oneAPI

SINGLE PROGRAMMING MODEL TO DELIVER CROSS-ARCHITECTURE PERFORMANCE

Henry A Gabb, Principal Engineer
Intel Architecture, Graphics, and Software
November, 2019
**DIVERSE WORKLOADS REQUIRE DIVERSE ARCHITECTURES**

The future is a *diverse* mix of scalar, vector, matrix, and spatial architectures deployed in CPU, GPU, AI, FPGA and other accelerators.
Diverse set of data-centric hardware

No common programming language or APIs

Inconsistent tool support across platforms

Each platform requires unique software investment
Project oneAPI delivers a unified programming model to simplify development across diverse architectures.

- Common developer experience across Scalar, Vector, Matrix and Spatial architectures (CPU, GPU, AI and FPGA)
- Uncompromised native high-level language performance
- Based on industry standards and open specifications
LOW LEVEL INTERFACE, LEVERAGED BY OTHERS

Industry Data-Centric Applications

Industry Middleware/Frameworks

oneAPI Language & Libraries

CPU
GPU
AI
FPGA

Millions
Thousands
ONEAPI FOR CROSS-ARCHITECTURE PERFORMANCE

Optimized Applications

Optimized Middleware & Frameworks

oneAPI Product

Direct Programming

API-Based Programming

Data Parallel C++

Libraries

Analysis &
Debug Tools

CPU

SCALAR

GPU

VECTOR

AI

MATRIX

FPGA

SPATIAL

Some capabilities may differ per architecture.

INTEL@SC19
Language to deliver uncompromised parallel programming productivity and performance across CPUs and accelerators

Based on C++ with language enhancements being driven through community project

Open, cross-industry alternative to single-architecture proprietary language
```cpp
#include <CL/sycl.hpp>
#include <iostream>

void main() {
    using namespace cl::sycl;
    float A[1024], B[1024], C[1024];
    {
        buffer<
            size_t, 1>
            bufA { A, range<1> {1024} };
        buffer<
            size_t, 1>
            bufB { B, range<1> {1024} };
        buffer<
            size_t, 1>
            bufC { C, range<1> {1024} };

        queue myQueue;
        myQueue.submit([&](handler& cgh) {
            auto A = bufA.get_access<access::read>(cgh);
            auto B = bufB.get_access<access::read>(cgh);
            auto C = bufC.get_access<access::write>(cgh);

            cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

DPC++ “Hello world”: Vector Addition

Single-source programming model

Based on C++11 and SYCL open specification from Khronos
```cpp
#include <CL/sycl.hpp>
#include <iostream>

void main() {
    using namespace cl::sycl;
    float A[1024], B[1024], C[1024];
    {
        buffer<
            size_t,
            1
        > bufA { A, range<
            1
        > {1024} };
        buffer<
            size_t,
            1
        > bufB { B, range<
            1
        > {1024} };
        buffer<
            size_t,
            1
        > bufC { C, range<
            1
        > {1024} };

        queue myQueue;
        myQueue.submit([&](handler& cgh) {
            auto A = bufA.get_access<
                access::read
            >(cgh);
            auto B = bufB.get_access<
                access::read
            >(cgh);
            auto C = bufC.get_access<
                access::write
            >(cgh);

            cgh.parallel_for<class vector_add>(range<
                1
            > {1024}, [=](id<
                1
            > i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = \" << C[i] << std::endl;
}```
```cpp
#include <CL/sycl.hpp>
#include <iostream>

void main() {
    using namespace cl::sycl;
    float A[1024], B[1024], C[1024];

    buffer<size_t, 1> bufA { A, range<1> {1024} };
    buffer<size_t, 1> bufB { B, range<1> {1024} };
    buffer<size_t, 1> bufC { C, range<1> {1024} };

    queue myQueue;
    myQueue.submit([&](handler &cgh) { 
        auto A = bufA.get_access<access::read>(cgh);
        auto B = bufB.get_access<access::read>(cgh);
        auto C = bufC.get_access<access::write>(cgh);

        cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
            C[i] = A[i] + B[i];
        });
    });

    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Create a queue to submit work to a device (including host).

#include <CL/sycl.hpp>
#include <iostream>

void main() {
    using namespace cl::sycl;
    float A[1024], B[1024], C[1024];
    {
        buffer<size_t, 1> bufA { A, range<1> {1024} };
        buffer<size_t, 1> bufB { B, range<1> {1024} };
        buffer<size_t, 1> bufC { C, range<1> {1024} };

        queue myQueue;
        myQueue.submit([&](handler& cgh) {
            auto A = bufA.get_access<access::read>(cgh);
            auto B = bufB.get_access<access::read>(cgh);
            auto C = bufC.get_access<access::write>(cgh);

            cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```cpp
#include <CL/sycl.hpp>
#include <iostream>

void main() {
    using namespace cl::sycl;
    float A[1024], B[1024], C[1024];
    {
        buffer<size_t, 1> bufA { A, range<1> {1024} };
        buffer<size_t, 1> bufB { B, range<1> {1024} };
        buffer<size_t, 1> bufC { C, range<1> {1024} };

        queue myQueue;
        myQueue.submit([&](handler& cgh) {
            auto A = bufA.get_access<access::read>(cgh);
            auto B = bufB.get_access<access::read>(cgh);
            auto C = bufC.get_access<access::write>(cgh);

            cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Vector addition kernel enqueues a parallel_for task.
```cpp
#include <CL/sycl.hpp>
#include <iostream>

void main() {
    using namespace cl::sycl;
    float A[1024], B[1024], C[1024];
    {
        buffer<size_t, 1> bufA { A, range<1> {1024} };  
        buffer<size_t, 1> bufB { B, range<1> {1024} };  
        buffer<size_t, 1> bufC { C, range<1> {1024} };  

        queue myQueue;
        myQueue.submit([&](handler& cgh) {
            auto A = bufA.get_access<access::read>(cgh);
            auto B = bufB.get_access<access::read>(cgh);
            auto C = bufC.get_access<access::write>(cgh);

            cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }

    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
```

Implicit barrier?
#include <CL/sycl.hpp>
#include <iostream>

void main() {
  using namespace cl::sycl;
  float A[1024], B[1024], C[1024];
  {
    buffer<size_t, 1> bufA { A, range<1> {1024} };
    buffer<size_t, 1> bufB { B, range<1> {1024} };
    buffer<size_t, 1> bufC { C, range<1> {1024} };

    queue myQueue;
    myQueue.submit([&](handler& cgh) {
      auto A = bufA.get_access<access::read>(cgh);
      auto B = bufB.get_access<access::read>(cgh);
      auto C = bufC.get_access<access::write>(cgh);

      cgh.parallel_for<class vector_add>(range<1> {1024}, [=](id<1> i) {
        C[i] = A[i] + B[i];
      });
    });
  }
  for (int i = 0; i < 1024; i++)
    std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
#include <CL/sycl.hpp>
#include <iostream>

void main() {
    using namespace cl::sycl;
    float A[1024], B[1024], C[1024];
    {
        buffer<
            size_t,
            1>
            bufA { A,
                range<
                    1> {1024} };
        buffer<
            size_t,
            1>
            bufB { B,
                range<
                    1> {1024} };
        buffer<
            size_t,
            1>
            bufC { C,
                range<
                    1> {1024} };

        queue myQueue;
        myQueue.submit([&](handler& cgh) {
            auto A = bufA.get_access<access::read>(cgh);
            auto B = bufB.get_access<access::read>(cgh);
            auto C = bufC.get_access<access::write>(cgh);

            cgh.parallel_for<class vector_add>(range<
                1> {1024}, [=](id<
                    1> i) {
                C[i] = A[i] + B[i];
            });
        });
    }
    for (int i = 0; i < 1024; i++)
        std::cout << "C[" << i << "] = " << C[i] << std::endl;
}
Graph of Asynchronous Execution

DPC++ queues are out-of-order by default. Data dependencies order kernel executions.

```cpp
myQueue.submit([&](handler& cgh) {
    auto A = a.get_access<access::mode::read>(cgh);
    auto B = b.get_access<access::mode::read>(cgh);
    auto C = c.get_access<access::mode::discardwrite>(cgh);
    cgh.parallel_for<class add1>(range<2>{N, M},
                               [=](id<2> index) { C[index] = A[index] + B[index]; });
});

myQueue.submit([&](handler& cgh) {
    auto A = a.get_access<access::mode::read>(cgh);
    auto C = c.get_access<access::mode::read>(cgh);
    auto D = d.get_access<access::mode::write>(cgh);
    cgh.parallel_for<class add2>(range<2>{P, Q},
                                 [=](id<2> index) { D[index] = A[index] + C[index]; });
});

myQueue.submit([&](handler& cgh) {
    auto A = a.get_access<access::mode::read>(cgh);
    auto D = d.get_access<access::mode::write>(cgh);
    auto E = e.get_access<access::mode::write>(cgh);
    cgh.parallel_for<class add3>(range<2>{S, T},
                                 [=](id<2> index) { E[index] = A[index] + D[index]; });
});
```
Domain-specific functions to accelerate compute intensive workloads

Custom-coded for uncompromised performance on SVMS (Scalar, Vector, Matrix, Spatial) architectures
Productive performance analysis across SVMS architectures

Performance Profiler
Parallelization Assistant
Debugger
ECOSYSTEM ADOPTION & SUPPORT

Drive adoption of Data Parallel C++ language & oneAPI library APIs

- Developer Enabling Programs
- Support
- Ecosystem Engagement
Diverse workloads for data-centric computing are driving the need for diverse compute architectures including CPUs, GPUs, FPGAs, and AI accelerators.

OneAPI unifies and simplifies programming of Intel CPUs and accelerators, delivering developer productivity and full native language performance.

OneAPI is based on industry standards and open specifications to encourage ecosystem collaboration and innovation.
NEW ANNOUNCEMENTS SUNDAY!
INTEL DEVELOPER CONFERENCE

Please attend Raja Koduri’s keynote
... followed by our Intel Networking Reception

• Raja Koduri is the Senior Vice President, Chief Architect, and General Manager, Intel Architecture, Graphics, and Software
• One Intel Station | Ellie Caulkins Opera House
  ✓ 1385 Curtis Street, Denver
• Sunday, November 17 | 4:00pm – 5:00pm
• Register at https://tinyurl.com/IntelDevcon

INTEL@SC19
NOTICES & DISCLAIMERS

Intel technologies’ features and benefits depend on system configuration and may require enabled hardware, software or service activation. Performance varies depending on system configuration.

No product or component can be absolutely secure.

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. For more complete information about performance and benchmark results, visit http://www.intel.com/benchmarks..

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit http://www.intel.com/benchmarks..

Intel Advanced Vector Extensions (Intel AVX) provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Cost reduction scenarios described are intended as examples of how a given Intel-based product, in the specified circumstances and configurations, may affect future costs and provide cost savings. Circumstances will vary. Intel does not guarantee any costs or cost reduction.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data are accurate.

© 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.