Benchmarking CUDA Gaussian Kernel

Jacob DeRosa

December 10, 2024

Contents

1 Code Analysis

1.1 Implementation of Gaussian Kernel Caller/Setup

```
void gaussian_kde(int n, float h, const std::vector<float>& x, std::
    vector<float>& y) {

    cudaDeviceProp device_prop;
    cudaGetDeviceProperties(&device_prop, 0);

    int threadsPerBlock = int(h);

    h = 0.01;

    int xblocks = (n + threadsPerBlock - 1)/ threadsPerBlock;

    float *d_x, *d_y;

    cudaMalloc(&d_x, sizeof(float)*n);
    cudaMemcpy(d_x, x.data(), n*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc(&d_y, sizeof(float)*n);

    float k = 1/(n*h);

    cudaDeviceSynchronize();

    cudaError_t err = cudaGetLastError();
    if (err != cudaSuccess) {
        std::cerr << "CUDA error: " << cudaGetErrorString(err) << std::
            endl;
    }

    kde_kernel<<<xblocks, threadsPerBlock, threadsPerBlock*sizeof(float
        )>>>(d_x, d_y, n, h, k);

    cudaDeviceSynchronize();

    err = cudaGetLastError();
    if (err != cudaSuccess) {
        std::cerr << "CUDA error: " << cudaGetErrorString(err) << std::
            endl;
    }

    cudaMemcpy(y.data(), d_y, n * sizeof(float), cudaMemcpyDeviceToHost
        );

    cudaFree(d_x);
    cudaFree(d_y);

} // gaussian_kde
```

Here I am allocating memory for the device and managing the cuda kernel. First I divided the input by the *threadsPerBlock* to determine the number of blocks. Next I allocated memory for the input and output arrays d_x and d_y on the device. I calculated the bandwidth constant outside of the kernel. Then I initialized the cuda kernel and copy the results from d_y back to host memory. This configuration assumes that the number of threads being spawned is less

than the amount of active cuda threads the SM's can handle before exhausting their register and shared memory allocation. It introduces a ceiling to the scaling capabilities of a single device.

1.2 Implementation of Density Estimator CUDA kernel

```
__device__ float K(float x)
{
    return (1/(sqrtf(2*M_PI)))*expf(-(x*x)/2);
}

__global__ void kde_kernel(const float *x, float *y, int n, int h,
   float k)
{
    int bx = blockIdx.x;
    int idx = blockDim.x * bx + threadIdx.x;

    if (idx < n)
    {
        float xi = x[idx];

        for (int j = 0; j < n; j++)
        {
            int xj = x[j];
            y[idx] += K((xi - xj) / h);
        }
    }
}
```

In The K (kernel) function I am using *sqrtf* and *expf* instead of their std::* counterparts becuase they are single percision and are optimized for use on cuda devices. In *kde_kernel* I calculate the global index of a cuda thread and then i loop through x and calculate the sum of the kernel densities. Each thread is calculating 1 entry in x. The total runtime is $\frac{n^2}{maxActiveThreads}$.

2 Hardware Specifications

2.1 GPGPU Hardware Specifications

Running deviceQuery from cuda-samples

```
Device 0: "NVIDIA GeForce RTX 2070 SUPER"
  CUDA Driver Version / Runtime Version          12.6 / 12.6
  CUDA Capability Major/Minor version number:    7.5
  Total amount of global memory:                 7767 MBytes
     (8144093184 bytes)
  (040) Multiprocessors, (064) CUDA Cores/MP:    2560 CUDA Cores
  GPU Max Clock rate:                            1815 MHz (1.81 GHz)
  Memory Clock rate:                             7001 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 4194304 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D
     =(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768),
     2048 layers
  Total amount of constant memory:               65536 bytes
```

```
  Total amount of shared memory per block:       49152 bytes
  Total shared memory per multiprocessor:        65536 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  1024
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
     65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 3 copy engine
     (s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Managed Memory:                Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with
        device simultaneously) >
```

2.2 CPU Node Hardware Specifications

3 Benchmarking Methodology

I decided to test on the problem size (n) vs. the Blockwidth on the device. Increasing the block width will allocate more registers and memory for each block which can exceed the device limits and induce resource contention, limiting the number of active blocks on a multiprocessor. I chose my n domain to start from 4 thousand and proceed by powers of 2. Its stops at $512 * 4000$ becuase it would take too long to compute. The Blockwidth domain starts at 2 and proceeds by powers of 2 until 1024. This will allow me to observe the behaviour change in both the size of the input and the blockwidth and make justifiable claims regarding the scaling and performance of the algorithm.

4 Benchmark Results

4.1 Problem size vs Compute time

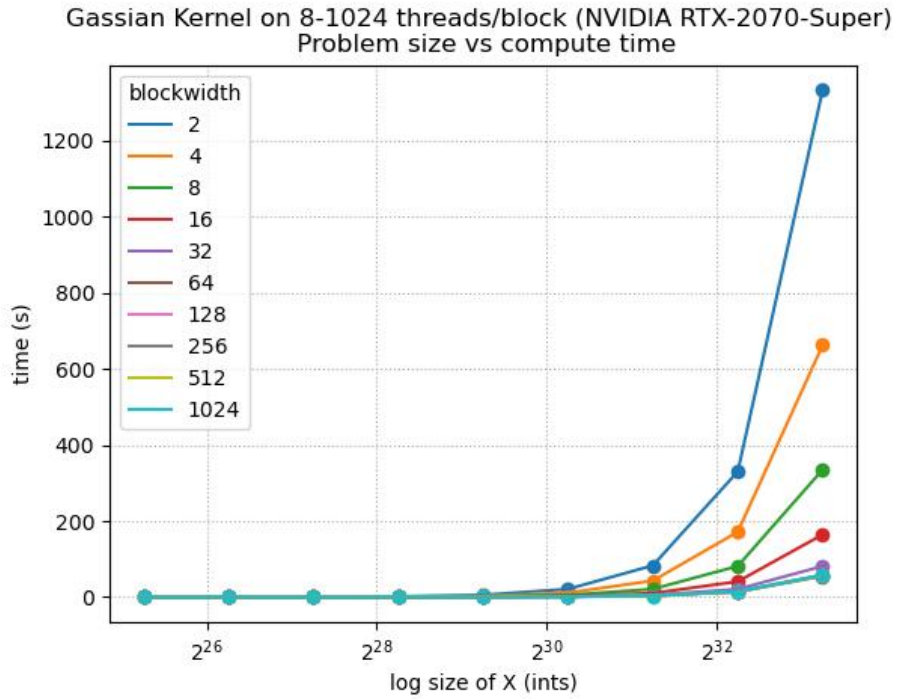

Figure 1: Problem size vs Compute time

n/blockwidth	2	4	8	16	32	64	128	256	512	1024
4.000000e+07	3.052020e-01	1.758780e-01	1.769630e-01	1.835630e-01	1.720690e-01	2.319700e-01	1.813730e-01	1.832930e-01	1.676040e-01	1.723740e-01
8.000000e+07	2.893290e-01	2.277250e-01	1.871440e-01	2.150180e-01	1.760350e-01	2.153480e-01	1.748930e-01	1.941180e-01	1.766300e-01	1.797690e-01
1.600000e+08	4.908900e-01	3.728290e-01	2.393260e-01	2.007340e-01	1.937200e-01	1.843730e-01	1.749530e-01	1.877660e-01	1.909870e-01	1.702180e-01
3.200000e+08	1.596510e+00	8.883190e-01	4.688960e-01	3.210500e-01	2.481700e-01	2.641660e-01	2.488860e-01	2.443090e-01	2.204080e-01	2.745390e-01
6.400000e+08	5.522840e+00	3.009720e+00	1.558310e+00	8.494580e-01	4.763050e-01	3.942630e-01	3.884680e-01	4.049350e-01	4.441060e-01	4.101540e-01
1.280000e+09	2.102470e+01	1.115750e+01	5.251560e+00	2.993060e+00	1.550870e+00	1.145890e+00	1.158820e+00	1.180960e+00	1.052840e+00	1.093080e+00
2.560000e+09	8.299380e+01	4.297450e+01	2.106370e+01	1.025870e+01	5.340040e+00	3.813890e+00	3.751260e+00	3.745390e+00	3.677860e+00	3.662880e+00
5.120000e+09	3.301000e+02	1.714060e+02	8.144240e+01	4.084920e+01	2.081850e+01	1.397690e+01	1.402400e+01	1.472360e+01	1.395540e+01	1.478020e+01
1.024000e+10	1.333280e+03	6.625260e+02	3.350830e+02	1.644160e+02	8.134540e+01	5.527930e+01	5.724430e+01	5.763870e+01	5.707140e+01	5.799370e+01

4.2 Blockwidth vs Speedup

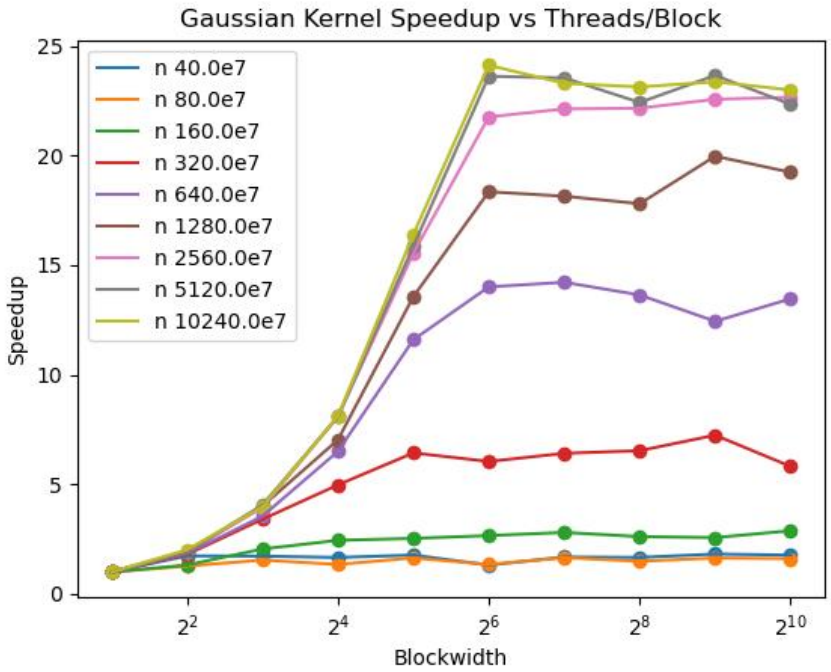

Figure 2: Blockwidth vs Speedup

n/blockwidth	2	4	8	16	32	64	128	256	512	1024
4.000000e+07	1.000000e+00	1.735305e+00	1.724666e+00	1.662655e+00	1.773719e+00	1.315696e+00	1.682731e+00	1.665105e+00	1.820971e+00	1.770580e+00
8.000000e+07	1.000000e+00	1.270519e+00	1.546023e+00	1.345604e+00	1.643588e+00	1.343542e+00	1.654320e+00	1.490480e+00	1.638051e+00	1.609449e+00
1.600000e+08	1.000000e+00	1.316663e+00	2.051135e+00	2.445475e+00	2.534018e+00	2.662483e+00	2.805839e+00	2.614371e+00	2.570280e+00	2.883890e+00
3.200000e+08	1.000000e+00	1.797226e+00	3.404828e+00	4.972777e+00	6.433131e+00	6.043586e+00	6.414624e+00	6.534798e+00	7.243430e+00	5.815239e+00
6.400000e+08	1.000000e+00	1.835001e+00	3.544122e+00	6.501605e+00	1.159518e+01	1.400801e+01	1.421698e+01	1.363883e+01	1.243586e+01	1.346528e+01
1.280000e+09	1.000000e+00	1.884356e+00	4.003515e+00	7.024483e+00	1.355671e+01	1.834792e+01	1.814320e+01	1.780306e+01	1.996951e+01	1.923437e+01
2.560000e+09	1.000000e+00	1.931234e+00	3.940134e+00	8.090089e+00	1.554179e+01	2.176093e+01	2.212425e+01	2.215892e+01	2.256579e+01	2.265807e+01
5.120000e+09	1.000000e+00	1.925837e+00	4.053171e+00	8.080942e+00	1.585609e+01	2.361754e+01	2.353822e+01	2.241979e+01	2.365393e+01	2.233393e+01
1.024000e+10	1.000000e+00	2.012419e+00	3.978954e+00	8.109186e+00	1.639036e+01	2.411897e+01	2.329105e+01	2.313168e+01	2.336161e+01	2.299008e+01

4.3 Blockwidth vs Efficiency

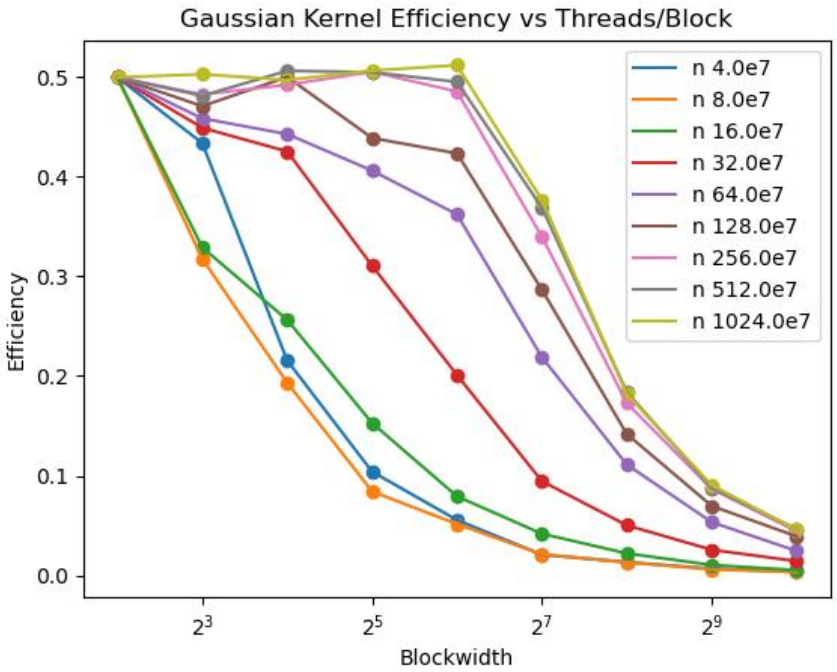

Figure 3: Blockwidth vs Efficiency

n/blockwidth	2	4	8	16	32	64	128	256	512
4.000000e+07	5.000000e-01	4.338263e-01	2.155832e-01	1.039160e-01	5.542871e-02	2.055775e-02	1.314634e-02	6.504314e-03	3.556584e-03
8.000000e+07	5.000000e-01	3.176298e-01	1.932529e-01	8.410023e-02	5.136212e-02	2.099284e-02	1.292438e-02	5.822188e-03	3.199319e-03
1.600000e+08	5.000000e-01	3.291656e-01	2.563919e-01	1.528422e-01	7.918807e-02	4.160130e-02	2.192062e-02	1.021239e-02	5.020077e-03
3.200000e+08	5.000000e-01	4.493065e-01	4.256034e-01	3.107986e-01	2.010353e-01	9.443103e-02	5.011425e-02	2.552656e-02	1.414732e-02
6.400000e+08	5.000000e-01	4.587503e-01	4.430152e-01	4.063503e-01	3.623492e-01	2.188752e-01	1.110701e-01	5.327668e-02	2.428879e-02
1.280000e+09	5.000000e-01	4.710890e-01	5.004394e-01	4.390302e-01	4.236473e-01	2.866863e-01	1.417437e-01	6.954320e-02	3.900295e-02
2.560000e+09	5.000000e-01	4.828084e-01	4.925167e-01	5.056306e-01	4.856811e-01	3.400146e-01	1.728457e-01	8.655828e-02	4.407380e-02
5.120000e+09	5.000000e-01	4.814592e-01	5.066464e-01	5.050589e-01	4.955028e-01	3.690241e-01	1.838923e-01	8.757730e-02	4.619907e-02
1.024000e+10	5.000000e-01	5.031048e-01	4.973693e-01	5.068242e-01	5.121986e-01	3.768590e-01	1.819613e-01	9.035813e-02	4.562815e-02

5 Discussion

In Figure 1, We can observe the quadratic nature of the algorithm as the problem size increases. Initially as the blockwidth increases we are observing super-linear speedup. Its speeding up as fast as the problem size is increasing. This most likely means that the SM's are able to handle the blocksize quadratically increasing in size up until we hit 64 threads per block. It totally levels off in performance increase after this point. My assumption was that the device had allocated the maximum number of concurrently running blocks while the actual grid size was much larger causing the registers to be exhausted.

We can see this abrupt limitation in performance gains in Figure 2. Where we see the speedup is scaling with the increase in the problem size until we hit $>$ 64 blockwidth. We are at least doubleing the speedup as the input size double which is a good sign but we can only achieve a max speedup of about 25 on 64 blockwidth. These speedup results are not great and suffer from inefficient parallelization methods (spawning n threads). Aside from the speedup limitation the quadratic speedup on a constant size input implies that the algorithm posesses strong scaling capabilities.

This can be seen in Figure 3. Where the max observed efficiency is about 50%. This is conidered to not be efficient on the device and could be improved by different parallelization techniques such as limiting the number of running threads and utilizing shared memory on the device. Even though the maximum efficiency is 50% The figure shows that efficiency is holding the same up until 128 blockwidth where it drops off but this is a sign of strong scaling.

6 Conclusion

Overall the algorithm will get the job done but it is by no means efficient or scalable past a certain hardware defined threshold. It possesses some strong scaling behaviors but is ultimately limited by the way it allocates the work to n threads which can exceed the devices limit of concurrent threads and register allocation.

I did try to implement a different version of the algorithm I believe would have been able to deal with the observed limitation but wasn't able to finish it for sumbmission. The idea was to split the work load into a 2D grid where each block has a WxH that add up to the threadsPerBlock. Each thread would calculate multiple items for a given input that would balance the number of threads and keep it from dramatically increasing. The partial sums for each x was calculated along the y axis. Then the would be reduced within the block and stored in global memory. Another kernel would then take the results from all the blocks and perform a global reduction sum into y. I was limited by how much data I could store into shared memory and didn't want to allocate so much global memory on the device.