Build to solve the impossible

That's the power of Intel Xeon®

RMACC Conference 2026

RMACC Conference 2026

Intel® and GCC* Compilers: Unleash the Power of Intel® Xeon® 6/6+ Processors

Xinmin Tian
Intel Fellow, Compiler Engineering Chief Architect

Notices & Disclaimers

- Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.
- Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.
- Your costs and results may vary.
- Intel technologies may require enabled hardware, software or service activation.
- © Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

intel

Agenda

- Xeon® 6 and 6+ (codenamed Clearwater Forest) Processors and AVX10
  - AVX-VNNI-INT16, AMX, SHA512, etc.
- Intel Compilers and Tools
  - Recent Focus
- Compiler Optimizations for Intel® Xeon® 6/6+ processors
  - Vectorization using VNNI Instructions
- Torch.compile Performance Tuning
- Performance Results
- Key Take Aways

Intel® Xeon® 6/6+ Processors and AVX10

intel

Intel® Xeon® 6+ (codenamed Clearwater Forest) Processors

- Why it Matters
  - Advanced ISA features for AI inference, cryptography, and efficient data movement
  - Security-enhancing instructions like SHA512, SM4, SM3
  - Privilege-level transitions with FRED and LKGS
- Compiler Enablement
  - Intel® oneAPI DPC++/C++ Compiler 2025.3/2026.0 and Intel® Fortran Compiler 2025.3/2026.0 fully enabled and tuned for Intel® Xeon® 6+ for optimal application performance
- Intel Cryptography Primitives Library – SM3, SM4, SHA512 algorithms optimized

Intel® AVX10: Unifying Enhanced Vector Performance Across All Cores

- Why Intel® AVX10 Matters
  - All the power of AVX-512, now scalable across P-cores and E-cores
  - AVX10.1: Introduced in Intel® Xeon 6, supports flexible vector sizes
  - AVX10.2: Adds AI, media, cryptography and WebAssembly acceleration
- Compiler Enablement for Maximum Reach
  - GCC* & LLVM/Clang*: Full support for AVX 10.2 for next-gen Xeon platforms
  - Intel® DPC++/C++/Fortran Compilers tuned for performance on next-gen Xeon platforms

Intel ISA Evolution at a Glance

Intel® AVX
- 128/256-bit FP
- 16 registers
- NDS (and AVX128)
- Improved blend
- MASKMOV
- Implicit unaligned

Intel® AVX2
- Float16
- 128/256-bit FP FMA
- 256-bit int
- PERMD
- Gather

Intel® AVX-512
- 128/256/512-bit FP/Int
- 32 vector registers
- 8 mask registers
- 512-bit embedded rounding
- Embedded broadcast
- Scalar/SSE/AVX "promotions"
- Native media additions
- HPC additions
- Transcendental support
- Gather/Scatter
- Flag-based enumeration
- Intel® Xeon P-core only

Intel® AVX10.1 (pre-enabling)
- 128/256/512-bit FP/Int
- 32 vector registers
- 8 mask registers
- 512-bit embedded rounding
- Embedded broadcast
- Scalar/SSE/AVX "promotions"
- Native media additions
- HPC additions
- Transcendental support
- Gather/Scatter
- **Version-based** enumeration
- Intel® Xeon P-core only

Intel® AVX10.2
- New data movement, transforms and type instructions
- 128/256/512-bit FP/Int
- 32 vector registers
- 8 mask registers
- 512-bit embedded rounding
- Embedded broadcast
- Scalar/SSE/AVX "promotions"
- Native media additions
- HPC additions
- Transcendental support
- Gather/Scatter
- **Version-based** enumeration
- **Supported on P-cores, E-cores**

Figure 1: Intel® ISA Families and Features

Intel® Compilers and Tools

intel

Intel® Compilers and Tools

Flexible, Comprehensive, Developer Software Stack – Powered by Intel, Optimized for Intel CPUs and GPUs

Build and Run	Analyze & Debug	Scale
Intel compilers (C/C++, SYCL, Fortran, OpenMP) Triton & SYCL-TLA (CUTLASS) oneDPL (DPC++ Library) oneMKL (Math Kernel Library) oneTBB (Threading Library) IPP & IPP Crypto (Performance Primitives) oneDNN (Deep Neural Network Library)	Intel® VTune™ Profiler Intel® Distribution for GDB*	Intel® MPI Library Intel® SHMEM Library oneCCL (Collective Communications Library)

HPC and AI Applications Optimized on Intel Platforms

Modeling & Simulation:	QMCPACK, Amber, and many more	Data Analytics :	
Computing:	Relion, AMReX SeisSol and many more	DL Inference and Training:	Intel® Neural Compressor
Scientific Computing:	NWChemEX, QUDA, and many more	Classical ML:	XGBoost

Recent Focus: Improving our Foundational Technology

Compilers	Programming Models	Distributed Programming	Analyzers & Debugger
▪ Unlocking more application performance on GNR, CWF, DMR, PTL, NVL ▪ Enhanced Fortran 2023 support ▪ Optimized for offload application performance on Intel® GPUs ▪ HWPGO-based application optimizations	▪ OpenMP* – Offload support – OpenMP* 6.0 conformance for oneMKL's offload ▪ SYCL* interfaces for libraries ▪ SYCL interop with VK/DX Graphics APIs ▪ SYCL runtime tuning ▪ Threading Building Blocks – C++11 rewrite	▪ Intel® MPI Library – MPI 5.0 standard – Efficient utilization of Xeon® capabilities ▪ Fabrics ▪ oneAPI Collective Communication Library (oneCCL) ▪ Intel® SHMEM ▪ Compute 2D/3D FFTs on multiple GPUs	▪ Intel® VTune Profiler – Support for Intel® Arc B-series GPU and Panther Lake – Finalization speedup – Extended Timeline with CPU/GPU Kernel Connections

*Other names and brands may be claimed as the property of others. SYCL is a trademark of the Khronos Group Inc.

Compiler Optimizations for Intel® Xeon 6/6+ Processors

intel

Compiler Optimizations for Xeon 6/6+ Processors

- Intel Compilers, Clang Compiler and GCC Compilers
  - -march=pantherlake enables AVX-VNNI-INT16, SHA512, SM3 and SM4 extensions
  - -march=clearwaterforest enables AVX-VNNI-INT16, PREFETCHI, SHA512, SM3, SM4 and USER_MSR ISA extensions
- Compiler Optimizations: A Quick Glance
  - Vectorization for AVX-VNNI-INT16 (e.g. vpdpwuud instruction)
  - Optimize 4-operand vex blendv instruction to a 3-instruction sequence (op1 & mask) | (op2 & ~mask)
    - 4-operand vex blendv decoding is slower
  - Select 128-bit length instead of 256-bit length to avoid cross-lane 256-bit vector permutations
    - Slow and can't be split into two parallel 128-bit uops for execution
  - Enable unroll-and-partial sum in the vectorizer for FMA/DOT_PRODUCT reduction
    - Break cross-iteration dependency on VNNI/FMA accumulator to increase SIMD parallelism
  - Split FMA to FMUL and FADD, if there is a cross-iteration dependency on FMA accumulator
    - FMUL and FADD take less cycles than FMA
  - Select a scalar instructions instead of gather instructions
    - Gather instructions decode slower

Use the latest compilers for maximize application performance

intel

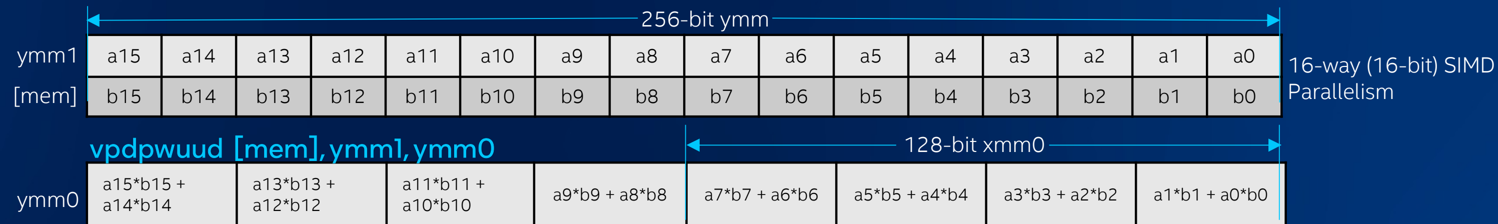
Automatic Vectorization: AVX-VNNI-INT16 vpdpwuud Code Generation

- vpdpwuud instruction

  v = Vector (VEX encoded)
  p = Packed
  dp = Dot Product
  wu = Word Unsigned (16-bit unsigned input)
  u = Unsigned (32-bit accumulators)
  d = Doubleword (32-bit output elements)

```
unsigned int udot_prod_hi (unsigned short * a, unsigned short * b, unsigned int c)  {
  for (int i = 0; i < 16; i++)
        c = c + ((int) a[i] * (int) b[i]);
  return c;
}
// Panther Lake: icpx –march=pantherlake vnni-int16.cpp
// Clearwater Forest: icpx –march=clearwaterforest vnni-int16.cpp
```

AVX-VNNI-INT16 vpdpwuud Execution Example




```
udot_prod_hi:
  ; Initialize accumulator with the initial sum value
  vmovd  %edx, %xmm0                ; xmm0 = [initial_sum, 0, 0, 0] (1x uint32_t + 3 zeros)
  ; Load 16 uint16_t elements from array b
  vmovups (%rsi), %ymm1             ; ymm1 = [b15 b14 ... b1 b0] (16x uint16_t = 256 bits)

  ; Compute dot product with zero-extension from 16-bit to 32-bit
  vpdpwuud (%rdi), %ymm1, %ymm0     ; ymm0 = ymm0 + (ymm1 * zero_extend([rdi]))

  ; Horizontal reduction to sum all partial results
  vextracti128 $1, %ymm0, %xmm1     ; Extract upper 128 bits: xmm1 = ymm0[255:128]
  vpaddd %xmm1, %xmm0, %xmm0        ; Add upper and lower halves: xmm0 = xmm0[127:0] + xmm1

  ; First stage reduction
  vpshufd $238, %xmm0, %xmm1        ; xmm1 = [xmm0[2], xmm0[3], xmm[2], xmm0[3]]
  vpaddd %xmm1, %xmm0, %xmm0        ; xmm0 = [xmm0[0]+xmm0[2], xmm0[1]+xmm0[3], ...]

  ; Final reduction
  vpshufd $85, %xmm0, %xmm1         ; xmm1 = [xmm0[1], xmm0[1], xmm0[1], xmm0[1]]
  vpaddd %xmm1, %xmm0, %xmm0        ; xmm0 = [final_sum, ...]
  vmovd  %xmm0, %eax                ; eax = final_sum
  vzeroupper                        ; Clear upper YMM registers (AVX-SSE transition)
  retq
```

vextracti128 $1, ymm0, xmm1

xmm1	a15*b15 + a14*b14	a13*b13 + a12*b12	a11*b11 + a10*b10	a9*b9 + a8*b8

vpaddd xmm1, xmm0, xmm0

xmm0	a15*b15 + a14*b14 + a7*b7 + a6*b6	a13*b13 + a12*b12 + a5*b5 + a4*b4	a11*b11 + a10*b10 + a3*b3 + a2*b2	a9*b9 + a8*b8 + a1*b1+a0*b0

4-way (32-bit) SIMD Parallelism

vpshufd $238, xmm0, xmm1 ; XMM1 = [XMM0[2], XMM0[3], XMM0[2], XMM0[3]]

xmm1	a15*b15 + a14*b14 + a7*b7 + a6*b6	a13*b13 + a12*b12 + a5*b5 + a4*b4	a15*b15 + a14*b14 + a7*b7 + a6*b6	a13*b13 + a12*b12 + a5*b5 + a4*b4

vpaddd xmm1, xmm0, xmm0

xmm0	Data no-used	Data no-used	a15*b15 + a14*b14 + a7*b7 + a6*b6 + a11*b11 + a10*b10 + a3*b3 + a2*b2	a13*b13 + a12*b12 + a5*b5 + a4*b4 + a9*b9 + a8*b8 + a1*b1+a0*b0

4-way (32-bit) SIMD Parallelism

vpshufd $85, xmm0, xmm1 ; XMM1 = [XMM0[1], XMM0[1], XMM0[1], XMM0[1]]

xmm1	Data no-used	Data no-used	a15*b15 + a14*b14 + a7*b7 + a6*b6 + a11*b11 + a10*b10 + a3*b3 + a2*b2	a15*b15 + a14*b14 + a7*b7 + a6*b6 + a11*b11 + a10*b10 + a3*b3 + a2*b2

vpaddd xmm1, xmm0, xmm0 ; XMM0 = [final_sum, ...]

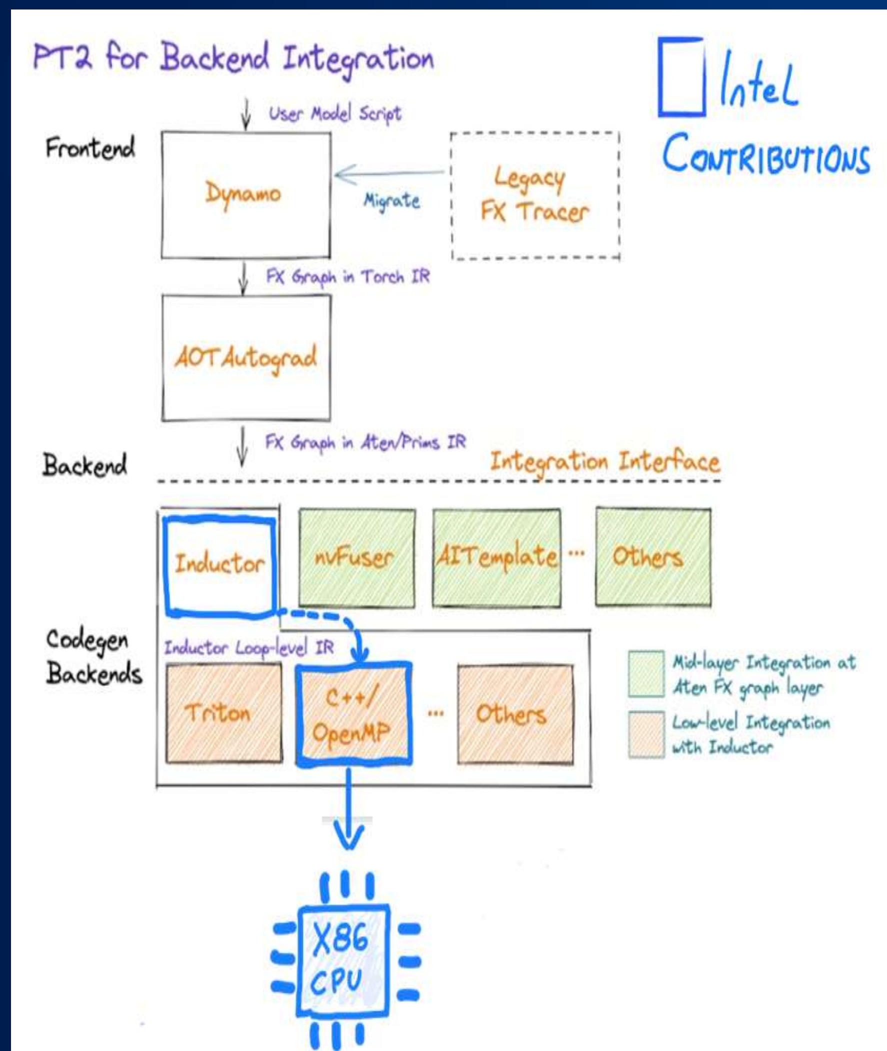
intel

Tuning Torch.compile

intel

TorchInductor Overview

- TorchInductor is a new Compiler Backend
  - Accelerate PyTorch models
  - Transform models into optimized code
    - Through a define-by-run Loop-Level IRs
- For CPUs, Torch Inductor generates OpenMP C/C++ code
  - Invoke native OpenMP compiler for code generate for modern CPUs and efficient parallel execution
- Intel® and Meta* maintain OpenMP C/C++ backend and optimize CPU performance
  - Kernel fusion, Kernel tuning, Intelligent thread scheduling, AMX and oneDNN integration etc.



Optimizations for Torch.Compile

- Enable compiler optimizations and the generation of non-temporal-stores
  - Providing more accurate alignment information
  - Use __restrict__ keyword
- Loop structures can be more efficient
  - Collapse loops
  - Restructure to increase parallelism
  - Restructure Remainder loops
- OpenMP
  - Generate additional directives
  - Unnecessary use of single in some places
- Vector math library
  - Consider substituting SVML functions

```
void kernel(float* __restrict__ in_out_ptr0,
            const float* __restrict__ in_ptr0,
            float* __restrict__ out_ptr0,
            float* __restrict__ out_ptr1)
```

```
bool throw_exception = false;
#pragma omp parallel shared(throw_exception)
     {
          . . .
          #pragma omp for
          for(int64_t x0=static_cast<int64_t>(
               x0<static_cast<int64_t>(16352L);
               x0+=static_cast<int64_t>(1L))
          {
               // code computing tmp11

               if (!((0 <= tmp11) & (tmp11 < 30
                  throw_exception = true;
                  #pragma omp cancel for
               }
```

```
   Vectorized<float> exp() const {
-    return Vectorized<float>(Sleef_expf8_u10(values));
+    return Vectorized<float>(_mm256_exp_ps(values));
   }
```

Restructuring Inefficient Loops

- Inefficient loop structure
  - Scalar remainder Loop is inside x1-loop, which can be hoisted out
  - Range checks (executed 72K times) are inside x1-loop, which can be removed

Before Loop Restructuring

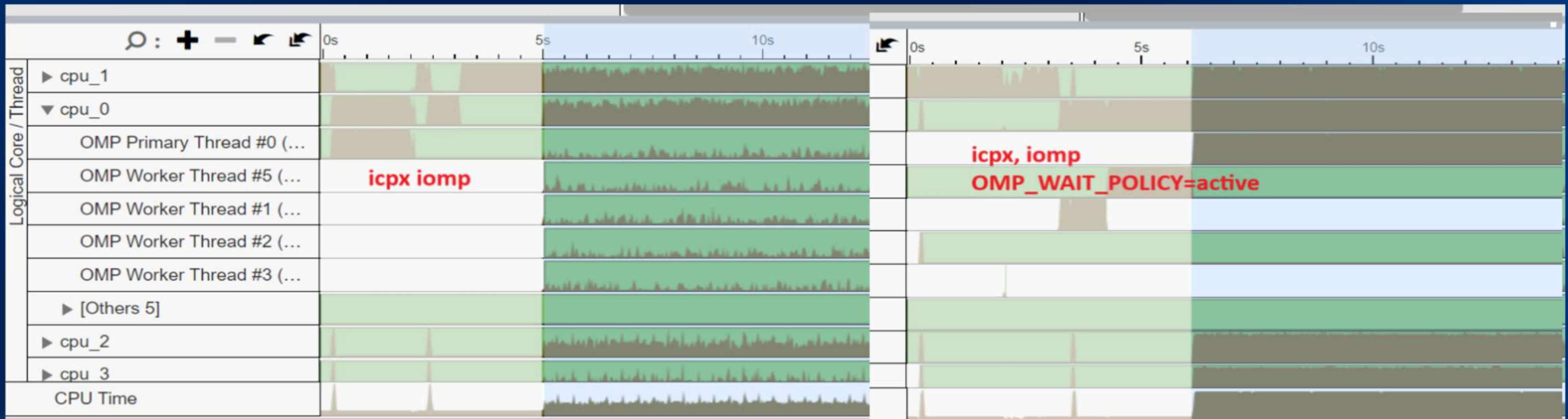
```
for(int64_t x0 = static_cast<int64_t>(0L);
            x0 < static_cast<int64_t>(2048L);
            x0 += static_cast<int64_t>(1L)){
  for(int64_t x1 = static_cast<int64_t>(0L);
              x1 < static_cast<int64_t>(36L);
              X1 += static_cast<int64_t>(16L)){
      if(C10_LIKELY(x1 >= static_cast<int64_t>(0) &&
                    x1 < static_cast<int64_t>(32L))){
        // Main Loop body - vectorized
      }
      if(C10_UNLIKELY(x1 >= static_cast<int64_t>(32L) &&
                      x1 < static_cast<int64_t>(36L))){
        for (int64_t x1_tail = static_cast<int64_t>(32L);
                     x1_tail < static_cast<int64_t>(36L);
                     x1_tail++){
          // Remainder Loop body - scalar
        }
      }
   } // end x1-loop
} // end x0-loop
```

After Loop Restructuring

```
for(int64_t x0=static_cast<int64_t>(0L);
            x0<static_cast<int64_t>(2048L);
            x0+=static_cast<int64_t>(1L)) {
  for(int64_t x1=static_cast<int64_t>(0L);
              x1<static_cast<int64_t>(32L);
              x1+=static_cast<int64_t>(16L)) {
    // Main Loop body - vectorized
  }
  for(int64_t x1_tail = static_cast<int64_t>(32L);
              x1_tail < static_cast<int64_t>(36L);
              x1_tail++){
    // Remainder Loop body - scalar
  }
}
```

BKMs for CPU Utilization Improvements

- Set system power policy to performance
- LD_PRELOAD=iomp (preload Intel OpenMP runtime, prevent load of gomp etc.)
- OMP_WAIT_POLICY=active (keep thread active)
  - Use Intel® VTune™ Profiler to discover CPU Utilization issue and verify tuning results
- KMP_BLOCKTIME=1 (keep thread active)
- KMP_AFFINITY=granularity=compact,1 or OMP_PROC_BIND=true OMP_PLACES=cores
  - pin thread, not recommend for hybrid CPUs

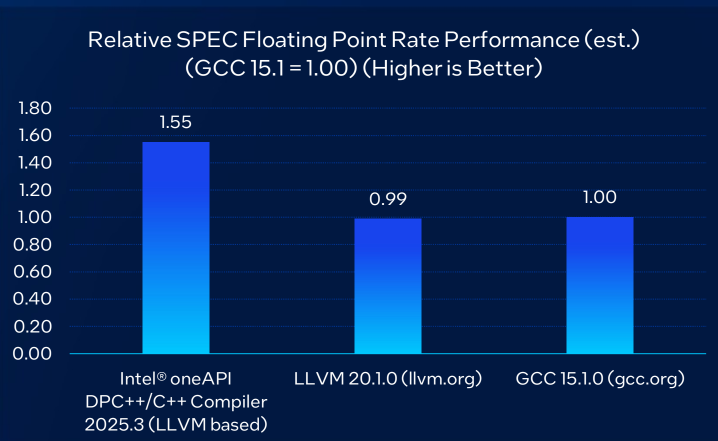


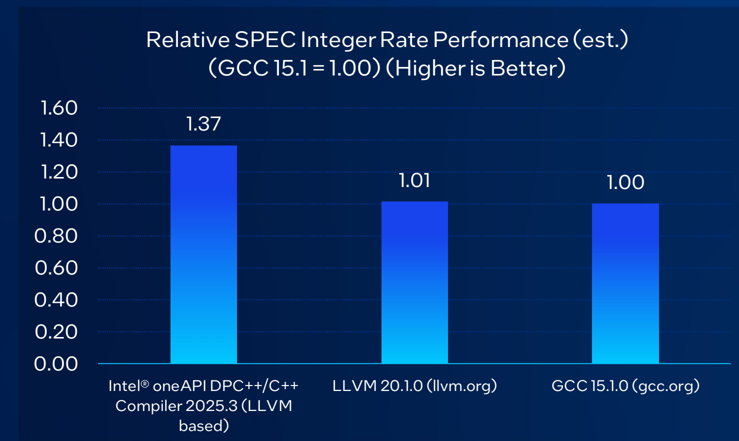
Performance on Xeon® 6 and Panther Lake Processor

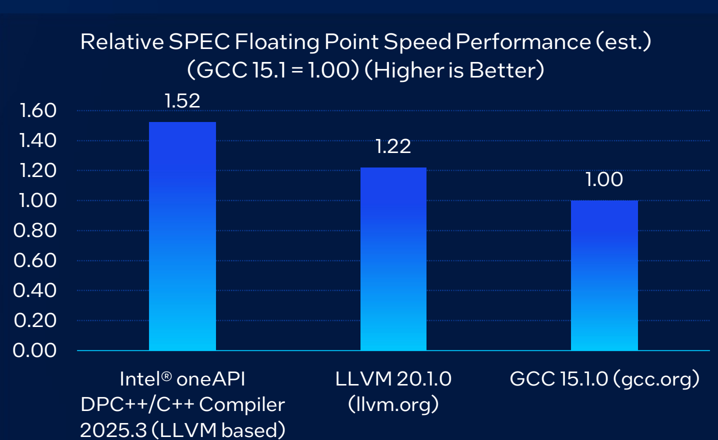
intel

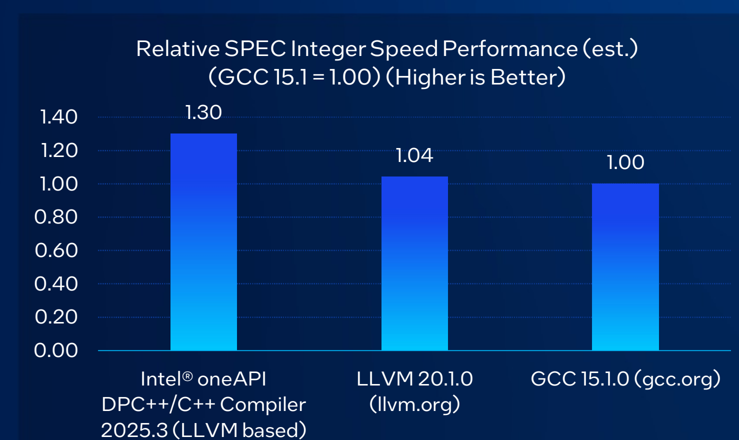
Compiler Performance on Intel® Xeon® 6 Processor

- Up to **1.55x more performance** for SPEC* FP Rate
- Up to **1.37x more performance** for SPEC Integer Rate
- Up to **1.52x more performance** for SPEC FP Speed
- Up to **1.30x more performance** for SPEC Integer Speed



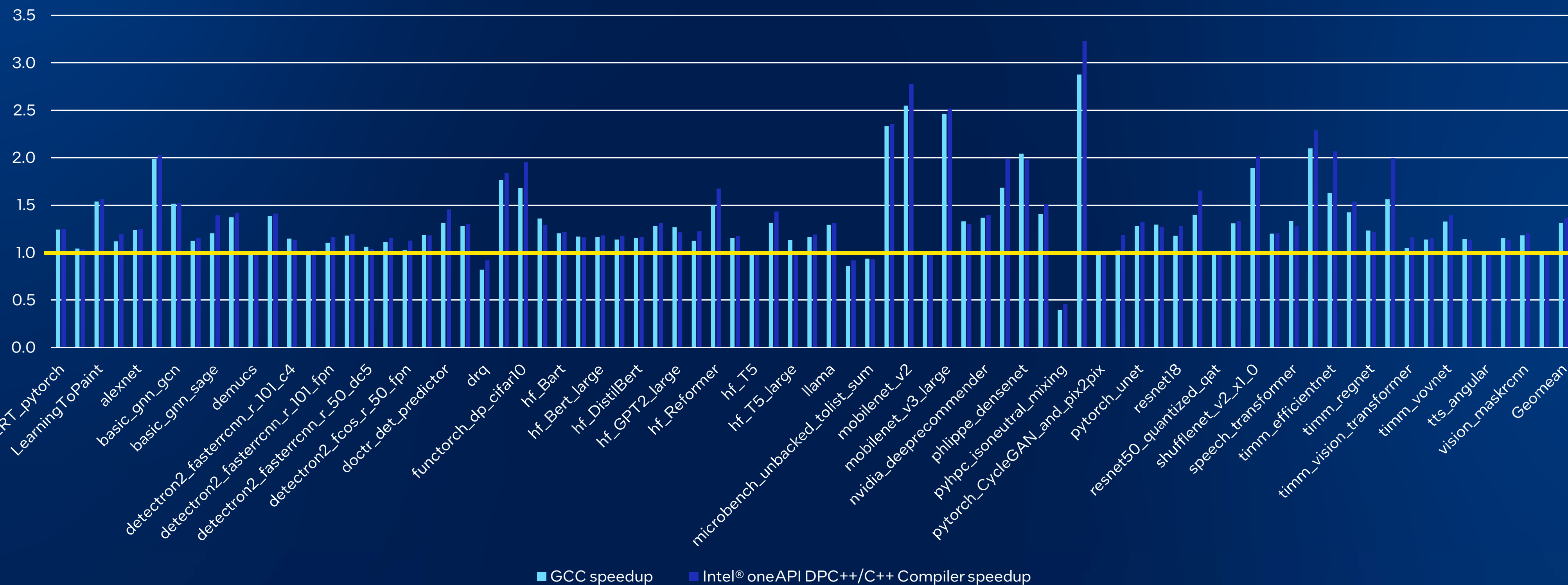






TorchBench Performance on Intel® Panther Lake Processors (Linux*)

- Torchbench* Performance Speedup with PyTorch* Version 2.9 (Higher is Better)



Testing Date: Performance results are **based on testing by Intel as of Oct 20, 2025** and may not reflect all publicly available security updates
Configuration Detail and Workload Setup: Panth Lake (Beta) with 32 GB total memory (2.20GHz, 10 Cores, 10 threads, 7467 MT/s). Turbo on, Ubuntu 24.04 (Kernel 6.13.0). Intel® oneAPI DPC++/C++ Compiler 2025.3.0, GCC 13.2.0 (Ubuntu 13.2.0-23ubuntu4), Python 3.11, PyTorch 2.9.0+cpu, Torchbench 819473f.
Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See configuration disclosure for details. No product or component can be absolutely secure.
Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex.
AI features may require software purchase, subscription or enablement by a software or platform provider, or may have specific configuration or compatibility requirements. Details at www.intel.com/AIPC. Results may vary.

intel

Key Take Away

Intel Compilers, GCC and Clang (LLVM) compilers are enabled and tuned for Intel® Xeon 6, Panther Lake and Intel® Xeon 6+ (Clearwater Forest) Processors

Leverage the latest Intel® oneAPI DPC++/C++ Compiler 2026.0, and Intel ® Fortran Compiler 2026.0 to unleash the power of Intel Processors

Call for Action

... have a rich history of innovation

... are based on open standards

... give you choice

Join us in driving the CPU programming ecosystem

Join us in creating an open GPU programming ecosystem

Use the tools you know and love on today's CPUs, integrated & discrete GPUs

From supercomputers to laptops!

intel

intel