A Brief Look at FPGAs, GPUs and Cell Processors

Michael L. Stokes
Ohio Supercomputer Center, Columbus, Ohio

In recent months, there has been much discussion about new advancements in processor technology that promise huge performance returns for a small investment. Field programmable gate arrays (FPGAs), cell processors and graphics processor units (GPUs) are all the rage. So, one question is, "What are the salient features of this new technology?" But perhaps the most important questions, however, are: "What does all of this mean to the test and evaluation (T&E) community? Should these new technologies be adopted, put on the top shelf until the technology matures, or [should they be] written off altogether?"

FPGAs, invented around 1984 by Ross Freeman, Xilinx cofounder, have a relatively large number of programmable logic components with programmable interconnects. The logic components can be programmed to duplicate basic logic functions (AND, OR, XOR and NOT) or grouped to form simple mathematical functions such as integer addition. A complex circuit can be constructed from logic and memory components similar to a programmable breadboard. These components can be reprogrammed after the manufacturing process; thus, they are *field programmable*.

Programming an FPGA is accomplished using hardware description language, which is very similar to assembly language programming for general-purpose central processing units (CPUs). The software developer must be aware of hardware timings, interrupts, signals, thread synchronization and other interfaces at the hardware level. Several research efforts are underway to develop higher-level languages for programming FPGAs to make software development easier and faster.

Typical applications that use FPGAs are cryptography, specialized routers or network edge devices, and medical imaging. These applications can be characterized as those that perform a substantial amount of computation with a small amount of memory, such as the Fast Fourier Transform (FFT). Logic designs that are independent of one another execute in parallel (up to available memory); therefore, multiple FFTs, for example, execute in the same time to execute just one. In this sense, FPGAs are highly scalable. The down side of FPGAs is that limited onboard memory and input/output (IO) bandwidth restricts scalability. FPGAs also do not support floating point operations, thus (intrinsically) limiting the appeal for programming broader applications found in the T&E community.

A relative newcomer to general-purpose programming is the GPU *(see Figure 1)*. A GPU is that fancy graphics card that costs a few hundred dollars for a PC to make

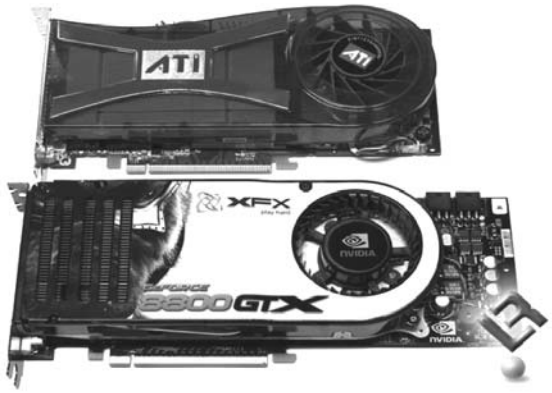

Figure 1. NVIDIA and ATI graphics processor units (GPUs)

video games vivid and realistic. The popular association of GPUs is with accelerating graphics, but the new architectures from manufactures such as NVIDIA Corporation and ATI are capable of performing general-purpose computing in addition to making animated monsters look more life-like. For a good overview of this exciting area, visit ***http://www.gpgpu.org***.

There are two approaches to consider when programming a GPU for general-purpose computing. The first is to pose the problem as a graphics problem and solve it using a graphics language such as OpenGL ***(see http://opengl.org)*** or DirectX ***(http://www.microsoft.com/directX)***. The second approach is to program the GPU directly. But be cautioned: This is not a conventional method of programming, so be prepared to think about the application in some new ways, because what is needed today to program GPUs may not be what is needed in the future.

The first approach to using GPUs is to recast the general-purpose application as a graphics problem, which is not always possible or simple to do. OpenGL or DirectX calls can be employed to define the geome-

try, then used to query the geometry to discover distances, intersections and other properties. The so-called *Line-of-sight* program has been solved this way using various techniques available in OpenGL. This approach is very nice when it works because GPUs are already optimized for OpenGL and DirectX in hardware. The programmer has to know little or nothing about the graphics accelerator or its capabilities. However, this model sometimes does not provide the programmer with enough flexibility. Graphics Language Shading Library is a relatively new capability in OpenGL (as of OpenGL 2.0) that allows additional instructions to be inserted in the vertex and pixel shader component of the OpenGL pipeline. While this offers substantial flexibility over the simple graphics API, the method still requires that the problem be cast in a geometric format. So, what if this is not possible?

One solution is the introduction of the Compute Unified Device Architecture (CUDA) ***(http://developer.nvidia.com/object/cuda.html)*** by NVIDIA. The CUDA environment is supported on the NVIDIA G8X and newer graphics adapters, and includes the CUDA toolkit for the Linux and Windows Operating Systems. This toolkit allows the user to access the GPU hardware directly.

The GPU hardware currently sports 16 multiprocessors. Each multiprocessor has a set of eight 32-bit subprocessors with a Single Instruction Multiple Data architecture shared instruction unit, or a total of 128 processors. Each multiprocessor has a set of 32-bit registers per processor, on-chip shared memory with fast access to the processors, a read-only constant cache memory, and a read-only texture cache. In addition, the device contains 768 MB of device memory with much slower access speeds than shared memory. Parallelism is achieved by associating up to 32 threads of execution in a group called a *warp,* of which up to 16 *warps* make up a *block.* Each multiprocessor executes one or more block(s) in parallel.

The peak performance from the NVIDIA 8800 is around 300+ GFLOPS for a unit that lists for around $600—a large performance-to-price ratio when compared to current-day high-performance computing (HPC). However, the GPU is not without its faults. In order to obtain the full throughput of the device, it is necessary to optimize onboard memory and keep the processors fully engaged, which can be difficult for some applications. There is an alternative to the GPU. It is called a cell processor *(see Figure 2)*, and it is one of the hottest options for accelerating HPC applications.

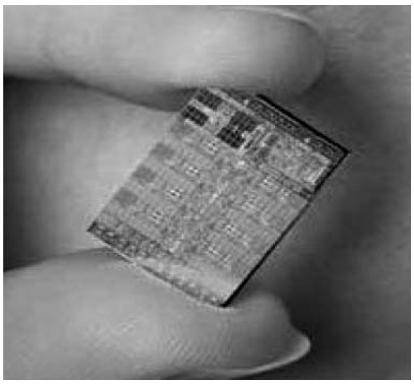

Figure 2. A cell with one Power Processor Element (PPE) and eight Synergistic Processor Elements (SPEs)

"Though sold as a game console, what will in fact enter the home is a Cell-based computer," according to Sony's Ken Kutaragi, in talking about Sony's PlayStation 3 (PS3). The Cell processor was designed from the very beginning to be a general-purpose computer, not as a gaming console, as often rumored. Most companies would have used specialized automated software in the design process, but IBM, the designer of the Cell processor (or simply Cell), decided to perform the design by hand, a choice costing millions more dollars but resulting in a more compact and power-efficient design. The resulting design is nothing less than stellar in the opinion of most industry observers due to its simplified design and power efficiency.

The architecture of the Cell is unique, although it shares a lot in common with older vector supercomputers (recall the Cray 1). The Cell can perform at rates similar to or better than GPUs, but it is much easier to program because it was designed for general-purpose application. A Cell is composed of a number of elements as shown in *Figure 3 (next page)*: one Power Processor Element (PPE), eight Synergistic Processor Elements (SPEs), one Element Interconnect Bus (EIB), one Direct Memory Access Controller (DMAC), two Rambus XDR memory controllers and one Rambus FlexIO interface.

The PPE primarily initiates and monitors jobs that run on the SPEs. The PPE runs the basic operating system and parts of the application, but compute-intensive components of the operating system and the applications are offloaded to the SPEs. The PPE is a 64-bit *Power Architecture* processor compatible with PowerPC binaries. The downside of this design is that one can expect poor performance when executing logic-heavy instructions, because it is an in-order processor (it does not pre-compute logic branches).

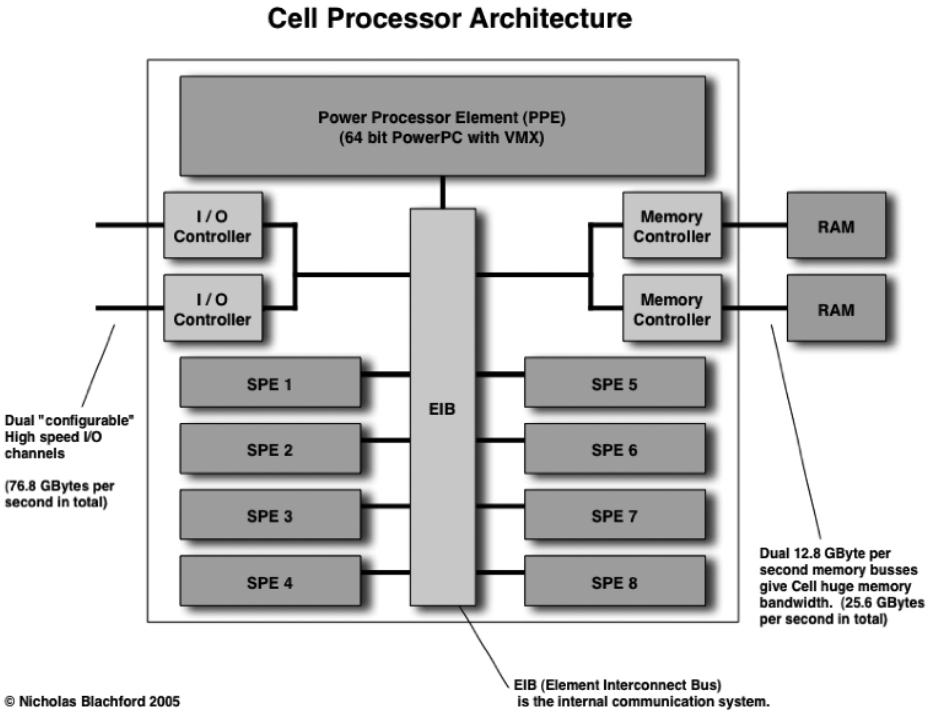

Figure 3. Cell processor architecture

SPEs, operating at clock speeds up to 4 GHz, contain 128 x 128 bit registers, along with 4 single precision floating point units with a maximum peak performance of 32 GFLOPS, 4 integer units capable of 32 GFLOPS, and a 256 K *local store* instead of a cache. Each SPE consumes less than 5 Watts at 4 GHz, making it a better performer than the current batch of GPUs (some of which recommend a 500 W or larger power supply). Like the PPE, SPEs are vector processors, namely, they perform multiple operations simultaneously with a single instruction. Each SPE is capable of 4 x 32 bit operations per cycle (8 if multiply-adds are included). The double precision calculations are Institute of Electrical and Electronics Engineers (IEEE) standard, whereas the single precision calculations are not, which is supposedly faster.

In the current implementation of the PS3, double precision calculations share the computational area of the single precision floating units. This saves a lot of room on the silicon, but it results in a large penalty in performance, or around 25 double precision GFLOPS at 4 Ghz. For a gaming box such as the PS3, double precision is not important, but IBM has hinted that future versions of the Cell might include a full-speed, double precision floating point unit that will perform at rates as high as integer performance, or 256 GLOPS per SPE. SPEs can execute instructions in parallel in synchronized loops, or can also be chained together, forming a stream processor through a process called pipelining. It is through pipelining that the Cell is capable of achieving near-peak performance.

In summary, as has always been the case, the choice of which acceleration architecture is best for any specific application has to be examined for each individual case. FPGAs, in general, are a good choice when seeking a hardware or embedded solution, while GPUs and Cells appear to be better choices for modeling and simulation and simulation and training applications. GPUs are well-suited for applications that can be cast in the form of a graphics solution and some general-purpose applications, while the Cell architecture seems to be the best choice when complex, general-purpose solutions are required. ❑

MICHAEL L. STOKES is an engineer that works for the Ohio Supercomputer Center, but resides at Redstone Technical Test Center (RTTC), Redstone Arsenal, Alabama. Funded by the High Performance Computing Modernization Program Office (HPCMPO) under the PET program, he works in the area of testing and evaluation. His current area of research is in the development of real-time ray tracers for multi- and hyperspectral synthetic scene injection. He received his doctorate degree in engineering science and mechanics at the University of Tennessee, Knoxville, in 1991. Comments and questions can be directed to him at ***michael.l.stokes@us.army.mil****, Room 27, Building 4500, Redstone Arsenal, Alabama 35898.*

Acknowledgment

Images/photographs in this article are copyrighted by, and printed with written permission from, Nicholas Blachford. The author gratefully acknowledges Blachford's contributions.