Performance Optimization

Performance Optimization

- Application performance optimization refers to the process of improving the speed, efficiency, and overall performance of software applications. It involves identifying performance bottlenecks, analyzing application behavior, and applying various techniques to enhance performance.

Performance Optimization

- Application performance optimization refers to the process of improving the speed, efficiency, and overall performance of software applications. It involves identifying performance bottlenecks, analyzing application behavior, and applying various techniques to enhance performance.

- Following are some common techniques used in performance optimizations:
  - Profiling
  - Cache Optimization
  - Algorithmic Optimization
  - Parallel Processing
  - Network and Memory Optimization

Performance Optimization

- **Profiling:**
  Profiling tools help identify sections of code that consume significant resources or cause performance issues. Profilers measure execution times, method calls, memory usage, and other metrics to pinpoint bottlenecks.

Performance Optimization

- **Profiling:**
  Profiling tools help identify sections of code that consume significant resources or cause performance issues. Profilers measure execution times, method calls, memory usage, and other metrics to pinpoint bottlenecks.

- **Caching:**
  Caching involves storing frequently accessed data or computations in memory or a faster storage medium. By avoiding expensive computations or database queries, caching can significantly improve application response times.

Performance Optimization

- **Algorithmic Optimization:**
  Analyzing and optimizing algorithms can lead to substantial performance gains. By choosing more efficient algorithms or improving existing ones, you can reduce the computational complexity and enhance overall application performance.

Performance Optimization

- **Algorithmic Optimization:**
  Analyzing and optimizing algorithms can lead to substantial performance gains. By choosing more efficient algorithms or improving existing ones, you can reduce the computational complexity and enhance overall application performance.

- **Parallel Processing:**
  Leveraging concurrency and parallelism can enhance performance by allowing multiple tasks or threads to execute simultaneously. Techniques like multi-threading, asynchronous programming, and parallel processing can improve application responsiveness and throughput.

Performance Optimization

- **Network Optimization:**
  Optimizing network communications can lead to faster data transfers and reduced latency. Techniques include minimizing round trips, compressing data, using efficient protocols, and implementing caching mechanisms.

Performance Optimization

- **Network Optimization:**

  Optimizing network communications can lead to faster data transfers and reduced latency. Techniques include minimizing round trips, compressing data, using efficient protocols, and implementing caching mechanisms.

- **Memory Optimization:**

  Efficient memory usage is crucial for application performance. Techniques like object pooling, lazy loading, and minimizing memory leaks can improve memory utilization and reduce the frequency of garbage collection, leading to better performance.

Performance Optimization

- These are few common techniques used for optimizations.
  To extend to these, many other techniques are used in practice, such as:
  - Database Optimization
  - Compression and Minification
  - Load Balancing
  - Vectorization
  - Distributed processing
  - and many more...

Performance Optimization

- **Performance Optimization using Compiler Flags:**
  Performance optimization using compiler flags involves utilizing specific compiler options or flags to improve the performance of the compiled code. Compiler flags can influence various aspects of code generation, optimization, and target platform-specific optimizations.

- Following are some commonly used compiler flags for performance optimization:
  Optimization Level Flags, Inlining, Loop Optimization, Vectorization, Compiler specific flags, Memory alignment, Dead-code elimination, Linker Flags and Target Specific Flags.

Performance Optimization

- **Optimization Level Flags:**
  Compilers typically provide multiple optimization levels, such as -O0 (no optimization), -O1 (basic optimizations), -O2, -O3, and -Ofast (aggressive optimizations). Higher optimization levels can result in faster and more efficient code but may also increase compilation time.

Performance Optimization

- **Optimization Level Flags:**

  Compilers typically provide multiple optimization levels, such as -O0 (no optimization), -O1 (basic optimizations), -O2, -O3, and -Ofast (aggressive optimizations). Higher optimization levels can result in faster and more efficient code but may also increase compilation time.

- **Inlining:**

  Inlining is the process of replacing a function call with its actual code. The inline keyword or flags like -finline-functions enable the compiler to decide which functions to inline, potentially reducing the overhead of function calls and improving performance.

Performance Optimization

- **Loop Optimization Flags:**

  Flags like -funroll-loops and -ftree-loop-optimize optimize loops by unrolling them (reducing loop overhead) or applying loop-specific optimizations. These flags can improve loop performance by eliminating unnecessary iterations or reducing branch mispredictions.

Performance Optimization

- **Loop Optimization Flags:**

  Flags like -funroll-loops and -ftree-loop-optimize optimize loops by unrolling them (reducing loop overhead) or applying loop-specific optimizations. These flags can improve loop performance by eliminating unnecessary iterations or reducing branch mispredictions.

- **Vectorization Flags:**

  Vectorization enables the compiler to transform scalar operations into SIMD (Single Instruction, Multiple Data) instructions, which can perform computations on multiple data elements simultaneously. Flags like -msse, -mavx, or -march specify the target architecture for vectorization.

Performance Optimization

- **Compiler-Specific Flags:**
  Different compilers provide their own set of flags for performance optimization. For example, GCC offers flags like -march=native to enable optimizations for the host architecture, while Clang provides flags like -march=target to optimize for a specific target architecture.

Performance Optimization

- **Compiler-Specific Flags:**
  Different compilers provide their own set of flags for performance optimization. For example, GCC offers flags like -march=native to enable optimizations for the host architecture, while Clang provides flags like -march=target to optimize for a specific target architecture.

- **Memory Alignment Flags:**
  Flags like -falign-functions or -falign-loops enforce memory alignment, which can enhance cache performance by ensuring that data is stored in aligned memory locations. This can reduce cache misses and improve overall performance.

Performance Optimization

- **Dead-code Elimination Flags:**
  Flags like -fdead-code-elimination enable the removal of unused or unreachable code during the compilation process. Eliminating dead code reduces executable size and improves performance by eliminating unnecessary computations.

Performance Optimization

- **Dead-code Elimination Flags:**

  Flags like -fdead-code-elimination enable the removal of unused or unreachable code during the compilation process. Eliminating dead code reduces executable size and improves performance by eliminating unnecessary computations.

- **Linker Flags:**

  Linker flags, such as -flto (Link Time Optimization), allow the compiler to optimize code across different translation units during the linking phase. Linker flags can improve performance by enabling more aggressive optimizations that consider the entire program.

Performance Optimization

- **Target Specific Flags:**
  Compilers often provide flags to optimize code specifically for the target platform. These flags may include architecture-specific optimizations, instruction set extensions, or hardware-specific features. Examples include -march, -mtune, or -mcpu flags.

Performance Optimization

- **Target Specific Flags:**
  Compilers often provide flags to optimize code specifically for the target platform. These flags may include architecture-specific optimizations, instruction set extensions, or hardware-specific features. Examples include -march, -mtune, or -mcpu flags.

- It's important to note that the availability and functionality of compiler flags may vary depending on the compiler and programming language used. Additionally, optimizing code using compiler flags may require balancing factors like code readability, portability, and compatibility with the target platform. Experimentation and profiling are often necessary to determine the most effective combination of compiler flags for a specific application.

Workout 1

A) Once a running sample program is created, use the compiler flags -O0 -O1 -O2 -O3 and -Ofast. List the execution timings using the listed flags. Do you observe any difference?

B) Verify the output values using a sum variable for each optimization level used. Do you observe any difference?

C) Experiment with static and dynamic arrays. Does that many any difference?

Write a program to perform Matrix-Matrix multiplication.

Note:
1) Explicitly define a Macro ‘N’ for size of Matrix.
2) Use square matrices for simplicity.
3) Use random values to initialize values of the matrix.
4) Set the value of N as 3 and verify if the output results are correct.
5) Set the value of N as 3000 and note the execution time of the program.

Performance Optimization

- **Performance Optimization using Libraries:**

  Performance optimization using libraries involves utilizing pre-existing software libraries or frameworks that are specifically designed to enhance the performance of certain tasks or provide optimized implementations of common functionality. These libraries offer optimized algorithms, data structures, and low-level optimizations that can improve application performance.

- Following are few libraries used in performance optimizations:

  Intel Math Kernel Library (MKL), Image and Signal processing (OpenCV), Database access, Network and Communication, Graphics and Development, Parallel processing, etc.

Performance Optimization

- In HPC, there exist commonly used libraries such as
  BLAS (Basic Linear Algebra Subprograms),
  LAPACK (Linear Algebra Package), and
  MKL (Math Kernel Library):

Performance Optimization

- In HPC, there exist commonly used libraries such as
  BLAS (Basic Linear Algebra Subprograms),
  LAPACK (Linear Algebra Package), and
  MKL (Math Kernel Library):

- BLAS (Basic Linear Algebra Subprograms):
  BLAS is a standard library that offers highly optimized implementations of common linear algebra operations, including vector dot product, matrix-vector multiplication, matrix-matrix multiplication, and various matrix factorizations.
  BLAS libraries are typically available in different implementations, such as OpenBLAS, Intel MKL, and ATLAS, each offering optimized routines for specific hardware architectures.

Performance Optimization

- In HPC, there exist commonly used libraries such as
  BLAS (Basic Linear Algebra Subprograms),
  LAPACK (Linear Algebra Package), and
  MKL (Math Kernel Library):

- LAPACK (Linear Algebra Package):
  LAPACK is a higher-level library that builds upon BLAS and provides a comprehensive set of linear algebra routines for solving linear equations, eigenvalue problems, and singular value decomposition.
  LAPACK is designed to handle dense and structured matrices and provides both real and complex arithmetic operations.

Performance Optimization

- In HPC, there exist commonly used libraries such as
  BLAS (Basic Linear Algebra Subprograms),
  LAPACK (Linear Algebra Package), and
  MKL (Math Kernel Library):

- MKL (Math Kernel Library):
  MKL is a highly optimized library developed by Intel that provides a broad range of mathematical functions and performance-enhanced implementations of BLAS, LAPACK, FFT (Fast Fourier Transform), and other mathematical routines.
  MKL is specifically designed to leverage the capabilities of Intel processors and architectures, including multi-core CPUs and Intel Xeon Phi processors.

Workout 2

A) Once a running sample program is created, use the compiler flags -O0 -O1 -O2 -O3 and -Ofast. List the execution timings using the listed flags. Do you observe any difference?

B) Verify the output values using a sum variable for each optimization level used. Do you observe any difference?

C) Experiment with static and dynamic arrays. Does that many any difference?

Modify Matrix-Matrix multiplication program for the following

Note:
1) Check which specific function can be used for Matrix-Multiplication from BLAS, LAPACK library.
2) Implement the library call for the specific operation.
3) Implement the library call for MKL version of the function.

“Optimization is like a race car. You spend hours tuning it, shaving off milliseconds, just to get to the finish line and realize you forgot your car keys.”

Questions?

Thanks!

Contact:

Vineet More
vineet.more.bfab@gmail.com
[Make sure to use HPCAP23 as the subject line for your queries]

LinkedIn Handle:
https://www.linkedin.com/in/vineet-more-c-programmer/