OPTIMIZATION BY PARTICLE SWARM USING SURROGATES VIA BUNCH-KAUFMAN PIVOTING AND STANDARD OPTIMIZATION

Ganlin Zhang, Deheng Zhang, Junpeng Gao, Yu Hong

ETH Zurich, Switzerland

ABSTRACT

Optimization by Particle Swarm Using Surrogates is a framework for expensive non-convex black-box optimization. In this project, we implemented a fast version of OPUS and speed it up to about 6 times compared to the baseline version. Blocking, standard C optimization and vectorization are used to achieve high performance. We also analysis our result and show that we have reached memory bound by roofline plot.

1. INTRODUCTION

Particle swarm optimization (PSO) [1] is a classical black box algorithm for finding optimal solution. As a bio-inspired algorithm, it is simple and has explainable parameters. So for black box optimization problems, particle swarm optimization is unavoidable and can meet great challenges due to computation complexity. One important work in solving PSO efficiently is the optimization by particle swarm using surrogates (OPUS) algorithm as proposed in [2] and our fast implementations is based on this paper.

The first difficulty is an effective baseline implementation as there does not exist an open source implementation. After a faith implementation of the pseudocode described in above mentioned paper which achieves exactly same result, we are faced with the difficulty of blocking, vectorization, and so on.

Contribution. The main contributions are three-fold:

- Firstly, we offer the first open source implementation of OPUS algorithm;
- Secondly, we implement Bunch-Kaufman pivoting for symmetric indefinite matrix in the first time;
- Thirdly, we implement an overall fastest OPUS implementation as we know.

Related work. In [2], Rommel G. Regis proposes a surrogate assisted PSO algorithm, OPUS, which employs surrogate function to select best trail position for each particle and then the overall best position. The OPUS algorithm, therefore, can solve higher dimensional black box optimization problems. To solve the linear system in OPUS, one crucial implementation and optimization is Bunch-Kaufman pivoting [3].

2. BACKGROUND ON THE ALGORITHM/APPLICATION

In this section, we state the particle swarm problem that we are dealing with and the OPUS algorithm which is our implementation baseline. We also provide background information of Bunch-Kaufman pivoting used in our optimization.

Particle Swarm Optimization. For a black box optimization problem which is time consuming in function evaluation step and it becomes the bottleneck of the overall optimization problem, the efficiency is crucial and one standard algorithm is particle swarm optimization.

For simplicity, we use the following notation: We denote by f the optimization objective function and $[a, b] \in \mathbb{R}^d$ the optimization constraint. The particle warm setting is the following: the number of the swarm is denoted by s, each particle has its position in the d-dimensional searching space. And as a live swarm, each particle has velocity. Set $\{1, 2, ..., s\}$ is denoted by $[s]$. For particle i, $i \in [s]$, it's initial position is denoted by $x^{(i)}(0)$ and after t steps, its position is denoted by $x^{(i)}(t)$ and its velocity is denoted by $v^{(i)}(t)$. We denote by $y^{(i)}(t)$ the best position and $\hat{y}(t)$ the overall best in s particles. Here, we assume the swarm evolves in the following way:

$$v^{(i)}(t+1) = \mathrm{i}(t)v^{(i)}(t) + \mu \cdot \omega_1^{(i)}(t) \cdot (y^{(i)}(t) - x^{(i)}(t)) + \nu \cdot \omega_2^{(i)}(t) \cdot (\hat{y}(t) - x^{(i)}(t))$$

$$x^{(i)}(t+1) = x^{(i)}(t) + v^{(i)}(t)$$

where $\mathrm{i}(t)$ is the inertia weight, μ, ν are the acceleration constants, $\omega_1^{(i)}, \omega_2^{(i)} \sim U[0, 1]$ are samples drawn from uniform distribution on $[0, 1]$. Naturally, the particles has a

Optimization by Particle swarm Using Surrogates. To deal with high dimensional optimization problems, OPUS

considers the surrogates assisted variant which elevates the difficulty of slow function evaluation[2]. The procedure can be summarized as follow:

Step 1 to step 4. Initialization. Firstly, choose a space-filling design, sample points from it, and evaluating these points by the objective function. Secondly, sort these function values and choose the initial swarm position, and set particles' velocities randomly. Then initialize the best position for each particle and the overall best position.

Step 5. Use all previously evaluated points to build a surrogate model of the object function.

Step 6. Determine new particle positions. (a) Update trial velocity and position of each particle. (b) Select promising position using the surrogate function from step 5.

Step 7. Evaluate the swarm position of each particle.

Step 8. Update the best position for each particle and the overall best position.

Step 9. Use all previously evaluated points to build a new surrogate model of the object function.

Step 10. Use the optimization solver for local refinement to find a global minimizer of the surrogate.

Step 11. Determine if minima of surrogate from step 10 is far from previous points. If so, evaluate it and update overall best. If result is not accurate enough, go back to step 5.

Cost Analysis. As mentioned above, OPUS is an iterative method to find the optima of an objective function. Since we cannot determine how many steps are needed in general, we will analyze it in one iteration.

The performance bottleneck of each iteration is step 5 and step 9, which are both solving linear systems, the complexity of them are determined by the factorization methods. For example, the complexity of QR decomposition is $O(n^3)$, where n is the matrix size, the complexity of LU is $O(M(n))$ [4] if two matrices of order n can be multiplied in time $M(n)$ ($O(n^{2.376})$ based on the Coppersmith–Winograd algorithm [5]). In our implementation, we use Bunch-Kaufman pivoting strategy to factorize the matrix, whose complexity is the same order as LU but constant times smaller [6], details will be explained in Section 3.

3. PROPOSED METHOD

In this section, we will introduce each of our implemented optimization steps in detail.

Baseline implementation. Since there is no open-source implementation of OPUS online, we first implement it using C++. Specifically, for the candidate particles initialization, we use Latin hypercube design; For surrogate function, we choose RBF cubic function as the original paper recommended; For the linear solver, we choose Bunch-Kaufman pivoting because of its good property for indefinite symmetric matrix.

Bunch-Kaufman pivoting. When factorizing a symmetric matrix, we prefer to choose $\mathrm{LDL^T}$ factorization, which costs only half of the flops compared to LU factorization. To factorize symmetric indefinite system, there are pivoting algorithms such as Bunch-Parlett, Bunch-Kaufman, and Bunch-Kaufman with rook pivoting[7], among which Bunch-Kaufman has the fewest flop costs but worst accuracy. However, Bunch-Kaufman works well for our systems, so we decide to use it as our pivoting algorithm.

Assume that we would like to factorize matrix A, and the permutation matrix corresponding to Bunch-Kaufman pivoting is P, then we have $PAP^T = LDL^T$, where L is a lower triangular matrix and D is a direct sum of 1-by-1 and 2-by-2 block matrix. We show the pseudo-code of Bunch-Kaufman as Algorithm 1, for further details, refer to [3].

Algorithm 1 Bunch-Kaufman pivoting algorithm

$\alpha = (1+\sqrt{17})/8$
$\lambda = |a_{r1}| = \max\{|a_{21}|, \ldots, |a_{m1}|\}$
if $\lambda > 0$ **then**
  if $|a_{11}| \geq \alpha\lambda$ **then**
    use a_{11} as 1-by-1 pivot
  else
    $\sigma = \max\{|a_{1r}|, \ldots, |a_{r-1,r}|, |a_{r+1,r}|, \ldots, |a_{mr}|\}$
    $|a_{pr}| = \sigma$
    if $|a_{11}|\,\sigma \geq \alpha\lambda^2$ **then**
      use a_{11} as 1-by-1 pivot
    else if $|a_{rr}| \geq \alpha\sigma$ **then**
      use a_{rr} as 1-by-1 pivot
    else
      use $\begin{bmatrix} a_{11} & a_{r1} \\ a_{r1} & a_{rr} \end{bmatrix}$ as 2-by-2 pivot
    end if
  end if
end if

Blocking Bunch-Kaufman. To obtain better cache locality, we prefer to do blocking factorization, and we also need to consider the permutations induced by the Bunch-Kaufman algorithm for blocking matrices. [7] gives very rough instructions about the Bunch-Kaufman algorithm for blocking factorization. We find that it is not trivial to derive it so we illustrate it here in detail.

Suppose that we have a symmetric matrix or we get this block matrix after some procedure of our algorithm,

$$A = \begin{pmatrix} A_{11} & & \\ A_{21} & A_{22} & \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$$

.

We first perform Bunch-Kaufman pivoting and LDLT decomposition for submatrix A_{11}, so we have $P_1 A_{11} P_1^T = L_{11} D_{11} L_{11}^T$. The permutation we do in every step should

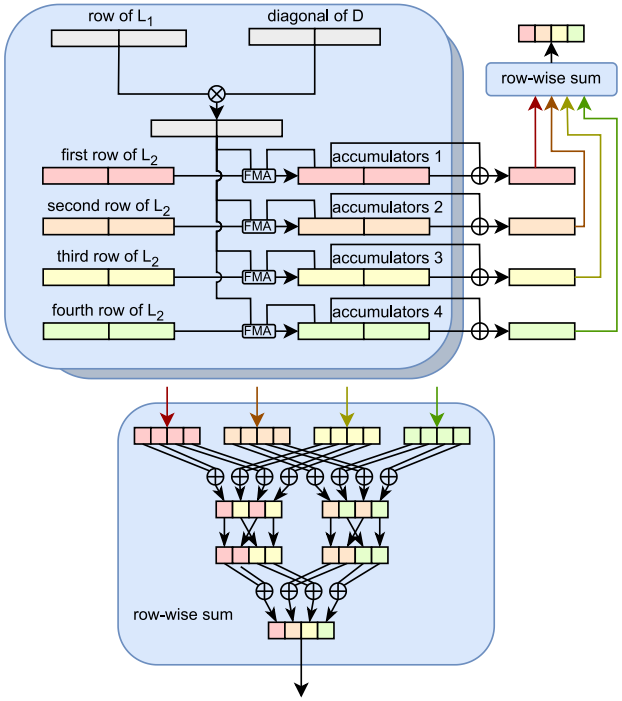


Fig. 1. Unrolling and AVX vectorization strategy we used in the function "matrix_update", each iteration we use 8 elements of L_1 in one row, 8 elements of the diagonal of D, 32 elements of L_2 in four rows (each row 8 elements) to compute, and can update 4 elements of A.

be a linear transformation, that is because, most times, we would like to solve the linear system $Ax = b$ later. Doing linear transformations to A could make b lie in the column space of A at all times so that we could simply permute vector b when solving $Ax = b$. Therefore, we have

$$\begin{pmatrix} P_1 & & \\ & I & \\ & & I \end{pmatrix} \begin{pmatrix} A_{11} & A_{21}^T & A_{31}^T \\ A_{21} & A_{22} & A_{32}^T \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} P_1^T & & \\ & I & \\ & & I \end{pmatrix}$$
$$= \begin{pmatrix} P_1 A_{11} P_1^T & P_1 A_{21}^T & P_1 A_{31}^T \\ A_{21} P_1^T & A_{22} & A_{32}^T \\ A_{31} P_1^T & A_{32} & A_{33} \end{pmatrix}.$$

After applying the decomposition to the first block, we obtain the permuted matrix

$$\tilde{A}_1 = \begin{pmatrix} L_{11} D_{11} L_{11}^T & P_1 A_{21}^T & P_1 A_{31}^T \\ A_{21} P_1^T & A_{22} & A_{32}^T \\ A_{31} P_1^T & A_{32} & A_{33} \end{pmatrix}.$$

Then we compute the Schur complement for the submatrix

$$A_2 = \begin{pmatrix} A_{22} & A_{32}^T \\ A_{32} & A_{33} \end{pmatrix}.$$

First solve the column submatrix L_{21} and L_{31} under L_{11} as

$$\begin{pmatrix} L_{11} D_{11} L_{11}^T & & \\ L_{21} D_{11} L_{11}^T & A_{22} & \\ L_{31} D_{11} L_{11}^T & A_{32} & A_{33} \end{pmatrix},$$

and further get

$$\tilde{P}_1 A \tilde{P}_1^T = L_1 D_1 L_1^T + \begin{pmatrix} 0 & \\ & \widehat{A}_2 \end{pmatrix}$$
$$= \begin{pmatrix} L_{11} & & \\ L_{21} & I & \\ L_{31} & & I \end{pmatrix} \begin{pmatrix} D_{11} & & \\ & I & \\ & & I \end{pmatrix} \begin{pmatrix} L_{11} & L_{21}^T & L_{31}^T \\ & I & \\ & & I \end{pmatrix}$$
$$+ \begin{pmatrix} 0 & \\ & \widehat{A}_2 \end{pmatrix}.$$

So we have,

$$A_2 = \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} D_{11} \begin{pmatrix} L_{21}^T & L_{31}^T \end{pmatrix} + \begin{pmatrix} \widehat{A}_{22} & \\ \widehat{A}_{32} & \widehat{A}_{33} \end{pmatrix}.$$

Then we repeat this process.

However, when doing permutation next time with matrix

$$\tilde{P}_2 = \begin{pmatrix} I & & \\ & P_2 & \\ & & I \end{pmatrix},$$

we need to consider how to deal with it and previous permutations together carefully.

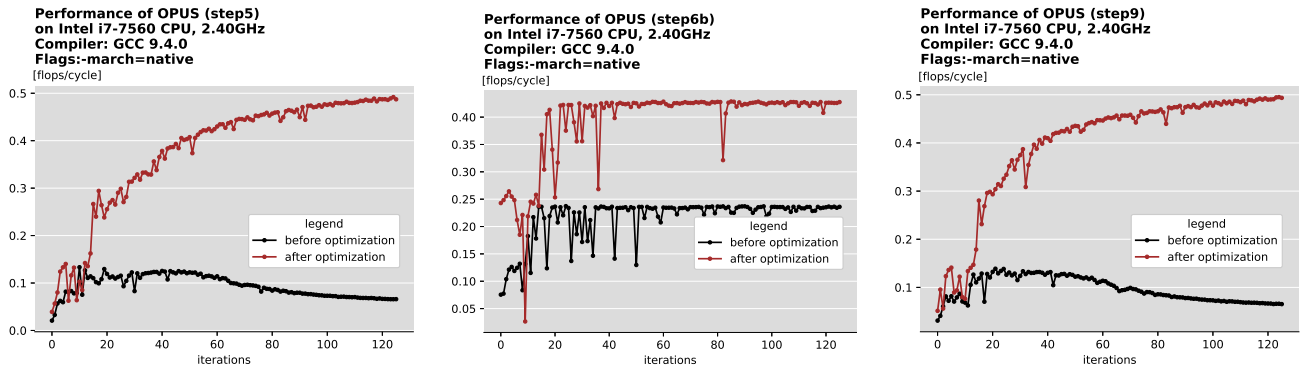
$$\tilde{P} A \tilde{P}^T = \tilde{P}_1 \tilde{P}_2 A \tilde{P}_2 \tilde{P}_1$$
$$= \begin{pmatrix} P_1 & & \\ & P_2 & \\ & & I \end{pmatrix} A \begin{pmatrix} P_1^T & & \\ & P_2^T & \\ & & I \end{pmatrix}$$
$$= \tilde{P}_2 L_1 D_1 L_1^T \tilde{P}_2^T + \tilde{P}_2 \begin{pmatrix} 0 & \\ & \widehat{A}_2 \end{pmatrix} \tilde{P}_2^T$$
$$= \begin{pmatrix} L_{11} & & \\ P_2 L_{21} & P_2 & \\ L_{31} & & I \end{pmatrix} D_1 \begin{pmatrix} L_{11} & P_2^T L_{21}^T & L_{31}^T \\ & P_2^T & \\ & & I \end{pmatrix}$$
$$+ \begin{pmatrix} 0 & & \\ & P_2 A_{22} P_2^T & A_{32}^T P_2^T \\ & P_2 A_{32} & A_{33} \end{pmatrix}.$$

Now we could observe that the trailing matrix,

$$\tilde{A}_2 = \begin{pmatrix} P_2 A_{22} P_2^T & A_{32}^T P_2^T \\ P_2 A_{32} & A_{33} \end{pmatrix}$$

shows the same form as the initial permuted 3×3 block matrix $\tilde{A}_1$, so we could repeat LDLT decomposition and compute schur complement to obtain the next block.

Automatic Tool. The block size of the blocking Bunch-Kaufman algorithm depends on the cache size, which is different for different machines. To find a suitable block size,



(a) (b) (c)

Fig. 2. Comparison of original implementation (black) and optimization(red) on bottleneck steps. (a), (b), (c) shows the performance improvement in step 5 (fit surrogate), step 6b (evaluation) and step 9 (refit surrogate), respectively.

we write an automatic tool to decide the size for different machines according to the speedup with a large matrix.

Spare operation. Besides, we use sparse matrix operations instead of dense operations to improve the performance. For the block diagonal matrix, we find the 2-by-2 block is very rare in practice. Therefore, sub-diagonal entries of the matrix are sparse. In our original implementation, we iterate the diagonal and two sub-diagonals using three for-loops, which is time-consuming. To reduce the flops, we record the indices of the 2-by-2 blocks and access the sub-diagonal with strides. Also, for the permutation, triangular, and diagonal matrices, we implement the solver of $Ax = b$ separately to take advantage of the special properties.

Dependency removal. The most important optimization in standard C optimization is dependency removal. Since our two bottlenecks (linear solver and evaluate surrogate) have accumulators. Both the two accumulators use one FMA for the accumulator, and one FMA has a dependency on one subtraction, and the other FMA has a dependency on one multiplication. The first strategy we tried is loop reordering. We replace the order of the innermost and the second innermost loop and use a different accumulator for each iteration of the second innermost loop. Although the dependency is removed, one of the arrays is accessed with strides, so the cache locality is not optimized. We use unrolling to remove the dependency in our final implementation. As shown in Table 1, FMA, multiplication, and subtraction have a latency of 4 throughputs 2 on the same ports. Therefore, we can execute 8 subtractions/multiplications first followed by 8 FMA to maximize the port utility. Therefore, the unrolling factors of the inner loop of linear solver and evaluate surrogate are 8. For the square root computation of evaluating the surrogate function, the latency of square root function is 15-16, which is much more than the latency of other operations. Therefore, we tried to move the square root function out of the loop, which improves the performance.

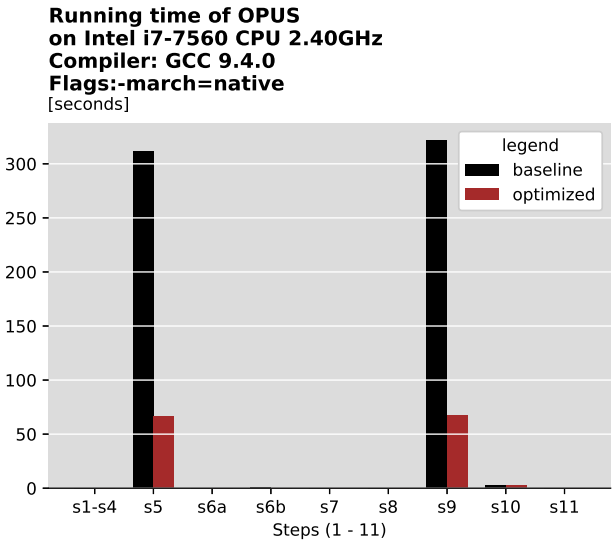


Fig. 3. Overall performance over OPUS pipeline. The baseline running time (black) is evaluated for each step, from step 1 to step 11. After optimization, the running time (red) is around 6 times faster than the original implementation in the bottleneck steps: step 5 and step 9.

AVX2 vectorization. We apply AVX2 vectorization to the matrix update function and the evaluate surrogate function. For the evaluate surrogate function, we have three nested for loops, the first for loop iterate the points to be evaluated, the second for loop iterate the basis of the surrogate function, and the last for loop iterate the feature dimension. For the matrix update function, we have five nested for

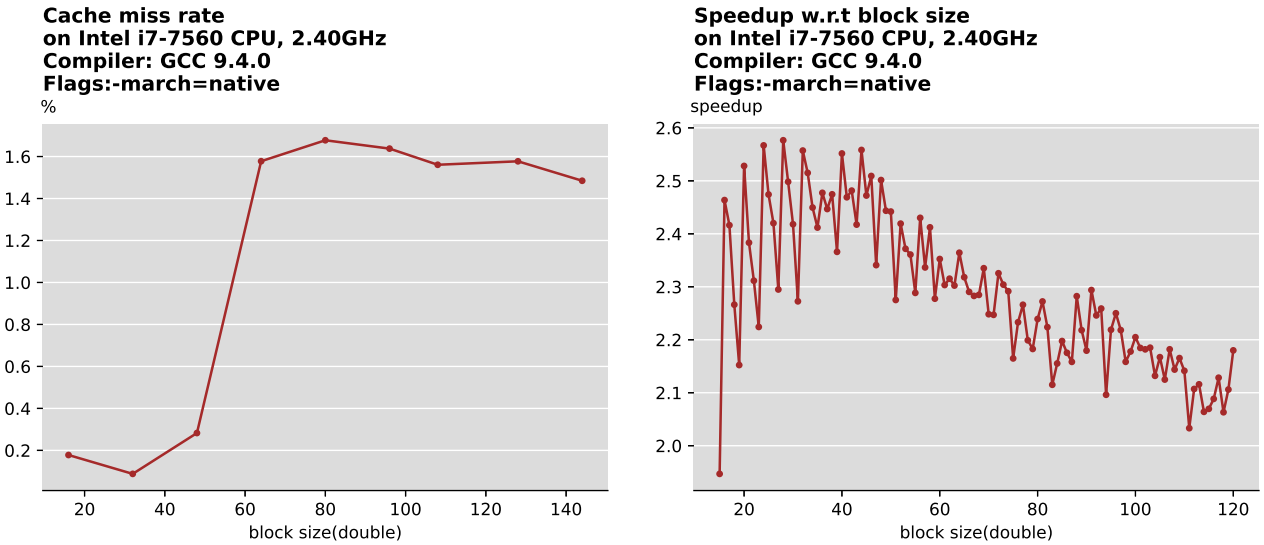


Fig. 4. (a) Cache miss rate of block Bunch-Kaufman algorithm for different block sizes. Block size 32 gives the minimal cache miss rate for the testing platform we used, which is about 0.1%. (b) The speedup of blocking the Bunch-Kaufman algorithm compared to the original non-block version for different block sizes. Block size 32 also gives the best speedup for the testing platform we used, which is about 2.6 times.

loops, the first two for loops are used to iterate the blocks, and the second two for loops are used to iterate the elements of the matrix to be updated, the last for loop is used to accumulate the value to be subtracted from the original matrix. If we only consider the last three for loops of matrix update, we have an accumulator in the innermost loop with similar operations (one FMA operation and another operation with the same port and latency) for these two functions. So we can use a similar approach to vectorize these two functions.

We use the vectorization of matrix update function $A = A - L_1 D L_2^T$ as an example. We first unroll by factor 4 in the for loop to iterate the row dimension of the matrix A (the third for loop) to take advantage of the vectorization and unroll by factor 8 in the last for loop to make full use of the port. The first unrolling means we need to compute the changes applied to four elements the matrix A (original matrix) at the same time. And the second unrolling means we need to use eight accumulators, which are stored in two AVX2 vectors in the last for-loop. Fig.1 shows the procedure to calculate the values to be subtracted from the four elements. The upper blue shaded part represents the procedure in the innermost loop. Since we unroll in the row dimension of the original matrix A, the corresponding variables are one row in L_1, D, and four rows in L_2. So we need 8 elements in matrix L_1 D, 4-by-8 elements in matrix L_2 and 4-by-8 accumulators for the innermost loop. Then we add the values to the accumulators using FMA. After this for loop, we will first add 8 accumulators to 4 accumulators for each element in A. And calculate the row-wise sum using three HADD and two permute intrinsic instructions.

Exec Unit (fp)	Latency [cycles]	TP [ops/c]	Gap [c/issue]	Port
FMA	4	2	0.5	0, 1
SUB	4	2	0.5	0, 1
MUL	4	2	0.5	0, 1
SQRT	15-16	4-6	0.1666-0.25	0

Table 1. Kaby Lake Micro-architecture

4. EXPERIMENTAL RESULTS

In this section, we demonstrate our experimental results in detail. Then, we show our performance plots for the overall pipeline and bottlenecks, and the cache misses and roofline model are illustrated. Finally, we explore how the optimization facilitates different optimization flags as well.

OPUS pipeline performance. As shown in Fig.3, the implementation of OPUS has bottleneck mainly in surrogate function fitting (step 5 and step 9) and evaluating (step 6b). Compare the original implementation and our final optimization, the improvement is around 6 times. Here we note that optimization (step 10) takes some time but are

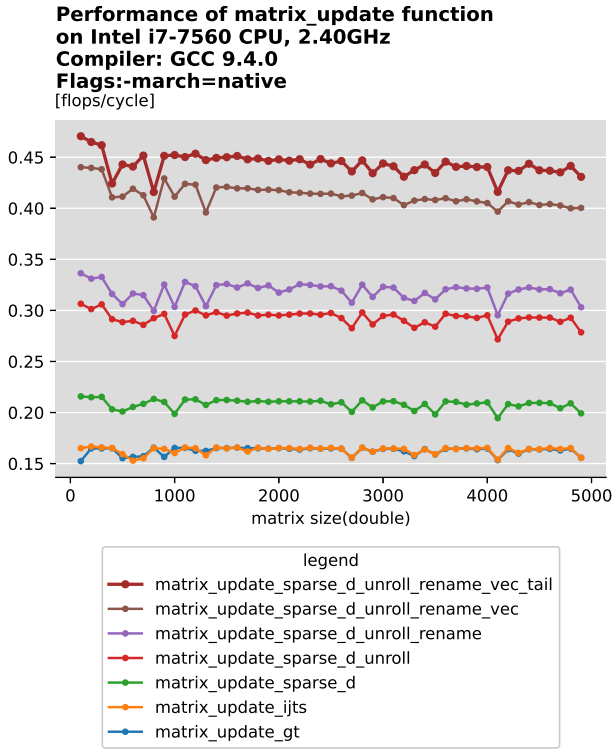


Fig. 5. Performance of the function "matrix_update", for every optimization attempt, the function is tested with different input matrix sizes, from 0 to 5000. The performance of the final optimized result (brown line) is about 3 times better than the original version.

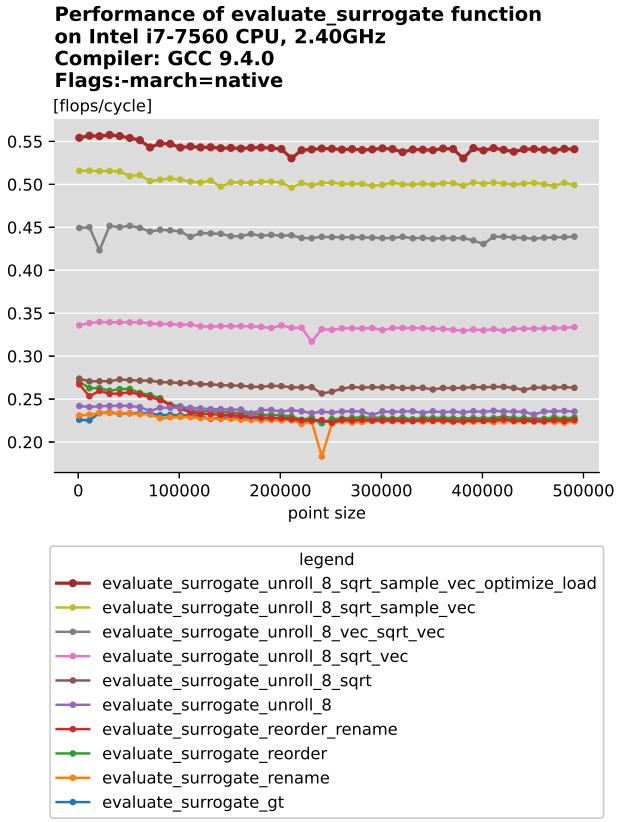


Fig. 6. Performance of the function "eval_surr", for every optimization attempt, the function is tested with different input particles number, from 0 to 500000. The performance of the final optimized result (brown line) is about 2.4 times better than the original version.

done by external lib CERES [8], so optimization is not implemented here.

Bottleneck performance. For each bottleneck steps mentioned above, we show detailed performance improvement in Fig.2. In (a) and (c), we see that before optimization, the performance drops as iterations increase and problem size increases. While after optimization, the performance increases stably. And the overall performance increases 5-8 times. Similarly, for (b) we around 2 times improvement.

Cache miss rate. Blocking Bunch-Kaufman is beneficial for cache locality since now each iteration we only touch a few blocks instead of the whole large matrix. As shown in Fig. 4(a) and Fig. 4(b), we test cache miss rate and speedup with respect to block size. Before using blocking, the cache miss rate is about 7.2%. With blocking, the minimum cache miss rate can decrease to 0.1% with block size 32. The big jump at block size 60 means that L1 cache cannot holds the full related data and it goes to L2 cache. Also with block size 32, the speedup is maximum, about 2.6 times comparing to non-blocking version. These results strongly support that our blocking optimization play an important role for speed improvement.

Matrix update function. We implemented 7 variants of optimization as shown in Fig.5. From bottom to up in the legend table, we implemented 1. original implementation; 2. loop reordering; 3. use sparsity of block diagonal matrix D; 4. unroll by factor 8 (remove dependency); 5. Index precomputation (enable by unrolling); 6. AVX optimization; 7. optimize the reminder part of unrolling.

Evaluate surrogate function. We implemented 10 variants of optimization as shown in Fig.6. From bottom to up in the legend table, we implemented 1. original implementation; 2. Index precomputation (enable by unrolling); 3. loop reordering; 4. rename variable after reordering; 5. unroll by factor 8 (remove dependency); 6. move sqrt operation out of the loop (remove dependency); 7. AVX optimization (only unroll part, along feature dimension); 8. AVX optimization (unroll part and sqrt part, along feature

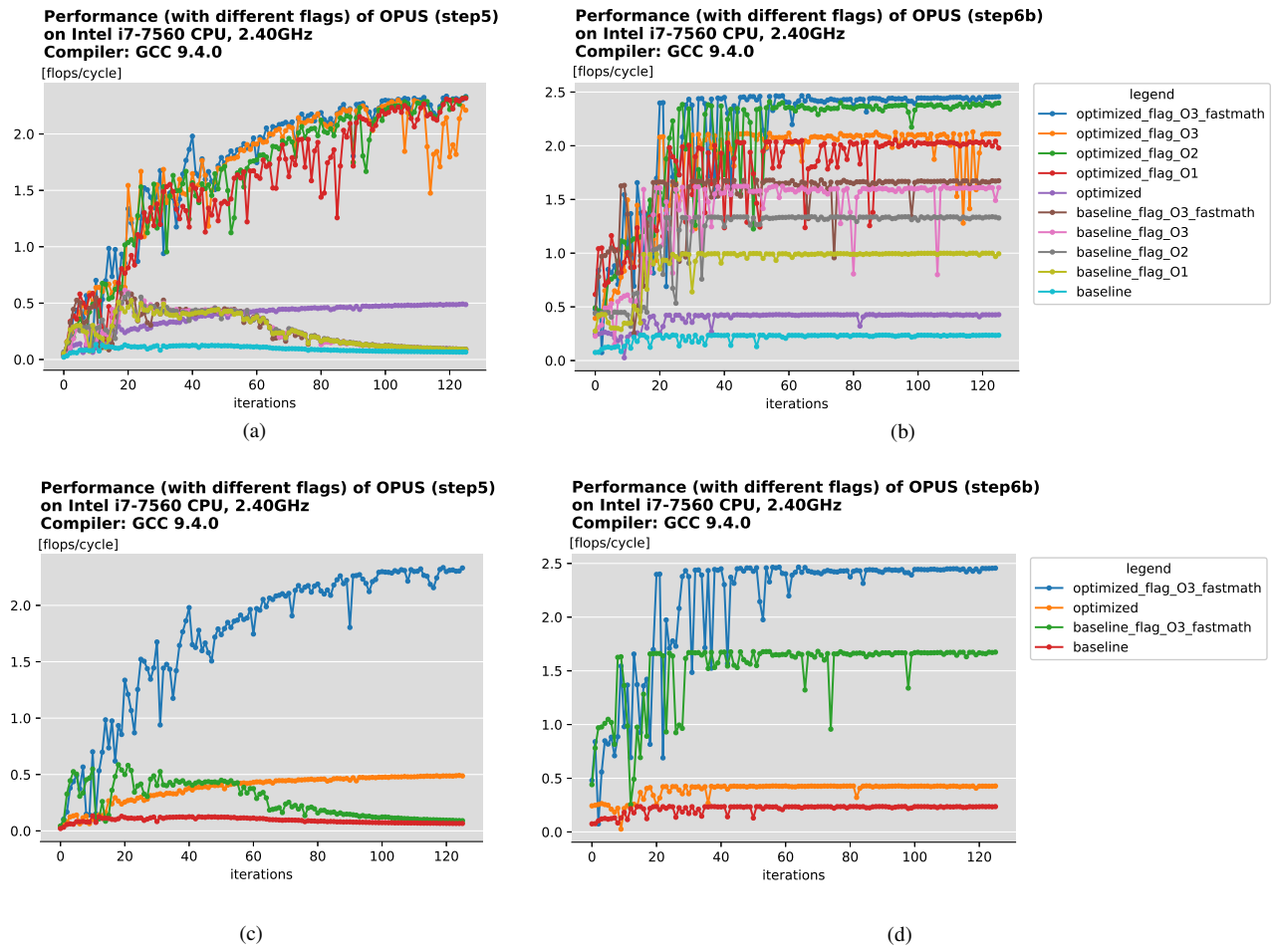


Fig. 7. (a) performance of step 5 (build surrogate function) with different flags. The Optimized version without any flag can outperform the baseline version with any flag especially when the input size is large. (b) performance of step 6b (evaluate surrogate function) with different flags. (c) performance of step 5 (build surrogate function) of baseline and optimized version with/without flags. (d) performance of step 6b (evaluate surrogate function) of baseline and optimized version with/without flags.

dimension); 9. AVX optimization (along sample dimension); 10. AVX optimization (along sample dimension), replace gather by load intrinsic operation.

Optimization with flags. In Fig.7(a) and Fig.7(b), we tested how the performance of different optimization flags changes after our optimization. Here er explore 4 optimization flags exception the vanilla version, which are ”-O1”, ”-O2”, ”-O3” and ”-O3 -ffast-math”. To show the effect, we provide a cleaner version in Fig.7(c) and Fig.7(d). Consider step 5, beside performance improvement of our optimization, we also notice that it enables much better and more stable flag optimization. For the evaluation part (step 6b), we got both improvement with and without optimization flag.

Roofline model. To conclude the main part of our optimization, we tested the roofline model of our current implementation in Fig.8. The green diamond and red triangle are the baseline implementation without and with optimization flag ”-O3 -ffast-math”. Their optimization counterpart are shown in the graph as purple square and blue square, which are both on the memory bound.

5. CONCLUSIONS

In this project, we use blocking, standard C optimization, and vectorization to speed up the OPUS algorithm. Compared to the baseline version, we achieved about 6 times speedup without any compilation flag. With “-O3 -ffast-math” compilation flag, our implementation enables more compiler improvement. Also, the roofline plot shows that we have achieved the memory bound.

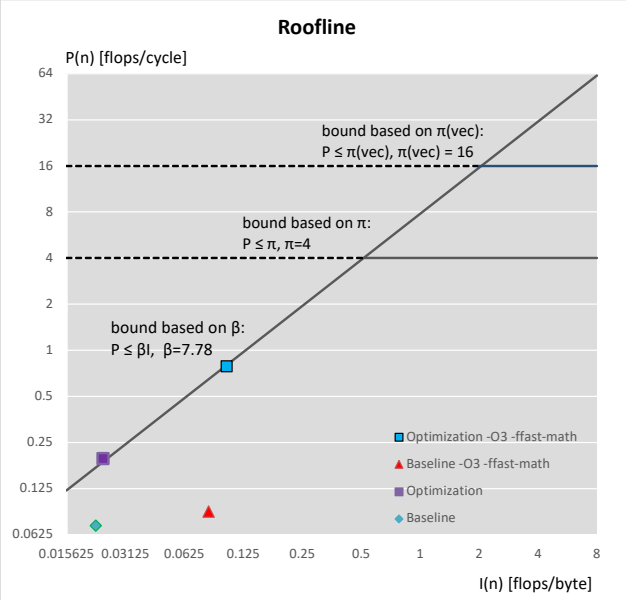


Fig. 8. Roofline plot for the whole OPUS pipeline. After our optimization, it hits the memory-bound of the machine we used for both compiling with no flag and compiling with "-O3 -ffast-math" flag.

Besides, our implemented blocking Bunch-Kaufman algorithm is not only useful in this OPUS pipeline but is helpful in any situation which needs to do factorization for indefinite symmetric matrices. Our results show that its performance would not suffer from increasing matrix size anymore.

6. REFERENCES

[1] Riccardo Poli, James Kennedy, and Tim Blackwell, "Particle swarm optimization," *Swarm intelligence*, vol. 1, no. 1, pp. 33–57, 2007.

[2] Rommel G Regis, "Particle swarm with radial basis function surrogates for expensive black-box optimization," *Journal of Computational Science*, vol. 5, no. 1, pp. 12–23, 2014.

[3] James Bunch and Linda Kaufman, "Some stable methods for calculating inertia and solving symmetric linear systems," *Mathematics of Computation - Math. Comput.*, vol. 31, pp. 163–163, 01 1977.

[4] James R Bunch and John E Hopcroft, "Triangular factorization and inversion by fast matrix multiplication," *Mathematics of Computation*, vol. 28, no. 125, pp. 231–236, 1974.

[5] Don Coppersmith and Shmuel Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, 1987, pp. 1–6.

[6] Mark T Jones and Merrell L Patrick, "Bunch–kaufman factorization for real symmetric indefinite banded matrices," *SIAM journal on matrix analysis and applications*, vol. 14, no. 2, pp. 553–559, 1993.

[7] Silvio Tarca, *Performance optimization of symmetric factorization algorithms*, Ph.D. thesis, Department of Computer Science, Cornell University, 2010.

[8] Sameer Agarwal, Keir Mierle, and The Ceres Solver Team, "Ceres Solver," 3 2022.