Michael D. McCool

Signal Processing and General-Purpose Computing on GPUs

Graphics processing units (GPUs) are the massively parallel high-performance processors used for graphics accelerators in personal computers. These processors have been driven to very high levels of performance and low price-points by the need for real-time computer graphics in mass-market gaming. However, GPUs are no longer specialized for computer graphics. Over the last five years, they have evolved a general-purpose programmable architecture and supporting ecosystem that make possible their use for a wide range of nongraphics tasks, including many applications in signal processing (SP). Commercial, high-performance SP applications that use GPUs as accelerators for general-purpose (GP) tasks (referred to as GP-GPU processing) are still emerging, but many aspects of the architecture of GPUs and their wide availability make them interesting options for implementing and deploying such applications. In this article, we will review some SP applications of GPUs, summarize their key architectural features, and survey the options available for programming these devices.

GPUs IN SP APPLICATIONS

Several signal and image processing algorithms have been implemented on the GPU, including basic operations (such as the fast Fourier transform [1], [2] and convolution); differential equation-based algorithms; and pattern recognition and vision algorithms (such as those for sequence alignment, optical flow estimation, tracking, and stereo reconstruction [3]). The GPU is well-suited for convolution and warping algorithms in image processing due to its hardware support for image interpolation and filtering. The GPU is also well-suited for robust edge detection in the presence of noise, image segmentation, and other applications that may require differential-equation-based approaches. In such cases, implementing explicit differential equation solvers is straightforward. Implementing implicit solvers requires the solution of large sparse systems of linear equations. In this case, it has been demonstrated that the GPU can be used to efficiently implement iterative methods such as the conjugate-gradient algorithm. The GPU is useful in implementations of both dynamic programming and hidden Markov model approaches to sequence alignment and pattern recognition. It should also be

GLOSSARY

- **Shader:** a function that is applied to every pixel drawn by a GPU in order to compute its color or to modify attributes attached to the vertices of polygons. The shading units are the primary programmable resources of GPUs.
- **Shading language:** a special-purpose programming language used to specify the computation to be performed by a shader.
- **Samplers** (in shading languages): objects that are used to specify the access mode to a texture. The graphics API has to bind a texture to a texture unit used by the sampler in order to enable the shader to access the data stored in the texture.
- **Stream kernel:** a function that is applied to one or more arrays of data, generating one or more output arrays. The input and output arrays are accessed in a coherent fashion, leading to greater performance than is possible with random access. Stream kernels are usually implemented using shaders when the stream processing model is mapped onto GPUs.
- **Stream processing:** a massively data-parallel processing model that emphasizes coherent access to data by stream kernels.
- **Texture:** an array holding image data, which may be accessed randomly by a shader. Normally, textures are used to hold images that can be pasted on surfaces during rendering, but they can also be used to hold arbitrary data. Texture fetch operations include interpolation and multiresolution filtering functionalities.
- **Rasterization:** a sampling process that determines which pixels are covered by a geometric primitive, such as a polygon.
- **Rendering:** the process of constructing a 2-D image from a 3-D scene description, performed on a GPU by drawing polygons into color and depth buffers. Polygons are rasterized, and a shader is invoked for each pixel. The resulting shaded pixels are drawn into a framebuffer holding both color and depth. Writes are predicated on depth; only the closest pixel to the eye is retained in the final image.
- **Framebuffer:** the destination array for a rendering operation. Framebuffers may also be allocated in off-screen memory, and for multipass algorithms may share memory with textures.
- **Gather:** a parallel random access memory read operation.
- **Scatter:** a parallel random access memory write operation.
- **Predicated write:** a write operation that is only performed if some condition is satisfied. An example is the depth test at the end of a GPU rendering pass, in which a shaded pixel is only written if it is closer to the eye than the pixel already in the framebuffer.

1053-5888/07/$25.00©2007IEEE

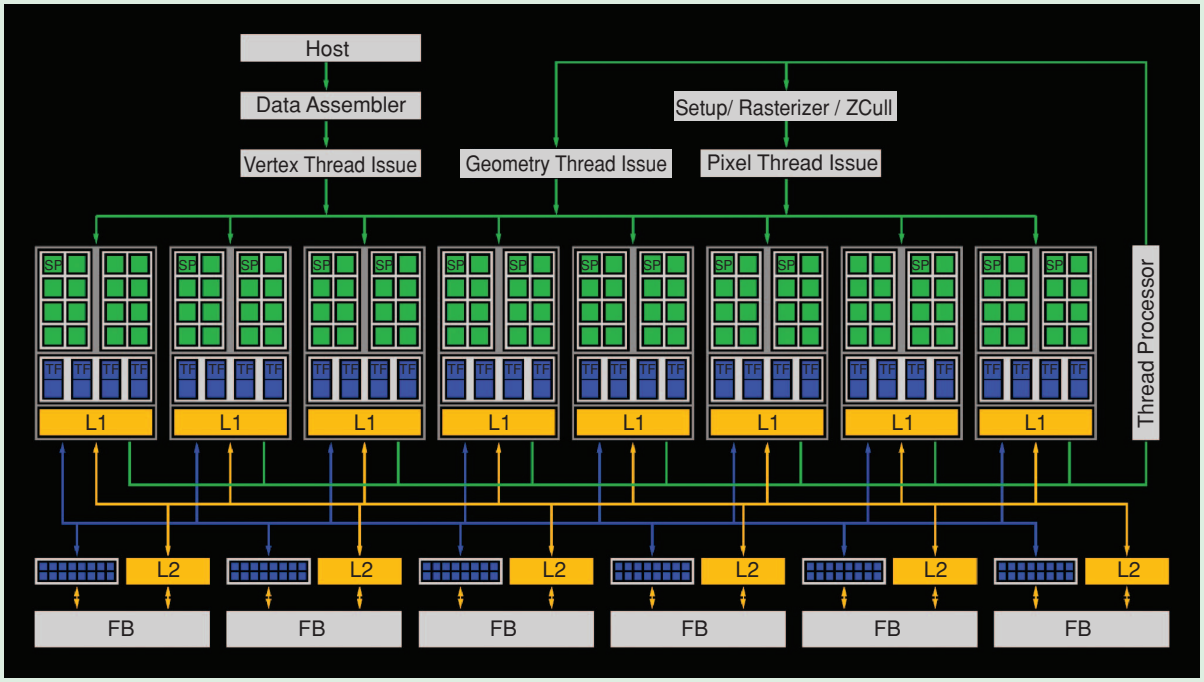
possible, especially on newer GPUs, to build implementations of computationally intensive applications such as video coding that could significantly outperform implementations on traditional central processor units (CPUs). Key components of such codecs, such as motion compensation and wavelet transformation, have already been implemented on GPUs.

What makes GPUs interesting for SP applications is that their peak computational performance can be more than an order of magnitude higher than that of traditional CPUs. They also support very high bandwidth to a relatively large amount of dedicated memory. For example, on the GeForce 8800 GTX video accelerator—using the latest NVIDIA consumer desktop GPU—up to 330 GFlop/s have been observed (based on the achievable issue rate of multiply-accumulate instructions [4]). NVIDIA also claims an 86.4 GB/s peak memory bandwidth. This is available on a consumer-level video card with 768 MB of dedicated high-speed memory for a street price of well under US$700, enabling a range of high-performance supercomputer-class applications on existing commodity computers.

GPU HARDWARE ARCHITECTURES

GPUs use massively parallel architectures with a very large number of floating-point functional units and extensive support for efficient data parallelism. Their architecture is built around the need to access data efficiently and schedule large amounts of parallel computation. Extensive use is made of single-instruction multiple-data (SIMD) parallelism, in which one instruction causes a single operation to take place on more than one value at the same time. SIMD parallelism in GPUs can be explicit (using vector registers and vector operations) or implicit (performing computations over blocks of pixels.) However, GPUs are not pure SIMD machines and do support a coarse-grained, structured version of multithreading; they are actually arrays of data-parallel processors.

GPUs are integrated into a memory hierarchy consisting of registers, cache, video memory, and host memory, with various capacities and latencies. Registers provide a small amount of high-bandwidth, low-latency memory while the DRAM in video and host memory provides higher capacities at higher latencies and lower bandwidth. The amount of high-bandwidth local memory available, including registers and cache, is especially important for algorithms that require a large "working set" (simultaneously accessed state). Until recently, registers were the only form of on-chip memory on GPUs supporting read-write operations; caches were read-only and relatively small. However, the latest generation of GPUs now includes support for a small amount of randomly accessible read/write high-speed on-chip local memory. This will greatly expand the class of algorithms that can be mapped efficiently onto GPUs. Note, however, that even without exploiting local memory, the implementation of important SP algorithms on GPUs can outperform CPUs by an order of magnitude.



[FIG1] Block diagram of the G80 GPU architecture used in the NVIDIA GeForce 8800 GTX (Copyright 2006, NVIDIA Corporation). SP is a scalar floating-point unit; TF is a texture fetch unit; L1 and L2 are on-chip local memories; and FB is a bank of external DRAM framebuffer memory.

A high-level view of one recent GPU architecture—the G80 GPU architecture used in the NVIDIA GeForce 8800 GTX—is illustrated in Figure 1. The G80 architecture consists of eight clusters, each of which has two data-parallel cores. Each core has eight floating-point scalar functional units. The cores in a cluster share a local memory (labeled L1 in the figure) and banks of specialized hardware (labeled TF) for implementing texture fetch operations. In addition, the multibanked memory system enables high-performance access to the framebuffer memory (FB) used to store both texture data and rendered images.

PROCESSING MODELS

For various reasons, direct access to GPU hardware is generally not made available by the hardware vendors. Instead, a number of application programming interfaces (APIs) and programming languages are available for programming GPUs at various levels of abstraction. Algorithms expressed in these languages are then translated into implementations that execute on the hardware.

STREAM PROCESSING

APIs and programming languages are designed around conceptual processing models. A processing model is an abstract model of an idealized computer architecture, used by programmers to reason about the execution of their programs. A rendering pipeline processing model is used for graphics APIs, but various forms of the data-parallel stream processing model are typically used for GP-GPU programming platforms.

The stream processing model supports scalable data parallelism and is independent of the number of processor cores; the workload is automatically distributed over whatever computational resources are available. Stream processing emphasizes coherent access to memory and memory locality, which is crucial for performance. Input data is streamed in from one or more input arrays, processed by a stream kernel, and then streamed out to one or more output arrays. A stream kernel can be thought of as a function that is applied in parallel to every element of one or more input arrays and produces one or more output arrays. Stream processing is especially well-suited to many SP applications, which tend to be data-intensive.

MASSIVE PARALLELISM

GPUs are designed to execute massively parallel workloads with thousands of available threads at the same time, usually many more than the number of available processor cores on the chip. The data-parallel strategy used by GPUs and their programming tools supports this level of parallelism, since an independent "thread" can (potentially, but not necessarily) be created for each element of large input data sets. The emphasis on massive parallelism also allows GPUs to use a variety of strategies to hide memory access and pipeline latencies and achieve higher performance. For instance, a single processing element in a GPU might run several threads at once and switch between these whenever a high-latency operation is encountered. The tradeoff is that the amount of memory available to each thread of execution is relatively small. Using more memory in each thread will reduce the number of active threads that can be supported and will reduce performance by exposing more latency.

CONTROL FLOW

GPUs support control flow and should actually be considered single-program multiple-data (SPMD) processors. When control flow is used on a GPU, however, SIMD masking is often used to implement it in a data-parallel fashion over a relatively large number of functional units. Masking executes an instruction over a number of functional units, but discards the results on a subset of these functional units. For instance, to execute an IF statement, a GPU will evaluate the condition over a number of threads synchronized at the instruction level. It will then mask out the functional units for which the condition is false and execute the instructions in the THEN block of the IF statement, then complement the mask and execute the instructions in the ELSE block. If, however, the evaluation of the condition returns all true or all false for all threads, then the execution of the ELSE or the THEN instructions can be omitted. Due to the use of masking, control flow will reduce efficiency if it is incoherent, and so it should only be used to avoid significant amounts of computation.

MEMORY ACCESS

Memory access operations on GPUs take four forms: random access reads (gathers); writes to predetermined array locations (structured writes); writes to random memory locations (scatters); and finally streaming reads (and writes, on the latest GPUs).

Random memory read operations can be implemented on GPUs with texture fetches, which involve much more than a simple "load" operation. During the use of a GPU for image synthesis (rendering), a texturing operation is available to paste images onto surfaces. Since pixels in a framebuffer (the target of a rendering operation) can be drawn in an order different than the storage order of image samples in textures, supporting texturing requires both random access into textures and hardware support for interpolation and antialiasing. This functionality can be used directly to implement image processing operations such as warping and perspective correction.

Structured write operations to predetermined locations in an output image are supported on GPUs. Structured writes support a number of read-modify-write operations on their destination framebuffers, including predicated writes and summation. The structured write operation can be very fast, as GPUs have numerous hardware optimizations for this mode of memory access.

Write operations to random locations in an output array and streaming reads and writes can also be implemented. Scatters, however, are relatively slow, and may have to be implemented using multiple passes through the rasterizer or other GPU-specific mechanisms. Streaming reads and writes access arrays in a predictable, coherent order that allows several optimizations such as prefetching and block transfer. Streaming access is fundamental to the stream

processing model and appears in newer graphics APIs as well, but on older GPUs must be simulated using random access reads and structured writes. Even on such GPUs, however, this implementation of streaming access maximizes bandwidth.

GPU PROGRAMMING FOR SP

To implement an SP application on a GPU and obtain maximum performance, it is necessary to choose and efficiently implement a suitable data-parallel algorithm. One approach is to use a graphics API. However, graphics APIs simply are not tuned for GP-GPU applications. In particular, their mechanisms for data management are weak, the costs for allocation and deallocation of memory are high, and host CPU time is spent managing graphics state and resources that are of no interest to GP-GPU applications. From a conceptual point of view, using a graphics API also forces a user only interested in GP computation to learn and then "abuse" many irrelevant graphics concepts.

Better solutions include using APIs from GPU vendors or third-party programming platforms that directly target GP-GPU applications. Third-party platforms also provide portability to other massively multicore processors. The available graphics APIs, the vendor tools, and the leading third-party programming platforms are reviewed next. Although more sophisticated SP applications can and have been implemented on GPUs [5], [6], for brevity we will use only a simple two-dimensional (2-D) finite impulse response (FIR) convolution for all examples.

GRAPHICS APIs

There are two main graphics APIs available for programming GPUs (OpenGL and Microsoft's Direct3D) and three main shading languages (NVIDIA's Cg, Microsoft's HLSL, and the OpenGL Shading Language—often known as GLSL). OpenGL has the advantage of being available on Linux, Windows, and OSX, whereas Direct3D is only available under Windows. Likewise, HLSL is only available under Windows; it is, however, quite similar to Cg, which can work with either OpenGL or Direct3D. Finally, GLSL is the built-in shading language for OpenGL. Semantically, all three shading languages have similar features, so we will only refer to Cg in the examples presented later.

VENDOR TOOLS

Recently, two new vendor tools have become available for programming GPUs. ATI has released a C interface using a stream processing model called Close-to-the-Metal (CTM), and NVIDIA has released an interface called CUDA that targets their latest generation of GPUs. The CTM interface exposes a simple virtual machine for managing memory and processing passes and exposes the instruction set architecture of the ATI x1900 series GPUs for defining stream kernels. Writing directly to the CTM interface will result in code that only works on certain ATI GPUs and also requires assembly language programming. However, the CTM interface is an excellent backend target for higher-level GP-GPU tools. CUDA supports a dialect of C with some extensions to allow stream kernels to be expressed in an inline fashion. It is based on a stream processing model but includes some generalizations to support thread synchronization and shared on-chip memory. When compiled, CUDA programs are transformed into a combination of C and NVIDIA assembly instructions. While higher level than CTM, the CUDA interface still has several NVIDIA-specific features and cannot be used to target non-NVIDIA GPUs.

THIRD-PARTY PROGRAMMING PLATFORMS

Three third-party GPU programming platforms—Accelerator, PeakStream, and RapidMind—target vendor-independent programming of GPUs and also apply to other high-performance multicore processors. These systems all use an embedded programming interface in which the concepts of API and programming language are combined. An embedded programming interface defines certain data types in a standard host language to emulate arrays and tuples of numbers. Operations on these types are then used to express a desired computation. However, rather than being executed on the host, sequences of operations on these types are captured by the programming interface, compiled dynamically, and then executed in parallel on one or more coprocessors. Embedded programming interfaces avoid the glue code needed to bind special-purpose languages such as Cg to the host application. Microsoft's Accelerator is embedded in C#, while RapidMind and PeakStream are embedded in C++. Accelerator and PeakStream support array types and infer stream kernels from operations on them. RapidMind combines an array type with explicit definition of stream kernels.

Cg CONVOLUTION EXAMPLE

To better understand some of the characteristics of the available GPU programming solutions, we will consider a simple convolution example. An implementation of this example written in Cg is given in Figure 2(a). Here, a 2-D FIR convolution stream kernel is expressed using the Cg shader function `convolve`. To actually use this shader function as a stream kernel, a graphics API (OpenGL or Direct3D) must also be used to load the image and filter data into texture maps, bind the textures and shaders, and then draw a polygon onto an invisible off-screen framebuffer. The polygon needs to be specified as a list of vertices with a pair of numbers called texture coordinates attached to each. These texture coordinates are interpolated over the framebuffer locations covered by the polygon. During rasterization of this polygon, the shader given in the figure is invoked for each pixel, with the interpolated texture coordinate assigned to the `pos` value. The interpolated texture coordinates give each pixel a unique part of the input data to process. The `uniform` keyword identifies values that are the same for all pixels/threads, and samplers are bound to texture units into which the image and filter data are loaded.

Cg Convolution Example 1

```
COLORTYPE
convolve(
  uniform samplerIMG image : TEXUNIT0,    // the input image
  uniform samplerRECT filter : TEXUNIT1,  // the filter texture
  uniform int filter_width,               // filter width
  uniform int filter_height,              // filter height
  uniform float2 filter_offset,           // filter offset
  uniform float2 texel_size,              // texture scale factor
  float2 pos : TEXCOORD0                  // position in image
) : COLOR
{
  COLORTYPE c = 0;
  for (int x = 0; x < filter_width; x++) {
    for (int y = 0; y < filter_height; y++) {
      float2 u = float2(x,y);
      float weight = texRECT(filter, u).r;
      float2 tap = pos - (u + filter_offset) * texel_size;
      c += texIMG(image, tap) * weight;
    }
  }
  return c;
}
```

(a)

Cg Convolution Example 2

```
COLORTYPE
convolve(
  uniform samplerIMG image,       // the input image
  uniform color weight[],         // filter coefficients
  uniform float2 offset[],        // filter offsets
  uniform int filter_taps,        // number of taps
  float2 pos : TEXCOORD0          // position in image
) : COLOR
{
  COLORTYPE c = 0;
  for (int i = 0; i < filter_taps; i++) {
    c += texIMG(image, pos - offset[i]) * weight[i];
  }
  return c;
}
```

(b)

[FIG2] **Cg convolution example.**

Although GPUs can excel at convolution, this implementation is not fully optimized. For instance, it might be the case that certain filter coefficients are zero, and so we do not have to actually perform any memory reads or computation for these. In this example, there is also an offset computation in the shader for each tap, as well as a memory access to obtain each filter coefficient. An improved implementation that depends on precompiled `weight` and `offset` arrays for each tap is included in Figure 2(b).

In these Cg implementations, a large amount of code is not shown. Using the graphics API, additional host CPU code is required to set up and execute the computation; to set up the `weight` and `offset` arrays by analyzing the filter for nonzero coefficients [in the case of Figure 2(b)]; and to bind and modify uniform variables. Shading languages such as Cg are sufficient for setting up the stream kernel, but do not handle data management and other tasks.

RAPIDMIND CONVOLUTION EXAMPLE

Another implementation of convolution using the RapidMind platform [7] is shown in Figure 3(a). The code given is in fact C++ and can be included in any C++ program that interfaces to the RapidMind runtime library. The `BEGIN` and `END` keywords indicate that the operations between them on `Value` types should be stored in the program object `convolve`, which represents a stream kernel. Once defined, a program object can be executed in parallel by applying it as a function to an array, generating a new array. In this case, the input to the program object is generated by `grid`, which generates a "virtual" array containing tuples of natural numbers. The implementation of virtual arrays is procedural and does not consume any memory.

The control flow and all uniform computations in Figure 3(a) are executed when the kernel is defined on the host CPU, not when it is executed on the GPU. This implementation is, in fact, equivalent to the optimized Cg implementation shown in Figure 2(b). Since only the operations that manipulate `Value` types are actually stored in the program object, if a filter coefficient at a certain offset is zero, the `if` statement in the inner loop ensures that no code is generated for that tap. The final code on the GPU contains no control flow, just a sequence of memory accesses and multiply-add instructions for the desired taps. It is, however, also possible to specify dynamic control flow (which *does* get embedded into the generated GPU code) when necessary using `FOR/ENDFOR` and `IF/THEN/ELSE/ENDIF` keywords. The computation of the offsets for the various taps actually used is automatically identified as a "uniform" computation that is performed once only on the host processor, making it unnecessary to explicitly define an offset array. Although the RapidMind example is

RapidMind Convolution Example 1

```
float filter[filter_width][filter_height];        // filter
Value2i filter_offset;                            // filter center
Array<2,Value1f> image(image_width,image_height); // image

// define the stream kernel
Program convolve = BEGIN {
  In<Value2i> pos;                  // position in input image
  Out<Value1f> result = Value1f(0.0f);
  for (int x = 0; x < filter_width; x++) {
    for (int y = 0; y < filter_height; y++) {
      if (filter[x][y] != 0.0f) {
        // accumulate only taps with non-zero weights
        Value2i tap = pos - (Value2i(x,y) + filter_offset);
        result += filter[x][y] * image[tap];
      }
    }
  }
} END;

// execute the stream kernel on all input pixels
image = convolve(grid(image_width,image_height));
```

(a)

RapidMind Convolution Example 2

```
Value2f weight[filter_taps];      // non-zero filter coefficients
int offset[filter_taps][2];       // tap offsets
Array<2,Value1f> image(image_width,image_height); // image

// define the stream kernel
Program convolve = BEGIN {
  In<Value1f> sample[filter_taps];
  Out<Value1f> result = Value1f(0.0f);
  for (int i = 0; i < filter_taps; i++) {
    result += weight[i] * sample[i]; // accumulate all taps
  }
} END;

// bind the stream kernel to all shifted inputs
for (int i = 0; i<filter_taps; i++) {
  // partially evaluate the kernel over each shifted input
  convolve = convolve << shift(image,offset[i][0],offset[i][1]);
}

// execute the fully pre-bound kernel
image = convolve;
```

(b)

[FIG3] RapidMind convolution example.

longer than that shown in Figure 2(b), it is complete; no other "glue" or setup code is required.

A final implementation, given in Figure 3(b), uses streaming rather than random memory access. Here a `shift` operation is used to create an offset array "view" for each filter tap. This implementation better expresses the coherent memory access patterns typical of convolution, exposing this structure for optimization by the compiler.

CONCLUSIONS

GP-GPU is just the first wave of a shift towards massive parallelism on desktop machines. Recently, there has been considerable activity in multicore processing and related areas; work includes the IBM Cell BE processor, the AMD Fusion project, the AMD Torrenza initiative, and the Geneso PCIe initiative by NVIDIA and Intel. These efforts are taking place in the high-volume consumer market, and so we can expect the widespread deployment of low-cost, high-performance, massively parallel processors on commodity systems. The additional performance enabled by these developments will be directly applicable to a wide variety of computationally demanding SP algorithms.

AUTHOR

Michael McCool (mmccool@rapidmind.net) is an associate professor at the University of Waterloo, Canada, and is also cofounder and chief scientist of RapidMind, Inc. His professional interests include parallel programming and languages, computer architectures, general-purpose computation on GPUs, signal processing, numerical methods, real-time graphics, and other applications of high-performance computing.

REFERENCES

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

[2] N.K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *Proc. SuperComputing*, 2006.

[3] S.M. Seitz, B. Curless, J. Diebel, D. Scharstein, and R. Szeliski, "A comparison and evaluation of multi-view stereo reconstruction algorithms," in *Proc. IEEE Computer Society Conf. Computer Vision and Pattern Recognition (CVPR)*, New York, Nov. 2006, vol. 1, pp. 519–526.

[4] "Low-level comparative benchmarks of GPU performance.," GPUBench Web site [Online]. Available: http://graphics.stanford.edu/projects/gpubench

[5] GPGPU Web site [Online]. Available: http://www.gpgpu.org [See especially the course notes (from Supercomputing 2006) presented in Nov. 2006.]

[6] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A.E. Lefohn, and T.J. Purcell, "A survey of general-purpose computation on graphics hardware," in *Proc. Computer Graphics Forum*, vol. 26, 2007, submitted for publication.

[7] M.D. McCool, "Data-parallel programming on the Cell BE and the GPU using the RapidMind Development Platform," in *Proc. GSPx Multicore Applications Conf.*, Santa Clara, Nov. 2006. SP