GSYNC: A GPU Synchronization Library

Georgios Anagnopoulos
csd3171@csd.uoc.gr

Introduction

For the ease of the programmers, a library was created called the ANL [1] M4 Macros (Argonne National Laboratory) that helped novice programmers to execute in parallel their programs. One of the recent additions in parallel computing is the GPGPU (General Purpose Graphics Processing Unit) architecture for general computing. In this project, we implement a basic synchronization library and extend the M4 Macros with it. We also port some of the SPLASH-2 Programs to GPU architecture and provide insight regarding our results.

1 Background

The CUDA [2] NVIDIA API offers atomic operations between threads, which were used for the implementation of the locks and barriers. It also offers environmental variables (eg. `threadId.x`) for the indexing of the CUDA threads.

2 Implementation

2.1 Atomic Operations

The first part was to create the atomic operations. Some of the operations included in our library are :

- INCREMENT
- FETCH_AND_DECREMENT
- CMPXCHG

2.2 Locks

CUDA threads are ”packed” into 32-thread groups called warps and execute in SIMD. Due to that functionality, there exists a limitation in our project. As all threads in the same warp are in a lockstep, threads cannot spin or pass in the same warp. So in our implementation, the number of processes declared by the user, is actually the number of **blocks** with 1 thread per block.

2.2.1 SpinLocks

Implemented procedures for the initialization of the spinlocks, and the lock and unlock operations. (ex.)

```
    __host__ gpulock_ *
gspinlock_init(gpulock_t * lock);
```

2.2.2 TicketLocks

Same procedures for the ticketlocks.

2.3 Barriers

GYNC also includes basic barrier functionality, including the initialization and wait procedures.

3 Merging with the M4 Macros

The first step was the CREATE macro expansion. As previously mentioned, our programs must execute with

$$Total = (P * blocks) * (1\ Thread/Block) \qquad (1)$$

So the number of ”processors” must be given as an argument to the CREATE call. So the proper use is CREATE(func, P, args...). The second obstacle was the memory allocation. As the M4 Macros target the abstraction of a program, the user should still use G_MALLOC like before. The ”traditional” way for the GPU kernels to operate on data, is to allocate two memory fragments; one in host(cpu) memory and the other on device (gpu) memory and memcopy them; `FromHostToDevice` before the kernel launch and `FromDeviceToHost` right after. But as this requires special pointer manipulation by the user, in our project we followed a different path (unfortunately with a huge trade-off to performance). We allocated memory with the `cudaMallocManaged()`, which is mapped for both the host and the device and all accessess are over the PCI-e. For the synchronization primitives, the LOCK and BARRIER macros were expanded with the GPU version. In order to add some extra functionality, the WAIT_FOR_END , GET_PID and GCLOCK were also implemented.

Evaluation

We tested our synchronization library on some of the SPLASH-2 Programs. Specifically, Matrix Multiplication, LU and Radix were totally ported; fully functional with the new library. The FFT is partially working, as it cannot perform FFT correctly for now. The Matrix Multiplication and the LU were measured for the evalation of our project. Due to the lockstep limitation and the mapped memory allocation, we can clearly observe that our implementation lacks of performance, compared to the CPU version.(Actual measurements and metrics can be found in the presentation.)

Conclusion

GSYNC is a library that utilizes the atomic operations provided by the CUDA API and offers synchronization between the GPU threads. It it also emerged with the M4 Macros. Regarding the limitations that were previously mentioned, although functional, GSYNC has performance issues.

References

[1] Argonne National Laboratory

[2] NVIDIA Compute Unified Device Architecture