ICCAD: U: Optimizing GPU Shared Memory Allocation in Automated C-to-CUDA Compilation

Xinfeng Xie Advisor: Jason Cong, Yun Liang
Peking University, 1300012767@pku.edu.cn, ACM#9536371, undergraduate category

I. Problem and Motivation

To ease the burden of GPGPU programming, several existing frameworks generate efficient CUDA code from C code with user-provided directives. In the automated shared memory allocation, which is the key to CUDA kernel performance, previous works mainly explore different data reuse schemes to minimize the number of global memory transactions. However, our study shows that this intuitive model is not accurate because it omits the impact of shared memory allocation on both the parallelism and locality, which further affects kernel performance. Allocating too large shared memory can improve the data locality by saving the global memory transactions and reducing cache contentions while it limits the thread-level parallelism (TLP). On the contrary, allocating too small shared memory can benefit TLP while hurting data locality.

Based on our observations, we develop a performance model to systematically consider the impact of allocating shared memory on the performance. Furthermore, we implement a C-to-CUDA compilation framework that optimizes the shared memory allocation and generates the CUDA code automatically according to the proposed performance model.

II. Background and Related Work

Recent researches on automated CUDA code generation based on user-provided directives [4], such as OpenACC, or polyhedral model [5] have shown promising results. Some of them provide user-managed directives to guide shared memory allocation. The others optimizing shared memory allocation automatically focus on minimizing global memory transactions. Our work differs from them as we optimize the automated shared memory allocation based on our proposed performance model balancing the parallelism and data locality.

Related to our work includes the GPU performance model [3], [1]. Previous works provide analytical models which exhibit high accuracy for predicting the kernel performance. Because our performance model is used in the compilation flow, which requires our model can predict results fast, we sacrifice the accuracy to make the model simple. We focus on the fidelity of our performance model in our automated C-to-CUDA compilation framework, therefore we are satisfied as the predicted results are strongly related to the real performance even the absolute values are different in the order of magnitudes. Furthermore, our performance model includes the impact of different shared memory sizes on the data locality performance including the hit rate of L2 cache, which is missing from previous works.

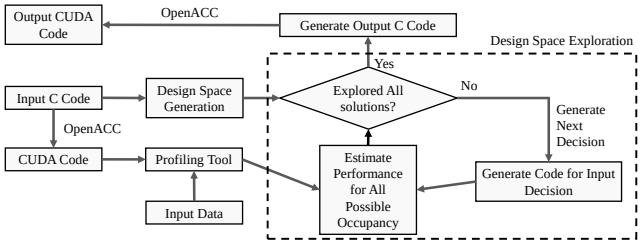

Fig. 1. Automated shared memory allocation framework.

III. Approach

A. Framework Overview

We provide an overview of our automated shared memory allocation framework as Figure 1. The design space of this tool includes different reuse schemes and allocated shared memory size. As the reuse scheme selected, the number of saved global memory transactions and the minimal size of shared memory can be calculated. Our framework will explore the second design variable, the allocated shared memory size, by increments until the maximal hardware constraints. We explore the second design variable discretely by the granularity of one thread block, which is reflected by the thread occupancy. Based on the performance model, the tool picks the shared memory allocation with the best performance, which is then applied to original C code by adding specific OpenACC pragmas. Eventually, it leverages OpenACC compiler to generate the output CUDA code.

B. Performance Model

The performance estimation in our model is based on the consideration of computation and memory components of the kernel, as shown in Equation 1.

$$T = max(T_{comp}, T_{mem}) \tag{1}$$

We estimate the minimal execution time of computation T_{comp} and memory access T_{mem} separately, and take the overall minimal execution time T as the maximum of these two numbers. The T_{comp} and T_{mem} are estimated based on computational instructions and memory (shared memory and global memory) instructions, respectively.

This model is based on the assumption that there is no true dependency among different instructions, and therefore, the computation and memory process can be totally overlapped. Note that our model can not predict the real execution time directly. However, the estimation of these numbers are based

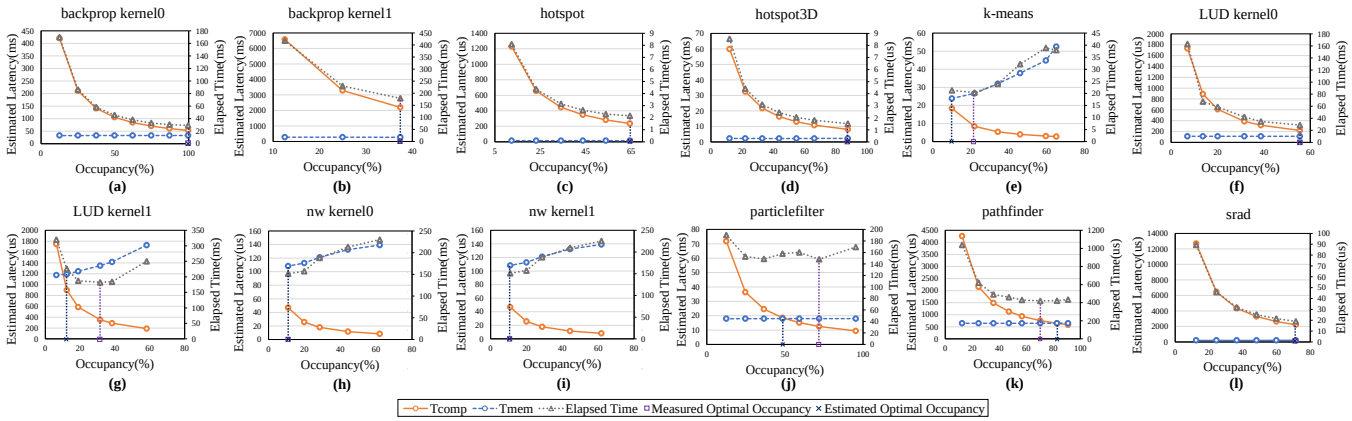

Fig. 2. Validation of the performance model under the same reuse scheme for different occupancy.

on the execution time of different instructions. Therefore, the overall T is strongly positively correlated to the real kernel execution time. In general, the higher the T is, the longer the real execution time of the kernel has.

The T_{comp} is estimated as the time of serially executing all computation instructions divided by the degree of TLP. As nowadays most GPUs group several threads to execute together, normally 32 on NVIDIA's GPU, the degree of TLP can be estimated by the number of active warps which can be further estimated by the number of active thread blocks and the number of threads per block.

The T_{mem} can be estimated as the overall data amount accessed in the global memory and the shared memory divided by their memory bandwidth respectively. The access to global memory can be separated into the access to L2 cache and DRAM according to the ratio of L2 cache hit rate. The design variables affect the T_{mem} from two perspectives. First, different reuse schemes affect the number of global memory transactions and shared memory transactions. Besides, allocated shared memory size affects the number of active warps which further affects the L2 cache hit rate because of cache contentions.

C. Memory Metric Collection

As our performance model needs the memory metrics including the number of global memory transactions, shared memory transactions, and the L2 cache hit rate, we collect these metrics by the statical analysis. The estimation for all these metrics in the compilation time for all CUDA kernels is challenging due to the uncertainty of irregular memory access. We develop the statical analysis methods for regular memory access targeting at the static control of programs (SCoPs) [5], which is widely used in the polyhedral model representations. For the SCoPs, we can obtain the number of global memory and shared memory transactions according to architecture details. We also develop a heuristic algorithm to estimate the L2 cache hit rate under different occupancy for different reuse schemes. As the whole compilation framework is practical and effectiveness, we remain these estimation methods to be further improved in the future work.

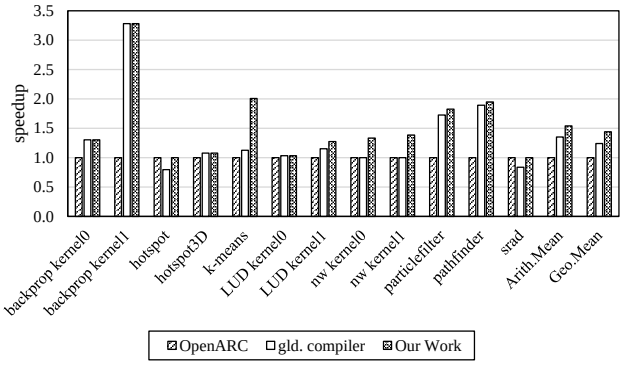

Fig. 3. Overall performance comparison.

IV. Results and Contributions

We conduct the experiments on the NVIDIA Tesla K40. The kernels for evaluations are selected from the Rodinia [2] excluding those violating the assumptions for the SCoPs. First, we demonstrate the fidelity of our performance model. Figure III-B shows the estimated time and measured for kernels from Rodinia under different occupancy which reflects the size of allocated shared memory. Among all 12 kernels, the proposed performance model can select the optimal occupancy for 8 kernels. For the remains, the performance of selected occupancy is very near to the optimal, which incurs up to 24.0% (LUD kernel1) performance difference. The average performance difference between the sub-optimal kernels we predict and the optimal ones is 3.1%. Therefore, we are convinced that our performance model is accurate enough to be used in the compilation framework.

The overall results considering two design variables, the reuse scheme and the size of allocated shared memory, are shown as Figure 3. The gld_compiler in the Figure 3 stands for the optimization considering only minimizing global memory transactions. Overall, our optimization framework leads to up to 3.28X speedup with the average of 1.54X over OpenARC baseline on designs from the Rodinia benchmarks. Compared to the optimization framework that only considers the number of global memory transactions, our work obtains 1.14X additional speedup.

REFERENCES

[1] 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 *ACM Sigplan Notices*, volume 45, pages 105–114. ACM, 2010.

[2] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[3] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 152–163. ACM, 2009.

[4] S. Lee and J. S. Vetter. Openarc: open accelerator research compiler for directive-based, efficient heterogeneous computing. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 115–120. ACM, 2014.

[5] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gomez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):54, 2013.