GENERAL POSSIBLE OPTIMIZATIONS IN C:

1. Loop unrolling
2. Avoid calculations in loop - **Most compilers are good at identifying** loop invariant computations. //makes it a must-implement case in an automated source code optimization tool then?
3. Avoid pointer dereference in loop - Better assign it to some temporary variable and use the temporary variable in the loop.
4. Use Register variables as counters of inner loops - Variables stored in registers can be accessed much faster than variables stored in memory.
5. Loop jamming - combine adjacent loops which loop over the same range of the same variable
6. Loop inversion - Assuming the loop is insensitive to direction. Some machines have a special instruction for decrement and compare with 0.
7. Mathematical optimizations
    a. Avoid integer division where possible because it is slow
    b. Use shift operators instead of * and / for powers of 2
    c. Strength reduction - Use an alternative cheaper operator than a costly one where possible
8. Switch instead of if-else when possible, especially when case labels can be contiguous because that creates a jump table which is fast.
9. Prefer int to char or short - C performs all operations of char with an integer. In all operations like passing char to a function or an arithmetic operation, the first char will be converted into integer and after compilation of operation, it will again be converted into char. For a single char, this may not affect the efficiency but suppose the same operation is performed 100 times in a loop then it can decrease the efficiency of the program. //is this useful info to us?
10. Short-circuit evaluation
11. Dead code evaluation

https://www.geeksforgeeks.org/basic-code-optimizations-in-c/
http://icps.u-strasbg.fr/~bastoul/local_copies/lee.html
Detailed types and explanations in https://en.wikipedia.org/wiki/Optimizing_compiler

COMPILER OPTIMIZATION FLAGS:
https://www.keil.com/support/man/docs/armclang_intro/armclang_intro_fnb1472741490155.htm

COMPARISON OF COMPILERS:
The most popular compilers for C are GCC (from GNU), Clang and MSVC.

GCC : Free software. It is an official compiler for the GNU and Linux systems, and a main compiler for compiling and creating other UNIX operating systems.

Clang: Low-level Virtual Machine (LLVM) contains a series of modularized compiler components and tool chains. It can optimize program languages and links during compilation, runtime, and idle time and generate code. LLVM can serve as a background for compilers in multiple languages. Clang is mainly used to provide performance superior to that of GCC.

MVSC: Closed, proprietary .

Comparison in terms of error handling (highlighting the pros of each):

GCC	Clang	MSVC
With the -O2 flag, Out of bounds error can be detected.	Cannot detect out of bounds even with -Warray-bounds or /Wall.	Cannot detect out of bounds even with -Warray-bounds or /Wall.
Doesn't detect the error.	Reports error for code that attempts to perform string concatenation using + operator.	Doesn't detect the error.
Unclear errors.	Unclear errors.	In case of if-else statements without braces for multi-lined body of if, and in the case of a returning function with a missing return statement, the errors are clear.

Comparison in terms of compilation process:

GCC	Clang
Compilation process is as follows: read the source file -> preprocess the source file -> convert it into an IR -> optimize -> generate an assembly file -> the assembler generates an object file.	The process of generating assembly files is omitted in the process of generating object files. The object file is generated directly from the IR.
GCC IR is not so concise.	The data structure of LLVM IR is more concise => occupies less memory during compilation, supports faster traversal. Clang and LLVM are advantageous in terms of the compilation time.

	Clang reduces the single-thread compilation time by 5% to 10% compared with GCC. Therefore, Clang offers more advantages for the construction of large projects.

Comparison in terms of execution performance:

GCC	Clang
Always advantageous in terms of performance optimization. On a code with no hotspots, 1% to 4% performance advantage over Clang and LLVM for most programs at the O2 and O3 levels.	Clang and LLVM are conservative in loop optimization and thus not advantageous in performance.
Growth not as rapid as Clang.	For AI-related programs, benchmarking showed a 3% better performance compared to GCC. It is fast improving.
GCC optimizes the vectors at the O3 level. Except for vectorized programs, GCC does not greatly improve the performance at the O3 level compared with that at the O2 level.	Clang and LLVM optimize the vectors at the O2 level, so benchmarking has shown it to perform better when hotspots involve vectorized regions.

https://alibabatech.medium.com/gcc-vs-clang-llvm-an-in-depth-comparison-of-c-c-compilers-899ede2be378
https://easyaspi314.github.io/gcc-vs-clang.html

REFERENCE PAPERS:

1: Click, Cliff & Cooper, Keith. (2000). Combining Analyses, Combining Optimizations. ACM Transactions on Programming Languages and Systems. 17. 10.1145/201059.201061.

- Global analysis: One obvious method for improving the translated code is to look for code fragments with common patterns and replace them with more efficient code fragments. These are called peephole optimizations. However, peephole optimizations are limited by their local view of the code. A stronger method involves global analysis, in which the compiler first inspects the entire program before making changes.

- Data-flow analysis: A common global analysis is called data-flow analysis, in which the compiler studies the flow of data around a program.

- Top-down and bottom-up: Central to most data-flow frameworks is the concept of a lattice. Analysis can be from top to bottom in the lattice or bottom to top.

- Multi-pass: Modern optimizing compilers make several passes over a program's intermediate representation in order to generate good code. Many of these optimizations exhibit a phase ordering problem. The compiler discovers different facts (and generates different code) depending on the order in which it executes the optimizations. Getting the best code requires iterating several optimizations until reaching a fixed point. This thesis shows that by combining optimization passes, the compiler can discover more facts about the program, providing more opportunities for optimization.

- Most constants and common subexpressions are easy to find; they can be found using a simple peephole analysis technique. This peephole analysis was done at parse-time lowering total compilation time by over 6%. To do strong peephole analysis at parse-time requires SSA form and use-def chains. We implemented a parser that builds SSA form and use-def chains incrementally, while parsing.

2: Youenn Lebras. Code optimization based on source to source transformations using profile guided metrics. Automatic Control Engineering. Université Paris-Saclay, 2019. English. ffNNT : 2019SACLV037ff. fftel-02443231f

- Common compiler-optimization techniques:
    - Data-flow analysis: gathers information about the possible set of values calculated at various points in a computer program. A program control flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate.
    - Partial evaluation, dead code elimination and common sub-expression elimination, to reduce code size.
    - Inline expansion
    - Instruction scheduling
    - Common loop optimizations include interchange, splitting, unrolling, etc.
    - Automatic parallelization by converting sequential code into multi-threaded or vectorized code in order to utilize multiple processors simultaneously
    - Strength reduction

- A lot of existing tools allow to perform source-to-source transformation, most of them focus on optimizing loops using the polyhedral model. The polyhedral method treats each loop iteration within nested loops as lattice points inside mathematical objects called the polyhedral. It performs affine transformations or more general non-affine transformations such as tiling on the polytopes, and then converts the transformed polytopes into equivalent, but optimized, loop nests through polyhedral scanning. The polyhedral model allows for good performance on loops that can be handled and is especially used for the parallelism, but it can be applied on a small set of loops.

- Pg 29-34 //check where we stand

SIMILAR EXISTING TOOLS:

- Compiler Explorer is an open source software/website that can compile and show the assembly created by a wide variety of compilers, platforms and settings.
  Someone asked for the exact tool we are building and it doesn't exist

  Some static code analysis tools:

- A set of diagnostics is implemented in the static analyzer PVS-Studio that enable you to find some situations where code can be optimized. However, PVS-Studio as any other static analyzer cannot serve as a replacement of the profiling tools.
  Released on the Internet on 31st December 2006. Latest version is 6.27 released on December 3rd, 2018.
  https://www.viva64.com/en/pvs-studio/

- Provides technical debt estimation and code visualization https://www.cppdepend.com/
  Version 1.0.0.1 (Trial Edition) / 1.0.0.1 (Professional Edition) released on September 17th, 2009. Latest version is v2021.1 released on January 17th, 2021

- Helix QAC is a static analysis testing tool for C and C++ code from Perforce (formerly PRQA). The tool comes with a single installer and supports platforms like Windows 7, Linex Rhel 5 and Solaris 10. This gives very clear diagnostics which helps in identifying the root cause and quick defect fixes. It is paid. https://www.perforce.com/products/helix-qac