DOI: https://doi.org/10.15276/eltecs.44.120.2025.8
UDC 004.94

Optimization of the Model for Calculating the Thermochemical Equilibrium of Combustion Products for the Construction of Libraries of Thermodynamic Parameters

Valentin Davydov, Olga Tarakhtiy, Ganna Lysuk
Odesa Polytechnic National University

Abstract. The paper presents a detailed analysis of the algorithm for calculating the thermochemical equilibrium of fuel combustion products in an isolated system. The sequence of stages of numerical modeling, the structure of the software implementation, and the profiling to identify critical code sections are considered. On the basis of the obtained results, a step-by-step optimization was carried out, including modification of numerical methods, reorganization of the data structure, reduction of the amount of computation, and attempts to use multithreading. It is shown that a significant performance gain was achieved mainly due to algorithmic solutions. All experiments were performed for a scenario of multiple repetition of calculations, which is typical for the tasks of building databases of equilibrium parameters. As a result of the optimization, the program execution time was reduced by more than 40 %, which confirms the effectiveness of the implemented solutions for massive numerical modeling.

Keywords: thermochemical equilibrium, fuel combustion, numerical modeling, performance optimization, code profiling, algorithmic improvements, multithreading.

Introduction

Thermochemical modeling plays a main role in numerous engineering and scientific fields, including the development of fuels, combustion systems, engines, and process safety analysis. One of the characteristic features of such problems is the presence of complex nonlinear dependencies between system parameters (temperature, pressure, volume, chemical composition), which requires multiple numerical integrations, solving systems of equations, and finding an equilibrium state according to thermodynamic criteria.

In real-world calculations, models describing combustion are often high-dimensional and contain dozens or hundreds of chemical components and reactions. Even when using simplified models (e.g., reduced to 10-15 components), the computational load remains significant, especially when iterative solutions are based on Newton's method or similar schemes. In this context, an efficient implementation of software for such models becomes critical both to ensure acceptable calculation time and to be able to scale to more complex configurations.

The problem is complicated by the fact that ready-made computational libraries or universal packages (e.g., Cantera, CHEMKIN) seldom allow to take note the specifics of a particular physical process or optimize computations at the level of the processor microarchitecture [1, 2]. That is why there is a need to investigate methods for targeted optimization of calculations, taking into account the data structure, peculiarities of the implementation of thermodynamics functions, reuse of intermediate results, and effective control of the accuracy and stability of iterative processes.

1 Analysis of the state of the art in thermochemical modeling

Optimization of calculations in thermochemistry problems is not only an engineering challenge but also a subject of scientific interest, since not only performance but also the ability to conduct numerical experiments in real time depends on the chosen implementation methods, which is especially important in conditions of limited resources or when using computational models within interactive simulators and decision support systems [3].

Modern methods of thermochemical modeling involve the construction of systems that reflect complex nonlinear dependencies between fuel mixture components and environmental parameters. Such systems are usually described by chemical equilibrium and enthalpy balance equations, which require iterative numerical algorithms that include internal optimization cycles, approximations of thermody-

namic functions, as well as derivative calculations and accurate interpolation [4, 5].

Of particular computational complexity are inverse problems of interpreting the composition of a gas mixture, which arise, for example, in cases where it is necessary to restore the original fuel formula based on the known characteristics of its combustion products. Such problems are characterized not only by the need to repeatedly solve the direct problem for different configurations, but also by a sharp increase in the number of possible options when the number of chemical elements taken into account in the model increases. As shown in [6], each addition of a new element to the gross formula causes an avalanche-like increase in the number of calculations, which, in turn, requires significantly more computing resources and time.

In addition, most of the available libraries and software packages for thermochemical calculations (in particular, CHEMKIN and Cantera) provide broad functionality but require significant adaptation when integrated into production or scientific computing chains. The issue of effective preparation of such systems for batch processing of a large number of tasks with different input data is especially relevant [1, 2].

Thus, thermochemical modeling is not only a scientific but also an engineering task in which the requirements for accuracy, reproducibility, and calculation speed must be balanced with limited resources and the need for scalability of solutions.

In modern thermochemical modeling, numerical libraries and software tools are widely used to provide effective solutions to problems of chemical equilibrium, kinetics, and heat and mass transfer. Among these tools are the following:

Cantera is an object-oriented software package that supports modeling of chemical kinetics, thermodynamics, and transport processes in the gas and condensed phases. Cantera is widely used to model combustion processes, reactors, and other thermochemical systems [1].

CHEMKIN is one of the oldest and most widely used packages for analyzing gas-phase chemical kinetics. It provides accurate modeling of complex chemical reactions under various conditions, including high-temperature processes [2].

Thermochimica is a library for calculating thermodynamic equilibrium in multicomponent multiphase systems, which allows modeling complex reaction media with high accuracy [7].

FastChem 2 is an updated version of the semi-analytical code for calculating chemical equilibrium in the gas phase, which supports a wide range of chemical elements and provides high computational speed [8].

Combustion Toolbox (CT) is a new open source software package for solving chemical equilibrium problems in the gas and condensed phases, which includes modern free energy minimization algorithms and supports the modeling of shock waves and rocket engines [9].

Common techniques used in thermochemical modeling include:

Gibbs free energy minimization to determine the equilibrium composition of the system at given temperatures and pressures. This approach is implemented in many modern software packages, such as Cantera and Combustion Toolbox [1, 9].

Iterative methods for solving systems of nonlinear equations that describe chemical kinetics and thermal processes. For example, the Newton-Raphson method is widely used to find roots in chemical equilibrium problems [9].

Use of approximation methods to reduce computational complexity, such as dimensionality reduction methods for chemical mechanisms. This reduces the number of required calculations without significant loss of accuracy [5].

Parallel computing and the use of high-performance computing resources to speed up the calculation of complex models. For example, FastChem 2 supports parallel computing, which can significantly reduce calculation time [8].

The use of these libraries and methods allows to efficiently model complex thermochemical processes, optimize computations, and ensure high accuracy of results.

2 Objective

In the context of modern thermochemical modeling, which is accompanied by high computational complexity, the problem of increasing the efficiency of numerical algorithms without losing the accuracy of the results is especially relevant. This article is devoted to the study of the possibilities of targeted optimization of computational processes in an applied fuel combustion model based on the Peng-Robinson equation with consideration of chemical equilibrium and soot formation products.

The mathematical apparatus, model structure, and its physicochemical assumptions are described in detail in a companion paper on the development of a combustion model and analysis of the influence of the parameters of the initial mixture on the temperature and thermal characteristics of the system [10]. In this publication, the focus is not on the physical or thermodynamic nature of the process, but on the justification of decisions related to the implementation of the numerical algorithm, its structural organization, and optimization of computational performance.

The peculiarity of the study is that the developed program is considered not as a one-time calculation tool, but as a tool for building a large-scale library of thermochemical data. Such a task involves millions of model runs for different initial conditions, which makes it impossible to perform long computations. Therefore, even minor improvements in performance, which may seem insignificant in an isolated case, become essential on the scale of the entire process.

Regarding this paper:

- analyzed bottlenecks in the computing structure based on the profiling results;
- local optimizations are implemented (computational minimization, transformation of cycles, elimination of redundant actions);
- architectural changes were proposed to better organize computational dependencies;
- The feasibility and potential of using parallel computing was checked;
- a step-by-step assessment of the impact of each change on the execution time was performed;
- recommendations for organizing the code to achieve maximum efficiency were formulated.

The ultimate goal of the study was to achieve the maximum possible speedup of the program, which would allow it to be effectively used to build a large-scale thermochemical data library. It was expected that the specifics of the model - numerous similar computations with predictable dependencies - would create favorable conditions for the widespread use of parallel computing and a significant performance gain. Based on this assumption, the study consistently implemented, tested, and analyzed both local and global optimization approaches in order to exhaustively use all potential performance reserves.

3 Problem statement

The mathematical model implements an iterative calculation of the combustion temperature, mixture temperature, and pressure corresponding to chemical equilibrium in a given volume. The initial data are the ambient temperature, pressure before the start of combustion, fuel mass, the ratio of chemical elements in the fuel, and the volume of ballast air. For each fuel, the stoichiometric ratio of oxygen is determined, the amounts of oxidizer and ballast components are calculated, and then the energy and volume balance is performed.

The task is nonlinear, multi-step, and involves iterative adjustment of several interrelated parameters - pressure, temperature, volume, and mixture composition - based on a combination of chemical, thermodynamic, and volumetric relationships.

The first step is to calculate the enthalpies and entropies of all components of the gas mixture at a given temperature. Based on these values, the chemical equilibrium constants for the main dissociation and recombination reactions are determined. Next, an internal cycle begins, in which mole volumes are calculated for each of the components using the Peng-Robinson equation, and the total volume of the mixture is compared to a fixed volume of the reaction chamber. By adjusting the pressure, a balance of volumes is achieved, which allows determining the combustion pressure.

The second stage involves calculating the chemical composition of the mixture that meets the condition of chemical equilibrium. The initial system of equations is specified as a matrix of stoichiometric coefficients for the main components and intermediate products. The matrix is inverted, and the vector of free terms is formed taking into account the current value of the mole amounts. By iteratively updating the composition of the mixture, the model ensures that the specified accuracy in terms of relative changes in concentrations is achieved.

The third stage consists of an external temperature iteration cycle: the calculation of the equilibrium constants and mixture composition is repeated for each new temperature approximation until the total enthalpy of the combustion products balances the enthalpy of the initial fuel and air. This is how the combustion temperature is determined.

The fourth stage simulates the process of mixing hot products with air residues. Given the known enthalpy of the resulting mixture, the temperature at which this enthalpy is provided by a mixture with a certain composition is selected. This makes it possible to set the temperature of the mixed gas phase after combustion.

The fifth step is to re-solve the Peng-Robinson equation, this time for the post-mixing conditions, at the found temperature and new pressure. The pressure value is selected so that the total volume of all gas components matches the geometric volume of the chamber. The result is the final pressure value of the mixture after all reactions and expansion processes have been completed.

Thus, the problem is solved as a hierarchy of nested iterations: from the local volume balance through chemical equilibrium to the global energy balance of the entire system. All calculations are performed with high accuracy and require numerical integration, solving systems of linear equations, and calculating polynomial approximations of enthalpies and entropies for 12 gases. The model is characterized by a high computational load, which necessitates its optimization for use in mass computing

tasks, for example, in the construction of a reference database [10].

The software implementation of the model is presented as a separate TModel class that encapsulates all the main computational functions, accompanying arrays of coefficients, and auxiliary data structures. This approach provides logical isolation of the computational part and allows for centralized control of both modeling stages and intermediate variables, including enthalpy and entropy characteristics.

The implementation uses static arrays of fixed size to reduce the cost of memory allocation. At the same time, the implementation of calculating the polynomial value is based on the Horner scheme, which significantly reduces the number of arithmetic operations while maintaining the accuracy of calculations.

The algorithm for solving the Peng-Robinson equation is implemented by solving a cubic equation with real coefficients. The search for the root is carried out in a complex form with the subsequent selection of a physically feasible (positive real) solution. The code for calculating gas volumes uses these roots without additional refinements, which minimizes the overall time spent.

For the convenience of analyzing the architecture of the program code and finding potential performance bottlenecks, Fig. 1 shows a generalized

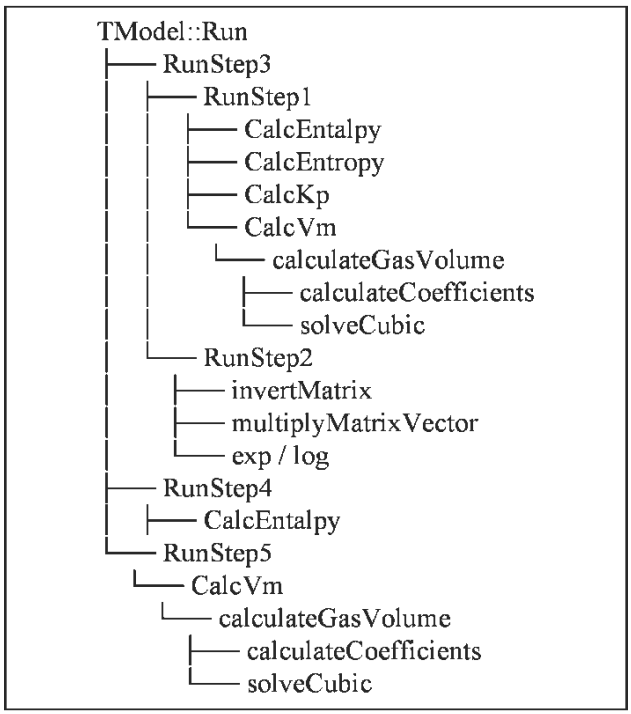


Fig. 1 - The structure of function calls in the TModel

diagram of function calls that are performed within the main cycle of the TModel::Run model. Each of the main stages (RunStep1 - RunStep5) performs specific calculations, some of which are interconnected through common auxiliary functions. The deepest involvement is observed in the RunStep1 function, which includes a chain of calls to calculate Gas Volume, calculate Coefficients, and solve Cubic.

It is important to note that some of the functions are reused in different stages, including Calc Vm, Calc Entalpy, and calculate Gas Volume. This level of modularity creates prerequisites for potential refactoring, local optimization, or transferring part of the calculations to a multi-threaded mode, which is the subject of further research.

4 Analysis of profiling results

Performance profiling in tasks related to numerical modeling often faces a specific problem: a significant portion of the program execution time is spent not on computational procedures, but on auxiliary calls related to environment initialization, event processing, input/output operations, etc. In such cases, the profiling results may be shifted towards system functions and not reflect the real load structure within the framework of the calculation algorithms.

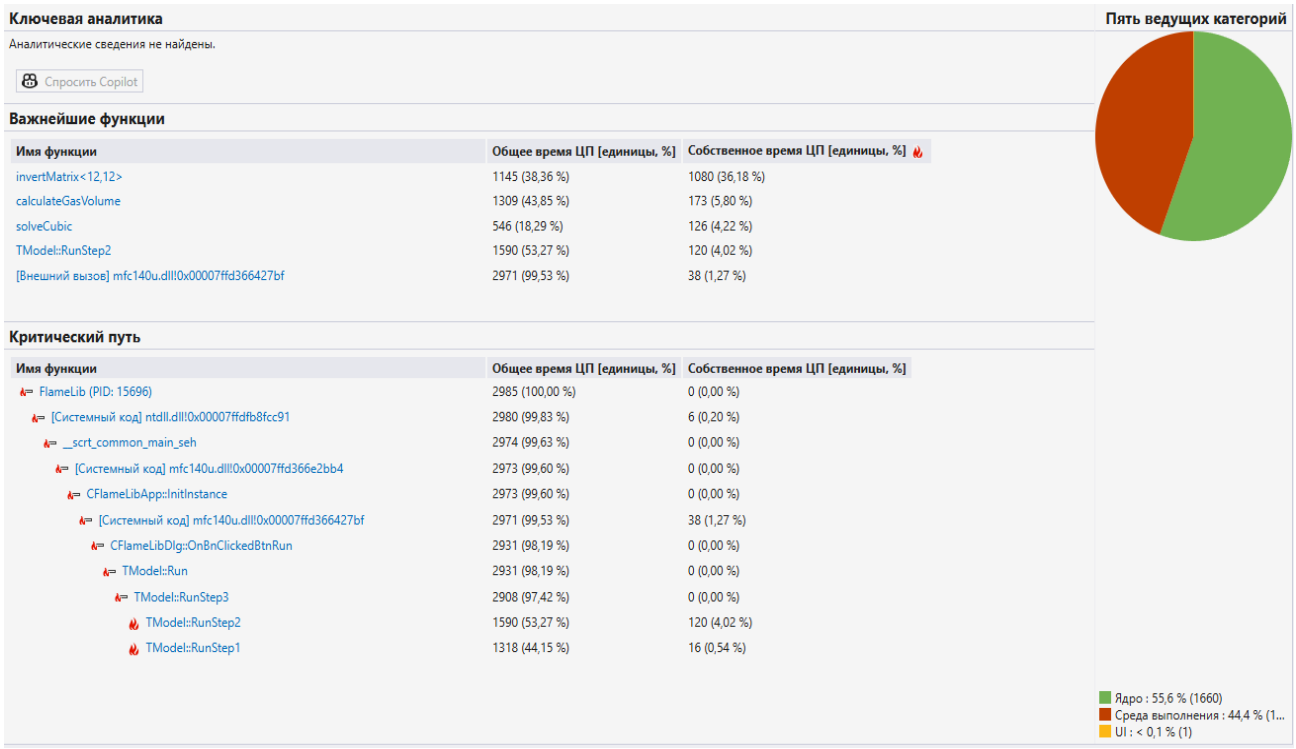
In the proposed implementation, a similar situation was manifested by the fact that in the conditions of a single call to the Run() calculation function, a significant part of the time was taken up by external and infrastructure calls, in particular those related to the MFC and vcruntime libraries. Thus, the standard performance profile displayed information mainly about functions that are not directly related to the mathematical essence of the model.

Another approach to measuring performance is to use timers directly, for example, through std: chrono. However, this method only allows you to record the total execution time of a function or code snippet, without allowing you to differentiate between individual stages of a complex multi-step algorithm. It is theoretically possible to manually insert time measurements around each block, but this requires significant time spent on modifying and maintaining the code.

As follows, a compromise decision was made: to run the computational model in a cycle with a fixed number of repetitions (in this case, 1000 iterations), keeping all parameters unchanged. This approach made it possible to shift the emphasis in the performance profile directly to the model's computational functions, such as Run Step, invert Matrix, solve Cubic, and calculate Gas Volume. This, in turn, created the conditions for a correct analysis of the time spent on critical calculation steps.

Analysis of the profiling results (see Fig. 2) allows us to make a number of important observations.

The largest time consumption is concentrated in the functions related to solving systems of equations (in particular, multiply Matrix Vector), calculating logarithms and exponents (log, exp) and constructing the inverse matrix (invert Matrix). This indicates that the most significant part of the time is spent on iterative refinement of the composition of combustion products, which is implemented within RunStep2.



Важнейшие функции

Имя функции	Общее время ЦП [единицы, %]	Собственное время ЦП [единицы, %]
invertMatrix<12,12>	1145 (38,36 %)	1080 (36,18 %)
calculateGasVolume	1309 (43,85 %)	173 (5,80 %)
solveCubic	546 (18,29 %)	126 (4,22 %)
TModel::RunStep2	1590 (53,27 %)	120 (4,02 %)
[Внешний вызов] mfc140u.dll!0x00007ffd366427bf	2971 (99,53 %)	38 (1,27 %)

Критический путь

Имя функции	Общее время ЦП [единицы, %]	Собственное время ЦП [единицы, %]
FlameLib (PID: 15696)	2985 (100,00 %)	0 (0,00 %)
[Системный код] ntdll.dll!0x00007ffdfb8fcc91	2980 (99,83 %)	6 (0,20 %)
__scrt_common_main_seh	2974 (99,63 %)	0 (0,00 %)
[Системный код] mfc140u.dll!0x00007ffd366e2bb4	2973 (99,60 %)	0 (0,00 %)
CFlameLibApp::InitInstance	2973 (99,60 %)	0 (0,00 %)
[Системный код] mfc140u.dll!0x00007ffd366427bf	2971 (99,53 %)	38 (1,27 %)
CFlameLibDlg::OnBnClickedBtnRun	2931 (98,19 %)	0 (0,00 %)
TModel::Run	2931 (98,19 %)	0 (0,00 %)
TModel::RunStep3	2908 (97,42 %)	0 (0,00 %)
TModel::RunStep2	1590 (53,27 %)	120 (4,02 %)
TModel::RunStep1	1318 (44,15 %)	16 (0,54 %)

Fig. 2 - The result of profiling the source code

In addition, the presence of calculate Gas Volume and its derived functions solve Cubic and calculate Coefficients is noticeable, which are used at several stages of the model (determining volumes when searching for pressure and temperature). Taken together, these calls take up a significant amount of time, especially within RunStep1, RunStep3, and RunStep5, where they are used in repeated cycles to select combustion parameters.

Against this background, other stages of the program, including initialization or recording results, have a negligible impact on the overall execution time. This confirms the correctness of the approach taken with the cyclic launch of the algorithm to shift the focus of profiling to computing components.

The average program execution time is 2892000 μs.

Based on the above analysis, further optimization actions should be aimed at

- improving or rewriting the RunStep2 block, in particular, reducing the number of logarithms and exponents calculations;

- optimization of operations on matrices - invert Matrix and multiply Matrix Vector;

- possible implementation of a more efficient method of solving a cubic equation in solve Cubic;

- checking the feasibility of caching results for the Get Entalpy and Get Entropy functions, which are called with high frequency.

5 Optimizing the computing process

During the initial analysis of the code of the RunStep2() function, two independent fragments were found that did not affect the calculation results but could potentially slow down the execution.

In the first fragment, we found a repeated calculation of exponents. After the main iteration process was completed, the exp (New Value[i]) values for each component were re-calculated and written to the exp New Value array. From a technical point of view, this is a duplication, since the exponent values have already been used in the iteration process. Nevertheless, this part of the code is executed only once after the iterations are completed, so its optimization is cosmetic and does not affect performance. The profiling showed a load reduction of less than 0.1%, which is practically not noticeable.

The next code snippet concerned the initialization block of the matrix A, which defines the structure of chemical equations. Some of its elements are filled with constant values that do not change during iterations. Moving this initialization outside the loop made it possible to eliminate unnecessary repetitive memory operations. In combination with the previ-

ous modification, this has already yielded a noticeable result: the total execution time of the RunStep2() function has decreased by 0.87%, and the average execution time of the entire program has decreased from about 2890000 μs to 2888000 μs.

On the one hand, such a gain may seem insignificant. However, it is worth emphasizing that in this case we are dealing with optimization at the most local level, where the goal is to eliminate any redundant actions, even if their effect is limited to a tenth of a percent. In practice, such micro-optimization may be perceived as excessive attention to detail. However, it is precisely this high density of local optimization that allows you to squeeze the maximum possible out of the model at the current stage without changing its overall structure.

One of the minor but systemically justified improvements was the replacement of the division by multiplication operation with the inverse value in the invert Matrix function. In the original version, each element of the current row was normalized by dividing by the diagonal element. It was proposed to calculate the inverse diagonal element once and replace all subsequent divisions with multiplications:

```
double invDiag = 1.0 / diagElement;
for (int j = 0; j < 2 * N; ++j) {
   augmented[i][j] *= invDiag;
}
```

This resulted in a slight but steady improvement in execution time - from 2888000 μs to 2878000 μs. The effect is typical of micro-optimizations: a slight reduction in floating point overhead, but storing the code in a more efficient form.

Further analysis revealed that the invert Matrix function is executed at each iteration of the chemical equilibrium search and therefore may affect the overall performance. A number of ideas for its possible optimization were considered.

Refusal to use std::swap in favor of manual implementation of the exchange of elements in three cycles. This approach, as the experiment showed, did not pay off: the program execution time increased by about 1%. The reason is that std::swap is usually implemented through a single machine instruction xchg, while the manual implementation generates more memory accesses and instructions. This result was expected.

Replacing std::fabs with std::abs in the nullity check of the leading element. The change was more of a stylistic one, but according to the profiling results, it gave a weak but fixed improvement (about 0.1%). This may be due to differences in the internal implementation of functions or the compiler's reaction to the argument type. It is worth noting that from the point of view of portability and compliance with the modern C++ standard, the use of std::abs for double is preferable.

Replacing division by multiplication with the inverse value (1.0 / x). Although theoretically such a replacement has the potential for optimization, in this case it did not give a noticeable gain. The reason is that modern compilers (e.g., GCC, Clang, MSVC) already automatically apply the appropriate transformation during optimization if the expression does not contain a risk of division by zero. Thus, manual replacement did not bring any additional benefits.

These results emphasize the peculiarity of the case under study: we are dealing with a highly loaded but small-sized (12×12 matrix dimension) code snippet where the compiler performs most micro-optimizations automatically. Under such conditions, manual optimization produces either a cosmetic effect or a slight increase that is quickly offset by the overall structure of the computation.

However, even such an analysis is important because it allows you to

- confirm the expediency of trusting the compiler optimizer,
- separate "aesthetic" code improvements from those that affect performance
- form an understanding of the limits of each approach's effectiveness.

In our case, this also once again confirms the nature of the task: we are trying to achieve maximum efficiency even at the level of tenths of a percent of performance, which is typical for optimizing specialized computational models in the field of technical thermochemistry.

By analyzing the critical path obtained as a result of profiling (Fig. 2), we found that the most resource-intensive function is RunStep3. It is this function that initiates the iterative process of selecting the flame temperature with numerous calls to auxiliary computational functions that form the core of the model. With this in mind, the first optimization attempt was focused on this particular fragment.

In the process of optimizing the RunStep3 function, we tried to implement a multi-threaded calculation of the sum of elements of the form Sum += exp New Value[i] * Entalpy Flame[i], where i runs only through the combustion products.

The initial implementation used a separate system of product identifiers, different from gas identifiers. As a result, calculations looked like sequential calls with fixed indices, for example:

```
Sum += exp New Value [pCO2] * Entalpy Flame [CO2];
Sum += exp New Value [pCO] * Entalpy Flame [CO];
// ...
```

This structure did not allow us to directly rewrite the calculation as a loop suitable for parallelization. Several solutions were proposed: introduce a map of correspondences, create an alternative list of identifiers, or completely unify the identification of products and gases. In the end, the latter option was chosen - abandoning the dual identification system in favor of single gas indices, which made it easier to access the arrays and opened the way for vectorization and multithreaded execution.

The refactoring process also changed the approach to storing the coefficients of polynomials for calculating enthalpy and entropy: instead of a two-dimensional array of double Koefs [19], [7], an array of structures of type Polynomial was introduced, each of which stores its own set of coefficients. This modification made it possible to: simplify data transfer to functions (const Polynomial&); reduce the likelihood of cache conflicts during parallel access; and increase caching efficiency due to compact access to objects.

As a result, it became possible to implement the calculation of the sum in a parallel loop using the #pragma omp parallel for reduction(+:Sum), which fully meets the requirements for correct multithreaded code.

However, experimental testing has shown that this variant, despite its algorithmic correctness, demonstrates an average of 1 ms worse result than the optimized serial code. The likely reason for this is the reduction of data locality, additional synchronization costs, and insufficient computation per iteration, which does not compensate for the overhead of thread management.

Instead, the very fact of refactoring, namely the refusal to duplicate identifiers, brought a steady increase in performance: the average execution time decreased by about 20000 μs within a cycle of 1000 repetitions, which corresponds to a gain of about 20 μs per calculation. This result proved that unifying the data structure is worthwhile, regardless of whether multithreading is used directly.

The next step in the analysis was to study the RunStep2 function, which, according to the profiling results, is also part of the critical path (Fig. 2). The algorithm of this function is responsible for constructing and solving a system of equations to adjust the mole amounts of reaction products. A preliminary analysis of the code did not reveal any fundamental opportunities to simplify or reduce the number of calculations without changing the essence of the algorithm.

It has been suggested that the use of an alternative method for solving a system of linear equations, such as the LU expansion, may provide a certain performance gain. Theoretically, this method allows solving the system of equations faster, especially in cases where the same coefficient matrix with different right-hand sides is reused. However, in our case, each matrix is constructed anew at each iteration, and its size remains small (12×12), which does not allow us to effectively use the advantages of the LU decomposition.

During the testing, it was found that the proposed implementation of the LU decomposition, despite the correctness of its operation, almost doubles the total model execution time: from about 2.89 seconds to more than 4.8 seconds. This result is quite expected, given that the original Gauss-Jordan algorithm is implemented as a compact cyclic code without unnecessary function calls and with a high degree of memory access locality.

Thus, the experimental comparison confirmed that for a problem with a fixed dimension and a single use of the system of equations, the classical Gauss-Jordan method remains optimal in terms of performance. Optimization of this part of the code at the algorithm level turned out to be impractical.

Further analysis has shown that the RunStep2 function contains several computational cycles that at first glance may look like candidates for parallel execution. In particular, these are the calculation of logarithms, the accumulation of the mole sum of substances, and other operations on arrays. However, all attempts to apply multithreading to these code fragments resulted in an increase in execution time, not a decrease.

The reason for this effect is the low complexity of the operations themselves, which are performed in each iteration of the loop. For example, adding or taking the logarithm of a single value does not create enough load to justify the cost of creating, coordinating, and synchronizing threads. In such circumstances, the overhead of parallelization exceeds the computational benefit.

This result once again confirms that the effectiveness of multithreading is closely related not only to the independence of computations, but also to their volume and "cost" in terms of CPU time.

Thus, further use of parallel computing in the RunStep2 function is inappropriate, and optimization efforts should be directed to other parts of the algorithm.

Continuing the analysis of bottlenecks based on the profiling results, the RunStep1 function was the next on the critical path. At first glance, its structure does not leave room for significant optimization: the basic logic is reduced to iterative pressure selection based on the calculation of the total volume of the gas mixture, and all mathematical operations are simple and performed in a fixed order.

However, a close examination of the calls made within RunStep1 revealed potential for improvement. In particular, the Calc Vm function, which is responsible for calculating the reduced mole volumes of components, contains several places where general mathematical functions are used. It was in this direction that we decided to carry out further optimization,

In the code for calculating the parameters in the Peng-Robinson equation, we found several calls to the pow(x, 0.5) function to find the square root. In the standard C++ library, this call is generic and less efficient than the specialized sqrt (x) function. After replacing all calls to pow (x, 0.5) with sqrt (x) in the calculate Gas Volume function, we obtained a significant improvement in execution time - from 2878000 μs to 2590000 μs.

This confirms that even in small mathematical expressions, the choice of the appropriate function makes a difference when multiple calls are made in loops.

As part of a further performance improvement, we introduced a pre-calculation of all temperature-dependent coefficients in the calculate Gas Volume function. This made it possible to avoid repeated calculations within one iteration and reduce the total number of calls to multiplication and division operations.

As a result of this improvement, the average program execution time was reduced by 20,000 μs. The gain is small, but in total, together with the previous optimizations, it creates a significant effect.

By the time these optimizations were completed, the average program execution time was 2570000 μs, which demonstrates a significant speedup compared to the initial value (2888000 μs). The gain of more than 11% was achieved solely through local changes without modifying the main algorithm.

In the process of step-by-step code improvement, it was found that the calculate Gas Volume function responsible for solving the Peng-Robinson cubic equation contains two additional calls - calculate Coefficients and solve Cubic. Both functions were implemented in compliance with the principles of structured programming: separating auxiliary calculations into sub functions, using structures to pass parameters and results, etc.

However, in terms of performance, this implementation was accompanied by excessive costs for function calls, passing arguments, and processing complex numbers. Given that this function is called for each component of the mixture at each iteration, it was decided to simplify its structure. All the necessary calculations - both determining the coefficients and finding the physically correct root of the cubic equation - were transferred directly to the body of calculate Gas Volume.

In addition, it was realized that the use of complex arithmetic when solving a cubic equation significantly complicates the calculations and slows down the program execution. Given that in the context of a physical problem, we are only interested in the real and positive root of the equation, processing complex solutions turned out to be redundant. Instead, we implemented a simplified algorithm for finding real roots, followed by checking for physical correctness, which not only reduced the load on the system but also increased the stability and transparency of the calculations.

This approach helped to avoid intermediate structures, minimize the cost of function calls, and reduce the number of objects in memory. According to the test results, the updated version reduced the execution time by ≈140000 μs.

When analyzing the Calc Vm function, which calculates gas volumes using the modified Peng-Robinson equation, the possibility of using multithreading was considered. Given that the calculation of the volume of each gas is independent, it was theoretically possible to execute the corresponding calls to the calculate Gas Volume function in parallel.

However, practical testing using Open MP showed that the total program execution time, on the contrary, increased by about 480000 μs. As in the cases discussed earlier, we faced the same limitations: individual function calls are too short to benefit from parallelization, and the cost of thread management outweighs any potential gain. Thus, the situation with Calc Vm is fully consistent with previous observations about the inefficiency of multithreading for small-scale computing.

The final stage of the study was to test the possibility of speeding up the calculation of enthalpies by parallelizing function calls using the #pragma omp parallel for directive. Despite the formal independence of calculations in a cycle of 19 iterations, the result was negative: the execution time increased by about one second.

This result fits the previously established trend: with a small number of iterations and a simple loop body, the cost of creating and synchronizing threads exceeds the potential gain. Despite the fact that the attempt was unsuccessful, it finally confirmed the limits of the effective use of multithreading within our task - such possibilities should be considered only for operations with significant computational costs or a large number of elements.

6 Final performance analysis and profiling interpretation

From the very beginning, the basic version of the code was executed on average in about 2892000 µs. After implementing a number of algorithmic improvements, rewriting computational blocks, reducing the number of functions with high complexity, and avoiding unnecessary calls to system mathematical libraries, we managed to reduce the average time to 2420000 µs, i.e., a ≈17% reduction in execution time compared to the initial state.

For objective analysis, the summarized results were profiled (Fig. 3). To interpret this data correctly, let's recall the basic concepts:

- CPU Time is the total amount of CPU time spent by a function, including all its internal calls.

- Self CPU Time is the part of the time that a function spends without taking into account calls to other functions.

- Critical Path - a sequence of function calls that forms the longest execution chain. It is this path that should be minimized to achieve speedup.

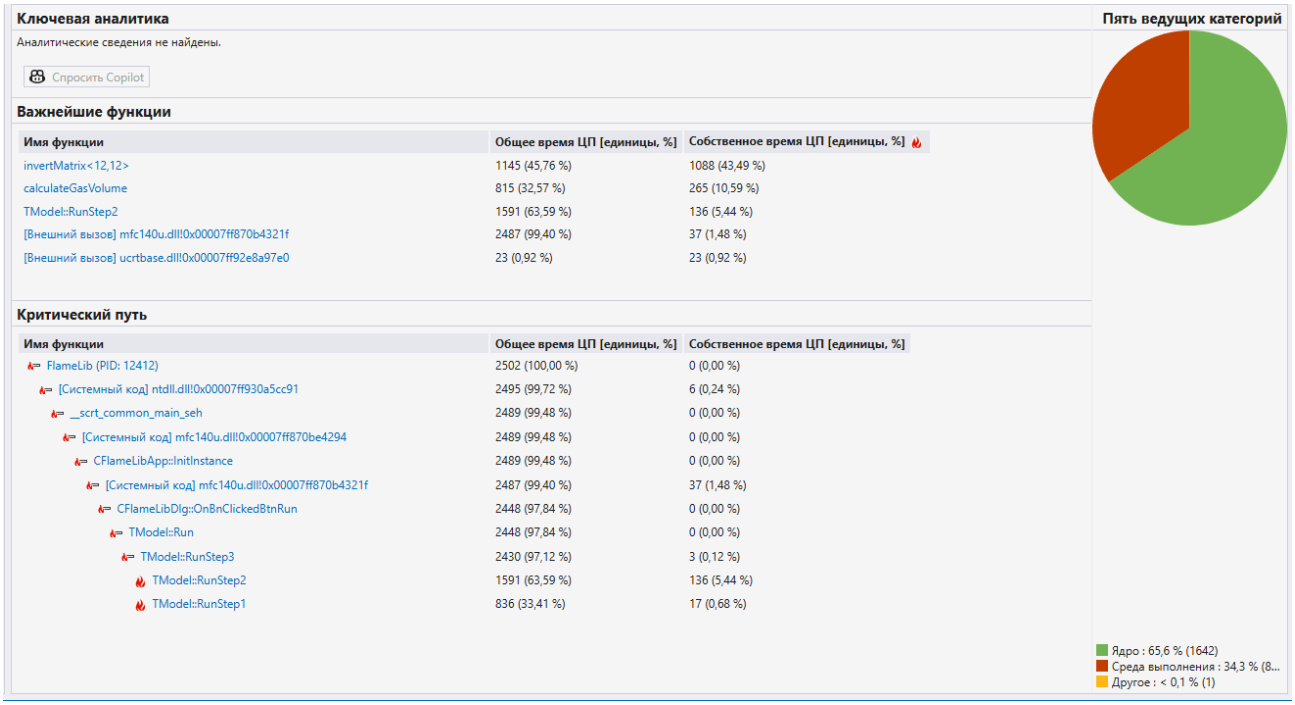


Fig. 3 - The result of code profiling after optimization

- The ideal situation is when most of the heavy computations are concentrated in the native code and the execution time is distributed evenly, without pronounced bottlenecks.

At the time of the final profiling, we can see that:

The main load is concentrated in the RunStep2 function, but its own time is only ≈5.4%, which indicates active delegation of calculations to nested functions.

The two most time-consuming operations - invert Matrix<12,12> (≈43.5%) and calculate Gas Volume (≈10.6%) - remain the main candidates for further analysis in case of need for deeper optimization.

The ratio of kernel load (≈65.6 %) to runtime load (≈34.3 %) indicates an efficient implementation of low-level computing and moderate use of system services.

Despite the fact that most attempts to use multithreading proved to be ineffective due to the small number of iterations in the cycles, the low computational complexity of operations, and the associated overhead, we managed to significantly improve performance through algorithmic transformations, data structure redesign, and reduction of redundant computations.

For profiling purposes, a special mode was used with a 1000-fold repetition of the full simulation, which allowed us to stabilize the results and reliably identify critical areas of the computing process. In this mode, the initial execution time was about 2892000 µs, and after all optimizations were implemented, it was 2420000 µs, which corresponds to about 17% performance improvement.

However, in the target - conditionally "working" - mode, where each calculation is performed only once, the total runtime was reduced from 3.7 ms to 2.4 ms, i.e. by 1.3 ms, which is equivalent to a 36% performance improvement. This result is extremely important in the context of large-scale automated creation of databases of thermochemical

parameters, where millions of such calculations are planned to be performed.

7 Conclusions and practical recommendations

In the course of the study, a comprehensive analysis of the numerical model of thermochemical equilibrium combustion was carried out, taking into account the physical properties of gases and the parameters of the Peng-Robinson model. Despite the initial expectations of a significant gain from multithreading, the results of experimental profiling and testing have shown that the main performance improvement is achieved precisely through algorithmic optimizations, elimination of redundant calculations, rewriting of formulas, local refactoring, and improvement of data cache locality.

The overall effectiveness of the optimization is confirmed by a reduction in the time required to perform a full calculation from 3.7 ms to 2.4 ms, i.e., by more than 36%, which is critical in the context of mass automated creation of thermodynamic parameter libraries. Some fragments of the algorithm, , despite the apparent independence of the calculations, proved to be immune to parallelization due to the too small amount of work in each iteration or the structure of data access, which contradicts the requirements of effective multithreading.

Our results allow us to formulate the following practical recommendations for developers of such computing systems:

Optimization should start with profiling. Without reliable data on critical code sections, any optimization is only a hypothesis.

Multithreading is only advisable for sufficiently large amounts of computation, and requires a preliminary analysis of the potential costs of thread management.

Replacing complex mathematical operations with equivalent but cheaper ones (for example, sqr t(x) instead of pow (x, 0.5)) can give a significant gain.

Eliminating intermediate indexes, unnecessary wrappers, and logical data reorganization not only increases performance but also improves code readability and maintainability.

The combined use of local optimizations, structural changes, and point refactorings can achieve a tangible effect even in cases where each individual improvement seems insignificant.

In general, the experience of implementing optimization within this model demonstrates that it is the combination of empirical measurements, gradual changes, and critical evaluation of the results that allows achieving a balanced and practically meaningful result.

References

1. Goodwin, D. G., Speth, R. L., Moffat, H. K., Weber, B. W. Cantera: An Object-oriented Software Toolkit for Chemical Kinetics, Thermodynamics, and Transport Processes. Version 3.0.0 [Electronic resource] / D. G. Goodwin, R. L. Speth, H. K. Moffat, B. W. Weber. - Zenodo, 2023: https://doi.org/10.5281/zenodo.10483256 - Date of access: 07.05.2025.

2. Kee, R. J., Rupley, F. M., Miller, J. A. CHEMKIN-II: A Fortran Chemical Kinetics Package for the Analysis of Gas-Phase Chemical Kinetics: technical report SAND89-8009 / R. J. Kee, F. M. Rupley, J. A. Miller. - Livermore, California: Sandia National Laboratories, 1989. - 90 p.

3. Kee, R. J., Coltrin, M. E., Glarborg, P. Chemically Reacting Flow: Theory and Practice / R. J. Kee, M. E. Coltrin, P. Glarborg. - Hoboken, NJ: Wiley-Interscience, 2003. - 800 p. - ISBN 978-0-471-40949-2.

4. Kee R.J., Rupley F.M., Miller J.A. *CHEMKIN-II: A Fortran chemical kinetics package for the analysis of gas-phase chemical kinetics*. Report No. SAND89-8009B. Sandia National Labs, Livermore, CA (USA), 1989.

5. Warnatz J., Maas U., Dibble R.W. *Combustion: Physical and Chemical Fundamentals, Modeling and Simulation, Experiments, Pollutant Formation*. Springer, 2006. 2nd ed. 360 p.

6. Brunetkin A. I., Davydov V. O. Determination of the composition of the combustible gas by the method of constraints as a task of model interpretation. 2024. URL: https://opendata.uni-halle.de//handle/1981185920/117684

7. Eriksson, G., Hack, K. ChemApp: A thermochemistry library and its applications. Calphad, 1993, 17(2), 189-205 p.

8. Stock, J. W., Kitzmann, D., Patzer, A. B. C. FastChem 2: An improved computer program to determine the gas-phase chemical equilibrium composition for arbitrary elemental distributions. arXiv preprint arXiv:2206.08247, 2022.

9. Cuadra, A., Huete, C., Vera, M. Combustion Toolbox: An open-source thermochemical code for gas- and condensed-phase problems involving chemical equilibrium. arXiv preprint arXiv:2409.15086, 2024.

10. Brunetkin, O., Sidelnykov, O., Maksymov, M., & Dobrynin, Y. (2025). Improving the model for determining the composition of gunpowder gases during thermal destruction of gunpowder in a limited volume space. Eastern-European Journal of Enterprise Technologies, 3(6 (135), 35–45. https://doi.org/10.15587/1729-4061.2025.330654

Оптимізація моделі розрахунку термохімічної рівноваги продуктів згоряння для побудови бібліотек термодинамічних параметрів

Валентин Давидов, Ольга Тарахтій, Ганна Лисюк
Національний університет «Одеська політехніка»

Анотація. У роботі представлено детальний аналіз алгоритму розрахунку термохімічної рівноваги продуктів згоряння палива в ізольованій системі. Розглянуто послідовність етапів чисельного моделювання, структуру програмної реалізації та проведено профілювання з метою виявлення критичних ділянок коду. На основі отриманих результатів здійснено поетапну оптимізацію, включаючи модифікацію чисельних методів, реорганізацію структури даних, зменшення обсягу обчислень та спроби використання багатопоточності. Показано, що суттєвий приріст продуктивності досягнуто переважно за рахунок алгоритмічних рішень. Усі експерименти виконано для сценарію багаторазового повторення розрахунків, типовому для задач побудови баз даних рівноважних параметрів. За підсумками оптимізації час виконання програми зменшено на понад 40 %, що підтверджує ефективність реалізованих рішень для масового чисельного моделювання.

Ключові слова: термохімічна рівновага, згоряння палива, чисельне моделювання, оптимізація продуктивності, профілювання коду, алгоритмічні вдосконалення, багатопоточність.

Отримано 10.08.2025

Давидов Валентин Олегович, к. т. н., доцент, доцент кафедри програмних і комп'ютерно-інтегрованих технологій, Національний університет «Одеська політехніка»; проспект Шевченка, 1, Одеса, 65044, Україна.
E-mail: davydov@op.edu.ua; тел.: +38 096 787 6331

Valentin Davydov, PhD, Assistant professor, Assistant professor of the Department of Software and Computer-Integration Technology, Odesa Polytechnic National University; 1, Shevchenko Avenue, Odesa, 65044, Ukraine.
E-mail: davydov@op.edu.ua; ph.: +38 096 787 6331

ORCID ID: https://orcid.org/0000-0003-3099-7596

Тарахтій Ольга Сергіївна, к. т. н., доцент кафедри програмних і комп'ютерно-інтегрованих технологій, Національний університет «Одеська політехніка»; проспект Шевченка, 1, Одеса, 65044, Україна.
E-mail: tarakhtij@op.edu.ua; тел.: +38 067 587 4071

Olga S. Tarakhtij, PhD, Assistant professor of the Department of Software and Computer-Integration Technology, Odesa Polytechnic National University; 1, Shevchenko Avenue, Odesa, 65044, Ukraine. E-mail: tarakhtij@op.edu.ua; ph.: +38 067 587 4071

ORCID ID: https://orcid.org/0000-0002-4266-3481

Лисюк Ганна Петрівна, аспірант кафедри програмних та комп'ютерно-інтегрованих технологій, Національний університет «Одеська політехніка»; проспект Шевченка, 1, Одеса, 65044, Україна.
E-mail: lysjuk@op.edu.ua; тел.: +38 067 297 5204

Hanna Lysiuk, PhD Student of the Department of Software and Computer-Integration Technology, Odesa Polytechnic National University; 1, Shevchenko Avenue, Odesa, 65044, Ukraine. E-mail: lysjuk@op.edu.ua; ph.: +38 067 297 5204

ORCID ID: https://orcid.org/0000-0002-0226-4160