1 Introduction

Computational speedup is required in many areas. For example, in virtual reality, a fast rendering of virtual scenes is required to provide a more authentic and smoother experience,[1] whereas in artificial intelligence, a rapid processing rate is crucial for self-driving cars to detect and respond to dangers in a timely fashion.[2]

Optical measurement techniques also have an increasing demand for a high computation speed due to the pursuit of higher measurement resolutions and the fact that more complicated algorithms have been developed to gain higher accuracy and robustness. These computational burdens greatly restrict them to the study of dynamic phenomena or integration into real-time systems.[3] For example, with a typical computer and a typical algorithm parameter setting, digital image correlation (DIC) consumes >10 s to process a 1024×1024 image with a 5-pixel interval between two adjacent points of interest (POIs), whereas digital volume correlation (DVC) consumes >4 days to process a $1024 \times 1024 \times 1024$ volume.

1.1 Acceleration by parallel computing

Parallel computing has been applied to accelerate optical measurement techniques.[3,4] Different parallel platforms were analyzed and compared in Ref. 3, while the techniques accelerated by the most commonly used central processing units (CPUs) and graphics processing units (GPUs) in the past 5 years were reviewed in Ref. 4. A high parallelism can always be exploited in optical measurement techniques so long as calculations at measured points have a certain level of independence.

Generally, there are three types of parallelism: task parallelism, data parallelism, and pipelined parallelism. Task parallelism can be exploited from the typical operation system of a computer, in which the tasks or processes can be concurrently executed, each by one CPU core.

Data parallelism, conversely, is not concerned with how many tasks have to be performed. Instead, it concentrates on how the data are processed within one certain task. The GPU is designed for data parallelism. The basic idea of data parallelism can be illustrated by the vector addition example (i.e., $\mathbf{c} = \mathbf{a} + \mathbf{b}$, $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c} \in \mathbb{R}^N$) shown in Fig. 1, in which the simple task is addition. The sequential implementation of the vector addition, as shown in Fig. 1(a), is merely a loop across all the elements. Its data parallel implementation is shown in Fig. 1(b), where each set of corresponding elements of vectors **a** and **b** can be added in parallel simultaneously.

Pipelined parallelism is a unification of task parallelism and data parallelism, in which tasks are dependent on each other and different data are required to be processed simultaneously. In optical measurement, it is mainly used to realize

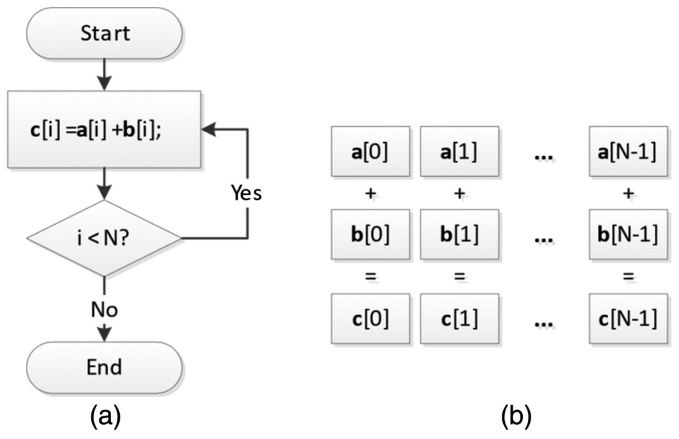

Figure 1 Schematic illustration of vector addition by (a) CPU implementation and (b) GPU parallel implementation.

high-speed systems. For example, tasks such as data acquisition, data processing, result display, and result saving can form a pipeline within which the computationally intensive data-processing part can be offloaded to the GPU.

Although the concept of parallel computing is simple, parallel programming may not be easy for optical researchers and engineers. This Spotlight provides a hands-on guide for how to apply parallel computing to accelerate optical measurement techniques. We focus on explaining the usage of GPUs because they are the most commonly employed parallel devices in optical measurement and bring the highest speedup ratios.

1.2 Graphics processing unit

GPUs were initially designed to process computationally intensive tasks for three-dimensional (3-D) graphics applications such as texturing a large set of polygons, calculating shading and lighting, and rendering the resulting two-dimensional (2-D) images on screen. Almost every modern computer has a GPU card. With the development of programmable shaders, the fixed GPU rendering pipeline can be manipulated to calculate different visual effects. Subsequently, GPUs began to be used as a general-purpose parallel computing device, and the so-called general-purpose GPU (GPGPU) programming based on shading languages emerged.

However, GPGPU programming at that time was very complex. Data in a shader should be represented as a set of polygon mesh vertices, but general-purpose data were difficult to represent in such a way. Even worse, shaders are usually tricky to debug, making it challenging to obtain the expected results without a very good understanding of computer graphics.

To make the GPGPU a real programmable device such as a CPU, many programming primitives have been proposed, for example, Cg, OpenCL, compute unified device architecture (CUDA), etc. Among them, CUDA, first released by NVIDIA in 2007, has become the mainstream choice by providing an easy-to-use C/C++ programming interface.

1.3 Heterogeneous computing with CUDA

From this section onward, to be consistent with CUDA, "host" and "device" are used as synonyms for CPU and GPU, respectively. CUDA C/C++ extends the C/C++ language and allows one to write device code in regular C/C++ with the extended functionalities and the predefined CUDA-specific variables. In addition, CUDA implements a heterogeneous computing architecture by providing its own "nvcc" compiler to compile the device code, while the host code is compiled by regular C/C++ compilers. The heterogeneous execution means that the device and the host can be scheduled to handle different tasks based on their specialized capabilities. As shown in Fig. 2, after the host starts to execute a process containing multiple tasks, it can offload the parallelizable tasks to the device by launching the corresponding CUDA kernels. A CUDA kernel is a special function that is launched by the host and executed on the device. The host then becomes idle, executes another portion of the process if it is independent to the device portion, or even executes other irrelevant tasks. This architecture provides CUDA with great portability and scalability[5] such that the program can be easily deployed on different platforms, and a higher speed can always be guaranteed for the platforms equipped with more powerful CUDA-capable GPUs.

The schematic in Fig. 2 also shows the common flowchart of a CUDA-accelerated application: the sequential code and the main logics are kept on the CPU side while the computationally demanding but parallelizable tasks are transferred to the GPU, after which the results are copied back to the CPU. The reason behind this workflow between the CPU and the GPU originates from their different design philosophies.

A CPU core is designed and optimized to be as "smart" as possible for sequential code execution by integrating low-latency arithmetic and logic units, multilevel caches, branch prediction units, etc. However, these optimizations trade in a large chip area and high power consumption, which limit the number of cores in the CPU. So far, even a high-end CPU can only contain 16 to 32 physical cores.

Contrary to the CPU, each GPU core is less smart and has higher latency. It is only good for very simple arithmetic operations and will get stuck when branches and loops are present. However, this seemingly foolish design gives room to a GPU chip so that it can comprise a great number of such cores (e.g., 2560 for an NVIDIA GTX 1080 GPU card), making the GPU superior in processing a huge amount of data processed by simple instructions. It is worth noting that, due to the high latency, a GPU can only achieve its peak performance when the problem size is large enough to fully utilize its computational resources.

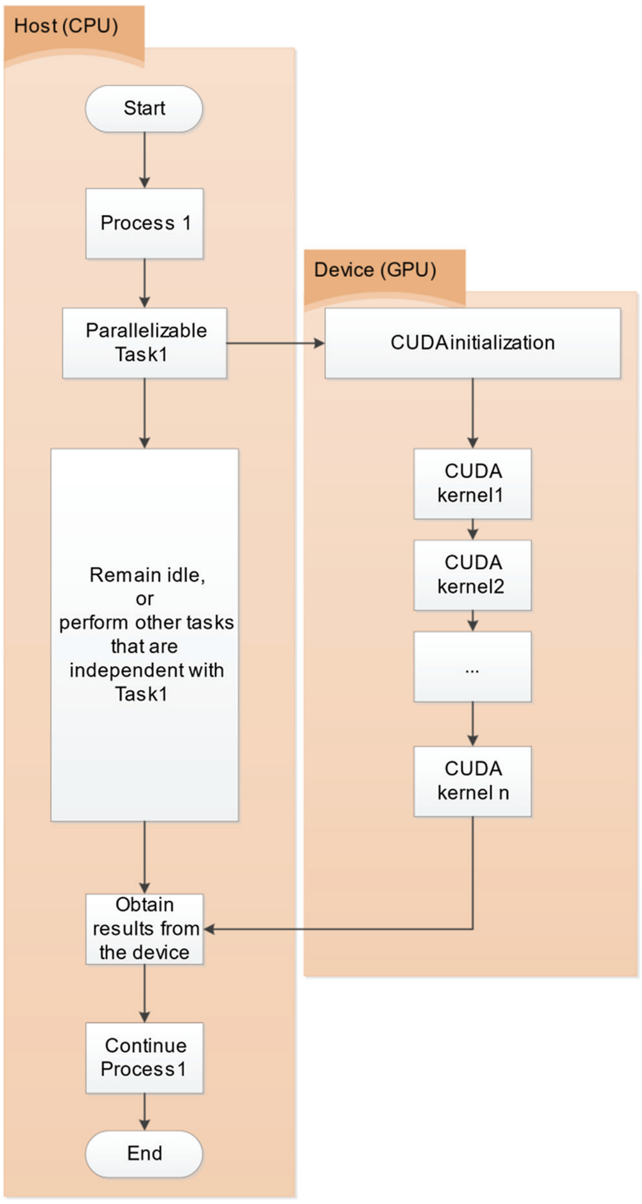

Figure 2 Schematic of the heterogeneous architecture of CUDA applied to compute a process that contains parallelizable tasks.

1.4 Common parallel patterns in optical measurement

In optical measurement, both data parallelism and pipeline parallelism are present. Such parallelism is the major reason for making the computation in optical measurement faster. We focus on data parallelism and its execution in a GPU, and very briefly introduce the concept of pipeline parallelism at the end of this Spotlight.

In practice, there are mainly four common data parallel patterns that can be observed from optical methods: pointwise, tiling, divide-and-conquer, and rendering and interpolation.

Pointwise pattern. Pointwise operations are the easiest data parallel pattern that can be extracted and parallelized. Many optical measurement techniques, although acquiring 2-D images or 3-D volumetric data, are essentially pointwise and thus contain the operations that should be performed on every individual point. The vector addition example in Fig. 1 is a typical pointwise pattern, in which its sequential CPU implementation is a *for* loop, but it can be executed in parallel on a GPU. In more complicated situations, however, pointwise loops may be contained within other outer loops. One may first consider flattening these loops because 2-D images or 3-D volumetric data can certainly be transformed into one-dimensional (1-D) arrays that can still be simply parallelized. The conversion method is detailed in Section 2.6.

Tiling pattern. Tiling is a variation of the pointwise pattern, which is typically used to efficiently deal with calculations at one point that should involve its neighboring points. Image gradient calculation, which is often used in optical measurement, can be considered as a point pattern (Section 2.7) and more efficiently as a tilted pattern (Section 2.8). Convolution, which is also commonly used, can be considered in the same way. As the name indicates, a whole dataset is first split into smaller data tiles (or blocks). However, to handle the boundary points of each tile, additional neighboring points should be included on each side of its boundary.

Divide-and-conquer pattern. The divide-and-conquer pattern is trickier to extract and parallelize than the previous two. It divides a large problem into smaller subproblems. After each subproblem is conquered, the original problem is solved. Typical divide-and-conquer algorithms in optical measurements include the fast Fourier transform (FFT) and a series of reduction algorithms.[6] Divide-and-conquer algorithms are always conceptualized and implemented as recursive processes. Although modern GPUs support recursion, its parallelism is not easy to extract. Even worse, a stack overflow may happen if the call depth of the recursion is too deep. Thus, in practice, we recommend converting recursions to iterations, which can then be solved or partially solved by the pointwise pattern. To avoid bugs, it is also preferred to consider employing existing libraries for parallelization of divide-and-conquer algorithms, for example, the CUFFT[7] for performing the FFT in CUDA.

Rendering and interpolation pattern. Fast rendering and interpolation, a sequence of processes that create 2-D images from 3-D scenes, are what GPUs were initially designed for in computer graphics. Many applications and systems in optical measurement require visualization of the results. If these results are computed on a GPU, they can be directly displayed following a graphics rendering pipeline without being transferred back to the CPU.

The first three parallel patterns will be used and explained in this Spotlight, whereas the last parallel pattern will be briefly mentioned at the end.

1.5 Use of the source code

All the source code accompanying this Spotlight can be downloaded or cloned from GitHub at https://github.com/TWANG006/GAOM.git. For Windows users, it is fairly easy to compile and play with the code: (1) install any version of Visual Studio higher than v14; (2) download and install CUDA[8] from https://developer.nvidia.com/cuda-downloads; (3) open "GAOM.sln" and choose to compile the code that you want to test. For Linux and Mac OSX users, you may also use the code by following the CUDA installation and compilation guide provided in http://docs.nvidia.com/cuda/index.html#installation-guides, and use the CUDA's "nvcc" compiler to compile the ".cu" files, while using the regular C/C++ compiler to compile the ".cpp" files. All of the experiments in this Spotlight are performed on a Dell Precision T3600 workstation equipped with an Intel® Xeon® CPU E5-1650 (6 cores, 3.20-GHz main frequency, 16.0 GB RAM) and an NVIDIA GeForce GTX 680 graphics card (8 SMs with 1536 CUDA cores and 2 GB 256-bit RAM). Note that we purposely choose a relatively low-end GPU for the experiments to show that GPU acceleration is generally affordable and achievable. However, the code is expected to run faster on higher-end GPUs. We have also done the experiements on an NVIDIA GeForce GTX 1080 GPU, which provide another 2+ times speedup.

1.6 Organization of the Spotlight

The rest of the Spotlight is organized as follows. Section 2 illustrates the basics of CUDA programming. A step-by-step image gradient calculation using CUDA is provided to demonstrate the implementation and optimization of a CUDA program. Sections 3 and 4 provide two example optical measurement techniques that can be highly accelerated using CUDA: the 2-D windowed Fourier filtering (WFF2) algorithm used in fringe pattern analysis and the inverse-compositional Gauss–Newton (IC-GN) algorithm used in subpixel DIC. Section 5 provides suggestions for further reading and practice.

2 CUDA Basics for GPU Programming

This section illustrates the basics of the CUDA programming model with a simple parallel vector addition program. Image gradient calculation, which is widely used in optical measurement methods, is then implemented using CUDA with detailed explanations. It is first implemented using the pointwise pattern and then optimized by the tiling pattern to help the readers quickly grasp the key knowledge and skills required to start accelerating their optical applications with CUDA.