Matrox Imaging White Paper

GPU-accelerated image processing

Abstract

Today's Graphics Processing Units (GPUs), with their massive parallel architectures and tremendous memory bandwidth, are particularly well suited for accelerating image processing. A careful understanding of a GPU's capabilities and available programming methods is needed to take full advantage of its computational power while achieving quick time-to-market and maintaining maximum flexibility.

Corporate headquarters:
Canada and U.S.A.
Matrox Electronic Systems Ltd.
1055 St. Regis Blvd. Dorval, QC H9P 2T4
Canada
Tel: +1 (514) 685-2630
Fax: +1 (514) 822-6273

Abstract

Today's Graphics Processing Units (GPUs), with their massive parallel architectures and tremendous memory bandwidth, are particularly well suited for accelerating image processing. A careful understanding of a GPU's capabilities and available programming methods is needed to take full advantage of its computational power while achieving quick time-to-market and maintaining maximum flexibility.

GPU capabilities

Not all GPUs offer the same performance level. Performance varies widely from an entry-level GPU integrated into a CPU bridge to a state-of-the-art GPU board. Performance differs as a result of the number of processor cores, core architecture and frequency, width and speed of the memory interface, and the type of host interface. While a GPU's memory bandwidth is measured in the tens of GB/s (e.g., 141.7 GB/s peak for an NVIDIA® GeForce® GTX 280), one must be aware of the overhead needed to get data to and from the GPU, which is measured in the low GB/s (see Figure 1). This overhead will cap the performance of a GPU in terms of the maximum overall data rate for a sequence of operations.

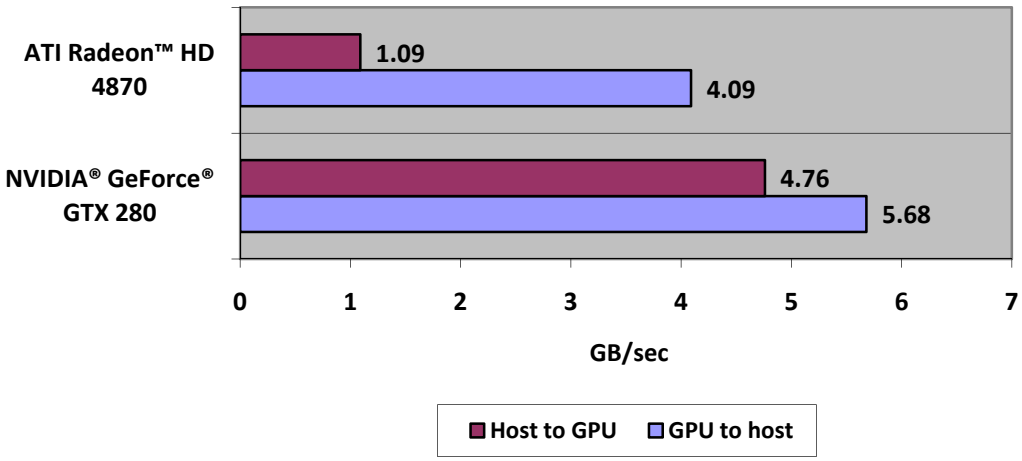

Figure 1 - Bandwidth to and from different GPU boards measured in GB/sec. Measurements were conducted using Matrox Imaging Library (MIL) 9.0 with Update 14 (i.e., based on DirectX® 10) and with the GPU board in a PCIe® 2.0 slot.

Because of their architecture, GPUs are only really suitable for image processing operations that lend themselves to data parallelization where an operation acts simultaneously on several parts of an image (see Figure 2). Moreover, GPUs are not very efficient at executing a sequence of operations that present alternate execution paths or branching. Consequently, GPUs are best at executing fairly long and fixed series of operations.

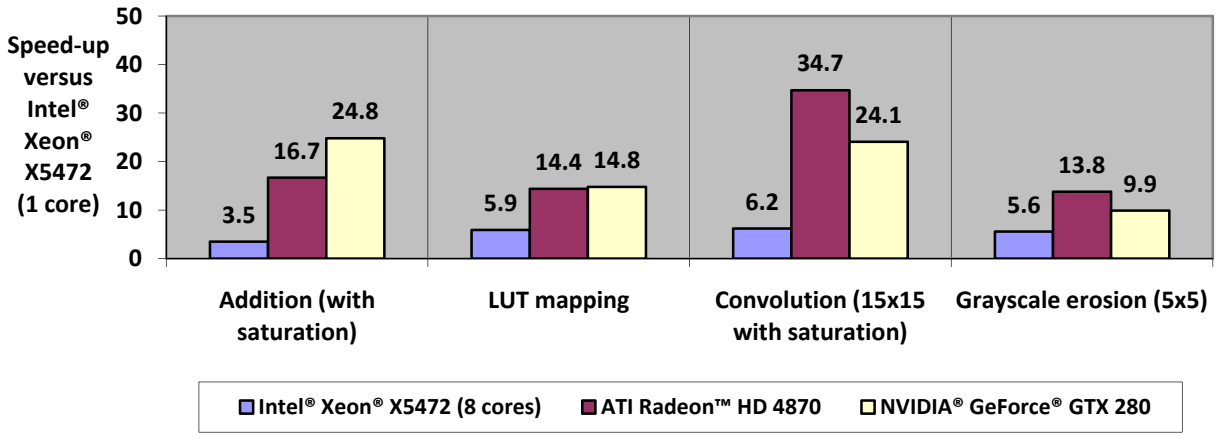

Figure 2 - Relative performance of select image processing operations executed on multi-core CPUs and GPUs as implemented in Matrox Imaging Library (MIL) 9.0 with Update 14 (i.e., based on DirectX® 10). The source image used was 2048 x 2048 x 8-bit and is located in the processor's main memory.

Programming options

A GPU can be programmed using the proprietary software framework provided by the GPU vendor: Stream SDK in the case of AMD/ATI and CUDA™ in the case of NVIDIA®. A GPU can also be programmed using industry standard software frameworks: Microsoft® DirectX® under Windows® and OpenCL. The proprietary software frameworks supplied by AMD/ATI and NVIDIA® are only for use with their respective line of GPUs although both intend to also support the industry standard frameworks within their proprietary frameworks. The industry standard frameworks provide support for GPUs from any vendor: AMD/ATI, Intel® or NVIDIA®. What's more, using an industry standard framework like DirectX®, instead of a proprietary framework like CUDA™, does not lead to a less optimal implementation in terms of performance.

Although both proprietary and industry standard frameworks provide developers with the means to create image processing functions, they do not include image processing specific

functions. Alternatively, the Matrox Imaging Library (MIL) 9.0 application development toolkit, which makes use of DirectX®, includes numerous functions (see Figure 3) that run on a GPU and support different pixel formats (i.e., 8-bit signed, 8-bit unsigned, 16-bit signed, 16-bit unsigned, BGR32, etc.). Additionally, the same MIL image processing functions, which run optimally on a GPU, also run optimally on multiple multi-core CPUs. Using MIL, a developer can easily create an application that uses one or more GPUs if present or just the host CPU(s). With MIL, a developer can also use the CPU, GPU and even FPGA resources concurrently, which enables the distributing of tasks in the best manner for a given application. Finally, an application can employ both MIL functions running on a GPU and custom GPU code using DirectX®, for example (see Appendix).

MbufBayer	MimClip	MimPolarTransform
MbufClear	MimConnectMap	MimProject
MbufCopy	MimConvert	MimRank
MbufCopyCond	MimConvolve	MimResize
MbufCopyClip	MimCountDifference	MimRotate
MbufTransfer	MimDilate	MimShift
MgenLutFunction	MimDistance	MimThick
MgenLutRamp	MimEdgeDetect	MimThin
MgenLutWarp	MimErode	MimTranslate
MimArith	MimFlip	MimWarp
MimArithMultiple	MimLutMap	MregTransformImage
MimBinarize	MimMorphic	

Figure 3 - Matrox Imaging Library (MIL) functions that can run on a GPU.

Conclusion

Making the proper and best use of a GPU for image processing requires a good understanding of its capabilities and available programming methods. The support of GPU-based image processing within MIL 9.0 was the result of extensive training, experimentation and debate regarding which functions to implement and how best to implement them.

MIL 9.0 delivers today a set of image processing functions that truly benefit from GPU acceleration in the most optimal fashion. Being based on DirectX®, MIL 9.0 supports the execution of these functions on present-day and future GPUs from AMD/ATI, NVIDIA® and Intel® (including Larrabee).

A heterogeneous application programming interface (API) gives MIL 9.0 the ability to offer these same functions also optimized to run on multiple multi-core CPUs. Through MIL system and buffer allocation, a programmer can easily assign the execution of these and other image processing functions between different processing resources, enabling load balancing across the different system resources. MIL 9.0 also provides support for Matrox processing hardware based on FPGA technology.

MIL 9.0 gives developers with the best of both worlds: the possibility to use already optimized image processing functions for the GPU with custom GPU code developed using DirectX® and even vendor-specific frameworks like CUDA™.

Finally, MIL 9.0 comes with the added benefit of presenting a developer with a single consistent API for performing image capture, processing, analysis, display and archiving. MIL 9.0 simplifies the development of imaging applications and accelerates their time to market.

Appendix – Example of a MIL-based application integrating custom DirectX® code.

```
/* ---------------------------------------------------------------------------------------
 *
 *  MilInteropWithDX9.cpp
 *
 *  This program performs an image processing operation (alpha blending) on a GPU.
 *  It uses MIL and its GPU driver to allocate the GPU, the display and the processing
 *  buffers and performs the processing using a custom DirectX operation.
 *
 */

#include <mil.h>
#include "d3d9.h"
#include "d3dx9.h"

/* Target image specifications. */
#define IMAGE_FILE1               M_IMAGE_PATH MIL_TEXT("BaboonMono.mim")
#define IMAGE_FILE2               M_IMAGE_PATH MIL_TEXT("Polar.mim")

#define DELTA    0.001
#define MAXIMUM  1.0
#define MINIMUM  0.0

/* Vertex definition */
struct CUSTOMVERTEX {FLOAT X, Y, Z; FLOAT U1, V1;};
#define CUSTOMFVF (D3DFVF_XYZ | D3DFVF_TEX1)

/* Prototypes declaration */
void BlendOnDX9withMILbuffers();
inline void ComputeAlphaAndDelta(MIL_DOUBLE &curAlpha, MIL_DOUBLE &curDelta);

/* Main function. */
/* -------------- */
int MosMain(void)
   {
   /* Run alpha blending using DX9 with MIL buffers */
   BlendOnDX9withMILbuffers();
   }

/* Function doing an alpha blending using DX9 with MIL buffers. */
/* ------------------------------------------------------------ */
void BlendOnDX9withMILbuffers(void)
   {
   MIL_ID   MilApplication,  /* Application identifier.         */
            MilSystem,       /* System identifier.              */
            MilDisplay,      /* Display identifier.             */
            MilDisplayImage, /* Image buffer identifier.        */
            MilSourceImage1, /* Image buffer identifier.        */
            MilSourceImage2; /* Image buffer identifier.        */

   MIL_INT    ImageWidth, ImageHeight, ImageType;
   MIL_DOUBLE Alpha = 1.0;
   MIL_DOUBLE Delta = DELTA;

   LPDIRECT3DDEVICE9 pD3DDevice = NULL;

   /* Allocate defaults. */
   MappAlloc(M_DEFAULT+M_DX_VERSION(9), &MilApplication);
```


MATROX IMAGING

```
MsysAlloc(M_SYSTEM_GPU, M_DEFAULT, M_DEFAULT, &MilSystem);
MdispAlloc(MilSystem, M_DEFAULT, MIL_TEXT("M_DEFAULT"), M_DEFAULT, &MilDisplay);

/* Restore the source image. */
MbufRestore(IMAGE_FILE1, MilSystem, &MilSourceImage1);
MbufRestore(IMAGE_FILE2, MilSystem, &MilSourceImage2);

/* Allocate a display buffer and show the source image. */
MbufAlloc2d(MilSystem, MbufInquire(MilSourceImage1, M_SIZE_X, &ImageWidth),
                       MbufInquire(MilSourceImage1, M_SIZE_Y, &ImageHeight),
                       MbufInquire(MilSourceImage1, M_TYPE,   &ImageType),
                       M_IMAGE+M_PROC+M_DISP, &MilDisplayImage);
MbufCopy(MilSourceImage1, MilDisplayImage);
MdispSelect(MilDisplay, MilDisplayImage);

/* Custom made shader processing */
/*-------------------------------*/

LPDIRECT3DTEXTURE9 SourceTexture1 = NULL;
LPDIRECT3DTEXTURE9 SourceTexture2 = NULL;
LPDIRECT3DSURFACE9 DestSurface    = NULL;

/* Inquire MIL D3D9 device */
MIL_INT VideoDevice = MsysInquire(MilSystem, M_ASSOCIATED_VIDEO_DEVICE_INDEX, M_NULL);
MdispInquire(M_VIDEO_DEVICE_ID + VideoDevice, M_MAIN_D3D9_OBJECT, (MIL_INT *)&pD3DDevice);

/* Compile shader */
LPD3DXEFFECT Effect = NULL;
D3DXCreateEffectFromFile(pD3DDevice, MT("Blend.fx"), NULL, NULL, 0, NULL, &Effect, M_NULL);

/* Create and fill vertex buffer */
VOID* pVoid;
LPDIRECT3DVERTEXBUFFER9 VertexBuffer   = NULL;
struct CUSTOMVERTEX VertexDefinition[] =
   {
      {-1,  1, 0, 0, 0},
      { 1,  1, 0, 1, 0},
      {-1, -1, 0, 0, 1},
      { 1, -1, 0, 1, 1},
   };
pD3DDevice->CreateVertexBuffer(4*sizeof(CUSTOMVERTEX), 0,
                               CUSTOMFVF, D3DPOOL_MANAGED, &VertexBuffer, NULL);

/* Run shader */
UINT Passes;

while (!MosKbhit())
   {
   /* Lock source image and inquire D3D9 texture */
   MbufControl(MilSourceImage1, M_LOCK+M_GPU_ACCESS+M_READ, M_DEFAULT);
   SourceTexture1 = (LPDIRECT3DTEXTURE9)MbufInquire(MilSourceImage1, M_D3D9_TEXTURE, M_NULL);
   MbufControl(MilSourceImage1, M_UNLOCK, M_DEFAULT);

   /* Lock source image and inquire D3D9 texture */
   MbufControl(MilSourceImage2, M_LOCK+M_GPU_ACCESS+M_READ, M_DEFAULT);
   SourceTexture2 = (LPDIRECT3DTEXTURE9)MbufInquire(MilSourceImage2, M_D3D9_TEXTURE, M_NULL);
   MbufControl(MilSourceImage2, M_UNLOCK, M_DEFAULT);

   /* Lock destination image and inquire D3D9 surface */
   MbufControl(MilDisplayImage, M_LOCK+M_GPU_ACCESS+M_WRITE, M_DEFAULT);
```

```
      DestSurface = (LPDIRECT3DSURFACE9)MbufInquire(MilDisplayImage, M_D3D9_SURFACE, M_NULL);
      MbufControl(MilDisplayImage, M_UNLOCK, M_DEFAULT);

      /* Acquire exclusive access to D3D device */
      MsysControl(MilSystem, M_LOCK+M_GPU_ACCESS, M_DEFAULT);

      /* Copy vertices position in vertex buffer */
      VertexBuffer->Lock(0, 0, (void**)&pVoid, 0);
      memcpy(pVoid, VertexDefinition, sizeof(VertexDefinition));
      VertexBuffer->Unlock();
      pD3DDevice->SetFVF(CUSTOMFVF);

      /* Blend two sources images */
      Effect->SetTechnique(Effect->GetTechniqueByName(MT("Blend")));
      Effect->SetFloat(Effect->GetParameterByName(NULL, MT("Alpha")), (float)Alpha);
      pD3DDevice->SetStreamSource(0, VertexBuffer , 0, sizeof(CUSTOMVERTEX));
      pD3DDevice->SetTexture(0, SourceTexture1);
      pD3DDevice->SetTexture(1, SourceTexture2);
      pD3DDevice->SetRenderTarget(0, DestSurface);
      Effect->Begin(&Passes, 0);

      for(UINT PassIndex = 0; PassIndex < Passes; PassIndex++)
         {
         Effect->BeginPass(PassIndex);
         while(FAILED(pD3DDevice->BeginScene()))
            MosSleep(0);
         pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
         pD3DDevice->EndScene();
         Effect->EndPass();
         }
      Effect->End();

      /* Release exclusive access to D3D device */
      MsysControl(MilSystem, M_UNLOCK+M_GPU_ACCESS, M_DEFAULT);

      /* Tell display that image was updated outside MIL */
      MbufControl(MilDisplayImage, M_MODIFIED, M_DEFAULT);

      /* Update alpha blending parameter */
      ComputeAlphaAndDelta(Alpha, Delta);
      }
   MosGetch();

   /* Free everything */
   VertexBuffer->Release();
   Effect->Release();

   MbufFree(MilSourceImage1);
   MbufFree(MilSourceImage2);
   MbufFree(MilDisplayImage);
   MappFreeDefault(MilApplication, MilSystem, MilDisplay, M_NULL, M_NULL);
   }

inline void ComputeAlphaAndDelta(MIL_DOUBLE &Alpha, MIL_DOUBLE &Delta)
   {
   Alpha -= Delta;
   if (Alpha < MINIMUM)
      {
      Alpha = MINIMUM;
      Delta = -Delta;
```

```
      }
   else if (Alpha > MAXIMUM)
      {
      Alpha = MAXIMUM;
      Delta = -Delta;
      }
   }
```

```
//////////////////////////////////////////////////////////////////
//
// Filename          :  Blend.fx
// Content           :  Direct3D HLSL effect file Blend
//
// Copyright © 2006 Matrox Electronic Systems Ltd.
// All Rights Reserved
//
//////////////////////////////////////////////////////////////////

////////////////////////////
// Globals
////////////////////////////

sampler BaseTex0;
sampler BaseTex1;
float Alpha;

////////////////////////////
// FRAGMENT SHADER
////////////////////////////

struct VS_INOUT
{
   vector   position    : POSITION;
   vector   texcoord0   : TEXCOORD0;
};

struct PS_INPUT
{
   float2   base0       : TEXCOORD0;
};

struct PS_OUTPUT
{
   vector   diffuse     : COLOR0;
};

VS_INOUT VS_BLEND( VS_INOUT input )
   {
   // Set default values
   VS_INOUT output = (VS_INOUT)0;

   output.position  = input.position;
   output.texcoord0 = input.texcoord0;

   return output;
   }

PS_OUTPUT PS_BLEND( PS_INPUT input )
   {
```

```
    // zero out members of output
    PS_OUTPUT output = (PS_OUTPUT)0;

    // sample textures
    vector a = tex2D(BaseTex0, input.base0);
    vector b = tex2D(BaseTex1, input.base0);

    // save the resulting pixel color
    output.diffuse = (a * Alpha) + (b * (1.0 - Alpha));

    return output;
    }

//////////////////////////
// TECHNIQUES
//////////////////////////

technique Blend
{
pass p0
    {
    VertexShader    = compile vs_3_0 VS_BLEND();
    PixelShader     = compile ps_3_0 PS_BLEND();
    }
}
```

Matrox Electronic Systems Ltd.
1055 St-Regis Blvd.
Dorval, Quebec, Canada
H9P 2T4
Tel: (514) 822-6000
Fax: (514) 822-6363