This document is written in Markdown and generated by Rippledoc. To read the code in below, it is highly recommended that the reader has a basic knowledge about GPGPU and CUDA C. As a courtesy, a math background about the kernel core function is provided in the appendix section.

## 1 Sample Code

### 1.1 Explaination

This piece of code is from my high-performance kernel smoothing library. I’m the type of person who really likes to write high performance codes. For this code, I used several tricks and techniques to speed up its execution.

- Reduce global memory traffic

  Global memory is slow on GPU. In this code, I used all the global memory data (data pointed by \( f_s, g_s, \tilde{H}s \) and \( x \)) only once. For input data (data pointed by \( x \)), I loaded them to the local variables (registers) and reused the local variables as much as possible. For output data (\( f_s, g_s, \) and \( \tilde{H}s \)), I updated their value only when I got the final value for them.

- Loop unrolling

  For a naive implementation, we may need to repeat part of the code based on the dimension of the data. But for this function the dimension of the data is only three, it is a good idea to unroll the code so that the loop control instructions can be avoided. Another benefit from loop unrolling is that we can then reorder the codes, which gives us more control over the program and enables more speed-up techniques.

- Reuse previous results as much as possible

  To avoid redundant calculation, I identified all the repeated expressions from math functions and stored the results of these expressions to local variables. Then
reused these variables as much as possible. For example, expression $Ce^{-rac{1}{2}y^Ty}$ is very “hot” in this code, thus I stored it to variable $f$ and reused it 10 more times.

- Minimize the number of local variables

Local variables or registers are very fast on GPU, but use too many local variables will decrease GPU’s occupancy, and therefore slow down its execution. Since reusing results of “hot” expressions requires more variables to be used, there is a trade-off between these two techniques. One way to balance the local variable number is to retire the values of no longer used expressions and update the values of corresponding local variables. To accomplish this scheme, we also need to reorder some codes so that some expressions can be retired as early as possible. For instance, $tmp4$ initially stored the value of $f \ast (scale0 \ast tmp1 + scale1 \ast tmp2 + scale2 \ast tmp3)$. Once this expression was not needed anymore, $tmp4$ could then be used to store the result of another expression and, thus, there was no need to use another variable. The other way to reduce local variable number is to expand some less “hot” expressions. For example, expression $x0 \ast x0 - 1$ was only used 3 times in this code. Hence, it was not necessary to use one more local variable to store its value.

- Constant memory

Note that the bandwidth (or scale) for each variable (test data) at the same observation (training data) is the same. It means each thread will use the same bandwidth data. And since the bandwidth remains unchanged at each function call, it is a good idea to put bandwidth in constant memory, which can be as fast as reading from a register in this case.

- Constant value

Since the loop was unrolled, the value of coefficients of Gaussian kernel could be easily evaluated before execution and thus made the function more efficient. In this code, constant value 0.063493635934241 was actually the result of expression $\left(\frac{1}{\sqrt{2\pi}}\right)^3$.

1.2 Code

```c
/** @file cudaKernelCore.cu
 * @brief The cuda version of kernel core methods
 * This file contains a series of kernel core methods for kernel density */
```
estimation and kernel density derivative estimation. The user should have
basic knowledge of cuda-c and kernel smoothing theory. The kernel used
throughout is the normal (Gaussian) kernel and all the matrix is in column-
major order.

@author Haofu Liao
*/

#include <math.h>

typedef float dat;

__constant__ dat SCALE[9];

/**
@brief A kernel density derivative estimator core for 3 dimensional data.
*
* This method is used to calculate density derivatives at each training point.
* The results will then be updated to the global density derivatives. This
* method is able to calculate density derivatives up to second order. It uses
* an unconstrained bandwidth and the kernel used is normal (Gaussian) kernel.
*
* @param fs The global density (zero order derivative). It is a n-by-1 matrix.
*            n is the number of observations (test data).
* @param gs The global density gradient (first order derivative). It is a n-by
*            -3 matrix. 3 is the dimension.
* @param Hs The global density curvature (second order derivative). It is a n-
*            by-9 matrix. Each row contains the second order partial derivatives
*            for one observation. The order of each partial derivatives in a row
*            it the same as the corresponding column-major order Hessian matrix
* @param x The observations. It is a n-by-d matrix.
* @param weight The weight of the density derivatives at each training point.
* @param n The number of observations.
*/
__global__ void kernelCore3D(dat *fs, dat *gs, dat *Hs, dat *x, dat weight, size_t n)
{
    int id = blockDim.x * blockIdx.x + threadIdx.x;
    if (id < n)
    {
        dat x0 = x[id];
        dat x1 = x[id + n];
        dat x2 = x[id + n * 2];

        // f = weight * gaussian(x0) * gaussian(x1) * gaussian(x2)
        // gaussian(x) = (1 / sqrt(2 * pi)) * exp(- 0.5 * x)
        // 0.063493635934241 = (1 / sqrt(2 * pi)) ^ 3
        dat f = weight * 0.063493635934241 * expf(-0.5 * (x0 * x0 + x1 * x1 +
                                                    x2 * x2));
        dat H01 = x0 * x1;
        dat H02 = x0 * x2;
        dat H12 = x1 * x2;

        dat scale0 = SCALE[0], scale1 = SCALE[1], scale2 = SCALE[2];
        dat scale3 = SCALE[6], scale4 = SCALE[7], scale5 = SCALE[8];

        dat tmp1 = scale0 * (x0 * x0 - 1) + scale1 * H01 + scale2 * H02;
        dat tmp2 = scale0 * H01 + scale1 * (x1 * x1 - 1) + scale2 * H12;
        dat tmp3 = scale0 * H02 + scale1 * H12 + scale2 * (x2 * x2 - 1);

        fs[id] += f;
        gs[id] += -f * (scale0 * x0 + scale1 * x1 + scale2 * x2);
        Hs[id] += f * (scale0 * tmp1 + scale1 * tmp2 + scale2 * tmp3);

        scale0 = SCALE[3];
        scale1 = SCALE[4];
scale2 = SCALE[5];

dat tmp4 = f * (scale0 * tmp1 + scale1 * tmp2 + scale2 * tmp3);

gs[id + n] += -f * (scale0 * x0 + scale1 * x1 + scale2 * x2);
Hs[id + n] += tmp4;
Hs[id + n * 3] += tmp4;

tmp4 = f * (scale3 * tmp1 + scale4 * tmp2 + scale5 * tmp3);

Hs[id + n + 2] += tmp4;
Hs[id + n * 6] += tmp4;

tmp1 = scale0 * (x0 * x0 - 1) + scale1 * H01 + scale2 * H02;
tmp2 = scale0 * H01 + scale1 * (x1 * x1 - 1) + scale2 * H12;
tmp3 = scale0 * H02 + scale1 * H12 + scale2 * (x2 * x2 - 1);
tmp4 = f * (scale3 * tmp1 + scale4 * tmp2 + scale5 * tmp3);

gs[id + n * 2] += -f * (scale3 * x0 + scale4 * x1 + scale5 * x2);
Hs[id + n * 4] += f * (scale0 * tmp1 + scale1 * tmp2 + scale2 * tmp3);
Hs[id + n * 5] += tmp4;
Hs[id + n * 7] += tmp4;
Hs[id + n * 8] +=

2 Appendix

2.1 Kernel Density Estimation

Let $X_1, ..., X_n$ denote a set of d-variate random samples from a common density $f$. The kernel density estimator of the underlying density is defined as

$$f(x) = \sum_{i=1}^{n} \omega_i K_{S_i}(x - X_i),$$

where $\omega_i$ and $S_i$ is the weight and scale (bandwidth) of each sample point, $K$ is a kernel function which is a symmetrical probability density function, and $K_S$ is the scaled kernel function which is defined by

$$K_S(x) = |S|K(Sx).$$

Typically, $\omega_i = 1/n$, $\forall i$. In the most general case, the scale is an unconstrained matrix and has distinct value at each data point $X_i$.

2.2 Kernel Density Derivative Estimation
The kernel density derivative estimator of \( r \)th derivative of \( f \) is

\[
\nabla^r f(x) = \sum_{i=1}^{n} \omega_i \nabla^r K_{S_i}(x - X_i).
\]

For now, the first and second derivatives are of interest. The kernel gradient estimator is defined to be

\[
g(x) = \sum_{i=1}^{n} \omega_i \nabla K_{S_i}(x - X_i),
\]

where \( \nabla \) is the column vector of the \( d \) partial first-order derivatives and

\[
\nabla K_S(x) = |S|S^T \nabla K(Sx).
\]

Similarly, the kernel curvature estimator

\[
H(x) = \sum_{i=1}^{n} \omega_i \nabla^2 K_{S_i}(x - X_i),
\]

where \( \nabla^2 \) denotes the matrix of all second-order partial derivatives, and

\[
\nabla^2 K_S(x) = |S|S^T \nabla^2 K(Sx)S.
\]

### 2.3 Kernel Core and Kernel Derivative Core

According to the definitions above, the corresponding kernel core and kernel derivative cores can be defined as follow,

\[
f_S(y) = \omega |S|K(y),
\]

\[
g_S(y) = \omega |S|S^T \nabla K(y),
\]

and

\[
H_S(y) = \omega |S|S^T \nabla^2 K(y)S.
\]

If a normal(Gaussian) kernel is used, then the kernel core and kernel derivative cores can be rewritten as

\[
f_S(y) = Ce^{-\frac{1}{2}y^Ty},
\]

\[
g_S(y) = -Ce^{-\frac{1}{2}y^Ty}S^Ty,
\]

and

\[
H_S(y) = Ce^{-\frac{1}{2}y^Ty}S^T(yy^T - I)S,
\]

where
\[ C = \omega |S| \left( \frac{1}{\sqrt{2\pi}} \right)^d. \]