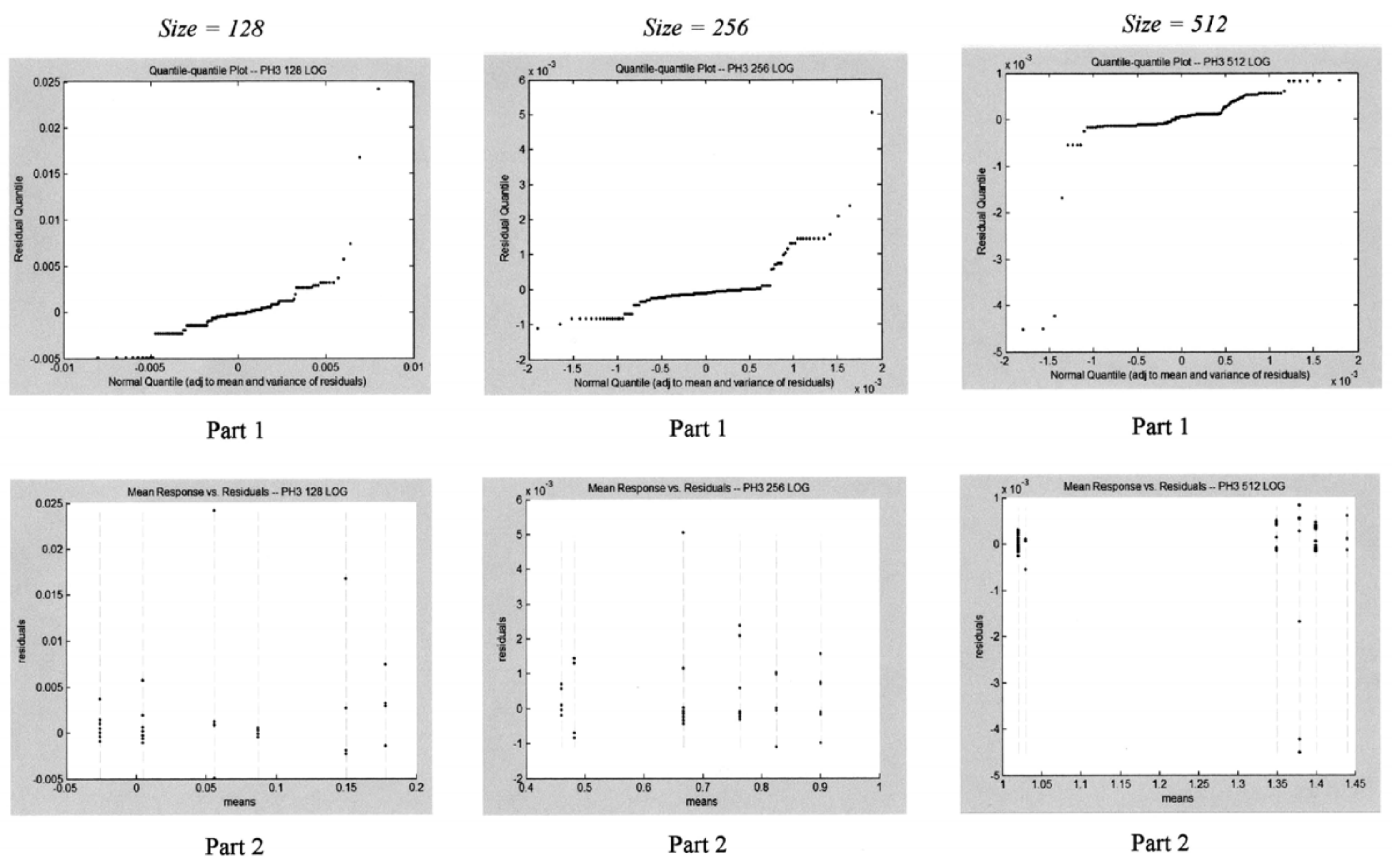


Figure A-3. Normal quantile-quantile and errors-versus-responses plots for Phase 3 (conical scan, GPU, C++, MKL) experiments. Quantile-quantile plot is reasonably linear, except for a few outliers. Errors-versus-responses plot shows no trend. Analysis model is therefore valid.

TABLE A-4

Analysis of Phase Four experiments, comparing baseline JMASS to Modified JMASS, Software and GPU-assisted versions.

JMASS Execution Time

	JMASS	JMASS	GPU Assisted	
Image Size	Orig	Modified	NV	ATi
128	579	**407**	**359**	**360**
256	2141	**1574**	**1411**	**1393**
512	8200	**6289**	**5525**	**5530**

JMASS Execution Time (log transform)

	JMASS	JMASS	GPU Assisted		sum of squares			
Image Size	Orig	Modified	NV	ATi				
128	2.763	2.610	2.555	2.556	7.632	6.810	6.529	6.535
256	3.331	3.197	3.150	3.144	11.093	10.221	9.920	9.884
512	3.914	3.799	3.742	3.743	15.318	14.429	14.005	14.008

MEAN	3.209
SSY	126.384
SSO	123.535
SST	2.848

Allocation of Variation

Mean Effects			ssq	multiplier	SSx	% variation explained
A image size			0.694	4	2.778	**98**
128	**-0.588**	0.345				
256	**-0.003**	0.000				
512	**0.591**	0.349				
B method			0.023	3	0.070	**2**
Orig	**0.127**	0.016				
Mod	**-0.007**	0.000				
NV	**-0.060**	0.004				
ATI	**-0.061**	0.004				
Interaction AB			0	1	0.001	**0**

	Orig	Mod	NV	ATI
128	**0.015**	**-0.005**	**-0.006**	**-0.004**
256	**-0.002**	**-0.001**	**0.004**	**0.000**
512	**-0.013**	**0.006**	**0.003**	**0.004**

squares

2.12E-04	2.05E-05	3.96E-05	1.41E-05
3.40E-06	2.19E-06	1.43E-05	2.16E-07
1.62E-04	3.61E-05	6.27E-06	1.78E-05

This page intentionally left blank.

Appendix B. GPU Implementation Code

Contents

```
// winAppGPU.cpp
//
// by:  Maj Sean Jeffers
// descr:  windows test application for GPU-based algorithms
// 27 dec 04 -- modified to output both normal and log-transformed execution time data
//
//
#include "stdafx.h"
#include "winAppGPU.h"
#define MAX_LOADSTRING 100

#include <iostream>
#include <fstream>
#include <iomanip>

#include <cmath>
#include "GPU_COMBINED.h" //combined.h or CLASS_ONEBYONE.h CLASS_ONEBYONE_R32F.h

// Global Variables:
HINSTANCE hInst;                                        // current instance
TCHAR szTitle[MAX_LOADSTRING];                          // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];                    // the main window class name

const int        EXPER              = 100;
const int        SCENE_SIZE         = 128;
const int        WL                 = 1;
const char*      BUS_str            = "PCI-e";
const char*      APRCH_str          = "ATI";
const int        REPS               = 2;
const int SIZE_SQ = SCENE_SIZE*SCENE_SIZE;

// WL 3 pt source vars
const int xmin = SCENE_SIZE/4;
const int ymin = SCENE_SIZE/4;
const int xmax = SCENE_SIZE-xmin;
const int ymax = SCENE_SIZE-ymin;
//initial conditions
int oldx = xmin;
int oldy = ymin;
int delx = -1;
int xinc = -1;
int dely = 0;
int yinc = -1;

// Forward declarations of functions included in this code module:
void    UpdateScene(int , float*);
ATOM    MyRegisterClass(HINSTANCE hInstance);
BOOL    InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
        float reticle[SIZE_SQ];
        float scene[SIZE_SQ];
        double answer[40];
        double times[REPS];
        double timeslog[REPS];
        double startTime;
        double endTime;

        double sum          = 0.0;
        double mean         = 0.0;
        double var          = 0.0;
        double stdev        = 0.0;
        double hi           = 0.0;
        double low          = 0.0;
        double sos          = 0.0;

        double sumlog       = 0.0;
        double meanlog      = 0.0;
        double varlog       = 0.0;
        double stdevlog     = 0.0;
        double soslog       = 0.0;
        double hilog        = 0.0;
        double lowlog       = 0.0;

        char WL_str[30];
        if (WL ==1){
                  strcpy(WL_str,"1 - non-changing");
        }
        else if (WL == 2){
                  strcpy(WL_str,"2 - fully-changing");
        }
        else {
                  strcpy(WL_str,"3 - moving pt source");
        }

        //instantiate GPU object
        Gpu gpu(hInstance,nCmdShow, SCENE_SIZE);
        //upload  reticles
        for (int i = 0; i<100; i++){
                  for (int j = 0; j<SIZE_SQ; j++){
                            reticle[j] = (float)i;
                  }
                  gpu.uploadReticle(i,reticle);
        }

        char algorithm[40];
        int alg =gpu.GetAlg();
        if (alg ==1){
                  strcpy(algorithm,"BIGTEX");
        }
        else if (alg == 2){
                  strcpy(algorithm,"ONEBYONE");
```

```
}
else if (alg == 3){
          strcpy(algorithm,"CONSCAN ONEBYONE R32F");
}
else {
          strcpy(algorithm,"NA");
}
// do experiment REPS times
for (int rep = 0; rep< REPS; rep++){
          //fill initial scene
          if  ( WL != 3){
                    for (int j = 0; j<SIZE_SQ; j++){
                              scene[j]= (float)j;
                    }
          }
          else {
                    for (int j = 0; j<SIZE_SQ; j++){
                              scene[j]= 0.0f;
                    }
          }

          startTime = (double)timeGetTime();
          // run algorithm 1000 x
          for (int i = 0; i<1000; i++){
     gpu.Process(i%100,scene,answer);
                    UpdateScene(WL,scene);
          }
          endTime = (double) timeGetTime();
          double timeDelta = (endTime-startTime)*0.001f;
          double timeDeltaLog = log10(timeDelta);
          times[rep]= timeDelta;
          timeslog[rep] = timeDeltaLog;
          sum += timeDelta;
          sumlog += timeDeltaLog;
}
//calc stats
mean = sum/(double)REPS;
meanlog = sumlog/(double)REPS;
hi = 0.0;
hilog = -1000.0;
low = 1000.0;
lowlog = 1000.0;

for (int i =0; i<REPS; i++){
          if (times[i]>hi)
                    hi = times[i];
          if (times[i]<low)
                    low = times[i];
          var += pow( (times[i]-mean),2.0)/(double)(REPS-1);
          sos += pow( times[i],2);
          if (timeslog[i]>hilog)
                    hilog = timeslog[i];
          if (timeslog[i]<lowlog)
                    lowlog = timeslog[i];
          varlog += pow( (timeslog[i]-meanlog),2.0)/(double)(REPS-1);
          soslog += pow( timeslog[i],2);
}
stdev = sqrt( var);
stdevlog = sqrt (varlog);
//write results to file
char* name ="results/results_";
char* ext  =".dat";
char num[4];
_itoa(EXPER,num,10);
char filename[40];
strcpy(filename,name);
strcat(filename,num);
strcat(filename,ext);
std::ofstream outFile(filename,std::ios::app);//out
if (!outFile){
          ::MessageBox(0, "can't open results file","GPU" , 0);
          exit(1);
}
outFile   <<"experiment#: "<<EXPER<<'\n'
                    <<"workload:    "<<WL_str<<'\n'
                    <<"bus:         "<<BUS_str<<'\n'
                    <<"approach:    "<<APRCH_str<<'\n'
                    <<"algorithm:   "<<algorithm<<'\n'
                    <<"size:        "<<SCENE_SIZE<<'\n'
                    <<"mean:        "<<mean<<'\n'
                    <<"variance:    "<<var<<'\n'
                    <<"stdev:       "<<stdev<<'\n'
                    <<"sum of sqrs: "<<sos<<'\n'
                    <<"low:         "<<low<<'\n'
                    <<"hi:          "<<hi<<'\n'
                    <<"mean log:        "<<meanlog<<'\n'
                    <<"variance log:    "<<varlog<<'\n'
                    <<"stdev log :      "<<stdevlog<<'\n'
                    <<"sum of sqrs log: "<<soslog<<'\n'
                    <<"low log:         "<<lowlog<<'\n'
                    <<"hi log:          "<<hilog<<'\n'
                    <<"reps:        "<<REPS<<'\n'
                    <<"data:        "<<'\n';
for (i =0;i<REPS; i++){
          outFile<<times[i];
          if (!((i+1)%5) || (i==REPS-1))
                    outFile<<'\t'<<" ..."<<'\n';
          else outFile<<'\t';
}
outFile<<'\n'<<"data log:        "<<'\n';
for (i =0;i<REPS; i++){
          outFile<<std::setprecision(6)<<std::setw(3)<<timeslog[i];
          if (!((i+1)%5) || (i==REPS-1))
                    outFile<<'\t'<<" ..."<<'\n';
          else outFile<<'\t';
}
outFile<<'\n'<<"answers:"<<'\n';
for (i =0;i<40; i++){
```

```
                    outFile<<std::setprecision(9)<<std::setw(18)<<
                              std::setiosflags(std::ios::scientific)<<answer[i];
                    if (!(( i+1)%4))
                              outFile<<'\n';
          }
          outFile<<'\n';

          // TODO: Place code here.
          //-----------------------------------------------------------------
          MSG msg;
          HACCEL hAccelTable;

          // Initialize global strings
          LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
          LoadString(hInstance, IDC_WINAPPGPU, szWindowClass, MAX_LOADSTRING);
          MyRegisterClass(hInstance);

          // Perform application initialization:
          if (!InitInstance (hInstance, nCmdShow))
          {
                    return FALSE;
          }

          hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_WINAPPGPU);

          // Main message loop:
          while (GetMessage(&msg, NULL, 0, 0))
          {
                    if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
                    {
                              TranslateMessage(&msg);
                              DispatchMessage(&msg);
                    }
          }

          return (int) msg.wParam;
}
//-------------------------------------------------------------
//  FUNCTION:  UpdateScene()
//------------------------------------------------------------
void UpdateScene(int p_WL, float* p_scene){
          if (p_WL == 1){
                    return;
          }
          if (p_WL == 2){
                    for (int j = 0; j < SIZE_SQ; j++){
                              p_scene[j] += 1.0f;
                    }
                    return;
          }
          else {
                    int x = delx + oldx;
                    int y = dely + oldy;
                    if ( (x<xmin) || (x>xmax) ){
                              x = oldx;
                              xinc = - xinc;
                              delx = delx+xinc;
                              dely = dely+yinc;
                              y += dely;
                    }
                    if ( (y<ymin) || (y>ymax) ) {
                              y = oldy;
                              yinc = -yinc;
                              delx += xinc;
                              dely += yinc;
                              x += delx;
                    }

                    int index = y*SCENE_SIZE + x;
                    int indexold = oldy*SCENE_SIZE + oldx;
                    p_scene[indexold] = 0.0f;
                    p_scene[index] = 1.0f;
                    oldx = x;
                    oldy =y;
                    return;
          }
}

//
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//
//    This function and its usage are only necessary if you want this code
//    to be compatible with Win32 systems prior to the 'RegisterClassEx'
//    function that was added to Windows 95. It is important to call this function
//    so that the application will get 'well formed' small icons associated
//    with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
          WNDCLASSEX wcex;

          wcex.cbSize = sizeof(WNDCLASSEX);

          wcex.style          = CS_HREDRAW | CS_VREDRAW;
          wcex.lpfnWndProc    = (WNDPROC)WndProc;
          wcex.cbClsExtra     = 0;
          wcex.cbWndExtra     = 0;
          wcex.hInstance      = hInstance;
          wcex.hIcon          = LoadIcon(hInstance, (LPCTSTR)IDI_WINAPPGPU);
          wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
          wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
          wcex.lpszMenuName   = (LPCTSTR)IDC_WINAPPGPU;
          wcex.lpszClassName  = szWindowClass;
          wcex.hIconSm        = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);
```

```
	return RegisterClassEx(&wcex);
}

//
//   FUNCTION: InitInstance(HANDLE, int)
//
//   PURPOSE: Saves instance handle and creates main window
//
//   COMMENTS:
//
//        In this function, we save the instance handle in a global variable and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Store instance handle in our global variable

   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//
//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND	- process the application menu
//  WM_PAINT	- Paint the main window
//  WM_DESTROY	- post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
	int wmId, wmEvent;
	PAINTSTRUCT ps;
	HDC hdc;

	switch (message)
	{
	case WM_COMMAND:
		wmId    = LOWORD(wParam);
		wmEvent = HIWORD(wParam);
		// Parse the menu selections:
		switch (wmId)
		{
		case IDM_ABOUT:
			DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
			break;
		case IDM_EXIT:
			DestroyWindow(hWnd);
			break;
		default:
			return DefWindowProc(hWnd, message, wParam, lParam);
		}
		break;
	case WM_PAINT:
		hdc = BeginPaint(hWnd, &ps);
		// TODO: Add any drawing code here...
		EndPaint(hWnd, &ps);
		break;
	case WM_DESTROY:
		PostQuitMessage(0);
		break;
	default:
		return DefWindowProc(hWnd, message, wParam, lParam);
	}
	return 0;
}

// Message handler for about box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
	switch (message)
	{
	case WM_INITDIALOG:
		return TRUE;

	case WM_COMMAND:
		if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
		{
			EndDialog(hDlg, LOWORD(wParam));
			return TRUE;
		}
		break;
	}
	return FALSE;
}
```

```
// file: GPU_combined.h
//
// by:   Maj Sean Jeffers
// requires external files:
//      GPU_UTILITY.h                 -- contains namespace d3d utility functions InitD3D()
//                                        GPU_WndProc CALLBACK and Gpu_WndClass definition
//  source/vs_bigtex.txt              -- vertex shader used by MAddReduce() for BIGTEX
//  source/ps_bigtex.txt              -- pixel shader used by  MAddReduce() for BIGTEX
//  source/ps_onebyone.txt            -- PS used for MAddReduce() for ONEBYONE
//  source/vs_onebyone.txt            -- PS used for MAddReduce() for ONEBYONE
//  source/vs_16tapredux_2.txt        -- vertex shader used by Redux()
//  source/ps_16tapredux_2.txt        -- pixel shader used by  Redux()
//
// 27 dec   -- combined BIGTEX for 128/256 and ONEBYONE for 512 size; modified both
//             BIGTEX and ONEBYONE pixel shaders to take 128-bit tex's in, but output
//             to R32F for speed;  ps_experimental and ps_maddreduce_new were modified
//
#ifndef GPU_CLASS_H_BY_MAJ_JEFFERS
#define GPU_CLASS_H_BY_MAJ_JEFFERS

#include <d3dx9.h>
#include "GPU_UTILITY.h"

#include <stdlib.h>
#include <cstring>
#include <cmath>

//------------------ CONSTANTS----------------
#define GPU_WINDOW_WIDTH    1024
#define GPU_WINDOW_HEIGHT   768
#define D3D_FORMAT          D3DFMT_A32B32G32R32F
#define STRIDE              16
//--------------------------------------------

class Gpu {

private:
       HINSTANCE                     hInst;
       int                           nCmdShow;
       IDirect3DDevice9*             Device;
       const int                     SceneSize;
       int                           ScenePixels;
       float                         fPixSizeX;
       float                         fPixSizeY;
       //D3DXVECTOR4                 DataArray[2048*2048];
       int                           OutTexSize;
       long                          OutPixels;
       //int                         ViewportSize;
       int                           ReduceIterations;
       int                           TexIndex;
       bool                          DualRT;
       int                           start1;
       int                           end1;
       int                           end2;

       //VS1
       IDirect3DVertexShader9*                 VS1_maddreduce;
       ID3DXConstantTable*                     VS1_VSCT;
       D3DXHANDLE                              VS1_PixelSizeHandle;
       //PS1
       IDirect3DPixelShader9*                  PS1_maddreduce;
       ID3DXConstantTable*                     PS1_PSCT;
       D3DXHANDLE                              PS1_PixelSizeHandle;

       //VS2
       IDirect3DVertexShader9*                 VS2_16tapreduce;
       ID3DXConstantTable*                     VS2_VSCT;
       D3DXHANDLE                              VS2_offsetHandle;

       //PS2
       IDirect3DPixelShader9*                  PS2_16tapreduce;
       ID3DXConstantTable*                     PS2_PSCT;
       D3DXHANDLE                              PS2_offsetHandle;
       D3DXHANDLE                              PS2_mulHandle;

       // PARAMETERS PASSED TO PS & VS
       D3DXVECTOR2                             offset[3][16];

       // VERTEX BUFFER & DECL
       LPDIRECT3DVERTEXDECLARATION9  m_pDecl;
       IDirect3DVertexBuffer9*       QuadVB;

       // TEXTURES & SURFACES
       IDirect3DTexture9*            Scene_Tex;
       IDirect3DSurface9*            Scene_Surface;

       IDirect3DTexture9*            Reticle_Tex[100];
       IDirect3DSurface9*            Reticle_Surface[100];

       IDirect3DTexture9*            RT_Tex;
       IDirect3DSurface9*            RT_Surface;

       IDirect3DTexture9*            Pal_Tex[4];
       IDirect3DSurface9*            Pal_Surf[4];

       IDirect3DTexture9*            RT_Reduce_Tex[3];
       IDirect3DSurface9*            RT_Reduce_Surface[3];
//     IDirect3DTexture9*            RT_Reduce_Tex2[3];
//     IDirect3DSurface9*            RT_Reduce_Surface2[3];

       IDirect3DTexture9*            Ones_Tex;
       IDirect3DSurface9*            Ones_Surface;

       // transformation matrices
       D3DXMATRIX                    mWorld;
       D3DXMATRIX                    mView;
       D3DXMATRIX                    mProj;
```

```
        // --------------------- STRUCTS  ----------------------

        struct CUSTOMVERTEX
        {
                FLOAT       x;
                FLOAT       y;
        };

public:
        //constructor
        Gpu(HINSTANCE p_hInst, int p_nCmdShow, int p_SceneSize)
                :hInst (p_hInst), nCmdShow(p_nCmdShow), SceneSize (p_SceneSize/2)
        {
                Device =0;

                ScenePixels         = SceneSize*SceneSize;

                fPixSizeX           = -1.0f / (float)SceneSize;
                fPixSizeY           =  1.0f / (float) SceneSize;

                //VS1
                VS1_maddreduce      = 0;
                VS1_VSCT            = 0;
                VS1_PixelSizeHandle = 0;
                //PS1
                PS1_maddreduce      = 0;
                PS1_PSCT            = 0;
                //VS2
                VS2_16tapreduce     = 0;
                VS2_VSCT            = 0;
                VS2_offsetHandle    = 0;

                //PS2
                PS2_16tapreduce     = 0;
                PS2_PSCT            = 0;
                PS2_offsetHandle    = 0;
                // vertex buffer ptr
                QuadVB              = 0;

                if(!d3d::InitD3D(hInst, nCmdShow,
                          GPU_WINDOW_WIDTH, GPU_WINDOW_HEIGHT, true, D3DDEVTYPE_HAL, &Device))
                {
                          ::MessageBox(0, "InitD3D() - FAILED", 0, 0);
                }

                if(!Setup()){
                          ::MessageBox(0, "Setup() - FAILED", 0, 0);
                }

                DualRT = false;
                // modular actions depending on algorithm
                InitShaders();
                InitReticlesAndScene();
                InitRenderTargets_hybrid();

        }//Gpu() CONSTRUCTOR

private:
        //---------------------------------------------------------
        //                InitShaders()
        //                creates & compiles shaders
        //---------------------------------------------------------
        bool InitShaders(){
                HRESULT hr = 0;

                // *** PS1

                ID3DXBuffer* PSBuffer         = 0;
                ID3DXBuffer* errorBuffer      = 0;

                char pathPS1[50]= "";
                char pathVS1[50]= "";
                char* psbigtex = "source/ps_bigtex.txt";
                char* vsbigtex = "source/vs_bigtex.txt";
                char* psonebyone = "source/ps_onebyone.txt";
                char* vsonebyone = "source/vs_onebyone.txt";

                if (SceneSize == 256){
                          strcpy(pathPS1,psonebyone);
                          strcpy(pathVS1,vsonebyone);
                }
                else {
                          strcpy(pathPS1,psbigtex);
                          strcpy(pathVS1,vsbigtex);
                }


                hr = D3DXCompileShaderFromFile(
                          pathPS1,
                          0,
                          0,
                          "PSMain", // entry point function name
                          "ps_2_0",
                          D3DXSHADER_SKIPVALIDATION,//DEBUG, // | D3DXSHADER_SKIPOPTIMIZATION
                          &PSBuffer,
                          &errorBuffer,
                          &PS1_PSCT);

                // output any error messages
                if( errorBuffer )
                {
                          ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
                          Release<ID3DXBuffer*>(errorBuffer);
                }
                if(FAILED(hr))
                {
                          ::MessageBox(0, "PS1--D3DXCompileShaderFromFile() - FAILED", 0, 0);
```

```
		return false;
	}

	// create pixel shader
	hr = Device->CreatePixelShader(
		(DWORD*)PSBuffer->GetBufferPointer(),
		&PS1_maddreduce);

	if(FAILED(hr))
	{
		::MessageBox(0, "CreatePixelShader PS1 - FAILED", 0, 0);
		return false;
	}

	Release<ID3DXBuffer*>(PSBuffer);

	// *** PS2

	ID3DXBuffer* PS2Buffer        = 0;
	ID3DXBuffer* errorBuffer2 = 0;

	hr = D3DXCompileShaderFromFile(
		"source/ps_16tapredux_2.txt",
		0,
		0,
		"PSMain", // entry point function name
		"ps_2_0",
		D3DXSHADER_SKIPVALIDATION,//DEBUG, // | D3DXSHADER_SKIPVALIDATION OPTIMIZATION
		&PS2Buffer,
		&errorBuffer2,
		&PS2_PSCT);

	// output any error messages
	if( errorBuffer2 )
	{
		::MessageBox(0, (char*)errorBuffer2->GetBufferPointer(), 0, 0);
		Release<ID3DXBuffer*>(errorBuffer2);
	}
	if(FAILED(hr))
	{
		::MessageBox(0, "PS2--D3DXCompileShaderFromFile() - FAILED", 0, 0);
		return false;
	}

	// create pixel shader
	hr = Device->CreatePixelShader(
		(DWORD*)PS2Buffer->GetBufferPointer(),
		&PS2_16tapreduce);

	if(FAILED(hr))
	{
		::MessageBox(0, "CreatePixelShader PS2 - FAILED", 0, 0);
		return false;
	}

	Release<ID3DXBuffer*>(PS2Buffer);

	// *** VS1

	ID3DXBuffer* VSBuffer      = 0;
	ID3DXBuffer* errorBuffer3 = 0;

	hr = D3DXCompileShaderFromFile(
		pathVS1,
		0,
		0,
		"Main", // entry point function name
		"vs_2_0",
		D3DXSHADER_SKIPVALIDATION,//DEBUG, // | D3DXSHADER_SKIPOPTIMIZATION
		&VSBuffer,
		&errorBuffer3,
		&VS1_VSCT);

	// output any error messages
	if( errorBuffer3 )
	{
		::MessageBox(0, (char*)errorBuffer3->GetBufferPointer(), 0, 0);
		Release<ID3DXBuffer*>(errorBuffer3);
	}
	if(FAILED(hr))
	{
		::MessageBox(0, "VS1--D3DXCompileShaderFromFile() - FAILED", 0, 0);
		return false;
	}

	// create vertex shader
	hr = Device->CreateVertexShader(
		(DWORD*)VSBuffer->GetBufferPointer(),
		&VS1_maddreduce);

	if(FAILED(hr))
	{
		::MessageBox(0, "CreateVertexShader VS1 - FAILED", 0, 0);
		return false;
	}

	Release<ID3DXBuffer*>(VSBuffer);

	// *** VS2
	ID3DXBuffer* VS2Buffer      = 0;
	ID3DXBuffer* errorBuffer4 = 0;

	hr = D3DXCompileShaderFromFile(
		"source/vs_16tapredux_2.txt",
		0,
		0,
		"Main", // entry point function name
		"vs_2_0",
                D3DXSHADER_SKIPVALIDATION,//DEBUG, // | D3DXSHADER_SKIPOPTIMIZATION
                &VS2Buffer,
                &errorBuffer4,
                &VS2_VSCT);

        // output any error messages
        if( errorBuffer4 )
        {
                ::MessageBox(0, (char*)errorBuffer4->GetBufferPointer(), 0, 0);
                Release<ID3DXBuffer*>(errorBuffer4);
        }
        if(FAILED(hr))
        {
                ::MessageBox(0, "VS2--D3DXCompileShaderFromFile() - FAILED", 0, 0);
                return false;
        }

        // create vertex shader
        hr = Device->CreateVertexShader(
                (DWORD*)VS2Buffer->GetBufferPointer(),
                &VS2_16tapreduce);

        if(FAILED(hr))
        {
                ::MessageBox(0, "CreateVertexShader VS2 - FAILED", 0, 0);
                return false;
        }

        Release<ID3DXBuffer*>(VS2Buffer);

        //get VS1 pixelsize constant handle
        VS1_PixelSizeHandle = VS1_VSCT->GetConstantByName(0, "PixelSize");

        if (SceneSize != 256){
                PS1_PixelSizeHandle = PS1_PSCT->GetConstantByName(0, "PixelSize");
        }
        // get PS2 and VS2 const handles
        VS2_offsetHandle = VS2_VSCT->GetConstantByName(0, "offset");
        PS2_offsetHandle = PS2_PSCT->GetConstantByName(0, "offset");

        return true;

}// InitShaders()

//-----------------------------------------------------------
//      InitReticlesAndScene()
// loads reticle images into GPU, creates reticle and scene surfaces
//  and textures in GPU memory
//----------------------------------------------------------
bool InitReticlesAndScene(){
        //---------------------------------------------
        // create scene texture and surface
        //---------------------------------------------
        HRESULT hr = 0;
        hr = D3DXCreateTexture(
                Device,
                SceneSize, SceneSize,
                1, // no mipmap chain
                D3DUSAGE_DYNAMIC, //was 0--keep DYNAMIC!
                D3D_FORMAT,
                D3DPOOL_DEFAULT,
                &Scene_Tex);
        if(FAILED(hr))
                return false;

        //get interface to top level surface of Scene_Tex
        hr = Scene_Tex->GetSurfaceLevel(0,&Scene_Surface);
        if(FAILED(hr))
                return false;

        // create 100 reticle textures or 4 8x8 pallettes, depending
        //   on whether image size is 512 or 256/128
        if (SceneSize == 256){
                //create 100 individual reticle textures
                for (int i = 0; i<100; i++){

                        hr = D3DXCreateTexture(
                                Device,
                                SceneSize, SceneSize,
                                1, // no mipmap chain
                                0,//D3DUSAGE_DYNAMIC, //usage
                                D3D_FORMAT,
                                D3DPOOL_DEFAULT,
                                &Reticle_Tex[i]);
                        if(FAILED(hr))
                                return false;

                        //get interface to top level surface of each tex
                        hr = Reticle_Tex[i]->GetSurfaceLevel(0,&Reticle_Surface[i]);
                        if(FAILED(hr))
                                return false;
                }
        }
        else {
                //create 4 reticle pallette textures
                for (int i = 0; i<4; i++){

                        hr = D3DXCreateTexture(
                                Device,
                                SceneSize*8, SceneSize*8,
                                1, // no mipmap chain
                                0,//D3DUSAGE_DYNAMIC, //usage; DYNAMIC loads faster
                                D3D_FORMAT,
                                D3DPOOL_DEFAULT,
                                &Reticle_Tex[i]);
                        if(FAILED(hr))
                                return false;
                        //get interface to top level surface of each tex
                        hr = Reticle_Tex[i]->GetSurfaceLevel(0,&Reticle_Surface[i]);
                        if(FAILED(hr))
                                return false;

                        //create 4 dynamic textures in SYSTEMMEM to build pallettes

                        hr = D3DXCreateTexture(
                                Device,
                                SceneSize*8, SceneSize*8,
                                1, // no mipmap chain
                                D3DUSAGE_DYNAMIC, //can't be DYNAMIC and RT
                                D3D_FORMAT,
                                D3DPOOL_SYSTEMMEM,
                                &Pal_Tex[i]);
                        if(FAILED(hr))
                                return false;

                        //get interface to top level surface
                        hr = Pal_Tex[i]->GetSurfaceLevel(0, &Pal_Surf[i]);
                        if(FAILED(hr))
                                return false;

                }
        }

        return true;
}  // InitReticlesAndScene()

//-------------------------------------------------------
//                InitRenderTargets_hybrid()
//-------------------------------------------------------
bool InitRenderTargets_hybrid() {

        HRESULT hr = 0;

        //Set Init_RTSize -- the size of RT resulting from first mul-reduce op
        //Set ReduceIterations -- controls how many times the 16:1 reduce will be
        //                        run after the 1st mul-reduce has been done
        int Init_RTSize;

        if (SceneSize == 64){
                Init_RTSize = 256;
                ReduceIterations = 2;
        }
        else if (SceneSize == 128){
                Init_RTSize = 512;
                ReduceIterations = 3;
        }

        else {
                Init_RTSize = 1024;
                ReduceIterations = 3;
                start1 = 0;
                end1 = 40;
        }
        //Set OutTexSize--the size of the final RT we will get our
        //   result(s) from; affects GetRTData()
        OutTexSize = 8*SceneSize/(2*((int)pow(4,ReduceIterations)));
        OutPixels = OutTexSize*OutTexSize;

        //SET TexIndex-- the array index of the RT_Reduce_Surface[] that will contain
        //  the final result; affects GetRTData()
        TexIndex = ReduceIterations-1;

        // create initial RT (half the reticle pallette size because of 4:1 reduction)
        hr = Device->CreateTexture(Init_RTSize,Init_RTSize,1,
                                D3DUSAGE_RENDERTARGET, D3DFMT_R32F,
                                D3DPOOL_DEFAULT,&RT_Tex,0);//D3D_FORMAT
        if(FAILED(hr))
                        return false;

        //get interface to top level surface
        hr = RT_Tex->GetSurfaceLevel(0, &RT_Surface);
        if(FAILED(hr))
                return false;

        // create render targets for reduction op (2 or 3)
        int Size = Init_RTSize/4;
        for (int i=0; i<ReduceIterations; i++){
                hr = D3DXCreateTexture(
                        Device,
                        Size, Size,
                        1, // no mipmap chain
                        D3DUSAGE_RENDERTARGET, //can't be DYNAMIC and RT
                        D3DFMT_R32F,//D3D_FORMAT
                        D3DPOOL_DEFAULT,
                        &RT_Reduce_Tex[i]);
                if(FAILED(hr))
                        return false;

                //get interface to top level surface
                hr = RT_Reduce_Tex[i]->GetSurfaceLevel(0, &RT_Reduce_Surface[i]);
                if(FAILED(hr))
                        return false;

                Size /= 4;
        }

        //create SYSTEMMEM tex to send result(s) to
        hr = D3DXCreateTexture(
                        Device,
                        OutTexSize, OutTexSize,
                        1, // no mipmap chain
                        D3DUSAGE_DYNAMIC, // try dynamic and zero
                        D3DFMT_R32F, //D3D_FORMAT,
                        D3DPOOL_SYSTEMMEM,
                        &Ones_Tex);
```

```
          if(FAILED(hr))
                    return false;
          //get interface to top level surface of Ones_Tex[]
          hr = Ones_Tex->GetSurfaceLevel(0, &Ones_Surface);
          if(FAILED(hr))
                    return false;

          // *** OFFSET ARRAYS FOR 16:1 REDUCE -- sent to vs and ps to calculate texcoords for adjacent
          //  pixels in the 4x4 block

          // PixelSize of input texture to first reduce op
          float PixelSize2 = 1/(float)Init_RTSize;

          // calculate displacements
          for (int k = 0; k<ReduceIterations; k++){
                    for (int i = 0; i<4; i++){
                              for(int j = 0; j<4; j++){
                                        offset[k][i*4 +j] = D3DXVECTOR2(PixelSize2*(float)j, PixelSize2*(float)i);
                              }
                    }
                    PixelSize2 *= 4.0f;
          }

          return true;
}// InitRenderTargets_hybrid()

//-----------------------------------------------------------
//                                                         Setup()
// Initializes geometry, renderstate, calls
//   InitRenderTargets, InitShaders, InitReticlesAndScene
//-----------------------------------------------------------
bool Setup() {

          HRESULT hr = 0;

          //---------------- DISABLE unneeded processing ------------------
          // turn off Stencil and Culling
          hr = Device->SetDepthStencilSurface(
                    0);
          if(FAILED(hr))
                    return false;
          hr = Device->SetRenderState(D3DRS_CULLMODE,D3DCULL_NONE);
          if(FAILED(hr))
                    return false;
          // disable lighting

          Device->SetRenderState(D3DRS_LIGHTING, false);

          //------------------ create geometry ----------------------------

          D3DVERTEXELEMENT9 decl[]=
          {
          {0, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
          D3DDECL_END()
          };
          // declare the vertex structure
          hr = Device->CreateVertexDeclaration(decl, &m_pDecl);
          if(FAILED(hr))
                    return false;
          // create VB with only x,y position
          hr = Device->CreateVertexBuffer(        56 * sizeof(CUSTOMVERTEX), //was 4
                                                  D3DUSAGE_WRITEONLY,
                                                  0,
                                                  D3DPOOL_DEFAULT,
                                                  &QuadVB,
                                                  NULL);
          if(FAILED(hr))
                    return false;

          float left          = -1.00f;
          float right         =  1.00f;
          float top           =  1.00f;
          float bottom        = -1.00f;
          float top_row2      =  0.75f;
          float top_row3      =  0.50f;
          float top_row4      =  0.25f;
          float top_row5      =  0.00f;
          float top_row7      = -0.50f;
          float top_row8      = -0.75f;
          float bot_row1      =  0.75f;
          float bot_row2      =  0.50f;
          float bot_row4      =  0.00f;
          float bot_row5      = -0.25f;
          float bot_row6      = -0.50f;
          float bot_row7      = -0.75f;

          CUSTOMVERTEX* v;
          QuadVB->Lock(0, 56 * sizeof(CUSTOMVERTEX), (VOID**)&v, 0);//was 4
          //quad 0 full square
          // left bottom
          v[0].x = left;
          v[0].y = bottom;//bottom

          // left top
          v[1].x = left;
          v[1].y = top;

          // right bottom
          v[2].x = right;
          v[2].y = bottom;//bottom

          // right top
          v[3].x = right;
          v[3].y = top;

          //quad 1 r1-5
          // left bottom
          v[4].x = left;
```

```
v[4].y = bot_row5;//5

// left top
v[5].x = left;
v[5].y = top;

// right bottom
v[6].x = right;
v[6].y = bot_row5;//5

// right top
v[7].x = right;
v[7].y = top;

//quad 2 r2-6
// left bottom
v[8].x = left;
v[8].y = bot_row6;

// left top
v[9].x = left;
v[9].y = top_row2;

// right bottom
v[10].x = right;
v[10].y = bot_row6;

// right top
v[11].x = right;
v[11].y = top_row2;

//quad 3 r3-7
// left bottom
v[12].x = left;
v[12].y = bot_row7;

// left top
v[13].x = left;
v[13].y = top_row3;

// right bottom
v[14].x = right;
v[14].y = bot_row7;

// right top
v[15].x = right;
v[15].y = top_row3;

//quad 4 r4-8
// left bottom
v[16].x = left;
v[16].y = bottom;

// left top
v[17].x = left;
v[17].y = top_row4;

// right bottom
v[18].x = right;
v[18].y = bottom;

// right top
v[19].x = right;
v[19].y = top_row4;

//quad 5 r1-6
// left bottom
v[20].x = left;
v[20].y = bot_row6;

// left top
v[21].x = left;
v[21].y = top;

// right bottom
v[22].x = right;
v[22].y = bot_row6;

// right top
v[23].x = right;
v[23].y = top;

//quad 6 r2-7
// left bottom
v[24].x = left;
v[24].y = bot_row7;

// left top
v[25].x = left;
v[25].y = top_row2;

// right bottom
v[26].x = right;
v[26].y = bot_row7;

// right top
v[27].x = right;
v[27].y = top_row2;

//quad 7 r3-8
// left bottom
v[28].x = left;
v[28].y = bottom;

// left top
v[29].x = left;
v[29].y = top_row3;
```

```
// right bottom
v[30].x = right;
v[30].y = bottom;

// right top
v[31].x = right;
v[31].y = top_row3;

//quad 8 T1
// left bottom
v[32].x = left;
v[32].y = bot_row1;

// left top
v[33].x = left;
v[33].y = top;

// right bottom
v[34].x = right;
v[34].y = bot_row1;

// right top
v[35].x = right;
v[35].y = top;

//quad 9 B2
// left bottom
v[36].x = left;
v[36].y = bottom;

// left top
v[37].x = left;
v[37].y = top_row7;

// right bottom
v[38].x = right;
v[38].y = bottom;

// right top
v[39].x = right;
v[39].y = top_row7;

//quad 10 T4
// left bottom
v[40].x = left;
v[40].y = bot_row4;

// left top
v[41].x = left;
v[41].y = top;

// right bottom
v[42].x = right;
v[42].y = bot_row4;

// right top
v[43].x = right;
v[43].y = top;

//quad 11 B1
// left bottom
v[44].x = left;
v[44].y = bottom;

// left top
v[45].x = left;
v[45].y = top_row8;

// right bottom
v[46].x = right;
v[46].y = bottom;

// right top
v[47].x = right;
v[47].y = top_row8;

//quad 12 B4
// left bottom
v[48].x = left;
v[48].y = bottom;

// left top
v[49].x = left;
v[49].y = top_row5;

// right bottom
v[50].x = right;
v[50].y = bottom;

// right top
v[51].x = right;
v[51].y = top_row5;

//quad 13 T2
// left bottom
v[52].x = left;
v[52].y = bot_row2;

// left top
v[53].x = left;
v[53].y = top;

// right bottom
v[54].x = right;
v[54].y = bot_row2;

// right top
v[55].x = right;
```

```
                v[55].y = top;

                QuadVB->Unlock();

                // set vertex declaration  (will not change again)
                Device->SetVertexDeclaration(m_pDecl);

                if (SceneSize == 256){
                          // set geometry (will not change again)
                          Device->SetStreamSource(0, QuadVB, 0, sizeof(CUSTOMVERTEX));
                }
                return true;
        }//Setup()


private:
        //------------------------------------------------------------
        // ---------------------- LOAD INPUT SCENE -------------------
        //------------------------------------------------------------
        bool LoadInputScene(float p_inputArray[]) {
                RECT SurfRect;
                SurfRect.left       = 0;
                SurfRect.top        = 0;
                SurfRect.right      = SceneSize;
                SurfRect.bottom     = SceneSize;

                HRESULT hr = 0;
                hr= D3DXLoadSurfaceFromMemory(
                          Scene_Surface,
                          0,
                          0,
                          p_inputArray,
                          D3D_FORMAT,
                          (16*SceneSize),  //16 for 4x32-bit, 8 for 2x32-bit, 4 for 1x32F format
                          0,
                          &SurfRect,
                          D3DX_FILTER_NONE,
                          0);
                if(FAILED(hr))
                          return false;
                return true;
        }//LoadInputScene()

        //----------------------------------------------------------------
        //        Redux()
        //        16:1 REDUCE OPERATION
        //        Uses files:  vs_16tapredux_2.txt and ps_16tapredux_2.txt
        //----------------------------------------------------------------
        bool Redux() {
                HRESULT hr = 0;

                // ---- set PS and VS shaders
                Device->SetVertexShader(VS2_16tapreduce);
                Device->SetPixelShader (PS2_16tapreduce);

                // initial source tex is result of maddreduce op
                Device->SetTexture( 0, RT_Tex);

                for (int i = 0; i < ReduceIterations; i++) {

                          hr = Device->SetRenderTarget( 0, RT_Reduce_Surface[i]);
                          if(FAILED(hr))
                                    return false;

                          if (i>0)
                                    Device->SetTexture( 0, RT_Reduce_Tex[i-1]);

                          // set VS offset constant array
                          hr =      VS2_VSCT->SetFloatArray(      Device,
                                                                  VS2_offsetHandle,
                                                                  (float*)&offset[i][0],
                                                                  16 );// 2*8 floats
                          if(FAILED(hr))
                                    return false;

                          // set PS offset constant array
                          hr =      PS2_PSCT->SetFloatArray(      Device,
                                                                  PS2_offsetHandle,
                                                                  (float*) &offset[i][8],
                                                                  16 );//2*8 floats
                          if(FAILED(hr))
                                    return false;

                          // render-- do 16:1 reduction on source image

                          //Device->Clear(0,0,D3DCLEAR_TARGET,0L,0,0);
                          Device->BeginScene();
                                    Device->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
                          Device->EndScene();
                }
                return true;
        } // Redux()

        //-----------------------------------------------------------
        //        bool MAddReduce()
        // for ONEBYONE (Palette) approach
        // uses files:  vs_onebyone.txt and ps_onebyone.txt
        //-----------------------------------------------------------
        bool MAddReduce(int p_retIndexStart){
                HRESULT hr = 0;

                // ---- set PS and VS shaders
                Device->SetVertexShader(VS1_maddreduce);
                Device->SetPixelShader(PS1_maddreduce);

                // set VS PixelSize const
                // PixelSizeX & Y initialized as Global constants
                hr =      VS1_VSCT->SetVector(Device, VS1_PixelSizeHandle,
                              &D3DXVECTOR4(fPixSizeX, fPixSizeY, 1.0f, 1.0f));
          if(FAILED(hr))
                    return false;

          // ---- set RT
          hr = Device->SetRenderTarget( 0, RT_Surface);
          if(FAILED(hr))
                    return false;

          Device->SetStreamSource(0, QuadVB, 0, sizeof(CUSTOMVERTEX));

          D3DVIEWPORT9 vp;

          vp.Width  = SceneSize/2;
          vp.Height = SceneSize/2;
          vp.MinZ   = 0.0f;
          vp.MaxZ   = 1.0f;

          Device->SetTexture( 0, Scene_Tex);//stage 0 = input scene

          for (int v = 0; v<5; v++){
                    for (int h = 0; h<8; h++){
                              vp.X = h*SceneSize/2;
                              vp.Y = v*SceneSize/2;
                              Device->SetViewport(&vp);
                              // set sampler 1 with reticle image for maddredux with scene
                              Device->SetTexture( 1, Reticle_Tex[(p_retIndexStart + (v*8+h))%100] );

                              // render-- madd scene with a single reticle image
                              Device->BeginScene();
                                        Device->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
                              Device->EndScene();
                    }
          }
          // set streamsource to 8x5 rectangle r1-r5, Quad 1
          Device->SetStreamSource(0, QuadVB, 4*sizeof(CUSTOMVERTEX), sizeof(CUSTOMVERTEX));
          return true;
} //MAddReduce()

//-----------------------------------------------------------------
//                                                  MAddReduce_hybrid()
// For 128 and 256 input scene sizes.
// multiplies input scene with a 8x8 reticle pallette using WRAPing when
// sampling the scene and does 4:1 reduction.  Scene width is 1/8 the width of the pallette.
// Adjusts size of rendering rectangle to cut out unneeded calculations.
// This rendering rectangle remains set for the reduction op, too.
//
// Uses files:  vs_bigtex.txt and ps_bigtex.txt
//-----------------------------------------------------------------
bool MAddReduce_hybrid(int p_retIndex){

          HRESULT hr = 0;
          int row = 0;
          int col = 0;
          int diff = p_retIndex;//=0

          //determines which of the 4 pallettes to use,
          // ensuring 40 contiguous retiles present in the pallette
          //set pallette as texture stage 0
          if (p_retIndex <= 24){
                    Device->SetTexture( 0, Reticle_Tex[0]);
                    row = p_retIndex/8;
                    col = p_retIndex%8;
          }
          else if (p_retIndex<=49){
                    Device->SetTexture( 0, Reticle_Tex[1]);
                    diff = p_retIndex-25;
                    row =  diff/8;
                    col = diff%8;
          }
          else if (p_retIndex<=74){
                    Device->SetTexture( 0, Reticle_Tex[2]);
                    diff = p_retIndex-50;
                    row = diff/8;
                    col = diff%8;
          }
          else {
                    Device->SetTexture( 0, Reticle_Tex[3]);
                    diff = p_retIndex-75;
                    row = diff/8;
                    col = diff%8;
          }

          //new code
          start1 = diff;
          end1 = diff+40;
          //

          if ( col == 0)
                    Device->SetStreamSource(0, QuadVB, (row+1)*4*sizeof(CUSTOMVERTEX), sizeof(CUSTOMVERTEX));
          else
                    Device->SetStreamSource(0, QuadVB, (row+1+4)*4*sizeof(CUSTOMVERTEX), sizeof(CUSTOMVERTEX));

          //set VS and PS
          Device->SetVertexShader(VS1_maddreduce);
          Device->SetPixelShader (PS1_maddreduce);

          hr =      VS1_VSCT->SetVector(Device, VS1_PixelSizeHandle,
                              &D3DXVECTOR4(1.0f/(float)(SceneSize*8),
                              1.0f/(float)SceneSize, 0.0f, 1.0f));

          // ---set tex stage 1 to be input scene
          Device->SetTexture(1,Scene_Tex);

          hr = Device->SetRenderTarget( 0, RT_Surface);
          if(FAILED(hr))
                    return false;
          // render: multiply input scene with reticle image and do 4:1 reduction

          //Device->Clear(0,0,D3DCLEAR_TARGET,0L,0,0);
          Device->BeginScene();
                    Device->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);//was 0
          Device->EndScene();

          return true;
}//MAddReduce_hybrid()



//-------------------------------------------------------------------
//                                                  GetRTData_hybrid()
//       retrieve FINAL data from lockable render-to surface
//-------------------------------------------------------------------
void GetRTData_hybrid(double p_b[]) {
          HRESULT hr = 0;

          int q;
          int col;
          int row;

          D3DLOCKED_RECT lockedRect;

          Device->GetRenderTargetData(RT_Reduce_Surface[TexIndex], Ones_Surface);
          Ones_Surface->LockRect(&lockedRect,
                    0 , //lock entire tex
                    D3DLOCK_READONLY ); //flags

          //D3DXVECTOR4* imageData = (D3DXVECTOR4*) lockedRect.pBits;
          float* imageData = (float*) lockedRect.pBits;

          //perform final 4:1 reduction if necessary and add up 4 components of each pixel
          if (OutTexSize>8){
                    for (int i = start1; i<end1; i++){

                              row = i/8;
                              col = i%8;
                              q = row*32 + col*2;
                              p_b[i-start1]=      imageData[q]+
                                                  imageData[q+1]  +
                                                  imageData[q+16] +
                                                  imageData[q+17];

                    }
          }
          else {
                    for (int i = start1; i<end1; i++){
                              p_b[i-start1]=imageData[i]; //.x + imageData[i].y +imageData[i].z + imageData[i].w;
                    }
          }
          Ones_Surface->UnlockRect();


          return;
}// GetRTData_hybrid()

//-------------------------------------------------------
//                              Release() and Delete()
// cleanup functions
//-------------------------------------------------------
template<class T> void Release(T t)     {
          if( t )   {
                    t->Release();
                    t = 0;
          }
}

template<class T> void Delete(T t){
          if( t ){
                    delete t;
                    t = 0;
          }
}

//-----------------------------------------------------------
//                                        Cleanup()
// releases textures/surfaces/interfaces/devices/memory
//  allocated during program
//-----------------------------------------------------------
void Cleanup()
{
          //vertex buffer and declaration
          Release<IDirect3DVertexBuffer9*>(QuadVB);
          Release<LPDIRECT3DVERTEXDECLARATION9>   (m_pDecl);

          //textures and surfaces
          Release<IDirect3DSurface9*>(Scene_Surface);
          Release<IDirect3DTexture9*>(Scene_Tex);

          int n = 4;
          if (SceneSize==256) n=100;
          for (int t = 0; t<n;t++){
                    Release<IDirect3DSurface9*>(Reticle_Surface[t]);
                    Release<IDirect3DTexture9*>(Reticle_Tex[t]);
          }

          //initial RT
          Release<IDirect3DTexture9*>(RT_Tex);
          Release<IDirect3DSurface9*>(RT_Surface);


          for (int t = 0; t<ReduceIterations; t++) {
                    Release<IDirect3DSurface9*>(RT_Reduce_Surface[t]);
                    Release<IDirect3DTexture9*>(RT_Reduce_Tex[t]);
          }
```

```
                Release<IDirect3DTexture9*>(Ones_Tex);
                Release<IDirect3DSurface9*>(Ones_Surface);

                //PS & VS
                Release<IDirect3DPixelShader9*>(PS1_maddreduce);
                Release<ID3DXConstantTable*>(PS1_PSCT);
                Release<IDirect3DVertexShader9*>(VS1_maddreduce);
                Release<ID3DXConstantTable*>(VS1_VSCT);
                Release<IDirect3DPixelShader9*>(PS2_16tapreduce);
                Release<ID3DXConstantTable*>(PS2_PSCT);
                Release<IDirect3DVertexShader9*>(VS2_16tapreduce);
                Release<ID3DXConstantTable*>(VS2_VSCT);

                Device->Release();

        }// Cleanup()

public:
        //----------------------------------------------------------
        //                                          Process()
        // user interface to GPU algorithm
        // input:  reference to scene image array variable
        // input:  starting index in reticle pallette
        // output: void (but 40 dot-product results are loaded to user array)
        void Process(int p_retIndex, float p_SceneArray[], double p_b[]) {
                LoadInputScene(p_SceneArray);
                if (SceneSize == 256){
                        //do one at a time algorithm
                        MAddReduce(p_retIndex);
                }
                else{
                        //do big tex
                        MAddReduce_hybrid(p_retIndex);
                }
                Redux();
                GetRTData_hybrid(p_b);
                return;
        }// Process()

        //----------------------------------------------------------------
        //              uploadReticle
        //----------------------------------------------------------------
        bool uploadReticle(int p_index, float p_array[]){
                HRESULT hr = 0;
                RECT rect;
                RECT srcRect;
                srcRect.top = 0;
                srcRect.bottom = SceneSize;
                srcRect.left = 0;
                srcRect.right = SceneSize;

                if (SceneSize == 256){
                        rect.left = 0;
                        rect.right = SceneSize;
                        rect.top = 0;
                        rect.bottom = SceneSize;

                        hr= D3DXLoadSurfaceFromMemory(
                                Reticle_Surface[p_index],
                                0,
                                0,
                                p_array,
                                D3D_FORMAT,
                                (16*SceneSize),  //16 for 4x32-bit, 8 for 2x32-bit, 4 for 1x32F format
                                0,
                                &srcRect,
                                D3DX_FILTER_NONE,
                                0);
                        if(FAILED(hr))
                                return false;
                }
                else {


                        if ( (p_index<=63) && (p_index >=0)){
                                rect.top    = SceneSize*(p_index/8);
                                rect.left   = SceneSize*(p_index%8);
                                rect.bottom = rect.top+SceneSize;
                                rect.right  = rect.left+SceneSize;

                                hr= D3DXLoadSurfaceFromMemory(
                                        Pal_Surf[0],//Reticle_Surface[0],
                                        0,
                                        &rect, //dest rect
                                        p_array,
                                        D3D_FORMAT,
                                        (16*SceneSize),  //16 for 4x32-bit, 8 for 2x32-bit, 4 for 1x32F format
                                        0,
                                        &srcRect,
                                        D3DX_FILTER_NONE,
                                        0);
                                if(FAILED(hr))
                                        return false;
                        }
                        if ((p_index>=25) && (p_index<=88)){
                                rect.top    = SceneSize*( (p_index-25)/8);
                                rect.left   = SceneSize*( (p_index-25)%8);
                                rect.bottom = rect.top+SceneSize;
                                rect.right  = rect.left+SceneSize;

                                hr= D3DXLoadSurfaceFromMemory(
                                        Pal_Surf[1],//Reticle_Surface[1],
                                        0,
                                        &rect, //dest rect
                                        p_array,
                                        D3D_FORMAT,
                                        (16*SceneSize),  //16 for 4x32-bit, 8 for 2x32-bit, 4 for 1x32F format
                                        0,
                                        &srcRect,
                                        D3DX_FILTER_NONE,
                                        0);
                                if(FAILED(hr))
                                        return false;
                        }
                        if ( (((p_index+50)%100)>=0) && ( ((p_index+50)%100)<=63)) {
                                rect.top    = SceneSize*( ((p_index+50)%100)/8);
                                rect.left   = SceneSize*( ((p_index+50)%100)%8);
                                rect.bottom = rect.top+SceneSize;
                                rect.right  = rect.left+SceneSize;

                                hr= D3DXLoadSurfaceFromMemory(
                                        Pal_Surf[2],//Reticle_Surface[2],
                                        0,
                                        &rect,
                                        p_array,
                                        D3D_FORMAT,
                                        (16*SceneSize),  //16 for 4x32-bit, 8 for 2x32-bit, 4 for 1x32F format
                                        0,
                                        &srcRect,
                                        D3DX_FILTER_NONE,
                                        0);
                                if(FAILED(hr))
                                        return false;
                        }
                        if ( (((p_index+25)%100)>=0) && ( ((p_index+25)%100)<=63)) {
                                rect.top    = SceneSize*( ((p_index+25)%100)/8);
                                rect.left   = SceneSize*( ((p_index+25)%100)%8);
                                rect.bottom = rect.top+SceneSize;
                                rect.right  = rect.left+SceneSize;

                                hr= D3DXLoadSurfaceFromMemory(
                                        Pal_Surf[3],//Reticle_Surface[3],
                                        0,
                                        &rect,
                                        p_array,
                                        D3D_FORMAT,
                                        (16*SceneSize),  //16 for 4x32-bit, 8 for 2x32-bit, 4 for 1x32F format
                                        0,
                                        &srcRect,
                                        D3DX_FILTER_NONE,
                                        0);
                                if(FAILED(hr))
                                        return false;
                        }
                        if (p_index == 99){
                                for (int i = 0; i<4;i++){
                                        Device->UpdateTexture(Pal_Tex[i],Reticle_Tex[i]);
                                        Release<IDirect3DSurface9*>(Pal_Surf[i]);
                                        Release<IDirect3DTexture9*>(Pal_Tex[i]);
                                }
                        }

                }
                return true;
        } // uploadReticle()

        int GetAlg(){
                if (SceneSize == 256)
                        return 2;//one by one
                else
                        return 1; //big tex
        }
        //-------------------------------------------------------------
        //                ~ Gpu()  DESTRUCTOR
        ~Gpu() {
                Cleanup();
        }// ~ Gpu()  DESTRUCTOR

};
#endif  // GPU_CLASS_H_BY_MAJ_JEFFERS
```

```
//
// file: vs_bigtex.txt
// BY MAJ SEAN JEFFERS
// 11 nov 04 -- multiplies 1x1 scene by 8x8 reticle pallette, then does
//               4:1 redux; results in RT that is quarter sized of pallette;
//                 sampling of scene done w/wrapping
//            -- PixelSize.x = 1/ret pallette width
//            -- PixelSize.y = 1/scene width
//            -- PixelSize.z = 0.0f (must!)
//            -- VS generates 8 texcoords for PS
// -------------------------------------------------------------

uniform float4  PixelSize;

// structures

struct VS_INPUT
{
       float4 Pos : POSITION;
};

struct VS_OUTPUT
{
       float4 Pos : POSITION;
       float2 Tex : TEXCOORD0;
       float2 Tex1: TEXCOORD1;
       float2 Tex2: TEXCOORD2;
       float2 Tex3: TEXCOORD3;
       float2 Tex4: TEXCOORD4;
       float2 Tex5: TEXCOORD5;
       float2 Tex6: TEXCOORD6;
       float2 Tex7: TEXCOORD7;
};


// -------------------------------------------------------------
// vertex shader function (input channels)
// -------------------------------------------------------------
VS_OUTPUT Main(VS_INPUT input)
{
       VS_OUTPUT output = (VS_OUTPUT)0;

       output.Pos.xy = input.Pos.xy;// + PixelSize.xy;
       output.Pos.z = 0.5f;
       output.Pos.w = 1.0f;

       output.Tex =  float2(0.5f, -0.5f) * input.Pos.xy + 0.5f.xx ;
       output.Tex1 = output.Tex + PixelSize.xz;
        output.Tex2 = output.Tex + PixelSize.zx;
        output.Tex3 = output.Tex + PixelSize.xx;

       output.Tex4 = 8.0f*output.Tex;
       output.Tex5 = output.Tex4 + PixelSize.yz;
        output.Tex6 = output.Tex4 + PixelSize.zy;
       output.Tex7 = output.Tex4 + PixelSize.yy;
    return output;
}
```

```
// file: ps_bigtex.txt
// depends on file:  vs_bigtex.txt
// BY MAJ SEAN JEFFERS
// 11 nov 04 -- multiplies 1x1 scene by 8x8 reticle pallette, then does
//              4:1 redux; results in RT that is quarter size of pallette;
//               sampling of small texture done w/wrapping
// 27 dec 04 -- modified to have AGRB32 in and R32F out with dot product
//
// ---------------------------------------------------------------
// globals
// ---------------------------------------------------------------

sampler Rendersampler;  // 8x8 reticle pallette (big texture)
sampler Rendersampler1; // scene 1x1 small texture

// -------------------------------------------
// structures
// -------------------------------------------

struct PS_INPUT
{
       float2 Tex : TEXCOORD0;
       float2 Tex1: TEXCOORD1;
       float2 Tex2: TEXCOORD2;
       float2 Tex3: TEXCOORD3;
       float2 Tex4: TEXCOORD4;
       float2 Tex5: TEXCOORD5;
       float2 Tex6: TEXCOORD6;
       float2 Tex7: TEXCOORD7;
};

struct PS_OUTPUT
{
       float4 clr : COLOR; //was COLOR0
};

// ---------------------------------------------------------------
// Pixel Shader (input channels):output channel
// ---------------------------------------------------------------

PS_OUTPUT PSMain(PS_INPUT input)
{
       PS_OUTPUT output = (PS_OUTPUT) 0;

       float4 a = tex2D(Rendersampler, input.Tex);
       float4 b = tex2D(Rendersampler, input.Tex1);
       float4 c = tex2D(Rendersampler, input.Tex2);
       float4 d = tex2D(Rendersampler, input.Tex3);

       float4 e = tex2D(Rendersampler1, input.Tex4);
       float4 f = tex2D(Rendersampler1, input.Tex5);
       float4 g = tex2D(Rendersampler1, input.Tex6);
       float4 h = tex2D(Rendersampler1, input.Tex7);

       output.clr = dot(a,e) +  dot(b,f) + dot(c,g) + dot(d,h);
                 //a*e + b*f + c*g + d*h;

       return output;
}
```

```
// file: vs_onebyone.txt  (was vs_experimental.txt)
// BY MAJ SEAN JEFFERS
// used by:  GPU_combined.h and GPU_CLASS_ONEBYONE_R32F.h
//
// 17 oct 04-- use PixelSize.y for Tex2-4 components instead of -.x
// 27 dec 04 -- renamed to vs_onebyone.txt
// -------------------------------------------------------------
// variables that are provided by the application
// -------------------------------------------------------------

uniform float4  PixelSize;

// structures

struct VS_INPUT
{
       float4 Pos : POSITION;
};

struct VS_OUTPUT
{
       float4 Pos : POSITION;
       float2 Tex : TEXCOORD0;
       float2 Tex2: TEXCOORD1;
       float2 Tex3: TEXCOORD2;
       float2 Tex4: TEXCOORD3;


};


// -------------------------------------------------------------
// vertex shader function (input channels)
// -------------------------------------------------------------
VS_OUTPUT Main(VS_INPUT input)
{
       VS_OUTPUT output = (VS_OUTPUT)0;

       output.Pos.xy = input.Pos.xy + PixelSize.xy;
       output.Pos.z = 0.5f;
       output.Pos.w = 1.0f;

       float2 Tex = float2(0.5f, -0.5f) * input.Pos.xy + 0.5f.xx  ;


       output.Tex  = Tex;
       output.Tex2 = Tex + float2(PixelSize.y, 0.0f);//use .y instead of -.x
       output.Tex3 = Tex + float2(0.0f, PixelSize.y);
       output.Tex4 = Tex + float2(PixelSize.y,PixelSize.y);


    return output;
}
```

```
// file: ps_onebyone.txt  (was ps_maddreduce_new.txt)
// PS for mult, add, reduce, 4:1; 1 scene tex madd with 6 reticles,
//   then result of last madd added in
//  BY MAJ SEAN JEFFERS
// 1 oct 04 -- 1st ver. had 2 samplers, v2 had 8
// 2 oct 04 -- modified for t1 * sum(t2-t7) + t8, where t8 result of
//             last pass; eliminates need for a 3rd PS/VS
// 5 oct 04 -- changed to add only single pixel from t8 (previous result) texture
//             because it is already a smaller, reduced texture
// 14 oct 04 -- removed output struct
// 17 oct 04 -- changed data type to float4 instead of vector
//
// new file name: ps_maddredce_new.txt
// 27 oct 04 -- this new version has only 2 samplers and no addback of
//               previous results
// 27 dec    -- changed to "dot" to accommodate R32F
// ---------------------------------------------------------------
// globals
// ---------------------------------------------------------------


sampler Rendersampler;
sampler Rendersampler2;


// -------------------------------------------
// structures
// -------------------------------------------

struct PS_INPUT
{
       float2 Tex : TEXCOORD0;
       float2 Tex2 : TEXCOORD1;
       float2 Tex3 : TEXCOORD2;
       float2 Tex4 : TEXCOORD3;

};

// struct PS_OUTPUT
// {
//     float4 clr : COLOR; //was COLOR0
// };

// ---------------------------------------------------------------
// Pixel Shader (input channels):output channel
// ---------------------------------------------------------------

float4 PSMain(PS_INPUT input) :COLOR
{

       float4 t1a = tex2D(Rendersampler, input.Tex);//float4
       float4 t1b = tex2D(Rendersampler, input.Tex2);
       float4 t1c = tex2D(Rendersampler, input.Tex3);
       float4 t1d = tex2D(Rendersampler, input.Tex4);

       float4 t2a = tex2D(Rendersampler2, input.Tex);
       float4 t2b = tex2D(Rendersampler2, input.Tex2);
       float4 t2c = tex2D(Rendersampler2, input.Tex3);
       float4 t2d = tex2D(Rendersampler2, input.Tex4);


       // madd src (t1) & one reticles (t2)

       float4 p1 = dot(t1a, t2a); //t1a*t2a; was float4

       float4 p2 = dot(t1b, t2b); //t1b*t2b;

       float4 p3 = dot(t1c, t2c); //t1c*t2c;

       float4 p4 = dot(t1d, t2d); //t1d*t2d ;

       //new
       return p1 + p2 + p3 + p4;

}
```

```
//
// VS_16tapredux_2.txt
// BY MAJ SEAN JEFFERS
// 1 oct 04 -- modified to output 4 texcoords for block-of-4 reduction op
//             note x-displacement is negated
// 3 oct 04 -- trying original approach to see if reduce error
// 17 oct 04 -- changed offset array size to [8] from [16]
//           -- removed redundant PixelSize
// -------------------------------------------------------------
// variables that are provided by the application
// -------------------------------------------------------------

//float4 PixelSize;

float2 offset[8]; //was 16


// structures

struct VS_INPUT
{
       float4 Pos : POSITION;
};

struct VS_OUTPUT
{
       float4 Pos : POSITION;

       float2 Tex0 : TEXCOORD0;
       float2 Tex1 : TEXCOORD1;
       float2 Tex2 : TEXCOORD2;
       float2 Tex3 : TEXCOORD3;
       float2 Tex4 : TEXCOORD4;
       float2 Tex5 : TEXCOORD5;
       float2 Tex6 : TEXCOORD6;
       float2 Tex7 : TEXCOORD7;
};


// -------------------------------------------------------------
// vertex shader function (input channels)
// -------------------------------------------------------------

VS_OUTPUT Main(VS_INPUT input)
{
       VS_OUTPUT output = (VS_OUTPUT)0;

       output.Pos.xy = input.Pos.xy ;//+  float2(-offset[1].x, offset[1].x);  //PixelSize.xy;
       output.Pos.z = 0.5f;
       output.Pos.w = 1.0f;

       float2 Tex = float2(0.5f, -0.5f) * input.Pos.xy + 0.5f.xx  ;
       //Tex *=  float2(1.0f,0.5f); //added float to test subset 12 nov

       output.Tex0 = Tex ;
       output.Tex1 = Tex + offset[1];
       output.Tex2 = Tex + offset[2];
       output.Tex3 = Tex + offset[3];
       output.Tex4 = Tex + offset[4];
       output.Tex5 = Tex + offset[5];
       output.Tex6 = Tex + offset[6];
       output.Tex7 = Tex + offset[7];


       return output;
}
```

```
// PS 16:1 reduce
//
// file: ps_16tapredux_2.txt
// BY MAJ SEAN JEFFERS
// 3 0ct 04 - trying original approach to see if reduce error
// 17 oct 04 -change offset array size to [8] from [16]
// --------------------------------------------------------------
// globals
// --------------------------------------------------------------


uniform float2 offset[8];


sampler Rendersampler;

// -------------------------------------------
// structures
// -------------------------------------------

struct PS_INPUT
{
       float2 Tex0 : TEXCOORD0;
       float2 Tex1 : TEXCOORD1;
       float2 Tex2 : TEXCOORD2;
       float2 Tex3 : TEXCOORD3;
       float2 Tex4 : TEXCOORD4;
       float2 Tex5 : TEXCOORD5;
       float2 Tex6 : TEXCOORD6;
       float2 Tex7 : TEXCOORD7;

};


// --------------------------------------------------------------
// Pixel Shader (input channels):output channel
// --------------------------------------------------------------

float4 PSMain(PS_INPUT input) : COLOR0
{
       //PS_OUTPUT output = (PS_OUTPUT) 0;


       float4 ColorSum = 0.0f;

       // sample first 8 taps (first 2 rows of 4x4 block)

       float4 c0 = tex2D(Rendersampler, input.Tex0);
       float4 c1 = tex2D(Rendersampler, input.Tex1);
       float4 c2 = tex2D(Rendersampler, input.Tex2);
       float4 c3 = tex2D(Rendersampler, input.Tex3);

       float4 c4 = tex2D(Rendersampler, input.Tex4);
       float4 c5 = tex2D(Rendersampler, input.Tex5);
       float4 c6 = tex2D(Rendersampler, input.Tex6);
       float4 c7 = tex2D(Rendersampler, input.Tex7);

       // add color values of first 8 taps

       ColorSum += c0;
       ColorSum += c1;
       ColorSum += c2;
       ColorSum += c3;
       ColorSum += c4;
       ColorSum += c5;
       ColorSum += c6;
       ColorSum += c7;


       // calculate texcoords for remaining 8 taps

       float2 Tap8  = input.Tex0 + offset[0]; //was 8-15
       float2 Tap9  = input.Tex0 + offset[1];
       float2 Tap10 = input.Tex0 + offset[2];
       float2 Tap11 = input.Tex0 + offset[3];

       float2 Tap12 = input.Tex0 + offset[4];
       float2 Tap13 = input.Tex0 + offset[5];
       float2 Tap14 = input.Tex0 + offset[6];
       float2 Tap15 = input.Tex0 + offset[7];

       // sample remaining 8 taps

       c0 = tex2D(Rendersampler, Tap8);
       c1 = tex2D(Rendersampler, Tap9);
       c2 = tex2D(Rendersampler, Tap10);
       c3 = tex2D(Rendersampler, Tap11);

       c4 = tex2D(Rendersampler, Tap12);
       c5 = tex2D(Rendersampler, Tap13);
       c6 = tex2D(Rendersampler, Tap14);
       c7 = tex2D(Rendersampler, Tap15);


       // add last 8 taps to sum

       ColorSum += c0;
       ColorSum += c1;
       ColorSum += c2;
       ColorSum += c3;
       ColorSum += c4;
       ColorSum += c5;
       ColorSum += c6;
       ColorSum += c7;

       return ColorSum;
}
```

```
// win_CONSCAN.cpp : Defines the entry point for the application.
// BY MAJ SEAN JEFFERS  --tests conscan gpu code

#include "stdafx.h"
#include "win_CONSCAN.h"
#define MAX_LOADSTRING 100

#include <iostream>
#include <fstream>
#include <iomanip>

#include <cmath>

#include "GPU_CONSCAN.h" //combined.h or CLASS_ONEBYONE.h CLASS_ONEBYONE_R32F.h

// Global Variables:
HINSTANCE hInst;                                        // current instance
TCHAR szTitle[MAX_LOADSTRING];                          // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];                    // the main window class name

const int        EXPER              = 100;
const int        SCENE_SIZE         = 0;
const int        SCENE_SIZE_X       = 512;
const int        SCENE_SIZE_Y       = 512;
const int        RETICLE_SIZE       = 128;
const int        WL                 = 1;
const char*      BUS_str            = "PCI-e";
const char*      APRCH_str          = "ATI";
const int        REPS               = 2;
const int SIZE_SQ = SCENE_SIZE_X*SCENE_SIZE_Y;
const int RET_SIZE_SQ  = RETICLE_SIZE*RETICLE_SIZE;

// WL 3 pt source vars
const int xmin = SCENE_SIZE_X/4;
const int ymin = SCENE_SIZE_Y/4;
const int xmax = SCENE_SIZE-xmin;
const int ymax = SCENE_SIZE-ymin;
//initial conditions
int oldx = xmin;
int oldy = ymin;
int delx = -1;
int xinc = -1;
int dely = 0;
int yinc = -1;

// Forward declarations of functions included in this code module:
void    UpdateScene(int , float*);
ATOM    MyRegisterClass(HINSTANCE hInstance);
BOOL    InitInstance(HINSTANCE, int);
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR    lpCmdLine,
                     int       nCmdShow)
{
        float reticle[RET_SIZE_SQ];
        float scene[SIZE_SQ];
        double answer[40];
        int retIndex[40];
        int xdisp[40];
        int ydisp[40];

        double times[REPS];
        double timeslog[REPS];
        double startTime;
        double endTime;

        double sum          = 0.0;
        double mean         = 0.0;
        double var          = 0.0;
        double stdev        = 0.0;
        double hi           = 0.0;
        double low          = 0.0;
        double sos          = 0.0;

        double sumlog       = 0.0;
        double meanlog      = 0.0;
        double varlog       = 0.0;
        double stdevlog     = 0.0;
        double soslog       = 0.0;
        double hilog        = 0.0;
        double lowlog       = 0.0;

        char WL_str[30];
        if (WL ==1){
                  strcpy(WL_str,"1 - non-changing");
        }
        else if (WL == 2){
                  strcpy(WL_str,"2 - fully-changing");
        }
        else {
                  strcpy(WL_str,"3 - moving pt source");
        }

        //instantiate GPUConscan object
        GpuConscan gpu(hInstance,nCmdShow, SCENE_SIZE_X, SCENE_SIZE_Y,
                  RETICLE_SIZE);

        //upload  reticles
        for (int i = 0; i<100; i++){
                  for (int j = 0; j<RET_SIZE_SQ; j++){
                            reticle[j] = (float)i;
                  }
                  gpu.uploadReticle(i,reticle);
        }
```

```
//initialize retIndex and x/y disp arrays
for (int i = 0; i<40 ; i++){
          retIndex[i] = 39-i;
          xdisp[i] = 1;
          ydisp[i] = 1;
}

char algorithm[40];
int alg =gpu.GetAlg();
if (alg ==1){
          strcpy(algorithm,"BIGTEX");
}
else if (alg == 2){
          strcpy(algorithm,"ONEBYONE");
}
else if (alg == 3){
          strcpy(algorithm,"CONSCAN ONEBYONE R32F");
}
else {
          strcpy(algorithm,"NA");
}
// do experiment REPS times
for (int rep = 0; rep< REPS; rep++){
          //fill initial scene
          if  ( WL != 3){
                    for (int j = 0; j<SIZE_SQ; j++){
                              scene[j]= (float)((j%SCENE_SIZE_X)/(SCENE_SIZE_X/2)+1);
                              if (j>=SIZE_SQ/2) scene[j]+=2;
                    }
          }
          else {
                    for (int j = 0; j<SIZE_SQ; j++){
                              scene[j]= 0.0f;
                    }
          }

          startTime = (double)timeGetTime();
          // run algorithm 1000 x
          for (int i = 0; i<1000; i++){
                    gpu.Process(retIndex,scene,answer,xdisp, ydisp);
                    UpdateScene(WL,scene);
          }
          endTime = (double) timeGetTime();
          double timeDelta = (endTime-startTime)*0.001f;
          double timeDeltaLog = log10(timeDelta);
          times[rep]= timeDelta;
          timeslog[rep] = timeDeltaLog;
          sum += timeDelta;
          sumlog += timeDeltaLog;
}
//calc stats
mean = sum/(double)REPS;
meanlog = sumlog/(double)REPS;
hi = 0.0;
hilog = -1000.0;
low = 1000.0;
lowlog = 1000.0;

for (int i =0; i<REPS; i++){
          if (times[i]>hi)
                    hi = times[i];
          if (times[i]<low)
                    low = times[i];
          var += pow( (times[i]-mean),2.0)/(double)(REPS-1);
          sos += pow( times[i],2);
          if (timeslog[i]>hilog)
                    hilog = timeslog[i];
          if (timeslog[i]<lowlog)
                    lowlog = timeslog[i];
          varlog += pow( (timeslog[i]-meanlog),2.0)/(double)(REPS-1);
          soslog += pow( timeslog[i],2);
}
stdev = sqrt( var);
stdevlog = sqrt (varlog);
//write results to file
char* name ="results/results_";
char* ext  =".dat";
char num[4];
_itoa(EXPER,num,10);
char filename[40];
strcpy(filename,name);
strcat(filename,num);
strcat(filename,ext);
std::ofstream outFile(filename,std::ios::app);//out
if (!outFile){
          ::MessageBox(0, "can't open results file","GPU" , 0);
          exit(1);
}
outFile   <<"experiment#: "<<EXPER<<'\n'
                    <<"workload:    "<<WL_str<<'\n'
                    <<"bus:         "<<BUS_str<<'\n'
                    <<"approach:    "<<APRCH_str<<'\n'
                    <<"algorithm:   "<<algorithm<<'\n'
                    <<"size:        "<<SCENE_SIZE<<'\n'
                    <<"mean:        "<<mean<<'\n'
                    <<"variance:    "<<var<<'\n'
                    <<"stdev:       "<<stdev<<'\n'
                    <<"sum of sqrs: "<<sos<<'\n'
                    <<"low:         "<<low<<'\n'
                    <<"hi:          "<<hi<<'\n'
                    <<"mean log:        "<<meanlog<<'\n'
                    <<"variance log:    "<<varlog<<'\n'
                    <<"stdev log :      "<<stdevlog<<'\n'
                    <<"sum of sqrs log: "<<soslog<<'\n'
                    <<"low log:         "<<lowlog<<'\n'
                    <<"hi log:          "<<hilog<<'\n'
                    <<"reps:        "<<REPS<<'\n'
                    <<"data:        "<<'\n';
```

```
	for (i =0;i<REPS; i++){
		outFile<<times[i];
		if (!((i+1)%5) || (i==REPS-1))
			outFile<<'\t'<<" ..."<<'\n';
		else outFile<<'\t';
	}
	outFile<<'\n'<<"data log:	"<<'\n';
	for (i =0;i<REPS; i++){
		outFile<<std::setprecision(6)<<std::setw(3)<<timeslog[i];
		if (!((i+1)%5) || (i==REPS-1))
			outFile<<'\t'<<" ..."<<'\n';
		else outFile<<'\t';
	}
	outFile<<'\n'<<"answers:"<<'\n';
	for (i =0;i<40; i++){
		outFile<<std::setprecision(9)<<std::setw(18)<<
			std::setiosflags(std::ios::scientific)<<answer[i];
		if (!(( i+1)%4))
			outFile<<'\n';
	}
	outFile<<'\n';


	MSG msg;
	HACCEL hAccelTable;

	// Initialize global strings
	LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
	LoadString(hInstance, IDC_WIN_CONSCAN, szWindowClass, MAX_LOADSTRING);
	MyRegisterClass(hInstance);

	// Perform application initialization:
	if (!InitInstance (hInstance, nCmdShow))
	{
		return FALSE;
	}

	hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_WIN_CONSCAN);

	// Main message loop:
	while (GetMessage(&msg, NULL, 0, 0))
	{
		if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
		{
			TranslateMessage(&msg);
			DispatchMessage(&msg);
		}
	}

	return (int) msg.wParam;
}

//------------------------------------------------------------
//  FUNCTION:  UpdateScene()
//-----------------------------------------------------------
void UpdateScene(int p_WL, float* p_scene){
	if (p_WL == 1){
		return;
	}
	if (p_WL == 2){
		for (int j = 0; j < SIZE_SQ; j++){
			p_scene[j] += 1.0f;
		}
		return;
	}
	else {
		int x = delx + oldx;
		int y = dely + oldy;
		if ( (x<xmin) || (x>xmax) ){
			x = oldx;
			xinc = - xinc;
			delx = delx+xinc;
			dely = dely+yinc;
			y += dely;
		}
		if ( (y<ymin) || (y>ymax) ) {
			y = oldy;
			yinc = -yinc;
			delx += xinc;
			dely += yinc;
			x += delx;
		}

		int index = y*SCENE_SIZE_X + x;
		int indexold = oldy*SCENE_SIZE_X + oldx;
		p_scene[indexold] = 0.0f;
		p_scene[index] = 1.0f;
		oldx = x;
		oldy =y;
		return;
	}
}

//
//  FUNCTION: MyRegisterClass()
//
//  PURPOSE: Registers the window class.
//
//  COMMENTS:
//    This function and its usage are only necessary if you want this code
//    to be compatible with Win32 systems prior to the 'RegisterClassEx'
//    function that was added to Windows 95. It is important to call this function
//    so that the application will get 'well formed' small icons associated
//    with it.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
	WNDCLASSEX wcex;
```

```
        wcex.cbSize         = sizeof(WNDCLASSEX);
        wcex.style          = CS_HREDRAW | CS_VREDRAW;
        wcex.lpfnWndProc    = (WNDPROC)WndProc;
        wcex.cbClsExtra     = 0;
        wcex.cbWndExtra     = 0;
        wcex.hInstance      = hInstance;
        wcex.hIcon          = LoadIcon(hInstance, (LPCTSTR)IDI_WIN_CONSCAN);
        wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
        wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
        wcex.lpszMenuName   = (LPCTSTR)IDC_WIN_CONSCAN;
        wcex.lpszClassName  = szWindowClass;
        wcex.hIconSm        = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);

        return RegisterClassEx(&wcex);
}

//
//   FUNCTION: InitInstance(HANDLE, int)
//
//   PURPOSE: Saves instance handle and creates main window
//
//   COMMENTS:
//
//        In this function, we save the instance handle in a global variable and
//        create and display the main program window.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;

   hInst = hInstance; // Store instance handle in our global variable

   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//  FUNCTION: WndProc(HWND, unsigned, WORD, LONG)
//
//  PURPOSE:  Processes messages for the main window.
//
//  WM_COMMAND   - process the application menu
//  WM_PAINT     - Paint the main window
//  WM_DESTROY   - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
        int wmId, wmEvent;
        PAINTSTRUCT ps;
        HDC hdc;
        switch (message)
        {
        case WM_COMMAND:
                wmId    = LOWORD(wParam);
                wmEvent = HIWORD(wParam);
                // Parse the menu selections:
                switch (wmId)
                {
                case IDM_ABOUT:
                        DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX, hWnd, (DLGPROC)About);
                        break;
                case IDM_EXIT:
                        DestroyWindow(hWnd);
                        break;
                default:
                        return DefWindowProc(hWnd, message, wParam, lParam);
                }
                break;
        case WM_PAINT:
                hdc = BeginPaint(hWnd, &ps);
                // TODO: Add any drawing code here...
                EndPaint(hWnd, &ps);
                break;
        case WM_DESTROY:
                PostQuitMessage(0);
                break;
        default:
                return DefWindowProc(hWnd, message, wParam, lParam);
        }
        return 0;
}
// Message handler for about box.
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
        switch (message)
        {
        case WM_INITDIALOG:
                return TRUE;
        case WM_COMMAND:
                if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
                {
                        EndDialog(hDlg, LOWORD(wParam));
                        return TRUE;
                }
                break;
        }
        return FALSE;
}
```

```
// file: GPU_CONSCAN.h
//
// by:   Maj Sean Jeffers
// requires external files:
//      GPU_UTILITY.h                 -- contains namespace d3d utility functions InitD3D()
//                                              GPU_WndProc CALLBACK and Gpu_WndClass definition
//  source/ps_CONSCAN.txt             -- PS used by MAddReduce()
//  source/vs_CONSCAN.txt             -- VS used by  MAddReduce()
//  source/vs_16tapredux_2.txt        -- vertex shader used by Redux()
//  source/ps_16tapredux_2.txt        -- pixel shader used by  Redux()
//
//  27 dec 04 -- modified old ONEBYONE to use non-packed R32F textures throughout
//            -- this is expected to be basis for CONSCAN
//  2 jan 05  -- renamed R32F to  GPU_CONSCAN.h; changed to use CONSCAN vs and ps
//            -- border color not supported; clamping is
#ifndef GPU_CLASS_H_BY_MAJ_JEFFERS
#define GPU_CLASS_H_BY_MAJ_JEFFERS

#include <d3dx9.h>
#include "GPU_UTILITY.h"

#include <stdlib.h>
#include <cstring>
#include <cmath>

//------------------ CONSTANTS----------------
#define GPU_WINDOW_WIDTH    1024
#define GPU_WINDOW_HEIGHT   768
#define D3D_FORMAT          D3DFMT_R32F //A32B32G32R32F
#define STRIDE              4           //  16
//-------------------------------------------

class GpuConscan {

private:
       HINSTANCE                     hInst;
       int                           nCmdShow;
       IDirect3DDevice9*             Device;
       //const int                   SceneSize;
       const int                     SceneSizeX;
       const int                     SceneSizeY;
       const int                     ReticleSize;
       //int                         ScenePixels;
       //float                       fPixSizeX;
       //float                       fPixSizeY;
       //D3DXVECTOR4                 DataArray[2048*2048];
       int                           OutTexSize;
       long                          OutPixels;
       //int                         ViewportSize;
       int                           ReduceIterations;
       int                           TexIndex;
       bool                          DualRT;

       //VS1
       IDirect3DVertexShader9* VS1_maddreduce;
       ID3DXConstantTable*           VS1_VSCT;
       D3DXHANDLE                    VS1_PixelSizeHandle;
       D3DXHANDLE                    VS1_DisplacementHandle;
       D3DXHANDLE                    VS1_AspectHandle;

       //PS1
       IDirect3DPixelShader9*        PS1_maddreduce;
       ID3DXConstantTable*           PS1_PSCT;
       //VS2
       IDirect3DVertexShader9*       VS2_16tapreduce;
       ID3DXConstantTable*           VS2_VSCT;
       D3DXHANDLE                    VS2_offsetHandle;

       //PS2
       IDirect3DPixelShader9*        PS2_16tapreduce;
       ID3DXConstantTable*           PS2_PSCT;
       D3DXHANDLE                    PS2_offsetHandle;
       D3DXHANDLE                    PS2_mulHandle;

       // PARAMETERS PASSED TO PS & VS
       D3DXVECTOR2                   offset[4][16];

       // VERTEX BUFFER & DECL
       LPDIRECT3DVERTEXDECLARATION9  m_pDecl;
       IDirect3DVertexBuffer9*       QuadVB;

       // TEXTURES & SURFACES
       IDirect3DTexture9*            Scene_Tex;
       IDirect3DSurface9*            Scene_Surface;

       IDirect3DTexture9*            Reticle_Tex[100];
       IDirect3DSurface9*            Reticle_Surface[100];

       IDirect3DTexture9*            RT_Tex;
       IDirect3DSurface9*            RT_Surface;


       IDirect3DTexture9*            RT_Reduce_Tex[4];
       IDirect3DSurface9*            RT_Reduce_Surface[4];
//     IDirect3DTexture9*            RT_Reduce_Tex2[3];
//     IDirect3DSurface9*            RT_Reduce_Surface2[3];

       IDirect3DTexture9*            Ones_Tex;
       IDirect3DSurface9*            Ones_Surface;

       // transformation matrices
       D3DXMATRIX                    mWorld;
       D3DXMATRIX                    mView;
       D3DXMATRIX                    mProj;

       // --------------------- STRUCTS  ----------------------

       struct CUSTOMVERTEX
```

```
        {
                FLOAT       x;
                FLOAT       y;
        };

public:
        //constructor
        GpuConscan(HINSTANCE p_hInst, int p_nCmdShow, int p_SceneSizeX,
                        int p_SceneSizeY, int p_ReticleSize)
                :hInst (p_hInst), nCmdShow(p_nCmdShow), SceneSizeX(p_SceneSizeX),
                SceneSizeY(p_SceneSizeY), ReticleSize(p_ReticleSize)// /2
        {
                Device =0;

                //fPixSizeX         = -1.0f / (float)SceneSize;
                //fPixSizeY         = 1.0f / (float) SceneSize;

                //VS1
                VS1_maddreduce      = 0;
                VS1_VSCT            = 0;
                VS1_PixelSizeHandle = 0;
                //PS1
                PS1_maddreduce      = 0;
                PS1_PSCT            = 0;
                //VS2
                VS2_16tapreduce     = 0;
                VS2_VSCT            = 0;
                VS2_offsetHandle    = 0;

                //PS2
                PS2_16tapreduce     = 0;
                PS2_PSCT            = 0;
                PS2_offsetHandle    = 0;
                // vertex buffer ptr
                QuadVB              = 0;

                if(!d3d::InitD3D(hInst, nCmdShow,
                        GPU_WINDOW_WIDTH, GPU_WINDOW_HEIGHT, true, D3DDEVTYPE_HAL, &Device))
                {
                        ::MessageBox(0, "InitD3D() - FAILED", 0, 0);
                }

                if(!Setup()){
                        ::MessageBox(0, "Setup() - FAILED", 0, 0);
                }

                DualRT = false;

                // modular actions depending on algorithm
                InitShaders();
                InitReticlesAndScene_OneByOne();
                InitRenderTargets_hybrid();

        }//Gpu() CONSTRUCTOR

private:
        //---------------------------------------------------------
        //                  InitShaders()
        //                  creates & compiles shaders
        //---------------------------------------------------------
        bool InitShaders(){
                HRESULT hr = 0;
                // *** PS1

                ID3DXBuffer* PSBuffer          = 0;
                ID3DXBuffer* errorBuffer = 0;

                hr = D3DXCompileShaderFromFile(
                        "source/ps_CONSCAN.txt",
                        0,
                        0,
                        "PSMain", // entry point function name
                        "ps_2_0",
                        D3DXSHADER_SKIPVALIDATION,//DEBUG, // | D3DXSHADER_SKIPOPTIMIZATION
                        &PSBuffer,
                        &errorBuffer,
                        &PS1_PSCT);

                // output any error messages
                if( errorBuffer )
                {
                        ::MessageBox(0, (char*)errorBuffer->GetBufferPointer(), 0, 0);
                        Release<ID3DXBuffer*>(errorBuffer);
                }
                if(FAILED(hr))
                {
                        ::MessageBox(0, "PS1--D3DXCompileShaderFromFile() - FAILED", 0, 0);
                        return false;
                }

                // create pixel shader
                hr = Device->CreatePixelShader(
                        (DWORD*)PSBuffer->GetBufferPointer(),
                        &PS1_maddreduce);

                if(FAILED(hr))
                {
                        ::MessageBox(0, "CreatePixelShader PS1 - FAILED", 0, 0);
                        return false;
                }

                Release<ID3DXBuffer*>(PSBuffer);

                // *** PS2

                ID3DXBuffer* PS2Buffer        = 0;
                ID3DXBuffer* errorBuffer2     = 0;
```

```
hr = D3DXCompileShaderFromFile(
          "source/ps_16tapredux_2.txt",
          0,
          0,
          "PSMain", // entry point function name
          "ps_2_0",
          D3DXSHADER_SKIPVALIDATION,//DEBUG, // | D3DXSHADER_SKIPVALIDATION OPTIMIZATION
          &PS2Buffer,
          &errorBuffer2,
          &PS2_PSCT);

// output any error messages
if( errorBuffer2 )
{
          ::MessageBox(0, (char*)errorBuffer2->GetBufferPointer(), 0, 0);
          Release<ID3DXBuffer*>(errorBuffer2);
}
if(FAILED(hr))
{
          ::MessageBox(0, "PS2--D3DXCompileShaderFromFile() - FAILED", 0, 0);
          return false;
}

// create pixel shader
hr = Device->CreatePixelShader(
          (DWORD*)PS2Buffer->GetBufferPointer(),
          &PS2_16tapreduce);

if(FAILED(hr))
{
          ::MessageBox(0, "CreatePixelShader PS2 - FAILED", 0, 0);
          return false;
}

Release<ID3DXBuffer*>(PS2Buffer);

// *** VS1

ID3DXBuffer* VSBuffer      = 0;
ID3DXBuffer* errorBuffer3 = 0;

hr = D3DXCompileShaderFromFile(
          "source/vs_CONSCAN.txt",
          0,
          0,
          "Main", // entry point function name
          "vs_2_0",
          D3DXSHADER_SKIPVALIDATION,//DEBUG, // | D3DXSHADER_SKIPOPTIMIZATION
          &VSBuffer,
          &errorBuffer3,
          &VS1_VSCT);

// output any error messages
if( errorBuffer3 )
{
          ::MessageBox(0, (char*)errorBuffer3->GetBufferPointer(), 0, 0);
          Release<ID3DXBuffer*>(errorBuffer3);
}
if(FAILED(hr))
{
          ::MessageBox(0, "VS1--D3DXCompileShaderFromFile() - FAILED", 0, 0);
          return false;
}

// create vertex shader
hr = Device->CreateVertexShader(
          (DWORD*)VSBuffer->GetBufferPointer(),
          &VS1_maddreduce);

if(FAILED(hr))
{
          ::MessageBox(0, "CreateVertexShader VS1 - FAILED", 0, 0);
          return false;
}

Release<ID3DXBuffer*>(VSBuffer);

// *** VS2
ID3DXBuffer* VS2Buffer      = 0;
ID3DXBuffer* errorBuffer4 = 0;

hr = D3DXCompileShaderFromFile(
          "source/vs_16tapredux_2.txt",
          0,
          0,
          "Main", // entry point function name
          "vs_2_0",
          D3DXSHADER_SKIPVALIDATION,//DEBUG, // | D3DXSHADER_SKIPOPTIMIZATION
          &VS2Buffer,
          &errorBuffer4,
          &VS2_VSCT);

// output any error messages
if( errorBuffer4 )
{
          ::MessageBox(0, (char*)errorBuffer4->GetBufferPointer(), 0, 0);
          Release<ID3DXBuffer*>(errorBuffer4);
}
if(FAILED(hr))
{
          ::MessageBox(0, "VS2--D3DXCompileShaderFromFile() - FAILED", 0, 0);
          return false;
}

// create vertex shader
hr = Device->CreateVertexShader(
          (DWORD*)VS2Buffer->GetBufferPointer(),
          &VS2_16tapreduce);
          if(FAILED(hr))
          {
                    ::MessageBox(0, "CreateVertexShader VS2 - FAILED", 0, 0);
                    return false;
          }

          Release<ID3DXBuffer*>(VS2Buffer);

          //---------------- get VS1 pixelsize constant handle
          VS1_PixelSizeHandle = VS1_VSCT->GetConstantByName(0, "PixelSize");
          VS1_DisplacementHandle = VS1_VSCT->GetConstantByName(0,"Displacement");
          VS1_AspectHandle = VS1_VSCT->GetConstantByName(0,"Aspect");

          // get PS2 and VS2 const handles
          VS2_offsetHandle = VS2_VSCT->GetConstantByName(0, "offset");
          PS2_offsetHandle = PS2_PSCT->GetConstantByName(0, "offset");

          //vertex decl and set stream source were here, moved to Setup
          return true;

}// InitShaders()


//----------------------------------------------------------
//        InitReticlesAndScene_OneByOne()
// loads reticle images into GPU, creates reticle and scene surfaces
//  and textures in GPU memory
//---------------------------------------------------------
// FOR TESTING PURPOSES ONLY
// load the 100 reticle textures, [0..99] with sample data:
//  places whole value equal to texture index into
//   each pixel of the texture;

bool InitReticlesAndScene_OneByOne(){
          //--------------------------------------------
          // create scene texture and surface
          //--------------------------------------------
          HRESULT hr = 0;
          hr = D3DXCreateTexture(
                    Device,
                    SceneSizeX, SceneSizeY,  //was SceneSize for both
                    1, // no mipmap chain
                    D3DUSAGE_DYNAMIC, //was 0--keep DYNAMIC!
                    D3DFMT_R32F,
                    D3DPOOL_DEFAULT,
                    &Scene_Tex);
          if(FAILED(hr))
                    return false;

          //get interface to top level surface of Scene_Tex
          hr = Scene_Tex->GetSurfaceLevel(0,&Scene_Surface);
          if(FAILED(hr))
                    return false;

          //generate 100 reticle textures (half the scene width for CONSCAN)
          for (int t = 0; t<100; t++){
                    hr = D3DXCreateTexture(
                              Device,
                              ReticleSize, ReticleSize, //was SceneSize/2
                              1, // no mipmap chain
                              0,//usage
                              D3DFMT_R32F,
                              D3DPOOL_DEFAULT,
                              &Reticle_Tex[t]);
                    if(FAILED(hr))
                              return false;

                    //get interface to top level surface of each tex
                    hr = Reticle_Tex[t]->GetSurfaceLevel(0,&Reticle_Surface[t]);
                    if(FAILED(hr))
                              return false;
          }
          return true;
}

//------------------------------------------------------
//                InitRenderTargets_hybrid()
//------------------------------------------------------
bool InitRenderTargets_hybrid() {

          HRESULT hr = 0;
          int Init_RTSize;
          // set initial RT size
          // scene can be  1024, 512 or 256
          // reticle can be 512, 256 or 128
          if (ReticleSize == 128){   //was SceneSize == 256
                    Init_RTSize = 512;   //scenesize/4 * 8
                    ReduceIterations = 3;
          }
          else if (ReticleSize == 256){
                    Init_RTSize = 1024;
                    ReduceIterations = 3;
          }

          else {
                    Init_RTSize = 2048;
                    ReduceIterations = 4;
          }
          //SET OutTexSize
          //  the size of the final RT we will get our
          //   result from
          //  affects GetRTData()
          OutTexSize = 8*ReticleSize/(2*((int)pow(4,ReduceIterations)));//was 4*SceneSize
          //changed for CONSCAN

          OutPixels = OutTexSize*OutTexSize;
          //SET TexIndex
```

```
        // the array index of the RT_Reduce_Surface[] that will contain
        //  the final result; affects GetRTData()
        TexIndex = ReduceIterations-1;

        // create initial RT (half the reticle pallette size)
        hr = Device->CreateTexture(Init_RTSize,Init_RTSize,1,D3DUSAGE_RENDERTARGET,
                                D3DFMT_R32F ,D3DPOOL_DEFAULT,&RT_Tex,0); //D3D_FORMAT
        if(FAILED(hr))
                        return false;
        //get interface to top level surface
        hr = RT_Tex->GetSurfaceLevel(0, &RT_Surface);
        if(FAILED(hr))
                return false;

        // create render targets for reduction op (2 or 3)
        int Size = Init_RTSize/4;
        for (int i=0; i<ReduceIterations; i++){
                hr = D3DXCreateTexture(
                        Device,
                        Size, Size,
                        1, // no mipmap chain
                        D3DUSAGE_RENDERTARGET, //could be DYNAMIC, but not DYNAMIC and RT
                        D3DFMT_R32F,// D3D_FORMAT,
                        D3DPOOL_DEFAULT,
                        &RT_Reduce_Tex[i]);
                if(FAILED(hr))
                        return false;

                //get interface to top level surface
                hr = RT_Reduce_Tex[i]->GetSurfaceLevel(0, &RT_Reduce_Surface[i]);
                if(FAILED(hr))
                        return false;

                Size /= 4;
        }

        //create SYSTEMMEM tex to send result to
        hr = D3DXCreateTexture(
                        Device,
                        OutTexSize, OutTexSize,
                        1, // no mipmap chain
                        D3DUSAGE_DYNAMIC, // usage could be DYNAMIC, but not DYNAMIC and RT
                        D3DFMT_R32F,// D3D_FORMAT,
                        D3DPOOL_SYSTEMMEM,
                        &Ones_Tex);
        if(FAILED(hr))
                return false;
        //get interface to top level surface of Ones_Tex[]
        hr = Ones_Tex->GetSurfaceLevel(0, &Ones_Surface);
        if(FAILED(hr))
                return false;

        //-------------------------------------------------------------------
        // *** OFFSET ARRAYS FOR 16:1 REDUCE
        //-------------------------------------------------------------------
        // PixelSize of input texture to first reduce op
        float PixelSize2 = 1/(float)Init_RTSize;

        // calculate displacements
        for (int k = 0; k<ReduceIterations; k++){
                for (int i = 0; i<4; i++){
                        for(int j = 0; j<4; j++){
                                offset[k][i*4 +j] = D3DXVECTOR2(PixelSize2*(float)j, PixelSize2*(float)i);
                        }
                }
                PixelSize2 *= 4.0f;
        }

        return true;
}// InitRenderTargets_hybrid()

//-----------------------------------------------------------
//                Setup()
// Initializes geometry, renderstate, calls
//   InitRenderTargets, InitShaders, InitReticlesAndScene
//-----------------------------------------------------------
bool Setup() {

        HRESULT hr = 0;

        //---------------- DISABLE unneeded processing ------------------
        // turn off Stencil and Culling
        hr = Device->SetDepthStencilSurface(
                0);
        if(FAILED(hr))
                return false;
        hr = Device->SetRenderState(D3DRS_CULLMODE,D3DCULL_NONE);
        if(FAILED(hr))
                return false;
        // disable lighting

        Device->SetRenderState(D3DRS_LIGHTING, false);

        //------------------ create geometry ----------------------------

        D3DVERTEXELEMENT9 decl[]=
        {
        {0, 0, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
        D3DDECL_END()
        };
        // declare the vertex structure
        hr = Device->CreateVertexDeclaration(decl, &m_pDecl);
        if(FAILED(hr))
                return false;
        // create VB with only x,y position
        hr = Device->CreateVertexBuffer( 56 * sizeof(CUSTOMVERTEX), //was 4
                                D3DUSAGE_WRITEONLY,
                                0,
                              D3DPOOL_DEFAULT,
                              &QuadVB,
                              NULL );
if(FAILED(hr))
          return false;

float left          = -1.00f;
float right         =  1.00f;
float top           =  1.00f;
float bottom        = -1.00f;
float top_row2      =  0.75f;
float top_row3      =  0.50f;
float top_row4      =  0.25f;
float top_row5      =  0.00f;
float top_row7      = -0.50f;
float top_row8      = -0.75f;
float bot_row1      =  0.75f;
float bot_row2      =  0.50f;
float bot_row4      =  0.00f;
float bot_row5      = -0.25f;
float bot_row6      = -0.50f;
float bot_row7      = -0.75f;

CUSTOMVERTEX* v;
QuadVB->Lock(0, 56 * sizeof(CUSTOMVERTEX), (VOID**)&v, 0);//was 4

//quad 0 full square
// left bottom
v[0].x = left;
v[0].y = bottom;//bottom

// left top
v[1].x = left;
v[1].y = top;

// right bottom
v[2].x = right;
v[2].y = bottom;//bottom

// right top
v[3].x = right;
v[3].y = top;

//quad 1 r1-5
// left bottom
v[4].x = left;
v[4].y = bot_row5;//5

// left top
v[5].x = left;
v[5].y = top;

// right bottom
v[6].x = right;
v[6].y = bot_row5;//5

// right top
v[7].x = right;
v[7].y = top;

//quad 2 r2-6
// left bottom
v[8].x = left;
v[8].y = bot_row6;

// left top
v[9].x = left;
v[9].y = top_row2;

// right bottom
v[10].x = right;
v[10].y = bot_row6;

// right top
v[11].x = right;
v[11].y = top_row2;

//quad 3 r3-7
// left bottom
v[12].x = left;
v[12].y = bot_row7;

// left top
v[13].x = left;
v[13].y = top_row3;

// right bottom
v[14].x = right;
v[14].y = bot_row7;

// right top
v[15].x = right;
v[15].y = top_row3;

//quad 4 r4-8
// left bottom
v[16].x = left;
v[16].y = bottom;

// left top
v[17].x = left;
v[17].y = top_row4;

// right bottom
v[18].x = right;
v[18].y = bottom;

// right top
```

```
v[19].x = right;
v[19].y = top_row4;

//quad 5 r1-6
// left bottom
v[20].x = left;
v[20].y = bot_row6;

// left top
v[21].x = left;
v[21].y = top;

// right bottom
v[22].x = right;
v[22].y = bot_row6;

// right top
v[23].x = right;
v[23].y = top;

//quad 6 r2-7
// left bottom
v[24].x = left;
v[24].y = bot_row7;

// left top
v[25].x = left;
v[25].y = top_row2;

// right bottom
v[26].x = right;
v[26].y = bot_row7;

// right top
v[27].x = right;
v[27].y = top_row2;

//quad 7 r3-8
// left bottom
v[28].x = left;
v[28].y = bottom;

// left top
v[29].x = left;
v[29].y = top_row3;

// right bottom
v[30].x = right;
v[30].y = bottom;

// right top
v[31].x = right;
v[31].y = top_row3;

//quad 8 T1
// left bottom
v[32].x = left;
v[32].y = bot_row1;

// left top
v[33].x = left;
v[33].y = top;

// right bottom
v[34].x = right;
v[34].y = bot_row1;

// right top
v[35].x = right;
v[35].y = top;

//quad 9 B2
// left bottom
v[36].x = left;
v[36].y = bottom;

// left top
v[37].x = left;
v[37].y = top_row7;

// right bottom
v[38].x = right;
v[38].y = bottom;

// right top
v[39].x = right;
v[39].y = top_row7;

//quad 10 T4
// left bottom
v[40].x = left;
v[40].y = bot_row4;

// left top
v[41].x = left;
v[41].y = top;

// right bottom
v[42].x = right;
v[42].y = bot_row4;

// right top
v[43].x = right;
v[43].y = top;

//quad 11 B1
// left bottom
v[44].x = left;
```

```
                v[44].y = bottom;

                // left top
                v[45].x = left;
                v[45].y = top_row8;

                // right bottom
                v[46].x = right;
                v[46].y = bottom;

                // right top
                v[47].x = right;
                v[47].y = top_row8;

                //quad 12 B4
                // left bottom
                v[48].x = left;
                v[48].y = bottom;

                // left top
                v[49].x = left;
                v[49].y = top_row5;

                // right bottom
                v[50].x = right;
                v[50].y = bottom;

                // right top
                v[51].x = right;
                v[51].y = top_row5;

                //quad 13 T2
                // left bottom
                v[52].x = left;
                v[52].y = bot_row2;

                // left top
                v[53].x = left;
                v[53].y = top;

                // right bottom
                v[54].x = right;
                v[54].y = bot_row2;

                // right top
                v[55].x = right;
                v[55].y = top;

                QuadVB->Unlock();

                // set vertex declaration
                Device->SetVertexDeclaration(m_pDecl);
                // set geometry
                Device->SetStreamSource(0, QuadVB, 0, sizeof(CUSTOMVERTEX));

                //Device->SetSamplerState(1,D3DSAMP_ADDRESSU,D3DTADDRESS_CLAMP);
                //Device->SetSamplerState(1,D3DSAMP_ADDRESSV,D3DTADDRESS_CLAMP);
                return true;
        }//Setup()


private:
        //-------------------------------------------------------------
        // ---------------------- LOAD INPUT SCENE -------------------
        //-------------------------------------------------------------
        bool LoadInputScene(float p_inputArray[]) {
                RECT SurfRect;
                SurfRect.left       = 0;
                SurfRect.top        = 0;
                SurfRect.right      = SceneSizeX;
                SurfRect.bottom     = SceneSizeY;

                HRESULT hr = 0;
                hr= D3DXLoadSurfaceFromMemory(
                        Scene_Surface,
                        0,
                        0,
                        p_inputArray,
                        D3DFMT_R32F,
                        (4*SceneSizeX),  //16 for 4x32-bit, 8 for 2x32-bit, 4 for 1x32F format
                        0,
                        &SurfRect,
                        D3DX_FILTER_NONE,
                        0);
                if(FAILED(hr))
                        return false;
                return true;
        }//LoadInputScene()

        //------------------------------------------------------------
        //                                      bool MAddReduce()
        //
        bool MAddReduce(int* p_retIndex, int* p_xdisp, int* p_ydisp){
                HRESULT hr = 0;

                // ---- set PS and VS shaders
                Device->SetVertexShader(VS1_maddreduce);
                Device->SetPixelShader(PS1_maddreduce);

                // set VS PixelSize const
                // .x = 1/reticle img width, .y = 1/scenewidthX, .z = 1/scenewidthY, .w = 0.0f
                hr =      VS1_VSCT->SetVector(Device, VS1_PixelSizeHandle,
                                        &D3DXVECTOR4(1.0f/(float)ReticleSize, 1.0f/(float)SceneSizeX,
                                        1.0f/(float)SceneSizeY, 0.0f));
                if(FAILED(hr))
                        return false;
                //was 2/SceneSize and 1/SceneSize
        // set Aspect in VS; allows for arbitrary scene dimensions
        D3DXVECTOR2 aspect;
        aspect.x = (float)ReticleSize/(float)SceneSizeX; //0.5f;
        aspect.y = (float)ReticleSize/(float)SceneSizeY; //0.5f;

        hr =  VS1_VSCT->SetFloatArray(Device,
                              VS1_AspectHandle,
                              (float*)aspect,2);
        if(FAILED(hr))
                return false;

        // ---- set RT
        hr = Device->SetRenderTarget(
                                      0,
                                      RT_Surface);
        if(FAILED(hr))
                return false;

        //Device->Clear(0,0,D3DCLEAR_TARGET,0L,0,0);

        Device->SetStreamSource(0, QuadVB, 0, sizeof(CUSTOMVERTEX));

        D3DVIEWPORT9 vp;

        vp.Width  = ReticleSize/2;
        vp.Height = ReticleSize/2;
        vp.MinZ   = 0.0f;
        vp.MaxZ   = 1.0f;

        Device->SetTexture( 1, Scene_Tex);//stage 1 = input scene

        D3DXVECTOR2 displacement;
        float f_dispx;
        float f_dispy;
        float halfRetX;
        float halfRetY;
        int   index = 0;

        for (int v = 0; v<5; v++){
                for (int h = 0; h<8; h++){
                        vp.X = h*ReticleSize/2;
                        vp.Y = v*ReticleSize/2;
                        Device->SetViewport(&vp);
                        // set sampler 0 with reticle image for maddredux with scene
                        index = v*8+h;
                        Device->SetTexture( 0, Reticle_Tex[ p_retIndex[index]] );

                        //new displacement vector added for conscan x/y offset in VS
                        f_dispx = (float)p_xdisp[index]/(float)SceneSizeX;
                        f_dispy = (float)p_ydisp[index]/(float)SceneSizeY;
                        halfRetX = (float)(ReticleSize/2)/(float)SceneSizeX;
                        halfRetY = (float)(ReticleSize/2)/(float)SceneSizeY;
                        displacement.x = 0.5f + f_dispx - halfRetX +1.0f/(2.0f*(float)SceneSizeX);;
                        displacement.y = 0.5f - f_dispy - halfRetY + 1.0f/(2.0f*(float)SceneSizeY);

                        hr =  VS1_VSCT->SetFloatArray(Device,
                                              VS1_DisplacementHandle,
                                              (float*)displacement,
                                              2);
                        if(FAILED(hr))
                                return false;

                        // render-- madd scene with a single reticle image
                        Device->BeginScene();
                                Device->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
                        Device->EndScene();
                }
        }
        // set streamsource to 8x5 rectangle r1-r5, Quad 1
        Device->SetStreamSource(0, QuadVB, 4*sizeof(CUSTOMVERTEX), sizeof(CUSTOMVERTEX));
        return true;
} //MAddReduce()

//------------------------------------------------------------------
//      Redux()
//      16:1 REDUCE OPERATION
//
//------------------------------------------------------------------
bool Redux() {
        HRESULT hr = 0;

        // ---- set PS and VS shaders
        Device->SetVertexShader(VS2_16tapreduce);
        Device->SetPixelShader (PS2_16tapreduce);

        // initial source tex is result of maddreduce op
        Device->SetTexture( 0, RT_Tex);

        for (int i = 0; i < ReduceIterations; i++) {

                hr = Device->SetRenderTarget( 0, RT_Reduce_Surface[i]);
                if(FAILED(hr))
                        return false;

                if (i>0)
                        Device->SetTexture( 0, RT_Reduce_Tex[i-1]);

                // set VS offset constant array
                hr =    VS2_VSCT->SetFloatArray(      Device,
                                              VS2_offsetHandle,
                                              (float*)&offset[i][0],
                                              16 );// 2*8 floats
                if(FAILED(hr))
                        return false;

                // set PS offset constant array
                hr =    PS2_PSCT->SetFloatArray(      Device,
                                              PS2_offsetHandle,
                                                            (float*) &offset[i][8],
                                                            16 );//2*8 floats
                    if(FAILED(hr))
                              return false;

                    // render-- do 16:1 reduction

                    //Device->Clear(0,0,D3DCLEAR_TARGET,0L,0,0);
                    Device->BeginScene();
                              Device->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
                    Device->EndScene();
          }
          return true;
} // Redux()

//--------------------------------------------------------------------
//        GetRTData_hybrid()
//       retrieve FINAL data from lockable render-to surface
//--------------------------------------------------------------------
void GetRTData_hybrid(double p_b[]) {
          HRESULT hr = 0;

          int q;
          int col;
          int row;

          D3DLOCKED_RECT lockedRect;

          Device->GetRenderTargetData(RT_Reduce_Surface[TexIndex], Ones_Surface);
          Ones_Surface->LockRect(&lockedRect,
                    0 , //lock entire tex
                    D3DLOCK_READONLY ); //flags

          float* imageData = (float*) lockedRect.pBits;

          //perform final 4:1 reduction if necessary and add 4 components of each pixel
          if (OutTexSize>8){
                    for (int i = 0; i<40; i++){

                              row = i/8;
                              col = i%8;
                              q = row*32 + col*2;
                              p_b[i]= imageData[q] + imageData[q+1] + imageData[q+16] + imageData[q+17];
                    }
          }
          else {
                    for (int i = 0; i<40; i++){
                              p_b[i]=imageData[i];
                    }
          }
          Ones_Surface->UnlockRect();

          return;
}// GetRTData_hybrid()

//--------------------------------------------------------
//        Release() and Delete()
//        cleanup functions
//--------------------------------------------------------
template<class T> void Release(T t)     {
          if( t )   {
                    t->Release();
                    t = 0;
          }
}

template<class T> void Delete(T t){
          if( t ){
                    delete t;
                    t = 0;
          }
}

//------------------------------------------------------------
//        Cleanup()
//        releases textures/surfaces/interfaces/devices/memory
//        allocated during program
//------------------------------------------------------------
void Cleanup()
{
          //vertex buffer and declaration
          Release<IDirect3DVertexBuffer9*>(QuadVB);
          Release<LPDIRECT3DVERTEXDECLARATION9>   (m_pDecl);

          //textures and surfaces
          Release<IDirect3DSurface9*>(Scene_Surface);
          Release<IDirect3DTexture9*>(Scene_Tex);

          int n = 100;
          for (int t = 0; t<n;t++){
                    Release<IDirect3DSurface9*>(Reticle_Surface[t]);
                    Release<IDirect3DTexture9*>(Reticle_Tex[t]);
          }


          Release<IDirect3DTexture9*>(RT_Tex);
          Release<IDirect3DSurface9*>(RT_Surface);


          for (int t = 0; t<ReduceIterations; t++) {
                    Release<IDirect3DSurface9*>(RT_Reduce_Surface[t]);
                    Release<IDirect3DTexture9*>(RT_Reduce_Tex[t]);
          }

          Release<IDirect3DTexture9*>(Ones_Tex);
          Release<IDirect3DSurface9*>(Ones_Surface);

          //PS & VS
```

```
                Release<IDirect3DPixelShader9*>(PS1_maddreduce);
                Release<ID3DXConstantTable*>(PS1_PSCT);
                Release<IDirect3DVertexShader9*>(VS1_maddreduce);
                Release<ID3DXConstantTable*>(VS1_VSCT);
                Release<IDirect3DPixelShader9*>(PS2_16tapreduce);
                Release<ID3DXConstantTable*>(PS2_PSCT);
                Release<IDirect3DVertexShader9*>(VS2_16tapreduce);
                Release<ID3DXConstantTable*>(VS2_VSCT);

                Device->Release();

        }// Cleanup()

public:
        //-----------------------------------------------------------------
        //              uploadReticle
        //-----------------------------------------------------------------
        bool uploadReticle(int p_index, float p_array[]){
                HRESULT hr = 0;

                RECT srcRect;
                srcRect.top = 0;
                srcRect.bottom = ReticleSize; // /2 for CONSCAN
                srcRect.left = 0;
                srcRect.right = ReticleSize; //change from SceneSize/2 to ReticleSize

                hr= D3DXLoadSurfaceFromMemory(
                        Reticle_Surface[p_index],
                        0,
                        0,
                        p_array,
                        D3DFMT_R32F,
                        (4*ReticleSize),  //16 for 4x32-bit, 8 for 2x32-bit, 4 for 1x32F format
                        0,                // SceneSize/2 for CONSCAN
                        &srcRect,
                        D3DX_FILTER_NONE,
                        0);
                if(FAILED(hr))
                        return false;

                return true;
        }

        //-----------------------------------------------------------
        //                                      Process()
        // user interface to GPU algorithm
        // input:  reference to scene image array variable-- scene[SceneSizeX *SceneSizeY]
        // input:  references to arrays in calling program:
        //            reticleIndex[40],, xdisp[40], ydisp[40], resultArray[40]
        // output: double result array[40]--output to JMASS
        void Process(int p_retIndex[], float p_SceneArray[], double p_resultArray[],
                        int p_xdisp[], int p_ydisp[]) {
                LoadInputScene(p_SceneArray);
                MAddReduce(p_retIndex, p_xdisp, p_ydisp);
                Redux();
                GetRTData_hybrid(p_resultArray);
                return;
        }// Process()

        int GetAlg(){
                return 3;//one by one, R32F, CONSCAN
        }
        //-----------------------------------------------------------
        //              ~ Gpu()  DESTRUCTOR
        ~GpuConscan() {
                Cleanup();
        }// ~ Gpu()  DESTRUCTOR

};
#endif  // GPU_CLASS_H_BY_MAJ_JEFFERS
```

```
//
// file: vs_CONSCAN.txt  (adapted from vs_experimental)
// BY MAJ SEAN JEFFERS
// 11 nov 04      -- multiplies 1x1 scene by 8x8 reticle pallette, then does
//                    4:1 redux; results in RT that is quarter sized of pallette;
//                    sampling of scene done w/wrapping
//                -- PixelSize.x = 1/ret pallette width
//                -- PixelSize.y = 1/scene width
//                -- PixelSize.z = 0.0f (must!)
//                -- VS generates 8 texcoords for PS
//  2 jan 05      -- scene is 4x RT width; ret is 2x RT width
//                -- implements CONSCAN
//                -- PixelSize.w = x-displacement texcoord wrt scene
//                     = x/scene_width , where x is pixel displacement {0..scene_width-1}
//                -- PixelSize.x = 1/ret width
//                -- PixelSize.y = 1/scene width
//  5 jan 05      -- modified to have separate displacement and aspect consts
// ------------------------------------------------------------

uniform float4  PixelSize;
uniform float2  Displacement;
uniform float2  Aspect;

// structures

struct VS_INPUT
{
       float4 Pos : POSITION;
};

struct VS_OUTPUT
{
       float4 Pos : POSITION;
       float2 Tex : TEXCOORD0;
       float2 Tex1: TEXCOORD1;
       float2 Tex2: TEXCOORD2;
       float2 Tex3: TEXCOORD3;
       float2 Tex4: TEXCOORD4;
       float2 Tex5: TEXCOORD5;
       float2 Tex6: TEXCOORD6;
       float2 Tex7: TEXCOORD7;
};


// ------------------------------------------------------------
// vertex shader function (input channels)
// ------------------------------------------------------------
VS_OUTPUT Main(VS_INPUT input)
{
       VS_OUTPUT output = (VS_OUTPUT)0;

       output.Pos.xy = input.Pos.xy;// + PixelSize.xy;
       output.Pos.z = 0.5f;
       output.Pos.w = 1.0f;

       //reticle tex coords (ret width = 2x RT width)
       output.Tex = float2(0.5f, -0.5f) * input.Pos.xy + 0.5f.xx ;
       output.Tex1 = output.Tex + PixelSize.xw;
       output.Tex2 = output.Tex + PixelSize.wx;
       output.Tex3 = output.Tex + PixelSize.xx;

       //scene tex coords (scene width = 4x RT width)
       output.Tex4 = Aspect*output.Tex + Displacement;//was 0.5f *adds in x-disp
       output.Tex5 = output.Tex4 + PixelSize.yw;
       output.Tex6 = output.Tex4 + PixelSize.wz;  // y now SceneSizeX
       output.Tex7 = output.Tex4 + PixelSize.yz;  //changed z to w, z now SceneSizeY
       return output;
}
```

```
// file: ps_CONSCAN.txt
//
// depends on:  file vs_onebyone.txt, GPU_CLASS_ONEBYONE_R32F.h
// By:  Maj Sean Jeffers
// descr:  modified version of ps_maddreduce_new for CONSCAN
//         PS for mult, add, reduce, 4:1; one-by-one approach using R32F textures only
//
// 27 dec 04      -- use with GPU_CLASS_ONEBYONE_R32F.h for CONSCAN
//                   does maddreduce op with input tex's R32F, output tex R32F
//                -- "dot" approach seems to work a little faster than other
//                    commented out approach; but both work
//                -- eliminated "noise" caused by dot product by assigning tex samples
//                    to individual float vector components vs. full float4
//  2 jan 05      -- modified to do conscan approach; accept 8 texcoords, all R32F tex's
// -------------------------------------------------------------
// globals
// -------------------------------------------------------------


sampler Rendersampler; //reticle img (2x RT width)
sampler Rendersampler2;//scene img (4x RT width)


// -------------------------------------------
// structures
// -------------------------------------------

struct PS_INPUT
{
       float2 Tex0 : TEXCOORD0;
       float2 Tex1 : TEXCOORD1;
       float2 Tex2 : TEXCOORD2;
       float2 Tex3 : TEXCOORD3;
       float2 Tex4 : TEXCOORD4;
       float2 Tex5 : TEXCOORD5;
       float2 Tex6 : TEXCOORD6;
       float2 Tex7 : TEXCOORD7;

};

// struct PS_OUTPUT
// {
//     float4 clr : COLOR; //was COLOR0
// };

// -------------------------------------------------------------
// Pixel Shader (input channels):output channel
// -------------------------------------------------------------

float4 PSMain(PS_INPUT input) :COLOR
{
       float4 t1;
       t1.r = tex2D(Rendersampler, input.Tex0);
       t1.g = tex2D(Rendersampler, input.Tex1);
       t1.b = tex2D(Rendersampler, input.Tex2);
       t1.a = tex2D(Rendersampler, input.Tex3);

       float4 t2;
       t2.r = tex2D(Rendersampler2, input.Tex4);
       t2.g = tex2D(Rendersampler2, input.Tex5);
       t2.b = tex2D(Rendersampler2, input.Tex6);
       t2.a = tex2D(Rendersampler2, input.Tex7);


       // madd ret (t1) & scene (t2)


       return dot(t1, t2);

}
```

```
#ifndef GPU_UTILITY_BY_MAJ_JEFFERS
#define GPU_UTILITY_BY_MAJ_JEFFERS

namespace d3d {


       LRESULT CALLBACK Gpu_WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
       {
                 return DefWindowProc(hWnd, message, wParam, lParam);
       }

       ATOM Gpu_WndClass(HINSTANCE hInstance ) {

                 WNDCLASSEX wcex;
                 wcex.cbSize         = sizeof(WNDCLASSEX);
                 wcex.style          = CS_HREDRAW | CS_VREDRAW;
                 wcex.lpfnWndProc    = (WNDPROC)d3d::Gpu_WndProc;//(WNDPROC)
                 wcex.cbClsExtra     = 0;
                 wcex.cbWndExtra     = 0;
                 wcex.hInstance      = hInstance;
                 wcex.hIcon          = LoadIcon(hInstance, (LPCTSTR)IDI_WIN_CONSCAN);
                 wcex.hCursor        = LoadCursor(NULL, IDC_ARROW);
                 wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
                 wcex.lpszMenuName   = 0;//no menu
                 wcex.lpszClassName  = "Gpu_WndClass";
                 wcex.hIconSm        = LoadIcon(wcex.hInstance, (LPCTSTR)IDI_SMALL);

                 return RegisterClassEx(&wcex);
       }

       bool InitD3D(       HINSTANCE hInstance, int nCmdShow,
                           int width, int height,
                           bool windowed,
                           D3DDEVTYPE deviceType,
                           IDirect3DDevice9** device)
       {
                 //  create GPU window
                 d3d::Gpu_WndClass(hInstance);
                 HWND hWnd2 = CreateWindow("Gpu_WndClass", "GPU", WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, NULL, NULL, hInstance, NULL);
                 if (!hWnd2){
                           return FALSE;
                 }
                 //ShowWindow(hWnd2, nCmdShow);
                 //UpdateWindow(hWnd2);

                 //
                 // Init D3D:
                 //

                 HRESULT hr = 0;

                 // Step 1: Create the IDirect3D9 object.

                 IDirect3D9* d3d9    = 0;
                 d3d9                = Direct3DCreate9(D3D_SDK_VERSION);

                 if( !d3d9 )
                 {
                           ::MessageBox(0, "Direct3DCreate9() - FAILED", 0, 0);
                           return false;
                 }

                 // Step 2: Check for hardware vp.

                 D3DCAPS9 caps;
                 d3d9->GetDeviceCaps(D3DADAPTER_DEFAULT, deviceType, &caps);

                 int vp = 0;
                 if( caps.DevCaps & D3DDEVCAPS_HWTRANSFORMANDLIGHT )
                           vp = D3DCREATE_HARDWARE_VERTEXPROCESSING;  //SOFTWARE if debug
                 else
                           vp = D3DCREATE_SOFTWARE_VERTEXPROCESSING;

                 // Step 3: Fill out the D3DPRESENT_PARAMETERS structure.

                 D3DPRESENT_PARAMETERS d3dpp;
                 d3dpp.BackBufferWidth            = width;
                 d3dpp.BackBufferHeight           = height;
                 d3dpp.BackBufferFormat           = D3DFMT_X8R8G8B8;
                 d3dpp.BackBufferCount            = 1;
                 d3dpp.MultiSampleType            = D3DMULTISAMPLE_NONE;
                 d3dpp.MultiSampleQuality         = 0;
                 d3dpp.SwapEffect                 = D3DSWAPEFFECT_DISCARD;
                 d3dpp.hDeviceWindow              = hWnd2;
                 d3dpp.Windowed                   = windowed;
                 d3dpp.EnableAutoDepthStencil     = false;
                 d3dpp.AutoDepthStencilFormat     = D3DFMT_D24S8;
                 d3dpp.Flags                      = 0;
                 d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
                 d3dpp.PresentationInterval       = D3DPRESENT_INTERVAL_IMMEDIATE;

                 // Step 4: Create the device.

                 hr = d3d9->CreateDevice(
                           D3DADAPTER_DEFAULT, // primary adapter
                           deviceType,         // device type
                           hWnd2,               // window associated with device
                           vp, // | D3DCREATE_FPU_PRESERVE, // vertex processing
                           &d3dpp,             // present parameters
                           device);            // return created device

                 if( FAILED(hr) )
                 {
                           // try again using a 16-bit depth buffer
                           d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
```

```
			hr = d3d9->CreateDevice(
				D3DADAPTER_DEFAULT,
				deviceType,
				hWnd2,
				vp,
				&d3dpp,
				device);

			if( FAILED(hr) )
			{
				d3d9->Release(); // done with d3d9 object
				::MessageBox(0, "CreateDevice() - FAILED", 0, 0);
				return false;
			}
		}

		d3d9->Release(); // done with d3d9 object

		return true;
	}

}	//namespace d3d

#endif //GPU_UTILITY_BY_MAJ_JEFFERS
```

This page intentionally left blank.

Bibliography

[Air04] Reticle resolution and scene update rates. Paper provided by AFIWC.

[BFH04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatabalian, M. Houston, P Hanrahan. Brook for GPUs: stream computing on graphics hardware. To appear at SIGGRAPH 2004. http://www.gpgpu.org/

[HeP96] J. Hennessy and D. Patterson. Computer architecture a quantitative approach. 2d ed., Morgan Kaufmann, 1996.

[Int04a] Intel 875 chipset. http://www.intel.com/design/chipsets/875P/pix/875_schematic.gif

[Int04b] PCI Express bandwidth. http://www.intel.com/technology/pciexpress/devnet/desktop.htm

[Jai91] R. Jain. The Art of Computer Systems Performance Analysis. John Wiley & Sons, 1991.

[Joi04] JMASS code portions provided by AFIWC.

[KrW03] J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. ACM Trans. Graph. 22, 3, 908 916, 2003.

[LaM01] E.S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. The International Conference for High Performance Computing and Communications, November 2001.

[LWK03] W. Li, X. Wei and A. Kaufman. Implementing Lattice Boltzmann Computation on Graphics Hardware. The Visual Computer, vol. 19, no.7-8, pp. 444-456, 2003.

[Mac03] M. Macedonia. The GPU enters computing's mainstream. IEEE Computer, October 2003.

[MaV83] J. May and M.E. Van Zee. Electro-optic and infrared sensors. Microwave Journal, September 1983.

[MoA03] K. Moreland and E. Angel. The FFT on a GPU. In Proceedings of Graphics Hardware, Eurographics Association, 2003.

[Mor03] A. Moravanszky. Dense matrix algebra on the GPU. To appear in ShaderX 2 Programming, Wordware, 2003. http://www.gpgpu.org/cgi-bn/blosxom.cgi/Scientific%20Computing/

[Msd04] HLSL flow control. http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000410

[Nvi04] NVidia GeForce 6800 technical specifications. http://www.nvidia.com/page/pg_20040406661996.html

[Pci04a] 3D labs. http://www.pciexpressdevnet.org/news/archive/msg00846.html

[Pci04b] PCI express available this year. http://www.pciexpressdevnet.org/news/archive/msg00842.html

[RuS01] M. Rumpf and R. Strzodka. Using graphics cards for quantized FEM computations. In Proceedings VIIP 2001, 2001.

[THO02] C. Thompson, S. Hahn, M. Oskin. Using modern graphics architectures for general-purpose computing: a framework and analysis. International Symposium on Microarchitecture, 2002.

[TrS01] C. Trendall and A. J. Stewart. General calculations using graphics hardware with applications to interactive caustics. In Rendering Techniques 2000: 11th Eurographics Workshop on Rendering, June 2001.

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 074-0188*
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**			
1. REPORT DATE *(DD-MM-YYYY)* 03-21-2005	**2. REPORT TYPE** **Master's Thesis**		**3. DATES COVERED** *(From – To)* Aug 2003 – Mar 2005
4. TITLE AND SUBTITLE Accelerating Missile Threat Engagement Simulations Using Personal Computer Graphics Cards		**5a. CONTRACT NUMBER**	
		5b. GRANT NUMBER	
		5c. PROGRAM ELEMENT NUMBER	
6. AUTHOR Sean E. Jeffers, Major, USAF		**5d. PROJECT NUMBER**	
		5e. TASK NUMBER	
		5f. WORK UNIT NUMBER	
7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765			**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GE/ENG/05-08
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) 453d EWS/EWA Attn: Ms. Chi Le 102 Hall Blvd STE 331 San Antonio, TX 78243-7020 DSN 969-2391; COMM (210) 977-2391			**10. SPONSOR/MONITOR'S ACRONYM(S)**
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.			
13. SUPPLEMENTARY NOTES			

14. ABSTRACT The 453rd Electronic Warfare Squadron supports on-going military operations by providing battlefield commanders with aircraft ingress and egress routes that minimize the risk of shoulder or ground-fired missile attacks on our aircraft. To determine these routes, the 453rd simulates engagements between ground-to-air missiles and allied aircraft to determine the probability of a successful attack. The simulations are computationally expensive, often requiring two-hours for a single 10-second missile engagement. Hundreds of simulations are needed to perform a complete risk assessment which includes evaluating the effectiveness of countermeasures such as flares, chaff, jammers, and missile warning systems. Thus, the need for faster simulations is acute. This research speeds up these mission critical simulations by using inexpensive commodity PC graphics cards to perform intensive image processing computations used to simulate a heat seeking missile's tracking system. The innovative techniques developed in this research reduce execution time by 33% and incorporate a user-selectable fidelity feature to perform high-fidelity simulations when required. Furthermore, these image processing computations use only 5% of the available computational capacity of the graphics cards, providing a ready source of additional computational power for future simulation enhancements. Analysts can now meet shorter suspenses with more accurate products, ultimately enhancing the safety of Air Force pilots and their weapon systems. With ongoing operations in Iraq and Afghanistan, and a growing threat at home and abroad posed by the proliferation of man-portable missiles, the speed of these simulations play an important role in protecting forces and saving lives.

15. SUBJECT TERMS
Computer graphics, Image Processing, Computerized Simulation, Combat Simulation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Rusty O. Baldwin
REPORT U	**ABSTRACT** U	**c. THIS PAGE** U	UU	171	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, ext 4445; e-mail: rusty.baldwin@afit.edu

Standard Form 298 (Rev: 8-98)
Prescribed by ANSI Std. Z39-18