GPU based Bayesian Optimization

Jaehun Ryu
20202665

1 Introduction

Bayesian optimization is used in many optimization problems and it is always the optimal choice for the global optimization of functions with expensive evaluations. It needs to querying a distribution defined by a relatively cheap surrogate model. An accurate model for this distribution over functions is critical to the effectiveness of the approach and is typically fit using Gaussian processes(GP) that mainly use prior probabilities such as the radical base kernel or Matern kernel. GP scale cubically with the number of observations, it has been challenging to handle objectives whose optimization requires many evaluations. On the other hand, when the input size is relatively small in the GPU, the performance is poor due to overhead due to data copying between host to device. Bayesian optimization is an operation that repeatedly performs fairly similar operations. The operation includes a kind of matrix operations and has the characteristic that the size increases gradually in the step. Therefore, in order to perform parallelization on the GPU to improve performance, it becomes an important issue to select the algorithm and when to perform the operation on the GPU. The project optimizes the kernel by finding a kernel with time to consume through profiling. We also investigate when it is most effective to use the CPU and GPU to cope with the size of the flexible input.

2 Related Work and Background

2.1 Related work

Munawar (1) works proposed a method of finding gene sequence in CUDA with Bayesian optimization. It solves the Gaussian process using a genetic algorithm and describes gene combinations that are not a common case.

Maximilian (2) work proposes a Bayesian optimization framework that can be used on CPU and GPU. This study deals with Bayesian optimization in general cases and processes matrix operations through BLAS operations. The implementation in this project will also implement BLAS, and in this process, I simplified the operations required for the Bayesian optimization process.

2.2 Background

As described in algorithm 1, in Bayesian optimization, matrix operations occur in two parts. First, the process of learning GP consists of calculating RBF kernels using the extracted samples. The RBF kernel can then be calculated by obtaining the matrix Euclidean distance difference between the two inputs and then taking exp operation. This process is divided into two matrix Euclidean distances calculations and one GEMM, and the detailed process is shown below.

$$K_{RBF}(X,Y) = \exp[-\frac{1}{2}||X-Y||^2] = \exp[-\frac{1}{2}||X||^2 - \frac{1}{2}||Y||^2]\exp[-\frac{1}{2}XY^T]$$

Next, operations that predict the following values using GPs require the calculation of the inverse matrix. This is usually done using the Pseudo inversion matrix like the Moore–Penrose method, but in the majority of problems where the input is a thin matrix, so right inverse is acceptable.

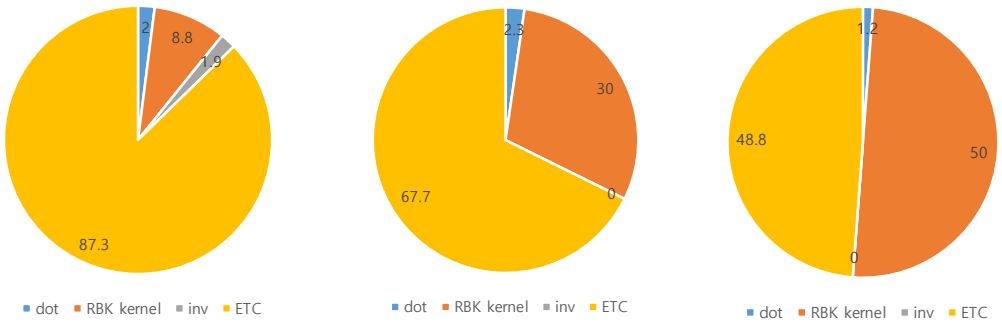
Algorithm 1 Bayesian Optimization

Randomly sample point x0
get y0 through model
Fit gaussian process through x0 and y0
for iteration $n = 1 \dots iteration$ **do**
    Sampling n points (size)
    Predicting the min value of samples using the gaussian process.
    One candidate ($x1$) value is extracted in samples using the UCB algorithm.
    get $y1$ value through a model.
    Fit Gaussian process through $x_1, x_0..$ and $y_1, y_0..$
end for There is a matrix operation in the red line.

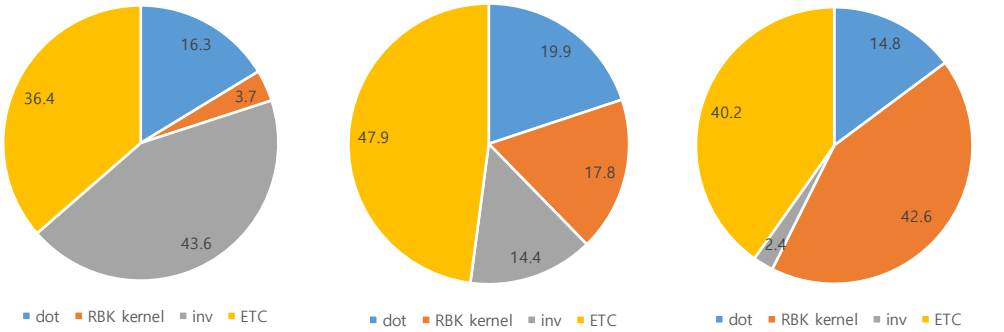
The right inverse matrix can be calculated through transpose operation and GEMM operation matrix as shown below.

$$AA^T(AA^T)^{-1} = A^{-1}$$

3 Implementation



(a) 25 iteration 128 sample (b) 25 iteration 512 sample (c) 25 iteration 2048 sample



(d) 100 iteration 128 sample (e) 100 iteration 512 sample (f) 100 iteration 2048 sample

Figure 1: Results of profiling in serial implementation

3.1 Detailed design

Figure 1 shows the profiling results in the CPP implementation. When the sample and iteration are large enough, it can be seen that more than half of the program parts are parallelizable. To parallelize the matrix operation used in Bayesian optimization, the following strategy was performed.

Algorithm 2 Parallel Gauss Jordan algorithm

```
    Read matrix
 2: Initialize n to size of matrix
    for iteration j = 0 . . . n do
 4:     Find k where matrix[k][j] is not 0
        Spawn n workitems in 1 workgroup
 6:     for workitem i of n in workgroup 1 do
            matrix[j][i]=matrix[j][i]+matrix[k][i] //Step1
 8:     end for
        Spawn n workitems in 1 workgroup
10:     for workitem i of n in workgroup 1 do
            matrix[j][i]=matrix[j][i]/matrix[k][i] //Step2
12:     end for
        Spawn n workitems in 1 workgroup
14:     for workitem i of n in workgroup r do
            matrix[j][r]=matrix[j][r]-matrix[i][j]*matrix[j][r] //Step3
16:     end for
        j++
18: end for
```

3.1.1 GEMM

In the case of GEMM, when implemented naive, a large number of off-chip memory accesses occur. At this time, total M*N*K*2 loads and M*N stores occur, and computational intensity decreases due to many memory operations. I solved this problem by manually creating **caching sub-blocks of the matrices(tile)** and **small accumulation registers**. The overall operation process is described in algorithm 3, and in addition to the tile method, a small amount of memory (3)is additionally used to reduce the load operation at consecutive addresses.

3.1.2 Transpose

The overall operation process is described in algorithm 4. Transpose is a memory intensity operation that changes the shape and order of a matrix. To relieve global memory write, we used shared memory that similar to the method above **caching sub-blocks of the matrices**.

3.1.3 Inv

There are many ways to find the inverse matrix in a square matrix. Many methods decompose a matrix into a triangular matrix and use the trsm operation and GEMM to find the inverse matrix. Gauss Jonathan's method was used to calculate the inverse matrix without implementing the blas operation. Matrix inversion was implemented using *parallel Gauss Jordan algorithm* (4). The overall process is described in algorithm 2, and four kernels were used for implementation. To explain the whole process, first, the A matrix and the I matrix are input from each row, and values other than the diagonal components are normalized. Second, the diagonal components are normalized by receiving the A matrix and the I matrix from each row. Third, perform Gauss Jordan on matrix A and matrix I in each row. Fourth, in each row, the value of the column where Gauss Jordan was executed is assigned as 0.

3.1.4 RBF kernel

The overall operation process is described in algorithm 5. To calculate matrix Euclidean distances, it needs to add the squares of each row of the matrix. I use an in-place reduction using shared memory to reduce global memory synchronization.

Algorithm 3 Tiled GEMM

```
Input : matrix A
Initialize WPT to size of Accumulation register acc as 0
for iteration j = 0 ... numTiles do
    for iteration k = 0 ... WPT do
        load A value and store local tile Asub
        load B value and store local tile Bsub
    end for
    synchronization
    for iteration j = 0 ... numTiles do
        Perform the computation for a single tile and store to acc register.
    end for
    synchronization
end for
for iteration k = 0 ... WPT do
    Store the result of acc in the C matrix.
end for
```

Algorithm 4 Tiled Transpose

```
if xIndex < width and yIndex < height then
    Load input from input matrix and store in shared memory tile
end if
synchronization
Calculate transposed index
if xIndex < height and yIndex < width then
    Load value in the tile and store to output matrix
end if
```

Algorithm 5 RBF kernel

```
matrix Euclidean distances of matrix A
matrix Euclidean distances of matrix B
GEMM (A, B^T)
Apply exponential while adding the above result
```

4 Experiments

4.1 System Configurations

Experiments are conducted on AMD Ryzen 7 3700X CPU , 16GB RAM and RTX 2070 Super The RTX 2070 super has 40 SM and 2560 cuda cores. It also has 64 KB of L1 cache per SM and 4 MB of L2 cache.

4.2 Problems

The experiment was conducted using the Brainn function used in the optimization benchmark (5). Brainn is a problem of finding 3 global minima by receiving x1 x2 input, which can be expressed by the following equation.

$$f(X) = (x_2 - \frac{5.1}{4\pi^2}{x_1}^2) + \frac{5}{\pi} - 6^{\ 2} + 10(1 - \frac{1}{8\pi}cos(x_1) + 10)$$

4.3 Runtime Evaluation

The size was fixed at 2048, and the experiment was conducted by adding parallel operation one by one in the CPU implementation, and also adding CPU operation one by one in the all parallel implementation. In small steps, the CPU implementation is fast because the memory copy overhead

is larger than gain for parallelization, but it is faster to implement only the inverse matrix and the GEMM implementation in parallel as in Figure 3a around 400 steps. As shown in Figure 3b, this shows the performance improvement of 40% and 50% for inverse matrix and GEMM operation, respectively. In the case of transpose and RBF kernels, the gain of parallelization with 700 steps is not appropriate.

Additionally, I analyzed which operation gives the most performance benefit. After applying all the operations in parallel(parallelization baseline), the execution time was measured by implementing one operation into a CPU. Results up to 700 steps are shown in Figure 3c.

Most of them showed better performance from 1.2×to 7.1× than CPU implementation at 700 steps, but both RBF and transpose showed 4.57 × and 3.55 × better than parallelization baseline respectively.

In the above results, it seems that the transpose and RBF kernels do not have sufficient parallelization gain at 700 steps. It is because a small matrix is used as input even if the step increases.

4.4 Qualitative Evaluation

In figure 4, it shows the result of the optimization in the actual brainn function. You can see that from figure 4b with larger iteration and size, it selects more candidates closer to the optimal value than figure 4a.

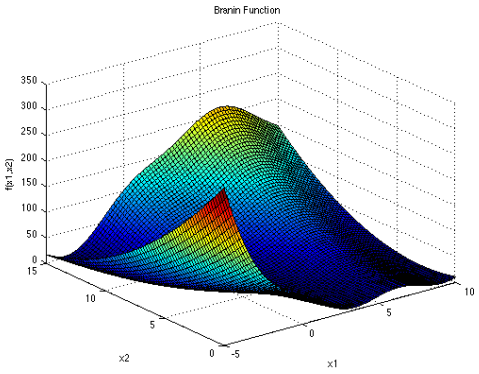


Figure 2: Brainn function

Appendices

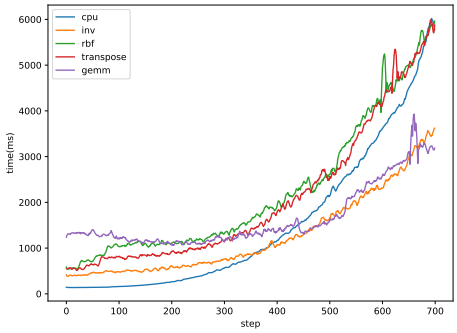
A How to run the executable

```
mkdir build&&cd build
cmake ../
make
./cl size iteration tile_size flag isprint
```

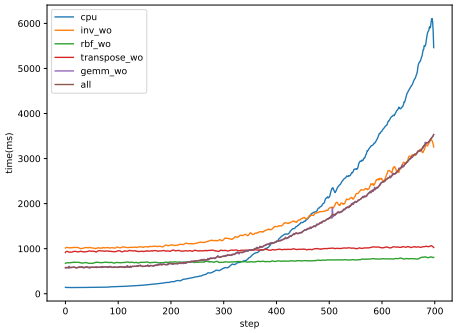
You need to modify the project path in etc.cpp. The flag determines which operation to perform with OpenCL.

FLAG LIST

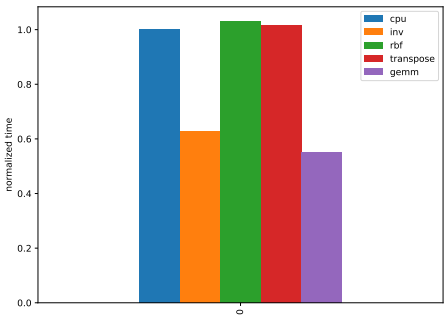
1. all use cpu
2. all use opencl
3. use rbf only
4. use transpose only
5. use inverse only
6. use gemm only



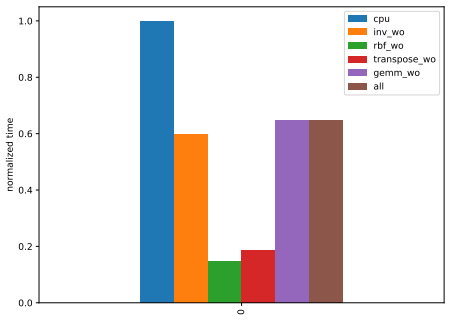
(a) Each step run-time evaluation between serial and one kernel parallel



(b) Each step run-time evaluation between serial and four kernel parallel

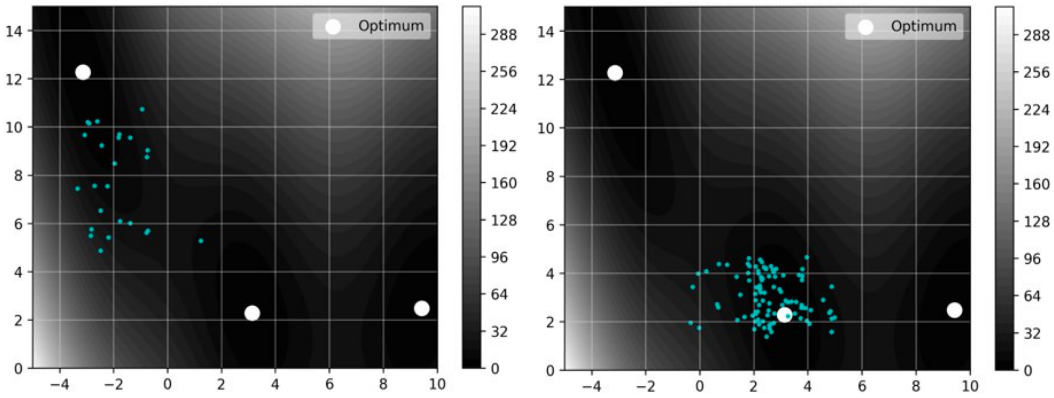


(c) comparison of the execution time of one kernel parallel and serial at 700 steps



(d) Comparison of the execution time of four kernel parallel and serial at 700 steps

Figure 3: Run time evaluation between serial and parallel implementation



(a) 25 iteration 512 size

(b) 100 iteration 2048 size

Figure 4: Brainn function optimization result

7. without rbf
8. without transpose
9. without inverse
10. without gemm

B Code structure

- main.cpp : Run a cycle of Bayesian optimization.
- etc.h,etc.cpp : Implementation of convenient functions for opencl
- util.hpp : Function implementation used in Bayesian optimization, including functions that execute Opencl kernel.
- gemm.cl,inv.cl,rbf.cl,transpose.cl : Opencl kernel implementation

References

[1] A. Munawar, M. Wahib, M. Munetomo, and K. Akama, "Theoretical and empirical analysis of a gpu based parallel bayesian optimization algorithm," in *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2009, pp. 457–462.

[2] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "Botorch: A framework for efficient monte-carlo bayesian optimization," 2020.

[3] "Tutorial: Opencl sgemm tuning for kepler," https://cnugteren.github.io/tutorial/pages/page5.html, accessed: 2020-12-27.

[4] G. Sharma, A. Agarwala, and B. Bhattacharya, "Technical note: A fast parallel gauss jordan algorithm for matrix inversion using cuda," *Comput. Struct.*, vol. 128, p. 31–37, Nov. 2013. [Online]. Available: https://doi.org/10.1016/j.compstruc.2013.06.015

[5] S. Surjanovic and D. Bingham, "Virtual library of simulation experiments: Test functions and datasets," Retrieved December 27, 2020, from http://www.sfu.ca/~ssurjano.