NK-GPGPU A GPGPU model for nested kernels

Qianli XING[1,a], Liang HU[2,b], Xilong CHE[3,c]

[1]College of Computer Science and Technology, Jilin University, ChangChun, 130012,China

[2]College of Computer Science and Technology, Jilin University, ChangChun, 130012,China

[3]College of Computer Science and Technology, Jilin University, ChangChun, 130012,China

[a]email:1043354124@qq.com, [b]email:hul@jlu.edu.cn, [c]email:chexilong@jlu.edu.cn

Keywords: GPGPU; Conceptual Model; Micro-architecture

Abstract. More and more scientific problems are now using GPGPU to solve. However , the existing GPGPU did not give a good solution for the problems such as recursive calls and multiple calls problems. Our proposed NKGPGPU model is a descriptive model that solves the key issues of the NK-GPGPU, including hardware architecture, task organization, task execution, and task scheduling. We also prove the validity of our model by optimizing an existing GPGPU performance prediction model.

Introduction

Nvidia's CUDA Toolkit supports a C-like programming language that allows the user to execute code on the corresponding GPU. In the Fermi structure, the CPU needs to start the GPU task, the GPU in the results back to the CPU, CPU and then start a new GPU task, so this application requires a lot of rewriting to run on the Fermi structure of the GPU. Kepler series GPUs support Dynamic Parallelism, which starts new tasks on the GPU side, synchronizes the results, and does not require the CPU to participate in the scheduling process. There are also many studies of dynamic parallelism. For example, [1]shows that dynamic parallelism can improve the performance of some clustering algorithms. [2] gives a solution to the existence of parallel code in the thread of the method, and hope that this method can be used instead of CUDA's dynamic parallelism.

There are, however, two major problems with dynamic parallelism. The first is the father of the kernel communication must pass global memory; the second is in the GPU side to start the Kernel's overhead is very large [2]. The official whitepaper gives more real-world descriptions of what a GPU's computing power can do, rather than how components work together. The programming guidelines give the syntax and the rules of the code, without giving the idea of what kind of microarchitecture has led to these rules. Without a detailed description of the hardware details, researchers and developers can not fully exploit the potential of hardware to improve application performance.

For the GPGPU study,[3], and [4] goals in optimizing the global memory access and achieve as much thread-level parallelism.[5], [6] In terms of structural improvements.

The purpose of this paper is to present a novel hardware component and internal mechanism of operation. The main work of this paper is summarized as follows:

We propose an NKGPGPU model, which includes task organization, hardware structure, scheduling mechanism and execution mechanism, in which task organization gives the data structure between multiple tasks. Finally, by referring to the ideas in [7], a predictive model is given.

NK-GPGPU Model Description

NKGPGPU model includes four main sub-models, namely, hardware structure model, task organization sub-model, task execution sub-model, task scheduling sub-model. The description model is mainly used to explain the major principles in the structural design of NKGPGPU, as well as in operation.

Hardware Structure Model

In the design of the hardware model, we summarize the main components that need to be considered in the chip design and software development, as well as the interaction between the two. NKGPGPU hardware structure of the sub-model shown in Figure 1, each of which we will give the corresponding parts of the explanation.

Processsing Element for Computing (PEc): A component of a task used to perform computations.

Processing Unit for Computing (PUc): A unit for performing computational tasks that is part of a PEc component.

Processsing Element for datamovement (PEc): A component used to perform data transfer tasks.

Processing Unit for datamovement (PUc): A unit for performing data transfers, which is part of a PEd component.

Sechduiling Element (SE): A component responsible for task scheduling.

Task Unit (TU): A component used to receive new tasks.

Buffer: used to store the state of the information components.

In the storage system abstraction process, regardless of the existence of cache at all levels and impact. Each PUc and PUd part in the figure has its own independent local memory. Shared memoy can be accessed and modified by all PEc and PEd components, and global memory can only be modified and accessed by PEd components. Specifically, the PEd component is responsible for data transfer between global memory and shared memory.

Task Organization Submodel

[8]proposed a flat multi-layer code structure, we learn from some of the definitions. And based on the NKGPGPU model to modify the relevant concepts, proposed a three-tier code model, these concepts have been given below:

• ProcessingScript forComputing (PSc): The processing code responsible for calculating the task.

• ProcessingScript for Datamovement (PSd): The processing code responsible for data transfer.

• Scheduling Script for Computing (SSc): The code responsible for calculating the task schedule.

• Scheduling Script for Datamovement (SSd): The code that is responsible for data transfer task scheduling.

• Script List for Computing (SLC): A section of code that contains all the information about parallel computing tasks. An SLC is a column of PSC and SSC.

• Script List for Datamovement (SLD) contains a section of code for all the information about parallel computing tasks. An SLC is a column of PSD and SSD.

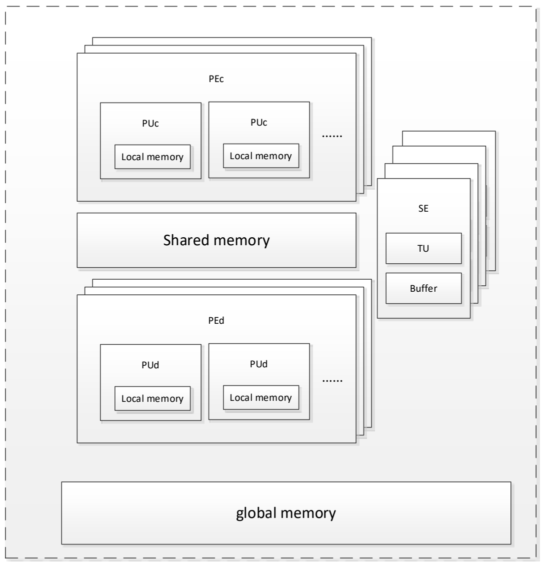
• The Script List for Program (SLP) contains all the code for all the information about a parallel task. An SLC is a column of SLC and SLD.

In the task of the logical structure, the reference CUDA task system, the current design for a three-tier task structure. The description of the minimum granularity task is given below.

• Computing Task Instance (CTI): A running SLC instance with a unique index. Each SLC instantiates a set of CTIs, which is the task width of the SLC. A CTI within a group shares the same SLC, but runs on a memory address based on the CTI index location.

• Datamovement Task Instance (DTI): An SLD running instance with a unique index. Each SLD instantiates a set of DTIs, which is the SLID task width. DTIs within a group share the same SLD, but run on memory addresses based on the DTI index location.

Task organization is actually a code structure and task structure mapping, the corresponding relationship shown in Figure 5. Each SLC will generate a CTI Grid, each CTI Grid contains a number of CTI Block, each DTI Block contains a number of CTI. CTI is divided into DTI warps in CTI blocks Executing such an SLP instantiates several DTI Grid and CTI Grid.



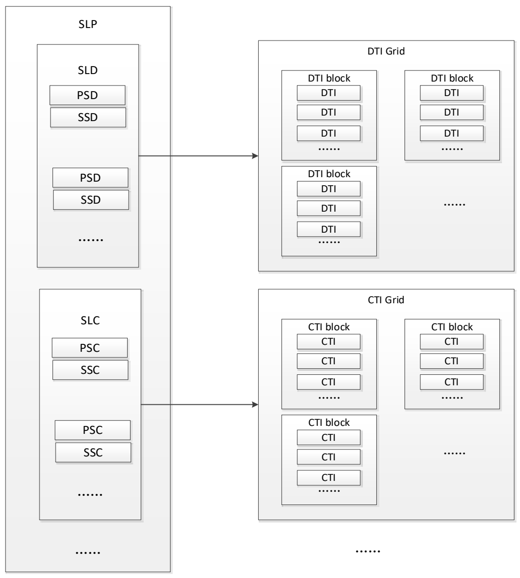


Fig.1. Hardware model Fig.2. Task Organization Chart

Task Execution Submodel

The task execution sub-model is a code-to-hardware mapping, as shown in Figure 3.

PSc Consumption: Logically a set of parallel CTIs is allocated to a set of available PEs for execution. Available indicates that the PEc has sufficient capacity to receive a new set of CTIs. Physically, the associated PSc is parsed by the SEc component, and the relevant information is stored in the TB component.

PSd Consumption: Logically a set of parallel CTIs is allocated to a set of available PEs for execution. Available indicates that the PEc has sufficient capacity to receive a new set of CTIs. Physically, the associated PSc is parsed by the TU in the SE part, and the relevant information is stored in the TB part.

SS Consumption: Both SSd and SSc are executed in the SE part.

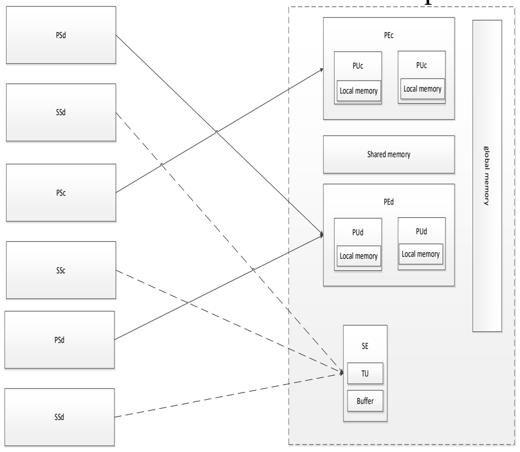


Fig.3. Task execution submodel

Model performance analysis

[7] gives a CUDA GPGPU performance prediction model that can be used to predict the execution time. In this paper,we made changes to adapt to our model, the specific parameters shown in Table 1.

DTI related parameters	
parameter	definition
N_B^D(SLD)	The number of DTI blocks allocated to a PE_D component in an SLD
N_w^D	The number of DTI warp in each DTI Block
N_t^D	The number of DTIs in each DTI warp
D^D	Pipeline depth of PE_D
N_c^D	The number of PU_Ds per PE_D
C_T(SLD)	The max model represents the maximum number of cycles required for a DTI in an SLD
C(SLD)	The number of cycles required for an SLD
CTI related parameters	
parameter	definition
N_w^C	The number of CTI warps in each CTI block
N_t^C	The number of CTIs in each CTI warp
D^C	Pipeline Depth of PE_E
N_c^C	The number of PE_C on each chip
C_T(SLC)	The max model indicates the maximum number of cycles required for a CTI in an SLC
C(SLC)	The number of cycles required for an SLC
SLP related parameters	
parameter	definition
C_SLD(SLP)	The MAX model represents the maximum number of cycles required for an SLD in a SLP
C_SLC(SLP)	The MAX model represents the maximum number of cycles required for an SLC in an SLP
C(SLP)	The number of cycles required for an SLP

Table 1

Equation (1) and (2) give the formula for the number of cycles that an SLD and an SLC need to perform. C_T ^ (SLD) in equation (1) is the representative quantity of a DTI, which is expressed by the maximum value of all DTIs in the formula. Similarly, C_T ^ (SLC) is a representative quantity of CTI in the formula (2), which is expressed by the maximum value of all CTIs.

$$\mathrm{C(SLD)} = N_B^D(SLD)\cdot N_W^D(SLD)\cdot N_T^D(SLD)\cdot C_T\ (SLD)\cdot\frac{1}{N_C^D\cdot D^D} \qquad (1)$$

$$\mathrm{C(SLC)} = N_B^C(SLC)\cdot N_W^C(SLC)\cdot N_T^C(SLC)\cdot C_T\ (SLC)\cdot\frac{1}{N_C^C\cdot D^C} \qquad (2)$$

Equation (3) gives the number of cycles required to calculate SLP. In this case, C (SLP) can be obtained through the MAX model, expressed as the formula (3); If the scheduling did not play a role in the whole (C) , Then C (SLP) can be obtained through SUM model, expressed as formula(4).

$$\mathrm{C(SLP)} = MAX(C(SLD), C(SLC)) \qquad (3)$$

$$\mathrm{C(SLP)} = SUM(C(SLD), C(SLC)) \qquad (4)$$

[7] proposed in the performance prediction model, a thread of the operating cycle is divided into the computing cycle and access cycle. The calculation cycle is denoted by Ncomp and the memory consumption cycle is denoted by Nmemory. In a best-case scenario, the computation task and the memory access task can be hidden from each other. Then, one thread runs for C (T) = MAX (Ncomp, Nmemory). If the scheduling has no effect, = SUM (Ncomp, Nmemory).

The analysis of the execution cycle portion of a thread, including the analysis of the various computational instructions and the cycles required for fetching instructions, is still used in this paper. The cache and synchronization effects are not considered in this paper and are not considered in this paper. For example, an example of a matrix multiplication of size N * N in CUDA is that the size of the block is 16 * 16. A thread needs to complete the computation cycle as 760N / 16 and the required access operation is 240N / 16. In the modified performance model, C_T ^ (SLD) = 240N /

16, C_T ^ (SLC) = 760N / 16, assuming that the size of CTI Block and DTI Blcok is 16 * 16. The rest of the formula is also set up in 7, can be obtained for the Kernel, that is, C (SLP) analysis. An analysis of all SLPs in a dependency tree can also be derived from this result, and is not described here.

Conclusion

In this paper, we give a new GPGPU hardware structure, the logical structure of the task, code structure, and the mapping between them. From the hardware structure, optimize the GPU to start a new task in the process. And is the first article gives the device-side start the task of the data structure of the article. This paper also optimizes an existing performance prediction model to fit the proposed NK-GPGPU.

References

[1] Jeffrey DiMarco, and Michela Taufer, 'Performance Impact of Dynamic Parallelism on Different Clustering Algorithms', in Modeling and Simulation for Defense Systems and Applications Viii, ed. by E. J. Kelmelis (2013).

[2] Yi Yang, and Huiyang Zhou, 'Cuda-Np: Realizing Nested Thread-Level Parallelism

in Gpgpu Applications', (2014), 93-106.

[3] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou, 'A Gpgpu Compiler for Memory Optimization and Parallelism Management', Acm Sigplan Notices, 45 (2010), 86-97.

[4] Liu Yixun, E. Z. Zhang, and X. Shen, 'A Cross-Input Adaptive Framework for Gpu Program Optimizations', in 2009 IEEE International Symposium on Parallel & Distributed Processing (2009), pp. 1-10

[5] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt, 'Improving Gpu Performance Via Large Warps and Two-Level Warp Scheduling', in Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (Porto Alegre, Brazil: ACM, 2011), pp. 308-17.

[6] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe, 'Portable Performance on Heterogeneous Architectures', SIGARCH Comput. Archit. News, 41 (2013), 431-44.

[7] Kishore Kothapalli, Rishabh Mukherjee, M. Suhail Rehman, Suryakant Patidar, P. J. Narayanan, and Kannan Srinathan, 'A Performance Prediction Model for the Cuda Gpgpu Platform', 16th International Conference on High Performance Computing (Hipc), Proceedings (2009), 463-72.

[8] Liang Hu, Xilong Che, and Si-Qing Zheng, 'A Closer Look at Gpgpu', Acm Computing Surveys, 48 (2016).