The Journal of Open Source Software

Boost.Int128: A C++14 Library for Portable and Performant 128-Bit Integer Arithmetic

Matt Borland [1]

1 The C++ Alliance, United States

DOI: 10.21105/joss.11293

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @Deln0r
- @GuilloteauQ

Submitted: 21 August 2026
Published: 23 September 2026

Summary

Boost.Int128 is a header-only, C++14 implementation of 128-bit integer arithmetic through two types, `int128` and `uint128`, that behave like built-in integers and integrate with the C++ Standard Library and other Boost libraries (Boost Authors, 2026). Both types are exactly 128 bits wide on every supported platform, and both are usable unchanged in GPU kernels through CUDA (Nickolls et al., 2008) and SYCL (Reyes et al., 2020). The library is available at https://github.com/boostorg/int128, and in the Boost distribution starting with version 1.93.

Statement of need

C++ specifies no integer type wider than `long long`, which in practice means 64 bits, yet quantities exceeding that range are common. An IPv6 address (Hinden & Deering, 2006) and a UUID (Davis et al., 2024) are each 128 bits, so each needs a 128-bit integer or a byte array that cannot be compared, masked, or incremented directly. The widening 64x64 to 128-bit product is the central primitive of unbiased random integer generation in an interval (Lemire, 2019) and of shortest round-trip floating-point formatting (Adams, 2018). GCC and Clang expose `__int128` only on 64-bit targets, with no `std::numeric_limits`, `<iostream>`, or `std::hash` support, and MSVC lacks it entirely, so dependent code either restricts its portability or maintains a second implementation. The portable alternative, a general multiprecision integer, pays for that generality in object size and in throughput. The types began as the arithmetic backend of Boost.Decimal (Borland & Kormanyos, 2026), which needs an exact 128-bit product on every platform Boost supports, and were extracted and generalized once that need proved broader than one library. What is needed is a type that is exactly 16 bytes everywhere, as fast as the compiler's extension where that exists, and usable across the host-to-device boundary without an interchange format.

State of the field

The portable host implementations are Abseil's `absl::int128` and `absl::uint128` (Google, 2017), Boost.Multiprecision's 128-bit `cpp_int` instantiation (Kormanyos & Maddock, 2026), and MSVC's undocumented `std::_Signed128` and `std::_Unsigned128` (Microsoft Corporation, 2026). Abseil is a close functional match, but it requires C++17 and GCC 10 or later and carries no device annotations. Boost.Multiprecision's types instantiate a general `number` template rather than a purpose-built class, so they carry an extra word of bookkeeping and cost more per operation.

On the device side, `nvcc` accepts `__int128` only when the host compiler supports it (NVIDIA Corporation, 2026), which excludes every MSVC and 32-bit host, and SYCL's `spir64` target has no native 128-bit integer at all (Khronos SYCL Working Group, 2023). The implementations

filling this gap, such as CUDA-uint128 (Seizert, 2016), are unsigned only and tied to one toolchain. We are not aware of another implementation offering one 128-bit type usable on the host and in both CUDA and SYCL device code with bitwise identical results.

Software design

The library requires only C++14, is header-only, and has no external dependencies, so a user need only `#include <boost/int128.hpp>`, even with a toolchain as old as `g++-5`. A single-header amalgamation is published, and under C++20 the library can be consumed as a module via `import boost.int128`. Relaxed C++14 `constexpr` covers nearly the whole interface.

Each type is a struct of two `std::uint64_t` words ordered by target endianness, so the object representation and alignment match a native 128-bit integer. Keeping both words unsigned lets the compiler treat the pair as one wide access, so loops over these types vectorize rather than scalarize. Operators forward to a native 128-bit type or intrinsic where one exists, and follow Knuth (1998) and Warren (2013) over the two words elsewhere, with identical results. The library also supplies the interfaces the extension lacks, among them `<limits>`, `<bit>`, `<charconv>`, `<format>`, `<iostream>`, hashing, Boost.Random and Boost.Math integration, saturating arithmetic, the C23 checked-arithmetic interface (ISO/IEC, 2024), and the `div_*` family of P3724 (Schultke, 2025).

Continuous integration runs the test suite natively on Linux, macOS, and Windows across x86_64, x86_32, aarch64, ARM32v7, ARM64, and big-endian s390x, plus PPC64LE under QEMU, using GCC 5 and later, Clang 5 and later, Visual Studio 2017 and later, Intel oneAPI DPC++, and NVCC. Because half of these targets have no native 128-bit type, the software fallbacks are tested as thoroughly as the intrinsic paths. GPU support is expressed through one annotation macro, `BOOST_INT128_HOST_DEVICE`, which expands to `__host__ __device__` under `nvcc` and `SYCL_EXTERNAL` under a SYCL compiler. More than eighty CUDA and eighty SYCL tests compare device results against the host, so agreement across that boundary is tested rather than assumed.

Where a native type exists, Boost.Int128 matches or beats it: with GCC 14.2 on Linux x86_64, `uint128` division and modulo are roughly 16 percent faster than `unsigned __int128`, and signed multiplication is 1.5 times faster on big-endian s390x. Where no native type exists, the margins are larger: signed multiplication is 1.7 times faster than `std::_Signed128` on MSVC x86_64 and 6.2 times faster on MSVC ARM64, and 32-bit x86 addition is 8 times faster than Boost.Multiprecision. A handful of cells favor an alternative, so the complete tables, rerun on every pull request, are published rather than a summary, at https://develop.int128.cpp.al/int128/u128_benchmarks.html and https://develop.int128.cpp.al/int128/i128_benchmarks.html. To enable reproducibility, information on how to run the benchmarks locally as well as run our provided analysis scripts can be found on the same benchmark pages under the "Running the Benchmarks" heading.

Figure 1 shows the difference between a 64-bit and a 128-bit product.

```cpp
#include <boost/int128.hpp>
#include <cstdint>
#include <iostream>

int main()
{
    using boost::int128::uint128;

    // Two values that each fit in 64 bits, but whose product does not
    constexpr std::uint64_t a {11400714819323198485ULL};
    constexpr std::uint64_t b {14029467366897019727ULL};
```

```
    // A 64-bit product silently wraps modulo 2^64
    std::cout << " 64-bit: a * b = " << a * b << std::endl;

    // The same multiplication is exact when the result is 128 bits wide
    constexpr uint128 exact {uint128{a} * uint128{b}};
    std::cout << "128-bit: a * b = " << exact << std::endl;

    // Like a built-in integer, the types work in constant expressions
    static_assert(exact / uint128{b} == uint128{a}, "Recovers a exactly");
    static_assert(sizeof(uint128) == 16, "Exactly 128 bits on every platform");

    return 0;
}
```

```
 64-bit: a * b = 17693923505768731003
128-bit: a * b = 159945956516994065446882290339781513595
```

Figure 1: The 64-bit product wraps modulo 2^64 while the `uint128` product is exact and compile-time checked. The lower block is the program output.

Research Impact Statement

As a component distributed through the Boost libraries, Boost.int128 is trivially in the reach of researchers everywhere.

Boost.int128 serves as the arithmetic backend to the Boost Decimal Library (Borland & Kormanyos, 2026). Using Boost.int128 is what allowed Boost.Decimal to be portable beyond the platforms served by competing libraries, for example, on Apple's ARM64. It also allowed portability to CUDA, enabling quantitative finance researchers to process data at much larger scales than previously available.

The Boost Graph Library (Siek et al., 2002) already relies on 128-bit integers for exact geometry, using `int128` in is_straight_line_drawing to compute overflow-free orientation determinants, showing that a dedicated Boost.Int128 would directly serve computational-geometry and planarity research with applications in VLSI circuit layout, graph visualization, network/map drawing, and GIS.

AI usage disclosure

Generative AI tools were used in developing this library and preparing this manuscript, for drafting and editing prose, platform-specific optimizations, and generating test cases. All library code, benchmark results, and text were reviewed and verified by the author, who takes full responsibility for their correctness.

Acknowledgements

We thank the C++ Alliance for sponsoring this work, the domain experts who peer-reviewed the library in July 2026 before its acceptance into the Boost library collection, and Arnaud Becheler for managing that review.

References

Adams, U. (2018). Ryu: Fast float-to-string conversion. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*, 270–282. https://doi.org/10.1145/3192366.3192369

Boost Authors. (2026). Boost C++ libraries. In *GitHub repository*. GitHub. https://github.com/boostorg/boost

Borland, M., & Kormanyos, C. (2026). Boost.Decimal: A C++14 library for decimal floating point arithmetic. *Journal of Open Source Software*, *11*(124), 10345. https://doi.org/10.21105/joss.10345

Davis, K. R., Peabody, B., & Leach, P. J. (2024). *Universally unique IDentifiers (UUIDs)* (RFC 9562). Internet Engineering Task Force. https://doi.org/10.17487/RFC9562

Google. (2017). Abseil common libraries. In *GitHub repository*. GitHub. https://github.com/abseil/abseil-cpp

Hinden, R. M., & Deering, S. E. (2006). *IP version 6 addressing architecture* (RFC 4291). Internet Engineering Task Force. https://doi.org/10.17487/RFC4291

ISO/IEC. (2024). *ISO/IEC 9899:2024, Programming Languages - C*. International Organization for Standardization. https://www.iso.org/standard/82075.html

Khronos SYCL Working Group. (2023). *SYCL 2020 Specification*. The Khronos Group. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html

Knuth, D. E. (1998). *The art of computer programming, volume 2: Seminumerical algorithms* (3rd ed.). Addison-Wesley. ISBN: 9780201896848

Kormanyos, C., & Maddock, J. (2026). Boost.Multiprecision. In *GitHub repository*. GitHub. https://github.com/boostorg/multiprecision

Lemire, D. (2019). Fast random integer generation in an interval. *ACM Transactions on Modeling and Computer Simulation*, *29*(1), 1–12. https://doi.org/10.1145/3230636

Microsoft Corporation. (2026). Microsoft C++ standard library: __msvc_int128.hpp. In *GitHub repository*. GitHub. https://github.com/microsoft/STL/blob/main/stl/inc/__msvc_int128.hpp

Nickolls, J., Buck, I., Garland, M., & Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue*, *6*(2), 40–53. https://doi.org/10.1145/1365490.1365500

NVIDIA Corporation. (2026). *CUDA C++ programming guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/

Reyes, R., Brown, G., Burns, R., & Wong, M. (2020). SYCL 2020: More than meets the eye. *Proceedings of the International Workshop on OpenCL (IWOCL '20)*. https://doi.org/10.1145/3388333.3388649

Schultke, J. (2025). *P3724: Integer division*. ISO/IEC JTC1/SC22/WG21. https://wg21.link/p3724

Seizert, C. (2016). CUDA-uint128: A 128 bit unsigned integer class for CUDA. In *GitHub repository*. GitHub. https://github.com/curtisseizert/CUDA-uint128

Siek, J. G., Lee, L.-Q., & Lumsdaine, A. (2002). *The Boost Graph library: User guide and reference manual*. Addison-Wesley. ISBN: 9780201729146

Warren, H. S. (2013). *Hacker's delight* (2nd ed.). Addison-Wesley. ISBN: 9780321842688