Introduction

Optimization options control compile time optimizations to generate an application with code that executes more quickly. Absoft Fortran 90/95 is an advanced optimizing compiler. Various optimizers can be turned on that discover different opportunities to optimize Fortran code. There are pros and cons when choosing optimizations; the application will execute much faster after compilation but the compilation speed itself will be slow. Some of the optimizations described below will benefit almost any Fortran code, while others should only be applied to specific situations.

No Optimization

The **–O0** option is the default optimization level and provides the fastest compilation speed. It disables all optimizations and is useful for debugging. The **-g** option is the common debugging flag used with this level of optimization.

Basic Optimizations

The **–O1** option will cause most code to run faster and enables optimizations that only span sequential code sequences. The optimizations include, but are not limited to, common subexpression elimination, constant propagation, dead code elimination, and instruction scheduling. This option is compatible with debugging options.

Normal Optimizations

The **–O2** option enables normal optimizers that can make most code run faster. It can substantially rearrange the code generated for a program. The optimizations include, but not limited to, strength reduction, partial redundancy elimination, innermost loop unrolling, control flow optimizations, advanced instruction scheduling. This option is not generally usable with debugging options.

Advanced Optimizations

The **–O3** option enables advanced optimizers that can significantly rearrange and modify the code generated for a program. It provides all optimizations applied with **–O1** and **–O2**. Additional optimizations include auto loop vectorization, loop permutation (loop reordering), loop tiling (improved cache performance), loop skewing, loop reversal, loop fusion and fission, unimodular transformations, forward substitution, and expression simplification. This option is not usable with debugging options.

Inter-Procedural Optimizations

The **-O4** or **-Ofast** option enables advanced optimizers that can significantly rearrange and modify the code generated for a program. The optimizations include all optimizations that are included with –O3 as well as turning on inter-procedural analysis with inlining optimizers.

Auto-Parallelization

The **–apo** option enables automatic parallelization of those loops in your source program to leverage multiple cores or processors.

SSE Optimizations

These optimizations leverage the *Streaming SIMD Extension* (SSE) instruction set of the CPU. These instructions were first introduced with the Intel Pentium 4 as part of the SIMD (single instruction, multiple data) processor supplementary instruction sets.

SSE2 instructions

The **–msse2** and **–mno-sse2** options enable and disable respectively the use of SSE2 instructions for floating-point operations. This **–msse2** option is automatically enabled on processors which support SSE2. It may be disabled with the **–mno-sse2** option.

SSE3 instructions

The **-msse3** option enables the use of SSE3 instructions for floating-point operations. This option is automatically turned on when the **-march=host** option is specified and the host supports SSE3 instructions.

SSE4a instructions

The **-msse4a** option enables the use of SSE4a instructions. This option is automatically turned on when the **-march=host** option is specified and the host supports SSE4a instructions.

SSSE4.1 instructions

The **-msse4.1** option enables the use of SSE4.1 instructions. This option is automatically turned on when the **-march=host** option is specified and the host supports SSSE4.1 instructions

Math Optimization Level

The **-speed_math=*n*** option enables aggressive math optimizations that may improve performance at the expense of accuracy. Valid arguments for ***n*** are 0-11. The following table describes the effect of each level:

n	effect
0	enable wrap around optimization
1	allow relational operator folding; may cause signed integer overflow
2	enable partial redundancy elimination for loads and stores
3	enable memory optimization for functions without aliased arrays
4	inline NINT and related intrinsics with limited-domain algorithm use fast_powf in libm instead of powf
5	use multiplication and square root for exp() where faster
6	allow optimizations that reassociate floating point operators
7	*see notes below*
8	allow use of reciprocal instruction; convert a/b to a*(1/b)
9	use fast algorithms with limited domains for complex norm and divide use x*rsqrt(x) for sqrt(x) on machines where faster dead casgn function elimination
10	use AMD ACML library if applicable
11	allow relational operator folding; may cause unsigned integer overflow use IEEE rounding instead of Fortran rounding for NINT intrinsics use IEEE rounding instead of Fortran rounding for ANINT intrinsics

NOTES:

A. Departure from strict rounding is applied at 3 levels: level 1 is applied at ***n=5***, level 2 is applied at ***n=7***, and level 3 is applied at ***n=10***.
B. Conformance to IEEE-754 arithmetic rules is relaxed at 2 levels: level 1 is applied at ***n=6***, level 2 is applied at ***n=10***
C. At ***n=10***, the loop unrolling constraints are modified: loop size is increased to 7000, limit is increased to 9, minimum iteration is decreased to 200.

Enable OpenMP Directives

The **-openmp** option enables the recognition of OpenMP directives. OpenMP directives usually begin in column one in the form of:

`C$OMP` for fixed source format
`!$OMP` for free source format

Please refer OpenMP Specification for the detail.

OpenMP optimization Level

The **–speed_openmp=*n*** enables progressively more aggressive OpenMP optimizations based on the value of ***n*** as follows:

n	effect
0	allow code optimization and movement through OpenMP Barrier
1	enable loose memory equivalence algorithm during optimization
2	Enable MU generation in SSA generation for OpenMP pragma
3	Enable CHI generation in SSA generation for OpenMP pragma
4	Allow loop unrolling for loops with OpenMP chunksize directive
5	Use a risky but faster algorithm to handle thread private common blocks

Each level includes all previous optimizations (e.g. **3** includes 0,1, and 2).

Safe Floating-Point

The **–safefp** option is used to disable optimizations that may produce inaccurate or invalid floating point results in numerically sensitive codes. The effect of this option is to preserve the FPU control word, enable NAN checks, disable `CABS` inlining, and disable floating-point register variables.

Optimization Diagnostics

These options do not perform optimizations; rather they are used to provide diagnostic information on which optimizations were performed. Equally importantly, they provide diagnostic information on which optimizations were not performed and why not.

Report Parallelization Results

The **–LNO:verbose=on** option is used to display the results of the –**apo** option. It will report which loops were parallelized and which were not and why not.

Report Vectorization Results

The **–LNO:simd_verbose=on** option is used to display the results of vectorization of loops which occurs at optimization levels greater than **–O3**. It will report which loops were vectorized and which were not and why not.

Loop Nest Optimizations

These options are used to control the loop nest optimizer in the compiler to deliver the best performance for a specific code. Usually, the default level is designed to achieve performance improvement for most situations. The aggressive level may achieve additional speed improvements, but it may also produce performance degradation. Thus, a set of options for individual optimizations is given to professional tuners for pursuing the best performance improvement. For loop nest optimizations, the options starts with **-LNO:**.

Loop Vectorization (SIMD Vectorization)

The **–LNO:simd=n** option is used to control loop vectorization and utilizes *Streaming SIMD Extension* (SSE) instruction sets provided by certain processors to work on multiple pieces of data at once. **n=0** disables loop vectorization. **n=1** is the default vectorization. **n=2** enables aggressive vectorization. When **–O3** is used, **-LNO:simd=1** is implied.

Loop Tiling/Blocking

The **–CG:blocking=[on|off]** option is used to enable/disable the loop tiling optimization, that transforms the loop space in order to significant improve cache locality.

Loop Fusion and Fission

The **–LNO:fusion=n** option is used to control loop fusion which merges two small consecutive loops into one larger loop in order to improve utilization of CPU resources. **n=0** disables fusion. **n=1** is the default level. **n=2** is the aggressive level. When **–O3** is used, **–LNO:fusion=1** is implied.

The **–LNO:fission=n** option is used to control loop fission which splits one large loop into two smaller loops to reduce excessive register usage. **n=0** disables fission. **n=1** is the default level. **n=2** is the aggressive level. When the **–O3** is used**–LNO:fission=2** is implied.

Note: When both fusion and fission are performed on a same loop, fusion has a higher priority than fission.

The option **–LNO:fusion_max_size=n,** where **n**=0-99999, can be used to prevent fusion of loops whose size is greater than **n**.

Unroll-and-Jam

The option **–LNO:fu=n**, where **n**=0-100, is used to control unroll-and-jam on the innermost loop, which completely unrolls the innermost loop body to a straight line of code if the iteration is more than the limit **n**. The enlarged loop body may improve utilization of CPU resources. However, if the loop body is too large, performance may degrade due to register pressure.

The option **–LNO:full_unroll_size=n**, where **n**=0-10000, is used to limit unroll-and-jam of the innermost loop. Unrolling is disabled if the size of the innermost loop body is greater than the limit **n**.

The option **–LNO:ou=n**, where **n**=1-32 is used to set the number of iterations that the outer loop is unrolled to the limit **n**.

Prefetch

The **–LNO:prefetch=n** option is used to control how the compiler generates prefetch instructions for memory loads and stores to improve cache locality. **n=0** disables prefetch generation. **n=1** is the default level. **n=2** is the aggressive level.

Code Generation Optimizations

The code generator contains its own optimizer and various options are designed to fine tune those optimizations. These options are prefixed with **–CG**:.

Global Code Motion

The **–CG:gcm=[on|off]** option is used to enable/disable the instruction level global code motion optimization. This optimization is designed to improve scheduling efficiency.

Instruction Level Loop Optimization

The **–CG:loop_opt=[on|off]** option is used to enable/disable the instruction level loop optimization, especially loop unrolling. Instruction level loop unrolling may make some OpenMP programs fail due to changes in iteration counts.

Peephole Optimizations

The **–CG:peephole_optimize=[on|off]** option is used to enable/disable the instruction level peephole optimizations. Peephole optimizations include many small optimizations to improve the efficiency of instructions generated by code generator.

Control Flow Optimization

The **–CG:cflow=[on|off]** option is used to enable/disable instruction level control flow optimization. This optimization rearranges the order of basic blocks to improve performance on critical paths.

Hyperblock Scheduling

The **–CG:hb_sched=[on|off]** option is used to enable/disable the hyperblock scheduling. Hyperblock scheduling is an advanced scheduling technique to schedule code based on a hyperblock instead of a super block or a basic block.

Inter-Procedural Analysis and Optimization

Inter-procedural analysis collects information from every piece of a program and analyzes their relationships in order to optimize for better performance. The options to control inter-procedural analysis start are prefixed with **–IPA:**.

Inlining

The **–IPA:inline=[on|off]** option is used to enable/disable the inliner. This optimization improves performance by removing those calls to small functions and incorporating them inline at the point of reference.

The option **–IPA:plimit=n**, where **n=**0-INT_MAX**,** is used to limit the inliner to a specific number of calls and basic blocks inside the function under consideration.

Cloning

The option **–IPA:multi_clone=n**, where **n=**0-INT_MAX**,** is used to control the maximum clones per call graph node to the upper bound **n**. When **n** is 0, cloning is disabled.

Inter-Procedural Structure Optimization

The **–IPA:optimize_struct=[0|1]** option is used to enable/disable the inter-procedural struct optimization. If set to 0, this optimization is disabled. If set to 1, this optimization is enabled. This optimization results in splitting structure types to improve performance.

Inter-Procedural Constant Propagation

The **–IPA:cprop=[on|off]** option is used to enable/disable inter-procedural constant propagation.

AMD ACML Library

The **–OPT:fast_math=[on|off]** option is used to enable/disable the utilization of the AMD ACML Library. The AMD ACML library is a free math library released by AMD for improving mathematic computation speed on AMD processors. When the option is **on**, the compiler will convert certain math functions to use the ACML library to improve performance. The library must be specified to the linker if this option is on.