# GPU computing in medical physics: A review

Guillem Pratx[a] and Lei Xing
*Department of Radiation Oncology, Stanford University School of Medicine, 875 Blake Wilbur Drive, Stanford, California 94305*

(Received 29 November 2010; revised 21 March 2011; accepted for publication 25 March 2011; published 9 May 2011)

The graphics processing unit (GPU) has emerged as a competitive platform for computing massively parallel problems. Many computing applications in medical physics can be formulated as data-parallel tasks that exploit the capabilities of the GPU for reducing processing times. The authors review the basic principles of GPU computing as well as the main performance optimization techniques, and survey existing applications in three areas of medical physics, namely image reconstruction, dose calculation and treatment plan optimization, and image processing.
[DOI: 10.1118/1.3578605]

Key words: graphics processing units, high-performance computing, image segmentation, dose calculation, image processing

## I. INTRODUCTION

Parallel processing has become the standard for high-performance computing. Over the last thirty years, general-purpose, single-core processors have enjoyed a doubling of their performance every 18 months, a feat made possible by superscalar pipelining, increasing instruction-level parallelism and higher clock frequency. Recently, however, the progress of single-core processor performance has slowed due to excessive power dissipation at GHz clock rates and diminishing returns in instruction-level parallelism. Hence, application developers—in particular in the medical physics community—can no longer count on Moore's law to make complex algorithms computationally feasible. Instead, they are increasingly shifting their algorithms to parallel computing architectures for practical processing times.

With the increased sophistication of medical imaging and treatment machines, the amount of data processed in medical physics is exploding; processing time is now limiting the deployment of advanced technologies. This trend has been driven by many factors, such as the shift from 3-D to 4-D in imaging and treatment planning, the improvement of spatial resolution in medical imaging, the shift to cone-beam geometries in x-ray CT, the increasing sophistication of MRI pulse sequences, and the growing complexity of treatment planning algorithms. Yet typical medical physics datasets comprise a large number of similar elements, such as voxels in tomographic imaging, beamlets in intensity-modulated radiation therapy (IMRT) optimization, k-space samples in MRI, projective measurements in x-ray CT, and coincidence events in PET. The processing of such datasets can often be accelerated by distributing the computation over many parallel threads.

Originally designed for accelerating the production of computer graphics, the graphics processing unit (GPU) has emerged as a versatile platform for running massively parallel computation. Graphics hardware presents clear advantages for processing the type of datasets encountered in medical physics: high memory bandwidth, high computation throughput, support for floating-point arithmetic, the lowest price per unit of computation, and a programming interface accessible to the nonexpert. These features have raised tremendous enthusiasm in many disciplines, such as linear algebra, differential equations, databases, raytracing, data mining, computational biophysics, molecular dynamics, fluid dynamics, seismic imaging, game physics, and dynamic programming.[1–4]

In medical physics, the ability to perform general-purpose computation on the GPU was first demonstrated in 1994 when a research group at SGI implemented image reconstruction on an Onyx workstation using the RealityEngine2.[5] Despite this pioneering work, it took almost 10 yr for GPU computing to become mainstream as a topic of research (Fig. 1). There were several reasons for this slow start. Throughout the 1990s, researchers were blessed with the doubling of single-core processor performance every 18 months. As a result, a single-core processor in 2004 could perform image reconstruction 100 times faster than in 1994, and as fast as SGIs 1994 graphics-hardware implementation.[5] However, the performance of recent single-core processors suggests that the doubling period might now be 5 yr.[6] As a result, vendors have switched to multicore architectures to keep improving the performance of their CPUs, a shift that has given a strong incentive for researchers to consider parallelizing their computations.

Around the same time, the programmable GPU was introduced. Unlike previous graphics processors, which were limited to running a fixed-function pipeline with 8-bit integer arithmetic, these new GPUs could run custom programs (called *shaders*) in parallel, with floating-point precision. The shift away from single-core processors and the increasing programmability of the GPU created favorable conditions for the emergence of GPU computing.

GPUs now offer a compelling alternative to computer clusters for running large, distributed applications. With the introduction of compute-oriented GPU interfaces, shared

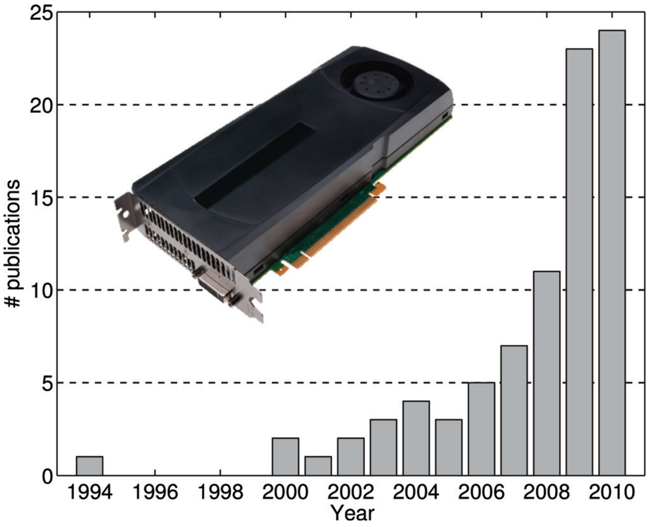

FIG. 1. Number of publications relating to the use of GPUs in medical physics, per year. Data were obtained by searching PubMed using the terms "GPU," "graphics processing unit," and "graphics hardware" and excluding irrelevant citations.

memory, and support for double-precision arithmetic, the range of computational applications that can run on the GPU has vastly increased. By off-loading the data-parallel part of the computation onto GPUs, the number of physical computers within a computer cluster can be greatly reduced. Besides reducing cost, smaller computer clusters also require less maintenance, space, power, and cooling. These are important factors to consider in medical physics given that the computing resources are typically located on-site, inside the hospital.

## II. OVERVIEW OF GPU COMPUTING

### II.A. Evolution of the GPU

Over the years, the GPU has evolved from a highly specialized pixel processor to a versatile and highly programmable architecture that can perform a wide range of data-parallel operations. The hardware of early 3-D acceleration cards (such as the 3Dfx Voodoo) was devoted to processing pixel and texture data. These cards offered no parallel processing capabilities, but freed the CPU from the computationally demanding task of filling polygon with texture and color. A few years later, the task of transforming the geometry was also moved from the CPU to the GPU, one of the first steps toward the modern graphics pipeline.

Because the processing of vertices and pixels is inherently parallel, the number of dedicated processing units increased rapidly, allowing commodity PCs to render ever more complex 3-D scenes in tens of milliseconds. Since 1997, the number of compute cores in GPU processors has doubled roughly every 1.4 yr (Fig. 2). Over the same period, GPU cores have become increasingly sophisticated and versatile, enriching their instruction set with a wide variety of control-flow mechanisms, support for double-precision floating-point arithmetic, built-in mathematical functions, a shared-memory model for interthread communications, atomic operations, and so forth. In order to sustain the increased computation throughput, the GPU memory bandwidth has doubled every 1.7 yr, and recent GPUs can achieve a peak memory bandwidth of 408 GB/s (Fig. 2).

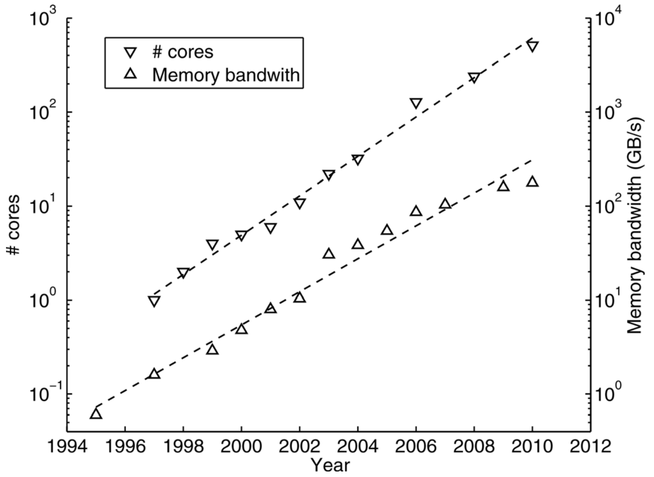

FIG. 2. Number of computing cores (▽) and memory bandwidth (Δ) for high-end NVIDIA GPUs as a function of year (data from vendor specifications).

With more computing cores, the peak performance of GPUs, measured in billion floating-point operations per second (GFLOPS), has been steadily increasing (Fig. 3). In addition, the performance gap between GPU and CPU has been widening, due to a performance doubling rate of 1.5 yr for CPUs versus 1 yr for GPUs (Fig. 3).

The faster progress of the GPUs performance can be attributed to the highly scalable nature of its architecture. For multicore/multi-CPU systems, the number of threads physically

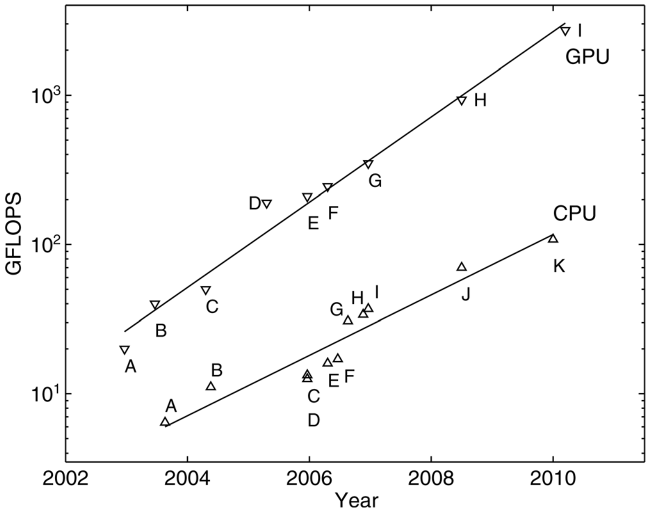

FIG. 3. Computing performance, measured in billion single-precision floating-point operation per second (GFLOPS), for CPUs (Δ) and GPUs (▽). GPUs: (A) NVIDIA GeForce FX 5800, (B) FX 5950 Ultra, (C) 6800 Ultra, (D) 7800 GTX, (E) Quadro FX 4500, (F) GeForce 7900 GTX, (G) 8800 GTX, (H) Tesla C1060, and (I) AMD Radeon HD 5870. CPUs: (A) Athlon 64 3200+, (B) Pentium IV 560, (C) Pentium D 960, (D) 950, (E) Athlon 64 X2 5000+, (F) Core 2 Duo E6700, (G) Core 2 Quad Q6600, (H) Athlon 64 FX-74, (I) Core 2 Quad QX6700, (J) Intel Core i7 965 XE, and (K) Core i7-980X Extreme (data from vendors).

residing in the hardware can be no greater than twice the number of physical cores (with hyperthreading). As a result, advanced PCs can run at most 100 threads simultaneously. In contrast, current GPU hardware can host up to 30 000 concurrent threads. Whereas switching between CPU threads is costly because the operating system physically loads the thread execution context from the RAM, switching between GPU threads does not incur any overhead as the threads are residing on the GPU for their entire lifetime. A further difference is that the GPU processing pipeline is for the most part based on a feed-forward, single-instruction multiple-data (SIMD) architecture, which removes the need for advanced data controls. In comparison, multicore multi-CPU pipelines require complex control logic to avoid data hazards.

## II.B. The graphics pipeline

In the early days of GPU computing, the GPU could only be programmed through a graphics rendering interface. In these pioneering implementations, computation was reformulated as a rendering task and programmed through the graphics pipeline. While new compute-specific interfaces have made these techniques obsolete, a basic understanding of the graphics pipeline is still useful for writing efficient GPU code.

Graphics applications (such as video games) use the GPU to perform the calculations necessary to render complex 3-D scenes in tens of milliseconds. Typically, 3-D scenes are represented by triangular meshes filled with color or *textures*. Textures are 2-D color images, stored in GPU memory, designed to increase the perceived complexity of a 3-D scene. The graphics pipeline decomposes graphics computation into a sequence of stages that exposes both *task parallelism* and *data parallelism* (Fig. 4). Task parallelism is achieved when different tasks are performed simultaneously at different stages of the pipeline. Data parallelism is achieved when the same task is performed simultaneously on different data. The computational efficiency is further improved by implementing each stage of the graphics pipeline using custom rather than general-purpose hardware.

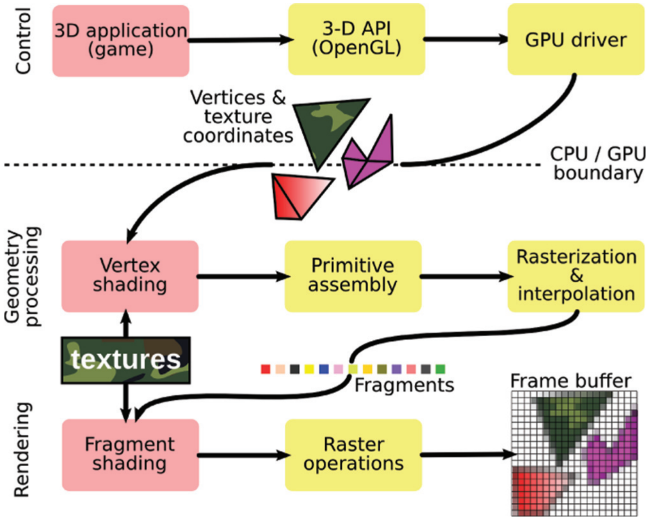

FIG. 4. The graphics pipeline. The boxes shaded in light red correspond to stages of the pipeline that can be programmed by the user.

Within a graphics application, the GPU operates as a stream processor. In the graphics pipeline, a stream of vertices (representing triangular meshes) is read from the host's main memory and processed in parallel by *vertex shaders* (Fig. 4). Typical vertex processing tasks include projecting the geometry onto the image plane of the virtual camera, computing the surface normal vectors and generating 2-D texture coordinates for each vertex.

After having been processed, vertices are assembled into triangles to undergo *rasterization* (Fig. 4). Rasterization, implemented in hardware, determines which pixels are covered by a triangle and, for each of these pixels, generates a *fragment*. Fragments are small data structures that contain all the information needed to update a pixel in the framebuffer, including pixel coordinates, depth, color, and texture coordinates. Fragments inherit their properties from the vertices of the triangles from which they originate, wherein properties are bilinearly interpolated within the triangle area by dedicated GPU hardware.

The stream of fragments is processed in parallel by *fragment shaders* (Fig. 4). In a typical graphics application, this programmable stage of the pipeline uses the fragment data to compute the final color and transparency of the pixel. Fragment shaders can fetch textures, calculate lighting effects, determine occlusions, and define transparency. After having been processed, the stream of fragments is written to the framebuffer according to predefined raster operations, such as additive blending.

All the stages of the graphics pipeline are implemented on the GPU using dedicated hardware. In a unified shader model, the system allocates computing cores to vertex and fragment shading based on the relative intensity of each task. The early GPU computing work focused on exploiting the high computational throughput of the fragment shading stage, which is easily accessible. For instance, a popular technique for processing 2-D arrays of data consisted in rendering a rectangle into the framebuffer with multiple-data arrays mapped as textures and custom fragment shaders enabled.

Graphics applications and video games favor throughput over latency, because, above a certain threshold, the human visual system is less sensitive to the frame-rate than to the level of detail of a 3-D scene. As a result, GPU implementations of the graphics pipeline are optimized for throughput rather than latency. Any given triangle might take hundreds to thousands of clock cycles to be rendered, but, at any given time, tens of thousands of vertices and fragments are in flight in the pipeline. Most medical physics applications are similar to video games in the sense that high throughput is considerably more important than low latency.

In graphics mode, the GPU is interfaced through a graphics API such as OpenGL or DirectX. The API provides functions for defining and rendering 3-D scenes. For instance, a 3-D triangular mesh is defined as an array of vertices and rendered by streaming the vertices to the GPU. Arrays of data can be moved to and from video memory as textures. Custom shading programs can be written using a high-level shading language such as CG, GLSL, or HLSL, and loaded on the GPU at run time. Early GPU computing programs written in such a framework have achieved impressive accelerations but suffered from several drawbacks: the code is difficult to develop and maintain because the computation is defined in terms of graphics concepts such as vertices, texture coordinates, and fragments; performance is compromised by the lack of access to all the capabilities of the GPU (most notably shared memory and scattered writes); and code portability is limited by the hardware-specific nature of some graphics extensions.

### II.C. GPU computing model

Compute-oriented APIs expose the massively parallel architecture of the GPU to the developer in a C-like programming paradigm. These commonly used APIs include NVIDIA CUDA, Microsoft DirectCompute, and OpenCL. For cohesion, this review focuses on CUDA, currently the most popular GPU computing API, but the concepts it presents are readily applied to other APIs. CUDA provides a set of extensions to the C language that allows the programmer to access computing resources on the GPU such as video memory, shading units, and texture units directly, without having to program the graphics pipeline.[7] From a hardware perspective, a CUDA-enabled graphics card comprises SGRAM memory and the GPU chip itself—a collection of streaming multiprocessors (MPs) and on-chip memory.

In the CUDA paradigm, a parallel task is executed by launching a multithreaded program called a *kernel*. The computation of a kernel is distributed to many threads, which are grouped into a *grid* of *blocks* (Fig. 5). Physically, the members of a thread block run on the same MP for their entire lifetime, communicate information through fast shared memory and synchronize their execution by issuing barrier instructions. Threads belonging to different blocks are required to execute independently of one another, in arbitrary order. Within one thread block, threads are further divided in groups of 32 called *warps*. Each warp executes in a SIMD fashion, with the MP broadcasting the same instruction to all its cores repeatedly until the entire warp is processed. When one warp stalls, for instance, because of a memory operation, the MP can hide this latency by quickly switching to a ready warp.

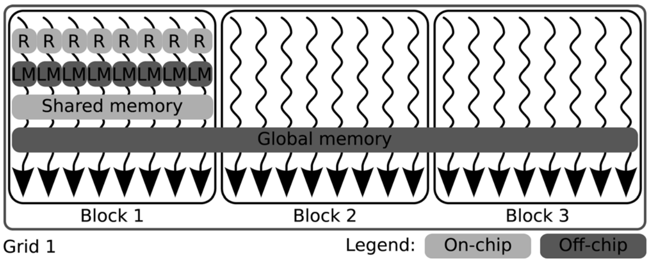

FIG. 5. GPU thread and memory hierarchy. Threads are organized as a grid of thread blocks. Threads within a block are executed on the same MP and have access to on-chip private registers (R) and shared memory. Additional global and local memories (LM) are available off-chip to supplement limited on-chip resources.

Even though each MP runs as a SIMD device, the CUDA programming model allows threads within a warp to follow different branches of a kernel. Such diverging threads are not executed in parallel but sequentially. Therefore, CUDA developers can safely write kernels which include *if* statements or variable-bounds *for* loops without taking into account the SIMD behavior of the GPU at the warp level. As we will see in Sec. II D, thread divergence substantially reduces performance and should be avoided.

The organization of the GPUs memory mirrors the hierarchy of the threads (Fig. 5). *Global memory* is randomly accessible for reading and writing by all threads in the application. *Shared memory* provides storage reserved for the members of a thread block. *Local memory* is allocated to threads for storing their private data. Last, private *registers* are divided among all the threads residing on the MP. Registers and shared memory, located on the GPU chip, have much lower latency than local and global memories, implemented in SGRAM. However, global memory can store several gigabytes of data, far more than shared memory or registers. Furthermore, while shared memory and registers only hold data temporarily, data stored in global memory persists beyond the lifetime of the kernels.

Two other types of memories are available on the GPU, called *texture* and *constant* memories. Both of these memories are read-only and cached for fast access. In addition, texture memory fetches are serviced by dedicated hardware units that can perform linear filtering and address calculations. In devices of compute capability 2.0 and greater, global memory operations are serviced by two levels of cache, namely a per-MP L1 cache and a unified L2 cache.

Unlike previous graphics APIs, CUDA threads can write multiple data to arbitrary memory locations. Such *scattered* writes are useful for many algorithms, yet, conflicts can arise when multiple threads attempt to write to the same memory location simultaneously. In such a case, only one of the writes is guaranteed to succeed. To safely write data to a common memory location, threads must use an *atomic operation*; for instance, an atomic add operation accumulates its operand into a given memory location. The GPU processes conflicting atomic writes in a serial manner to avoid data write hazards.

An issue important in medical physics is the reliability of memory operations. Errors introduced while reading or writing memory can have harmful consequences for dose calculation or image reconstruction. The memory of consumer-grade GPUs is optimized for speed because the occasional bit flip has little consequence in a video game. Professional and high-performance computing GPUs are designed for a higher level of reliability that they achieve using specially chosen hardware operated at lower clock rate. For further protection against unavoidable cosmic radiations and other sources of error, some of the more recent GPUs store redundant error-correcting codes (ECCs) to ensure the correctness of the memory content. In NVIDIA Tesla cards, these ECC bits occupy 12.5% of the GPU memory and reduce the peak memory bandwidth.

### II.D. Performance optimization

Even though CUDA has greatly simplified the development of distributed computing applications for the GPU, achieving optimal performance requires careful allocation of compute and storage resources.

As stated by Amdahl's law, the highest achievable speedup is determined by the sequential fraction of the program. Hence, to achieve GPU acceleration, developers must first focus on parallelizing as much sequential code as possible by reformulating their algorithm.

High-end GPUs boast more than one TFLOPS of theoretical peak performance (Fig. 3). However, a major concern is that the memory bandwidth does not allow data to be read and written as fast as it is consumed. For instance, the C1060 GPU can perform 933 GFLOPS, but due to its memory bandwidth of 408 GB/s, it can only read and write 102 floating-point values per second. Therefore, a kernel that performs less than ten floating-point operations for each memory access is memory-bound, a situation frequent in medical physics. As a general design rule, computation should be formulated to maximize the ratio of arithmetic to memory operations, a quantity called the *arithmetic intensity*.

Global memory also has higher latency and lower bandwidth than on-chip memory resources. Fetching data from global memory takes hundreds of clock cycles, whereas on-chip registers and shared memory can be accessed in a single cycle. Efficient GPU codes use shared memory to minimize global memory access, for example, by grouping threads into blocks such that threads work on common data in shared memory. If memory is only read, constant memory provides faster access than global memory and allows constant values to be broadcast to all the threads. Texture memory is also useful for accessing large datasets (such as volume images representing patient models) because it is cached and it can be linearly interpolated in hardware.

However, for a wide range of applications, the use global memory is unavoidable. In these situations, global memory performance can be dramatically increased by carefully coordinating memory accesses. Most GPUs have a very wide memory bus, capable of fetching up to 128 bytes of data in a single memory transaction. When multiple threads issue load instructions simultaneously, optimal bandwidth utilization is achieved when these accesses are serviced with a minimal number of transactions. Such a situation occurs when all the threads within a warp coalesce to access contiguous memory locations. For instance, it is common to process arrays of vectors, such as triplets representing points in space. In C, these data would typically be stored as an array of triplets. However, in a CUDA kernel, array elements are processed in parallel, whereas triplet components are processed sequentially. Therefore, optimal data access requires that the array of triplets be stored as three contiguous scalar arrays.

Another important performance bottleneck is memory transfer via the PCIe bus between the computer host and the device memory. Minimizing data transfers between GPU and CPU is essential for high-performance computing. Portions of the code that are not data-parallel should sometimes be ported onto the GPU to avoid such transfers. Likewise, when a data-parallel task has very low arithmetic intensity, it may be performed more efficiently directly on the CPU. For instance, the GPU is not efficient at adding two matrices stored on the host because such operation requires three GPU-CPU data transfers for each floating-point operation.

One last memory optimization to consider is avoiding scattered writes. Scattered writes can create write hazards, unless slower atomic operations are used. Some computation problems can be reformulated in a *gather* fashion. In a gather approach, threads read data from an array of addresses, while in a scatter approach, they write data to an array of addresses. When possible, scatter operations should be replaced with equivalent gather operations that are more efficient on the GPU.

Avoiding thread divergence is another key to achieving high performance. The MPs can be programmed as scalar multiprocessors, but diverging execution paths are serialized. Efficient GPU implementations use various strategies to avoid thread divergence. For instance, branching can be moved up the pipeline to ensure that the members of a thread block follow the same execution path. Sometimes, branches can be avoided altogether and replaced with equivalent arithmetic expressions. Loops can be performed with constant bounds and unrolled by the compiler.

On a recent GPU, each MP is capable of hosting thousands of threads. To achieve high compute throughput, the number of threads residing on the MPs should be maximized to provide the warp scheduler with a large supply of ready warps. However, *thread occupancy*, defined as the number of active threads per MP as a percentage of the device full capacity, often competes with *thread efficiency*, defined as the overall computational efficiency of the individual threads.

Thread occupancy is determined by the number and size of the thread blocks, the amount of shared memory per block, and the number of registers per thread. Shared memory and registers are scarce resources that are divided among thread blocks and threads, respectively. GPU implementations that achieve high thread occupancy use few on-chip resources per thread. As a result, they have poor thread efficiency because they have to rely on slower global and local memories in place of shared memory and registers. Reciprocally, GPU codes with high thread efficiency cannot maintain high thread occupancy. As a general design principle, the amount of on-chip resources allocated to each thread should be finely tuned to optimize the trade-off between thread occupancy and thread efficiency.

The GPU supports slow and fast math, and single- and double-precision computing. Fast math uses special functional units to accelerate the evaluation of mathematical expressions such as sine, cosine, and exponential, at the cost of a loss of accuracy. When high accuracy is not required, fast math functions and single-precision arithmetic are preferred.

## III. GPU COMPUTING IN MEDICAL PHYSICS

GPU computing has become a useful research tool for a wide range of medical physics applications (Fig. 1). We review some of the most notable implementations, illustrating how the principles described in the previous section are applied for high-performance computing.

### III.A. Image reconstruction

With progress in imaging systems and algorithms, the computational complexity of image reconstruction has increased dramatically. Yet, fast image reconstruction is often required in the clinic to allow the technologist to review the images while the patient is waiting. Reconstruction speed is even more critical in real-time imaging applications, such as intraoperative cone-beam CT (CBCT),[8] MR-guided cardiac catherization,[9] or online adaptive therapy.[10] Ultrafast image reconstruction could also allow the clinician to adjust reconstruction parameters interactively and optimize the noise/spatial resolution trade-off.[11]

In the past, real-time reconstruction speeds have been achieved by designing custom hardware systems based on application-specific integrated circuits (ASICs),[12] field-programmable gate arrays (FPGAs),[8] and digital signal processors (DSPs).[13] In most cases, the GPU can exceed the performance of these designs for a fraction of their cost.

Most imaging systems being linear, image reconstruction often consists in inverting a linear transformation, for instance, a Radon or a Fourier transform. These transforms can be discretized and mathematically represented by a matrix–vector multiplication, where the vector is a discretization of the sample being imaged, and the matrix describes the linear response of the imaging system. The matrix is frequently sparse and, therefore, the matrix–vector multiplication is not performed explicitly, using the matrix multiplication formula, but implicitly, using a procedural algorithm. The transpose of the matrix is also often used in reconstruction, for example, in a backprojection operation. With few exceptions, these matrix–vector and transposed matrix–vector multiplications consume the bulk of the computation in image reconstruction.

The acceleration of the filtered backprojection (FBP) algorithm[5] represented the first successful implementation of a nongraphics compute application on dedicated graphics hardware. This technique relied on texture-mapping hardware for efficiently mapping filtered projections to image voxels along projective lines. Over the years, the GPU pipeline has been enriched with new features; nevertheless, texture-mapping has remained the most efficient and popular approach for performing backprojection on the GPU. The backprojection operation fits the GPU pipeline formidably well. Parallel- and cone-beam projections can be modeled using orthographic and perspective transformations, respectively. The projection values, stored in texture memory, can be interpolated bilinearly by the hardware with no added computation. The very first implementation of this design proved very challenging because, on SGI workstations, arithmetic computations and framebuffer depth were limited to 8 bit and 24 bit integers, respectively.[5] Later work, which focused on consumer-grade GPUs, had to face even greater challenges as these cheaper devices provided only 8 bit for the accumulation buffer.[14] However, it was shown that the multiple color channels could be combined to virtually extend the accuracy of the hardware to 12 bit.[14] This technique solved a critical problem in the backprojection process, as it allowed the backprojection of multiple views to be accumulated directly on the hardware, without requiring costly transfers to the CPU.

In 2002, GPU vendors added support for floating-point arithmetic to overcome the limitations of the integer graphics pipeline and enable high-dynamic-range graphics. With higher bit depth, the accuracy of GPU-based CBCT reconstruction steadily improved, first based on 16 bit,[15] and, later, 32 bit floating-point arithmetic.[16,17] A similar texture-mapping approach was investigated for exact Katsevich reconstruction of helical CBCT.[18] With the launch of compute-specific APIs, CBCT reconstruction was adapted to run using Brook[19] and CUDA.[16,20,21] Although CUDA still uses a texture-mapping mechanism for mapping projection values to voxels, it lacks some of the efficient features only available in the graphics pipeline.[16] CBCT reconstruction is highly similar to graphics rendering, therefore useful computation can be inserted at nearly every stage of the graphics pipeline.[16] Yet, CUDA also provides access to features not available in the standard graphics pipeline, such as shared memory and scattered writes. As a result, it is unclear which approach is truly superior, and reports have presented contradictory results.[6,21]

In CUDA, CBCT backprojection is implemented by assigning one thread to each voxel. Thread blocks can be formed as subrows of the image[20] or as squares to enhance data locality.[21] Projection views, stored in texture memory, are accessed using hardware bilinear interpolation. Shared memory, although not a suitable storage of projection data for lack of hardware filtering, can be used to store projection matrix coefficients.[20] GPU texture-mapping was also applied to reconstruct digital tomosynthesis[22] and digitally reconstructed radiographs.[23]

Iterative reconstruction is computationally challenging and was seen early on as a critical target for GPU acceleration. Whereas analytical reconstruction is fully parallel, iterative reconstruction is fundamentally sequential. Hence, algorithms that perform minimal computation within each iteration are not efficiently implemented on the GPU for lack of a sufficiently parallel workload.[14] For instance, the algebraic reconstruction technique (ART) is not suitable for the GPU because each iteration only processes a single projection line. A more suitable algorithm is simultaneous ART (SART), which updates the image after the backprojection of an entire projection view.[14] Several other iterative algorithms were also adapted to the GPU, including expectation-maximization,[15,24] ordered-subsets expectation-maximization,[24,25] the ordered-subsets convex algorithm,[26] and total-variation reconstruction.[27]

Iterative reconstruction for projective imaging modalities such as PET, SPECT, and CT alternates forward- and backprojection operations. In principle, these operations can be performed either in a voxel-driven or line-driven manner. Output-driven operations are gather operations, while input-

TABLE I. Scatter and gather operations in iterative reconstruction for computed tomography.

| | Forward projection | Backprojection |
|---|---|---|
| Line-driven | Gather | Scatter |
| Voxel-driven | Scatter | Gather |

driven operations are scatter operations (Table I). Gather and scatter operations model the sparse system matrix in a compressed-row and compressed-column fashion, respectively. For instance, in a line-driven forward projection, each thread accumulates all the voxels traversed by a given projection line. Likewise, in a voxel-driven forward projection, each thread adds the contribution of a given voxel to all the projection lines it intersects.

It is important to note that both gather and scatter formulations produce the same output and have the same theoretical complexity. However, on the GPU, gather operations are more efficient than equivalent scatter operations because memory reads and writes are asymmetric: memory reads can be cached and are therefore faster than memory writes; furthermore, memory reads can exploit hardware-accelerated trilinear filtering. Last, by writing data in an orderly fashion, gather operations avoid write hazards. Scatter operations require slower atomic operations to avoid such write hazards.

In PET and SPECT, measurements are sometimes acquired in list-mode format. Unlike sinogram bins, list-mode projection lines are not ordered but are stored in a long list, in the order individual events are acquired. List-mode is an increasingly popular method for dealing with data sparsity, in particular, for time-of-flight (TOF) PET and dynamic studies. Because projection lines are not ordered, reconstruction is limited to line-driven approaches, a limitation that precludes the use of hardware texture-mapping. A further complicating factor is that projection lines can have arbitrary positions and orientations. The first implementation of list-mode reconstruction was developed for the GPU[28] using the graphics pipeline. In the backprojection, it achieves scattered writes by using the GPU rasterizer to select which frame-buffer pixels are written to. An application of the technique to TOF PET reconstruction was also demonstrated.[29] A similar list-mode reconstruction technique, implemented using CUDA, uses shared memory as a managed cache for processing image slices in parallel and avoids thread divergence by splitting lines into homogeneous groups.[30]

GPU computing has also been investigated for MRI image reconstruction. Most MRI pulse sequences are designed so that image reconstruction can be performed with a simple fast Fourier transform (FFT). The FFT—already quite efficient on the CPU—can be further accelerated on the GPU.[31] An optimized GPU implementation of the FFT called CUFFT is also provided with CUDA.

Some other MRI schemes require more complex image reconstruction algorithms. Two particularly computationally demanding schemes have been accelerated using the GPU, namely non-Cartesian k-space sampling[32] and sensitivity-encoded parallel imaging.[11] Both imaging methods aim at accelerating data acquisition by reduced sampling.

Non-Cartesian reconstruction can be performed using either an analytical method called *gridding* or an iterative solver. Gridding includes four steps, namely density correction, convolution, FFT, and deapodization. The gridding convolution—the most challenging step—can be accomplished either in gather or scatter fashion.[32] For radial k-space sampling, the GPU rasterizer can also be used to perform the gridding convolution in a scatter fashion.[31] While gather and scatter are optimal with respect to either writing grid cells or reading k-space samples, respectively, a hybrid approach was proposed and shown to yield superior performance.[32] The compute power of the GPU can also be harnessed to implement more complex iterative approaches, such as those based on a conjugate gradient solver.[33]

Parallel imaging techniques such as SENSE or GRAPPA use sensitivity-encoded information from many receive coils to dealias undersampled measurements. Each set of unaliased pixels are computed by solving a system of linear equations. The system matrix, formed by adding a regularization term to the sensitivity matrix, defines a system of equations that can be solved by Cholesky factorization and forward/backward substitution. The Cholesky factorization can be accelerated on the GPU by setting $N$ threads to collaborate, where $N$ is the aliasing factor.[11] The forward/backward substitution step is harder to parallelize and must be handled by a single thread.[11] Parallel imaging and non-Cartesian acquisition can be combined in a single reconstruction algorithm using a conjugate gradient solver.[34] In this approach, the linear system, which is solved using the CUBLAS library, encapsulates the coil sensitivities and a nonequispaced FFT.[32] A CPU/GPU low-latency high-frame-rate reconstruction pipeline was developed based on this approach to enable MR imaging in real-time interventional applications.[9]

Water–fat separation using IDEAL reconstruction is also amenable to GPU acceleration.[35] Vectorization is particularly efficient because the observation matrix to be inverted is independent of voxel coordinates.

GPU computing was also investigated for other imaging modalities. In optical imaging, the GPU can accelerate the FFT in optical coherence tomography[36] and the FBP in optical projection tomography.[37] In ultrasound imaging, the GPU was used in place of dedicated hardware to implement real-time Doppler imaging,[38] 2-D temperature imaging,[39] and free-hand 3-D reconstruction.[40] In breast tomosynthesis, iterative reconstruction was made practical by GPU acceleration.[41]

## III.B. Dose calculation and treatment plan optimization

Radiation therapy (RT) uses ionizing radiation to inflict lethal damage to cancer cells while striving to spare normal tissue. Computer dose calculations are routinely relied upon for planning the delivery of RT. For instance, in IMRT, the fluence maps for each beam angle are obtained by solving an optimization problem which involves such dose calculations. Currently, clinical IMRT treatment plans are obtained in a trial-and-error fashion: multiple feasible plans are generated sequentially until a good trade-off between dose to the target and sparing of the organs at risk is reached. Accelerating dose calculation and IMRT optimization would help reduce the clinical workload substantially.[42] Fast treatment planning is also a key component of online adaptive RT, which has been proposed as a new paradigm for improving the quality of radiation treatments.[10] In this scheme, a custom plan is prepared for each treatment fraction based on an online volumetric model of the patient and on the full history of past dose fractions.

Dose calculation has traditionally been achieved using three major models: pencil beam (PB), convolution-superposition (CS), and Monte-Carlo (MC), by order of accuracy and computational complexity. MC methods are the gold standard for accuracy but are substantially slower than analytical methods. In MC, dose is computed by simulating and aggregating the histories of billions of ionizing particles. MC methods are often qualified as "embarrassingly parallel" because they are ideally suited for parallel computing architectures, such as computer clusters. Indeed, particles histories can be calculated independently, in parallel, provided that the random number generators are initialized carefully. Unfortunately, implementing computationally efficient MC methods on GPU hardware is extremely challenging. Several high-energy MC codes have been ported to the GPU, including PENELOPE[43] and DPM,[44] but the speed-ups achieved were modest in comparison to those published for GPU-based image reconstruction or analytical dose calculation.

The main obstacle to efficient MC simulation on the GPU is that SIMD architectures cannot compute diverging particle histories in parallel. In high-energy MC, particles can be electrons, positrons, or photons and can undergo a wide range of physical processes, such as photoelectric absorption or pair production. A complicating factor is that secondary particles can be created as a result of these processes. On the GPU, MC is implemented by assigning one thread per particle.[43,44] However, this requires threads within a warp to compute different physical processes, and, as a result, these divergent threads are serialized by the scheduler. Hence, at any time, each GPU MP might compute only a few particle histories in parallel. Scoring is also a challenging task for the GPU, because aggregating information from thousands of particle histories into a summary data structure requires slower atomic operations to avoid data write hazards.

Several solutions have been proposed, such as creating a global queue of particles to be processed, placing newly created secondary particles in such queue and processing electrons and photons sequentially rather than concurrently.[45] The GPUMCD package, which implements these features, achieved a speed-up of 200 over DPM while providing good numerical agreement.

MC methods are also standard for modeling optical photon migration in turbid media. Because these simulations are often much simpler, high speed-ups have been achieved. For instance, a speed-up factor of 1000 was achieved for a simple homogeneous slab geometry.[46] Optical MC codes only need to track one type of particle, namely photons. Furthermore, the same sequence of steps is applied to all particles, which eliminates thread divergence. Computational efficiency is reduced for more complex simulations, in particular when using heterogeneous 3-D media[47] or multiple layers of tissue.[48] Complex tissue geometries can also be described using triangular meshes, but these methods are benefiting only modestly from GPU acceleration[49] because efficient GPU calculation of the intersection of a ray with a triangular mesh remains a challenging problem and an intense focus of research in computer graphics.[50]

Simpler dose calculation models rely on raytracing and analytical kernels to represent dose deposition in matter. One of such models, the PB model, decomposes the continuous diverging radiation beam into a superposition of discrete beamlets. The dose deposition in a voxel is obtained by summing the contributions from all the beamlets. These contributions depend upon the radiological distance to the source and off-axis effects. Computing the radiological distance is computationally demanding because large electron density maps have to be integrated over many beamlets, using raytracing methods such as Siddon's algorithm. While Siddon's algorithm can be parallelized on the GPU by assigning each thread a line to raytrace,[42] it is less efficient than other methods. Frequent branching in Siddon's algorithm cause threads within a warp to diverge. Furthermore, within a given line, voxels have to be processed sequentially, which limits the potential for parallel processing.

An alternative approach computes the radiological distance by uniformly sampling the electron density map along the ray.[51] By storing the electron density map in texture memory, hardware trilinear interpolation is achieved and memory transactions are automatically cached. Furthermore, in this approach, the GPU can process voxels in parallel within a raytrace operation, a feature that makes finely grained parallelism possible. After precomputing the radiological distance, beamlets contributions to the voxels are evaluated in parallel, using one thread block per beamlet and one thread per voxel traversed.[51]

The CS dose calculation method offers a good compromise between accuracy and computational efficiency and is now the standard method for clinical treatment planning optimization. The method decomposes dose calculation into two steps, energy release and energy deposition. The first step computes the total energy released per mass (TERMA) along the path of the primary photon beam. Secondary particles (electron, positrons, and photons) spread this radiant energy to a larger volume of tissue, a process that is modeled by convolving the TERMA distribution with a polyenergetic kernel.

The CS algorithm was independently ported onto the GPU by two research groups.[52,53] Although physical deposition of dose in a medium is more intuitively formulated in a scatter fashion (i.e., ray-driven), these implementations were based on a gather approach (i.e., voxel-driven), more amenable to GPU computing. For each voxel, they calculate the contribution of the source to the TERMA by casting a ray backward toward the source. Because a ray must be traced back to the source for each voxel, the gather formulation has $O(n^4)$ theoretical complexity, higher than the $O(n^3)$ complexity of the scatter approach, which raytraces a fixed number of rays through the voxel grid.[52,54] However, the gather approach avoids potential data write hazards and allows for efficient coalesced memory access, with a large degree of cache reuse. Furthermore, it eliminates certain discretization artifacts by sampling all voxels with an equal number of rays.

It is interesting to note that the first attempt to port the CS algorithm onto the aimed at keeping the original structure of the algorithm intact.[55] Although this GPU version did run about 20 times faster than the CPU version, the GPU-optimized version of the same code was 900 times faster than the original CS code.[53] This example illustrates the importance of considering the specific architecture of the GPU when building GPU applications for optimal performance.

GPU-based CS methods have been further improved to achieve real-time performance and more accurate computed dose distributions. Computational performance improvements were achieved by maximizing thread coherence and occupancy, while accuracy was improved through better multifocal source modeling, kernel tilting, and physically accurate multispectral attenuation.[56] A related approach implemented on the GPU combines MC and CS methods, wherein photons are first simulated by MC but second particles handled by CS kernels.[57]

Another variation of the CS method was proposed using a nonvoxel-based broad-beam (NVBB) framework[54] and implemented using a GPU-based CS dose calculation engine.[58] The NVBB approach represents the objective function in a continuous manner rather than using discrete voxels, a feature that removes the need for storing large beamlet matrices.[54] Furthermore, while the approach performs raytracing in a scatter manner, write hazards and voxel sampling artifacts are avoided by using diverging pyramids in place of rays to represent photon transport from the source.[54] These pyramidal rays do no overlap yet cover the entire volume uniformly. To facilitate raytracing along pyramidal rays, computations are performed in the beam-eye view coordinate system. As a result of these features, the NVBB approach has $O(n^3)$ complexity but none of the problems associated with scatter raytracing.

In addition to dose computation, treatment planning optimization involves three other operations, namely objective function calculation, gradient calculation, and fluence update. To avoid slow GPU-CPU communications, these operations can be moved to the GPU. An IMRT treatment planning approach, based on an iterative gradient projection and the PB dose calculation model, was implemented using CUDA.[59] The fluence gradient was computed in parallel by assigning one thread per beamlet intensity. The objective function was evaluated by performing a parallel reduction with one thread per voxel.

Treatment plan optimization can be further accelerated using adaptive full dose correction methods, wherein fast approximate dose calculation is performed unless the fluence map changes substantially.[54] This treatment planning system can run in near real-time on a single GPU.

Aperture-based optimization methods, which derive MLC segment weights and shapes directly within IMRT optimization,[60] have also been implemented using CUDA. In one approach, the optimization problem is solved using a column generation method, which repeatedly selects the best aperture to be added to the MLC leaf sequence.[61] After an aperture has been added, a full optimization is ran to find the optimal intensities for each MLC segment, using the method previously outlined. Aperture selection is also executed on the GPU by assigning one thread per MLC leaf pair and computing the optimal position for the upper and lower leafs.[61]

The direct aperture optimization method was also extended to VMAT treatment planning optimization.[62] Optimization of VMAT is computationally demanding because dose distributions must be computed for many discrete beam angles. In this GPU implementation, the gantry rotation is discretized into 180 beam angles and the apertures generated one by one in a sequential way, until all angles are occupied. At each step, full fluence rate optimization is performed using previously developed GPU-based tools.[59]

### III.C. Image processing

Two fundamental tools in medical physics are image registration and segmentation. Image registration—the transformation of two or more images into a common frame of reference—enables physicians and researchers to analyze the variations between images taken at different time points or combine information from multiple imaging modalities. Image segmentation–the grouping of voxels with common properties–is used routinely in RT to delineate the various anatomical structures from CT scans. Manual organ contouring is tedious and time consuming, and automated methods can be used to streamline the clinical workflow.

Image registration and segmentation are computationally demanding procedures. Soon after the introduction of the first programmable GPU, computer graphics researchers realized that GPUs were ideally suited to accelerate data-parallel tasks such as image deformation and partial differential equations (PDEs) computation. For instance, the GPU can rigidly deform 3-D images using 3-D texture-mapping with trilinear interpolation. Nonrigid deformations can also be implemented on the GPU using parametric or nonparametric models.[63]

Nonparametric methods represent the deformation by a dense displacement field, which is optimized to minimize the discrepancy between reference and target image, while favoring smooth displacement fields. Various metrics have been proposed to measure the discrepancy between two images, such as an energy function[64] or an optimal mass transport criterion.[65] On the GPU, the optimal displacement field can be computed using a gradient solver.[64,65] In these approaches, the gradient of the objective is computed efficiently on the GPU because the pattern of memory accesses exhibits high locality.

In parametric models, the displacement field between a reference and a target image is defined by a smooth function of the spatial coordinate, such as a Bezier function,[66] a thin-plate spline,[67] or a B-spline.[68,69] B-spline image registration, a method popular for its flexibility and robustness, computes a fine displacement field by interpolating a coarse grid of uniformly spaced control points using piecewise continuous B-splines basis functions. However, B-spline registration is computationally intensive, mostly because of two operations: the interpolation from coarse to fine and the computation of the objective function gradient.[68] Key acceleration strategies for GPU-based B-spline registration include (1) aligning the fine voxel grid with the coarse control-point grid such that the value of B-spline basis functions only depends on the voxel offset within a tile and (2) optimizing data-parallel computation to fit the SIMD model of the GPU.[68]

A popular algorithm in image registration is the demons algorithm. In this algorithm, the voxels in the reference image (the "demons") apply local forces that displace the voxels in the target image. To mitigate the underdetermined nature of the registration problem, the displacement field is smoothed at each iteration. Multiple research groups have implemented the demons algorithm on the GPU using Brook[19] and CUDA.[70–72] Each of the five steps of the algorithm (namely spatial gradient, displacement, smoothing, image deformation, and stopping criterion) are implemented by as many kernels. For most kernels, the vectorization is trivial: image voxels and motion field components are processed independently, in parallel; furthermore, global memory bandwidth is optimized because memory operations are coalesced.

Image registration can also be based on a finite-element model (FEM). Such approach allows the incorporation of a biomechanical model of tissue deformation within image registration.[73] In one such approach, a sparse registration is first obtained by performing block matching on the GPU,[74] wherein thread blocks process different features in parallel and threads within a block compute the matching metric for various offsets. Next, the dense deformation field is computed by running an incremental finite-element solver on multicore CPUs.[74]

Coregistration can also be achieved between images from different imaging modalities, for instance, by maximizing their mutual information. However, evaluating the mutual information entails computing joint histograms of image intensities, which is nontrivial on the GPU.[75] The naive implementation, which consists in distributing the loop over the voxels to many threads, is not efficient because it induces data write hazards that are only avoided at the cost of computationally expensive atomic operations. However, the histogram procedure can be equivalently formulated as a "sort and count" algorithm to remove all memory conflicts.[75]

Automated segmentation has also attracted the attention of the computer graphics community. Level-set segmentation, one of the most popular approaches, defines the segmentation surface implicitly as the isocontour (isosurface) of a 2-D (3-D) function. The function, initialized with a user-defined seed, is iteratively updated according to a system of PDEs, which are highly amenable to GPU implementation. Performing the segmentation on the GPU has an additional advantage: the user can adjust the segmentation parameters interactively while visualizing GPU-accelerated volume renderings of the segmented volume.

The first implementation on the GPU of an image segmentation algorithm was achieved by formulating level-set segmentation as a sequence of blending operations.[76] Programmable shaders, introduced later, provided greater flexibility and enabled segmenting 3-D images using curvature regularization to favor smooth isosurfaces.[77–79] A key optimization is to avoid wasting computation on voxels that have no chance of entering the segmentation during the current iteration. One approach used a dynamic compressed image representation to limit the processing to active tiles.[77,79] Alternatively, it is possible to use a computation mask formed by dilating the segmentation.[78] Level-set segmentation is readily implemented in CUDA.[80] More complex segmentation schemes can also be implemented, for instance, robust statistical segmentation using adaptive region growing.[81]

### III.D. Other applications

GPU computing has been applied to other computational problems worth mentioning briefly. Many research groups have been interested in using the GPU for visualizing medical datasets in real-time, exploiting the inherent graphics abilities of the GPU. Various codes have been written to render a variety of datasets, including digitally reconstructed radiographs,[82] cardiac CT scans,[83] coregistered multimodal cardiac studies,[84] CT-based virtual colonoscopy,[85] and sinus virtual endoscopy.[86] GPU computing techniques have also been developed to aid surgical planning, for instance, through illustrative visualization[87] or biomechanical simulation.[88]

## IV. DISCUSSION

Most of the computing applications reviewed in this survey can be formulated as matrix–vector multiplications. Typically, the vector is a 2-D or 3-D image that represents a medical imaging scan or a dose distribution, and the matrix is almost always sparse. Most GPU implementations also rely on common optimization techniques for achieving high performance. Efficient GPU implementations reformulate computation to enhance parallelism and minimize CPU-GPU communications. They also typically store medical imaging datasets using 2-D or 3-D textures to exploit 2-D/3-D cache locality and built-in linear interpolation. When possible, they use shared memory to minimize costly data transfers to and from global memory or they coalesce global memory transactions. They perform arithmetic operations using GPU-intrinsic functions and native support for floating point. Last, they optimize the trade-off between thread occupancy and thread efficiency by adjusting block size and on-chip resource allocation.

Despite their unrivalled performance GPUs have several limitations when compared to CPUs. First, developing GPU code takes substantially more time because many more parameters must be considered. Large-scale software projects which mix CPU and GPU codes are also harder to debug and maintain. Programming the GPU requires learning new programming paradigms that are less intuitive than single-threaded programming. For instance, the programmer must consider the SIMD nature of the MPs to achieve high performance when using loops and branches.

Another issue is that currently published GPU implementations tend to overestimate the speed-up they achieve by comparing optimized GPU implementations against unoptimized, single-threaded CPU implementations. While some algorithms are very efficient on the GPU, other will only achieve modest accelerations and would not perform much faster than an optimized multithreaded CPU implementation. A complicating issue is that in large software packages, substantial portions of code are sequential and cannot be accelerated by GPUs, which limits the overall acceleration that GPUs can achieve. Last, precise computing using 64 bit floating-point arithmetic and slow math is substantially slower on the GPU than 32 bit fast math.

A further concern is that many of the tools that have become *de facto* standards for programming the GPU, such as CUDA, are proprietary. Compute APIs compatible with multiple GPU vendors exist but have not so far encountered much success in academia. Furthermore, if the past is any guide, GPU APIs have been changing rapidly in the last ten years, whereas most medical software evolve at a much slower pace.

As most hardware architectures, the GPU is constantly evolving to address the needs of its users. Starting from a small number of expert GPU hackers, the usage of the GPU has expanded to the masses of developers in industry and academia. To fit the needs of a much broader user base, the GPU programming interfaces are becoming increasingly abstract and automated. For instance, CUDA devices of compute capability 2.0 include a per-MP L1 cache and a unified L2 cache that services all global memory transactions. With such cache, it becomes less important to implement an efficient memory management strategy. However, these features, designed to make the programming interface more general-purpose, use transistor resources that could have otherwise been allocated to ALUs; hence, such "improvements" will reduce the speed-ups achieved by expert GPU programmers.

The introduction of GPUs in medical physics reignited interest in scientific computing, a topic that was previously quietly ignored beyond a small circle of researchers. The number of publications relating to the use of GPUs in medical physics (Fig. 1) is a measure of the tremendous enthusiasm that has captured the medical physics research community. While many developers were driven to the GPU because they were no longer getting a free ride from Moore's law, other simply found the GPU to be a fascinating device which, in the early days, presented a new source of compute power that could be tapped by ingeniously programming the graphics pipeline. Nowadays, the GPU is one of the standard tools in high-performance computing, and is being adopted throughout industry and academia.

## ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant 0854492 and by a Stanford Dean's fellowship.

[a] Electronic mail: pratx@stanford.edu

[1] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," Comput. Graph. Forum **26**(1), 21–51 (2008).
[2] J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips, "GPU computing," Proc. IEEE **96**(5), 879–899 (2008).
[3] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," IEEE MICRO **28**(4), 13–27 (2008).
[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," J. Parallel Distrib. Comput. **68**(10), 1370–1380 (2008).
[5] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *Proceedings of the Volume Visualization* (ACM, New York, USA, 1994), pp. 91–98.
[6] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick, "The landscape of parallel computing research: A view from Berkeley," Technical Report, 2006 (UCB/EECS-2006-183).
[7] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," in *Comp. Graph* (ACM, New York, USA, 2008), pp. 1–14.
[8] J. Li, C. Papachristou, and R. Shekhar, "An FPGA-based computing platform for real-time 3D medical imaging and its application to cone-beam CT reconstruction," J. Imaging Sci. Technol. **49**, 237–245 (2005).
[9] S. Roujol, B. D. de Senneville, E. Vahala, T. Sorensen, C. Moonen, and M. Ries, "Online real-time reconstruction of adaptive TSENSE with commodity CPU/GPU hardware," Magn. Reson. Med. **62**(6), 1658–1664 (2009).
[10] L. Xing, L. Lee, and R. Timmerman, "Adaptive radiation therapy and clinical perspectives," in *Image Guided and Adaptive Radiation Therapy*, edited by R. Timmerman and L. Xing (Lippincott Williams & Wilkins, Baltimore, MD, 2009), pp. 16–40.
[11] M. Hansen, D. Atkinson, and T. Sorensen, "Cartesian SENSE and k-t SENSE reconstruction using commodity graphics hardware," Magn. Reson. Med. **59**(3), 463–468 (2008).
[12] G. Dasika, A. Sethia, V. Robby, T. Mudge, and S. Mahlke, "Medics: Ultra-portable processing for medical image reconstruction," in *Proceedings of the PACT' 10*, (ACM, New York, NY, 2010), pp. 181–192.
[13] R. A. Neri-Calderon, S. Alcaraz-Corona, and R. M. Rodriguez-Dagnino, "Cache-optimized implementation of the filtered backprojection algorithm on a digital signal processor," J. Electron. Imaging **16**(4), 043010 (2007).
[14] K. Mueller and R. Yagel, "Rapid 3-D cone-beam reconstruction with the simultaneous algebraic reconstruction technique (SART) using 2-D texture mapping hardware," IEEE Trans. Med. Imaging **19**(12), 1227–1237 (2000).
[15] F. Xu and K. Mueller, "Accelerating popular tomographic reconstruction algorithms on commodity PC graphics hardware," IEEE Trans. Nucl. Sci. **52**(3), 654–663 (2005).
[16] F. Xu and K. Mueller, "Real-time 3D computed tomographic reconstruction using commodity graphics hardware," Phys. Med. Biol. **52**(12), 3405–3419 (2007).
[17] X. Zhao, J.-J. Hu, and P. Zhang, "GPU-based 3D cone-beam CT image reconstruction for large data volume," Int. J. Biomed. Imaging **2009**, 1 (2009).
[18] G. Yan, J. Tian, S. Zhu, C. Qin, Y. Dai, F. Yang, D. Dong, and P. Wu, "Fast Katsevich algorithm based on GPU for helical cone-beam computed tomography," IEEE Trans. Inf. Technol. Biomed. **14**(4), 1053–1061 (2010).
[19] G. C. Sharp, N. Kandasamy, H. Singh, and M. Folkert, "GPU-based streaming architectures for fast cone-beam CT image reconstruction and demons deformable registration," Phys. Med. Biol. **52**(19), 5771–5783 (2007).

[20] P. B. Noel, A. M. Walczak, J. Xu, J. J. Corso, K. R. Hoffmann, and S. Schafer, "GPU-based cone beam computed tomography," Comput. Methods Programs Biomed. **98**(3), 271–277 (2010).

[21] Y. Okitsu, F. Ino, and K. Hagihara, "High-performance cone beam reconstruction using CUDA compatible GPUs," Parallel Comput. **36**(2–3), 129–141 (2010).

[22] H. Yan, D. J. Godfrey, and F.-F. Yin, "Fast reconstruction of digital tomosynthesis using on-board images," Med. Phys. **35**(5), 2162–2169 (2008).

[23] H. Yan, L. Ren, D. J. Godfrey, and F.-F. Yin, "Accelerating reconstruction of reference digital tomosynthesis using graphics hardware," Med. Phys. **34**(10), 3768–3776 (2007).

[24] K. Chidlow and T. Moller, "Rapid emission tomography reconstruction," in *Proceedings of the Eurographics* (ACM, New York, USA, 2003), pp. 15–26.

[25] F. Xu, W. Xu, M. Jones, B. Keszthelyi, J. Sedat, D. Agard, and K. Mueller, "On the efficiency of iterative ordered subset reconstruction algorithms for acceleration on GPUs," Comput. Methods Programs Biomed. **98**(3), 261–270 (2010).

[26] J. S. Kole and F. J. Beekman, "Evaluation of accelerated iterative x-ray CT image reconstruction using floating point graphics hardware," Phys. Med. Biol. **51**(4),875–889 (2006).

[27] X. Jia, Y. Lou, R. Li, W. Y. Song, and S. B. Jiang, "GPU-based fast cone beam CT reconstruction from undersampled and noisy projection data via total variation," Med. Phys. **37**(4), 1757–1760 (2010).

[28] G. Pratx, G. Chinn, P. Olcott, and C. Levin, "Accurate and shift-varying line projections for iterative reconstruction using the GPU," IEEE Trans. Med. Imaging **28**(3), 415–422 (2009).

[29] G. Pratx, S. Surti, and C. Levin, "Fast list-mode reconstruction for time-of-flight PET using graphics hardware," IEEE Trans. Nucl. Sci. **58**(1), 105–109 (2011). (Place: San Francisco, CA)

[30] G. Pratx, J.-Y. Cui, S. Prevrhal, and C. S. Levin, "3-D tomographic image reconstruction from randomly ordered lines with CUDA," in *GPU Computing Gems Emerald Edition*, edited by W. mei Hwu (Morgan Kaufmann, San Francisco, CA 2011), pp. 679–691.

[31] T. Schiwietz, T. C. Chang, P. Speier, and R. Westermann, "MR image reconstruction using the GPU," Proc. SPIE **6142**, 61423T (2006).

[32] T. Sorensen, T. Schaeffter, K. Noe, and M. Hansen, "Accelerating the nonequispaced fast Fourier transform on commodity graphics hardware," IEEE Trans. Med. Imaging **27**(4), 538–547 (2008).

[33] S. Stone, J. Haldar, S. Tsao, W. W. Hwu, B. Sutton, and Z.-P. Liang, "Accelerating advanced MRI reconstructions on GPUs," J. Parallel Distrib. Comput. **68**(10), 1307–1318 (2008).

[34] T. Sorensen, D. Atkinson, T. Schaeffter, and M. Hansen, "Real-time reconstruction of sensitivity encoded radial magnetic resonance imaging using a graphics processing unit," IEEE Trans. Med. Imaging **28**(12), 1974–1985 (2009).

[35] D. Johnson, S. Narayan, C. A. Flask, and D. L. Wilson, "Improved fat–water reconstruction algorithm with graphics hardware acceleration," J. Magn. Reson. Imaging **31**(2), 457–465 (2010).

[36] Y. Watanabe and T. Itagaki, "Real-time display on Fourier domain optical coherence tomography system using a graphics processing unit," J. Biomed. Opt. **14**(6), 060506 (2009).

[37] C. Vinegoni, L. Fexon, P. F. Feruglio, M. Pivovarov, J.-L. Figueiredo, M. Nahrendorf, A. Pozzo, A. Sbarbati, and R. Weissleder, "High throughput transmission optical projection tomography using low cost graphics processing unit," Opt. Express **17**(25), 22320–22332 (2009).

[38] L. W. Chang, K. H. Hsu, and P. C. Li, "Graphics processing unit-based high-frame-rate color doppler ultrasound processing," IEEE Trans. Ultrason. Ferroelectr. Freq. Control. **56**(9), 1856–1860 (2009).

[39] D. Liu and E. Ebbini, "Real-time two-dimensional temperature imaging using ultrasound," IEEE Trans. Biomed. Eng. **57**(1), 12–16 (2010).

[40] P. Coupé, P. Hellier, N. Azzabou, and C. Barillot, "3D freehand ultrasound reconstruction based on probe trajectory," Lecture Notes in Computer Science (Springer, Berlin, Germany, 2005), Vol. **3749**, pp. 597–604.

[41] I. Goddard, T. Wu, S. Thieret, A. Berman, and H. Bartsch, "Implementing an iterative reconstruction algorithm for digital breast tomosynthesis on graphics processing hardware," Proc SPIE **6142**(1), 61424V (2006).

[42] M. de Greef, J. Crezee, J. C. van Eijk, R. Pool, and A. Bel, "Accelerated ray tracing for radiotherapy dose calculations on a GPU," Med. Phys. **36**(9), 4095–4102 (2009).

[43] A. Badal and A. Badano, "Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit," Med. Phys. **36**(11), 4878–4880 (2009).

[44] X. Jia, X. Gu, J. Sempau, D. Choi, A. Majumdar, and S. B. Jiang, "Development of a GPU-based Monte Carlo dose calculation code for coupled electron-photon transport," Phys. Med. Biol. **55**(11), 3077–3086 (2010).

[45] S. Hissoiny, B. Ozell, H. Bouchard, and P. Despres, "GPUMCD: A new GPU-oriented Monte Carlo dose calculation platform," Med. Phys. **38**(2), 754–764 (2011).

[46] E. Alerstam, T. Svensson, and S. Andersson-Engels, "Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration," J. Biomed. Opt. **13**(6), 060504 (2008).

[47] Q. Fang and D. A. Boas, "Monte Carlo simulation of photon migration in 3D turbid media accelerated by graphics processing units," Opt. Express **17**(22), 20178–20190 (2009).

[48] W. Lo, T. Han, J. Rose, and L. Lilge, "GPU-accelerated Monte Carlo simulation for photodynamic therapy treatment planning," Proc. SPIE **7373**, 737313 (2009).

[49] H. Shen and G. Wang, "A tetrahedron-based inhomogeneous Monte Carlo optical simulator," Phys. Med. Biol. **55**(4), 947–962 (2010).

[50] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: A general purpose ray tracing engine," ACM Trans. Graphics **29**(4), 1–13 (2010).

[51] X. Gu, D. Choi, C. Men, H. Pan, A. Majumdar, and S. B. Jiang, "GPU-based ultra-fast dose calculation using a finite size pencil beam model," Phys. Med. Biol. **54**(20), 6287–6297 (2009).

[52] R. Jacques, R. Taylor, J. Wong, and T. McNutt, "Towards real-time radiation therapy: GPU accelerated superposition/convolution," Comput. Methods Programs Biomed. **98**(3), 285–292 (2010).

[53] S. Hissoiny, B. Ozell, and P. Despres, "A convolution-superposition dose calculation engine for GPUs," Med. Phys. **37**(3), 1029–1037 (2010).

[54] W. Lu, "A non-voxel-based broad-beam (NVBB) framework for IMRT treatment planning," Phys. Med. Biol. **55**(23), 7175–7210 (2010).

[55] S. Hissoiny, B. Ozell, and P. Despres, "Fast convolution-superposition dose calculation on graphics hardware," Med. Phys. **36**(6), 1998–2005 (2009).

[56] R. Jacques, J. Wong, R. Taylor, and T. McNutt, "Real-time dose computation: GPU-accelerated source modeling and superposition/convolution," Med. Phys. **38**(1), 294–305 (2011).

[57] B. Zhou, C. X. Yu, D. Z. Chen, and X. S. Hu, "GPU-accelerated Monte Carlo convolution/superposition implementation for dose calculation," Med. Phys. **37**(11), 5593–5603 (2010).

[58] Q. Chen, M. Chen, and W. Lu, "Ultrafast convolution/superposition using tabulated and exponential cumulative-cumulative-kernels on GPU," in *Proceedings of the 16th International Conference on the Use of Computers in Radio Therapy*, edited by J.-J. Sonke (2010).

[59] C. Men, X. Gu, D. Choi, A. Majumdar, Z. Zheng, K. Mueller, and S. B. Jiang, "GPU-based ultrafast IMRT plan optimization," Phys. Med. Biol. **54**(21), 6565–6573 (2009).

[60] C. Cotrutz and L. Xing, "Segment-based dose optimization using a genetic algorithm," Phys. Med. Biol. **48**(18), 2987–2998 (2003).

[61] C. Men, X. Jia, and S. Jiang, "GPU-based ultra-fast direct aperture optimization for online adaptive radiation therapy," Phys. Med. Biol. **55**, 4309–4319 (2010).

[62] C. Men, H. E. Romeijn, X. Jia, and S. B. Jiang, "Ultrafast treatment plan optimization for volumetric modulated arc therapy (VMAT)," Med. Phys. **37**(11), 5787–5791 (2010).

[63] R. Shams, P. Sadeghi, R. Kennedy, and R. Hartley, "A survey of medical image registration on multicore and the GPU," IEEE Signal Process. Mag. **27**(2), 50–60 (2010).

[64] R. Strzodka, M. Droske, and M. Rumpf, "Image registration by a regularized gradient flow. A streaming implementation in DX9 graphics hardware," Computing **73**, 373–389 (2004).

[65] T. ur Rehman, E. Haber, G. Pryor, J. Melonakos, and A. Tannenbaum, "3D nonrigid registration via optimal mass transport on the GPU," Med. Image Anal. **13**(6), 931–940 (2009).

[66] G. Soza, M. Bauer, P. Hastreiter, C. Nimsky, and G. Greiner, "Non-rigid registration with use of hardware-based 3D Bezier functions," *Lecture Notes in Computer Science* (Springer, Berlin, Germany, 2002), Vol. **2489**, pp. 549–556.

[67] D. Levin, D. Dey, and P. Slomka, "Acceleration of 3D, nonlinear warping using standard video graphics hardware: Implementation and initial validation," Comput. Med. Imaging Graph. **28**(8), 471–483 (2004).

[68] J. A. Shackleford, N. Kandasamy, and G. C. Sharp, "On developing B-spline registration algorithms for multi-core processors," Phys. Med. Biol. **55**(21), 6329–6351 (2010).

[69] M. Modat, G. R. Ridgway, Z. A. Taylor, M. Lehmann, J. Barnes, D. J. Hawkes, N. C. Fox, and S. Ourselin, "Fast free-form deformation using graphics processing units," Comput. Methods Programs Biomed. **98**(3), 278–284 (2010).

[70] S. S. Samant, J. Xia, P. Muyan-Ozcelik, and J. D. Owens, "High performance computing for deformable image registration: Towards a new paradigm in adaptive radiotherapy," Med. Phys. **35**(8), 3546–3553 (2008).

[71] X. Gu, H. Pan, Y. Liang, R. Castillo, D. Yang, D. Choi, E. Castillo, A. Majumdar, T. Guerrero, and S. B. Jiang, "Implementation and evaluation of various demons deformable image registration algorithms on a GPU," Phys. Med. Biol. **55**(1), 207–219 (2010).

[72] P. Muyan-Ozcelik, J. D. Owens, J. Xia, and S. S. Samant, "Fast deformable registration on the GPU: A CUDA implementation of Demons," in *Proceedings of the International Conference on Computational Sciences and Its Applications* (IEEE Computer Society Washington, DC, USA, 2008), pp. 223–233.

[73] G. R. Joldes, A. Wittek, and K. Miller, "Real-time nonlinear finite element computations on GPU—Application to neurosurgical simulation," Comput. Methods Appl. Mech. Eng. **199**(49–52), 3305–3314 (2010).

[74] Y. Liu, A. Fedorov, R. Kikinis, and N. Chrisochoides, "Real-time nonrigid registration of medical images on a cooperative parallel architecture," in *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine* (IEEE Computer Society Washington, DC, USA, 2009), pp. 401–404.

[75] R. Shams, P. Sadeghi, R. Kennedy, and R. Hartley, "Parallel computation of mutual information on the GPU with application to real-time registration of 3D medical images," Comput. Methods Programs Biomed. **99**(2), 133–146 (2010).

[76] M. Rumpf and R. Strzodka, "Level set segmentation in graphics hardware," in *Proceedings of IEEE International Conference on Image Processing* (IEEE Computer Society Washington, DC, USA, 2001), Vol. 3, pp. 1103–1106.

[77] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker, "Interactive deformation and visualization of level set surfaces using graphics hardware," in *Proceedings of the IEEE Visualization* (IEEE Computer Society Washington, DC, USA, 2003), p. 11.

[78] A. Sherbondy, M. Houston, and S. Napel, "Fast volume segmentation with simultaneous visualization using programmable graphics hardware," in *Proceedings of the IEEE Visualization* (IEEE Computer Society Washington, DC, USA, 2003), pp. 171–176.

[79] J. E. Cates, A. E. Lefohn, and R. T. Whitaker, "GIST: An interactive, GPU-based level set segmentation tool for 3D medical images," Med. Image Anal. **8**(3), 217–231 (2004).

[80] W.-K. Jeong, J. Beyer, M. Hadwiger, A. Vazquez, H. Pfister, and R. Whitaker, "Scalable and interactive segmentation and visualization of neural processes in EM datasets," IEEE Trans. Vis. Comput. Graph. **15**(6), 1505–1514 (2009).

[81] A. Narayanaswamy, S. Dwarakapuram, C. Bjornsson, B. Cutler, W. Shain, and B. Roysam, "Robust adaptive 3-D segmentation of vessel laminae from fluorescence confocal microscope images and parallel GPU implementation," IEEE Trans. Med. Imaging **29**(3), 583–597 (2010).

[82] J. Spoerk, H. Bergmann, F. Wanschitz, S. Dong, and W. Birkfellner, "Fast DRR splat rendering using common consumer graphics hardware," Med. Phys. **34**(11), 4302–4308 (2007).

[83] Q. Zhang, R. Eagleson, and T. M. Peters, "Dynamic real-time 4D cardiac MDCT image display using GPU-accelerated volume rendering," Comput. Med. Imaging Graph. **33**(6), 461–476 (2009).

[84] D. Levin, U. Aladl, G. Germano, and P. Slomka, "Techniques for efficient, real-time, 3D visualization of multi-modality cardiac data using consumer graphics hardware," Comput. Med. Imaging Graph. **29**(6), 463–475 (2005).

[85] T.-H. Lee, J. Lee, H. Lee, H. Kye, Y. G. Shin, and S. H. Kim, "Fast perspective volume ray casting method using GPU-based acceleration techniques for translucency rendering in 3D endoluminal CT colonography," Comput. Biol. Med. **39**(8), 657–666 (2009).

[86] A. Kruger, C. Kubisch, G. Strauss, and B. Preim, "Sinus endoscopy—Application of advanced GPU volume rendering for virtual endoscopy," IEEE Trans. Vis. Comput. Graph. **14**(6), 1491–1498 (2008).

[87] C. Kubisch, C. Tietjen, and B. Preim, "GPU-based smart visibility techniques for tumor surgery planning," Int. J. Comput. Assist. Radiol. Surg. **5**(6), 667–678 (2010).

[88] Z. Taylor, M. Cheng, and S. Ourselin, "High-speed nonlinear finite element analysis for surgical simulation using graphics processing units," IEEE Trans. Med. Imaging **27**(5), 650–663 (2008).