THE GPU COMPUTING ERA

GPU COMPUTING IS AT A TIPPING POINT, BECOMING MORE WIDELY USED IN DEMANDING CONSUMER APPLICATIONS AND HIGH-PERFORMANCE COMPUTING. THIS ARTICLE DESCRIBES THE RAPID EVOLUTION OF GPU ARCHITECTURES—FROM GRAPHICS PROCESSORS TO MASSIVELY PARALLEL MANY-CORE MULTIPROCESSORS, RECENT DEVELOPMENTS IN GPU COMPUTING ARCHITECTURES, AND HOW THE ENTHUSIASTIC ADOPTION OF CPU+GPU COPROCESSING IS ACCELERATING PARALLEL APPLICATIONS.

John Nickolls
William J. Dally
NVIDIA

●●●●●●As we enter the era of GPU computing, demanding applications with substantial parallelism increasingly use the massively parallel computing capabilities of GPUs to achieve superior performance and efficiency. Today GPU computing enables applications that we previously thought infeasible because of long execution times.

With the GPU's rapid evolution from a configurable graphics processor to a programmable parallel processor, the ubiquitous GPU in every PC, laptop, desktop, and workstation is a many-core multithreaded multiprocessor that excels at both graphics and computing applications. Today's GPUs use hundreds of parallel processor cores executing tens of thousands of parallel threads to rapidly solve large problems having substantial inherent parallelism. They're now the most pervasive massively parallel processing platform ever available, as well as the most cost-effective.

Using NVIDIA GPUs as examples, this article describes the evolution of GPU computing and its parallel computing model, the enabling architecture and software developments, how computing applications use CPU+GPU coprocessing, example application performance speedups, and trends in GPU computing.

GPU computing's evolution

Why have GPUs evolved to have large numbers of parallel threads and many cores? The driving force continues to be the real-time graphics performance needed to render complex, high-resolution 3D scenes at interactive frame rates for games.

Rendering high-definition graphics scenes is a problem with tremendous inherent parallelism. A graphics programmer writes a single-thread program that draws one pixel, and the GPU runs multiple instances of this thread in parallel—drawing multiple pixels in parallel. Graphics programs, written in shading languages such as Cg or High-Level Shading Language (HLSL), thus scale transparently over a wide range of thread and processor parallelism. Also, GPU computing programs—written in C or C++ with the CUDA parallel computing model,[1,2] or using a parallel computing API inspired by CUDA such as Direct-Compute[3] or OpenCL[4]—scale transparently over a wide range of parallelism. Software scalability, too, has enabled GPUs to rapidly increase their parallelism and performance with increasing transistor density.

GPU technology development

The demand for faster and higher-definition graphics continues to drive the

Published by the IEEE Computer Society

Table 1. NVIDIA GPU technology development.

Date	Product	Transistors	CUDA cores	Technology
1997	RIVA 128	3 million	—	3D graphics accelerator
1999	GeForce 256	25 million	—	First GPU, programmed with DX7 and OpenGL
2001	GeForce 3	60 million	—	First programmable shader GPU, programmed with DX8 and OpenGL
2002	GeForce FX	125 million	—	32-bit floating-point (FP) programmable GPU with Cg programs, DX9, and OpenGL
2004	GeForce 6800	222 million	—	32-bit FP programmable scalable GPU, GPGPU Cg programs, DX9, and OpenGL
2006	GeForce 8800	681 million	128	First unified graphics and computing GPU, programmed in C with CUDA
2007	Tesla T8, C870	681 million	128	First GPU computing system programmed in C with CUDA
2008	GeForce GTX 280	1.4 billion	240	Unified graphics and computing GPU, IEEE FP, CUDA C, OpenCL, and DirectCompute
2008	Tesla T10, S1070	1.4 billion	240	GPU computing clusters, 64-bit IEEE FP, 4-Gbyte memory, CUDA C, and OpenCL
2009	Fermi	3.0 billion	512	GPU computing architecture, IEEE 754-2008 FP, 64-bit unified addressing, caching, ECC memory, CUDA C, C++, OpenCL, and DirectCompute

development of increasingly parallel GPUs. Table 1 lists significant milestones in NVIDIA GPU technology development that drove the evolution of unified graphics and computing GPUs. GPU transistor counts increased exponentially, doubling roughly every 18 months with increasing semiconductor density. Since their 2006 introduction, CUDA parallel computing cores per GPU also doubled nearly every 18 months.

In the early 1990s, there were no GPUs. Video graphics array (VGA) controllers generated 2D graphics displays for PCs to accelerate graphical user interfaces. In 1997, NVIDIA released the RIVA 128 3D single-chip graphics accelerator for games and 3D visualization applications, programmed with Microsoft Direct3D and OpenGL. Evolving to modern GPUs involved adding programmability incrementally—from fixed function pipelines to microcoded processors, configurable processors, programmable processors, and scalable parallel processors.

Early GPUs

The first GPU was the GeForce 256, a single-chip 3D real-time graphics processor introduced in 1999 that included nearly every feature of high-end workstation 3D graphics pipelines of that era. It contained a configurable 32-bit floating-point vertex transform and lighting processor, and a configurable integer pixel-fragment pipeline, programmed with OpenGL and Microsoft DirectX 7 (DX7) APIs.

GPUs first used floating-point arithmetic to calculate 3D geometry and vertices, then applied it to pixel lighting and color values to handle high-dynamic-range scenes and to simplify programming. They implemented accurate floating-point rounding to eliminate frame-varying artifacts on moving polygon edges that would otherwise sparkle at real-time frame rates.

As programmable shaders emerged, GPUs became more flexible and programmable. In 2001, the GeForce 3 introduced the first programmable vertex processor that executed vertex shader programs, along with a configurable 32-bit floating-point pixel-fragment pipeline, programmed with OpenGL and DX8. The ATI Radeon 9700, introduced in 2002, featured a programmable 24-bit floating-point pixel-fragment processor programmed with DX9 and OpenGL. The GeForce FX and GeForce 6800[5] featured

....

HOT CHIPS

programmable 32-bit floating-point pixel-fragment processors and vertex processors, programmed with Cg programs, DX9, and OpenGL. These processors were highly multithreaded, creating a thread and executing a thread program for each vertex and pixel fragment. The GeForce 6800 scalable processor core architecture facilitated multiple GPU implementations with different numbers of processor cores.

Developing the Cg language[6] for programming GPUs provided a scalable parallel programming model for the programmable floating-point vertex and pixel-fragment processors of GeForce FX, GeForce 6800, and subsequent GPUs. A Cg program resembles a C program for a single thread that draws a single vertex or single pixel. The multithreaded GPU created independent threads that executed a shader program to draw every vertex and pixel fragment.

In addition to rendering real-time graphics, programmers also used Cg to compute physical simulations and other general-purpose GPU (GPGPU) computations. Early GPGPU computing programs achieved high performance, but were difficult to write because programmers had to express non-graphics computations with a graphics API such as OpenGL.

Unified computing and graphics GPUs

The GeForce 8800 introduced in 2006 featured the first unified graphics and computing GPU architecture[7,8] programmable in C with the CUDA parallel computing model, in addition to using DX10 and OpenGL. Its unified streaming processor cores executed vertex, geometry, and pixel shader threads for DX10 graphics programs, and also executed computing threads for CUDA C programs. Hardware multithreading enabled the GeForce 8800 to efficiently execute up to 12,288 threads concurrently in 128 processor cores. NVIDIA deployed the scalable architecture in a family of GeForce GPUs with different numbers of processor cores for each market segment.

The GeForce 8800 was the first GPU to use scalar thread processors rather than vector processors, matching standard scalar languages like C, and eliminating the need to manage vector registers and program vector operations. It added instructions to support C and other general-purpose languages, including integer arithmetic, IEEE 754 floating-point arithmetic, and load/store memory access instructions with byte addressing. It provided hardware and instructions to support parallel computation, communication, and synchronization—including thread arrays, shared memory, and fast barrier synchronization.

GPU computing systems

At first, users built personal supercomputers by adding multiple GPU cards to PCs and workstations, and assembled clusters of GPU computing nodes. In 2007, responding to demand for GPU computing systems, NVIDIA introduced the Tesla C870, D870, and S870 GPU card, deskside, and rackmount GPU computing systems containing one, two, and four T8 GPUs. The T8 GPU was based on the GeForce 8800 GPU, configured for parallel computing.

The second-generation Tesla C1060 and S1070 GPU computing systems introduced in 2008 used the T10 GPU, based on the GPU in GeForce GTX 280. The T10 featured 240 processor cores, 1-teraflop-per-second peak single-precision floating-point rate, IEEE 754-2008 double-precision 64-bit floating-point arithmetic, and 4-Gbyte DRAM memory. Today there are Tesla S1070 systems with thousands of GPUs widely deployed in high-performance computing systems in production and research.

NVIDIA introduced the third-generation Fermi GPU computing architecture in 2009.[9] Based on user experience with prior generations, it addressed several key areas to make GPU computing more broadly applicable. Fermi implemented IEEE 754-2008 and significantly increased double-precision performance. It added error-correcting code (ECC) memory protection for large-scale GPU computing, 64-bit unified addressing, cached memory hierarchy, and instructions for C, C++, Fortran, OpenCL, and DirectCompute.

GPU computing ecosystem

The GPU computing ecosystem is expanding rapidly, enabled by the deployment of more than 180 million CUDA-capable

GPUs. Researchers and developers have enthusiastically adopted CUDA and GPU computing for a diverse range of applications,[10] publishing hundreds of technical papers, writing parallel programming textbooks,[11] and teaching CUDA programming at more than 300 universities. The CUDA Zone (see http://www.nvidia.com/object/cuda_home_new.html) lists more than 1,000 links to GPU computing applications, programs, and technical papers. The 2009 GPU Technology Conference (see http://www.nvidia.com/object/research_summit_posters.html) published 91 research posters.

Library and tools developers are making GPU development more productive. GPU computing languages include CUDA C, CUDA C++, Portland Group (PGI) CUDA Fortran, DirectCompute, and OpenCL. GPU mathematics packages include MathWorks Matlab, Wolfram Mathematica, National Instruments Labview, SciComp SciFinance, and PyCUDA. NVIDIA developed the parallel Nsight GPU development environment, debugger, and analyzer integrated with Microsoft Visual Studio. GPU libraries include C++ productivity libraries, dense linear algebra, sparse linear algebra, FFTs, video and image processing, and data-parallel primitives. Computer system manufacturers are developing integrated CPU+GPU coprocessing systems in rackmount server and cluster configurations.

CUDA scalable parallel architecture

CUDA is a hardware and software coprocessing architecture for parallel computing that enables NVIDIA GPUs to execute programs written with C, C++, Fortran, OpenCL, DirectCompute, and other languages. Because most languages were designed for one sequential thread, CUDA preserves this model and extends it with a minimalist set of abstractions for expressing parallelism. This lets the programmer focus on the important issues of parallelism—how to design efficient parallel algorithms—using a familiar language.

By design, CUDA enables the development of highly scalable parallel programs that can run across tens of thousands of concurrent threads and hundreds of processor cores. A compiled CUDA program executes
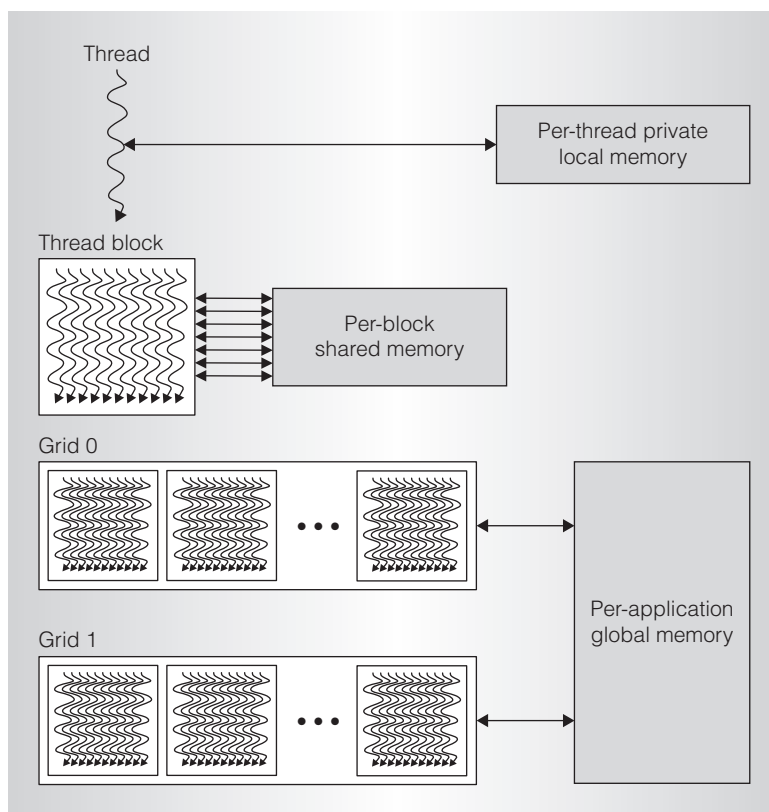

Figure 1. The CUDA hierarchy of threads, thread blocks, and grids of blocks, with corresponding memory spaces: per-thread private local, per-block shared, and per-application global memory spaces.

on any size GPU, automatically using more parallelism on GPUs with more processor cores and threads.

A CUDA program is organized into a host program, consisting of one or more sequential threads running on a host CPU, and one or more parallel kernels suitable for execution on a parallel computing GPU. A kernel executes a sequential program on a set of lightweight parallel threads. As Figure 1 shows, the programmer or compiler organizes these threads into a grid of thread blocks. The threads comprising a thread block can synchronize with each other via barriers and communicate via a high-speed, per-block shared memory.

Threads from different blocks in the same grid can coordinate via atomic operations in a global memory space shared by all threads. Sequentially dependent kernel grids can synchronize via global barriers and coordinate via global shared memory. CUDA requires that thread blocks be independent, which

....

HOT CHIPS

```
void saxpy(uint n, float a,
           float *x, float *y)
{
    uint i;
    for(i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

void serial_sample()
{
    // Call serial SAXPY function

    saxpy(n, 2.0, x, y);
}
(a)
```

```
__global__ void saxpy(uint n, float a,
                      float *x, float *y)
{
    uint i = blockIdx.x*blockDim.x + threadIdx.x;
    if(i < n)
        y[i] = a*x[i] + y[i];
}

void parallel_sample()
{
    // Launch parallel SAXPY kernel
    // using ⌈n/256⌉ blocks of 256 threads each
    saxpy<<<ceil(n/256),256>>>(n, 2.0, x, y);
}
(b)
```

Figure 2. Serial (a) and parallel CUDA (b) SAXPY kernels computing $\mathbf{y} = a\mathbf{x} + \mathbf{y}$.

provides scalability to GPUs with different numbers of processor cores and threads. Thread blocks implement coarse-grained scalable data parallelism, while the lightweight threads comprising each thread block provide fine-grained data parallelism. Thread blocks executing different kernels implement coarse-grained task parallelism. Threads executing different paths implement fine-grained thread-level parallelism. Details of the CUDA programming model are available in the programming guide.[2]

Figure 2 shows some basic features of parallel programming with CUDA. It contains sequential and parallel implementations of the SAXPY routine defined by the basic linear algebra subroutines (BLAS) library. Given scalar a and vectors $\mathbf{x}$ and $\mathbf{y}$ containing n floating-point numbers, it performs the update $\mathbf{y} = a\mathbf{x} + \mathbf{y}$. The serial implementation is a simple loop that computes one element of $\mathbf{y}$ per iteration. The parallel kernel executes each of these independent iterations in parallel, assigning a separate thread to compute each element of $\mathbf{y}$. The `__global__` modifier indicates that the procedure is a kernel entry point, and the extended function-call syntax `saxpy<<<B, T>>>(...)` launches the kernel `saxpy()` in parallel across B blocks of T threads each. Each thread determines which element it should process from its integer thread block index `blockIdx.x`, its thread index within its block `threadIdx.x`, and the total number of threads per block `blockDim.x`.

This example demonstrates a common parallelization pattern, where we can transform a serial loop with independent iterations to execute in parallel across many threads. In the CUDA paradigm, the programmer writes a scalar program—the parallel `saxpy()` kernel—that specifies the behavior of a single thread of the kernel. This lets CUDA leverage standard C language with only a few small additions, such as built-in thread and block index variables. The SAXPY kernel is also a simple example of data parallelism, where parallel threads each produce assigned result data elements.

GPU computing architecture

To address different market segments, GPU architectures scale the number of processor cores and memories to implement different products for each segment while using the same scalable architecture and software. NVIDIA's scalable GPU computing architecture varies the number of streaming multiprocessors to scale computing performance, and varies the number of DRAM memories to scale memory bandwidth and capacity.

Each multithreaded streaming multiprocessor provides sufficient threads, processor cores, and shared memory to execute one or more CUDA thread blocks. The parallel processor cores within a streaming multiprocessor execute instructions for parallel threads. Multiple streaming multiprocessors provide coarse-grained scalable data and task parallelism to execute multiple coarse-grained thread blocks (possibly running different kernels) in parallel. Multithreading and parallel-pipelined processor cores within each streaming multiprocessor implement
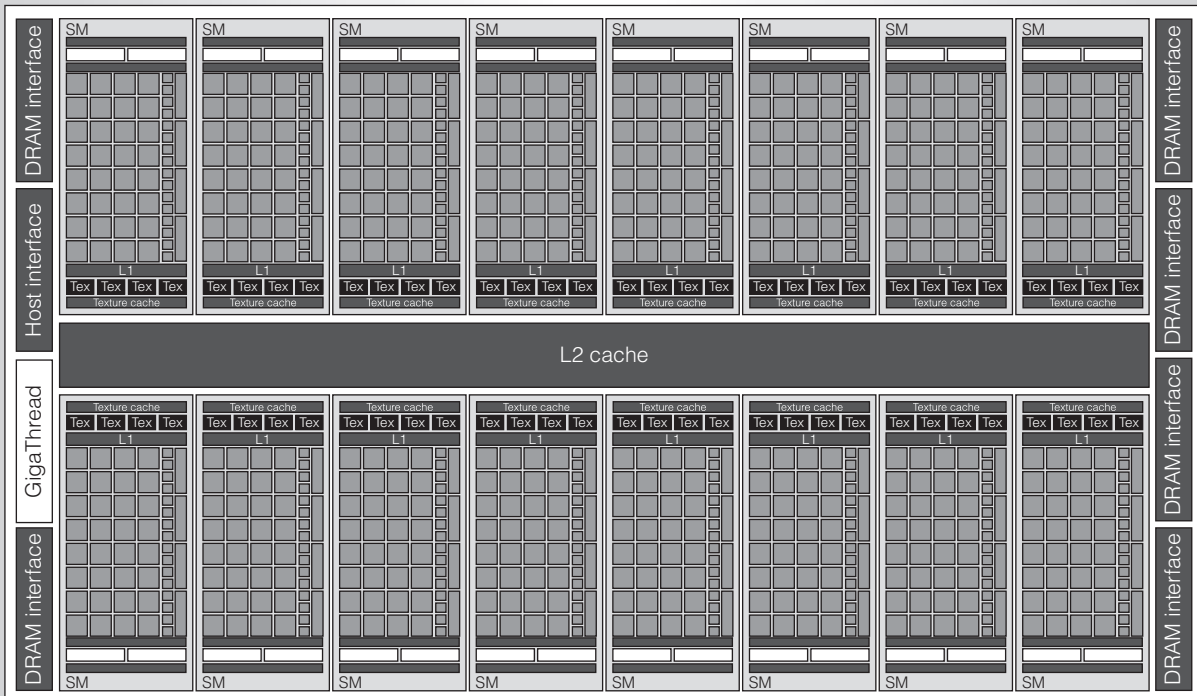
Figure 3. Fermi GPU computing architecture with 512 CUDA processor cores organized as 16 streaming multiprocessors (SMs) sharing a common second-level (L2) cache, six 64-bit DRAM interfaces, and a host interface with the host CPU, system memory, and I/O devices. Each streaming multiprocessor has 32 CUDA cores.

fine-grained data and thread-level parallelism to execute hundreds of fine-grained threads in parallel. Application programs using the CUDA model thus scale transparently to small and large GPUs with different numbers of streaming multiprocessors and processor cores.

Fermi computing architecture

To illustrate GPU computing architecture, Figure 3 shows the third-generation Fermi computing architecture configured with 16 streaming multiprocessors, each with 32 CUDA processor cores, for a total of 512 cores. The GigaThread work scheduler distributes CUDA thread blocks to streaming multiprocessors with available capacity, dynamically balancing the computing workload across the GPU, and running multiple kernel tasks in parallel when appropriate. The multithreaded streaming multiprocessors schedule and execute CUDA thread blocks and individual threads. Each streaming multiprocessor executes up to 1,536 concurrent threads to

help cover long latency loads from DRAM memory. As each thread block completes executing its kernel program and releases its streaming multiprocessor resources, the work scheduler assigns a new thread block to that streaming multiprocessor.

The PCIe host interface connects the GPU and its DRAM memory with the host CPU and system memory. The CPU+GPU coprocessing and data transfers use the bidirectional PCIe interface. The streaming multiprocessor threads access system memory via the PCIe interface, and CPU threads access GPU DRAM memory via PCIe.

The GPU architecture balances its parallel computing power with parallel DRAM memory controllers designed for high memory bandwidth. The Fermi GPU in Figure 3 has six high-speed GDDR5 DRAM interfaces, each 64 bits wide. Its 40-bit addresses handle up to 1 Tbyte of address space for GPU DRAM and CPU system memory for large-scale computing.

....

HOT CHIPS

Cached memory hierarchy

Fermi introduces a parallel cached memory hierarchy for load, store, and atomic memory accesses by general applications. Each streaming multiprocessor has a first-level (L1) data cache, and the streaming multiprocessors share a common 768-Kbyte unified second-level (L2) cache. The L2 cache connects with six 64-bit DRAM interfaces and the PCIe interface, which connects with the host CPU, system memory, and PCIe devices. It caches DRAM memory locations and system memory pages accessed via the PCIe interface. The unified L2 cache services load, store, atomic, and texture instruction requests from the streaming multiprocessors and requests from their L1 caches, and fills the streaming multiprocessor instruction caches and uniform data caches.

Fermi implements a 40-bit physical address space that accesses GPU DRAM, CPU system memory, and PCIe device addresses. It provides a 40-bit virtual address space to each application context and maps it to the physical address space with translation lookaside buffers and page tables.

ECC memory

Fermi introduces ECC memory protection to enhance data integrity in large-scale GPU computing systems. Fermi ECC corrects single-bit errors and detects double-bit errors in the DRAM memory, GPU L2 cache, L1 caches, and streaming multiprocessor registers. The ECC lets us integrate thousands of GPUs in a system while maintaining a high mean time between failures (MTBF) for high-performance computing and supercomputing systems.

Streaming multiprocessor

The Fermi streaming multiprocessor introduces several architectural features that deliver higher performance, improve its programmability, and broaden its applicability. As Figure 4 shows, the streaming multiprocessor execution units include 32 CUDA processor cores, 16 load/store units, and four special function units (SFUs). It has a 64-Kbyte configurable shared memory/L1 cache, 128-Kbyte register file, instruction cache, and two multithreaded warp schedulers and instruction dispatch units.

Efficient multithreading

The streaming multiprocessor implements zero-overhead multithreading and thread scheduling for up to 1,536 concurrent threads. To efficiently manage and execute this many individual threads, the multiprocessor employs the single-instruction multiple-thread (SIMT) architecture introduced in the first unified computing GPU.[7,8] The SIMT instruction logic creates, manages, schedules, and executes concurrent threads in groups of 32 parallel threads called *warps*. A CUDA thread block comprises one or more warps. Each Fermi streaming multiprocessor has two warp schedulers and two dispatch units that each select a warp and issue an instruction from the warp to 16 CUDA cores, 16 load/store units, or four SFUs. Because warps execute independently, the streaming multiprocessor can issue two warp instructions to appropriate sets of CUDA cores, load/store units, and SFUs.

To support C, C++, and standard single-thread programming languages, each streaming multiprocessor thread is independent, having its own private registers, condition codes and predicates, private per-thread memory and stack frame, instruction address, and thread execution state. The SIMT instructions control the execution of an individual thread, including arithmetic, memory access, and branching and control flow instructions. For efficiency, the SIMT multiprocessor issues an instruction to a warp of 32 independent parallel threads.

The streaming multiprocessor realizes full efficiency and performance when all threads of a warp take the same execution path. If threads of a warp diverge at a data-dependent conditional branch, execution serializes for each branch path taken, and when all paths complete, the threads converge to the same execution path. The Fermi streaming multiprocessor extends the flexibility of the SIMT independent thread control flow with indirect branch and function-call instructions, and trap handling for exceptions and debuggers.

Thread instructions

Parallel thread execution (PTX) instructions describe the execution of a single thread in a parallel CUDA program. The PTX
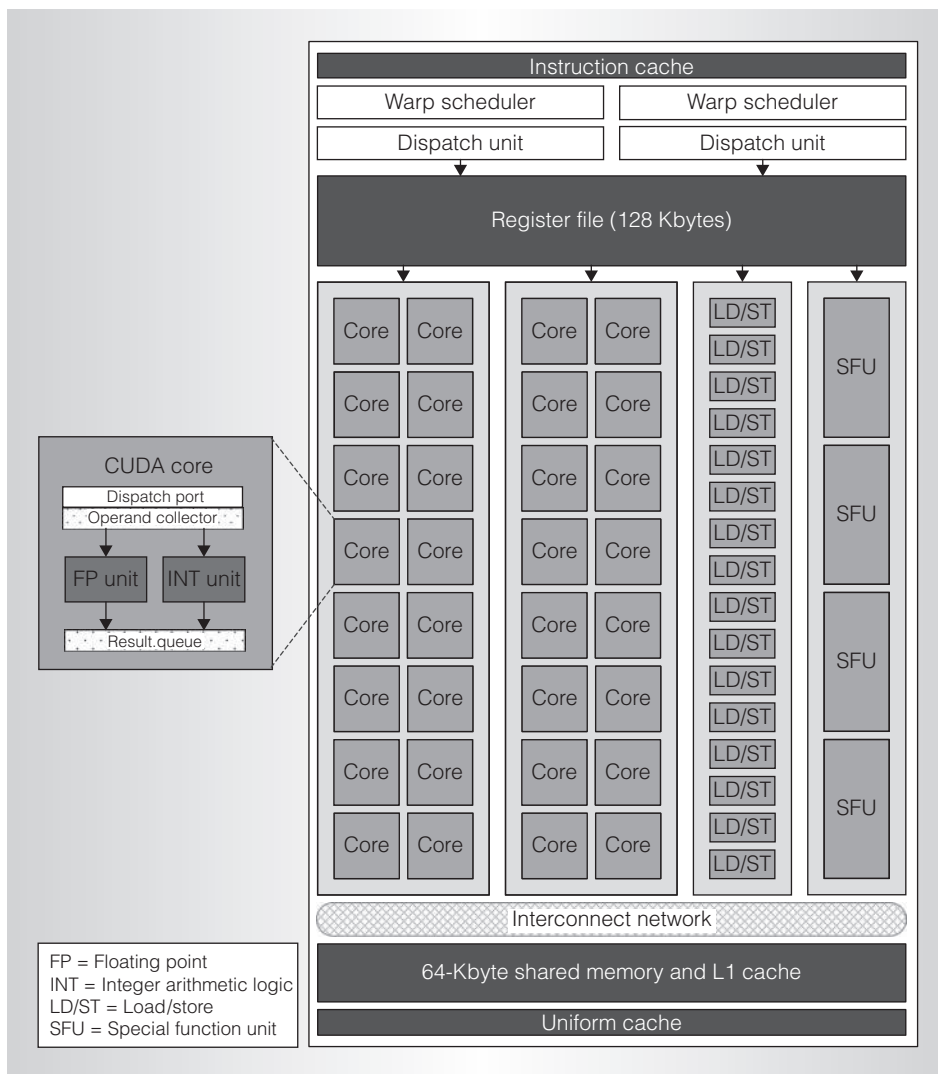
Figure 4. The Fermi streaming multiprocessor has 32 CUDA processor cores, 16 load/store units, four special function units, a 64-Kbyte configurable shared memory/L1 cache, 128-Kbyte register file, instruction cache, and two multithreaded warp schedulers and instruction dispatch units.

instructions focus on scalar (rather than vector) operations to match standard scalar programming languages. Fermi implements the PTX 2.0 instruction set architecture (ISA), which targets C, C++, Fortran, OpenCL, and DirectCompute programs. Instructions include

- 32-bit and 64-bit integer, addressing, and floating-point arithmetic;
- load, store, and atomic memory access;
- texture and multidimensional surface access;

- individual thread flow control with predicated instructions, branching, function calls, and indirect function calls for C++ virtual functions; and
- parallel barrier synchronization.

CUDA cores

Each pipelined CUDA core executes a scalar floating point or integer instruction per clock for a thread. With 32 cores, the streaming multiprocessor can execute up to 32 arithmetic thread instructions per clock.

....

H<small>OT</small> C<small>HIPS</small>

The integer unit implements 32-bit precision for scalar integer operations, including 32-bit multiply and multiply-add operations, and efficiently supports 64-bit integer operations. The Fermi integer unit adds bit-field insert and extract, bit reverse, and population count.

IEEE 754-2008 floating-point arithmetic

The Fermi CUDA core floating-point unit implements the IEEE 754-2008 floating-point arithmetic standard for 32-bit single-precision and 64-bit double-precision results, including fused multiply-add (FMA) instructions. FMA computes $D = A * B + C$ with no loss of precision by retaining full precision in the intermediate product and addition, then rounding the final sum to form the result. Using FMA enables fast division and square-root operations with exactly rounded results.

Fermi raises the throughput of 64-bit double-precision operations to half that of single-precision operations, a dramatic improvement over the T10 GPU. This performance level enables broader deployment of GPUs in high-performance computing. The floating-point instructions handle subnormal numbers at full speed in hardware, allowing small values to retain partial precision rather than flushing them to zero or calculating subnormal values in multicycle software exception handlers as most CPUs do.

The SFUs execute 32-bit floating-point instructions for fast approximations of reciprocal, reciprocal square root, sin, cos, exp, and log functions. The approximations are precise to better than 22 mantissa bits.

Unified memory addressing and access

The streaming multiprocessor load/store units execute load, store, and atomic memory access instructions. A warp of 32 active threads presents 32 individual byte addresses, and the instruction accesses each memory address. The load/store units coalesce 32 individual thread accesses into a minimal number of memory block accesses.

Fermi implements a unified thread address space that accesses the three separate parallel memory spaces of Figure 1: per-thread local, per-block shared, and global

memory spaces. A unified load/store instruction can access any of the three memory spaces, steering the access to the correct memory, which enables general C and C++ pointer access anywhere. Fermi provides a terabyte 40-bit unified byte address space, and the load/store ISA supports 64-bit byte addressing for future growth. The ISA also provides 32-bit addressing instructions when the program can limit its accesses to the lower 4 Gbytes of address space.

Configurable shared memory and L1 cache

On-chip shared memory provides low-latency, high-bandwidth access to data shared by cooperating threads in the same CUDA thread block. Fast shared memory significantly boosts the performance of many applications having predictable regular addressing patterns, while reducing DRAM memory traffic.

Fermi introduces a configurable-capacity L1 cache to aid unpredictable or irregular memory accesses, along with a configurable-capacity shared memory. Each streaming multiprocessor has 64 Kbytes of on-chip memory, configurable as 48 Kbytes of shared memory and 16 Kbytes of L1 cache, or as 16 Kbytes of shared memory and 48 Kbytes of L1 cache.

CPU+GPU coprocessing

Heterogeneous CPU+GPU coprocessing systems evolved because the CPU and GPU have complementary attributes that allow applications to perform best using both types of processors. CUDA programs are coprocessing programs—serial portions execute on the CPU, while parallel portions execute on the GPU. Coprocessing optimizes total application performance.

With coprocessing, we use the right core for the right job. We use a CPU core (optimized for low latency on a single thread) for a code's serial portions, and we use GPU cores (optimized for aggregate throughput on a code's parallel portions) for parallel portions of code. This approach gives more performance per unit area or power than either CPU or GPU cores alone.

The comparison in Table 2 illustrates the advantage of CPU+GPU coprocessing using

Table 2. CPU+GPU coprocessing execution time, assuming that a CPU core is 5× faster and 50× the area of a GPU core.

Program type	Configuration	Processing time for 1 CPU core	Processing time for 500 GPU cores	Processing time for 10 CPU cores	Processing time for 1 CPU core + 450 GPU cores
	Area	50	500	500	500
Parallel-intensive program	0.5% serial code	1.0	5.0	1.0	1.00
	99.5% parallelizable code	199.0	0.4	19.9	0.44
	Total	200.0	5.4	20.9	1.44
Mostly sequential program	75% serial code	150.0	750.0	150.0	150.0
	25% parallelizable code	50.0	0.1	5.0	0.11
	Total	200.0	750.1	155.0	150.11

Amdahl's law. The table compares the performance of four configurations:

- a system containing one latency-optimized (CPU) core,
- a system containing 500 throughput-optimized (GPU) cores,
- a system containing 10 CPU cores, and
- a coprocessing system that contains a single CPU core and 450 GPU cores.

Table 2 assumes that a CPU core is 5× faster and 50× the area of a GPU core—numbers consistent with contemporary CPUs and GPUs. The coprocessing system devotes 10 percent of its area to the single CPU core and 90 percent of its area to the 450 GPU cores.

Table 2 compares the four configurations on both a parallel-intensive program (with a serial fraction of only 0.5 percent) and on a mostly sequential program (with a serial fraction of 75 percent). The coprocessing architecture is the fastest on both programs. On the parallel-intensive program, the coprocessing architecture is slightly slower on the parallel portion than the pure GPU configuration (0.44 seconds versus 0.40 seconds) but more than makes up for this by running the tiny serial portion 5× faster (1 second versus 5 seconds). The heterogeneous architecture has an advantage over the pure throughput-optimized configuration here because serial performance is important even for mostly parallel codes.

On the mostly sequential program, the coprocessing configuration matches the performance of the multi-CPU configuration on the code's serial portion but runs the parallel portion 45× as fast, giving a slightly faster overall performance. Even on mostly sequential codes, it's more efficient to run the code's parallel portion on a throughput-optimized architecture.

The coprocessing architecture provides the best performance across a wide range of the serial fraction because it uses the right core for each task. By using a latency-optimized CPU to run the code's serial fraction, it gives the best possible performance on the serial fraction—which is important even for mostly parallel codes. By using throughput-optimized cores to run the code's parallel portion, it gives near-optimal performance on the parallel fraction as well—which becomes increasingly important as codes become more parallel. It's wasteful to use large, inefficient latency-optimized cores to run parallel code segments.

Application performance

Many applications consist of a mixture of fundamentally serial control logic and inherently parallel computations. Furthermore, these parallel computations are frequently data-parallel in nature. This directly matches the CUDA coprocessing programming model, namely a sequential control thread

....

HOT CHIPS

Table 3. Representative CUDA application coprocessing speedups.

Application	Field	Speedup
Two-electron repulsion integral[12]	Quantum chemistry	130×
Gromacs[13]	Molecular dynamics	137×
Lattice Boltzmann[14]	3D computational fluid dynamics (CFD)	100×
Euler solver[15]	3D CFD	16×
Lattice quantum chromodynamics[16]	Quantum physics	10×
Multigrid finite element method and partial differential equation solver[17]	Finite element analysis	27×
N-body physics[18]	Astrophysics	100×
Protein multiple sequence alignment[19]	Bioinformatics	36×
Image contour detection[20]	Computer vision	130×
Portable media converter*	Consumer video	20×
Large vocabulary speech recognition[21]	Human interaction	9×
Iterative image reconstruction[22]	Computed tomography	130×
Matlab accelerator**	Computational modeling	100×

* Elemental Technologies, Badaboom media converter, 2009; http://badaboomit.com.
** Accelereyes, Jacket GPU engine for Matlab, 2009; http://www.accelereyes.com.

capable of launching a series of parallel kernels. The use of parallel kernels launched from a sequential program also makes it relatively easy to parallelize an application's individual components rather than rewrite the entire application.

Many applications—both academic research and industrial products—have been accelerated using CUDA to achieve significant parallel speedups.[10] Such applications fall into a variety of problem domains, including quantum chemistry, molecular dynamics, computational fluid dynamics, quantum physics, finite element analysis, astrophysics, bioinformatics, computer vision, video transcoding, speech recognition, computed tomography, and computational modeling.

Table 3 lists some representative applications along with the runtime speedups obtained for the whole application using CPU+GPU coprocessing over CPU alone, as measured by application developers.[12-22] The speedups using GeForce 8800, Tesla T8, GeForce GTX 280, Tesla T10, and GeForce GTX 285 range from 9× to more than 130×, with the higher speedups reflecting applications where more of the work ran in parallel on the GPU. The lower speedups—while still quite attractive—represent applications that are limited by the code's CPU portion, coprocessing overhead, or by divergence in the code's GPU fraction.

The speedups achieved on this diverse set of applications validate the programmability of GPUs—in addition to their performance. The breadth of applications that have been ported to CUDA (more than 1,000 on the CUDA Zone) demonstrates the range of the programming model and architecture. Applications with dense matrices, sparse matrices, and arbitrary pointer structures have all been successfully implemented in CUDA with impressive speedups. Similarly, applications with diverse control structures and significant data-dependent control, such as ray tracing, have achieved good performance in CUDA.

Many real-world applications (such as interactive ray tracing) are composed of many different algorithms, each with varying degrees of parallelism. OptiX, our interactive ray-tracing software developer's kit built in the CUDA architecture, provides a mechanism to control and schedule a wide variety of tasks on both the CPU and GPU. Some tasks are primarily serial and execute on the CPU, such as compilation, data structure management, and coordination with the operating system and user interaction.

Other tasks, such as building an acceleration structure or updating animations, may run either on the CPU or the GPU depending on the choice of algorithms and the performance required. However, the real power of OptiX is the manner in which it manages parallel operations that are either regular or highly irregular in a fine-grained fashion, and this is executed entirely on the GPU.

To accomplish this, OptiX decomposes the algorithm into a series of states. Some of these states consist of uniform operations such as generating rays according to a camera model, or applying a tone-mapping operator. Other states consist of highly irregular computation, such as traversing an acceleration structure to locate candidate geometry, or intersecting a ray with that candidate geometry. These operations are highly irregular because they require an indefinite number of operations and can vary in cost by an order of magnitude or more between rays. A user-provided program that executes within an abstract ray-tracing machine controls the details of each operation.

To execute this state machine, each warp of parallel threads on the GPU selects a handful of rays that have matching state. Each warp executes in SIMT fashion, causing each ray to transition to a new state (which might not be the same for all rays). This process repeats until all rays complete their operations. OptiX can control both the prioritization of the states and the scope of the search for rays to target trade-offs in different GPU generations. Even if rays temporarily diverge to different states, they'll return to a small number of common states—usually traversal and intersection. This state-machine model gives the GPU an opportunity to reunite diverged threads with other threads executing the same code, which wouldn't occur in a straightforward SIMT execution model.

The net result is that CPU+GPU coprocessing enables fast, interactive ray tracing of complex scenes while you watch, which is an application that researchers previously considered too irregular for a GPU.

GPU computing is at the tipping point. Single-threaded processor performance is no longer scaling at historic rates.

Thus, we must use parallelism for the increased performance required to deliver more value to users. A GPU that's optimized for throughput delivers parallel performance much more efficiently than a CPU that's optimized for latency. A heterogeneous coprocessing architecture that combines a single latency-optimized core (a CPU) with many throughput-optimized cores (a GPU) performs better than either alternative alone. This is because it uses the right processor for the right job— the CPU for serial sections and critical paths and the GPU for the parallel sections.

Because they efficiently execute programs across a range of parallelism, heterogeneous CPU+GPU architectures are becoming pervasive. In high-performance computing, technical computing, and consumer media processing, CPU+GPU coprocessing has become the architecture of choice. We expect the rate of adoption of GPU computing in these beachhead areas to accelerate, and for GPU computing to spread to broader application areas. In addition to accelerating existing applications, we expect GPU computing to enable new classes of applications, such as building compelling virtual worlds and performing universal speech translation.

As the GPU computing era progresses, GPU performance will continue to scale at Moore's law rates—about 50 percent per year—giving an order of magnitude increase in performance in five to six years. At the same time, GPU architecture will evolve to further increase the span of applications that it can efficiently address. GPU cores will not become CPUs—they will continue to be optimized for throughput, rather than latency. However, they will evolve to become more agile and better able to handle arbitrary control and data access patterns. GPUs and their programming systems will also evolve to make GPUs even easier to program—further accelerating their rate of adoption. MICRO

Acknowledgments

....

References

1. J. Nickolls et al., ''Scalable Parallel Programming with CUDA,'' *ACM Queue,* vol. 6, no. 2, 2008, pp. 40-53.

2. NVIDIA, *NVIDIA CUDA Programming Guide,* 2009; http://developer.download.nvidia. com/compute/cuda/2_3/toolkit/docs/ NVIDIA_CUDA_Programming_Guide_2.3.pdf.

3. C. Boyd, ''DirectCompute: Capturing the Teraflop,'' *Microsoft Personal Developers Conf.,* 2009; http://ecn.channel9.msdn. com/o9/pdc09/ppt/CL03.pptx.

4. Khronos, *The OpenCL Specification,* 2009; http://www.khronos.org/OpenCL.

5. J. Montrym and H. Moreton, ''The GeForce 6800,'' *IEEE Micro,* vol. 25, no. 2, 2005, pp. 41-51.

6. W.R. Mark et al., ''Cg: A System for Programming Graphics Hardware in a C-like Language,'' *Proc. Special Interest Group on Computer Graphics* (Siggraph), ACM Press, 2003, pp. 896-907.

7. E. Lindholm et al., ''NVIDIA Tesla: A Unified Graphics and Computing Architecture,'' *IEEE Micro,* vol. 28, no. 2, 2008, pp. 39-55.

8. J. Nickolls and D. Kirk, ''Graphics and Computing GPUs,'' *Computer Organization and Design: The Hardware/Software Interface,* D.A. Patterson and J.L. Hennessy, 4th ed., Morgan Kaufmann, 2009, pp. A2-A77.

9. NVIDIA, ''Fermi: NVIDIA's Next Generation CUDA Compute Architecture,'' 2009; http:// www.nvidia.com/content/PDF/fermi_white_ papers/NVIDIA_Fermi_Compute_Architecture_ Whitepaper.pdf.

10. M. Garland et al., ''Parallel Computing Experiences with CUDA,'' *IEEE Micro,* vol. 28, no. 4, 2008, pp. 13-27.

11. D.B. Kirk and W.W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach,* Morgan Kaufmann, 2010.

12. I.S. Ufimtsev and T.J. Martinez, ''Quantum Chemistry on Graphical Processing Units. 1. Strategies for Two-Electron Integral Evaluation,'' *J. Chemical Theory and Computation,* vol. 4, no. 2, 2008, pp. 222-231.

13. M.S. Friedrichs et al., ''Accelerating Molecular Dynamic Simulation on Graphics Processing Units,'' *J. Computational Chemistry,* vol. 30, no. 6, 2009, pp. 864-872.

14. J. Tölke and M. Krafczyk, ''TeraFLOP Computing on a Desktop PC with GPUs for 3D CFD,'' *Int'l J. Computational Fluid Dynamics,* vol. 22, no. 7, 2008, pp. 443-456.

15. T. Brandvik and G. Pullan, ''Acceleration of a 3D Euler Solver Using Commodity Graphics Hardware,'' *Proc. 46th Am. Inst. of Aeronautics and Astronautics* (AIAA) *Aerospace Sciences Meeting,* AIAA, 2008; http:// www.aiaa.org/agenda.cfm?lumeetingid= 1065&dateget=08-Jan-08#session8907.

16. M.A. Clark et al., ''Solving Lattice QCD Systems of Equations Using Mixed Precision Solvers on GPUs,'' *Computer Physics Comm.,* 2009; http://arxiv.org/abs/0911. 3191v2.

17. D. Göddeke and R. Strzodka, ''Performance and Accuracy of Hardware-Oriented Native-, Emulated-, and Mixed-Precision Solvers in FEM Simulations (Part 2: Double Precision GPUs),'' *Ergebnisberichte des Instituts für Angewandte Mathematik* [Reports on Findings of the Inst. for Applied Mathematics], Dortmund Univ. of Technology, no. 370, 2008; http://www. mathematik.uni-dortmund.de/~goeddeke/ pubs/GTX280_mixedprecision.pdf.

18. R.G. Belleman, J. Bedorf, and S.P. Zwart, ''High Performance Direct Gravitational N-body Simulations on Graphics Processing Units II: An Implementation in CUDA,'' *New Astronomy,* vol. 13, no. 2, 2008, pp. 103-112.

19. Y. Liu, B. Schmidt, and D.L. Maskell, ''MSA-CUDA: Multiple Sequence Alignment on Graphics Processing Units with CUDA,'' *Proc. 20th IEEE Int'l Conf. Application-Specific Systems, Architectures and Processors,* IEEE CS Press, 2009, pp. 121-128.

20. B. Catanzaro et al., ''Efficient, High-Quality Image Contour Detection,'' *Proc. IEEE Int'l Conf. Computer Vision,* IEEE CS Press, 2009; http://www.cs.berkeley.edu/~catanzar/ Damascene/iccv2009.pdf.

21. J. Chong et al., ''Data-Parallel Large Vocabulary Continuous Speech Recognition on Graphics Processors,'' tech. report UCB/EECS-2008-69, Univ. of California at Berkeley, 2008; http://www.eecs.berkeley. edu/Pubs/TechRpts/2008/EECS-2008-69.pdf.

22. Y. Pan et al., ''Feasibility of GPU-Assisted Iterative Image Reconstruction for Mobile C-Arm CT,'' *Proc. Int'l Soc. for Photonics and Optics* (SPIE), vol. 7258, SPIE 2009; http://www.sci.utah.edu/~ypan/Pan_ SPIE2009.pdf.

23. J.H. Huang, ''2009: The GPU Computing Tipping Point,'' *Proc. IEEE Hot Chips 21,* 2009; http://www.hotchips.org/archives/hc21.

John Nickolls is the director of architecture at NVIDIA for GPU computing. His technical interests include developing parallel processing products, languages, and architectures. Nickolls has a PhD in electrical engineering from Stanford University.
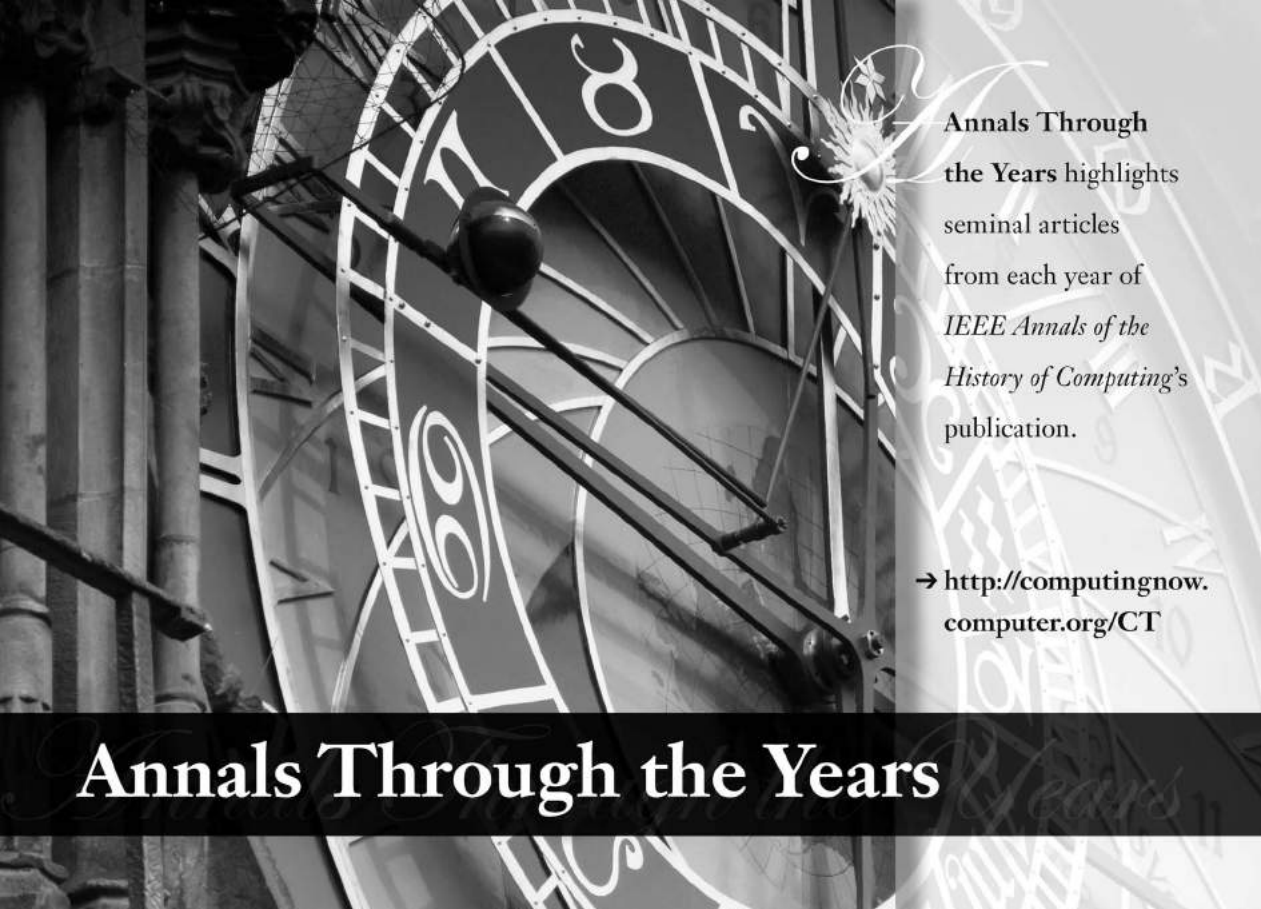
William J. Dally is the chief scientist and senior vice president of research at NVIDIA and the Willard R. and Inez Kerr Bell Professor of Engineering at Stanford University. His technical interests include parallel computer architecture, parallel programming systems, and interconnection networks. He's a member of the National Academy of Engineering and a fellow of IEEE, ACM, and the American Academy of Arts and Sciences. Dally has a PhD in computer science from the California Institute of Technology.

Direct questions and comments about this article to John Nickolls, NVIDIA, 2701 San Tomas Expressway, Santa Clara, CA 95050; jnickolls@nvidia.com.

cn *Selected CS articles and columns are also available for free at http://ComputingNow. computer.org.*

Annals Through the Years highlights seminal articles from each year of *IEEE Annals of the History of Computing*'s publication.

→ http://computingnow. computer.org/CT

Annals Through the Years