GPU Saturation for Multiple Matrix-Vector Multiplications

Tyler Chang

September 24, 2016

Table of contents

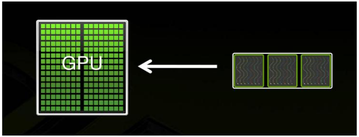
What is a GPU

- A **Graphics Processing Unit** (GPU) is a massively parallel network of processors optimized for graphics processing
- A standard quad-core CPU can process only 4-threads in parallel
- GPUs can utilize parallelization and latency hiding to process threads in much higher orders of magnitude
- GPUs are now used for many general-purpose scientific tasks that could benefit from massive parallelization

GPU Image

Figure: Actual image of a physical GPU.

Figure: Conceptual image of a GPU. Notice **threads** are arranged into **blocks** in a **grid**. Block and grid IDs are manipulated to determine behavior.



MV Multiplication on GPU

One common usage of GPUs is to multiply $m \times n$ matrices by n element vectors to produce m element output vectors.

- Let the elements of the matrix be denoted a_{ij} where i and j represent row and column numbers respectively.
- Let the elements of the input vector be denoted $x_1 - x_n$.
- Then the elements of the output vector $y_1 - y_m$, are given by $y_k = a_{k1}x_1 + a_{k2}x_2 + \ldots + a_{kn}x_n$.
- Thus, for large values of m and n, each of the m output elements can be computed by one GPU thread.
- In a perfect world, this could effectively reduce computation time by $\frac{1}{m}$.

MV Multiplication Image

Below is a conceptual image demonstrating the multiplication of an $m \times n$ matrix by an n element vector:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \\ \dots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \end{pmatrix}$$

Statement of Problem

An $m \times n$ matrix can be multiplied by an n element vector on a GPU to reduce time. But multiplication of i of these matrices are still done sequentially. If $m <$ the maximum threadcount, **can we compute multiple solution vectors in parallel?** Furthermore, **will doing so save any time?**

Specs

Custom built Linux

- Processor: Quad-Core, Intel (i5-4690K @ 3.50GHz)
- OS: Ubuntu 15.10

GeForce GTX 960 Card

- 8 multiprocessors
- 1024 threads per multiprocessor
- 2GB device memory

Coded in C with CUDA 6.5 and MPICH 3.1

Replicating Existing Algorithm

- First step: replicate existing algorithm for matrix-vector multiplication on GPU
- Function MvMul.cu written as standard for modification and testing purposes

```
__global__
void cudMvMul(float *y, float *A, float *x, size_t m, size_t n)
{
        // get block ID
        size_t id = threadIdx.x + blockIdx.x * blockDim.x;
        // each block loops by gridDim
        for(size_t i = 0; i <= m / (gridDim.x * blockDim.x); i++)
        {
                // check for overflow
                if(id < m)
                {
                        // fill idx of soln vector
                        y[id] = 0;
                        for(size_t j = 0; j < n; j++)
                                y[id] += A[id*n + j] * x[j];
                }
                // get next idx
                id += gridDim.x * blockDim.x;
        }
}
                                                              25,1
```

- Benchmarks done to establish speed of standard algorithm.

Benchmarking

The following are benchmark results for tests launched on a $30kx30k$ matrix using N blocks of 512 threads. Times were recorded from kernel launch until all threads completed.

- $N = 1 Time = 752ms$
- $N = 2 Time = 396ms$
- $N = 4 Time = 343ms$
- $N = 8 Time = 448ms$
- $N = 16 Time = 567ms$
- $N = 32 Time = 958ms$

Algorithm for Multiple Matrices

To multiply multiple matrices and vectors, the following steps were taken:

- Create a separate CPU thread in MPICH for each matrix-vector multiplication
- Use MPICH to launch a single GPU kernel for each matrix-vector multiplication
- Use CUDA's Multi Process Service (MPS) to spawn a daemon process to collect kernel launch calls
- Launch all kernels as a single kernel through MPS

Algorithm Conceptual Image

Figure: MPICH and CUDA without MPS

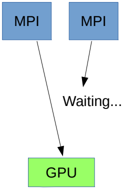
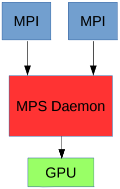


Figure: MPICH and CUDA with MPS



Challenges: Limited RAM

- Originally planned to test on much larger matrices
- Personal device memory limited to 2GB
- Matrices scaled down to 1800×1800 to fit device memory

Challenges: CUDA and MPICH compilers

- CUDA uses nvcc compiler while MPICH uses mpicc
- CUDA code will not build with mpicc and MPICH will not build in nvcc
- Makefile created which compiles code separately then links together

Complete Project

- MPICH succesfully integrated with CUDA and MPS to multiply multiple matrices and vectors in parallel
- Total of 6 Scripts
- 2 program files (1 C file w/ CUDA, 1 C file w/ MPICH)

Profiling

Profiling done on 16 pairs of 1800×1800 matrices and 1800 element vectors. Launched n blocks of 512 threads, and recorded results when run with and without MPS daemon. GPU session times, as measured with NVPROF tool. Note approximate speedup by a factor of **3**:

n	w/o MPS	w/ MPS
2	1538 ms	611.2 ms
4	1473 ms	615.4 ms
8	1424 ms	595.7 ms

Future Works

- Scale up tests on a larger system
- Construct a single kernel launch without MPS daemon to save overhead
- Investigate running multiple versions of other large problems

Resources

Kraus & Messmer. Multi GPU Programming with MPI. NVIDIA Corp. April, 2014

Multi-Process Service, vR352. NVIDIA Corp. May, 2015

NVIDIA CUDA C Programming Guide, V4.2. NVIDIA Corp. April, 2012

Thanks

- Dr. Wang, Prof. Ames and VWC CS dept.
- VWC Math dept.
- Dr. Zubair from ODU
- NASA Langley for research opportunity.