Adding Artificial Barriers in GPU kernels

Quey-Liang Kao, Hao-Ping Kang*

I. Motivation

Writing an efficient GPU program that can fully utilize the computational power of GPUs is a difficult job, owing to the complex relations among programming flows and the architecture parameters. In this research, we would like to answer the question of artificial barrier problem, i.e. when and where to add artificial synchronization can improve the performance of CUDA programs.

II. Artificial Barrier Synchronizations

Most of the performance optimization guidelines suggest that one should eliminate as many synchronizations as possible. Nonetheless, our previous study shows that it may not be true on GPU. We give a performance model to explain the possible reasons for such counter intuitive phenomenon as follows.

- All/Most of the threads access to device memory and the accessed data has some level of locality
- Communication dominates the whole execution time
- An additional barrier regulates warp execution

III. GPU Ocelot

GPU Ocelot is a dynamic compilation framework for GPU computing on multiple targets, including an translator to LLVM[16] for x86 multi-core CPUs, AMD GPUs, NVIDIA GPUs, etc. In this work, we mainly take advantage of GPU Ocelot's PTX-to-PTX transformation infrastructure.

IV. Methods

We design a pass and embed it into the GPU Ocelot's compilation framework, so that we can add artificial barriers automatically following the guidelines mentioned in section II.

During the compilation phase of a CUDA kernel, two attributes are collected, which are

- T, total number of instructions, and
- R, the ratio of memory operations in the T instructions.

Once R and T over some thresholds, we add a barrier at the end of the code segment. In this work, we choose the threshold values by heuristic.

V. Evaluation

To evaluate our performance model and compilation pass, We rewrite the vecadd kernel in CUDA SDK with a tuning parameter Tiling Factor(TF). The pseudo code is as Figure1.

```
Vecadd_Origin ( array A, B, C , int len ){
    for i from 0 to len by TF
      parallel for j from i to i+TF
        C[j] = A[j] + B[j];
}
```

Figure 1. Original Kernel

```
BB2_2:
    ld.global.f32   %f1, [%rd18];
    ld.global.f32   %f2, [%rd17];
    add.f32         %f3, %f2, %f1;
    st.global.f32   [%rd16], %f3;
    cvt.s64.s32     %rd15, %r5;
    add.s64         %rd18, %rd18, %rd15;
    add.s64         %rd17, %rd17, %rd15;
    add.s64         %rd16, %rd16, %rd15;
    add.s32         %r18, %r18, 1;
    setp.lt.s32     %p2, %r18, %r4;
    @%p2 bra        BB2_2;
    bar.sync        0; //insert here
```

Figure 2. Compilation Phase Processing

After the kernel pass through the framework of GPU Ocelot, our pass can read the PTX code and analyze it. As shown in Figure 2, the pass adds an artificial barrier at the final line.

VI. Results

We run the experiment on Tesla m2090. As shown in Figure 3, kernel processed with barrier performs better than original kernel when TF is greater than 1, with improvement up to 4%. That is to say, our pass to adding artificial barrier synchronizations is a novel tool to accelerate a CUDA kernel.

VII. Contributions

- We discover a counter-intuitive phenomenon that GPU kernels may get improvement from non-necessary barriers.
- We give the performance model to describe the phenomenon.
- We design a compilation pass in the framework of GPU Ocelot, which analyzes an input kernel, retrieve its feature, and insert barriers in the right places.
- Using our pass, we obtain further improvement on memory-intensive kernels.
- This research shows the possibility to improve existing kernels without much effort.

VIII. References

1. Kao, Q., Kang, H., & Lee, C. In-Kernel CUDA Profiler and Its Applications in Artificial Barriers Analysis
2. Lo, S., Lee, C., Kao, Q., Chung, I., & Chung, Y. Improving GPU Memory Performance with Artificial Barrier Synchronization.

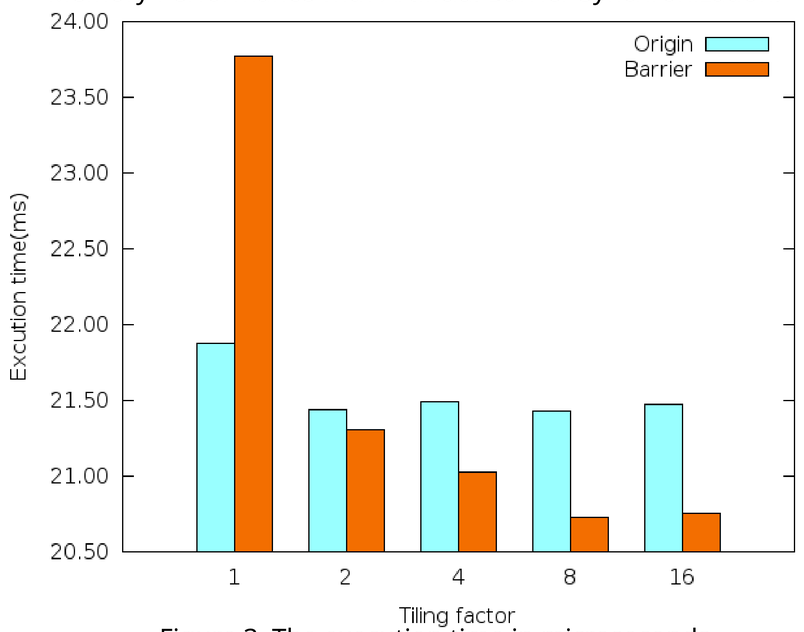

Figure 3. The execution time in microseconds

*National Tsing Hua University, Department of Computer Science SCOPE Lab