CUDA-Zero: A Framework for Porting Shared Memory GPU Applications to Multi-GPUs

Dehao Chen
Tsinghua University
chendh05@mails.thu.edu.cn

Graphic Processing Unit (GPU) is increasingly being used for general purpose applications. Its significant computation power can impact the applications especially scientific computations. At the same time, people are pursuing using multiple GPUs to further improve performance, which is motivated by two reasons:

1. It’s always desirable to enlarge the workload or make it finer grained to achieve better precision. Though 1Tflops has already been achieved in one GPU, it’s far away from enough for many scientific computations. Thus scaling it to multiple devices is a good choice.

2. The total memory capacity in one GPU device is too limited. For example, in matrix multiplication, the size of the matrix cannot exceed 16k if the GPU device has 6Gbytes of device memory. It’s necessary to scale to multiple GPUs if larger workload is required.

There are three categories of multiple GPUs: multiple GPUs in one node; multiple nodes with one GPU in each node; multiple nodes with multiple GPUs in each node.

One commonly adopted approach to program multi-GPUs is to rewrite the application using message passing libraries such as MPI, and port the computation intensive parts of the application to GPU. This approach is preferable for the second category of multi-GPUs, i.e. a cluster of multiple nodes with one GPU installed in each node. However, it incurs great amount of effort to forge an efficient program because both steps (writing MPI programs and writing efficient GPU programs) are not trivial. And when it develops to the third category, i.e. a cluster of multiple nodes with multiple GPUs in each node, extra effort is required. NAMD is a good example of this approach, which took several skilled programmers more than 6 months of effort to write the MPI version of the program, another 3 months to port computation intensive kernels to GPU, and another 1 month to port to multiple GPUs.

An alternative approach is to start from an existing GPU implementation, and parallelize it to multi-GPUs. This approach is effective for certain kinds of applications. Unfortunately, the parallelization work is tedious because:

1. Multi-GPUs are controlled using multi-threading, which programmers need to be aware of. Thus it’s required that the control structure of the program be modified manually.

2. Currently, most programming models of single-GPU adopts shared-memory model. For multi-GPUs, each GPU has its own memory space, which can only be accessed by itself. Thus it’s a distributed memory system, and programmers need to partition the workload data manually.

As observed from many prior works, it is already difficult to write programs for even single GPU. Programmers need to design delicate algorithms to distribute workloads to massively parallel processors. Porting single-GPU programs to multi-GPUs incurs another round of workload distribution, which not only adds extra burden to programmers, but also tends to confuse programmers when they manage two levels of workload distribution. Thus it is intuitive to design an approach to “reuse” the effort programmers have paid during the composition of shared-memory single-GPU programs, and use compiler transformations to generate multi-GPU enabled version automatically. This intuition motivated us to build CUDA-Zero, a framework to automate the parallelization for multi-GPUs. Specifically, we choose CUDA as the programming model for our input single-GPU programs, because it is so far the most widely adopted programming model for GPU computing. However, our methodology can be easily adopted by other GPU programming models such as OpenCL and Brooks.

Our framework is named CUDA-Zero because for certain kinds of CUDA applications, zero effort is required for programmers to modify the source code, the compiler will take care of all the parallelization work automatically. Additionally, annotation schemes are designed for programmers to specify data access patterns and gain better control of the parallelization process. However, experiments show that most of the time, user annotation is unnecessary. We also reason about why fully automatic parallelization is possible for certain kinds of GPU applications. Currently, our implementation is applicable for parallelization on the first category of multi-GPUs, i.e. multiple GPUs in on node. We’ll discuss about the possibilities of adopting CUDA programming model to other two categories. It’s also applicable for the application written in MPI+CUDA for the second category multi-GPUs to upgrade to the third category. This paper has made the following contributions:

1. As best of our knowledge, it’s the first work to fully automate the process of porting shared-memory GPU applications to multi-GPUs.

2. It’s the first attempt to utilize profiling to speculatively derive the access patterns of shared-memory SPMD kernels. A highly efficient profiling mechanism is proposed on GPU.