Appendix B and C to:

Understanding GPU Programming for Statistical Computation: Studies in Massively Parallel Massive Mixtures

Published in the
Journal of Computational and Graphical Statistics

Marc A. Suchard, Quanli Wang, Cliburn Chan, Jacob Frelinger,
Andrew Cron & Mike West

Appendix B: CUDA Storage and Padding for Optimization

This Appendix gives the code snippet for the kernel function that combines the strategies and steps related to use of shared memory and registers as discussed in Section 3.2.6 of the paper.

```
#define DATA_IN_BLOCK 32
#define DENSITIES_IN_BLOCK 16
__global__ void mvNormalPDF(
 REAL* inData, /** Data-vector; padded */
 REAL* inDensityInfo, /** Density info; already padded */
 REAL* outPDF, /** Resultant PDF */
 int p, int n, int k, int DATA_PADDED_DIM, int PACK_DIM) {
 const int thidx = threadIdx.x;
 const int thidy = threadIdx.y;
 const int dataBlockIndex = blockIdx.x*DATA_IN_BLOCK;
 const int datumIndex = dataBlockIndex+thidx;
 const int densityBlockIndex = blockIdx.y*DENSITIES_IN_BLOCK;
 const int densityIndex = densityBlockIndex+thidy;
 const int pdfIndex = datumIndex*kJ+densityIndex;
 extern __shared__ REAL sData[];
 REAL *densityInfo = sData;
 const int data_offset = DENSITIES_IN_BLOCK*PACK_DIM;
 REAL *data = &sData[data_offset];
 if (thidy < p ) {
   data[thidx*p+thidy] = inData[DATA_PADDED_DIM*datumIndex+thidy];
 }
 // Read in density info by chunks
 for(int chunk =0; chunk < PACK_DIM; chunk+= DATA_IN_BLOCK) {
   if (chunk+thidx < PACK_DIM) {
     densityInfo[thidy*PACK_DIM+chunk+thidx] =
     inDensityInfo[PACK_DIM*densityIndex+chunk+thidx];
   }
 }
 __syncthreads();
 REAL* tData = data+thidx*p;
 REAL* tDensityInfo = densityInfo+thidy*PACK_DIM;
 REAL d = mvnpdf(tData,tDensity);
 outPDF[pdfIndex] = d;
}
```

The device function mvnpdf simply evaluates one density. This code allows 512 threads in one thread block, each evaluating one density and writing results to the device global memory in parallel. The initial lines allow threads to identify themselves through thread ids and block ids, and locate the global position of data and densities to be processed. As many threads as are needed then read data into the shared memory "sData"; this is followed by reading all density values into shared memory "densityInfo". Padded dimensions and reading densities by chunking allows coalesced memory transactions. A thread synchronization function call is followed to make sure all data are read into the shared memory before evaluations.

Appendix C: GPU Parallel Reduction for BEM

This Appendix gives the code snippet for the kernel function for element-wise computation of the $\boldsymbol{S}_j$ as discussed in Section 3.2.10 of the paper.

Algorithm 5B GPU Parallel Reduction

```
1
2  #define SIGMA_BLOCK_SIZE 512
3  __global__ void calcSigma(
4   REAL* S, /** Resultant (r,s) entry of S_j */
5   REAL* X,  /** Input data; transposed for coalescing */
6   REAL* M,  /** Weighted sample means in row vectorized form */
7   REAL* Pi,  /** Configuration probability matrix in row vectorized form */
8   double mr, double ms,  /** (m,s) elements of the component sample mean */
9   int n, int p, int r, int s, int k, int j) {
10  const int tid = threadIdx.x;
11  const int bid = blockIdx.x;
12  const int gridSize = SIGMA_BLOCK_SIZE*gridDim.x;
13   __shared__ REAL sum[SIGMA_BLOCK_SIZE];
14  sum[tid] = 0.0;
15  unsigned int i = bid*SIGMA_BLOCK_SIZE + tid;
16   while (i<n) {
17      sum[tid] += Pi[i*iT + j] * (X[r*Ntotal + i] - mr) * (X[s*Ntotal + i] - ms);
18      i += gridSize;
19   }
20   __syncthreads();
21   /** then, sum over partial sums */
22 }
```

This kernel launches 512 threads that read from X in a coalesced fashion to maximize memory throughput. After each thread has read and summed it's portion of the data, the partial sums can be quickly summed from shared memory, and this is very fast.