Exploiting Data Locality for Unified CPU/GPU Memory Space using OpenMP

ABSTRACT

To facilitate GPU programming and accelerate applications with large working set, recent GPUs support unified CPU/GPU memory space addressing. In such systems, CPU and GPU can conveniently address each other's memory and data is moved between different memory on demand. However, our investigation shows that the current data migration mechanism is not aware of data locality, resulting in poor performance and redundant data movement. The upcoming OpenMP 5.0 will include new data locality features to address the complex memory hierarchy in today's systems, however, current proposed features do not take unified memory into consideration and cannot address its performance problem. To solve this problem, we propose to extend OpenMP data locality features to improve unified memory performance. The proposed extension exposes different GPU memory management choices for programmers to exploit GPU data locality explicitly. In scenarios with complex data locality, programmers are also allowed to pass hints so that OpenMP compiler and runtime make better GPU data management decisions. Preliminary evaluation shows that our proposal can significantly improve GPU performance and reduce redundant data movement between CPU and GPU for benchmarks with large working set.

KEYWORDS

GPU, unified virtual memory, compiler optimization

ACM Reference format:
. 2017. Exploiting Data Locality for Unified CPU/GPU Memory Space using OpenMP. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 2 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Accelerators are widely used in today's computer systems to improve overall performance and energy efficiency. GPU is currently the most popular accelerator. It has its own memory to

On the other hand, managing unified memory space is challenging

Since version 4.0, OpenMP has included GPU offloading support, which makes it an attractive option for GPU programming.

In this paper, we focus on discrete GPUs.

2 SOLUTION

2.1 Avoid Data Thrashing

Based on the experimental results shown in Fig. **??** and **??**, we concludes that the default unified memory management policy can result in data thrashing when GPU memory oversubscription happens. To avoid data thrashing, first, we propose to pin data with good locality to GPU memory. In doing this, those data cannot be replaced by others with poor locality before being reused. For data with poor locality, there are two choices. One is to pin them to CPU memory, and the other is to allocate them in unified memory space and let the underlying system software manage it automatically. Our experiments show that the second one has much better performance. In the cases of allocating data in unified memory space, on an access, the page contained the accessed data will be brought to GPU memory if it is not present currently. GPU can directly get other data within that page on following accesses. On the other hand, GPU needs to retrieve data from CPU memory every time when data are pinned to CPU memory. As a result, pinning data in CPU memory fails to exploit page level data locality, and we propose to allocate data with poor locality in unified memory space.

2.2 Reduce Redundant Data Movement

Another problem of the default unified memory management scheme is data can be transferred back and forth, as shown in Fig. **??**. Redundant data transfer results in both performance and energy penalty. Exploiting data locality by avoiding data thrashing helps reduce redundant data movement since data can be found in GPU memory more frequently.

Besides, there are other opportunities to optimize data transfers, especially for those from GPU to CPU. By default, there is only one valid data copy in the system. Therefore when page replacement happens in GPU memory, system software needs to write the replaced page back to CPU memory to ensure correctness. The advantage of this one data copy scheme is maintaining data coherence is simpler, and its major disadvantage is data sharing is less efficient. In cases that shared data are only read by both CPU and GPU, it is a better practice to have multiple copies for the same data in their own memory. The benefit is twofold: 1) redundant data transfer is reduced since there is no need to transfer data back and forth between CPU and GPU, and as a result, 2) data sharing is more efficient. A potential problem of doing this is redundant data storage will use more memory space. However, since GPU memory is much smaller compared to its CPU counterpart, duplicating GPU data in CPU memory should not cause a problem in most cases. The difference between the one data copy scheme and the multiple copy scheme is analogous to that between MI and MSI protocols in cache coherence: an extra shared status is allowed.

Thus, we propose to duplicate read sharing data to reduce redundant data movement and enable more efficient data sharing. In CUDA 8.0, an API called cudaMemAdvise() is provided to enable multiple data copies in different memory.

2.3 Implementation

Since cudaMemCpy() implies device-host synchronization, to ensure host code can get the correct data produced by GPU, we insert

Conference'17, July 2017, Washington, DC, USA
2017. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

cudaDeviceSynchronize() before data are used by CPU to do explicit synchronization,

REFERENCES