A Performance Model for GPU Architectures that Considers On-Chip Resources: Application to Medical Image Registration

Junhao wu, Xuan Yang, Zhengrui Zhang, Guoliang Chen, and Rui Mao

Abstract—Graphics processing units (GPUs) have become extremely important devices for accelerating computing performance in many applications. However, there have been few accurate models to estimate the performance of such applications running on modern GPUs. In this paper, we propose a performance model to estimate the execution times for massively parallel programs running on NVIDIA GPUs, one that takes on-chip resources and cost of data transfer between CPU and GPU into consideration. Four different GPUs with different architectures were used to evaluate our model. We demonstrated the effectiveness of the proposed model by applying it to various tasks in medical image registration. Experiments have demonstrated that by capturing on-chip GPU resources and data transfer time with our model, we were able to obtain a more accurate prediction of the actual running time, compared to the traditional model. Moreover, by using the optimal value of the block size parameter, estimated by our model, to accelerate the landmark tracking task on GPU devices, speedups of approximately 80×, 100×, 200× and 800×, on the C2050, K20c, M5000 and P100 can be achieved, making it possible to track massive numbers of landmarks and thereby improving the registration accuracy.

Index Terms—Performance model, graphics processing unit, on-chip resources, medical image registration

1 Introduction

Over the last few decades, the performance and capabilities of graphics processing units (GPUs) have increased remarkably. They have evolved to become highly parallel processors with tremendous computational power and high memory bandwidth [1]. At present, GPUs are one of the most important components of high-performance computing (HPC) systems. High-performance computing with GPU platforms has gained in popularity for scientific research since its emergence. By launching thousands of threads to utilize a large number of cores, GPU programs achieve significant speedups over a single-threaded program executing on a CPU.

Applications for GPUs are generally written using software development tools provided by GPU device manufacturers. For NVIDIA GPUs, NVIDIA provides a parallel computing platform and programming model called CUDA (Compute Unified Device Architecture) [2] that allows programmers to use these massively parallel architectures for many applications, such as linear algebra [3], [4], neural data analysis [5], [6], [7], and medical image registration [8], [9], [10]. However, even though the CUDA programming model is user-friendly, identifying program bottlenecks and estimating the benefits of potential optimizations using GPUs is still complicated by several factors, including resource contention and the unique GPU memory model. Therefore, to achieve optimal performance for a scientific application on a GPU platform, programmers might need to try all combinations of the variables to find the best configurations.

Thus, there is a significant need for methodologies designed for predicting the performance of GPU applications to use them efficiently. In the last decade, a number of studies have explored this issue. Lopez-Novoa et al. [11] conducted a survey of GPU performance modeling, which classified existing performance models for GPUs into four classes based on the output generated by the model: execution time estimation [4], [12], [13], [14], [15], [16], [17], [18], [19], bottleneck identification [20], [21], [22], [23], power consumption estimation [24], [25], [26], and simulation [27], [28], [29]. The execution time estimation model is the most commonly used performance model; it aims to predict the execution time of a parallel application on GPU platforms by considering various characteristics of the GPUs, such as the number of cores, memory latency, memory access conflicts, cost of computing, and scheduling. The most cited one is the Hong and Kim's model [12], which estimates the actual running time of a program based on memory-level and thread-level parallelism. Kothapalli et al. [13] proposed model estimates the relationships between the various components of the NVIDIA GPU architecture such as the number of cores, effects of memory latency, and memory access conflict. Some execution time models use program skeletons to estimate the potential performance when an application is run on GPUs [15], [16].

- *J. Wu, X. Yang, G. Chen, and R. Mao are with the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, Guangdong 518000, China.*
E-mail: jacobwuu@foxmail.com, {yangxuan, glchen, mao}@szu.edu.cn.
- *Z. Zhang is with the College of Information Engineering, Shenzhen University, Shenzhen, Guangdong 518000, China. E-mail: 437028792@qq.com.*

Manuscript received 28 Mar. 2018, revised 9 Mar. 2019, accepted 11 Mar. 2019, Date of publication 15 Mar. 2019; date of current version 7 Aug. 2019.
(Corresponding author: Junhao Wu.)
Recommended for acceptance by Z. Chen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2019.2905213

1045-9219 © 2019 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Moreover, Machine-learning techniques can also be used to train data to predict the execution time of a program running on GPUs [17], [18]. Execution time prediction models for specific purposes have also been proposed [4], [19]. However, none of these execution time estimation models has attempted to take into consideration the on-chip GPU resources, such as the number of registers and the amount of shared memory. When programs are run on a GPU, each streaming multiprocessor (SM) can execute multiple active warps in a time-sharing fashion while one or more warps are waiting for memory data. The parallelism with which warps access memory simultaneously plays an important role in implementation performance. To maximize the improvement in the performance of parallel programs running on GPUs, the warp parallelism should be as high as possible. However, the warp parallelism is determined by the number of active warps, which is limited by the GPUs' on-chip resources. Although the number of active warps is included in Hong and Kim's model, the influence of active warps is counteracted in the model when the execution time is dominated by memory access cost. The result is that the estimated running time using Hong and Kim's model is not sensitive to the usage of on-chip resources, and thus their model cannot accurately predict implementation performance.

In this paper, we propose a novel analytical model to estimate the performance of massively parallel programs running on NVIDIA GPUs. We extend Hong and Kim's model [12] to take on-chip GPU resources into consideration. To our knowledge, this is the first work to propose an execution time estimation model that considers on-chip resources of the GPU. Moreover, data transfers between the CPU and GPU affect the performance of a GPU program under normal usage [15], [30]; Gregg and Hazelwood [31] demonstrated that data transfers between CPU and GPU can influence the reported GPU performance and argued that the reported GPU speedup should include the cost of data transfer. For this reason, we also take the cost of data transfer into consideration in our model to obtain a more accurate prediction of the GPU performance.

To validate our performance model, we implemented the following tasks, which related to medical image registration, on GPU platforms, including landmark tracking, spatial transformation, and image interpolation. Image registration is the process of transforming different sets of data into one coordinate system by estimating an optimal transformation between different images. However, many of the existing image registration methods are computationally expensive, and their registration accuracy is limited by the high computation cost. GPUs provide a parallel computing platform for massive data processing, making it possible to accelerate registration processing. Medical image registration by means of parallel processing on GPU platforms has been proposed [8], [9], [32], [33], [34], [35]. An accurate performance model can help take full advantage of the high performance of GPUs and eliminate the optimization barrier of image registration caused by the high computation cost of a single-threaded program running on CPUs. For example, the number of landmarks tracked by search algorithms can be increased considerably as a benefit of efficient implementation on GPUs, resulting in improved registration accuracy for methods based on landmark tracking.

Using our performance model, optimal implementation performance was achieved by predicting performance given different configuration parameters. For landmark tracking, thousands of landmarks in 4D CT lung images were tracked using a structure tensor tracking algorithm [36]. Experiments showed that less than 1 s of running time is required on the NVIDIA P100, and 800× speedups are achieved compared with the single-thread program running on a Xeon E5-2620 v3 CPU. Moreover, more accurate registration results were obtained compared to existing algorithms because of a massive number of landmarks was tracked. For spatial transformation, 3D volume data were deformed by a given transformation, and experiments showed that 150× speedups are achieved on the NVIDIA C2050, 400× on the NVIDIA K20c, 500× on the NVIDIA M5000 and 1500× on the NVIDIA P100. For image interpolation, deformed 3D volumes were obtained with greater than 50× speedups on the NVIDIA C2050, 60× speedups on the NVIDIA K20c, 70× speedups on the NVIDIA M5000 and 250× speedups on the NVIDIA P100.

The contributions of this paper are as follows. (1) We developed a performance model that consists of the computation cost and memory cost at the warp level and takes into consideration on-chip GPU resources and the cost of data transfer between CPU and GPU. The execution times estimated using the proposed model are closer to the actual execution times on different GPUs compared to the existing model. Our model can be used to predict the execution performance of a program as implemented on GPUs with different architectures. (2) Using the performance model, we estimated the optimal value of the block size parameter for implementing landmark tracking, spatial transformation, and image interpolation for medical image registration. Using the optimal parameter value, we optimized the implementation of a landmark tracking algorithm, spatial transformation, and image interpolation on four different GPU platforms and achieved approximately 80–800×, 150–1500×, and 50–250× speedups, respectively, over a single-thread program running on a CPU for these tasks. (3) Benefiting from the optimized GPU implementation, massive numbers of landmarks were tracked using GPUs, and more accurate registration results were thereby obtained for 4D CT lung images.

The remainder of this paper is structured as follows: Section 2 reviews related work. Section 3 briefly introduces the architecture of GPUs. Section 4 describes our performance model in detail. Results of various kinds of experiments are provided in Section 5 and validate the proposed model for various individual tasks. Finally, Section 6 provides conclusions.

2 Related Work

To estimate the performance of GPU-based applications, many performance prediction models have been proposed. As our proposed model belongs to the category of execution time estimation models, we provide details of the past work related to this kind of performance model in this section.

Hong and Kim [12] proposed a performance model for the GPU architecture using two key metrics: memory warp parallelism (MWP) and computation warp parallelism (CWP). The first one, MWP, estimates the number of parallel memory requests that can be executed concurrently and is calculated by considering the number of running warps and the memory bandwidth. The second, CWP, represents the number of warps that can execute instructions, while one warp is waiting for memory data. Using both MWP and CWP, the memory cost and computation cost required to run a kernel can be estimated. Kothapalli et al. [13] proposed a model outlining the relationship between the various components of the NVIDIA GPU architecture, including the number of cores, effects of memory latency, memory access conflicts, cost of computing, scheduling, and pipelining. This model can be used to analyze pseudocode for CUDA and predict the performance. Baghsorkhi et al. [14] used a control flow graph to represent the relationship between instructions, where nodes represent instructions and arcs represent latencies. They proposed an analytical model based on a work flow graph (WFG) to predict the effect of control flow divergence and the memory hierarchy on the performance of a GPU application.

Instead of using the source code to predict the execution time, Meng et al. [16] proposed a GPU performance projection framework on CPU code skeletons that can estimate the performance benefit of GPU acceleration without needing any actual GPU programming or hardware. Boyer [15] extended Meng et al.'s framework by accounting for the data transfer time between CPU and GPU.

The functional performance model is another type of performance model; it represents the processor speed as a function of problem size [37]. The speed is defined as the number of computation units performed by the processor per unit of time. Data partitioning algorithms based on functional performance models are designed to balance the computational workload of heterogeneous multiprocessor systems for data-parallel applications. Zhong et al. [38] used functional performance models to optimally partition the workload of data-parallel applications on heterogeneous multiprocessor systems with multicore and multi-GPU platforms. They modeled a multicore and multi-GPU system as a number of abstract processors to built functional performance models to measure their performance.

As an alternative to evaluating a number of predefined formulas and then reporting the execution time quickly, machine-learning (ML) techniques can be used to train data to learn the relationship between program features and execution time [17], [18], [39]. Che et al. [17] proposed a model based on machine-learning techniques, which can obtain a very high accuracy level when used to predict the execution times of applications on the GPUs used to build the model. However, accuracy is decreased when execution time for a new GPU is estimated. Kerr et al. [39] used the polynomial form of linear regression to determine a relationship between static program metrics and the total execution time of an application. Dao et al. [18] proposed two GPU performance models—a sampling-based linear model and a model based on ML techniques—that are applicable to modern GPUs with and without caches. The drawback of these machine-learning-based approaches, however, is that the process used to calibrate the model has not been described in sufficient detail [11].

On the contrary, models for specific programs have also been proposed. For example, Li et al. [19] analyzed the performance of sparse matrix multiplication on a GPU. Their performance model is based on probability mass function, which fully reflects the distribution of nonzero elements in a sparse matrix. When this is combined with the hardware parameters of the GPU, the performance of matrix multiplication can be estimated. Chen et al. [4] developed a hybrid parallel lower-upper (LU) factorization approach combining task-level and data-level parallelism on GPUs. A parametric performance model was presented to analyze the bottlenecks of the proposed LU factorization approach.

However, existing models do not characterize the influence of on-chip GPU resources, which are related to the number of warps running concurrently on an SM and the parameter block size. In this work, we focus on a performance model of GPUs that considers on-chip resources to accurately predict the performance.

3 Brief Introduction of GPU Architecture

GPUs are capable of accelerating scientific computing because of their parallel computing ability. The GPU is designed for calculation, especially for floating point calculation, without redundant and complex logical control units. CUDA makes it possible to solve many complex computational problems in a more efficient way. CUDA comes with a software environment that allows developers to use C as a high-level programming language to implement parallel computing conveniently. In CUDA, a kernel function is called by the host and then executes on the GPU. A kernel is executed as a grid, which is composed of blocks of threads. Each block is completely independent. A block is composed of threads that can communicate within this block. Thirty-two threads in the same block form a warp, which is executed physically in parallel on a streaming multiprocessor in the GPU.

The architecture of a typical GPU has a set of SMs and a DRAM, referred to as global memory, which can be accessed by any SM. Each SM contains several streaming processors (SPs) and other on-chip resources, including registers, shared memory, constant cache, and texture cache. In the CUDA parallel programming model, each thread has a per-thread private memory space used for register spills, function calls, and automatic array variables. Each thread block has a per-block shared memory space used for inter-thread communication, data sharing, and result sharing in parallel algorithms [40]. Data in shared memory can be accessed by threads in the same block.

Registers are on-chip resources with the highest access speed and the lowest latency. Variables declared in a kernel without any qualifier automatically reside in registers, except for arrays. The shared memory is also on-chip, and the latency to access shared memory is low, tens of cycles. However, accessing global memory can take hundreds of cycles. GPUs hide such latency for global memory access by having multiple warps running concurrently, called active warps, on each SM. Whenever a warp is suspended pending completion of its memory access, an SM can switch to another warp and continue execution. On early devices, the latency to access global memory can vary according to data access patterns. Devices with compute capability 2.0 or higher, however, such as the Tesla C2050, have an L1 cache with a 128-byte line size in each multiprocessor, which coalesces global memory accesses by threads in a warp into as

TABLE 1
Summary of Model Parameters (“conf” Represents Configuration; “code analysis” Refers to CUDA Source Code Analysis)

Model parameter	Definition	Obtained from
Comp_cycles	Cost of computation for a warp	Code analysis
Comu_bytes	Number of bytes transferred between host and device	Code analysis
Comu_BW	Data transfer bandwidth between host and device	Machine conf
Departure_delay	Delay between two global memory transactions	Machine conf
Freq	Clock frequency of the streaming multiprocessors (SMs)	Machine conf
Max_B	Maximum number of active blocks in each SM	Machine conf
Max_R	Maximum number of registers in each SM	Machine conf
Max_W	Maximum number of active warps in each SM	Machine conf
Max_SMem	Shared memory allocation size per SM	Code analysis
Mem_insts_global	Number of global memory access instructions in one thread	Code analysis
Mem_insts_local	Number of local memory access instructions in one thread	Code analysis
Mem_insts_shared	Number of shared memory access instructions in one thread	Code analysis
Mem_L_global	Global memory access latency	Machine conf
Mem_L_shared	Shared memory access latency	Machine conf
N_{b}	Total number of blocks running on the GPU	Code analysis
N_{conf}	Average bank conflict per shared memory access instruction	Code analysis
N_{sm}	Number of SMs on GPU chip	Machine conf
N_{ws}	Number of warp schedulers in each SM	Machine conf
Reg_per_thread	Number of registers used by each thread	Code analysis
SMem_unit	Shared memory allocation unit size	Machine conf
SMem_used	Shared memory to be used by each block	Code analysis
Sync_insts	Number of synchronization instructions in one thread	Code analysis
Threads_per_block	Number of threads per block	Code analysis
Warp_size	Number of threads per warp	Machine conf

few cache lines as possible, resulting in a negligible effect of alignment on throughput for sequential memory accesses across threads [41].

4 Proposed Performance Model for GPUs

To estimate the cost of a GPU implementation strategy, we propose a performance model that accounts for on-chip resources. Hong and Kim [12] proposed an analytical model for the GPU architecture by using memory warp parallelism and computation warp parallelism. MWP is used to estimate the maximum number of warps that can access memory concurrently during one memory access period, and CWP is used to estimate the number of computations that can be performed during one memory access period. However, this analytical model does not consider the limitation of on-chip GPU resources in MWP, such as registers and shared memory; this results in estimated execution times that are mostly the same even though the usage of on-chip resources are different. We extend Hong and Kim’s model by considering on-chip resources to handle this issue.

Our aim is to fully use GPU resources and accurately predict the implementation performance of different parallel computation tasks. In our performance model, the memory access cost and the computation cost are considered. The memory cost is evaluated by estimating the execution time of memory access and the exact number of active warps executing concurrently. Suppose the total time required for memory access is t; the number of active warps k_w determines the number of warps that are able to access memory concurrently, and thus the real memory access time is proportional to t/k_w when the memory bandwidth is sufficient. This means that the more active warps there are, the lower is the memory access time. As we know, the on-chip resources on GPUs limit the number of active warps, which in turn influences the memory access cost. Therefore, it is necessary to take the number of active warps as a factor in predicting the memory access time.

For computation cost, on the other hand, we take the number of warp schedulers on GPUs into account. The more warp schedulers there are, the greater the number of active warps that can be scheduled, which implies that more threads can be executed concurrently in the same clock cycle when resources on the GPU are sufficient for these threads. The number of warp schedulers is different for various GPU architectures, such as the Maxwell [42] architecture and the Pascal [43] architecture. Therefore, it is reasonable to consider the number of warp schedulers as an influencing factor. Moreover, we also take data transfer time between CPU and GPU into consideration. Table 1 lists the parameters used in our performance model.

4.1 Cost of Computation

CUDA splits problems into grids of blocks, each containing multiple threads. Blocks are allocated to any SM that has free slots. The SM schedules threads in each block in groups of 32 parallel threads, called warps, and each warp is scheduled by a warp scheduler for execution. Threads in the same warp are executed concurrently. Hence, calculating the cost of computation for a warp is equivalent to calculating the cost of computation for a thread. For a thread, the key in computation cost prediction is to estimate the number of instruction cycles for each arithmetic operation.

Hong and Kim used the number of parallel thread execution (PTX) [2] instructions to calculate the cost of computation and assumed that each PTX instruction translates to one native binary microinstruction. However, some PTX instructions, such as the *sqrt* instruction, are expanded as multiple binary instructions. Furthermore, as a low-level instruction set architecture, PTX has poor readability. For these reasons, we use a different method to calculate the cost of computation. First, we use the kernel code to count the number of each type of arithmetic operation, such as multiplication operations or division operations. Then, the latency of each type of arithmetic operation is estimated using microbenchmarks. The details of the method used to estimate the latencies are provided in [44]. Next, the execution time of each arithmetic operation is represented by its specific latency. Finally, the cost of computation for a warp, denoted as $Comp\_cycles$ and representing the total execution time for all arithmetic operations, is calculated by summing the execution time of each arithmetic operation.

4.2 Cost of Memory Access

Benefiting from the hundreds of SPs on GPUs, the time to execute all instructions of a warp is cheap. However, it is expensive to access data from memory. When a warp issues a memory fetch request, it spends many clock cycles waiting for the requested data. During this period, all threads in this warp are suspended. Global memory is the slowest memory in GPUs and is thus the performance bottleneck for parallel implementation on GPUs. Shared memory is an on-chip memory resource with a low access cost. However, the size of shared memory is very limited. Local memory is used to store local variables spilled from the register. Details of register spilling will be introduced later. In this section, we estimate the cost of memory access at the warp level by modeling these three types of memory access.

4.2.1 Cost of Global Memory Access

On devices with compute capability 2.0 or higher, the device coalesces global memory accesses by threads in a warp into as few transactions as possible, resulting in a negligible effect of alignment on throughput for sequential memory accesses across threads [41]. For this reason, there is no need to consider misaligned data access in global memory. The cost of global memory accesses by each warp is simplified as

$$Mem\_cycles\_global = Mem\_L\_global \times Mem\_insts\_global, \tag{1}$$

where Mem_L_global is the global memory access latency, and Mem_insts_global is the number of global memory access instructions in each thread, which can be counted easily using the kernel code.

4.2.2 Cost of Shared Memory Access

For devices with compute capability 2.0 or higher, shared memory is arranged in 32 banks that are 32 bits wide. Band conflicts occur when addresses requested by multiple threads are mapped to the same memory bank. If n threads within a warp cause a bank conflict, then n memory accesses are executed serially, resulting in a factor-of-n slowdown on the performance of shared memory access. However, if all threads in a warp access the same memory address, a broadcast is performed and no bank conflict occurs, and thus only one memory transaction is needed. Let N_{conf} be the average number of conflicts per shared memory access instruction issued. Then the cost of shared memory access for each warp is

$$\begin{aligned} Mem\_cycles\_shared = {} & Mem\_insts\_shared \\ & \times N_{\text{conf}} \times Mem\_L\_shared, \end{aligned} \tag{2}$$

where Mem_L_shared is the shared memory access latency, and Mem_insts_shared is the number of shared memory access instructions in a thread. Note that when n threads access the same shared memory bank, then $N_{\text{conf}} = n$. An exception is a case in which all threads in a warp access the same shared memory address; in this case, $N_{\text{conf}} = 1$. For a specific kernel, we can easily count the number of shared memory access instructions and then analyze the number of bank conflicts that will occur, thereby obtaining the values of Mem_insts_shared and N_{conf}.

4.2.3 Cost of Local Memory Access

In most cases, local variables in a kernel are held in registers, which consume no clock cycles to access. However, there are three cases for which the compiler will place variables in local memory (called register spilling):

1) Arrays whose indexes cannot be determined to be constant quantities;
2) Large structures or arrays that cannot be held in registers;
3) Any variables that would require more registers than the kernel has available.

Local memory is an abstraction to the scope of a thread; it does not exist physically. Devices with compute capability 2.0 or higher spill registers to the L1 cache, and older devices spill registers to global memory. Hence, in our performance model, the cost of accessing local memory is as high as that for accessing the L1 cache. This means that the latency of accessing local memory is the same as the latency of accessing shared memory without bank conflict and broadcasting. The cost of local memory access for each warp is

$$\begin{aligned} Mem\_cycles\_local = {} & Mem\_L\_shared \\ & \times Warp\_size \times Mem\_insts\_local, \end{aligned} \tag{3}$$

where Mem_insts_local is the number of local memory access instructions executed by each thread, which is obtained by running the compiler report with the ptxas option [2].

4.3 Real Warp Parallelism

The CUDA parallel programming model guides the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads. Each sub-problem is partitioned into finer pieces that can be solved cooperatively in parallel by all threads within a block [2]. At the chip level, a global work distribution engine schedules thread blocks to various SMs, while at the SM level, each warp scheduler distributes warps of 32 threads to its execution units [40]. The more the number of warp schedulers, the more the number of warps that can be issued and executed concurrently. For example, in the Fermi architecture, such as that of NVIDIA Tesla C2050, each SM features two warp schedulers, which allows two warps to be issued and executed concurrently [40].

We use real warp parallelism (RWP) to represent the exact number of warps per SM that can execute concurrently. The RWP values for computation and memory access are separate in our model and are denoted as RWP_C and RWP_M, respectively. The RWP for computation is equal to the number of warp schedulers in each SM, denoted by N_{ws}. However, when the number of warps in a block is less than the number of warp schedulers, which means there are not enough warps in a block to be scheduled, RWP_C is less than N_{ws}. The RWP for computation is formulated as

$$RWP\_C = MIN\left\{\frac{Threads\_per\_block}{Warp\_size}, N_{\mathrm{ws}}\right\}. \tag{4}$$

Furthermore, when a warp issues memory requests, it will be suspended until all of the memory requests are served. During this period, the SM can switch to another active warp so that all computer resources are kept busy. Consider the delay between two memory transactions, denoted as $Departure\_delay$. There are $Mem\_L\_global/Departure\_delay$ warps that can access memory concurrently, where Mem_L_global is the global memory latency. However, the number of active warps is also influenced by the resources of each SM. The more resources each warps occupied, the fewer warps that can be executed concurrently. For memory access, we extend the memory warp parallelism proposed by Hong and Kim [12] to take on-chip resources into consideration and define the RWP_M as follows:

$$RWP\_M = MIN\left\{\frac{Mem\_L\_global}{Departure\_delay}, Warps\_active\right\}, \tag{5}$$

where the number of active warps per SM, $Warps\_active$, is dependent on the number of active blocks per SM, $Blocks\_active$, and the block size, $Threads\_per\_block$, and is calculated as

$$Warps\_active = Blocks\_active \times \frac{Threads\_per\_block}{Warp\_size}, \tag{6}$$

where the number of active blocks $Blocks\_active$ is constrained by on-chip resources. Limited by on-chip resources, each SM can accept up to 8 to 32 blocks running concurrently, depending on the GPU architecture. When a block is finished, resources used by that block become free. These resources will then be reallocated to a new block to make it active. For this reason, the block size is a tradeoff between the number of active blocks and the amount of on-chip resources used by each block. This means that the more resources each block occupies, the fewer blocks can run concurrently in each SM. Although Hong and Kim provided the definition of $Blocks\_active$, their model is not sensitive to it, and no method for obtaining the value of $Blocks\_active$ was provided. Here, we propose an approach for estimating $Blocks\_active$ based on SM resources, and this can be used to optimize GPU implementation. For a specific number of threads per block, $Blocks\_active$ is determined by the limitation of on-chip resources, including registers and shared memory. The relationship between $Blocks\_active$ and on-chip resources is

$$Blocks\_active = MIN\{Max\_B, Max\_B\_W, Max\_B\_R, Max\_B\_S\}, \tag{7}$$

$$Max\_B\_W = \frac{Max\_W \times Warp\_size}{Threads\_per\_block}, \tag{8}$$

$$Max\_B\_R = \frac{Max\_R}{Threads\_per\_block \times Reg\_per\_thread}, \tag{9}$$

$$Max\_B\_S = \frac{Max\_SMem}{SMem\_per\_block}, \tag{10}$$

$$SMem\_per\_block = \left\lceil \frac{SMem\_used}{SMem\_unit} \right\rceil \times SMem\_unit, \tag{11}$$

where Max_B is the maximum number of blocks per SM, which is determined by the compute capability of the GPUs; Max_B_W is the maximum number of blocks allowed in an SM constrained by the maximum number of warps per SM Max_W, and $Threads\_per\_block/Warp\_size$ calculates the number of warps in a block; Max_B_R is the maximum number of blocks allowed by available registers, Max_R is the number of available registers provided by each SM, and $Threads\_per\_block \times Reg\_per\_thread$ is the number of registers requested by each thread block, where the value of Reg_per_thread can be obtained from the compiler reports of the kernel run with the ptxas option [2]; Max_B_S is the maximum number of blocks tolerated by shared memory, Max_SMem is the size of shared memory provided by each SM, and $SMem\_per\_block$ is the size of shared memory occupied by each block; $SMem\_used$ is the amount of shared memory requested by each block and $SMem\_unit$ is the size of the shared memory allocation unit. Note that the programmer can configure the size of shared memory Max_SMem to be 16 KB or 48 KB for devices with the Fermi or Kepler architecture, while the size of shared memory is 96 KB and 64 KB for devices with the Maxwell and the Pascal architecture, respectively; clearly, $SMem\_used$ can also be declared by the programmer in the kernel.

Besides, the value of memory warp parallelism in Hong and Kim's model is also limited by the memory bandwidth and the bandwidth occupied by all concurrently running warps, denoted as MWP_peak_BW. We eliminate the limitation of MWP_peak_BW, as we found that it has a negative influence on the model. Details are given in Section 5.3.

Note that in most cases, the number of active warps does not influence computation-intensive tasks, whose execution time is dominated by computation. For computation-intensive tasks, the cost of arithmetic instructions is much more than the cost of memory access and the memory access latency is hidden during execution of arithmetic instructions. In these circumstances, the number of active warps is much greater than the number that the warp schedulers can schedule. Therefore, for computation-intensive tasks, it is not necessary to consider the number of active warps. Memory-intensive tasks, on the other hand, whose execution time is dominated by memory access, are not affected by the number of warp schedulers. For these tasks, the cost of memory access is greater than the cost of computation and the number of active warps must be sufficient to hide the memory latency time. The more the number of active warps, the more memory latency can be hidden. Therefore, the memory-intensive tasks are heavily influenced by the number of active warps but not by the number of warp schedulers because there are no delays in switching between active warps [2].

It is observed that the number of active blocks defined in Eq. (7) is related to the resources in an SM, such as the available registers and shared memory, a factor that is not provided in [12]. To optimize the programming implementation on GPUs, the resources requested by a program and the available resources provided by GPUs should be balanced. The key contribution of our paper is to provide an estimate of the number of active blocks, which will make the performance prediction more accurate. Furthermore, an optimized implementation can be obtained using our performance model.

4.4 Cost of Synchronization

Typically, all of the threads in a block are executed asynchronously. However, a barrier is needed when some threads need to synchronize to share data with each other. In the CUDA programming model, programmers can specify synchronization points in the kernel by calling the $\_syncthreads()$ intrinsic function. This function acts as a barrier at which all threads in the block must wait before any of them is allowed to proceed [2]. In [12], Hong and Kim demonstrated that the additional delay per synchronization instruction in a block is the product of $Departure\_delay$ and $RWP\_M - 1$. Thus, the cost of synchronization for each block is calculated as

$$Sync\_block = Departure\_delay \times (RWP\_M - 1) \times Sync\_insts, \tag{12}$$

where $Sync\_insts$ is the number of synchronization instructions in one thread. We divide $Sync\_block$ by the number of warps in a block and obtain the cost of synchronization at the warp-level as

$$Sync\_cycles = \frac{Sync\_block \times Warp\_size}{Threads\_per\_block}. \tag{13}$$

4.5 Total Execution Time

As illustrated above, we separate memory access and computations as far as a single warp is concerned. As there are multiple warps executing concurrently on GPUs, we take the real warp parallelism into consideration and extend the definition of cycles per instruction (CPI) from the instruction level to the warp level, as cycles per warp (CPW). CPW is used to measure the average cost of computation and the average cost of memory access for a program at the warp level, which are denoted as CPW_C and CPW_M, respectively. CPW_C is related to the cost of computation and the number of warps that can execute computation concurrently, and CPW_M is related to the cost of memory access and the number of warps that can access memory concurrently. The values of these two metrics are calculated as

$$CPW\_C = \frac{Comp\_cycles}{RWP\_C}, \tag{14}$$

$$CPW\_M = \frac{Mem\_cycles}{RWP\_M}, \tag{15}$$

where Mem_cycles is composed of the cycles used by global memory, shared memory, and local memory, as follows:

$$Mem\_cycles = Mem\_cycles\_global + Mem\_cycles\_shared + Mem\_cycles\_local. \tag{16}$$

When $CPW\_C > CPW\_M$, the total execution time is dominated by computation; otherwise, the total execution time is dominated by memory access. By considering computation cycles, memory access cycles, and synchronization cycles together, we can calculate the average number of cycles to execute a warp as

$$Exec\_cycles\_warp = MAX\{CPW\_C, CPW\_M\} + Sync\_cycles. \tag{17}$$

Moreover, a multithread program is partitioned into blocks of threads that execute independently of each other. Blocks are scheduled to execute on SMs, and each block must execute from start to finish on one SM. Let N_b be the total number of blocks running on a GPU and N_blocks be the number of blocks to be executed in each SM. Then the number of warps to be executed in each SM is

$$N\_warps = N\_blocks \times \frac{Threads\_per\_block}{Warp\_size}, \tag{18}$$

where

$$N\_blocks = \left\lceil \frac{N_b}{N_{sm}} \right\rceil, \tag{19}$$

and thus the total number of cycles to execute all warps in each SM is

$$Exec\_cycles\_SM = N\_warps \times Exec\_cycles\_warp. \tag{20}$$

As all SMs on a GPU are running concurrently, the total execution time for a GPU is the same as the execution time of one SM. To transform the required number of cycles into units of time (seconds), Eq. (20) is divided by the clock rate of the GPU core as follows:

$$Exec\_times\_GPU = \frac{Exec\_cycles\_SM}{Freq}. \tag{21}$$

To consider the time for data transfer between host and device, denoted as T_c, we rewrite Eq. (21) as

$$Exec\_times\_GPU = \frac{Exec\_cycles\_SM}{Freq} + T_c, \tag{22}$$

where

$$T_c = \frac{Comu\_bytes}{Comu\_BW}, \tag{23}$$

in which $Comu\_bytes$ is the number of bytes transferred between CPU and GPU, whose value is known for a given task, and $Comu\_BW$ is the memory bandwidth between CPU and GPU, whose value is estimated using microbenchmarks.

For a specific implementation on a particular GPU platform, N_b is constant. There is then a proportional relationship between $Exec\_cycles\_warp$ and the execution time of our parallel program; see Eq. (20). The value of $Exec\_cycles\_warp$

is inversely proportional to the real warp parallelism, RWP_C and RWP_M; see Eqs. (17), (14), and (15). For computation-intensive tasks, RWP_C is limited by $Threads\_per\_block$ and the number of warp schedulers N_{ws} (Eq. (4)); for memory-intensive tasks, as $Mem\_L\_global/Departure\_delay$ is constant, the value of RWP_M is determined by the number of active warps $Warps\_active$ (Eq. (5)). Since $Warps\_active$ is determined by $Blocks\_active$, which is also related to $Threads\_per\_block$, the total execution time of GPUs in our performance model is indeed a cost function based on the parameter $Threads\_per\_block$. By selecting an optimal $Threads\_per\_block$ value, the total cost of implementation on GPUs can be minimized, which means an optimal implementation strategy can be obtained.

5 Experimental Evaluation

Three tasks used for medical image registration were implemented to evaluate the performance of our proposed model. The first is the image registration based on a landmark tracking algorithm, the second is the calculation of a spatial transformation for a 3D image using a given transformation function, and the third is the trilinear interpolation. In our implementation, the landmark tracking task is memory-intensive owing to its frequent accessing of global memory. For spatial transformation, two different kernels were implemented to evaluate the accuracy of our model: one that accesses global memory, and another that accesses shared memory, which we used to demonstrate the influence of on-chip resources on the execution time. Since the trilinear interpolation task uses very few on-chip resources, we used this task to validate the performance of our model for those tasks that are not influenced by on-chip resources.

To calculate the speedup of our GPU parallel implementation, the execution time for a single-threaded implementation running on CPU was used to compare with the execution time of the multi-threaded implementation running on GPUs. The single-threaded program was written using the C programming language and was compiled with the Microsoft Visual Studio C++ compiler 11.0 without any acceleration. For comparison, only the execution time for the portion of the program that was run both on the CPU and on the GPUs was used to calculate the speedup.

5.1 Experiment Settings

To evaluate the performance of the proposed model, four different GPU platforms, the NVIDIA Tesla C2050, the NVIDIA Tesla K20c, the NVIDIA Quadro M5000, and the NVIDIA Tesla P100, were employed. The specifications of different GPUs are listed in Table 2. The test machine ran the 64-bit Windows 7 and the NVIDIA CUDA toolkit 7.0. Our single-threaded programs were executed on an E5-2620 v3 CPU, and the multi-threaded CUDA kernels were executed on the C2050, the K20c, the M5000 and the P100, respectively.

5.2 Landmark Tracking

Landmark-based image registration is based on finding the corresponding landmarks in images. To improve the registration accuracy, a large number of landmarks are required to be matched or tracked, which could be performed concurrently on GPU platforms because of their independent relationships.

TABLE 2
The Specifications of Different GPUs ("N_{sp}" Represents the Number of Streaming Processors)

Specification	C2050	K20c	M5000	P100
N_{sm}	14	13	16	56
N_{sp}	448	2496	2048	3584
Processor Clock (MHz)	575	706	861	1190
Memory Size (GB)	3	5	8	16
Memory Bandwidth (GB/s)	144	208	211.6	732.2
Memory Clock (MHz)	750	1300	1653	715
Computing Version	5.1	3.5	5.2	6.0

In this experiment, a landmark tracking algorithm for 4D CT lung images, the spatially extended structure tensor (SEST)-based landmark tracking algorithm [36], was implemented on GPU platforms to accelerate its performance.

5.2.1 The SEST Landmark Tracking Algorithm

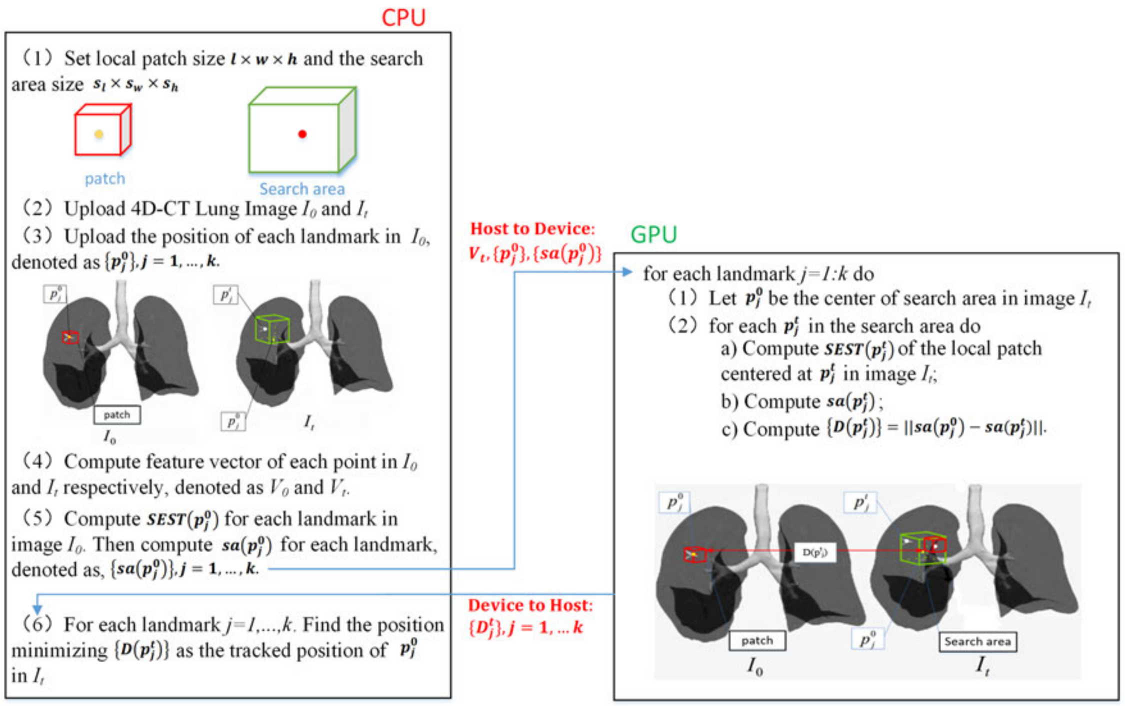
To illustrate the GPU implementation of the SEST landmark tracking algorithm in detail, we first give a brief introduction of the SEST landmark tracking algorithm in this section. Details of SEST can be found in [36].

The basic idea of SEST is to extract tensor features from the local region centered at a landmark in an image at a given point of time and to search for the corresponding landmarks in images taken at other times based on the similarity of the tensor features. For 4D CT landmark tracking, the 4D CT image acquired with N phases is denoted as $\{I_t|t=0,1,\ldots,N-1\}$, and I_t is supposed to be the image for phase t. We extract k landmarks p_j^0, $j=1,2,\ldots,k$, from I_0 and track their corresponding positions p_j^t in target images I_t. Given a 3D local patch P_j^0 centered at p_j^0, the local patch is uniformly partitioned into eight sub-patches $P_{j,i}^0$, $i=1,\ldots,8$. Let $SP_{j,i}$ be the structure tensor of the ith sub-patch $P_{j,i}^0$; then

$$\begin{aligned} SP_{j,i} &= \sum_{\mu\in P_{j,i}^0} ST(\mu), \qquad ST(\mu)=V_\mu V_\mu^T, \\ V_\mu &= [I_0(\mu), I_{0,xx}(\mu), I_{0,yy}(\mu), I_{0,zz}(\mu), \\ &\quad I_{0,xy}(\mu), I_{0,xz}(\mu), I_{0,yz}(\mu)], \end{aligned} \tag{24}$$

where $I_0(\mu)$ is the image intensity of coordinate μ, and the others are the second partial derivatives in different directions at μ in image I_0. The structure tensor $SESP(p_j^0)$ is defined as $SESP(p_j^0)=[SP_{j,1},SP_{j,2},\ldots,SP_{j,8}]$. $SP_{j,i}$ can be decomposed as $SP_{j,i}=U_iU_i^T,\ i=1,\ldots,8$, by Cholesky decomposition [45]; each U_i is a lower triangular matrix with real entries and is represented as a vector a_i. All a_i are cascaded as a long vector to represent the structure tensor, denoted as $sa=[a_1^T,a_2^T,\ldots,a_8^T]$. The euclidean distance between different vectors sa is the measure of similarity between different structure tensors. The most similar point in the search area is the one tracked for p_j^0.

To improve image registration accuracy, any landmark with a displacement that is not consistent with the average displacement of surrounding landmarks is eliminated. The relaxed thin-plate spline algorithm [46] was employed to ensure the topology preservation of the deformation field.


CPU

(1) Set local patch size $l \times w \times h$ and the search area size $s_l \times s_w \times s_h$

(2) Upload 4D-CT Lung Image I_0 and I_t

(3) Upload the position of each landmark in I_0, denoted as $\{p_j^0\}, j = 1, \dots, k$.

(4) Compute feature vector of each point in I_0 and I_t respectively, denoted as V_0 and V_t.

(5) Compute $SEST(p_j^0)$ for each landmark in image I_0. Then compute $sa(p_j^0)$ for each landmark, denoted as, $\{sa(p_j^0)\}, j = 1, \dots, k$.

(6) For each landmark $j = 1, \dots, k$. Find the position minimizing $\{D(p_j^t)\}$ as the tracked position of p_j^0 in I_t

Host to Device: $V_t, \{p_j^0\}, \{sa(p_j^0)\}$

Device to Host: $\{D_j^t\}, j = 1, \dots k$

GPU

for each landmark $j = 1:k$ do

(1) Let p_j^0 be the center of search area in image I_t

(2) for each p_j^t in the search area do

a) Compute $SEST(p_j^t)$ of the local patch centered at p_j^t in image I_t;

b) Compute $sa(p_j^t)$;

c) Compute $\{D(p_j^t)\} = ||sa(p_j^0) - sa(p_j^t)||$.

Fig. 1. CPU–GPU model for spatially extended structure tensor (SEST) tracking algorithm.

5.2.2 CPU–GPU Implementation Strategy

The CPU–GPU cooperative processing model for the SEST tracking algorithm is described in Fig. 1. In the CPU, we uploaded the reference image I_0, the target image I_t, and the position of each landmark in I_0. Once all the data that we needed were prepared, $SESP(p_j^0)$ and $sa(p_j^0)$ of each landmark in image I_0 were computed in the CPU. After that, we transferred $\{sa(p_j^0), j = 1, \dots, k\}$ from CPU (host) to GPU (device). Simultaneously, the feature vector of each point in target image, denoted as V_t, and the center position of the search area for p_j^0 were also transferred to the device.

In the GPU, for each candidate landmark p_j^t, $j = 1, \dots, k$, $SESP(p_j^t)$ and $sa(p_j^t)$ were computed. The euclidean distance between $sa(p_j^0)$ and $sa(p_j^t)$ is denoted as similarity $D(p_j^t)$. All values $\{D(p_j^t), j = 1, \dots, k\}$ were transferred from GPU to CPU to find the tracked position of p_j^0.

As the computation of $D(p_j^t)$ for each candidate p_j^t in image I_t is data independent, our strategy was to invoke thousands of threads to compute $\{D(p_j^t)\}$, with each thread performing the computation of $SESP(p_j^t)$ and $sa(p_j^t)$ for a candidate point p_j^t in the search area. Furthermore, each thread implemented the computation of the euclidean distances between $sa(p_j^0)$ and $sa(p_j^t)$, including the computation of SP_i, U_i, and a_i $(i = 1, \dots, 8)$.

In CUDA, we used a 2D grid and 1D block model for this task, each line of blocks in the grid corresponding to a tracking landmark p_j^t, $j = 1, \dots, k$, and each thread in these blocks corresponding to one candidate point matching this landmark. Therefore, the total number of blocks in the grid, N_b, was

$$N_\mathrm{b} = \left\lceil \frac{s_l \times s_w \times s_h}{Threads\_per\_block} \right\rceil \times k, \tag{25}$$

where s_l, s_w, s_h are the size of the search region.

In our implementation of landmark tracking, the cost of memory access was much more than the cost of computation for each thread. Therefore, it was necessary to optimize memory access to the extent possible. For the GPU, our input data were V_t, $\{p_j^0\}$, and $\{sa(p_j^0)\}$, and output data were $\{D(p_j^t)\}$. Considering that $\{p_j^0\}$ and $\{sa(p_j^0)\}$ are accessed frequently, and all threads in the same block correspond to the same landmark, which means that they are required to access the same element in $\{p_j^0\}$ and $\{sa(p_j^0)\}$, we stored these two variables in shared memory. However, for each thread, a local patch centered at a specific point is needed for the computation of $SESP$. Each element of the local patch is a seven-dimensional vector obtained from the feature vector V_t. Suppose the patch size is $11 \times 11 \times 9$, and V_t is stored in 32-bit single-precision floating-point format. Then it takes approximately $11 \times 11 \times 9 \times 7 \times 32$ bits ≈ 29.8 KB of memory space to store one patch. Moreover, as each block contains at least 32 threads (a warp), and each thread accesses adjacent patches, it requires at least $(11 + 31) \times 11 \times 9 \times 7 \times 32$ bits ≈ 113.7 KB of memory space to store all patches for a thread block. Since the maximum size of shared memory is 48 KB for devices with the Fermi or Kepler architecture, storing all patches for the threads in the same block in shared memory space is impossible. Therefore, we stored the data for all patches—the feature vector V_t—in global memory space.

5.2.3 Performance Estimation and Comparison

To evaluate our performance estimation model, ten individual patients' lung CT images provided in the DIR-Lab dataset [47] were used in our experiments. Each case includes the set of five phases of CT 16-bit integer images from T00 to T50, which were acquired in the inspiratory period. For each case, 75 corresponding landmarks from T00 to T50 are provided by experts, and 300 landmarks in the maximum inhalation and the maximum exhalation are also provided

TABLE 3
Image Sizes and Number of Landmarks Extracted (T00) and Remaining (T10–T50)

Case	Image size	T00	T10	T20	T30	T40	T50
1	$256 \times 256 \times 94$	2602	2288	1926	2045	2081	2045
2	$256 \times 256 \times 112$	4053	3298	3228	3075	3073	3059
3	$256 \times 256 \times 104$	3055	2390	2348	2274	2230	2129
4	$256 \times 256 \times 99$	2128	1736	1456	1383	1356	1340
5	$256 \times 256 \times 106$	2771	1971	1944	1967	1823	1834
6	$325 \times 325 \times 128$	2246	1832	1452	1428	1260	1326
7	$325 \times 325 \times 136$	2779	2188	1981	1840	1747	1672
8	$325 \times 325 \times 128$	4288	3283	2989	2913	2744	2636
9	$325 \times 325 \times 128$	1489	1122	1092	1055	1001	909
10	$325 \times 325 \times 120$	2454	1910	1616	1478	1467	1491

by experts, which can be used to evaluate image registration accuracy.

In our experiment, we extracted thousands of landmarks from I_0, the image at the maximum inhalation, listed in the T00 column in Table 3, and tracked these landmarks in I_t, $t = 1, \ldots, 5$, to evaluate the performance of our parallel implementation strategy. Columns T10–T50 in the table list the number of landmarks remaining after mistracked landmarks were eliminated. Note that we extracted the regions of interest that contained the lung regions from the original images provided in the DIR-Lab dataset; therefore, the image sizes listed in the table are not consistent with the original image sizes in DIR-Lab.

To evaluate the performance of our model, first, we estimated the execution time of the SEST landmark tracking algorithm using our performance model and using Hong and Kim's model. Next, the actual execution times for landmark tracking were recorded and compared to the estimated times. Fig. 2 illustrates the actual execution times and estimated times for landmark tracking. As mentioned above, the total execution time estimated using our model is a cost function based on the parameter $Threads\_per\_block$. Here, different values for $Threads\_per\_block$ are employed to show its influence on the execution time.

As shown in Fig. 2, even though the numbers of threads in a block are different, the execution times estimated using Hong and Kim's model are almost the same because the $Threads\_per\_block$ parameter is neutralized in the model. As we know, $Threads\_per\_block$ is closely related to the on-chip resources, which means Hong and Kim's model cannot represent the influence of on-chip resources. In contrast, the times as estimated by our model and the actual times are approximately equal to each other on different platforms, thus confirming the accuracy of our performance model. Moreover, by using the optimal $Threads\_per\_block$ value, the best implementation performance can be achieved using our performance model.

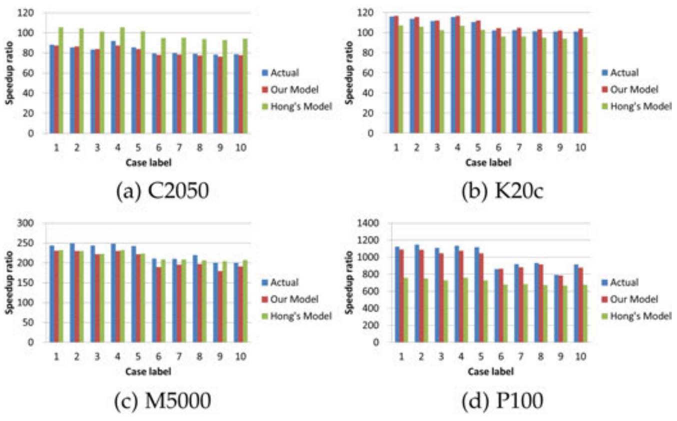


Fig. 3. Comparison of the actual and estimated speedups for landmark tracking.

The speedups of the landmark tracking, calculated using the optimal $Threads\_per\_block$ for the C2050, K20c, M5000 and P100, are shown in Fig. 3. Here, both the estimated speedups using our performance model and those using Hong and Kim's model have presented as well as the actual speedups. It can be seen that the speedup ratio of our parallel implementation for the SEST landmark tracking algorithm is approximately 80, 100, 200 and 800 for the C2050, K20c, M5000 and P100, respectively. Moreover, it is observed that our estimated speedups are closer to the actual speedup compared to Hong and Kim's model, illustrating the accuracy of our proposed model.

Using the parameters illustrated above, we also compare the speedup ratios calculated for different numbers of tracked landmarks on different platforms to demonstrate the influence of data transfers on the speedup. Here, all ten cases were employed to track 300 landmarks and thousands of landmarks. The experiment results are shown in Fig. 4. It is observed that the speedup ratio for thousands of tracked

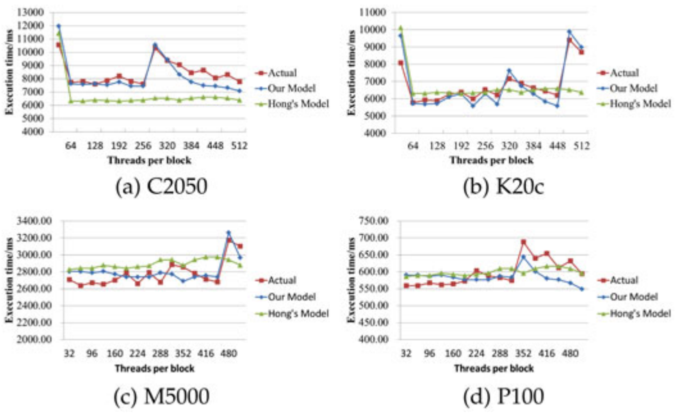


Fig. 2. Actual execution times and execution times estimated using our model and Hong and Kim's model for landmark tracking.

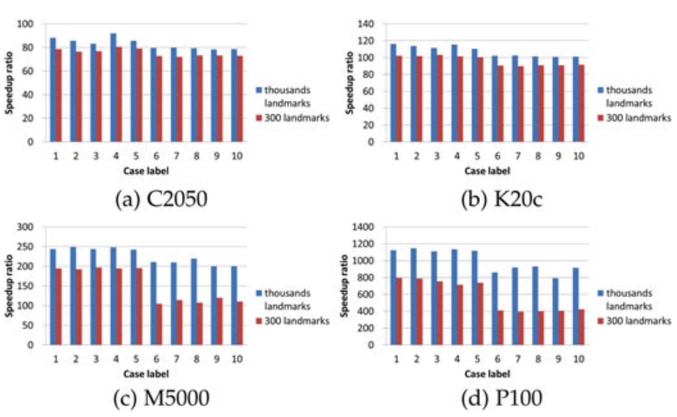


Fig. 4. Speedup ratios for tracking 300 landmarks and thousands of landmarks.

TABLE 4
Mean (and Standard Deviation) of Target Registration Errors (mm) for the 300 Landmark Points between Maximum Inhalation (MI) and Maximum Exhalation (ME) Phases of the DIR-Lab Dataset

Case	Initial	Castillo *et al.*	Wu *et al.* (without super-resolution)	Metz *et al.*	Heinrich *et al.*	Original SEST	Our method
1	3.89 (2.78)	0.97 (1.02)	0.85 (0.82)	1.02 (0.50)	0.97 (0.5)	0.94 (1.33)	0.76 (0.94)
2	4.34 (3.90)	0.86 (1.08)	0.79 (0.65)	1.06 (0.56)	0.96 (0.5)	0.78 (0.94)	0.67 (0.85)
3	6.94 (4.05)	1.01 (1.17)	0.92 (0.54)	1.19 (0.66)	1.21 (0.7)	0.92 (1.11)	0.87 (1.07)
4	9.83 (4.86)	1.40 (1.57)	1.12 (0.82)	1.57 (1.20)	1.39 (1.0)	1.43 (1.49)	1.32 (1.23)
5	7.48 (5.51)	1.67 (1.79)	1.43 (0.96)	1.73 (1.49)	1.72 (1.6)	1.49 (1.53)	1.28 (1.45)
6	10.89 (6.97)	1.58 (1.65)	6.95 (4.06)	–	1.49 (1.0)	1.31 (1.41)	1.30 (1.20)
7	11.03 (7.43)	1.46 (1.29)	3.64 (2.15)	–	1.58 (1.2)	1.15 (1.76)	1.24 (1.08)
8	14.99 (9.01)	1.77 (2.12)	4.05 (2.64)	–	2.11 (2.4)	1.68 (1.81)	1.41 (1.32)
9	7.92 (3.98)	1.19 (1.12)	3.96 (1.85)	–	1.36 (0.7)	1.10 (0.99)	1.21 (1.06)
10	7.30 (6.35)	1.59 (1.87)	3.25 (2.68)	–	1.55 (1.6)	1.29 (1.32)	1.22 (1.10)

landmarks is greater than that for the 300 tracked landmarks for the all ten cases, on different platforms, because T_c is constant for a specific case. When the number of landmarks increases, the ratio between computation cost and communication cost increases as well, resulting in an increase in speedup. In addition, when the number of landmarks is fixed, the speedup ratio differs because of the differing quantities of data to be transferred (note that the data quantities for cases 6 to 10 are much larger than those for cases 1 to 5).

In addition, we analyze the theoretical performance of memory access in GPUs to explain the reason for our results. The CPU Xeon E5-2620 runs single-threaded programs, which accesses a single data point per instruction and no memory access instruction can be issued during the memory waiting period. By contrast, the NVIDIA GPU architecture is built around a scalable array of SMs, each of which can access the memory simultaneously during one memory warp waiting period. Moreover, benefiting by coalescing access to global memory, DRAM bandwidth required by a warp can be minimized by 32 times. Thus, the C2050, K20c, M5000 and P100 can achieve up to $32 \times 14 = 448$, $32 \times 13 = 416$, $32 \times 16 = 512$ and $32 \times 56 = 1792$ times data throughput, respectively, compared to the single-thread memory access by a CPU.

To evaluate the registration accuracy, the euclidean distance between the landmark positions marked by experts and the tracked results, commonly known as the target registration error (TRE) [48], was used. The lower the TREs, the better the registration accuracy. For comparison with the tracking accuracy of the GPU implementation of SEST, various methods were employed, including the methods of Castillo et al. [49], Wu et al. [50], Metz et al. [51] (who provided results for only five cases), and Heinrich et al. [52], as well as the original SEST results [36]. In the SEST algorithm, the patch size and search area size are $11 \times 11 \times 9$ and $25 \times 13 \times 17$, respectively. Table 4 lists the registration errors as evaluated using 300 expert points.

As listed in Table 4, the SEST tracking algorithm outperforms other methods for most cases, particularly for cases 5 to 8, which have large deformations. Moreover, by comparing the tracking results for the different numbers of landmarks, it can be seen that image registration accuracy can be considerably improved if a large number of landmarks is tracked. This implies that a major benefit for landmark-based image registration methods can be had by implementing them effectively on GPUs; that is, when the average execution time for tracking a single landmark is substantially decreased, a large number of landmarks can be tracked. As a result, a more accurate transformation can be estimated based on dense corresponding relationships between landmarks, and the registration accuracy will subsequently improve as well.

5.3 Spatial Transformation of Images

In spatial transformation, the coordinates of a source image are mapped to the coordinate system of a reference image using a given mapping function. For 3D images, the mapped positions of voxels can be calculated concurrently because they are independent of each other.

The transformation function we employed is as follows:

$$f(\mu) = r_1 + r_2 x + r_3 y + r_4 z + \sum_{j=1}^{k'} w_j \| p_j^0 - \mu \|, \tag{26}$$

which maps the coordinate $\mu = (x, y, z)$ to $f(\mu)$. $[r_1, r_2, r_3, r_4, w_1, \ldots, w_{k'}]$ is the transformation coefficient vector, which was estimated using the relaxed thin-plate spline algorithm, and k' is the number of landmarks remaining, as listed in Table 3. Then, we used the transformation function $f(\mu)$ to map all coordinates of a 3D image to the coordinate system of another image.

We used a 3D block grid to implement the 3D spatial transformation. Each thread has a 3D index (x, y, z), corresponding to a 3D coordinate position μ. When thousands of landmarks are tracked, the transformation coefficients $r_1, r_2, r_3, r_4, w_1, \ldots, w_{k'}$ and the coordinates of each landmark p_j^0, $j = 1, \ldots, k'$, will consume a large amount of memory space. Shared memory is allocated per thread block. If we store these data in shared memory, the amount of shared memory requested by each block will be very large, and the number of active blocks will be very small (Eq. (7)). Therefore, we stored the transformation coefficients and the landmark coordinates in global memory space.

Fig. 5 shows the actual execution times and the execution times estimated using our model and Hong and Kim's model, for the spatial transformation task. Note that the times estimated using our model are approximately the same as the actual execution times on different platforms, which confirms the accuracy of our performance model. On the C2050, K20c and M5000, this task is mainly dominated by computation. When $Threads\_per\_block$ is small, the value of warp parallelism for computation (RWP_C) is limited by the value of $Threads\_per\_block$; otherwise, the value of

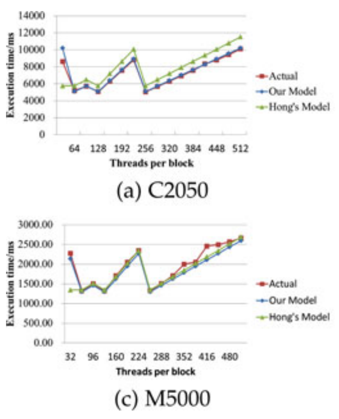


Fig. 5. Actual execution times and execution times estimated using our model and Hong and Kim's model for spatial transformation without using shared memory.

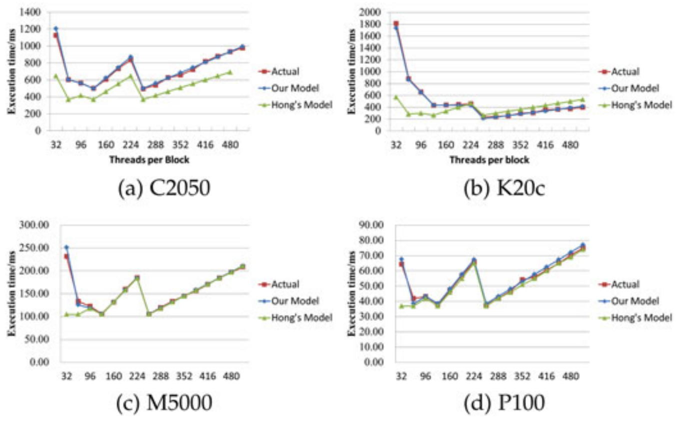


Fig. 7. Actual execution times and execution times estimated using our model and Hong and Kim's model for spatial transformation with shared memory.

RWP_C is limited by the number of warp schedulers, which is a constant, as shown in Eq. (4). However, Hong and Kim's model does not consider the influence of $Threads\_per\_block$ on the execution time when the execution time is dominated by computation. As a result, Hong and Kim's model cannot predict performance accurately when $Threads\_per\_block$ is small. However, on P100, the execution time of this task is dominated by memory, due to the architecture of P100, and the value of warp parallelism is not limited by on-chip resources. Therefore, both the execution times estimated using our model and those using Hong and Kims model are approximately the same as the actual execution times.

Fig. 6 displays the speedup ratios of our parallel implementation for spatial transformation on different GPU platforms. As in the case of the landmark tracking task, speedups estimated by our performance model are closer to the actual speedups than are those of Hong and Kim's model for the spatial transformation task.

Furthermore, to demonstrate the performance of our model using shared memory, we re-implemented spatial transformation by storing the transformation coefficients and the landmark coordinates in shared memory. In this experiment, 300 landmarks provided by experts were tracked to ensure enough active blocks. For each block, we used the first thread of the block to load the transformation coefficients and the landmark coordinates from global memory and stored them in shared memory. After that, all threads in the same block could use the data in shared memory for computation. As a result, for this implementation, the cost of memory access is greater than the cost of computation, and thus the execution times in this experiment were dominated by memory access.

As shown in Fig. 7, the execution times estimated using our model are approximately the same as the actual execution times on different platforms, whereas the execution times estimated using Hong and Kim's model vary considerably owing to the lack of consideration of on-chip resources. Note that the execution times estimated using Hong and Kim's model are smaller than the execution times estimated using our model on the C2050 because the number of active warps estimated using Hong and Kim's model is greater than that estimated using our model. Similar results can be observed on the M5000 and P100 when the value of $Threads\_per\_block$ is small. In addition, on K20c, when $Threads\_per\_block > 96$, the value of warp parallelism in Hong and Kim's model is constant, limited by MWP_peak_BW; when $96 < Threads\_per\_block < 192$, the value of warp parallelism in Hong and Kim's model is greater than the one in our model; in contrast, when $Threads\_per\_block > 256$, the value of warp parallelism in Hong and Kim's model is smaller than the one in our model. Compared with the actual execution times, the value of warp parallelism estimated by our model is more accurate than Hong and Kim's model.

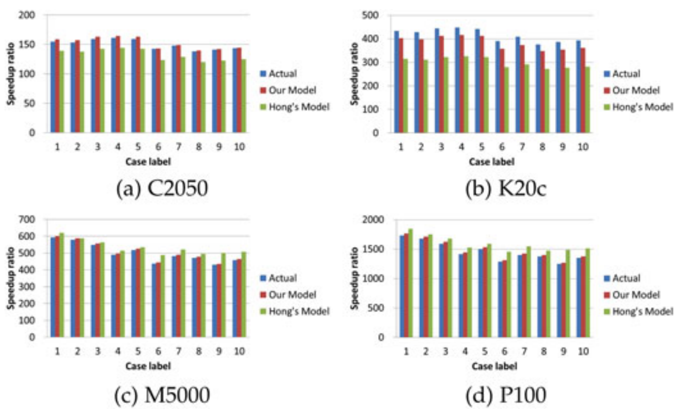


Fig. 6. Comparison of the actual and estimated speedups for spatial transformation.

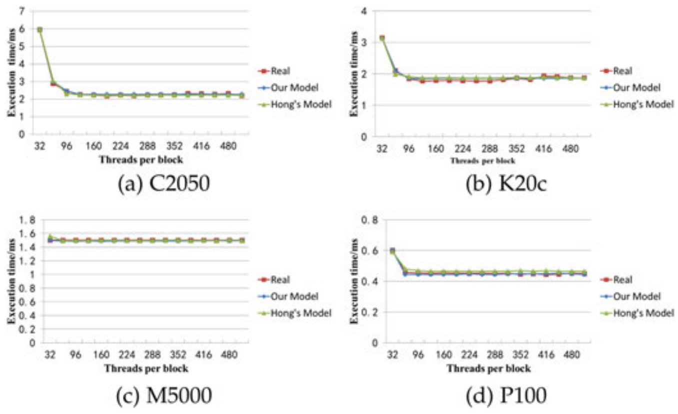


Fig. 8. Actual execution times and execution times estimated using our model and Hong and Kim's model for trilinear interpolation.

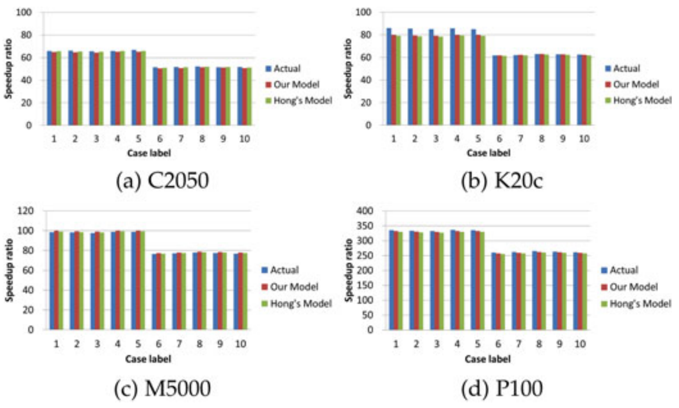


Fig. 9. Comparison of the actual and estimated speedups for trilinear interpolation.

5.4 Trilinear Interpolation

After spatial transformation, the coordinates of the source image are mapped to non-grid positions, and then an interpolation technique is needed to estimate the pixel intensities at the grid positions. To accomplish this, trilinear interpolation [53] is employed. In the GPU implementation, we launched thousands of threads to compute the pixel intensities, each thread calculating a pixel value at (x, y, z) with very little computation cost. However, each thread accesses the image intensities of eight pixels located on a unit cube in an image, which results in global memory access. In this experiment, the execution time was dominated by memory access. Here, very few local variables are declared and no shared memory access is required, meaning that the execution time is not limited by on-chip resources. This experiment was used to evaluate the performance of our model for predicting memory-intensive tasks not influenced by on-chip resources.

The actual execution times and the execution times estimated using our model and Hong and Kim's model are shown in Fig. 8. Note that both the execution times estimated using our model and those using Hong and Kim's model are approximately the same as the actual execution times, which confirms the accuracy of our model when on-chip resources are not influenced. The speedup ratios of our parallel implementation for trilinear interpolation are shown in Fig. 9. It can be observed that both the speedups estimated using our model and those using Hong and Kim's model are very close to the actual speedups.

6 Conclusions

In this paper, we have proposed a novel performance estimation model to predict the execution time of applications running on GPU platforms. We take the on-chip GPU resources into consideration to improve the accuracy of the performance prediction. Moreover, the cost of data transfer between CPU and GPU is also captured in our model. Using the performance model, the optimal parameters for implementing an application on GPUs can be estimated. We evaluated our performance model on four NVIDIA platforms, the C2050 (Fermi architecture), the K20c (Kepler architecture), the M5000 (Maxwell architecture) and the P100 (Pascal architecture). For performing this evaluation, the following tasks in medical image registration, including landmark tracking, spatial transformation, and image interpolation, were implemented. The experimental results demonstrate that the execution times estimated using our performance model and the actual execution times are closer to each other compared to the existing model. For landmark tracking, speedups of 80× for the C2050, 100× for the K20c, 200× for the M5000 and 800× for the P100 were achieved when tracking thousands of landmarks for all ten cases of data from the DIR-Lab dataset. Moreover, as a benefit of efficient implementation on GPUs, massive numbers of landmarks can be tracked for methods based on landmark tracking, thereby increasing the registration accuracy.

Acknowledgments

This paper is supported by the National Natural Science Foundation of China (61871269), the Shenzhen Fundamental Research Program (JCYJ20170818100006280) and the Science and Technology Project of Jiangxi Provincial Education Department (GJJ170515).

References

[1] Y. Ma, L. Chen, P. Liu, and K. Lu, "Parallel programing templates for remote sensing image processing on gpu architectures: design and implementation," *Comput.*, vol. 98, no. 1/2, pp. 7–33, 2016.
[2] NVIDIA Corporation, "Cuda c programming guide version 9.1.85," 2017.
[3] C. Ozcan and B. Sen, "Investigation of the performance of lu decomposition method using cuda," *Procedia Technol.*, vol. 1, no. 1, pp. 50–54, 2012.
[4] X. Chen, L. Ren, Y. Wang, and H. Yang, "Gpu-accelerated sparse lu factorization for circuit simulation with performance modeling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 3, pp. 786–795, Mar. 2015.
[5] D. Chen, Y. Hu, L. Wang, A. Y. Zomaya, and X. Li, "H-PARAFAC: Hierarchical parallel factor analysis of multidimensional big data," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 1091–1104, Apr. 2017.
[6] D. Chen, X. Li, L. Wang, S. U. Khan, J. Wang, K. Zeng, and C. Cai, "Fast and scalable multi-way analysis of massive neural data," *IEEE Trans. Comput.*, vol. 64, no. 3, pp. 707–719, Mar. 2015.
[7] D. Chen, Y. Hu, C. Cai, K. Zeng, and X. Li, "Brain big data processing with massively parallel computing technology: challenges and opportunities," *Softw.: Practice Experience*, vol. 47, no. 3, pp. 405–420.
[8] D. Ruijters, B. M. ter Haar Romeny, and P. Suetens, "Gpu-accelerated elastic 3d image registration for intra-surgical applications," *Comput. Methods Programs in Biomed.*, vol. 103, no. 2, pp. 104–112, 2011.
[9] 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.*, vol. 99, no. 2, pp. 133–146, 2010.
[10] Z. Zhong, L. Zhuang, X. Gu, J. Wang, H. Chen, and X. Zhen, "Tu-ab-202–05: Gpu-based 4d deformable image registration using adaptive tetrahedral mesh modeling," *Med. Phys.*, vol. 43, no. 6Part33, pp. 3737–3737, 2016.
[11] U. Lopez-Novoa, A. Mendiburu, and J. Miguel-Alonso, "A survey of performance modeling and simulation techniques for accelerator-based computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 272–281, Jan. 2015.
[12] S. Hong and H. Kim, "An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, 2009.
[13] K. Kothapalli, R. Mukherjee, M. S. Rehman, S. Patidar, P. Narayanan, and K. Srinathan, "A performance prediction model for the cuda gpgpu platform," in *Proc. Int. Conf. High Perform. Comput.*, 2009, pp. 463–472.
[14] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W. M. W. Hwu, "An adaptive performance modeling tool for gpu architectures," in *Proc. ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2010, pp. 105–114.
[15] M. Boyer, J. Meng, and K. Kumaran, "Improving gpu performance prediction with data transfer modeling," in *Proc. Parallel Distrib. Process. Symp. Workshops PhD Forum*, 2013, pp. 1097–1106.

[16] J. Meng, V. A. Morozov, K. Kumaran, V. Vishwanath, and T. D. Uram, “Grophecy: Gpu performance projection from cpu code skeletons,” in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–11.

[17] S. Che and K. Skadron, “Benchfriend: Correlating the performance of gpu benchmarks,” *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 238–250, 2014.

[18] T. T. Dao, J. Kim, S. Seo, B. Egger, and J. Lee, “A performance model for gpus with caches,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 7, pp. 1800–1813, Jul. 2015.

[19] K. Li, W. Yang, and K. Li, “Performance analysis and optimization for spmv on gpu using probabilistic modeling,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 1, pp. 196–205, Jan. 2015.

[20] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[21] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, “An adaptive performance modeling tool for gpu architectures,” *ACM SIGPLAN Notices*, vol. 45, no. 5, pp. 105–114, 2010.

[22] S. S. Baghsorkhi, I. Gelado, M. Delahaye, and W.-m. W. Hwu, “Efficient performance evaluation of memory hierarchy for highly multithreaded graphics processors,” *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 23–34, 2012.

[23] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, “A performance analysis framework for identifying potential benefits in gpgpu applications,” *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 11–22, 2012.

[24] S. Hong and H. Kim, “An integrated gpu power and performance model,” *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 280–289, 2010.

[25] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, “Gpuwattch: Enabling energy optimizations in gpgpus,” *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 487–498, 2013.

[26] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, “How a single chip causes massive power bills gpusimpow: A gpgpu power simulator,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw*, 2013, pp. 97–106.

[27] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing cuda workloads using a detailed gpu simulator,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw*, 2009, pp. 163–174.

[28] S. Collange, M. Daumas, D. Defour, and D. Parello, “Barra: A parallel functional simulator for gpgpu,” in *Proc. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2010, pp. 351–360.

[29] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, “Multi2sim: a simulation framework for cpu-gpu computing,” in *Proc. 21st Int. Conf. Parallel Archit. Compilation Tech.*, 2012, pp. 335–344.

[30] T. C. Carroll and P. W. H. Wong, “An improved abstract gpu model with data transfer,” in *Proc. Int. Conf. Parallel Process. Workshops*, 2017, pp. 113–120.

[31] C. Gregg and K. Hazelwood, “Where is the data? why you cannot debate cpu vs. gpu performance without the answer,” in *Proc. IEEE Int. Symp. PERFORMANCE Anal. Syst. Softw.*, 2011, pp. 134–144.

[32] R. Shams, P. Sadeghi, R. A. Kennedy, and R. I. Hartley, “A survey of medical image registration on multicore and the gpu,” *IEEE Signal Process. Mag.*, vol. 27, no. 2, pp. 50–60, Mar. 2010.

[33] T.-Y. Huang, Y.-W. Tang, and S.-Y. Ju, “Accelerating image registration of mri by gpu-based parallel computation,” *Magn. Resonance Imaging*, vol. 29, no. 5, pp. 712–716, 2011.

[34] D. P. Shamonin, E. E. Bron, B. P. Lelieveldt, M. Smits, S. Klein, and M. Staring, “Fast parallel image registration on cpu and gpu for diagnostic classification of alzheimer’s disease,” *Frontiers Neuroinformatics*, vol. 7, 2014, Art. no. 50.

[35] N. D. Ellingwood, Y. Yin, M. Smith, and C.-L. Lin, “Efficient methods for implementation of multi-level nonrigid mass-preserving image registration on gpus and multi-threaded cpus,” *Comput. Methods Programs Biomed.*, vol. 127, pp. 290–300, 2016.

[36] B. H. JunhaWu and X. Yang, “Deformable registration of 4d-ct lung image using landmark tracking” *Biomed. Res.*, vol. 27, no. 3, pp. 801–811, 2016.

[37] A. Lastovetsky and R. Reddy, “Data partitioning with a functional performance model of heterogeneous processors,” *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 1, pp. 76–90, 2007.

[38] Z. Zhong, V. Rychkov, and A. Lastovetsky, “Data partitioning on multicore and multi-gpu platforms using functional performance models,” *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2506–2518, Sep. 2015.

[39] A. Kerr, G. Diamos, and S. Yalamanchili, “Modeling gpu-cpu workloads and systems,” in *Proc. 3rd Workshop General-Purpose Comput. Graph. Process. Units*, 2010, pp. 31–42.

[40] NVIDIA Corporation, “Whitepaper nvidias next generation cuda compute architecture: Fermi,” 2009.

[41] M. Harris, “How to access global memory efficiently in cuda c/c++ kernels,” Website, 2013. [Online]. Available: https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/

[42] NVIDIA GeForce GTX, “980: Featuring maxwell, the most advanced gpu ever made,” *White paper, NVIDIA Corporation*, 2014.

[43] NVIDIA Tesla, “P100 gpu,” *Pascal Architecture White Paper*, 2016.

[44] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, “Demystifying gpu microarchitecture through microbenchmarking,” in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 235–246.

[45] H. Yu, C. Chung, K. Wong, H. Lee, and J. Zhang, “Probabilistic load flow evaluation with hybrid latin hypercube sampling and cholesky decomposition,” *IEEE Trans. Power Syst.*, vol. 24, no. 2, pp. 661–667, May 2009.

[46] G. Wahba, *Spline Models for Observational Data*, vol. 59. Philadelphia, PA, USA: SIAM, 1990.

[47] R. Castillo, E. Castillo, R. Guerra, V. E. Johnson, T. McPhail, A. K. Garg, and T. Guerrero, “A framework for evaluation of deformable image registration spatial accuracy using large landmark point sets,” *Phys. Med. Biol.*, vol. 54, no. 7, 2009, Art. no. 1849.

[48] G. Xiong, C. Chen, J. Chen, Y. Xie, and L. Xing, “Tracking the motion trajectories of junction structures in 4d ct images of the lung,” *Phys. Med. Biol.*, vol. 57, no. 15, 2012, Art. no. 4905.

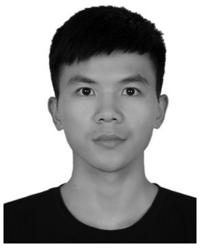
[49] E. Castillo, R. Castillo, J. Martinez, M. Shenoy, and T. Guerrero, “Four-dimensional deformable image registration using trajectory modeling,” *Phys. Med. Biol.*, vol. 55, pp. 305–327, 2010.

[50] G. Wu, Q. Wang, J. Lian, and D. Shen, “Estimating the 4d respiratory lung motion by spatiotemporal registration and super-resolution image reconstruction,” *Med. Phys.*, vol. 40, pp. 0 317 101–03 171 017, 2013.

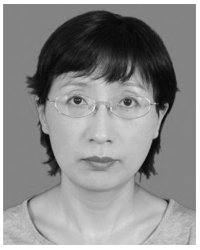
[51] C. T. Metz, S. Klein, M. Schaap, T. van Walsum, and W. J. Niessen, “Nonrigid registration of dynamic medical imaging data using nd + t b-splines and a groupwise optimization approach,” *Med Image Anal.*, vol. 15, pp. 238–249, 2011.

[52] M. P. Heinrich, M. Jenkinson, S. M. Brady, and J. A. Schnabel, “MRF-based deformable registration and ventilation estimation of lung CT,” *IEEE Trans. Med. Imaging.*, vol. 32, pp. 1239–1248, Jul. 2013.

[53] Y. Bai and D. Wang, “On the comparison of trilinear, cubic spline, and fuzzy interpolation methods in the high-accuracy measurements,” *IEEE Trans. fuzzy Syst.*, vol. 18, no. 5, pp. 1016–1022, Oct. 2010.



Junhao Wu received the bachelor’s degree from the College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, Guangdong, China, in 2015. He is currently working toward the PhD degree in the College of Computer Science and Software Engineering, Shenzhen University. His research interests include image registration, parallel computing, and deep learning.



Xuan Yang received the PhD degree in communication and information systems from Xi’an Jiaotong University, Xi’an, China, in 1998. She worked at the postdoctoral mobile station of electronic science and technology of Xi’an University from 1999 to 2001. She has worked in teaching and scientific research with Shenzhen University since December 2001. Her current research interests include image processing, medical image processing, and deep learning.

Zhengrui Zhang received the MS degree from the College of Information Engineering, Shenzhen University, Shenzhen, Guangdong, China, in 2014. He is currently working toward the PhD degree in the College of Information Engineering, Shenzhen University. His research interests include image registration, image segmentation, and deep learning.

Guoliang Chen received the BSc degree from Xi'an Jiaotong University, Xi'an, China, in 1961. Since 1973, he has been with the University of Science and Technology of China, Hefei, China, where he is currently the Academic Committee Chair of the Nature Inspired Computation and Applications Laboratory, a professor with the Department of Computer Science and Technology, and the director of the School of Software Engineering. From 1981 to 1983, he was a visiting scholar at Purdue University, Indiana. He is currently also the director of the National High Performance Computing Center at Hefei, Hefei, China. His research interests include parallel algorithms, computer architecture, computer networks, and computational intelligence. He has published nine books and more than 200 research papers. He is an Academician of the Chinese Academy of Sciences. He was the recipient of the National Excellent Teaching Award of China in 2003.

Rui Mao received the BS and MS degrees in computer science from the University of Science and Technology of China, in 1997 and 2000, respectively, and another MS degree in statistics, in 2006, and the PhD degree in computer science from the University of Texas at Austin, Texas, in 2007. After three years with Oracle Corporation, he joined Shenzhen University in 2010, where he is now an associate professor with the College of Computer Science and Software Engineering. His research interests include universal data management and analysis and high-performance computing. He has about 50 publications, and his work on the pivot space model was awarded the Similarity Search and Applications (SISAP) 2010 Best Paper award.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.