OPTIMIZING "OPTIMIZATION BY PARTICLE SWARM USING SURROGATES"

Gavin Gray, Valentin Ogier
Xavier Servot, York Schlabrendorff

Department of Computer Science
ETH Zürich, Switzerland

ABSTRACT

Numerical high-performance computing is an ever important domain with broad applications. The goal of this work is to iterate upon existing frameworks for optimizing black-box functions via particle swarm optimization – creating a high-performance framework for use within mathematical modeling. The target framework is the optimization by particle swarm using surrogates algorithm, or OPUS in short. The result is a well-tested, high-performance framework showing approximately a 5x speedup over baseline implementations.

1. INTRODUCTION

Mathematical modeling is the process of creating a mathematical representation of any real-world scenario. Such modeling is pervasive in the natural sciences, engineering, and even social sciences. These models contain many factors and can be quite complex observed in usages such as groundwater bioremediation [1], energy-storage [2], and watershed calibration [3]. These problems result in a complex high-dimensional optimization function that is evaluated many times. Despite the ever-increasing performance of hardware, evaluating these models are expensive and problem sizes keep growing.

Work to model these processes marches forward and many have worked to develop frameworks to reduce invocations of these expensive black-box functions. Typically such advances take advantage of mathematical properties to guide the optimization problem even when dealing with black-box functions.

Our efforts aim to build upon previously developed mathematical frameworks for evaluating such black-box functions. However, we improve upon this body of work by taking into account the system resources in an attempt to increase the performance of these frameworks. The result is a high-performance framework individuals can use with models to find the optimums of a black-box function.

2. BACKGROUND

One framework with which one can optimize black-box functions is referred to as *particle swarm optimization* (PSO). At a high-level, PSO numerically simulates a swarm of agents as they iteratively find the global optimum of black-box functions when the objective is *unknown*, while not requiring the function to be differentiable, in contrast to other optimization models such as gradient descent.

Introduced by Regis [4], *Optimization by Particle swarm Using Surrogates* (OPUS) is a specific variant of PSO where surrogates are used to minimize invocations of the black-box function which is expected to be expensive.

OPUS [4] focuses on a class of bound minimization problems of the form: min $f(x)$ s.t. $x \in \mathbb{R}^d$, $a \leq x \leq b$. Understanding the general PSO framework at a high-level is required before highlighting the advantages that OPUS has.

Standard PSO. Basic PSO schemes iteratively look for the minimum of a black-box function f. At each step, the function is evaluated over a set of candidate positions called the *particle population*. Between each step, these particles "move" in order to explore the region $\mathbf{R}^d$ and evaluate f at different points. Each particle is randomly initialized with a velocity and position; then, while it moves it will keep track of its "best evaluation" so far. At each time step the function is evaluated for each particle position and the new positions are determined accordingly. As the particles walk through the space they move according to their velocity but are also influenced by their local best position (cognition) and the global best position (social). The cognition and social parameters are determined at the start by the framework user and also randomly scaled at each iteration. These parameters largely influence how far apart the particles drift from each other and how biased they are to the stored local optimum. PSO will run iteratively until either (1) particles converge to an optimum position or (2) a user-defined maximum of iterations was reached.

OPUS. The contribution of OPUS is to accelerate convergence with a surrogate model approximating f, which is used as a heuristic when particle positions are updated. The

particle movement is now guided by positions that this surrogate has deemed promising. A set of *trial positions* are computed for each particle following standard PSO procedure and then the particle that minimizing the surrogate is chosen. The core idea is to keep surrogate evaluations cheap compared to an expensive invocation of f, which now only needs to be evaluated on new positions. This surrogate is interpolated from the set of points given by each particle evaluation. Keeping the surrogate up-to-date is easy as an evaluation of f gives a new pair $(x, f(x))$, which extends the set of points used for surrogate interpolation. Regis claims that this approach results in qualitatively better minima and faster convergence for an equal number of evaluations of f. It significantly outperforms the other methods on all the problems of a standard test set in which every method was allowed a maximum of 300 function evaluations. The step numbers referenced throughout this paper correspond to the steps given by Regis.

Fit Surrogate. Regis proposed using the cubic radial basis function (RBF) as a surrogate model, specifically: given n distinct particle positions $u^{(1)}, \ldots, u^{(n)} \in \mathbb{R}^d$ and their corresponding function evaluations $f(u^{(1)}), \ldots, f(u^{(n)})$ the RBF's interpolant is $s(x) = \sum_{i=0}^{n} \lambda_i \phi(\|x - u^{(i)}\|) + p(x), x \in \mathbb{R}^d$, where $\|\cdot\|$ is the Euclidean norm, $\lambda_i \in \mathbb{R}$ for $i = 1, \ldots, n$, $p(x)$ is a linear polynomial, and $\phi(r) = r^3$ is cubic.

To fit the cubic RBF model, define the matrices:
$\Phi \in \mathbb{R}^{n \times n}$ by $\Phi_{ij} := \phi(\|u^{(i)} - u^{(j)}\|), i, j = 1, \ldots n$;
$P \in \mathbb{R}^{n \times (d+1)}$ by row $p_i := [1, (u^{(i)})^T]$;
$F := [f(u^{(1)}, \ldots, f(u^{(n)})]^T \in \mathbb{R}^n$;
$\lambda := (\lambda_1, \ldots, \lambda_n)^T \in \mathbb{R}^n$;
$c := (c, \ldots, c_{d+1})^T \in \mathbb{R}^{d+1}$;

resulting in the linear system

$$\left[\begin{array}{c|c} \Phi & P \\ \hline P^T & 0_{(d+1)\times(d+1)} \end{array}\right] \left[\begin{array}{c} \lambda \\ \hline c \end{array}\right] = \left[\begin{array}{c} F \\ \hline 0_{d+1} \end{array}\right], \quad (1)$$

where c consists of the coefficients for the linear polynomial $p(x)$.

Determining the next positions of the particles. The calculation to generate a new particle position follows the same steps as in standard PSO, with the addition of trial particles. Namely, multiple trial velocity vectors are generated for each particle resulting in multiple trial particles. Each trial particle is evaluated on the surrogate model to only select the particles minimizing the surrogate, which are then in turn evaluated on function f in Step 7 & 8.

Cost Analysis. As a cost measure we defined floating-point operations (FLOPS), but did not further differentiate between the costs of the arithmetic operations or comparisons. As we will show in-depth later in section 5.2 solving two linear systems per main-loop iteration with $\mathcal{O}(N^3)$ dominate the program's asymptotic complexity, growing each iteration by up to the swarm size. Evaluation the surrogate grows linearly with swarm size and each time-step. The complexity of the remaining steps of the algorithm grow linearly with respect to either swarm size, dimensionality or number of trial particles.

3. BASELINE & ROADMAP

The original OPUS paper does not provide a publicly available reference implementation and only gives a high-level description of the steps of the algorithm. Provided here is an overview of the choices made for the baseline implementation as well as an evaluation of the performance bottlenecks. Following will be an outline of the optimization attempts for each subsystem and the respective optimizations applied, or attempted.

Unique points. In step 5, during the first surrogate fit, it is necessary to build the set of all previously evaluate *distinct* points. These points can either be previous particle positions or refinement points obtained in step 10 from the local minimizer. To ensure numerical stability of the surrogate a bitwise comparison is insufficient. To prevent numerical instability in the system solver, points that are close to each other should be considered equal.

Two approaches were attempted to check for unique points. The first approach tiled the point-space storing each occupied tile in a bloom filter. Each new point can be checked for by taking a hash and querying the bloom filter for the existence of a pre-existing equivalent point. This approach is solely dependent upon the dimension of the point-space and is reliable with the high confidence level of the bloom filter's accuracy.

The second approach naively computes all pairwise distances between the new point and the existing points. The cost of this approach increases linearly with the number of points.

We found that in practice the expense of hashing was too great in higher dimensions hence the naïve approach was chosen for the baseline implementation.

Local minimization. Step 10 requires finding an approximation of a local minimum of the surrogate function. Given that evaluating the surrogate is cheap, traditional non-convex optimization techniques apply, although we will restrict ourselves to using a naive grid base approach.

Profiling the baseline.

Table 1 presents the relative runtime of each step as a percentage of the total PSO runtime. The runtime gathered excludes memory initialization and is presented with the optimized version to demonstrate a fair baseline.

Apparent in *Table* 1 is the complete dominance of steps 5 and 9. These steps correspond to fitting the surrogate, thus, a separate line was introduced to demonstrate how

Step	Relative runtime (% total)	
	baseline	optimized
Step 5 & 9	0.99	0.99
↪fit_surrogate	0.99	0.99
↪sys_solver	0.97	0.97
Other	0.01	0.01

Table 1. Relative time spent in the different parts of the algorithm as percentage of the total runtime (excluding initialization). The runtimes were measured on the baseline and optimized version of the program, compiled with `icc` at the O3 optimization level. (population size = 20, dimension= 23, time max=120)

much of this step is within surrogate fitting and subsequently how much of surrogate fitting is spent solving linear systems.

Given these results our primary targets for optimization will be in linear system solving as presented in *Section* 5. Remaining parts of the system were not neglected and optimizations of them will first be presented and discussed in *Section* 4.

4. MINOR STEPS

The OPUS framework is comprised of many small parts. As previously mentioned the main amount of work is done within the system solver, however, this section concerns itself with everything else.

4.1. Caching and locality of `fit_surrogate`

`fit_surrogate` builds and solves a linear system. In the baseline, at each iteration, the matrix given to the system solver must be recomputed. As the matrix changes in minor ways between iterations, it is possible to avoid large amounts of computation by storing and reusing intermediate results Using the notations from Eq. 1, Φ (resp. P) is only modified between time t and $t+1$ by appending new rows and columns; not by changing any of the existing coefficients.

The values in P can be directly copied from the coordinates in the set of distinct points. The values in Φ however are the cubed pairwise distances between these distinct points. These can be cached between two calls to `fit_surrogate`. As an additional optimization, the distances to the newly added points are already computed in the caller, either step 5 or step 9, when checking for distinctiveness: if the distances are stored at this step then `fit_surrogate` can defer all floating-point operations to the system solver.

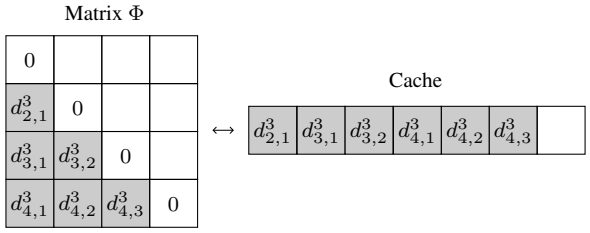

Fig. 1. Layout the cache for the block Φ in `fit_surrogate`.

The layout of the point cache is important for locality, namely, the symmetry of the matrix Φ and the fact that it has a zero diagonal can be exploited. The layout of stored coefficients is presented in *Figure* 1. By contiguously storing the coefficients, traversal incurs more cache hits and new points are simply appended to the end. We can further optimize by copying from the point cache into the matrix Φ in blocks. For example, after reading 16 values from the cache it is possible to write row by row two 4x4 blocks in Φ, one under the diagonal and one above the diagonal. As verified in the generated assembly, the 16 values are all loaded into floating-point registers, while the naïve unblocked approach would write column by column above the diagonal.

4.2. Low-hanging Vectorization

Many of the algorithm steps contain simple loops with independent calculation sin their body. These will be briefly outlined for completeness' sake though they are not a major focus.

Step 6a, generating trial particles. Determining the next position for each particle, as described in paragraph 2, results in a triple-loop dependent on swarm size, the number of trial particles, and the problem dimension. Optimization over the dimension-bound loop was achieved using vector instructions. The cognition and social factors are multiplied by random weights $\omega_1, \omega_2 \sim U[0,1]$, which requires two slow calls to `rand` in the inner loop. Avoiding the calls to `rand` can be avoided by moving this step to initialization because along with memory bounds, this is also bound by the maximum number of time steps.

Step 6b, surrogate evaluation. According to equation 2 all trial particles are evaluated on the surrogate model which is done via a double-loop. The outer loop, corresponding to set of distinct points where f has been evaluated, was unrolled by factor 2 and the inner-loop, depending on the problem's dimension, by factor 4.

Distinctiveness check. The distinctiveness check relies on computing the squared euclidean distances between one new point and all the existing points in the point cloud. The

```
for k ← [0, ..., N − 1):
    for i ← [k + 1, ..., N):
        r = A[i][k] / A[k][k]
        for j ← [k, ..., N + 1):
            A[i][j] -= r * A[k][j]
```

Fig. 2. Baseline Gaussian Elimination working on the augmented matrix $[A \mid b]$

computation is very similar to the one done at *Step 6b* and was vectorized with similar unrollings.

5. LINEAR SYSTEM SOLVING

A major focus of optimizations went into solving linear systems as almost the entire runtime is spent within the system solver (c.f. *Figure* 1). Dense linear algebra is a highly researched field with many known optimizations. In this work, the implementation of three algorithms were explored; each providing its own unique opportunities for optimization.

It is infeasible to present all algorithms in great detail thus some information will be omitted. For each of the following algorithms a form of pivoting will be required and is used in the implementation despite it being elided from pseudocode.

5.1. Gaussian Elimination

A good starting place for system solving is Gaussian Elimination. This grade school algorithm is widely known and is simple enough as demonstrated in the baseline pseudocode *Figure* 2.

The apparent problem with Gaussian elimination is that each row needs to be processed in succession. This dependency is introduced as a combination of finding a pivot (not shown in *Figure* 2) and the subsequent row subtractions.

Breaking the row dependency is difficult and was not achieved using Gaussian Elimination. Further discussion of blocking row iterations will be made in *Section* 5.3. Instead, focus will be placed on optimizing each individual step when possible.

The first observation to be made is the communication cost of pivoting. This logic is not present in *Figure* 2, nevertheless, finding a pivot will traverse down a column inducing roughly N cache misses per row iteration, where N is the height of matrix A. With a row-major layout, avoiding these cache misses is difficult. Further exploration of layout changes will be deferred to *Section* 5.3. Ignoring high-cost operations such as pivoting and memory movement operations such as swapping rows, there is very little left to optimize. Specifically, we can vectorize the inner loop to use Fused Multiply-Add (FMA) instructions. The performance gain thus obtained is not substantial, as seen in *Figure* 9. From an early stage, it was apparent that using a simple Gaussian Elimination algorithm would not suffice in order to break computational dependencies and that the cost of memory movement would remain high.

5.2. Block Triangular Solving

One attempt at reducing memory movement is the mathematical observation that our linear system (1) is equivalent to the following:

$$\left[\begin{array}{c|c} P & \Phi \\ \hline 0_{(d+1)\times(d+1)} & P^T \end{array}\right] \left[\begin{array}{c} c \\ \hline \lambda \end{array}\right] = \left[\begin{array}{c} 0_{d+1} \\ \hline F \end{array}\right]$$

With p_t defined as the population size at time step t, we have a matrix that has the following dimensions.

$$M = \begin{array}{c} \\ \\ \end{array} \begin{array}{c} \begin{array}{cc} d+1 & p_t \end{array} \\ \left[\begin{array}{cc} P & \Phi \\ 0 & P^T \end{array}\right] \end{array} \begin{array}{c} \\ p_t \\ d+1 \end{array}$$

We can take advantage of the structure of M in the Gaussian Elimination algorithm. In particular, for pivot $k \in [\![1; d+1]\!]$, finding the max pivot and performing Gaussian Elimination can be contained to rows $[\![1; p_t]\!]$. For pivots $k \in [\![d+2; N]\!]$, nothing changes.

Limitations. d is fixed with time while p_t grows linearly with time. Therefore, p_t eventually dominates the matrix size. Our algorithm ends up being very similar to standard Gaussian Elimination, reaching equivalent performance on large matrices.

5.3. LU Decomposition

In a hierarchical memory system, improving locality and reducing communication costs is crucial to improve performance. As observed in *Section* 5.1 the locality of Gaussian Elimination is insufficient and reducing this via triangle solving methods in *Section* 5.2 also failed. Thus, attention needs to be turned elsewhere in hopes of achieving higher performance. Many high-performance libraries offer system solving via LU Decomposition and not Gaussian Elimination because it allows the solving of subsequent right hand sides when the matrix A has not changed. However, it provides additional algorithmic potential for reducing memory movement as will be discussed. The naïve implementation of LU Decomposition does not differ much from Gaussian Elimination as apparent in *Figure* 3.

A naïve implementation of LU Decomposition will suffer from the problem of locality seen with Gaussian Elimination, in fact, it will perform worse due to the additional

```
for i ← [0,...,N):
    for j ← [i+1,...,N):
        A[j][i] /= A[i][i]
        for k ← [i+1,...,N):
            A[j][k] -= A[j][i] * A[i][k]
```

Fig. 3. Naïve LU Decomposition without pivoting.

writes below the diagonal. One major performance killer is pivoting and scaling a column when using a typical row-major array layout in C. An observation about this specific type of system solving is that the incoming matrices are always symmetric. This fact was exploited in *Section* 5.2 however it was not enough. Here, by using a column-major layout, the frequent operations of pivoting and scaling a column will be much faster due to better spatial locality. [1] For the remainder of the discussion on LU Decomposition, including figures, it will be assumed that matrices are in column-major order.

Performance gains can be achieved by modifying the algorithm slightly to use blocking and thus improve locality on a hierarchical memory system. The main idea of this modified algorithm is to decompose a column of the matrix applying batched swaps and updates to the trailing submatrix. This approach is similar to the recursive partition algorithm proposed by Sivan Toledo [5], however, unlike the recursive algorithm, this iterative version requires constant updates to the trailing submatrix. Toledo concluded that the recursive implementation performed fewer reads and writes and does not depend on choosing a block size, however, the iterative version has been demonstrated to come within a factor of two on square input matrices. An example of such an LU Decomposition can be seen in wide use by LAPACK's blocking `dgetrf` subroutine [6].

The pseudocode provided in *Figure* 4 shows a general outline but it also follows the naming convention of the BLAS subroutines. Although it should be noted that in the implementation these subroutines are hand-coded and optimized; only the naming convention was borrowed.

The subroutine `dgetrf2` performs the non-blocked LU Decomposition and `dtrsm_L` is a specific version of the `dtrsm` subroutine that assumes a *lower* triangle matrix with *unit-diagonal*. With the new blocked LU Decomposition several implicit advantages were introduced: (1) A system-dependent variable. Namely, the block size `b` which implicitly defines the frequency at which a `dgemm` is performed. This variable can be chosen due to system specifications or via input in a search based optimizer such as ATLAS.

[1] Due to the symmetry of the problem, this "column-major" layout can simply be achieved by swapping the row and column indices when indexing into an array.

```
for j ← [0,...,N):
    dgetrf2(N-j, b, A[j][j])
    if (j+b < n):
        dtrsm_L(b, N-j-b, A[j][j], A[j][j+b])
        dgemm(N-j-b, N-j-b, b,
            A[j+b][j], A[j][j+b],
            A[j+b][j+b])
```

Fig. 4. Outline of blocked LU Decomposition.

(2) BLAS-3 subroutine `dgemm`. It is well known that the BLAS-3 interface for dense linear algebra is more performant than subroutines performed on vectors and `dgemm` in particular is well studied.

Now that the algorithm has been described as a simple composition of BLAS subroutines, each part can be analyzed in isolation and the optimizations outlined. Many of these subroutines can be optimized in some form for the specific needs of system solving. However, several are either uninteresting in their implementation and are composed of others, or do not lend themselves well for optimization nor are they a main bottleneck for the computation. These subroutines will be discussed no further and include:

- `dtrsm` : solves the matrix equation $A*X = B$ when A is lower / upper triangular. This is equivalent to a rank-r update and the number of reads / writes is less than performing the decomposition.
- `dswap` : interchanges vectors.
- `dlaswp` : interchanges a series of rows.
- `dgetrs` : solves the full system with an already LU decomposed A.

Matrix Multiplication (`dgemm`). As previously stated, canonical matrix multiplication $C = C - A * B$, also referred to as *MMM*, is a well studied problem and many high-performance implementations exist. The dimension of A, B, C are $M \times K$, $K \times N$, and $M \times N$ respectively. Due to its familiarity, time will not be spent discussing blocking and the base algorithm is assumed to use both cache and a form of register blocking for better locality. Terms *mini MMM* and *micro MMM* will be used to describe an MMM blocked for cache and an MMM blocked for registers, respectively. Discussion on the size of blocks at both the cache and register levels will be discussed later.

In performance computing a good mental model to have is that of a stream of bytes constantly arriving at the time they are needed for computation. The trick to maintaining this stream is avoiding unnecessary cache misses, namely, when false assumptions are made about the location of memory. Take for example a *mini MMM* on small matrices.

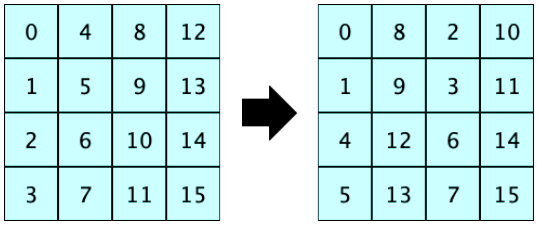

Fig. 5. Visual representation of packing mini matrix A assuming only 2x2 micro matrix computations are available.

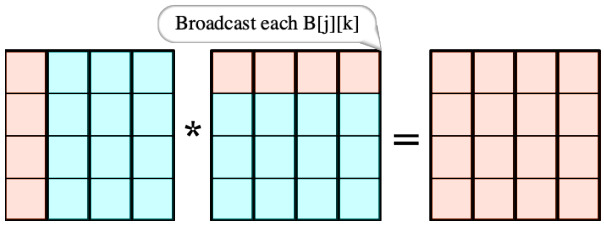

Fig. 6. A 4x4 micro MMM $C = C - A * B$ where elements of B are broadcast to a `__m256d`.

Though the dimension of the working set is small, the leading dimension of the matrix in which the set is embedded could be much larger. An observation for this project is that the leading dimension grows very fast, and not before too many iterations, a single matrix row can span multiple memory pages. To avoid TLB misses, a step in the right direction is to pack the mini matrices A and B into local buffers. However, how the bytes are packed before the mini MMM can make a difference in performance later; namely, we can coordinate the micro MMM computations and the packing to improve locality.

A visual example of packing mini matrix A can be seen in *Figure* 5. Before packing, there is a 4x4 column-major matrix which gets packed into a local buffer assuming *only* 2x2 micro MMM computations are available. This packing strategy has the advantage of allowing the following micro MMM computations to read sequentially from memory, improving spatial locality.

Up to this point, the main concern with `dgemm` has been avoiding both cache and TLB misses. These TLB misses can occur when the outer matrix has a large leading dimension and data stored in adjacent columns could be stored in different memory pages; the introduced solution is to pack matrices locally. However, a discussion on choosing the optimal M, N, and K blocks was deferred to this point. Historically, a neat way to find the optimal parameters was through a search as demonstrated by ATLAS [7]. In 2005, a *model-based* approach was introduced by Yotov et al. [8] where such parameters can be chosen using system variables. One caveat with the model approach is that it fails to address all possible optimizations. One such example is tiling for L2 cache as mentioned by the authors in the *Additional details*. In this project an auto-tuning search was used in addition to hand selected parameters which optimized for L2 and L3 cache tiling. In summary, a block size of $N = M = K = 160$ was found to be optimal for our purposes. A cutdown version of the autotune results can be found in the Appendix at *Figure* 11.

An appropriate ending to a long journey of matrix multiplication is to discuss vectorization of the *micro MMMs*. As previously mentioned, when matrices A and B are packed into contiguous memory, the data is accessed sequentially.

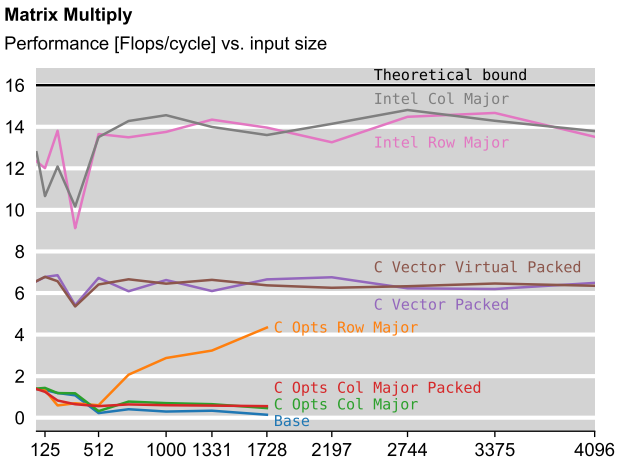

Fig. 7. Performance results of `dgemm` for random inputs when $M = N = K$.

To show how these micro MMMs can be vectorized it is best to view loads in their unpacked state as in *Figure* 6. There, a 4x4 micro MMM is depicted, that is $M_U = N_U = K_U = 4$. Observable in the figure is that a single column of A, namely A_j, gets loaded into a single `ymm` register. Then, each row value of B gets broadcast into its own `ymm` register allowing for 4 `FMA` instructions before the vectors are stored back in memory.

Examples of 2x2 and 4x4 micro MMMs have been mentioned, however, a micro MMM can be generated for each combination of dimensions in the form $A \times B$: $\{8, 4, 2, 1\} \times \{4, 2, 1\}$. These dimensions ensure that all matrix sizes are handled independently of the blocking sizes chosen, and that all values needed for one iteration in the k direction are held in registers.

In *Figure* 7 the performance of each algorithmic change is displayed when evaluated on random input matrices when $M = N = K$. As seen, the steps outlined above did in fact end in a performance improvement. The less performant variants were killed early during testing due to slowdowns, nevertheless, interest is in the *C Vector Virtual Packed* variant. For reference, the Intel MKL [9] implementation was included to demonstrate implementation shortcomings.

Basic LU Decomposition (`dgetf2`). Within the higher-order problem of creating an iterative algorithm for LU Decomposition, there still is a need to perform basic LU Decomposition. The main logic of the LU Decomposition happens within the subroutine `dgetf2`. The logic is quite straightforward and mimics that of Gaussian Elimination. It can be simplified into three simple steps: (1) find pivot and swap rows, (2) scale the column vector, and (3) perform a rank-1 update of the form $A = A - u * v$ Description of finding a pivot row will be deferred until later via the subroutine `idamax`, however, the remaining two steps are surprisingly simple to do trivial optimizations on. Scaling a column vector can be done with the standard vector instruction `_mmm256_mul_pd` because memory is laid out sequentially. This again being another benefit to using a column-major memory format. A rank-1 update is also trivially optimized by turning the inner computation into the FMA instruction `_mm256_fmadd_pd`.

Index of maximum absolute value (`idamax`). Without partial pivoting, LU Decomposition suffers from numerical instability due to roundoff error. A crucial part of pivoting is finding the index of the value with maximum absolute value. With a row-major layout, traversing the column and finding a pivot is a very costly operation as previously discussed. However, this operation can be vectorized with a column-major layout. This vectorization may not be obvious at first because comparisons are dependent on the previous. Breaking the dependency chain can be done with accumulator values, a similar technique used when performing a sum reduce. Choosing the maximum can also be achieved easily using a series of compares and blends. After the main loop, the accumulator is stored in a local buffer and final maximum chosen with three compares.

Now that all BLAS subroutine components have been outlined to some degree, composing them creates an elegant implementation of Toledo's iterative blocked LU Decomposition algorithm. The BLAS interface name for this subroutine is `dgetrf`. The last piece to this performance puzzle is choosing an optimal block size for the outer decomposition. In his work, Toledo suggests that for square matrices a block of size $max(\frac{M}{n}, \sqrt{m})$ is optimal in almost all cases. To find the best system specific block size auto tuning was used. This process generated multiple versions of the `dgetrf` subroutine with varying block sizes and benchmarked them against each other. The results can be seen in *Figure* 8 where it can be observed that a block of $b = 64$ performed the best.

After such a long journey of performance promises some concrete data needs to be shown. *Figure* 9 shows the relative performance of each system solver. As expected, both the triangle and Gaussian Elimination solvers degrade with larger input sizes and data collection was paused due to large wait times. The LU Decomposition plotted is that described in *Section* 5.3. To give perspective on the performance of both the iterative Toledo algorithm and the custom implementation of `dgemm` two versions utilizing the Intel MKL have been included. The variant *LU Intel* `dgemm` uses the same custom LU Decomposition algorithm but substitutes the custom `dgemm` for Intel's. The scary red line is Intel's full implementation of system solving (`dgesv`).

Wading through performance code is not easy but even modest improvements in performance as achieved in this study are satisfying and highly educational. Finishing this system solving story, due to algorithmic changes that resulted in memory optimization and utilization of the BLAS-3 interface, LU Decomposition was able to achieve roughly a 5x speedup over baseline Gaussian Elimination.

6. EXPERIMENTAL RESULTS

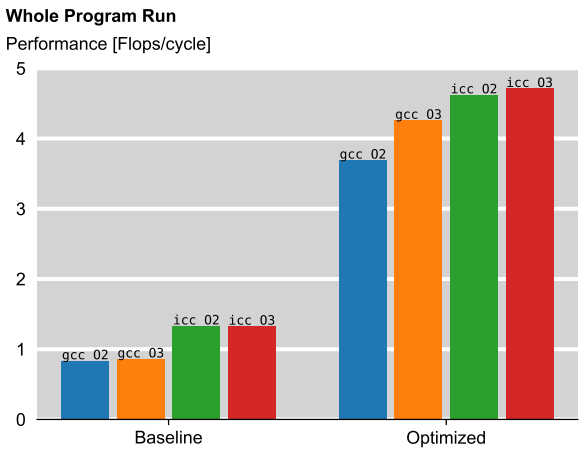

Fig. 10. Average performance of the baseline and optimized implementations on full solver runs when compiled with different compilers (icc or gcc) and optimization level (O2 or O3). The execution times were measured on the computation steps only, excluding the initial memory allocation phase.

The algorithms and system presented above have been extensively tested to ensure that all system changes did not modify results in any significant way. These tests are comprised of thousands of randomly generated inputs for linear algebra routines and results are automatically tested against the Julia standard *Linear Algebra* module implementation [10]. Overall system runs were checked for backward compatibility with the baseline.

Experimental Setup. We ran all the tests on a Dell Latitude 7480 with an Intel Core i5-6300U CPU running Ubuntu 22.10, with Intel Turbo Boost disabled. The CPU's L1, L2, L3 caches are resp. 128ḰiB, 512ḰiB, 3ṀiB big.

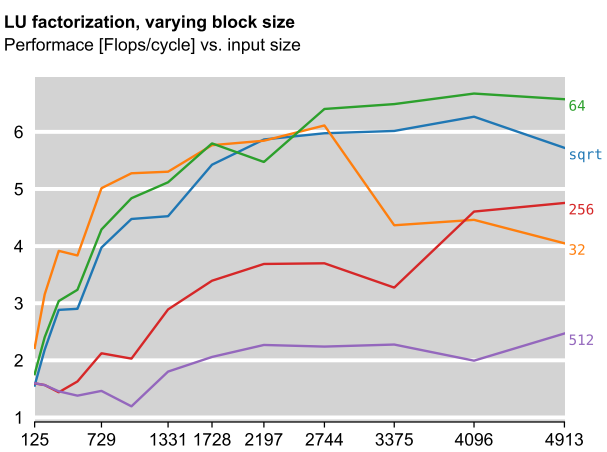

Fig. 8. Performance of LU Decomposition with varying block sizes for `dgetrf`.

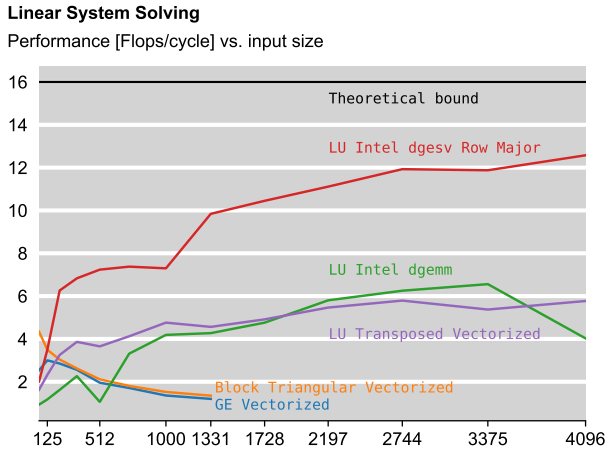

Fig. 9. Performance plots of the best system solver from each category.

We experimented with two compilers: the GNU C Compiler (gcc) and the Intel C Compiler (icc). As presented on Figure 10, the Intel C Compiler results in better performance for the baseline and the optimized version. Additionally, we tried various options of compilation flags: using a higher general optimization level (O3 instead of O2) can result in larger code size, which can be detrimental for performance. In our case however, the effects were positive overall, if limited with icc. The use of O3 and icc already enables various important compilation flags such as the auto-vectorization and non-IEEE compliant floating point arithmetic. Additionally, we enabled link-time optimization (LTO), and disabled a number of runtime security features (Control Flow protection, Stack Protectors, Position Independent Executable); they all resulted in minor performance boosts at best.

The resulting performance of both the whole solver optimizing a black-box function, and of the core components on different problem sizes was evaluated. The table 2 summarizes the performance improvement of each step on an example run of OPUS: as expected, a good amount of performance was gained in the system_solver (x4.6 increase), which immediately converts in performance gain in fit_surrogate and step 5 and step 9.

Step	Performance (flops/cycle)	
	baseline	**optimized**
step5	0.93	3.95
step6	1.94	3.07
step7	1.52	1.31
step8	0.00	0.00
step9	0.94	4.18
step10	1.99	2.89
step11	2.13	2.26
fit_surrogate	0.93	4.07
system_solver	0.90	4.16

Table 2. Top: performance of each step in the main loop of the algorithm. Bottom: focus on the functions fit_surrogate and system_solver (used in step 5 and 9). The performance was measured using the Performance API (PAPI) interface on the baseline and optimized version of the program, compiled with `icc` at the O3 optimization level. (population size = 20, dimension= 23, time max=120)

7. CONCLUSION

With this paper, we added to the large body of work on PSO. More specifically, while most of the research in this field focuses on finding new PSO variants and mathematical formulations, our work succeeded in delivering a high-performance PSO framework by devising an efficient implementation for OPUS, an existing, unaltered PSO variant. In particular, our work achieved its stated goal by demonstrating the necessity of implementing high-performance linear algebra primitives: a linear system solver, and underneath fast matrix multiply. After trying generally applicable optimizations, our work was aided by the specific characteristics of the problem at hand, and of the target machine. Using hand-written optimizations, and through auto-tuning, our implementation achieves a 4.0 times performance improvement over a baseline overall, with a 4.5 times performance improvement specifically in the system solver.

8. CONTRIBUTIONS OF TEAM MEMBERS

Gavin Gray. Worked on the implementation and optimization of the Gaussian Elimination and LU Decomposition pieces of the linear system solving framework. This includes basic C-optimization (scalar replacement, etc), basic blocking, memory optimization as discussed in *Section* 5.3, vectorization, and implementation of the iterative Toledo algorithm discussed in *Section* 5.3. These optimizations include the BLAS subroutines: `dgemm`, `dgetf2`, `dgetrs`, `dlaswp`, `dswap`, `dtrsm`, `idamax`. Additionally, he worked on the auto tuning framework for system dependent variables.

Valentin Ogier. Worked on optimizing the surrogate fitting function: basic C optimizations, caching (and cache layout for locality) not only at the `fit_surrogate` level but across steps, blocking the copy. Basic C optimizations and vectorization of the distinctiveness check. Experiments with compilers and compiler flags. Performance measurements of individual steps and whole system runs with PAPI.

Xavier Servot. Worked on the block triangular system solver. This includes basic C-optimization, basic blocking and vectorization.

York Schlabrendorff. Worked on cost analysis, and optimizations of `surrogate_eval`, Step 4 and 6, namely C Optimizations and SIMD.

9. REFERENCES

[1] Jae-Heung Yoon and Christine A Shoemaker, "Comparison of optimization methods for ground-water bioremediation," *Journal of Water Resources Planning and Management*, vol. 125, no. 1, pp. 54–63, 1999.

[2] Thongchart Kerdphol, Kiyotaka Fuji, Yasunori Mitani, Masayuki Watanabe, and Yaser Qudaih, "Optimization of a battery energy storage system using particle swarm optimization for stand-alone microgrids," *International Journal of Electrical Power & Energy Systems*, vol. 81, pp. 32–39, 2016.

[3] CHRISTINE A. SHOEMAKER, ROMMEL G. REGIS, and RYAN C. FLEMING, "Watershed calibration using multistart local optimization and evolutionary optimization with radial basis function approximation," *Hydrological Sciences Journal*, vol. 52, no. 3, pp. 450–465, 2007.

[4] 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.

[5] Sivan Toledo, "Locality of reference in lu decomposition with partial pivoting," *SIAM Journal on Matrix Analysis and Applications*, vol. 18, no. 4, pp. 1065–1081, 1997.

[6] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, Pennsylvania, USA, third edition, 1999.

[7] "Atlas homepage," `http://math-atlas.sourceforge.net/`, Accessed: 2022-06-23.

[8] K. Yotov, Xiaoming Li, Gang Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill, "Is search really necessary to generate high-performance blas?," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 358–386, 2005.

[9] Intel Corporation, "Developer reference oneapi math kernel library – c," 2021.

[10] Jeff Bezanson, Stefan Karpinski, Viral B Shah, and Alan Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

A. APPENDIX

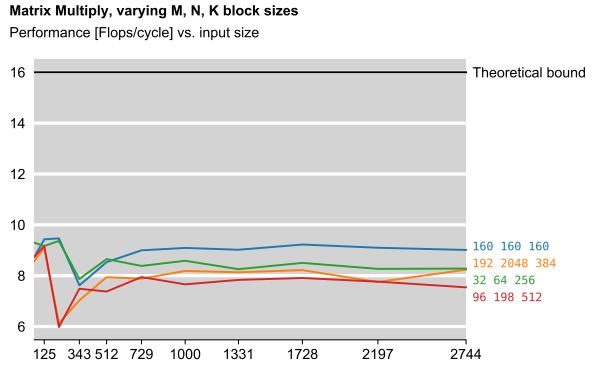

Fig. 11. Pruned auto tune results for matrix multiplication for block sizes M, N, K.