Exploiting Dynamic Parallelism to Efficiently Support Irregular Nested Loops on GPUs

Da Li Hancheng Wu Michela Becchi

Department of Electrical and Computer Engineering
University of Missouri, Columbia, MO, USA
{da.li, hancheng.wu}@mail.missouri.edu, becchim@missouri.edu

ABSTRACT

Graphics Processing Units (GPUs) have been used in general purpose computing for several years. The newly introduced *Dynamic Parallelism* feature of Nvidia's Kepler GPUs allows launching kernels from the GPU directly. However, the naïve use of this feature can cause a high number of nested kernel launches, each performing limited work, leading to GPU underutilization and poor performance. We propose workload consolidation mechanisms at different granularities to maximize the work performed by nested kernels and reduce their overhead. Our end goal is to design automatic code transformation techniques for applications with irregular nested loops.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel Programming

General Terms

Algorithms, Performance

Keywords

GPU, Dynamic Parallelism, Irregular Applications

1. MOTIVATION

Irregular nested loops – illustrated in Figure 1(a) – exhibit the following property: the number of iterations of inner loops varies across the iterations of the outer loop. Figure 1(b) shows the pseudo code of a simple parallelization template that maps iterations of the outer loop onto different threads, such that each thread performs a different amount of work (i.e., different threads are assigned differently sized inner loops). It is well-known that, on Nvidia GPUs, threads are grouped into SIMT units called *warps*. Within each warp, the thread with the longest execution path is the execution bottleneck. Therefore, the uneven workload distribution across GPU threads shown in Figure 1(b) will lead to hardware underutilization and hurt the performance.

Dynamic Parallelism provides a mechanism to improve the mapping of irregular nested loops onto GPUs. As shown in Figure 1(c), load balancing of uneven workloads can be achieved by invoking nested kernels. Specifically, a thread will check the amount of work it needs to perform (i.e., the number of iterations of the inner loop assigned to it), and it will then determine whether to process such workload or defer its execution to a child kernel. Hence, large inner loops are executed by nested kernels, which can consist of a dynamic, potentially large number of threads. However, launching nested kernels is not free. This naïve solution might result in a large number of small nested kernel invocations, thus incurring a significant overhead. Our experiments show that, on irregular nested loops with very uneven work distributions, the code of Figure 1(c) can underperform that of Figure 1(b) by a significant factor (even in the order of 100x).

(a) irregular nested loop

```
foreach (int i=1 to N)
  foreach (int j=1 to f[i])
    work(i,j);
```

(b) basic parallelization template

```
thread-mapped-loop(i) {
  for (int j=1 to f[i]) work(i,j);
}
```

(c) dynamic parallelism – naïve

```
thread-mapped-loop(i) {
  if (f[i] <lb_THRES)
    for (int j=1 to f[i]) work(i,j);
  else child_kernel_naive(i);
}
```

(d) dynamic parallelism with consolidation

```
thread-mapped-loop(i) {
  if (f[i] <lb_THRES)
    for (int j=1 to f[i]) work(i,j);
  else buffer.add(i);
}
child_kernel_cons(buffer);
```

Figure 1: Parallelization templates for irregular nested loops

2. METHODOLOGY

The problem of this naïve solution is that it spawns child kernels at the thread level, leading to a large number of potentially small nested kernel invocations. To efficiently use *Dynamic Parallelism*, we propose a mechanism to consolidate workloads across multiple threads and thereby reduce the number of child kernel launches – Figure 1(d). The basic idea is to hold workloads corresponding to large inner loops in a *consolidation buffer* and defer their handling to one or several relatively large child kernels (*child_kernel_cons()*). Based on the software hierarchy of GPUs, we can perform workload consolidation at three granularities: *warp-level* (consolidate workloads among threads within the same warp), *block-level* (consolidate workloads among threads within the same thread-block) and *grid-level* (consolidate workloads in the whole kernel). These mechanisms have pros and cons in the following aspects: (1) synchronization granularity; (2) load balancing granularity; (3) memory access patterns. We plan to study these trade-offs in depth.

3. EXPERIMENTS

To evaluate their performance, we implemented the proposed consolidation mechanisms on two applications: Single Source Shortest Path (SSSP) and Sparse Matrix-Vector Multiplication (SpMV). We run our experiments on an Nvidia K20 GPU using a citation network consisting of 0.4 million vertices and 16 million edges. Our preliminary results show that, on SSSP, the speedups over the basic non-DP implementation are 2.6x, 3.1x and 3.5x, for warp-, block-, and grid-level consolidation, respectively. On SpMV, these speedups vary from 1.4 to 2.1x.

COSMIC'15, Feb 08-08, 2015, San Francisco Bay Area, CA, USA
ACM 978-1-4503-3316-0/15/02.
http://dx.doi.org/10.1145/2723772.2723780