# A Closer Look at GPGPU

LIANG HU and XILONG CHE, Jilin University
SI-QING ZHENG, University of Texas at Dallas



The lack of detailed white box illustration leaves a gap in the field of GPGPU (General-Purpose Computing on the Graphic Processing Unit), thus hindering users and researchers from exploring hardware potential while improving application performance. This article bridges the gap by demystifying the micro-architecture and operating mechanism of GPGPU. We propose a descriptive model that addresses key issues of most concerns, including task organization, hardware structure, scheduling mechanism, execution mechanism, and memory access. We also validate the effectiveness of our model by interpreting the software/hardware cooperation of CUDA.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)

General Terms: Design, Management, Performance

Additional Key Words and Phrases: GPGPU, conceptual model, micro-architecture, SIMD, parallel computing

**ACM Reference Format:**
Liang Hu, Xilong Che, and Si-Qing Zheng. 2016. A closer look at GPGPU. ACM Comput. Surv. 48, 4, Article 60 (March 2016), 20 pages.
DOI: http://dx.doi.org/10.1145/2873053

## 1. INTRODUCTION

Multicore CPUs and manycore GPUs have emerged and gradually dominated state-of-the-art high-performance computing. Although contemporary CPUs and GPUs are manufactured using the same semiconductor technology, the computational performance of GPUs increases more rapidly than that of CPUs. Divergent design choices drive them into devices of different capabilities given the same order of transistor count. CPUs are optimized for high-performance, task-parallel workloads since more transistors are dedicated to control logics such as branch prediction and out-of-order execution in each processing element. GPUs are optimized for high-performance data-parallel workloads since more transistors are dedicated to arithmetic logics such as floating-point calculation and transcendental function in each processing element.

This work is funded by the European Seventh Framework Program (FP7) under Grant No. GA-2011-295222, the National Natural Science Foundation of China under Grant No. 61073009, the National High Tech R&D Program 863 of China under Grant No. 2011AA010101, the National Sci-Tech Support Plan of China under Grant No. 2014BAH02F03, and by the Youth Science Foundation of Jilin Province of China under Grant No. 20160520011JH.
Authors' addresses: L. Hu and X. Che (corresponding author), Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education & College of Computer Science and Technology, Jilin University, No. 2699 Qianjin Street, Changchun, 130012, China; emails: {hul, chexilong}@jlu.edu.cn; S.-Q. Zheng, Department of Computer Science, Erik Jonsson School of Engineering and Computer Science, University of Texas at Dallas, 800 West Campbell Road, EC 31, Richardson, TX, 75080-3021; email: sizheng@utdallas.edu.

ACM 0360-0300/2016/03-ART60
DOI: http://dx.doi.org/10.1145/2873053

Furthermore, GPUs have also evolved rapidly from being graphic specific to general purpose and have opened a new computing era [Nickolls and Dally 2010], namely, General-Purpose Computing on the Graphic Processing Unit (GPGPU).

Although the introduction of programming paradigms such as Compute Unified Device Architecture (CUDA) [NVIDIA 2015a] and Open Computing Language (OpenCL) [AMD 2014; NVIDIA 2009a] have relieved users of recasting computation with graphics terms, effective GPGPU programming is still not as easy as learning a new language. In order to better harness such a state-of-the-art device for general-purpose computing, a full understanding of the underlying hardware and internal operation mechanisms is indispensable. However, up-to-date literature fails to provide sufficient white-box description, thus leaves a hidden gap that hinders the users and researchers from exploring hardware potential while improving application performance. This gap derives from different perspectives taken by different communities, including manufacturing companies, micro-architecture researchers, and application developers.

Most application reports focus on presenting domain-specific designs. They either show a simplified GPU structure that lacks operation details or ignore the structure by treating the GPU as a black-box device for providing general-purpose computing power. Official white papers (e.g., NVIDIA [2009b, 2012b, 2014, 2015c]) focus more on what capabilities and functions a GPU furnishes than on how the internal components work with each other. Programming guides (e.g., NVIDIA [2015a, 2012a, 2015b, 2009a]; AMD [2014]) focus on introducing coding syntaxes/rules without adequately explaining the underlying micro-architecture designs that lead to such rules. Micro-architecture papers (e.g., Brunie et al. [2012], Fung and Aamodt [2011], and Gebhart et al. [2011]) are concerned about both overall structure and certain operation details, but the information is biased on the new functions proposed while simplifying or even ignoring the rest.

Product patents (e.g. Nickolls and Lew [2011], Lindholm et al. [2010], Coon et al. [2008], Duluk Jr. [2010], and Mills et al. [2011]) are important sources for studying the internal details of GPU operation, although they are not absolutely equivalent to the commercial device fabricated. However, the information presented in patents is not complete or cohesive, since the patent writers took their respective views rather than sharing a unified one even when they are serving the same company. Notable technical tutorials (e.g., Fatahalian and Houston [2008] and Luebke and Humphreys [2007]) tried to illustrate the internal operating mechanisms of GPU computing in a rather straightforward manner, concentrating on a perspective of graphic-specific processing rather than general-purpose computing. Owens et al. [2007] reviewed the historical background of GPGPU and provided valuable insights into both hardware techniques and software developments. New-generation devices are surpassing the vision of the survey with new features/designs, calling for an updated and supplemented version for the GPGPU community.

This article intends to illustrate the underlying hardware components and internal operation mechanisms with respect to the GPU from a general-purpose perspective. We believe our effort will bridge the visions of application developers and hardware architects for pursuing the extreme of hardware power. The main contributions of this article can be summarized as follows.

- We propose a descriptive model of GPGPU that covers the most important aspects, including task organization, hardware structure, scheduling mechanism, execution mechanism, and memory access.
- We validate the effectiveness of our model by interpreting a typical GPGPU implementation in which a CUDA program runs on a CUDA compliant GPU.

Table I. Acronyms

| | |
|---|---|
| PE | Processing Element |
| SE | Scheduling Element |
| PS | Processing Script |
| SS | Scheduling Script |
| SC | Script Command |
| SL | Script List |
| TI | Task Instance |
| TB | Task Buffer |
| TU | Task Unit |

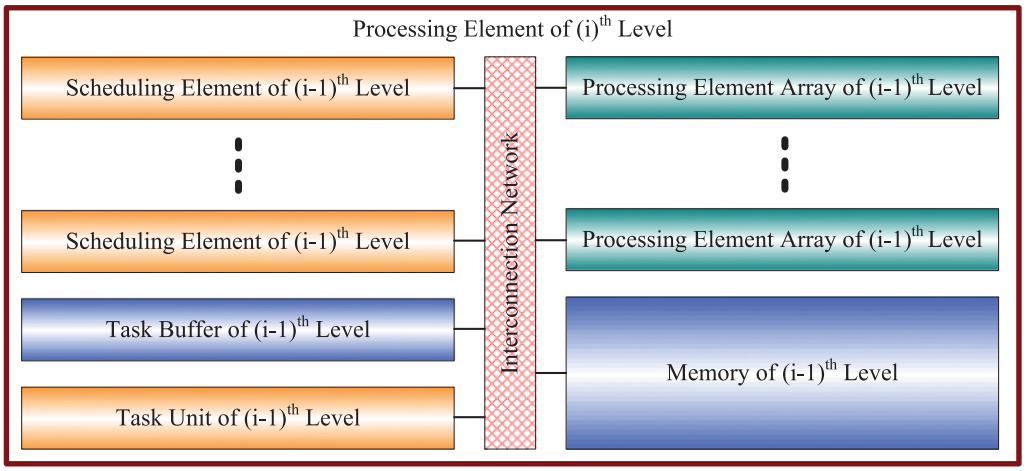


Fig. 1. Hardware structure submodel.

- We summarize the key characteristics that deserve attention in macro-/micro-architecture design, application software development, and hardware/software interactions.

## 2. A DESCRIPTIVE MODEL OF GPGPU

We summarize the operating principles of mainstream GPU products, then propose a hierarchical descriptive model of GPGPU. The model consists of five correlated submodels, including a hardware-structure submodel, task-organization submodel, task-execute submodel, task-dispatch submodel, and data-move submodel. It combines the key ideas behind the OpenCL model [AMD 2014; NVIDIA 2009a] and multi-BSP model [Valiant 2011], but with some revisions and extensions.

The OpenCL model is proposed with architecture abstraction for code transplanting across different architectures. The multi-BSP model is proposed with numerical parameters for performance evaluation given processor number, memory/cache size, communication cost and synchronization cost. The primary goal of our descriptive model is to explain the architectural choices and operational principles of GPGPU. Table I presents the acronyms defined by this model.

### 2.1. Hardware-Structure Submodel

The hardware-structure submodel of GPGPU is a $d$-level hierarchical structure of nested components, as shown in Figure 1. For each component in the figure, we give its definition. Note that we use $X^i$ to represent an X in the $i$-th level, and [X] to represent

an array of Xs. We define the depth (or the number of levels) of the model as $d$, with level 0 being the bottommost level and level $(d-1)$ being the topmost level. We do not fix $d$, as the GPGPU architecture is still evolving.

- *Processing Element (PE)*. A component that executes tasks of general-purpose computing. An array of processing elements that collaborate to execute parallel tasks is denoted as [PE].
- *Scheduling Element (SE)*. A component that manages the execution context for tasks of general-purpose computing.
- *Task Unit (TU)*. A component that accepts newly arrived tasks and creates execution contexts for them.
- *Task Buffer (TB)*. A component that stores the execution contexts of tasks in special-purpose counters/stacks/buffers/registers.
- *Memory*. A component that stores task data for computing purposes.

In addition, we define the interoperation rules among the components as follows.

- *Atomicity Rule*. A component (array) in the $(i-1)$-th level is atomic from the viewpoint of a component (array) in the $i$-th level. Components in the same level are atomic with respect to each other.
- *Locality Rule*. A component (array) can only interoperate with components sharing a common father component. This rule applies for most components. Exceptions exist that will be pointed out later.

### 2.2. Task-Organization Submodel

The task-organization submodel of GPGPU is a hierarchical mapping model from code structure to task topology, as shown in Figure 2. The code structure takes a flat structure containing a set of Script Lists of different levels. The task topology is a hierarchical topology of nested task instances. In what follows, we give related definitions.

- *Script Command (SC)*. A command that contains certain information for describing an operation of a component (array).
- *Scheduling Script (SS)*. A type of script that describes context operations with respect to a task of the current level. An SS is a sequence of SCs.
- *Processing Script (PS)*. A type of script that describes computing operations with respect to a lower-level task. A PS is also a sequence of SCs.
- *Script List (SL)*. A list of scripts that contains all the information with respect to a parallel task of the current level. An SL is a list of SSs and PSs.
- *Task Instance (TI)*. A running instance of the SL with a unique index called task index (just like an object is a running instance of a class with a unique pointer). An SL instantiates one TI group with its population defined as the SL's task width. TIs belonging to one group share the common SL but operate on respective index-based memory addresses. Figure 2 shows the condition for which the task width of each TI group equals 4.

We also define the interoperation rules among the TIs, as follows.

- *Atomicity Rule*. A $\mathrm{TI}^{i-1}$ is atomic from the viewpoin of a $\mathrm{TI}^{i}$.
- *Locality Rule*. A TI can only interoperate with TIs sharing a common SL.

### 2.3. Task-Execute Submodel

The task-execute submodel of GPGPU is a hierarchical mapping model from code structure to hardware structure, as shown in Figure 3. The code structure takes a flat structure containing a set of SLs of different levels. The hardware structure is a hierarchical structure of nested components. The execution operations of an $\mathrm{SL}^{i}$

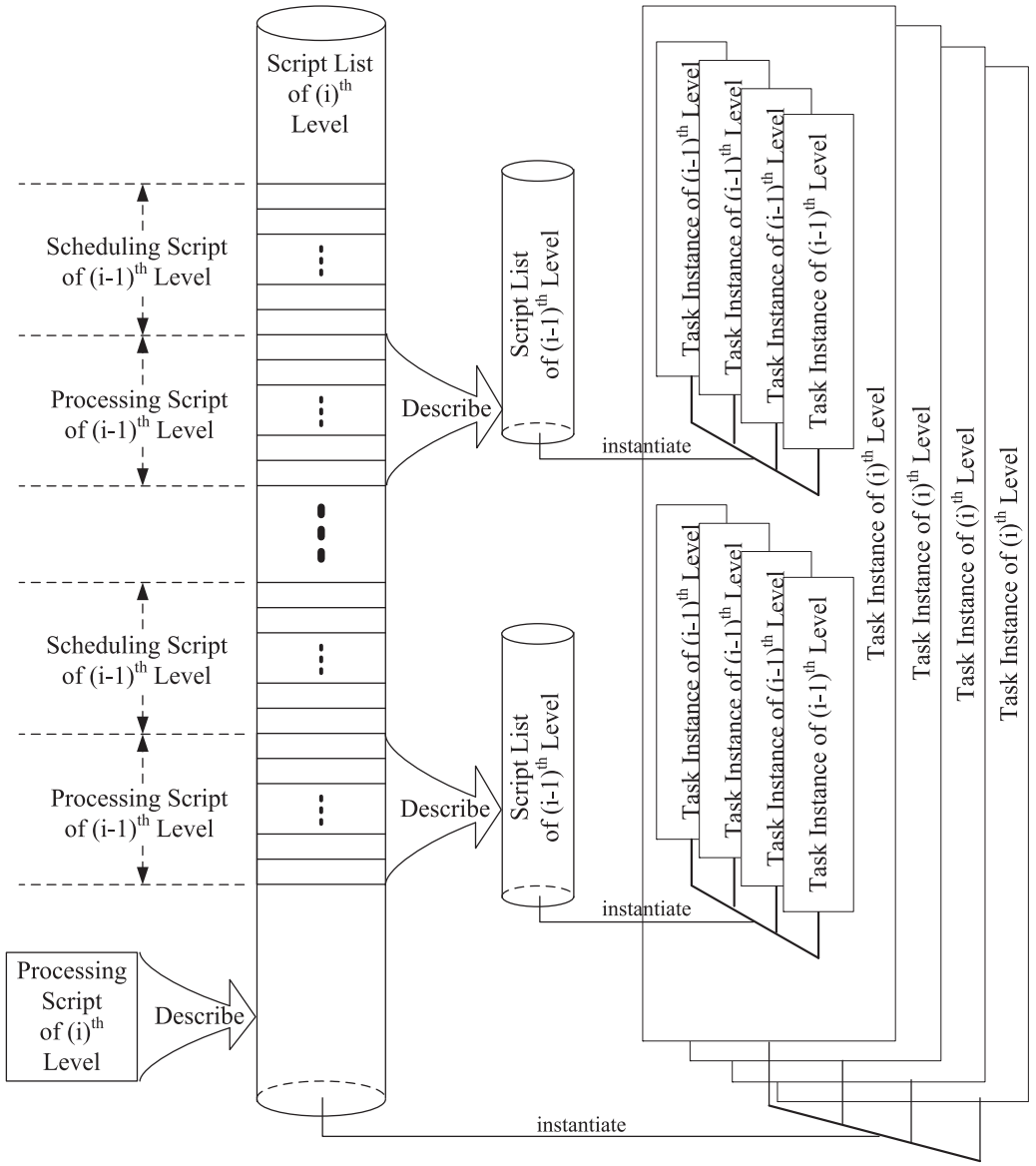


Fig. 2. Task-organization submodel.

map to the consumption procedures on both itself ($SL^i$) and the associated $PS^i$ that describes it.

- *$PS^i$ Consumption*. Logically, a group of parallel $TI^i$s are dispatched to an available $[PE]^i$ and enqueued for execution. Note that the availability here means that the $[PE]^i$ has sufficient capacity to accept one more $TI^i$ group. Physically, the associated $PS^i$ is parsed by a $TU^{i-1}$ inside the $PE^i$ to make sure that all the task information with respect to $SL^i$ is acquired and stored in the $TB^{i-1}$ for context maintenance.
- *$SL^i$ Consumption*. Each $SL^i$ consists of $SS^{i-1}$s and $PS^{i-1}$s. $SS^{i-1}$s are consumed by $SE^{i-1}$s (inside the $PE^i$), while $PS^{i-1}$s are consumed by $[PE]^{i-1}$s (inside the $PE^i$).

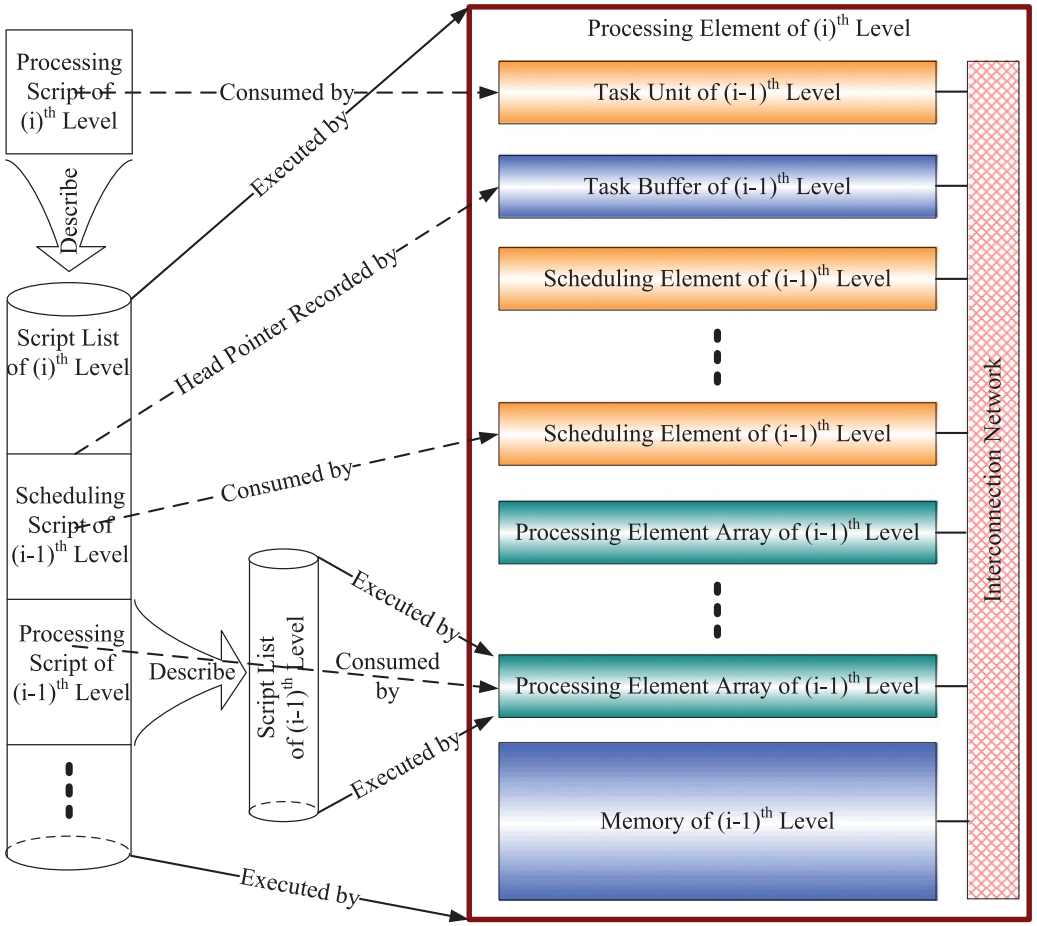


Fig. 3. Task-execute submodel.

The execution of one $SL^i$ goes through several pipeline stages. We define three baseline pipeline stages, as follows.

- *Context Allocation*. The $TU^{i-1}$ creates the execution context for the $SL^i$ by parsing its associated $PS^i$. In this stage, unoccupied (special-purpose) counters/stacks/buffers/registers are assigned to the $SL^i$ for holding its context information.
- *Script Consumption*. The $SE^{i-1}$s and $PE^{i-1}$s execute certain operations driven by the $SS^{i-1}$s and $PS^{i-1}$s within the $SL^i$.
- *Context Deallocation*. The $TU^{i-1}$ deallocates the execution context for the $SL^i$ by monitoring the end of its SC consumption. In this stage, associated (special-purpose) counters/stacks/buffers/registers are freed from occupation and recycled for serving the next $SL^i$ and $PS^i$ pair.

### 2.4. Task-Schedule Submodel

The task-schedule submodel of GPGPU is a hierarchical mapping model from task topology to hardware structure, as shown in Figure 4. The task topology is a hierarchical topology of nested TIs. The hardware structure is a hierarchical structure of nested components. Since each group of TIs is dispatched to one [PE], it is a common condition

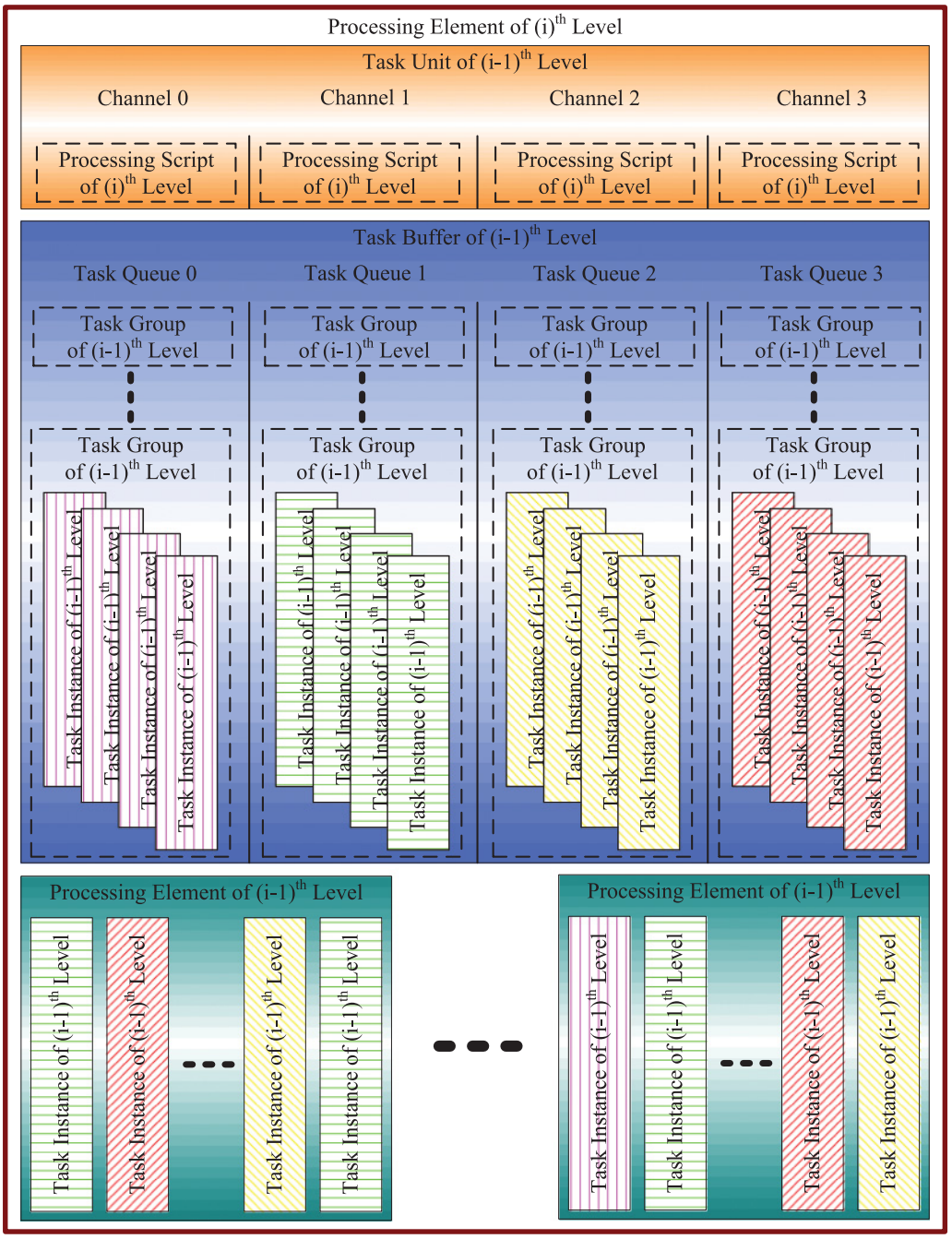


Fig. 4. Task-schedule submodel.

that multiple TIs (from one or multiple TI groups) are assigned to one PE. Therefore, the PE may allow for these TIs going through in a pipelined or parallel manner. The purpose of the task schedule is to retain the sequential execution of dependent workloads and out-of-order execution of independent ones. In order to achieve this purpose, the task-schedule submodel has to support four features:

- *Multiple Working Channels*. A TU handles multiple logical channels. Each task group is assigned a channel ID, which is enclosed in the PS. The TIs going through a common channel are dispatched in strict sequential order, while the TIs going through different channels are dispatched in any order. Note that, under a restricted condition in which the TU has only one hardware channel, multiple logical channels share one hardware channel in a multiplexing manner [NVIDIA 2012b].
- *Multiple FIFO Queues*. A TB maintains multiple First In First Out (FIFO) task queues. Each queue is associated with one logical channel. As a dual to logical channel, TI groups going through a common queue are dispatched in strict sequential order, while the TI groups going through different queues are dispatched in any order. The TB has limited capacity. Its capacity is measured by width and depth. The width of a TB is defined as the count of queues. The depth of a TB is defined as the maximal count of TI groups that can stay simultaneously inside a queue.
- *Intrinsic Synchronization*. Although TIs of a common TI group run asynchronously, they are intrinsically synchronized at the start and end of the TI group execution. For example, consider three consecutive TI groups sharing the same channel ID. The first TI of the second TI group cannot be executed until all the TIs of the first group finish execution, the first TI of the third TI group cannot be executed until all the TIs of the second TI group finish execution, and so on.
- *Hardware Multithreading*. The sequential order execution mechanism prevents the overlap execution between dependent workloads. This guarantees that all the workloads already dispatched are independent of each other. The [PE](s) switch among residing tasks in a multiplexing manner and expose a hardware multithreaded behavior. Stall cycles of certain workloads are hidden by the execution cycles of other workloads, thus lead to a high hardware occupancy.

### 2.5. Data-Move Submodel

The data-move submodel of GPGPU is a hierarchical mapping model from access distribution to memory hierarchy, as shown in Figure 5. In the previous sections, we do not differentiate "processing elements for computing" and "processing elements for data movement," but rather treat them uniformly as "processing elements." Since this section focuses on the movement of data, we need to treat them separately. Further illustration is based on an assumption that all the data is initially stored in the memory of the topmost level (i.e., level $(d-1)$). In what follows, we provide a set of definitions for explaining the data-move submodel.

- *Processing Element for Computing ($PE_C$)*. A type of PE that executes computing operations. An array of $PE_C$s collaborating to execute parallel-computing operations is denoted as $[PE_C]$. A $[PE_C]$ may contain one or multiple $PE_C$s. It can only access memory of the current level.
- *Processing Element for Data Movement ($PE_D$)*. A type of PE that executes data-movement operations. An array of $PE_D$ collaborating to execute data-movement operations is denoted as $[PE_D]$. A $[PE_D]$ may contain one or multiple $PE_D$s. It can access memory of the current level and that of a higher level. $PE_D$s are the exception of hardware components that break the locality rule.
- *Processing Script for Computing ($PS_C$)*. A type of PS that contains information for describing a computing task. A $PS_C$ is logically executed by a $PE_C$ (array).

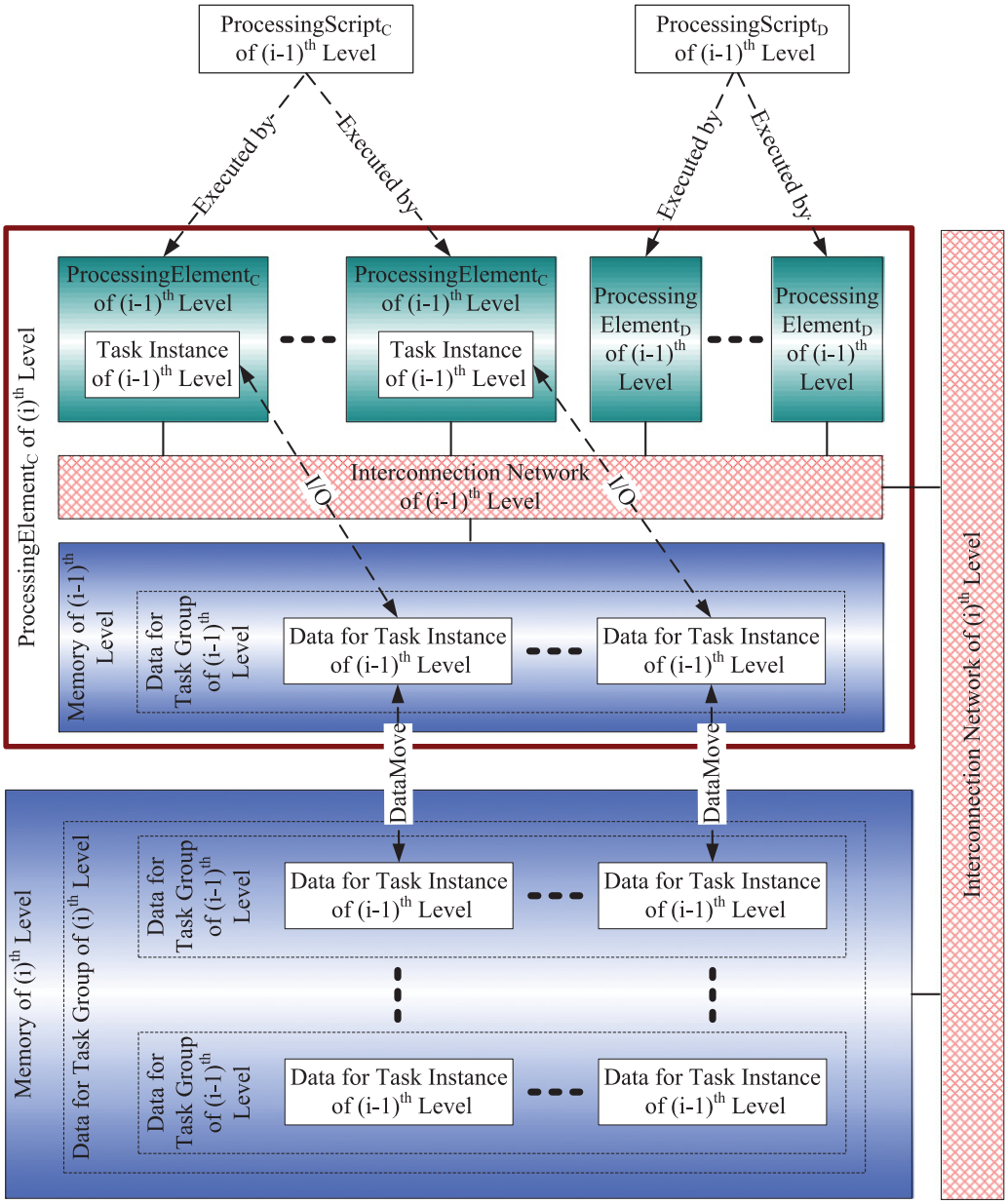


Fig. 5. Data-move submodel.

- *Processing Script for Data movement ($PS_D$)*. A type of PS that contains information for describing a data-movement task. A $PS_D$ is logically executed by a $PE_D$ (array).

A computing TI cannot execute unless its data is in position beforehand. In the data-move submodel, "in position" means that the TI and its data (operands) are residing on hardware components of the same level for efficient interaction. We examine three types of data movements in this section. A Stage-In movement transfers data from

high-level memory (memory$^{i+1}$) to low-level memory (memory$^{i}$). A Stage-Out movement does that in the opposite way. An equipotent movement copies data within the same memory. From the viewpoint of data-movement, the computing pattern of the GPGPU is essentially based on resource reservation. A baseline computing procedure consists of three baseline steps: Stage-In, Compute, and Stage-Out. Note that multiple Compute steps may share one Stage-In/Stage-Out step.

- *Stage-In*. A $PE^i_D$ (array) is driven by a $PS^i_D$ to facilitate a data transfer from memory$^{i+1}$ to memory$^{i}$.
- *Compute*. A $PE^i_C$ (array) is driven by a $PS^i_C$ to execute a $TI^i$ group. The $TI^i$s consume operands and produce results.
- *Stage-Out*. A $PE^i_D$ (array) is driven by a $PS^i_D$ to facilitate a data transfer from memory$^{i}$ to memory$^{i+1}$.

## 3. A CASE STUDY ON CUDA

In this section, we demonstrate on how to employ our model to interpret the behavior of GPGPU. There are two representative development platforms for coding GPGPU tasks: OpenCL and CUDA. OpenCL hides the low-level interface for code transplanting across different architectures (such as CPU and GPU). CUDA provides the low-level interface for code optimization on CUDA-compliant device architectures. From the perspective of application optimization, it would achieve better performance if the software structure and hardware architecture are coupled tightly. This is especially true when coding with CUDA on CUDA-compliant GPUs. Therefore, we choose CUDA to do the case study in this article. Note that our model also applies to OpenCL programs running on OpenCL-compliant GPUs, including both workstation devices such as AMD FirePro [Wallossek 2014] and Intel Haswell [Hammarlund 2013], or mobile devices such as Qualcomm Adreno [Qualcomm 2014], Imagination PowerVR [Voica 2014], and ARM Mali [Smith 2014].

Since it is unlikely that the GPU manufacturers would ever openly provide in-depth design details [Voicu 2010], we have to follow the way of Brunie et al. [2012], Fung and Aamodt [2011], and Gebhart et al. [2011] to define a conceptual CUDA-compliant GPU by considering details found in official patents/guides and research papers. In this section, CUDA-related terminologies commonly accepted in these patents/guides/papers are also retained, such as block, warp, thread, SM, warp scheduler, and shared/global memory. In addition, we assume that all general-purpose computation is undertaken by the GPU. Heterogeneous computation (which leverages workloads on both the CPU and GPU) is beyond the scope of this article. Table II presents a brief overview of general-purpose computing on a conceptual GPU.

### 3.1. Task Organization

A CUDA program takes a two-level code structure containing one host command script and multiple kernels [NVIDIA 2015a].

- *Host command script*. The host command script ($SL^2$) is a script of Push Buffer Streams (PBSs), in which each PBS consists of a script of Push Buffer Commands (PBCs) [Duluk Jr. et al. 2009]. Note that one PBC is the implementation of $SC^1$ in our model. There are three types of PBSs: computing PBS ($PS^1_C$), guiding the computing; datamove PBS ($PS^1_D$), guiding the data movement; and control PBS ($SS^1$), guiding the context maintenance. Table III lists exemplary PBCs, along with their function descriptions. For example, consider the "Launch" PBC. Each time a "Launch" PBC is consumed, one kernel block gets dispatched to an available Streaming Multiprocessor (SM) for execution.

Table II. A Brief Overview of General-Purpose Computing on a Conceptual GPU

| Hardware Component | Script Command |
|---|---|
| $[PE_C^0]$=ALU array | $PS_C^0$=Computing instruction |
| $[PE_D^0]$=LSU array | $PS_D^0$=Data access instruction |
| $TB^0$={Program counters, reconvergence stacks, barrier counters, instruction buffer, scoreboard buffer} | N/A |
| $memory^0$=Local/shared registers | N/A |
| $SE^0$=Warp scheduler | $SS^0$=Control instruction |
| $TU^0$=Thread controller | $SS^1$=Control PBS |
| $[PE_C^1]$=SM array | $PS_C^1$=Computing PBS |
| $[PE_D^1]$=Data Transfer Engine array | $PS_D^1$=Datamove PBS |
| $TB^1$={State registers, reference counters, push buffer} | N/A |
| $memory^1$=Local/global memory | N/A |
| $SE^1$=CUDA work distributor | $SS^1$=Control PBS |
| $TU^1$=GigaThread Engine | $SS^2$=GPU driver call |

Table III. Exemplary Push Buffer Commands

| Push Buffer Command | Function Description |
|---|---|
| DefineSemaphore | Define a semaphore variable at a given memory location. |
| AcquireSemaphore | Wait for a previous kernel to release a semaphore. |
| SetLanchID | Associate the blocks of the current kernel to a reference counter. |
| SetRefCntValue | Set a reference counter value to be used by ResetRefCnt and WaitRefCnt. |
| ResetRefCnt | Wait for the blocks of the previous kernel to complete. |
| WaitRefCnt | Wait for the blocks of the current kernel to complete. |
| Launch | Launch a block of the current kernel for execution. |
| SetParameterSize | Allocate space for blocks to accept kernel parameters. |
| Parameter | Transfer the kernel parameters to the blocks when launched. |

Table IV. Exemplary Kernel Instructions

| Kernel Instruction | Function Description |
|---|---|
| IADD/IMUL/IMAD | 32b integer addition/multiply/multiply–addition |
| FADD/FMUL/FFMA | 32b floating point addition/multiply/multiply–addition |
| DADD/DMUL/DFMA | 64b floating-point addition/multiply/multiply–addition |
| LD/ST/TEX | 32b load/store/texture |
| SHL/SHR | 32b shift left/right |
| BRA/JMP/CAL/RET | branch/jump/call/return |
| BAR/MEMBAR | synchronization on computing/data access |

- *Kernel*. A kernel ($SL^1$) is a script of kernel instructions. Note that one kernel instruction is the implementation of $SC^0$ in our model. There are three types of kernel instructions: computing instruction ($PS_C^0$, which contains one instruction only), such as "DADD" or "IMUL"; data-access instruction ($PS_D^0$, which contains one instruction only), such as "LD" or "TEX"; and control instruction ($SS^0$, which contains one instruction only), such as "BRA" or "BAR." Table IV lists exemplary kernel instructions, along with their function descriptions [NVIDIA 2012a]. For example, consider the "FMUL" instruction. Each time an "FMUL" instruction is consumed, one "FMUL" gets dispatched to an available ALU array for execution.

Like the code structure, a CUDA process takes a two-level task topology. At the first level, a CUDA process contains one or multiple kernels. Each kernel ($SL^1$) instantiates a group of kernel blocks, for which each block ($TI^1$) runs operations of one kernel. At the 0th level, a kernel contains multiple instructions. Each instruction ($SL^0$) instantiates a group of threads, for which each thread ($TI^0$) runs the operation of one instruction.

The task organization reflects a design trade-off on the task granularity. First, the task of a kernel might be too large to fit the capacity of SMs. Partitioning a big workload (kernel grid) into small independent shares (kernel blocks) guarantees that each share is small enough to reside on one SM. Furthermore, the number of ready-to-go kernels is usually less than the SM count within a GPU. Such partitioning feeds each SM with multiple shares, thus remarkably increasing the hardware occupancy. Moreover, these blocks asynchronously run on different SMs, or some run while the rest wait unassigned. The kernel-scoped blocks have to be independent from each other, thus interblock interactions are not allowed.

### 3.2. Hardware Structure

The GPU follows a two-level structure of nested components. First, the atomicity of the GPU is broken into hardware components of the first level. GigaThread Engine (GTE) [NVIDIA 2012b] accepts calls to the GPU and creates $TI^1$s according to kernels. $TB^1$ stores state information for residing kernels. CUDA Work Distributor (CWD) [NVIDIA 2012b] consumes control PBS and maintains the execution order of kernel blocks and data transmission. Data Transfer Engine (DTE) consumes datamove PBS and transfers data between the host memory and device memory. SM array consumes computing PBSs and undertakes the computation of kernel blocks. Local/global memory stores "in position" data for kernel blocks during computation.

The atomicity of the SM is broken into hardware components of the 0th level. Thread controller [Donovan and Lindholm 2010] accepts calls to the SM and creates $TI^0$s according to kernel instructions. $TB^0$ stores state information for residing instructions. Warp scheduler [NVIDIA 2012b] consumes control instructions and maintains the execution order of computing/data access instructions. LSU arrays consume data access instructions and transfer data between local/global memory and local/shared registers. ALU arrays consume computing instructions and undertake the computation of kernel instructions. Local/shared registers store "in position" data for kernel instructions during computation.

In what follows, we compare the components of two levels in a pairwise manner, and explain implementation issues that deserve attention, especially in macro-/micro-architecture design.

- *ALU array versus SM array*. There are many ALU arrays that process computing instructions. For instance, CUDA core arrays process "integer/floating-point/bitwise/convert" instructions [Siu and Oberman 2007]. Special Function Unit (SFU) arrays process "sine/cosine/reciprocal/square root" instructions [NVIDIA 2009b]. Double-Precision Unit (DPU) arrays process double-precision instructions [Oberman et al. 2012; NVIDIA 2012b]. Each ALU array is associated with a common program counter [Lindholm et al. 2010] so that all the ALUs within have to operate synchronously in an instruction stepwise manner. Likewise, there is an array of SMs that process all the residing kernel blocks. They operate asynchronously in a kernel stepwise manner, thus interblock operations are prohibited within a kernel.
- *LSU array versus DTE array*. LSU arrays transfer data between local/global memory and local/shared registers. They work just like ALU arrays: all the LSUs within the array share one program counter and operate synchronously in an instruction stepwise manner. Besides LSU arrays, there are Texture Unit arrays that are graphic-oriented but can also serve as special LSU arrays in general-purpose computing with optimized performance for certain data accesses [Donovan and Lindholm 2010]. Likewise, there is an array of DTEs that operate asynchronously and serve bidirectional data transfers between host memory and device memory.

- *Local/shared registers versus local/global memory*. The local registers and local memory store task-private data. They are allocated/deallocated in a "TI"-wise manner. The shared registers (also called shared memory) and the global memory store task-public data. They are allocated/deallocated in a "TI group"-wise manner. The shared registers are used by block-scoped kernel threads to exchange data with each other. They act as a programmable L1 cache, thus play an important role in code optimization.
- *$TB^0$ v.s. $TB^1$*. $TB^0/TB^1$ is a logical component for maintaining the context of $TI^0$s/$TI^1$s. A task buffer physically consists of a set of special buffers/stacks/counters/registers that cooperate to build the task pool. The program counter array traces the address pointers for the current instructions requiring fetch operation [Lindholm et al. 2010]. The scoreboard buffer traces register usage status in order to resolve register collision [Coon et al. 2008]. The reconvergence stack array pushes/pops context tokens for control-transfer operations [Coon and Lindholm 2008]. The barrier counter array stalls the arrival TIs at a certain barrier until synchronization is achieved [Nickolls and Lew 2011]. The instruction buffer holds and prioritizes the current instructions requiring execution [Mills et al. 2011]. The state register array records the state information for residing kernels [Duluk Jr. 2010], such as block size, instance indices, and valid state. The reference counter array monitors the execution procedure of residing kernels by counting the assignment and completion of their kernel blocks [Duluk Jr. et al. 2009]. The push buffer works as a multiple FIFO queue to store host command scripts (PBSs) for the GPU. It is an exception of the component unit, since it is in fact implemented by software instead of hardware (physically resides in host memory).
- *Warp scheduler versus CWD*. The warp scheduler/CWD is also a logical component. It physically consists of a set of control logics that manipulate the special buffers/stacks/counters/registers in $TB^0/TB^1$ driven by control instructions/PBSs. The key function of the warp scheduler/CWD is to change the context of the task pool according to control instructions/PBSs to guarantee the executing sequence of instructions/kernels, that is, dependency check and task dispatch. One SM/GPU may be equipped with one or multiple warp schedulers/CWDs that manage a partition of the workspace, respectively. CWD also endows GPU with the capability of generating a recursive workload for itself without external operation [NVIDIA 2012b]. This feature allows for the direct invocation from one kernel to another kernel, thus supports more complicated task organization.
- *Thread Controller versus GTE*. The thread controller/GTE plays the role of front end for the SM/GPU. On one hand, it monitors the working state of the SM/GPU and dynamically judges whether the SM/GPU can accept more TIs for execution. On the other hand, it allocates resources for arrival TIs and accommodates them in the task pool.

### 3.3. Task Execution

The hardware structure focuses on the functions of respective components, while the task execution focuses on the cooperation of components to move forward workloads (TIs). The mapping from code structure to hardware structure has two levels. At the first level, a CUDA process breaks the computation flow into kernels, in which each kernel instantiates a group of kernel blocks ($TI^1$s). At the 0th level, a kernel breaks the computation flow into instructions, in which each instruction instantiates a group of instruction threads ($TI^0$s).

- *Kernel block execution*. A kernel block is first created by a GTE while the GTE acquires the state information by parsing associated computing PBSs. Then, the kernel

block waits in the $TB^1$ until a CWD dispatches it to an available SM for execution. A reference counter monitors the running state of the kernel block by logging the assignment and completion of its execution. A portion of the SM computing resource is allocated to the kernel block during its execution. Finally, the SM flushes out the kernel block and recycles the resources for newly arrived blocks.

- *Instruction thread execution*. When a kernel block is assigned to an SM, the thread controller breaks the atomicity of the kernel. The computation of a kernel block is treated as a sequence of computing instructions, in which each computing instruction is executed by a group of kernel threads. Each computing instruction thread waits in the $TB^0$ until a warp scheduler dispatches it to an available ALU for execution. A program counter logs the running of instructions. The ALU advances the instruction thread going through its micro-pipeline [Galal and Horowitz 2011; Donovan and Lindholm 2010; Siu and Oberman 2007; Oberman et al. 2012] until flush-out.

Since the execution time of each kernel is application determined, we need to be concerned only with the execution time of a single instruction. In the GPU, the time is generally measured by clock cycles. The execution cycles for each instruction is influenced by many factors. First, instructions go through different execution stages according to their instruction types. For example, a control instruction goes through logic circuits within a warp scheduler. A computing/data access instruction goes through logic circuits (micro-pipeline) of both a warp scheduler and a PE. Second, the going through latency diverges across different PE types since PEs have different micro-pipeline structures [Galal and Horowitz 2011; Donovan and Lindholm 2010]. For example, a "SIN" instruction goes through the SFU micro-pipeline. And a "DADD" instruction goes through the DPU micro-pipeline. Third, the going through latency diverges across different instruction types since each type of instruction goes through a respective route within a PE micro-pipeline. For example, an SFU array consumes more clock cycles processing a reciprocal instruction than a cosine instruction. Research projects such as AsFermi [2014] and Wong et al. [2010] represent efforts on estimating micro-pipeline latency for respective kernel instructions.

### 3.4. Task Schedule

In the previous section, we focus on individual task instance by addressing how it is forwarded among components until completing execution. In this section, we are concerned with the scheduling of all residing task instances by addressing the order of their dispatch and execution. First, we discuss the scheduling of kernels in a GPU. Each kernel is assigned a stream (logical channel) ID by the CUDA program [NVIDIA 2015a]. One reference counter is associated with this ID, and exclusively serves all the kernels sharing the ID. Each time that the reference counter is allocated to one kernel, the stream is flagged as busy and the blocks of that kernel are dispatched to SMs one after another (driven by "Launch" PBCs). A reference counter contains one launch counter and one complete counter [Duluk Jr. et al. 2009]. The launch counter is increased by one when one kernel block gets assigned to an SM, while the complete counter is increased by one when one kernel block finishes execution. The kernel blocks run asynchronously but are intrinsically synchronized at the start and end of kernel execution. When all these blocks finish execution, the stream is flagged as vacant and the reference counter is recycled for serving the next kernel in turn. An array of reference counters guarantee that the kernels with a common stream ID are served in strict FIFO sequence, while the kernels with different IDs can be served out-of-order. This mechanism realizes the parallel execution of independent kernels (blocks) as well as the sequential execution of dependent kernels.

Furthermore, the atomicity of a kernel block is broken when it gets assigned to an SM. A kernel block is treated as a matrix of instruction threads. Each column (of the matrix) represents an individual instance of the kernel instruction sequence with a respective thread ID. Each row represents a group of threads that run a common instruction. One SM may accept multiple kernel blocks residing, that is, there may be multiple such matrices served at the same time. One program counter is associated with several columns (threads) and advances their execution procedure in a partially row-stepwise manner. More specifically, let us assume that one SM has $k$ residing blocks and each block has $t$ threads, while each [PE] has $p$ PEs and each PE accepts one thread entering its micro-pipeline in each cycle.

The count of program counters reflects a design trade-off. At one extreme, setting $k$ counters means that all block-scoped threads share one counter; thus, they share the dispatch and execution of each instruction. That is, each [PE] will cost $t/p$ cycles to load one instruction. Note that $t$ is usually big (i.e., $t = 256$) and $p$ is usually small (i.e., $p = 16$). Thus, $t/p$ is a large number that leads to poor multithreaded performance, especially when processing dependent instructions. At the other extreme, setting $k * t$ counters means that each thread occupies one counter, respectively. That is, each [PE] contains one PE only ($p = 1$) and each PE cost 1 cycle to load one instruction. $k * t$ is also a large number that results in a remarkable rise in the area consumption of control circuits. In such case, the basic law of GPGPU (dedicating more transistors to arithmetic logics) is violated since the ratio of control over computing is overturned.

The concept of "warp" (and "Wavefront" for OpenCL) is introduced to balance these two extreme conditions. From the viewpoint of task, the threads are grouped into warps. Thirty-two threads share one program counter, thus share the dispatch and execution of each instruction. From the viewpoint of hardware, $k * t/32$ program counters occupy an acceptable circuit area. Each [PE] consumes $32/p$ cycles loading an instruction to maintain a multithreaded performance. The 32 threads in a warp are partitioned into $32/p$ waves (shares) to feed the [PE] but are intrinsically synchronized at the start and end of instruction execution. A "warp" can be considered as a logical channel in our task-schedule submodel. The program counters are handed over among warps like reference counters among kernels. An array of program counters guarantee that the instructions of one warp are served in strict FIFO sequence, while arbitration mechanism achieves the out-of-order selection among instructions of different (ready) warps [Lindholm et al. 2010]. Like the stream mechanism, the warp mechanism realizes the parallel execution of kernel warps (threads) as well as the sequential execution of dependent instructions.

### 3.5. Data Movement

Like the two-level hardware structure, the data movements of GPU have two granularity levels. At the first level, the dataset to be processed (including both input and output) by a kernel is considered atomic, denoted as a kernel dataset. A kernel dataset is assumed to reside on host memory before the execution of a kernel (or a sequence of consecutive kernels when multiple Compute steps share one Stage-In/Stage-Out step). First, one DTE ($PE^1_D$) consumes a datamove PBS ($PS^1_D$) and launches a Stage-In movement (from host memory to device memory), so that the kernel dataset becomes "in position" for kernel operation. Furthermore, the kernel blocks run on SMs and access the kernel dataset in device memory. Finally, one DTE consumes a datamove PBS and launches a Stage-Out movement (from device memory to host memory), so that the kernel dataset is (fully or partially) returned for result output.

At the 0th level, the dataset to be processed (including both input and output) by an instruction warp is considered atomic, denoted as a warp dataset. A warp dataset is assumed to reside on device memory before the execution of an instruction (or a sequence of consecutive instructions when multiple Compute steps share one

Stage-In/Stage-Out step), since it is a part of the kernel dataset that stays "in position" in device memory for kernel operation. First, one LSU array ($[PE_D^0]$) consumes a data load instruction ($PS_D^0$) and launches a Stage-In movement (from device memory to local/shared registers), so that the warp dataset becomes "in position" for instruction operation. Furthermore, the instruction threads run on an ALU array and access the warp dataset in local/shared registers. Finally, one LSU array consumes a data-store instruction and launches a Stage-Out movement (from local/shared registers to device memory), so that the warp dataset is (fully or partially) returned for result output.

The register array has a short-access latency since it stays inside the GPU chip, while the device memory has a long-access latency since it stays outside the GPU chip. A buffer mechanism is introduced to balance the access speed between the two, that is, an L2/L1 cache. Caches of any type are transparent to task instances. They are considered a hot subset of device memory with superior access performance from the viewpoint of a task instance. An L2 cache contains an array of cache slices, where each slice physically resides inside one DRAM controller of device memory [Glasco et al. 2011]. The L2 cache incorporates with Raster Operation Partition units to achieve advanced global memory operations such as access coalescing and atomic operation [Buck et al. 2009]. The L2 cache buffers both instruction and data, and is shared by all the SMs. In contrast, there is an L1 instruction cache and an L1 data cache inside each SM [Moy and Lindholm 2006]. A hierarchical caching mechanism reflects the fusion trend between the CPU and GPU.

## 4. FINAL REMARKS

There is a wealth of literature describing GPU design practices along with case studies and performance results. But there is much less information available to the GPGPU community about how these special processors actually function to execute tasks. This article presents a detailed white box description for a better understanding of GPGPU. Key issues of most concerns are touched, including task organization, hardware structure, scheduling mechanism, execution mechanism, and memory access. In what follows, we enumerate key characteristics with respect to these issues concerning GPGPU of the current generation while presenting our vision for future paradigms.

- The task organization of GPGPU contains multiple levels. In each level, a group of task instances share a common code (script list) but operate on respective index-based memory addresses. Its essence is hierarchical single-instruction-multiple-data. We can expect the programming model of future-generation support to be more than 2 levels, in which all levels share a common code representation and processing hardware. For example, each script list is labeled with a level index, and a compliant hardware can allow task instances of multiple levels to reside on it simultaneously, even if some are subtasks of others. The recursive kernel invocation of CUDA [NVIDIA 2012b] takes a remarkable step towards this direction.
- The hardware structure of a GPU contains multiple levels. In each level, scheduling elements manipulate the task pool, processing elements undertake computations, and the memory component serves data accesses. Its essence is hierarchical Von Neumann (instruction driven and detached data-control-ALU). We can expect the GPU of future generations to move from instruction driven to data driven, or even break the borderline between computation and storage: more specifically, an architecture that mixes ALUs and memory cells together in which an access procedure is also a computing procedure and vice versa. A circuit like this might have a similar architecture to a neuromorphic chip [Hof 2014] and process multiple dataflows simultaneously in an efficient manner.

- The schedule mechanism of GPGPU contains multiple levels. In each level, parallel task instances are dispatched out of order, while the in-order execution within each task instance is maintained. Its essence is hierarchical multithreading. We can expect the GPU of future generations to support cross-level scheduling, in which all levels share a common dispatch hardware. More specifically, each task instance is labeled with a task index and a level index, and the dispatch components can allow task instances of different levels to be dispatched in a hybrid manner.
- The execution mechanism of GPGPU can be considered as a combination of both parallel execution and pipelined execution. For a task instance of any granularity to be atomic, its execution goes through several pipeline stages one after another. However, when we break the atomicity of the task, its execution is achieved by the parallel execution of its subtasks. These two execution patterns seamlessly fuse to pursue the extreme of hardware power. Its essence is throughput computing. In upcoming GPUs, this mechanism may persist over generations.
- The memory access of GPGPU contains multiple levels. In each level, a Stage-In/Out procedure acts as a sufficient and necessary condition for operand feed/results collection. Its essence is hierarchical move-data-to-computing, which derives from the inconsistency between data distribution and access distribution. We can expect the GPU of future generations to reduce or even remove the Stage-In/Out overhead by reversing move-data-to-computing, namely, move-computing-to-data. For example, an architecture mixes ALUs and interconnection networks together, in which a transmission procedure is also a computing procedure and vice versa. A circuit like this might borrow the success of multicore and Hadoop [Murthy et al. 2011] and expose a multiple-instruction-multiple-data behavior.

GPUs are born to equip a large matrix of programmable processing elements. One can get a taste of high-performance computing by harnessing one (GPU) chip in a desktop/laptop computer, or grasp impressive super power by clustering chips together. This is the driving force for the popularity and development of GPGPU. Hardware superiority brings not only opportunities but also challenges. From our point of view, four problems may hinder the development of GPGPU.

- The computing resource per chip increases remarkably. The key to high performance per watt still relies on new sharing patterns to keep devices as busy as possible. Device virtualization as a promising solution for green computing has been introduced to partition the horsepower of a GPU chip/cluster [Herrera 2015]. But more efforts are needed before it becomes a widely accepted industry standard.
- OpenCL has shown its potential not only as a promising industry standard by obtaining support from devices of both desktop and mobile markets, but also as a cross-platform toolkit aggregating gains from both scientific research and commercial applications. However, there are still remarkable factors that slow down the pace of its popularization. First, domain-specific users are accustomed to the massive algorithm libraries already built with CUDA. Therefore, it would require a huge amount of work to transplant these libraries from the CUDA version to the OpenCL version. Automatic converting tools might be needed, but the users are not guaranteed to accept the conversion. Second, respective business corporations might intentionally shield the support to OpenCL for the purpose of selling their own tools with similar functions, for example, Android Renderscript [Guihot 2012].
- The descriptive model proposed in this article is a qualitative model that depicts the behavior of GPGPU. The community still needs a quantitative model that depicts the computing complexity of current and upcoming GPUs. Classic quantitative models such as PRAM [Fortune and Wyllie 1978], BSP [Valiant 1990], and LogP [Culler et al. 1993] do not match specific features of GPGPU. Later attempts (e.g., Valiant

[2011] and Kothapalli et al. [2009]) have emerged, but they may not be sufficient to cover evolving trends. By further incorporating numerical parameters that reflect physical constraints of a specific GPU architecture, our model can act as a qualitative model targeting increased programmability and optimized performance for current and upcoming GPGPU paradigms.

- Human resource departments in the software market are going to encounter a shortage of algorithm engineers targeting GPGPU solutions, who stand at the crossroad among hardware operations, software interfaces, as well as algorithmic design and analysis of domain-specialized, data-parallel computing.

## REFERENCES


AMD. 2014. AMD Accelerated Parallel Processing OpenCL$^{TM}$ User Guide, Version 1.0. Retrieved Aug 10, 2015 from http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_OpenCL_Programming_User_Guide2.pdf.

Asfermi. 2014. Asfermi: An Assembler for the NVIDIA Fermi Instruction Set. Retrieved February 24, 2016 from http://code.google.com/p/asfermi/.

Nicolas Brunie, Sylvain Collange, and Gregory Diamos. 2012. Simultaneous branch and warp interweaving for sustained GPU performance. In *Proceedings of the 39th International Symposium on Computer Architecture (ISCA'12)*. Portland, Oregon, 49–60. DOI:http://dx.doi.org/10.1109/ISCA.2012.6237005.

Ian A. Buck, John R. Nickolls, Michael C. Shebanow, and Lars S. Nyland. 2009. Atomic Memory Operators in a Parallel Processor. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7627723.html. United States Patent No. 7,627,723 B1 (Assignee NVIDIA Corp.), Filed Sept. 21, 2006, Issued Dec. 1, 2009.

Brett W. Coon and John Erik Lindholm. 2008. System and Method for Managing Divergent Threads in a SIMD Architecture. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7353369.html. United States Patent No. 7,353,369 B1 (Assignee NVIDIA Corp.), Filed July 13, 2005, Issued Apr. 1, 2008.

Brett W. Coon, Peter C. Mills, Stuart F. Oberman, and Ming Y. Siu. 2008. Tracking Register Usage During Multithreaded Processing Using A Scoreboard Having Separate Memory Regions and Storing Sequential Register Size Indicators. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7434032.html. United States Patent No. 7,434,032 B1 (Assignee NVIDIA Corp.), Filed Dec 13, 2005, Issued Oct 7, 2008.

David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. 1993. LogP: Towards a realistic model of parallel computation. *ACM SIGPLAN Notices* 28, 7, 1–12. DOI:http://dx.doi.org/10.1145/173284.155333

Walter E. Donovan and John Erik Lindholm. 2010. Programmable Graphics Processor for Generalized Texturing. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7852346.html. United States Patent No. 7,852,346 B1 (Assignee NVIDIA Corp.), Filed Nov. 22, 2005, Issued Dec. 14, 2010.

Jerome F. Duluk Jr. 2010. Predicated Launching of Compute Thread Arrays. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7697007.html. United States Patent No. 7,697,007 B1 (Assignee NVIDIA Corp.), Filed July 12, 2006, Issued Apr. 13, 2010.

Jerome F. Duluk Jr., Stephen D. Lew, and John R. Nickolls. 2009. Counter-Based Delay of Dependent Thread Group Execution. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7526634.html. United States Patent No. 7,526,634 B1 (Assignee NVIDIA Corp.), Filed Sept. 27, 2006, Issued Apr. 28, 2009.

Kayvon Fatahalian and Mike Houston. 2008. A closer look at GPUs. *Communications of the ACM* 51, 10, 50–57. DOI:http://dx.doi.org/10.1145/1400181.1400197

Steven Fortune and James C. Wyllie. 1978. Parallelism in random access machines. In *10th Annual ACM Symposium on Theory of Computing (STOC'78)*. ACM, San Diego, CA, 114–118. DOI:http://dx.doi.org/10.1145/800133.804339. May 1–3.

Wilson W. L. Fung and Tor M. Aamodt. 2011. Thread block compaction for efficient SIMT control flow. In *Proceedings of the 17th International Symposium on High-Performance Computer Architecture (HPCA'17)*. San Antonio, TX, 25–36. DOI:http://dx.doi.org/10.1109/HPCA.2011.5749714. Feb 12–16.

Sameh Galal and Mark Horowitz. 2011. Energy-efficient floating-point unit design. *IEEE Transactions on Computers* 60, 7, 913–922. DOI:http://dx.doi.org/10.1109/TC.2010.121

Mark Gebhart, Daniel R. Johnson, David Tarjan, Stephen W. Keckler, William J. Dally, Erik Lindholm, and Kevin Skadron. 2011. Energy-efficient mechanisms for managing thread context in throughput

processors. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. San Jose, CA, 235–246. DOI:http://dx.doi.org/10.1145/2000064.2000093. June 4–8.

David B. Glasco, Peter B. Holmqvist, George R. Lynch, Patrick R. Marchand, James Roberts, and John H. Edmondson. 2011. System, Method and Frame Buffer Logic for Evicting Dirty Data from a Cache Using Counters and Data Types. Retrieved February 24, 2016 from http://www.freepatentsonline.com/8060700.html. United States Patent No. 8,060,700 B1 (Assignee NVIDIA Corp.), Filed Dec. 8, 2008, Issued Nov. 15, 2011.

Hervé Guihot. 2012. RenderScript. In *Pro Android Apps Performance Optimization*. Apress, 231–263. DOI:http://dx.doi.org/10.1007/978-1-4302-4000-6_9.

Per Hammarlund. 2013. 4th Generation Intel Core$^{TM}$ Processor, codenamed Haswell. Retrieved Aug 10, 2015 from http://www.hotchips.org/wp-content/uploads/hc_archives/hc25/HC25.80-Processors2-epub/HC25.27.820-Haswell-Hammarlund-Intel.pdf.

Alex Herrera. 2015. NVIDIA GRID vGPU: Delivering Scalable Graphics-rich Virtual Desktops. Retrieved Aug 10, 2015 from http://images.nvidia.com/content/pdf/grid/whitepaper/NVIDIA-GRID-WHITEPAPER-vGPU-Delivering-Scalable-Graphics-Rich-Virtual-Desktops.pdf.

Robert D. Hof. 2014. Neuromorphic chips. *MIT Technology Review* 117, 3, 56–59.

Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan, and Kannan Srinathan. 2009. A performance prediction model for the CUDA GPGPU platform. In *16th International Conference on High Performance Computing (HiPC'09)*. IEEE, Kochi, India, 463–472. DOI:http://dx.doi.org/10.1109/HIPC.2009.5433179.

John Erik Lindholm, Brett Coon, and Simon S. Moy. 2010. Across-Thread Out-of-Order Instruction Dispatch in a Multithreaded Microprocessor. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7676657.html. United States Patent No. 7,676,657 B2 (Assignee NVIDIA Corp.), Filed Oct. 10, 2006, Issued Mar. 9, 2010.

David Luebke and Greg Humphreys. 2007. How GPUs work. *IEEE Computer* 40, 2, 96–100. DOI:http://dx.doi.org/10.1109/MC.2007.59

Peter C. Mills, John Erik Lindholm, Brett W. Coon, Gary M. Tarolli, and John Matthew Burgess. 2011. Scheduler In Multi-threaded Processor Prioritizing Instructions Passing Qualification Rule. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7949855.html. United States Patent No. 7,949,855 B1 (Assignee NVIDIA Corp.), Filed Apr. 28, 2008, Issued May 24, 2011.

Simon S. Moy and John Erik Lindholm. 2006. Shader Cache Using a Coherency Protocol. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7103720.html. United States Patent No. 7,103,720 B1 (Assignee NVIDIA Corp.), Filed Oct 29, 2003, Issued Sep 5, 2006.

Arun C. Murthy, Chris Douglas, Mahadev Konar, Owen OMalley, Sanjay Radia, Sharad Agarwal, and Vinod K. V. 2011. Architecture of Next Generation Apache Hadoop Mapreduce Framework. Retrieved February 24, 2016 from https://issues.apache.org/jira/secure/attachment/12486023/MapReduce_NextGen_Architecture.pdf.

John Nickolls and William J. Dally. 2010. The GPU computing era. *IEEE Micro* 30, 2, 56–69. DOI:http://dx.doi.org/10.1109/MM.2010.41

John R. Nickolls and Stephen D. Lew. 2011. Parallel Data Processing Systems and Methods Using Cooperative Thread Arrays. Retrieved February 24, 2016 from http://www.freepatentsonline.com/y2011/0087860.html. United States Patent Application No. US2011/0087860 A1 (Assignee NVIDIA Corp.), Filed Dec. 17, 2010, Published Apr. 14, 2011.

NVIDIA. 2009a. OpenCL Programming Guide for the CUDA Architecture, Version 2.3. Retrieved February 24, 2016 from http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf.

NVIDIA. 2009b. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Retrieved February 24, 2016 from http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

NVIDIA. 2012a. Cuobjdump. Retrieved February 24, 2016 from http://www.ece.lsu.edu/koppel/gp/refs/cuobjdump.pdf.

NVIDIA. 2012b. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. Retrieved February 24, 2016 from http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

NVIDIA. 2014. Whitepaper: GeForce GTX 750 Ti: Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt. (Feb 2014). Retrieved February 24, 2016 from http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf.

NVIDIA. 2015a. CUDA C Programming Guide, Version 7.0. Retrieved February 24, 2016 from http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf.

NVIDIA. 2015b. CUDA Compiler Driver NVCC, Version 7.0. Retrieved February 24, 2016 from http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf.

NVIDIA. 2015c. Tuning CUDA Applications for Maxwell. Retrieved February 24, 2016 from http://docs.nvidia.com/cuda/pdf/Maxwell_Tuning_Guide.pdf.

Stuart Oberman, Ming Y. Siu, and David C. Tannenbaum. 2012. Fused Multiply-Add Functional Unit. Retrieved February 24, 2016 from http://www.freepatentsonline.com/8106914.html. United States Patent No. 8,106,914 B2 (Assignee NVIDIA Corp.), Filed Dec. 7, 2007, Issued Jan. 31, 2012.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. 2007. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum* 26, 1, 80–113. DOI:http://dx.doi.org/10.1111/j.1467-8659.2007.01012.x.

Qualcomm. 2014. The Rise of Mobile Gaming on Android: Qualcomm Snapdragon Technology Leadership. Retrieved Aug 10, 2015 from www.qualcomm.com/media/documents/files/the-rise-of-mobile-gaming-on-android-qualcomm-snapdragon-technology-leadership.pdf.

Ming Y. Siu and Stuart F. Oberman. 2007. Multi-Purpose Floating Point and Integer Multiply-Add Functional Unit with Multiplication-Comparison Test Addition and Exponent Pipelines. Retrieved February 24, 2016 from http://www.freepatentsonline.com/7225323.html. United States Patent No. 7,225,323 B2 (Assignee NVIDIA Corp.), Filed Nov. 10, 2004, Issued May 29, 2007.

Ryan Smith. 2014. ARMs Mali Midgard Architecture Explored. Retrieved Aug 10, 2015 from http://www.anandtech.com/show/8234/arms-mali-midgard-architecture-explored.

Leslie G. Valiant. 1990. A bridging model for parallel computation. *Communications of the ACM* 33, 8, 103–111. DOI:http://dx.doi.org/10.1145/79173.79181

Leslie G. Valiant. 2011. A bridging model for multi-core computing. *Journal of Computer and System Sciences* 77, 1, 154–166. DOI:http://dx.doi.org/10.1016/j.jcss.2010.06.012

Alexandru Voica. 2014. PowerVR Series7XT GPUs Push Graphics and Compute Performance to the Max. Retrieved February 24, 2016 from http://blog.imgtec.com/powervr/powervr-series7xt-gpus-push-graphics-and-compute-performance

Alex Voicu. 2010. NVIDIA Fermi GPU and Architecture Analysis. Retrieved February 24, 2016 from http://www.beyond3d.com/content/reviews/55.

Igor Wallossek. 2014. AMD FirePro W9100 Review: Hawaii Puts On Its Suit And Tie. Retrieved February 24, 2016 from http://www.tomshardware.com/reviews/firepro-w9100-performance,3810.html.

Henry Wong, Misel-Myrto Papadopoulou, Maryam Sadooghi-Alvandi, and Andreas Moshovos. 2010. Demystifying GPU microarchitecture through microbenchmarking. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'10)*. IEEE, White Plains, NY, 235–246. DOI:http://dx.doi.org/10.1109/ISPASS.2010.5452013. Mar 28–30.

Received May 2014; revised September 2015; accepted January 2016