Design and Implementation of Constrained-RPCA Decomposition in CUDA

Fabian Prada-Nino

May 13, 2014

1 Thesis

The thesis of this project is the convenience of massively parallel processors architectures for the design and implementation of the Alternating Direction Method of Multipliers(ADMM) (an iterative descent method) on the solution of a particular optimization problem.

The optimization problem we want to solve is motivated by the constrained-RPCA decomposition of a signal. Given a vector x, a matrix M, and a parameter α we solve:

$$\min_{m,e} ||[M|m]||_* + \alpha||e||_1 \qquad \text{s.t. } x = m + e \tag{1}$$

This provide a decomposition of the input signal x in a pseudo-projection on the space spanned by the columns of M, and a sparse vector e.

2 Development

2.1 Algorithm

```
Constrained RPCA
Initialization();
for i = 1 : total iterations
    UpdateVectors();
    Reduction();
    SVD();
end for
```

Initialization() is a tiny portion of the serial algorithm. We will focus our efforts on optimizing the iterative section.

The task decomposition induced by the *for loop* is order dependent: each iteration strictly requires completion of the iteration before. The parallel optimizations in this project will focus on the design and implementations of the *UpdateVectors()*,*Reduction()*, and *SVD()* methods.

2.2 Decomposition

UpdateVectors()

This is an embarrassingly parallel function. Transformations per-coordinate can be excecuted concurrently. This defines a natural Data/Task decomposition: each thread runs the transformation for a single coordinate.

Reduction()

This function corresponds to the computation of a dot product between set of arrays. To take advantage of the multicore architecture this operation will be decomposed in two reduction steps: a *per-block-reduction* and a *partial-results-reduction.*

Per-block-reduction decompose the arrays in blocks and maps a thread to compute the product between pair of blocks : given a block size B, *thread(i,j)* computes $q_i((j-1)B : jB-1) \cdot m((j-1)B : jB-1)$.

Partial-results-reduction maps a thread to accumulate the partial results: *thread(i)* sums $q_i((j-1)B : jB-1) \cdot m((j-1)B : jB-1)$ for $j = 1 : num-blocks$.

SVD()

This function involve a Singular Value Decomposition of a constant size matrix. Parallel optimization of this function[1] in a massively parallel processor architecture is challenging: (1) For loop iterations are order dependent (2) Parallel portion of the algorithm is small (3) Lots of conditional statements (4) Data size is too small to actually use a significant amount of computing resources. Due to these constraints this task was entirely mapped to a single thread.

2.3 Implementation

Kernels

Ideally the entire solution should be computed from a single kernel call (to avoid kernel overhead). In our problem this was not possible by two reasons:

1. Thread communication is required for the *Reduction()* step, which would force to use a single block in a one-kernel-call implementation. However, using a single block impose constrains on the number of threads and amount of shared memory that can be used. This was inappropriate to our problem.

[1]I took as start point an implementation provided at http://www.cs.colorado.edu/ grudic/teaching/CSCI4202/svd.c.

2. Computing *SVD()* in a single CUDA core was painfully slow. Modifiying the little parallelizable sections to run in parallel in a warp make the things worse: the overhead imposed by the conditional statements and the small data size overshadow parallelism. It turns out that transferring data to CPU and run the function on the host was around 10× faster.

UpdateVectors() and *Reduction()* were implemented as CUDA kernels and *SVD()* as host function. This structure compensates the kernel call overhead by (1) mapping *UpdateVectors()* and *Reduction()* to larger computational resources and (2) alleviating the *SVD()* bottleneck. The memory transfer between host and device is very small (less than 1Kb per iteration) but still a relevant portion of the running time.

Memory
The matrix M in equation 1 is keep fixed along the entire computation. Solving equation 1 for a set of k vectors $x_1, x_2, \dots, x_k$, requires $2k$ reading passes on M per iteration. To compare performance of different CUDA memory types, matrix M was either stored as a (1) linear array in global memory,(2) a 2D cudaArray in texture memory, (3) or a partial copy in shared memory. Using the last strategy we just require one reading pass through M in global memory and $2k$ passes in shared memory. Data that required successive updates (e.g. m,e) or multiple reading from global memory (e.g. x) was either stored as registers or shared memory.

3 Results

I compared the performance of the algorithm on two problem instances and different implementation configurations. For each problem instance M is a common 4096×15 matrix, and the loop runs for 200 iterations. For the ***Single Instance*** x is a 4096×1 vector. For the ***Multiple Instance*** x is a 4096×16 matrix, where each column corresponds to a different instance.

Running Time Per Method (seconds)

	MATLAB	CUDA GLOBAL	CUDA TEXTURE	CUDA SHARED
Single Instance	0.362	0.032	0.031	0.028
Multiple Instance	5.673	0.296	0.245	0.241

Running Time Per Section in CUDA SHARED[2] (seconds)

	UpdateVectors()	*Reduction()*	*SVD()*	Memory Transfer
Single Instance	0.004	0.003	0.016	0.009
Multiple Instance	0.028	0.003	0.203	0.009

[2]Similar results were obtained for CUDA GLOBAL and CUDA TEXTURE. *UpdateVectors()* in CUDA GLOBAL took 0.008 and 0.084 seconds in Single and Multiple Instance respectively due to the larger amount of global memory accesses.

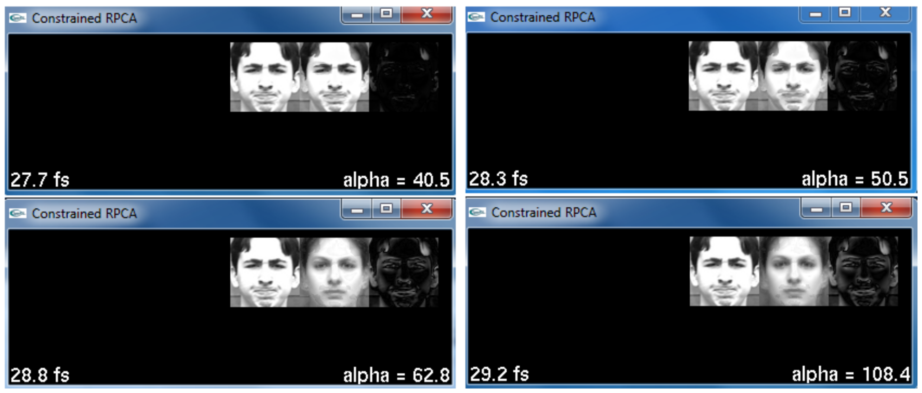

Figure 1: Result above was generated mapping a CUDA memory location to an OpenGL pixel buffer using CUDA graphics interoperability. Solution to single instances of the problem on a 64×64 vector run at 25-30 fps.

4 Conclusions

In this project we discovered some of the gains and challenges of implementing an iterative algorithm in CUDA. The algorithm choosen provide a rich variety of tasks: data independent tasks (i.e., embarrassingly parallel), data sharing tasks (i.e.,thread communication required), and order dependent tasks (inherently serial). This diversity of task allowed identify different features of CUDA programming :

- Kernel call overhead does not seem to be as harmful to performance as initially expected. Instead, host-device memory transaction represented a huge cost on the overall performance. In our results, cost of transactions were independent of data size which suggest that small transactions should be avoided as possible.
- GPU is very efficient on task scheduling and use of computational resources, therefore, granularity must be the central principle on the task decomposition. As can be observed from the results, running our implementation on 16 instances was in average more efficient than running on a single instance.
- Running a serial and highly branched method on a single CUDA core was a bad deal. Certainly, GPU is not designed for this, but I did not expected as poor results. I should clarify this situation. Running this portion of the code on the CPU and even paying for the memory transactions was a better deal.