15-618 Project Report

Swadhin Thakkar, Albert Davies

May 9, 2018

1 Title

CudaMon: A Dynamic GPU partition controller to improve Real Time schedulability on Volta Architecture.

2 Summary

The latest GPU architecture, NVIDIA Volta, allows the partitioning of SMs among various tasks using NVIDIA Multi Process Service. MPS allows us to partition and set a maximum limit on the number of SMs that a process can use. We have leveraged this partitioning to improve schedulability when multiple real time processes are running simultaneously. We have designed and implemented a controller to launch and manage all the tasks in the system. The core contention avoidance algorithm was based on TCP AIMD. We have tested our controller with synthetic as well as Deep learning based tasks to show its usefulness in real world workloads.

3 Background

3.1 Real Time Tasks

A real time task needs to complete its execution within a stipulated time. Such tasks are common in safety critical applications like space and autonomous vehicles. A typical self-driving car would have several real time tasks executing in a tight loop for every video frame. Some examples of these tasks are Moving Object Detection, Lane Detection, Workzone detection, Path Planning etc. Each of these tasks will have an associated deadline before which they must complete for their results to be useful. While there are Real Time Operating Systems that can schedule these tasks on the CPU as per the constraints, we do not have an equivalent when the tasks use GPU. This is mainly due to the nature of GPUs which do not allow preemption once a task is launched.

3.2 MPS

Nvidia Multi-Process Service[1] is an alternative implementation of CUDA API. It allows multiple CUDA kernels to be processed concurrently on the same GPU. Figures 1 and 2 taken from [1] illustrate the difference in execution of simultaneous kernels with and without MPS. The key takeaway is that without

MPS the kernels are time multiplexed while with MPS they can potentially run in parallel.

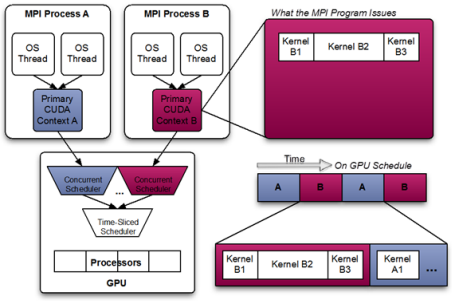

Figure 1: Execution of cuda kernels without mps

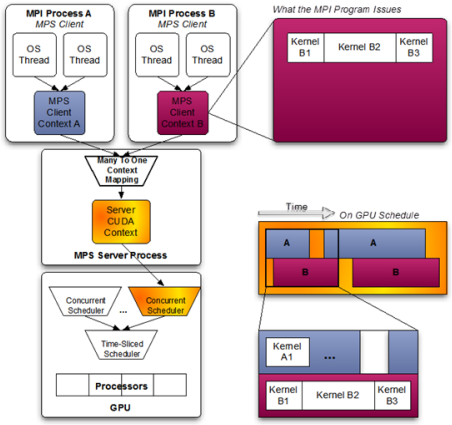

Figure 2: Execution of cuda kernels with mps

MPS follows a client server architecture. The MPS control daemon has to be running for MPS to take effect. Whenever a CUDA process is launched the CUDA driver attempts to connect to the MPS control daemon. If the connection attempt fails, the process runs without MPS. If the connection succeeds then the CUDA driver registers the process with the control daemon. Any subsequent kernel launches are funneled through the MPS control daemon. There are 3 ways to set a partition for a process using MPS:

1. $set\_default\_active\_thread\_percentage < percentage >$: This will only take effect after restarting the MPS server. Applies to all clients afterwards.

2. $set\_active\_thread\_percentage < mps\_server\_pid >< percentage >$: This will take effect for all subsequent clients of that mps server. The server does not need restart, but current clients need to be restarted.

3. CUDA_MPS_ACTIVE_THREAD_PERCENTAGE environment variable: This will take effect for all new clients on a per client basis.

1 and 2 sets the same partition for all subsequent clients. Since we require setting different partitions for different processes, we could only use option 3.

3.3 PRNG kernel

We used a linear congruential generator[2] as our CUDA kernel to conduct the various feasibility experiments as well as to test CudaMon. We chose to implement a pseudo random number generator based kernel for the following reasons:

1. LCG ensures that the calculation is not optimized out by the compiler.
2. LCG provides meaningful work to be performed by each thread for several iterations.
3. Since this kernel only has one read at the beginning and one write at the end of all the iterations, it is compute bound. Therefore we exclude out any artifacts due to memory latency hiding.
4. Once the kernels have finished execution, the same LCG can be simulated sequentially and the results compared to ensure correctness of parallel version.

The code for the CUDA kernel can be found at [3].

3.4 AIMD

For our core contention avoidance mechanism we take inspiration from TCP congestion control. TCP uses Arithmetic Increase Multiplicative Decrease (AIMD) [4] to ensure fairness between different flows. For every packet loss TCP reduces its congestion window size by half and then increases it by 1 for every RTT. We implement a variation of this algorithm by halving the partition of all lower priority tasks when a certain task misses its deadline. Consequently, when a task meets its deadline, we increase the partition of all lower priority tasks by 1%.

One of the reasons AIMD is used in TCP is because it is distributed. Each TCP stream can perform AIMD on its own stream and as shown by the analysis in [5] the whole network converges to a fair distribution. However, in our implementation since the controller has a global view and control of each individual task's partition, we envision a more optimal algorithm to control the partitions of the tasks.

4 Approach

We have developed our implementation to be easily applicable to real programs while working within the constraints enforced by MPS.

4.1 Design Goals

We had the following goals while designing the architecture of CudaMon in decreasing order of preference:

1. It must require minimum changes in the individual processes being launched using CudaMon. The interface with which each individual process informs the controller of missed deadlines must be simple.

2. It must be possible to launch a shell script using CudaMon. This is applicable where a task must need two different applications running together to proceed.

4.2 Constraints

We were restricted in our design space by the constraints imposed by the MPS architecture.

1. The partition for a process is set during initialization of the process. This means after the partition is changed the process needs to be restarted for the new partition to take effect.

2. If a process using MPS is killed midway between kernel execution then the GPU is left is an undefined state. This results in all the subsequent CUDA kernels launched by other shared processes to fail.

4.3 Architecture

The initial architecture that we decided upon was socket based. CudaMon would listen on a predefined socket. The tasks will register themselves with the daemon over the socket. After registration the tasks would relay their response times over this socket for every period of execution. Once CudaMon decides to update the partition of a task, it will send a kill command over the corresponding socket. On the receipt of the kill command the task would gracefully exit itself.

This architecture had the following drawbacks:

1. Since CudaMon needs to restart the task for every update of partition, it needs to know the path to launch the process. This can be relayed to CudaMon by the task during registration. However in case there is a wrapper script to launch the task, CudaMon will not know of this wrapper script and directly launch the task after updation of partition.

2. Each individual task needs significant addition in terms of lines of code to connect, register and relay the response times over the socket. The task also needs to poll the socket to listen for the kill command from the server. This would require the task to use select/poll on the fd or a dedicated thread listening on the socket.

Due to the above mentioned drawbacks we updated our architecture to a Child Process based one. This architecture is shown in Fig: 3. When a task is launched using CudaMon, CudaMon creates a pipe, starts a monitoring thread for this task and forks a child process. The child process redirects its stderr to the input of the pipe. It then sets the environment variable corresponding to the partition and runs the task using execve. The parent process returns back to processing the next command in its command line. The monitoring thread for this process waits on read from the output of the pipe. In this way the task can output its response time to its stderr and the monitoring thread will

receive it. On receipt the monitoring thread updates the required partitions of the lower priority tasks and relaunches each of them using the same steps. During relaunch CudaMon sends SIGINT to the task process and waits until it dies gracefully. After this CudaMon proceeds to relaunch the task eschewing the creation of monitoring thread since it is already running.

The code for CudaMon is available at [6].

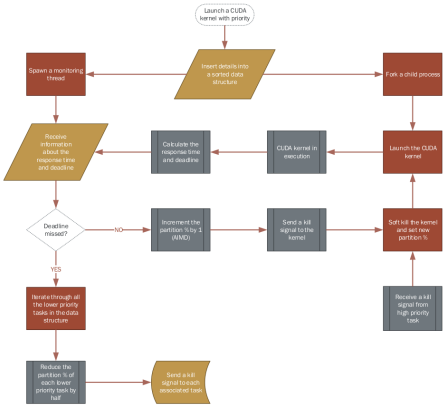

Figure 3: Architecture of CudaMon

4.4 Usage

Prior to running CudaMon, nvidia-mps-control -d is used to launch the MPS daemon.

A task is launched using CudaMon with the following command : *launch priority path*. *launch* is a keyword reserved for CudaMon command line. Priority is used by CudaMon to determine the tasks whose partitions must be updated. Path is the path to the binary or shell script that launches the actual task. The stdout of the task is outputted in the current shell in which CudaMon is active. However, the stderr of the task is consumed by CudaMon to determine the response times.

ls command can be used in CudaMon command line to list the current tasks in the system along with their current partition values.

The task must be modified for the following:

1. The SIGINT handler must be overloaded to die gracefully instead of in the middle of a kernel launch. We suggest this be done by setting a volatile flag and checking its status before processing the next frame.

2. The task must write the string "missed" to its stderr whenever it misses a deadline. Alternatively it must write the string "pass" to its stderr whenever it meets its deadline.

5 Results

5.1 Volta GPU experimentation

During the initial phase of the project several experiments were performed on Tesla V100 GPU to determine its characteristics.

The first experiment performed was to analyse how many threads can run in parallel in a single SM with no interleaving. For this we used our PRNG kernel with seeds to fit a single block. The number of threads per block were varied from 1 to 1024. The graph of the response time vs the number of threads in the single SM is shown in Fig: 4. From this graph we can see that upto 128 threads can execute simultaneously with no interleaving as we see no increase in response time until 128 threads. This was confirmed from the datasheet of Tesla V100, which mentions each SM supporting 4 simultaneous warps with each warp having 32 threads each.

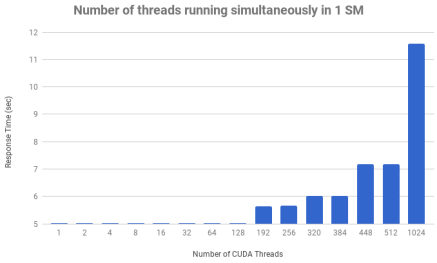

Figure 4: Response time of Kernel vs. Number of Threads in the SM

The next experiment performed was to analyze the response time of the kernel as the number of blocks was increased. For this experiment we kept the threads per block constant and varied the number of blocks from 1-1280. The graph for this is shown in Fig: 5.

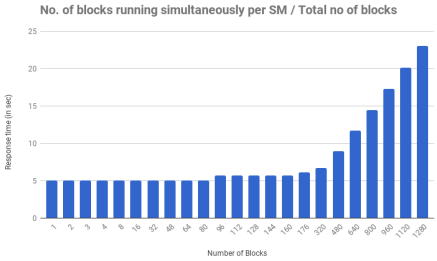

Figure 5: Response time of Kernel vs. Number of Blocks

5.2 Feasibility studies

We then performed few experiments to determine the feasibility of using MPS to partition the GPU.

5.2.1 Manual Partition with PRNG kernel

First the partition was tested using two PRNG kernels. The kernel was configured to run for 200,000 iterations. The total number of seeds was configured at $7168 = 128 * 80 * 0.7$. This ensures that the kernel will finish in its minimum possible time given it gets 70% of SMs. When this kernel is executed in isolation it takes 4.2 ms to complete. A deadline of 4.5 ms was set for it. When both tasks were run with each having a partition of 100%, the response time is measured to be 4.74 ms for both, causing both to miss their deadlines. Next, the lower priority task was manually capped at 20% partition. This yielded a response time of 5.64ms for the lower priority task, but the high priority task was now able to complete in 4.2ms. This experiment gave us confidence that partitioning using MPS can be used to improve real time schedulability. The video of this experiment can be found at [7].

5.2.2 Manual Partition with Deep Learning

To ensure that our PRNG kernel was not a pathological case, we tested the same manual partitioning with a Tensorflow application. The same PRNG kernel was run along with a TensorFlow object detection code to introduce contention. We noticed that our TensorFlow application was using not more than $3-4\%$ of the GPU as per nvidia-smi tool. Therefore the presence of PRNG kernel was not affecting the response time of Tensorflow significantly. However by setting the deadline to be only slightly above its response and measuring the number of deadlines missed, we could observe the number of deadlines missed drop from 56 to 32 with manual partitioning. The video for this experiment can be found at [8].

5.3 CudaMon

In the final experiment for this project, we tested the performance of CudaMon. For this we launched two PRNG kernels using CudaMon and recorded the response times and partition values for both the kernels over time. The graph of response time of both kernels vs. Time is shown in Fig: 6. The graph of the partition values set by CudaMon for both kernels vs. Time is shown in Fig: 7. As we can see from the response times, initially both the kernels always miss their deadline. Subsequently CudaMon halves the partition allotted to the low priority task until the high priority task starts meeting its deadline. Once the high priority task meets its deadline, the partition allotted to the low priority task is again increased slowly until the high priority task again misses a deadline. At this point the partition for low priority task is halved again. This pattern is recurring and is reminiscent of the congestion window size when two TCP streams are contending for a bottleneck link.

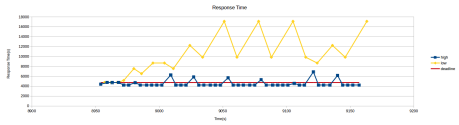

Figure 6: Response time of both kernels vs. Time

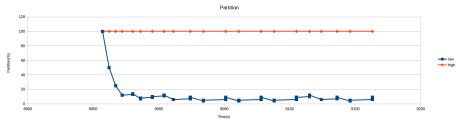

Figure 7: Partition of both Kernels vs. Time

6 Division of Work

Equal work was performed by both project members.

References

[1] "Nvidia mps." [Online]. Available: "https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf"

[2] C. Fontaine, "Linear congruential generator," *Encyclopedia of Cryptography and Security*, pp. 721–721, 2011.

[3] A. Davies, "albd/cudakernel." [Online]. Available: https://github.com/albd/CudaKernel

[4] V. Jacobson, "Congestion avoidance and control," in *ACM SIGCOMM computer communication review*, vol. 18, no. 4. ACM, 1988, pp. 314–329.

[5] D.-M. Chiu and R. Jain, "Analysis of the increase and decrease algorithms for congestion avoidance in computer networks," *Computer Networks and ISDN systems*, vol. 17, no. 1, pp. 1–14, 1989.

[6] A. Davies, "albd/cudamon." [Online]. Available: https://github.com/albd/cudaMon

[7] ——, "Prng experiment video." [Online]. Available: https://asciinema.org/a/T945JcRx3vVh5cKadyi0FNKFp

[8] ——, "Tensorflow experiment video." [Online]. Available: https://asciinema.org/a/Qk0OOaO9dEzWjyt3tFGGT8Kx3