SM-Centric Transformation: Circumventing Hardware Restrictions for Flexible GPU Scheduling

Bo Wu*, Guoyang Chen*, Dong Li+, Xipeng Shen*, Jeffrey S. Vetter+
*The College of William and Mary, Virginia, USA
+Oak Ridge National Laboratory, Tennessee, USA
*{bwu,gchen01,xshen}@cs.wm.edu, +{lid1,vetter}@ornl.gov

ABSTRACT

To circumvent the limitation from the hardware scheduler on GPU, we create an SM-centric transformation technique. This technique enables complete control of the mapping between tasks and streaming multi-processors (SMs), and enables controlling the number of active thread blocks on each SM. Results show that our approach achieves better speedup than previous ones with kernel co-run cases.

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Processors—*optimization,compilers*

General Terms: Performance, Experimentation.

Keywords: GPGPU, Scheduling, Program Co-Run

1. INTRODUCTION

GPU's high throughput relies on its massive parallelism: A GPU application typically generates tens of thousands of threads with hundreds of them remaining active during any execution point. Scheduling, which determines when and where a task is processed, is critical for tapping into the full power of GPU. However, unlike scheduling on CPU which is controllable at software levels, GPU's scheduler is implemented in a hardware component. This situation makes GPU scheduling out of reach of compilers and runtime; also, the fact that how the hardware-based GPU scheduling mechanism works is undisclosed makes the situation even worse. As a result, some important optimizations based on scheduling are difficult to apply to GPU.

Persistent threads [2] is a technique to circumvent the limitation of hardware scheduler. Its central idea is to generate only a small number of thread blocks which remain active until all tasks have been processed. *Persistent threads* flexibly determines the total number of active thread blocks and the execution order of tasks. However, it has two limitations. The first limitation comes from the randomness in the hardware scheduler. Our experiments showed that the scheduling is not round-robin, contrary to common perception, and does not guarantee an even distribution of thread blocks, even if the number of thread blocks is small. Hence, *persistent threads* may underutilize hardware resource if some SMs obtain less thread blocks than others. Second, *persistent threads* has no support for deciding on which SM a task should run, which is important for some optimizations of locality enhancement.

This work proposes SM-centric transformation, a pure software technique to enable complete control of task scheduling on GPU. It solves the limitation of *persistent threads*. Its two key components, SM-centric task selection and a filling-retreating scheme, work hand in hand to allow flexible control of the number of active thread blocks on each SM and the mapping between tasks and SMs. It opens up new optimization opportunities.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.
ACM 978-1-4503-2809-8/14/08.
http://dx.doi.org/10.1145/2628071.2628130.

2. SM-CENTRIC TRANSFORMATION

In this section, we first describe the idea of SM-centric task selection and its correctness problem induced by the hardware scheduler. Then, we explain how the filling-retreating scheme solves the problem and also enables flexible parallelism control on each SM.

2.1 SM-Centric Task Selection

The GPU threads are grouped into thread blocks (the workers), each processing a job. Usually, the programmer builds the mapping between jobs and workers, as shown in Figure 1 (a). Since the workers are arbitrarily scheduled to SMs, the mapping between jobs and SMs is also arbitrary. *Persistent threads*, as illustrated in Figure 1 (b), maintains a global job queue. The workers grab jobs from the queue once they are idle. As such, *persistent threads* can map jobs to workers. However, the thread block's placement on SM is controlled by hardware, so is the binding between tasks and SMs.

The SM-centric task selection breaks the binding between workers and jobs, but maps jobs to SMs instead. As shown in Figure 1 (c), before the kernel launches, a job queue is created for each SM. When a worker begins execution, it first checks the SM it runs on and processes the next job in the job queue associated with the host SM. Note that different from *persistent threads*, a worker only processes one job. Therefore, for this idea to work, the number of workers scheduled by an SM should be at least the same as the number of jobs in the associated job queue. This property, however, does not hold due to the randomness in the hardware scheduler, and create a program correctness problem.

2.2 Filling-Retreating Scheme

To solve the correctness problem of SM-centric task selection, we design a filling-retreating scheme, shown in Fig-

```
kernel_org()
  jobID = f (workerID);
  processJob (jobID);
```

(a) original kernel

```
createJobQ4workers(); // on CPU

kernel_persist()
  while (jobID = JobQ[workerID].next() != null)
    processJob (jobID);
```

(b) with persistent thread

```
createJobQ4sms(); // on CPU

kernel_smc()
  smID = getSMID();
  jobID = JobQ[smID].next();
  if (jobID != null)
    processJob (jobID);
```

(c) with SM-centric task selection

Figure 1: Conceptual relations among jobs, workers, and SMs.

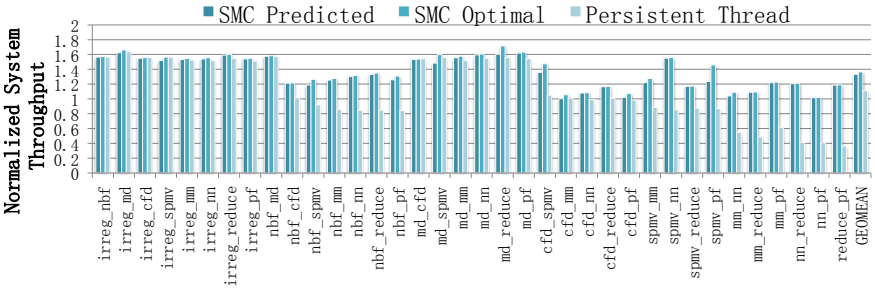

Figure 3: Improvement on system throughput.

```
createJobQ4sms(); // on CPU

kernel_smc()
  smID = getSMID();
  workers = workerCounters[smID]++; // atomic
  if (workers > wantedNumPerSM)
    return;
  while (jobID = JobQ[smID].next() != null)
    processJob (jobID);
```

Figure 2: Pseudo code of a GPU kernel in a *filling-retreating* scheme.

ure 2. The basic idea is to maintain a fixed number (usually small) of workers on each SM, which process all jobs in the corresponding queue. By analyzing and characterizing the GPU kernel at runtime, we determine the maximum number of active thread blocks on each SM, denoted as M. Given N SMs, we generate $M \times N$ workers. We found that the hardware scheduler always distributes all workers to SMs, as long as the hardware resources are enough to accommodate that many. This process is called *filling*. Afterwards, to control the number of active workers per SM (i.e., $wantedNumPerSM$, and $wantedNumPerSM < M$), the extra workers ($M - wantedNumPerSM$) exit immediately. This process is called *retreating*. Based on these two processes, we can control which jobs should be processed by which SM and how many workers per SM.

3. PRELIMINARY RESULTS

Previous studies [3] report that some GPU applications show sub-linear speedups with the increase of SMs. To more efficiently use hardware resource, SM partitioning [1] was evaluated in simulation to support kernel co-runs. But due to the hardware scheduler limitation, the partitioning has never been able to be evaluated on real hardware. SM-centric transformation, through precise control of the mapping between tasks and SMs, enables SM partitioning on real GPUs.

We evaluate SM-centric transformation with the cases of co-runs of two kernels on an NVIDIA Tesla M2075 GPU card. For each co-run case, we must decide the number of SMs allocated to each kernel and the number of active thread blocks on each SM. For this purpose, we design an online sampling and search algorithm to make the configuration decision based on a runtime performance model.

Figure 3 shows the performance results for the co-run of each pair of nine kernels. "SMC Predicted" and "SMC Optimal" represent performance improvement because of SM-centric transformation with the configurations predicted by the online model and with the best configuration through exhaustive offline search. We observe that *persistent threads* produces 11% performance improvement over the original kernels, but it slows down some co-runs (e.g., nn and pf) by as much as 60% because of the lack of enough parallelism. SM-centric transformation yields on average 37% performance gain, and hence is a much better choice.

4. CONCLUSIONS

We present SM-centric transformation, a method to map tasks to SMs and enable parallelism control on GPU. Our exploration on optimizing kernel co-runs show a lot of promises with this new optimization method.

Acknowledgment This material is based upon work supported by NSF Grants (1320796 and CAREER) and DOE Early Career Award.

5. REFERENCES

[1] J. T. Adriaens, K. Compton, N. S. Kim, and M. J. Schulte. The Case for GPGPU Spatial Multitasking. In *International Symposium on High Performance Computer Architecture*, 2012.

[2] K. Gupta, J. A. Stuart, and J. D. Owens. A study of persistent threads style gpu programming for gpgpu workloads. In *Innovative Parallel Computing*, 2012.

[3] S. Hong and H. Kim. An integrated gpu power and performance model. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.