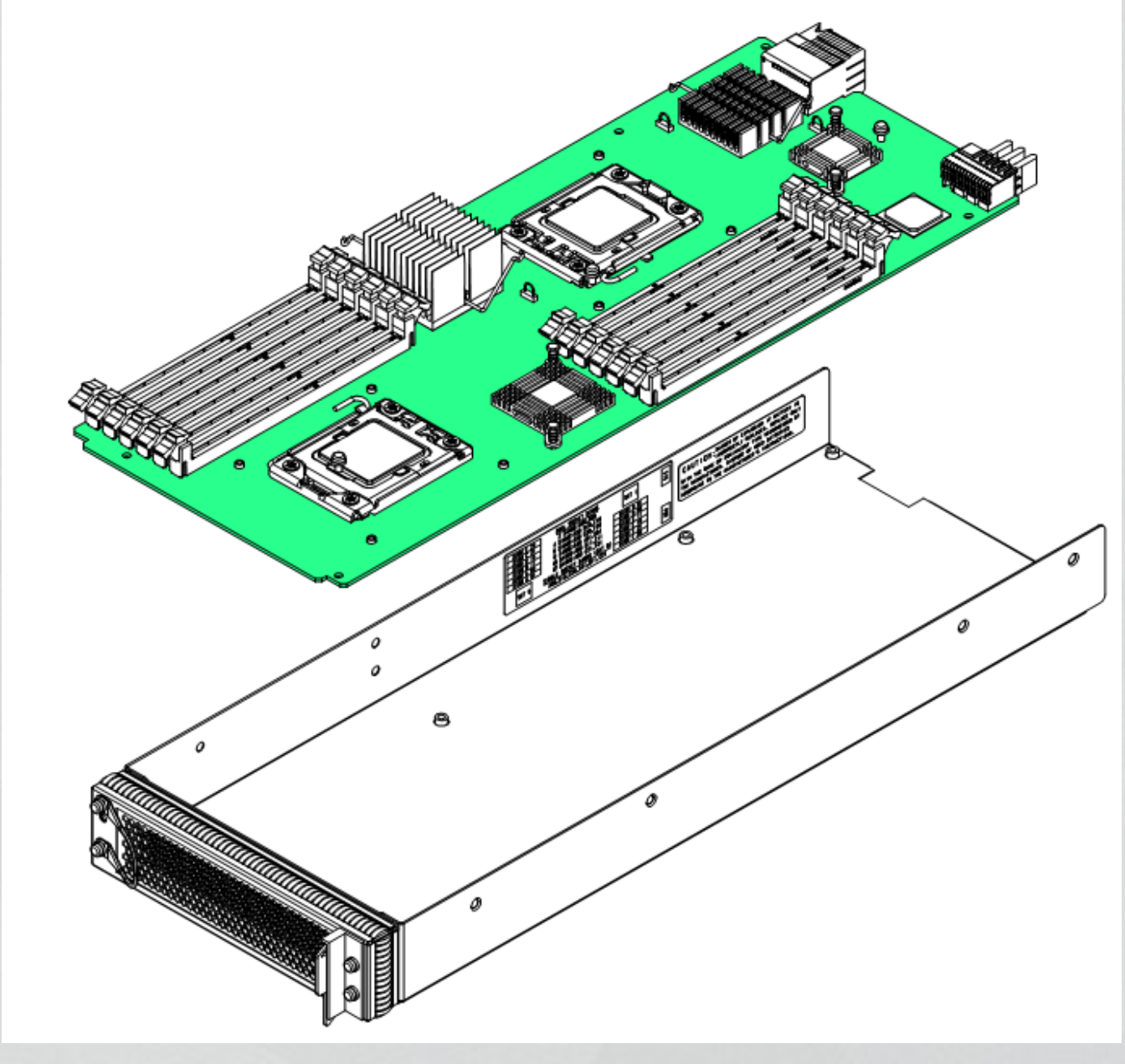



Master Class: Programming on Supercomputers

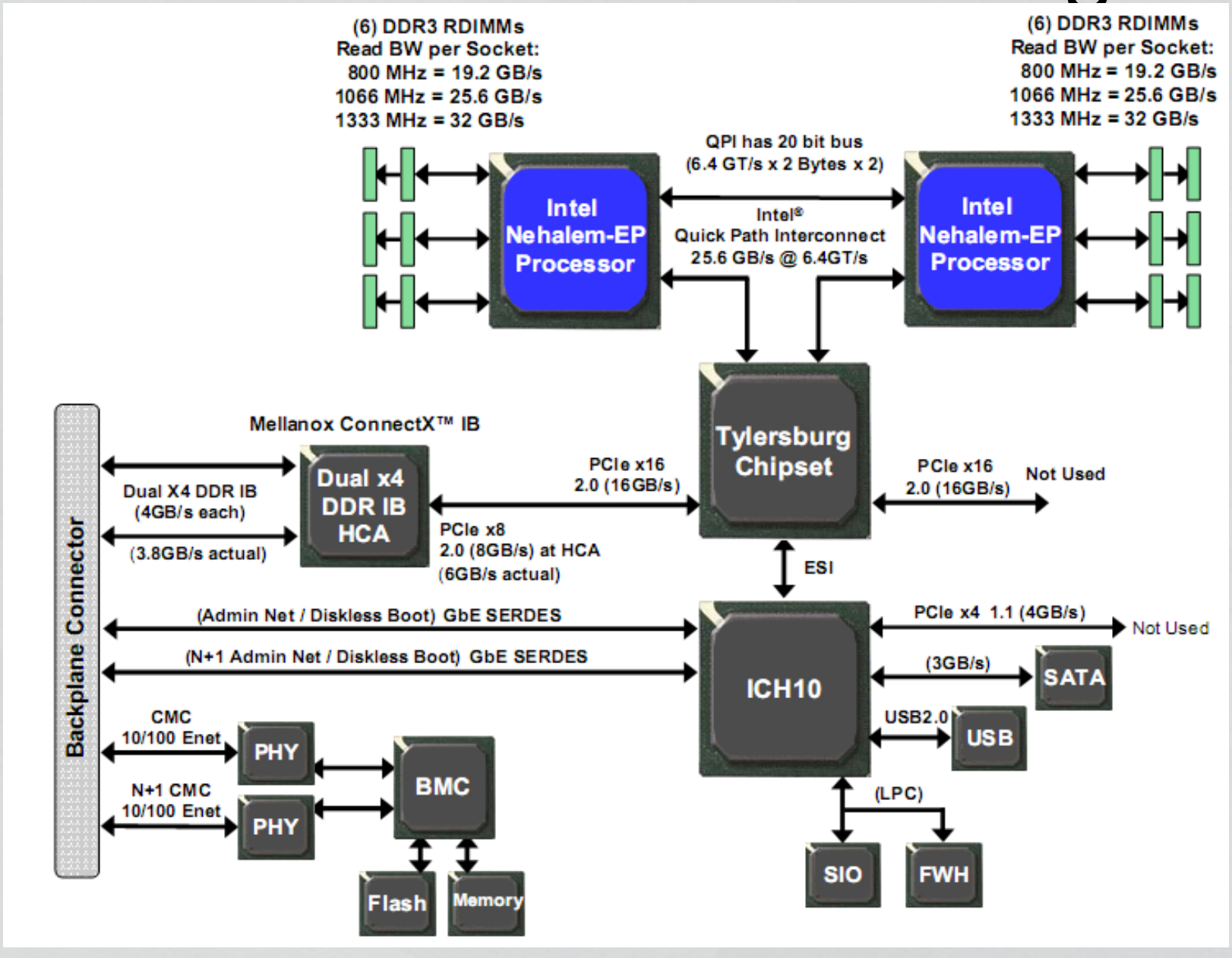
Resource – Intel X86_64 Application Dev

Intel XEON 5500 (Nehalem) Microprocessor

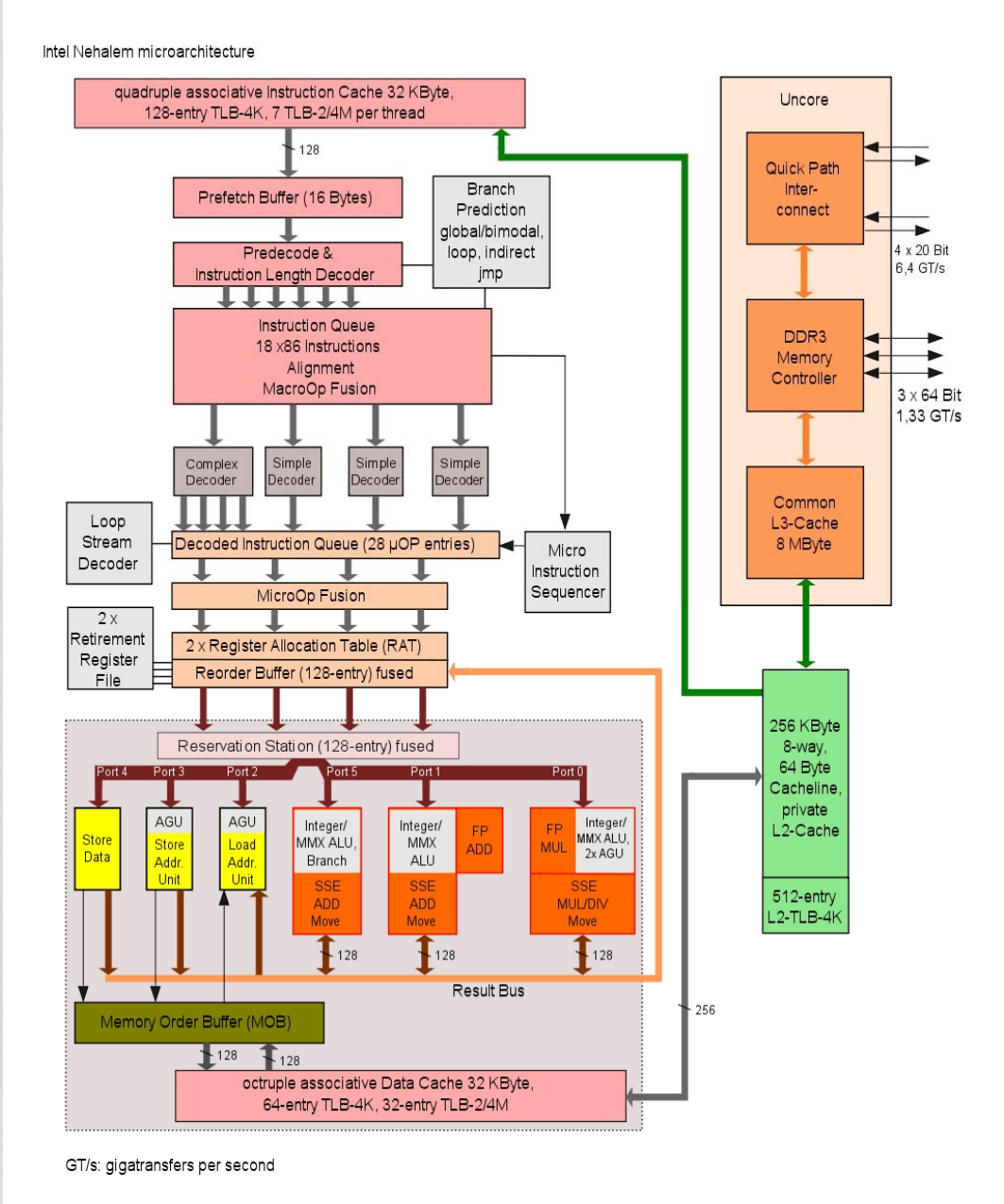
SGI ICE IP95 Blade



Nehalem node block diagram



Intel Micro-Architecture



Intel I7 Register Sets

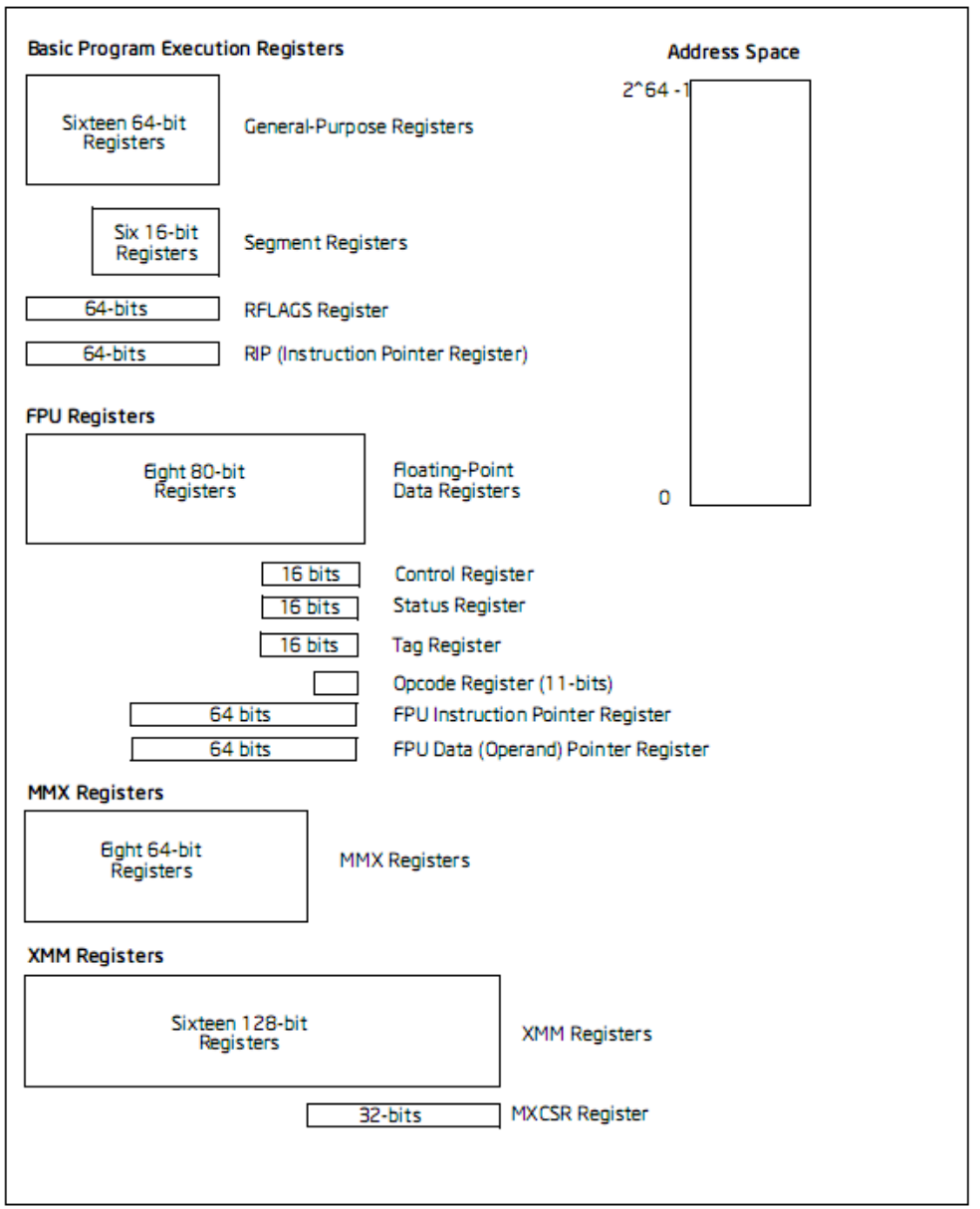


Figure 3-2. 64-Bit Mode Execution Environment

Intel I7 SIMD extension

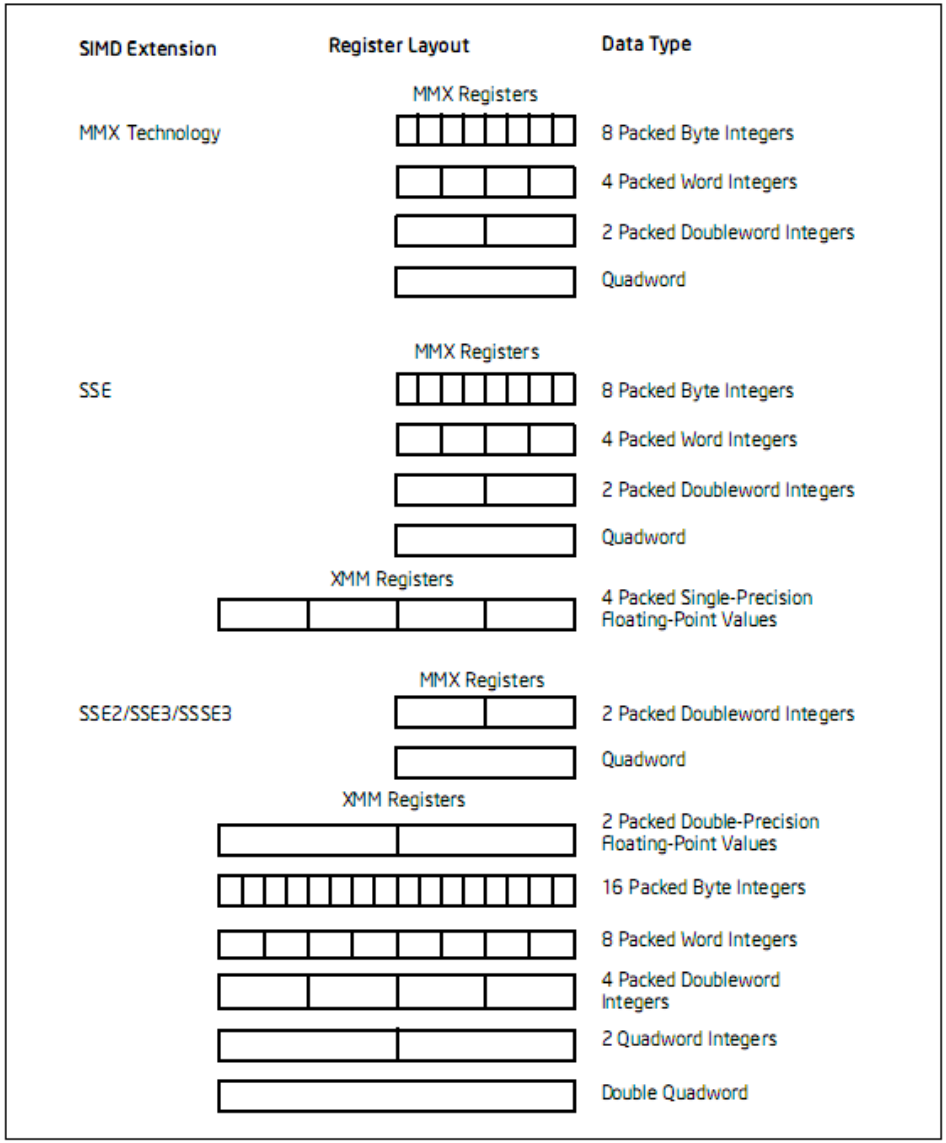
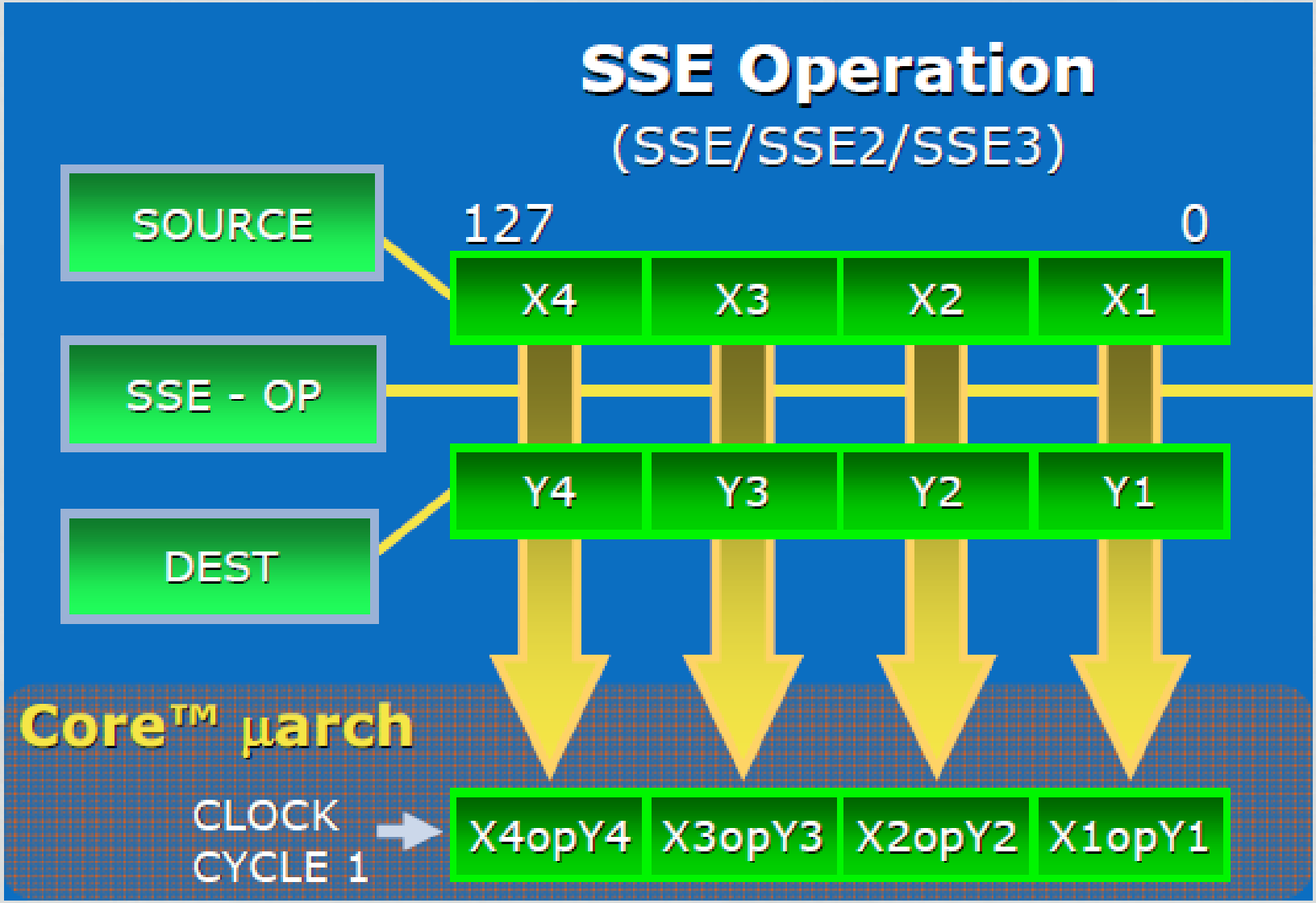
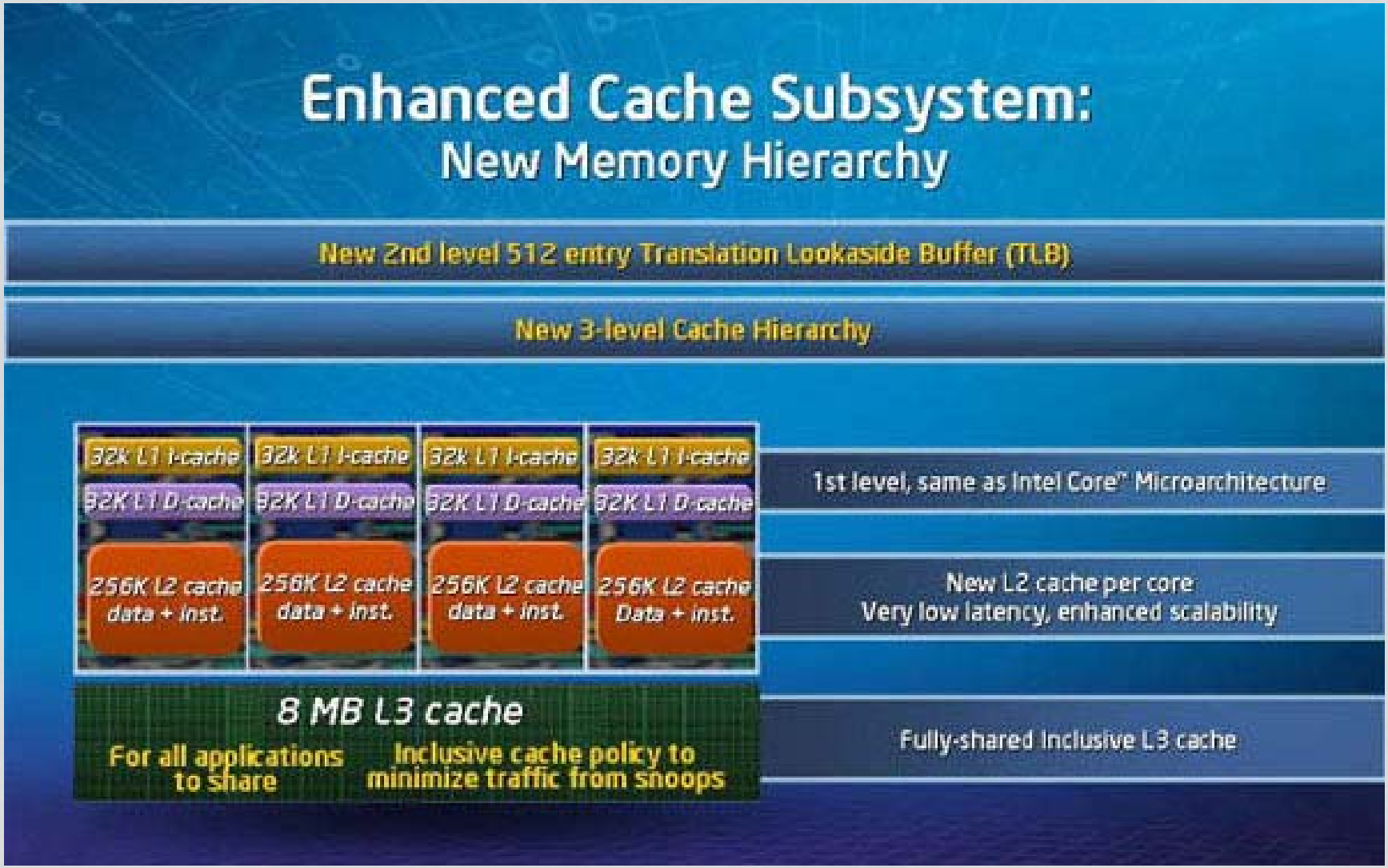


Figure 2-4. SIMD Extensions, Register Layouts, and Data Types

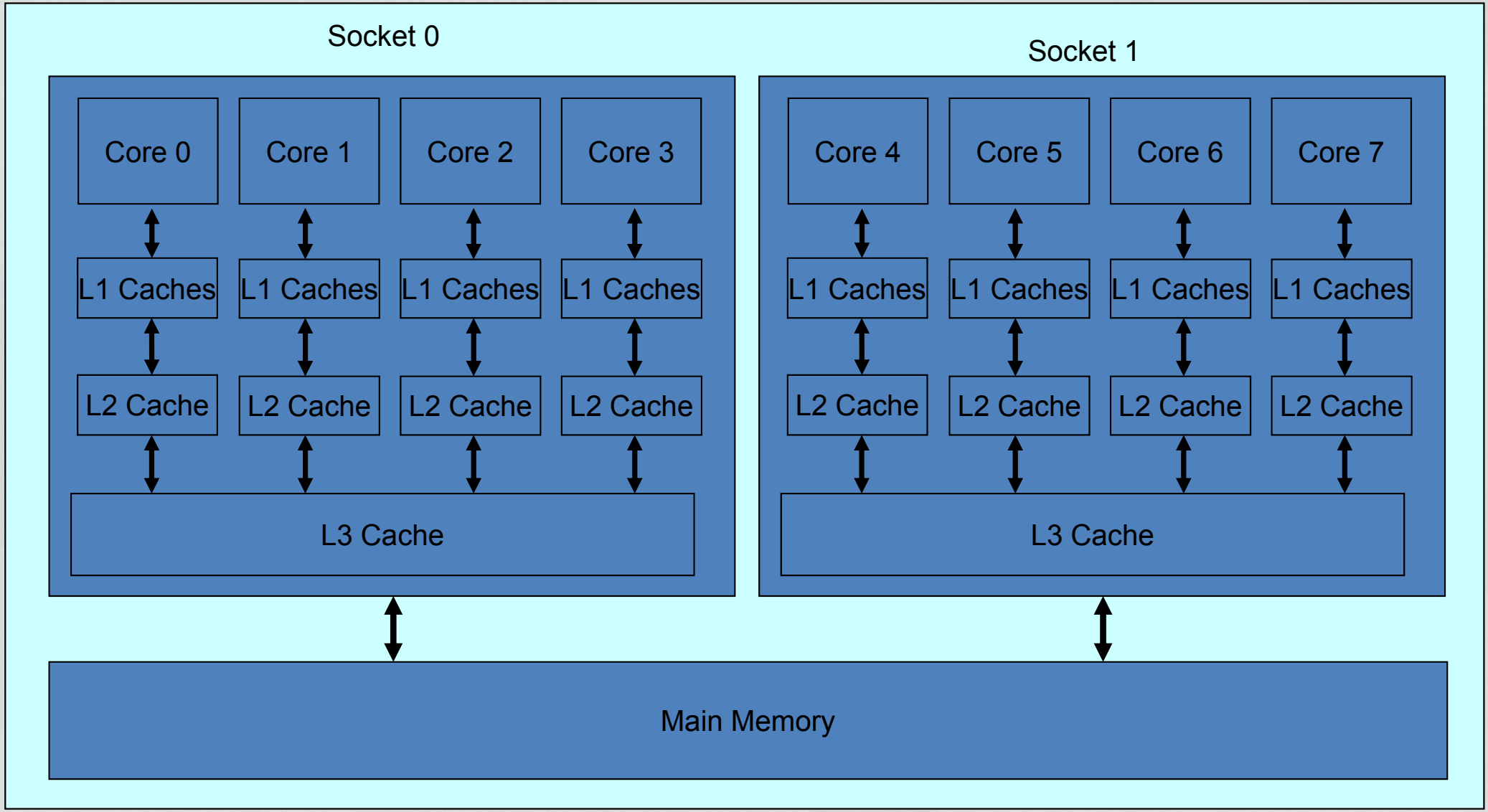
Intel XMM (Vector) Registers



Intel XEON 5500 Cache Subsystem



Nehalem node: application programming perspective



Intel C / C++ Compiler

Intel C / C++ Compiler : Environment

Steps to using the Intel C/C++ compiler:

- Set up the environment:
  - `> source <install-dir>/bin/iccvars.sh <arg>`
  - `> source <install-dir>/bin/iccvars.csh <arg>`
  - The scripts take an argument, `<arg>` specifying architecture:
    - `ia32` Compiler and libraries for IA-32 architectures only
    - `intel64` Compiler and libraries for Intel® 64 architectures only
    - `ia64` Compiler and libraries for IA-64 architectures only
  - Usually add this to your `.login`, `.profile` or `.cshrc` startup script

  OR

  `> module load <intel-cc>`
  - where modules are used, `<intel-cc>` will be site dependent
- Invoke the compiler:
  - C compiler
    `> icc`
  - C++ compiler
    `> icpc`

Intel C / C++ Compiler: Getting Started

Getting started:

- `> icc -help` for a summary of command line options
- `> man icc` man is your best friend!
- to find all the definitive documentation:

```
> which icc
/sw/sdev/intel/Compiler/11.0/074/bin/intel64/icc
ls /sw/sdev/intel/Compiler/11.0/074/Documentation
```

Compiler version information:

```
> icc –v
> icc –V
> icc --version
```

Intel C / C++ Compiler: Hello World!

The simplest case: compile AND link `test1.c` to produce executable `test1`:

```
> icc –o test1 test1.c
```

The following separates the compile and link (ld) phases:

```
> icc –c test1.c          compiles an object file: test1.o
> icc –o test1 test1.o    builds an executable file: test1
```

Save compiler version information in the executable:

```
 > icc –sox –o test1 test1.c
 > strings -a test1 |grep comment
-?comment:Intel(R) C++ Compiler Professional for applications running
   on Intel(R) 64, Version 11.0    Build 20081105 %s  : test1.c : -sox
   -o test1 -?comment:Intel(R) C++ Compiler Professional for
   applications running on Intel(R) 64, Version 11.0    Build 20081105
   %s  : test1.c : -sox -o test1 -defaultlib:libirc -defaultlib:libirc
```

Intel C / C++ Compiler: Default Behaviour

If you do not specify any options when you invoke the Intel® C/C++ Compiler, the compiler performs the following:

- produces an executable file (`a.out` by default)
- invokes options specified in a configuration file first
- invokes options specified on the command line
- searches for header files in known locations
- sets 16 bytes as the strictest alignment constraint for structures
- displays error and warning messages
- uses ANSI with extensions
- performs standard optimizations (usually `-O2`)
- on operating systems that support characters in Unicode* (multi-byte) format, the compiler will process file names containing these characters

Intel C / C++ Compiler: Input File Processing

Input File Name	Interpretation	Action
file.c	C source file	Passed to compiler
file.C, file.CC, file.cc, file.cpp, file.cxx	C++ source file	Passed to compiler
file.a, file.so	Library file	Passed to linker
file.i	Preprocessed file	Passed to stdout
file.o	Object file	Passed to linker
file.s	Assembly file	Passed to assembler

Intel C / C++ Compiler: Output Files Produced

Compiler output files:

file.i Preprocessed file -- produced with the `-P` option.

file.o Object file -- produced with the `-c` option.

file.s Assembly language file -- produced with the `-S` option.

a.out Executable file -- produced by default compilation

Intel C / C++ Compiler: Debugging

Debugging:

```
> icc –g test1 test1.c
```

Generates code to support symbolic debugging.

Warning: This option changes the default optimization from `–O2` to `–O0`
That is: **no optimisation**!

If this is not what you want, then use:

```
> icc –g –O2 test1 test1.c
```

The compiler supports `gdb`, `idb` (gui version), and `idbc` (Command lLne Version).

Intel C / C++ Compiler: Stack Traceback

- `traceback` tells the compiler to generate extra information in the object file to provide source file traceback information when a severe error occurs at run time.

The default (`-notraceback`) is to produce no traceback information.

Intel C / C++ Compiler: Warnings

Use the following compiler options to control remarks, warnings, and errors:

Option	Result
`-w`	Suppress all warnings
`-w0`	Display only errors
`-w1`	Display only errors and warnings (**default**)
`-w2`	Display errors, warnings, and remarks
`-Wbrief`	Display brief one-line diagnostics
`-Wcheck`	Enable more strict diagnostics
`-Werror-all`	Change all warnings and remarks to errors
`-Werror`	Change all warnings to errors

Intel C / C++ Compiler: Target Processor

Ensure that you are compiling for the correct target system.

This is especially important on cluster systems because the login-in (service) nodes may not have the same CPU as the compute nodes.

`-x[processor]` specifies the processor for which code is generated.

Possible values are:

`-xhost`	
`-xsse2`	
`-xsse3`	
`-xssse3`	Clovertown
`-xsse4.1`	Harpertown
`-xsse4.2`	i5, i7, Nehalem

Intel C / C++ Compiler: linux cpuinfo

```
> cat /proc/cpuinfo
processor       : 3
vendor_id       : GenuineIntel
cpu family      : 6
model           : 15
model name      : Intel(R) Xeon(R) CPU           X5350  @ 2.66GHz
stepping        : 7
cpu MHz         : 1600.000
cache size      : 4096 KB
physical id     : 1
siblings        : 4
core id         : 1
cpu cores       : 4
fpu       : yes
fpu_exception   : yes
cpuid level     : 10
wp        : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
    dts acpi mmx fxsr sse sse2 ss ht tm syscall nx lm constant_tsc pni monitor ds_cpl vmx est tm2
    ssse3 cx16 xtpr dca lahf_lm
bogomips        : 5333.55
clflush size    : 64
cache_alignment     : 64
address sizes   : 36 bits physical, 48 bits virtual
power management    :
```

Intel C/C++ Compiler: floating point options

`-fp-model`	Specifies semantics used in floating-point calculations. Values are `precise`, `fast` [`=1/2`], `strict`, `source`, `double`, `extended`, [`no-`]`except`.
`-fp-speculation`	Specifies the speculation mode for floating-point operations. Values are `fast`, `safe`, `strict`, and `off`.
`-prec-div`	Attempts to use slower but more accurate implementation of floating-point divide. Use this option to disable the divide optimizations in cases where it is important to maintain the full range and precision for floating-point division. Using this option results in greater accuracy with some loss of performance. Specifying `-no-prec-div` enables optimizations that result in slightly less precise results than full IEEE division.
`-complex-limited-range`	Enables the use of basic algebraic expansions of some arithmetic operations involving data of type COMPLEX. This can cause performance improvements in programs that use a lot of COMPLEX arithmetic. Values at the extremes of the exponent range might not compute correctly.
`-ftz`	This option only has an effect when the main program is being compiled. This option flushes denormal results to zero when the application is in the gradual underflow mode. It may improve performance if denormal values are not critical to your application's behavior.

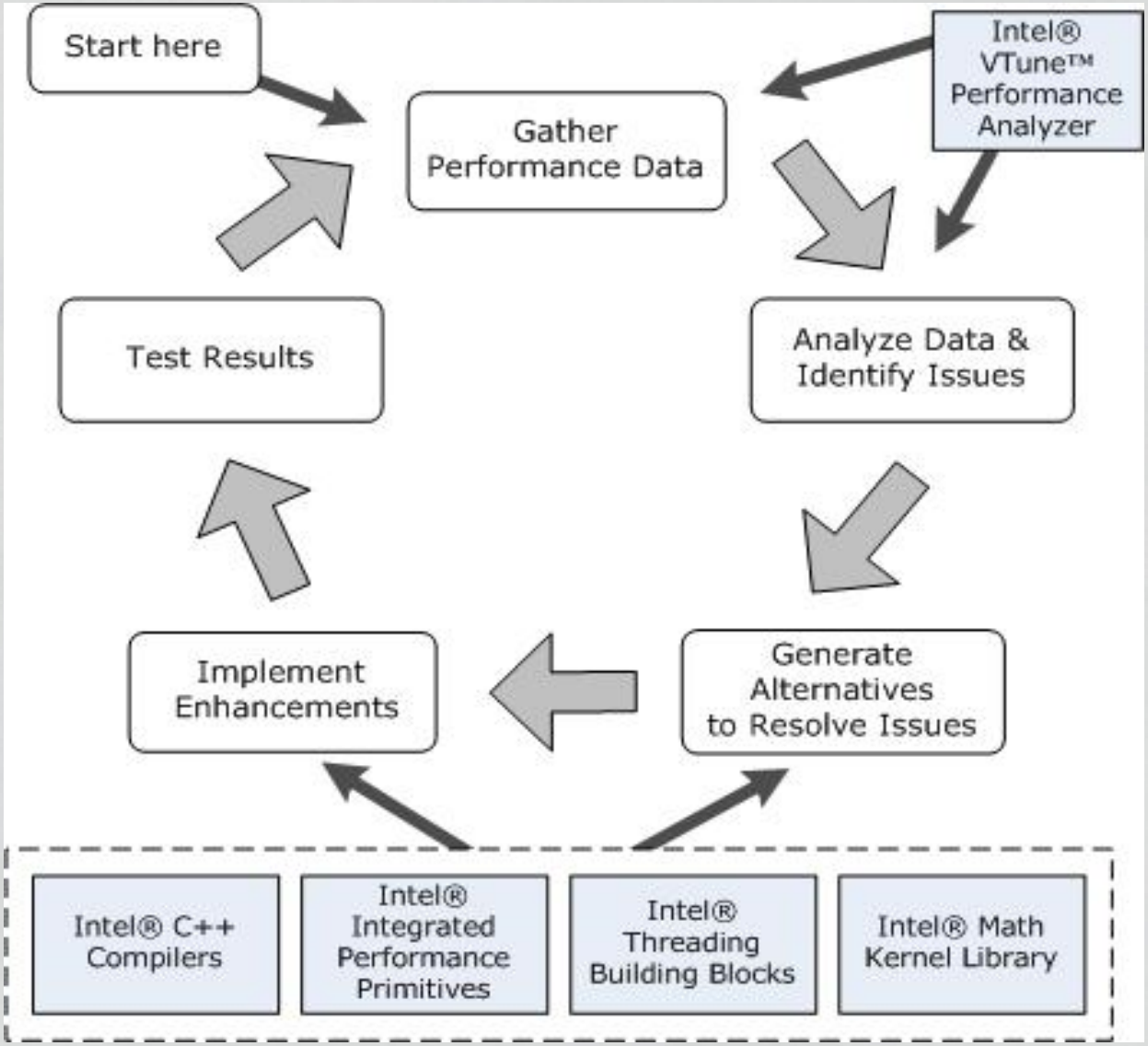
Intel C/C++ Compiler: floating point options (continued)

`-prec-sqrt`	Improves the accuracy of square root implementations, but using this option may impact speed.
`-pc`	Changes the floating point significand precision. Use this option when compiling applications. The application must use `main()` as the entry point, and you must compile the source file containing `main()` with this option.
`-rcd`	Disables rounding mode changes for floating-point-to-integer conversions
`-fp-port`	Causes floating-point values to be rounded to the source precision at assignments and casts.
`-mp1`	This option rounds floating-point values to the precision specified in the source program prior to comparisons. It also implies `-prec-div` and `-prec-sqrt`. This option has less impact to performance and disables fewer optimizations than the `-fp-model` precise option.

Intel C/C++ Compiler: -fp-model options

`precise`	Enables value-safe optimizations on floating-point data.
`fast=1,fast=2`	Enables more aggressive optimizations on floating-point data.
`strict`	Enables `precise` and `except`, disables contractions, and enables pragma stdc fenv_access.
`source`	Rounds intermediate results to source-defined precision and enables value-safe optimizations.
`double`	Rounds intermediate results to 53-bit (double) precision.
`extended`	Rounds intermediate results to 64-bit (extended) precision .
`[no-]except`	Determines whether floating-point exception semantics are used.

Intel optimization flow chart



Intel C / C++ Compiler: Optimization

Option	Intel	gcc
`-O0`	Turns off optimization.	**Default**. Turns off optimization. Same as -O.
`-O1`	Decreases code size with some increase in speed	Decreases code size with some increase in speed.
`-O2` `-O`	**Default**. Favors speed optimization with some increase in code size. Same as -O. Intrinsics, loop unrolling, and inlining are performed.	Optimizes for speed as long as there is not an increase in code size. Loop unrolling and function inlining, for example, are not performed
`-O3`	Enables -O2 optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop and memory access transformations	Optimizes for speed while generating larger code size. Includes -O2 optimizations plus loop unrolling and inlining. Similar to -O2 -ip on Intel compiler.
`-fast`	Enables a collection of common, recommended optimizations for run-time performance. Can introduce architecture dependency	

Intel C/C++ Compiler: assembly listing

```
double add_2(double a, double b) {
   return a + b;
}
```

The –S option compiles to an assembly file, `add_2.s` containing the assembler code

```
> icc –S add_2.c
```

```
add_2:
# parameter 1: %xmm0
# parameter 2: %xmm1
..B1.1:                         # Preds ..B1.0
..___tag_value_add_2.1:                                         #1.34
        addsd     %xmm1, %xmm0                                  #2.15
        ret                                                     #2.15
```

```
> icc –c add_2.s
```

Will create a `add_2.o` file.

Intel C/C++ Compiler: automatic vectorization overview

The automatic vectorizer (also called the auto-vectorizer) is a component of the Intel® compiler that automatically uses SIMD instructions in the MMX™, Intel® Streaming SIMD Extensions (Intel® SSE, SSE2, SSE3 and SSE4 Vectorizing Compiler and Media Accelerators) and Supplemental Streaming SIMD Extensions (SSSE3) instruction sets.

The vectorizer detects operations in the program that can be done in parallel, and then converts a number of sequential operations to one SIMD instruction that processes 2, 4, 8 or 16 elements in parallel, depending on the data type.

Intel C/C++ Compiler: automatic vectorization overview

- The compiler supports a variety of directives that can help the compiler to generate effective vector instructions.
  - automatic vectorization
  - vectorization with user intervention
- Vectorization options:

`-x`	Generates specialized code to run exclusively on processors with the extensions specified as the *processor* value. See *Targeting IA-32 and Intel® 64 Architecture Processors Automatically* for more information about using the option.
`-ax`	Generates, in a single binary, code specialized to the extensions specified as the *processor* value and also generic IA-32 architecture code. The generic code is usually slower. See *Targeting Multiple IA-32 and Intel® 64 Architecture Processors for Run-time Performance* for more information about using the option
`-vec` `-no-vec`	Enables or disables vectorization and transformations enabled for vectorization. The default is that vectorization is enabled.
`-vec-report`	Controls the diagnostic messages from the vectorizer.

Intel C/C++ Compiler: vectorization reports

`-vec-report[n]`

`0`	Tells the vectorizer to report no diagnostic information.
`1`	Tells the vectorizer to report on vectorized loops. (**default**)
`2`	Tells the vectorizer to report on vectorized and non-vectorized loops.
`3`	Tells the vectorizer to report on vectorized and non-vectorized loops and any proven or assumed data dependences.
`4`	Tells the vectorizer to report on non-vectorized loops.
`5`	Tells the vectorizer to report on non-vectorized loops and the reason why they were not vectorized.

Intel C/C++ Compiler: guidelines for vectorization

Programming Guidelines for Vectorization

- The goal of vectorizing compilers is to exploit single-instruction multiple data (SIMD) processing automatically. Users can help, however, by supplying the compiler with additional information; for example, by using directives.

Guidelines

- You will often need to make some changes to your loops. Follow these guidelines for loop bodies.

Use:

- straight-line code (a single basic block)
- vector data only; that is, arrays and invariant expressions on the right hand side of assignments. Array references can appear on the left hand side of assignments.
- only assignment statements

Intel C/C++ Compiler: tips for vectorization

Avoid:

- function calls
- unvectorizable operations (other than mathematical)
- mixing vectorizable types in the same loop
- data-dependent loop exit conditions

To make your code vectorizable, you will often need to make some changes to your loops. However, you should make only the changes needed to enable vectorization and no others. In particular, you should avoid these common changes:

- loop unrolling; the compiler does it automatically.
- decomposing one loop with several statements in the body into several single-statement loops.

Intel C/C++ Compiler: restrictions for vectorization

Restrictions:

There are a number of restrictions that you should consider. Vectorization depends on two major factors:

- **Hardware**

  The compiler is limited by restrictions imposed by the underlying hardware. In the case of Streaming SIMD Extensions, the vector memory operations are limited to stride-1 accesses with a preference to 16-byte-aligned memory references. This means that if the compiler abstractly recognizes a loop as vectorizable, it still might not vectorize it for a distinct target architecture

- **Style of source code**

  The style in which you write source code can inhibit optimization. For example, a common problem with global pointers is that they often prevent the compiler from being able to prove that two memory references refer to distinct locations. Consequently, this prevents certain reordering transformations.

Intel C/C++ Compiler: Interprocedural Optimization (IPO)

InterProcedural Optimization is an automatic, multi-step process: compilation and linking; however, IPO supports two compilation models:

- `-ip` single-file compilation and
- `-ipo` multi-file compilation.

Intel C/C++ Compiler: Interprocedural Optimization (IPO)

- Single-file compilation results in one real object file for each source file being compiled. During single-file compilation the compiler performs inline function expansion for calls to procedures defined within the current source file.

- The compiler performs some single-file interprocedural optimization at the default optimization level, `-O2`, additionally the compiler performs some inlining for the `-O1` optimization level, like inlining functions marked with inlining pragmas or attributes (GNU C and C++) and C++ class member functions with bodies included in the class declaration.

Intel C/C++ Compiler: Interprocedural Optimization (IPO)

`-ipo`

Multi-file compilation results in one or more mock object files rather than normal object files.

Additionally, the compiler collects information from the individual source files that make up the program.

Using this information, the compiler performs optimizations across functions and procedures in different source files.

Inlining is the most powerful optimization supported by IPO.

```
icc -opt-report n -opt-report-phase=ipo
```

Where `n`, if specified, can be `0`, `1`, `2`, or `3`. Otherwise the default is `2`.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

Profile-Guided Optimization (PGO)

- PGO improves application performance by:
  - Reorganizing code layout to reduce instruction-cache problems,
  - Shrinking code size, and
  - reducing branch mispredictions.

- PGO provides information to the compiler about the areas in an application that are most frequently executed.

- By knowing these areas, the compiler is able to be more selective and specific in optimizing the application.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

PGO consists of three phases or steps:

Step one: Instrument the program.

In this phase, the compiler creates and links an instrumented program from your source code with special code from the compiler.

Step two: Run the instrumented executable.

Each time you execute the instrumented code, the instrumented program generates a dynamic information file, which is used in the final compilation.

Step three: Final compilation.

When you compile the second time, the dynamic information files are merged into a summary file. Using the summary of the profile information in this file, the compiler attempts to optimize the execution of the most heavily traveled paths in the program.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

PGO enables the compiler to take better advantage of the processor architecture, more effective use of instruction paging and cache memory, and make better branch predictions. PGO provides the following benefits:

- Use profile information for register allocation to optimize the location of spill code.
- Improve branch prediction for indirect function calls by identifying the most likely targets. (Some processors have longer pipelines, which improves branch prediction and translates into high performance gains.)
- Detect and do not vectorize loops that execute only a small number of iterations, reducing the run time overhead that vectorization might otherwise add.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

InterProcedural Optimization (IPO) and PGO can affect each other.

Using PGO can often enable the compiler to make better decisions about function inlining, which increases the effectiveness of interprocedural optimizations.

Unlike other optimizations, such as those strictly for size or speed, the results of IPO and PGO vary. This variability is due to the unique characteristics of each program, which often include different profiles and different opportunities for optimizations.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

Performance Improvements with PGO

PGO works best for code with many frequently executed branches that are difficult to predict at compile time.

An example is a code with intensive error-checking in which the error conditions are false most of the time.

The infrequently executed (cold) error-handling code can be relocated so the branch is rarely predicted incorrectly.

Minimizing cold code interleaved into the frequently executed (hot) code improves instruction cache behavior.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

When you use PGO, consider the following guidelines:

- Minimize changes to your program after you execute the instrumented code and before feedback compilation.
  During feedback compilation, the compiler ignores dynamic information for functions modified after that information was generated.
  If you modify your program, the compiler issues a warning that the dynamic information does not correspond to a modified function.

- Repeat the instrumentation compilation if you make many changes to your source files after execution and before feedback compilation.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

- Know the sections of your code that are the most heavily used.
  If the data set provided to your program is very consistent and displays similar behavior on every execution, then PGO can probably help optimize your program execution.

- Different data sets can result in different algorithms being called.
  The difference can cause the behavior of your program to vary for each execution. In cases where your code behavior differs greatly between executions, PGO may not provide noticeable benefits.
  If it takes multiple data sets to accurately characterize application performance then execute the application with each of these data sets then merge the dynamic profiles; this technique should result in an optimized application.

You must insure the benefit of the profiled information is worth the effort required to maintain up-to-date profiles.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)

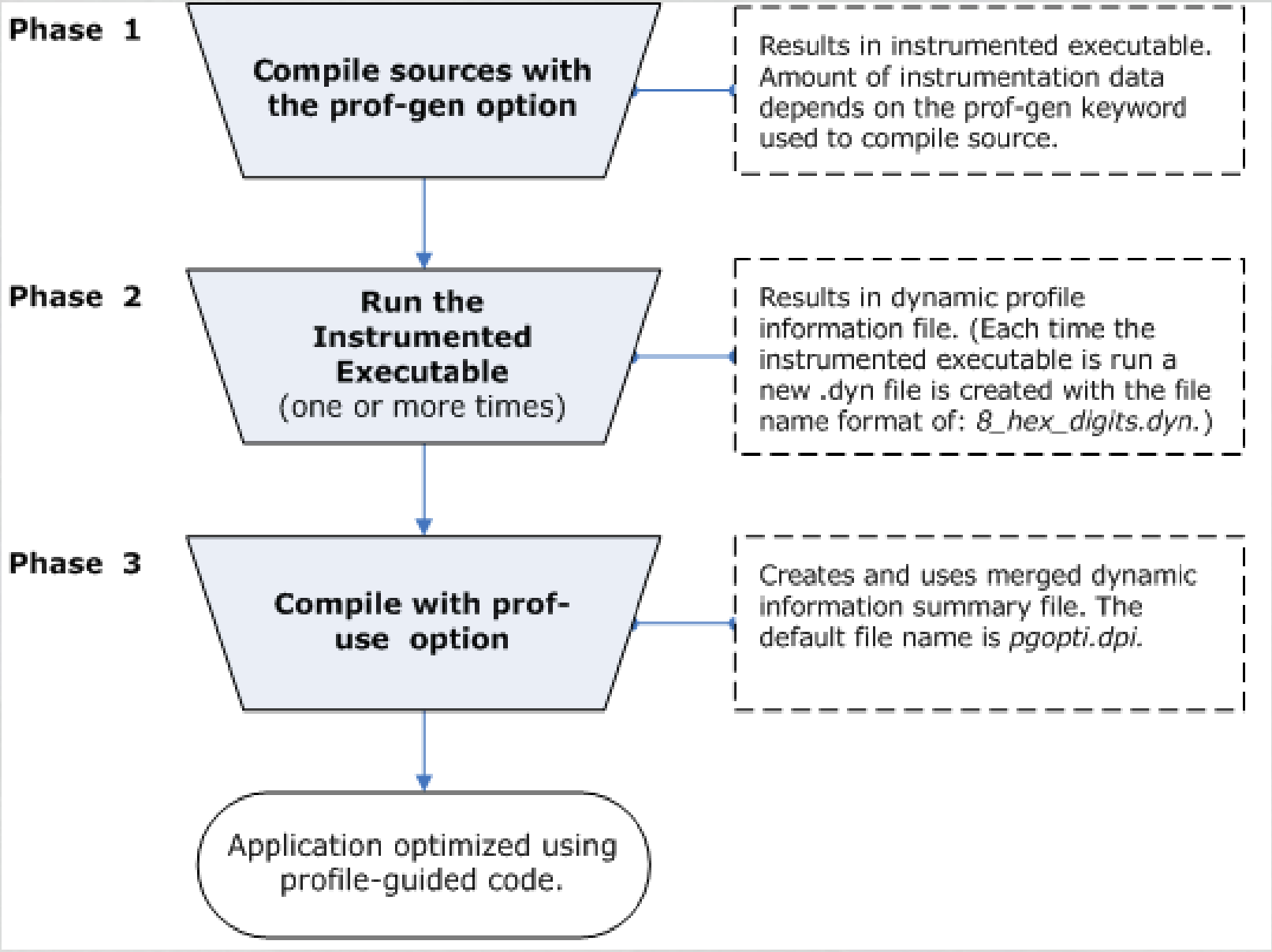
PGO consists of three phases (or steps):

1. Generating instrumented code by compiling with the `-prof-gen` option when creating the instrumented executable.

2. Running the instrumented executable, which produces dynamic-information (`.dyn`) files.

3. Compiling the application using the profile information using the `-prof-use` option.

Intel C/C++ Compiler: Profile-Guided Optimization (PGO)



Intel C/C++ Compiler: Code Coverage Tool

The code coverage tool, codecov, provides software developers with a view of how much application code is exercised when a specific workload is applied to the application.

To determine which code is used, the code coverage tool uses Profile-guided Optimization (PGO) options and optimizations.

The major features of the code coverage tool are:

- Visually presenting code coverage information for an application with a customizable code coverage coloring scheme
- Displaying dynamic execution counts of each basic block of the application
- Providing differential coverage, or comparison, profile data for two runs of an application

Intel C/C++ Compiler: Code Coverage Tool

To use `codecov` on an application, the following items must be available:

- The application sources.
- The `.spi` file generated by the compiler when compiling the application for the instrumented binaries using the `-prof-gen=srcpos` options.
- A `pgopti.dpi` file that contains the results of merging the dynamic profile information (`.dyn`) files, which is most easily generated by the `profmerge` tool.
  This file is also generated implicitly by the Intel® compilers when compiling an application with `-prof-use` options with available `.dyn` and `.dpi` files.

Intel C/C++ Compiler: Code Coverage Tool

CODE_COVERAGE.HTML

Intel C/C++ Compiler: High-Level Optimization (HLO)

High-Level Optimization (HLO)

HLO exploits the properties of source code constructs (for example, loops and arrays) in applications developed in high-level programming languages. Within HLO, loop transformation techniques include:

- Loop Permutation or Interchange
- Loop Distribution
- Loop Fusion
- Loop Unrolling
- Data Prefetching
- Scalar Replacement
- Unroll and Jam
- Loop Blocking or Tiling
- Partial-Sum Optimization

Intel C/C++ Compiler: High-Level Optimization (HLO)

- Loadpair Optimization
- Predicate Optimization
- Loop Versioning with Low Trip-Count Check
- Loop Reversal
- Profile-Guided Loop Unrolling
- Loop Peeling
- Data Transformation: Malloc Combining and Memset Combining
- Loop Rerolling
- Memset and Memcpy Recognition
- Statement Sinking for Creating Perfect Loopnests

Intel C/C++ Compiler: High-Level Optimization (HLO)

Some HLO is done with default optimization, `-O2`

However, most HLO is done at optimization level, `-O3`

`-unroll`*n* Specifies the maximum number of times a loop is unrolled.

`-unroll0` Disables loop unrolling

```
#pragma unroll
#pragma unroll(n)
#pragma nounroll
```

Indicates to the compiler to unroll or not to unroll a counted loop. These pragmas are supported only with `-O3`.

Intel C/C++ Compiler: High-Level Optimization (HLO)

When loops are dependent, improving loop performance becomes much more difficult. Loops can be dependent in several, general ways:

- **Flow Dependency**, read after write:

```
for (int j=1; j<1024; j++)
  A[j]=A[j-1];
```

- **Anti Dependency**, write after read:

```
for (int j=1; j<1024; j++)
  A[j]=A[j+1];
```

- **Output Dependency**, write after write:

```
for (int j=1; j<1024; j++) {
   A[j]=B[j];
   A[j+1]=C[j];
}
```

Intel C/C++ Compiler: High-Level Optimization (HLO)

Reductions: The Intel® compiler can successfully vectorize most loops containing reductions on simple math operators like multiplication (*), addition (+), subtraction (-), and division (/). Reductions collapse array data to scalar data by using associative operations:

```
sum = 0.0;
for (int j=0; j<MAX; j++) {
    sum += c[j];
}
```

Intel C/C++ Compiler: Automatic Parallelization

The auto-parallelization feature of the Intel® compiler automatically translates serial portions of the input program into equivalent multi-threaded code.

The auto-parallelizer analyzes the dataflow of the loops in the application source code and generates multi-threaded code for those loops which can safely and efficiently be executed in parallel.

This allows applications to exploit the full potential of the parallel architecture found in current symmetric multi-processor (SMP) systems.

The parallel run-time support provides the same run-time features as found in OpenMP, such as handling the details of loop iteration modification, thread scheduling, and synchronization.

Intel C/C++ Compiler: Automatic Parallelization

`-parallel`	Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel. Depending on the program and level of parallelization desired, you might need to set the `KMP_STACKSIZE` environment variable to a larger size.
`-par-threshold[n]`	Sets a threshold for the auto of loops based on the probability of profitable execution of the loop in parallel. Valid values of n can be 0 to 100. (default is 100)
`-par-schedule-`*`keyword[=n]`*	Specifies the scheduling algorithm or a tuning method for loop iterations. It specifies how iterations are to be divided among the threads of the team.
`-par-report[n]`	Controls the diagnostic levels in the auto-parallelizer optimizer. Valid values of n can be 0, 1, 2, 3 (default is 1)

Intel C/C++ Compiler: Automatic Parallelization

`-par-schedule-keyword[=n]`

`auto`	Lets the compiler or run-time system determine the scheduling algorithm .
`static`	Divides iterations into contiguous pieces.
`static-balanced`	Divides iterations into even-sized chunks.
`static-steal`	Divides iterations into even-sized chunks, but allows threads to steal parts of chunks from neighboring threads.
`dynamic`	Gets a set of iterations dynamically.
`guided`	Specifies a minimum number of iterations.
`guided-analytical`	Divides iterations by using exponential distribution or dynamic distribution.
`runtime`	Defers the scheduling decision until run time.

Intel C/C++ Compiler: Automatic Parallelization

Three requirements must be met for the compiler to automatically parallelize a loop:

1. The number of iterations must be known before entry into a loop so that the work can be divided in advance. A while-loop, for example, cannot usually be parallelized.
2. There can be no jumps into or out of the loop.
3. The loop iterations must be independent.

If you know that parallelizing a particular loop is safe and that potential aliases can be ignored, you can instruct the compiler to parallelize the loop using:
`#pragma parallel`

Intel C/C++ Compiler: Parallel Environment Variables

Auto-parallelization uses the following OpenMP and KMP environment variables:

`OMP_NUM_THREADS`	Sets the maximum number of threads to use for parallel regions.
`KMP_STACKSIZE`	Sets the number of bytes to allocate for each OpenMP* thread to use as the private stack for the thread. Use the optional suffixes: b (bytes), k (kilobytes), m (megabytes), g (gigabytes), or t (terabytes) to specify the units.
`OMP_SCHEDULE`	Sets the run-time schedule type and an optional chunk size.
`KMP_VERSION`	Enables (1) or disables (0) the printing of OpenMP run-time library version information during program execution. Default is 0.

Intel C/C++ Compiler: Optimization Reports

`-opt-report[n]`

`0`	Tells the compiler to generate no optimization report.
`1`	Tells the compiler to generate a report with the minimum level of detail.
`2`	Tells the compiler to generate a report with the medium level of detail. This is the default if n is not specified.
`3`	Tells the compiler to generate a report with the maximum level of detail.

Intel C/C++ Compiler: Optimization Reports

`-opt-report-phase=`

`ipo`	The Interprocedural Optimizer phase
`hlo`	The High Level Optimizer phase
`hpo`	The High Performance Optimizer phase
`ilo`	The Intermediate Language Scalar Optimizer phase
`pgo`	The Profile Guided Optimization phase
`all`	All optimizer phases

The default is to produce no optimization reports.