Open Approaches to Heterogeneous Programming are Key for Surviving the New Golden Age of Computer Architecture

James Reinders, engineer
May 2022

2022

intel®

Heterogeneous Systems – Programming them

My talk today:

1. Heterogeneous Systems are here to stay
   and will be ubiquitous (like parallelism)

2. Standardizing support is HARD, and we keep getting it WRONG

3. Look at the essentials of SYCL (this is just for C++)

4. We need
   open, multivendor, multiarchitecture support
   that spans programming languages

5. This is OUR problem – let's solve it together

2022: 25th Anniversary ASCI Red supercomputer takes #1 spot

#1 system for seven Top 500 lists (still a record) - from June 1997 through June 2000

- First TeraFLOP/s computer in the world.
- 7264 processors (cores) of Intel Pentium Pro processors @200MHz for 1.45 TeraFLOP/s. Later upgraded to 9632 Pentium II Over-Drive processors @333MHz for 3.21 TeraFLOP/s.
- Parallel programming focused on distributed parallelism (message passing)

2022: 25th Anniversary ASCI Red supercomputer takes #1 spot

#1 system for eight Top 500 lists (still a record) - from June 1997 through June 2000

- First TeraFLOP/s computer in the world.
- 7264 processors (cores) of Intel Pentium Pro processors @200MHz for 1.45 TeraFLOP/s. Later upgraded to 9632 Pentium II Over-Drive processors @333MHz for 3.21 TeraFLOP/s.
- Parallel programming focused on distributed parallelism (message passing)

What has happened in the 25 years since ?

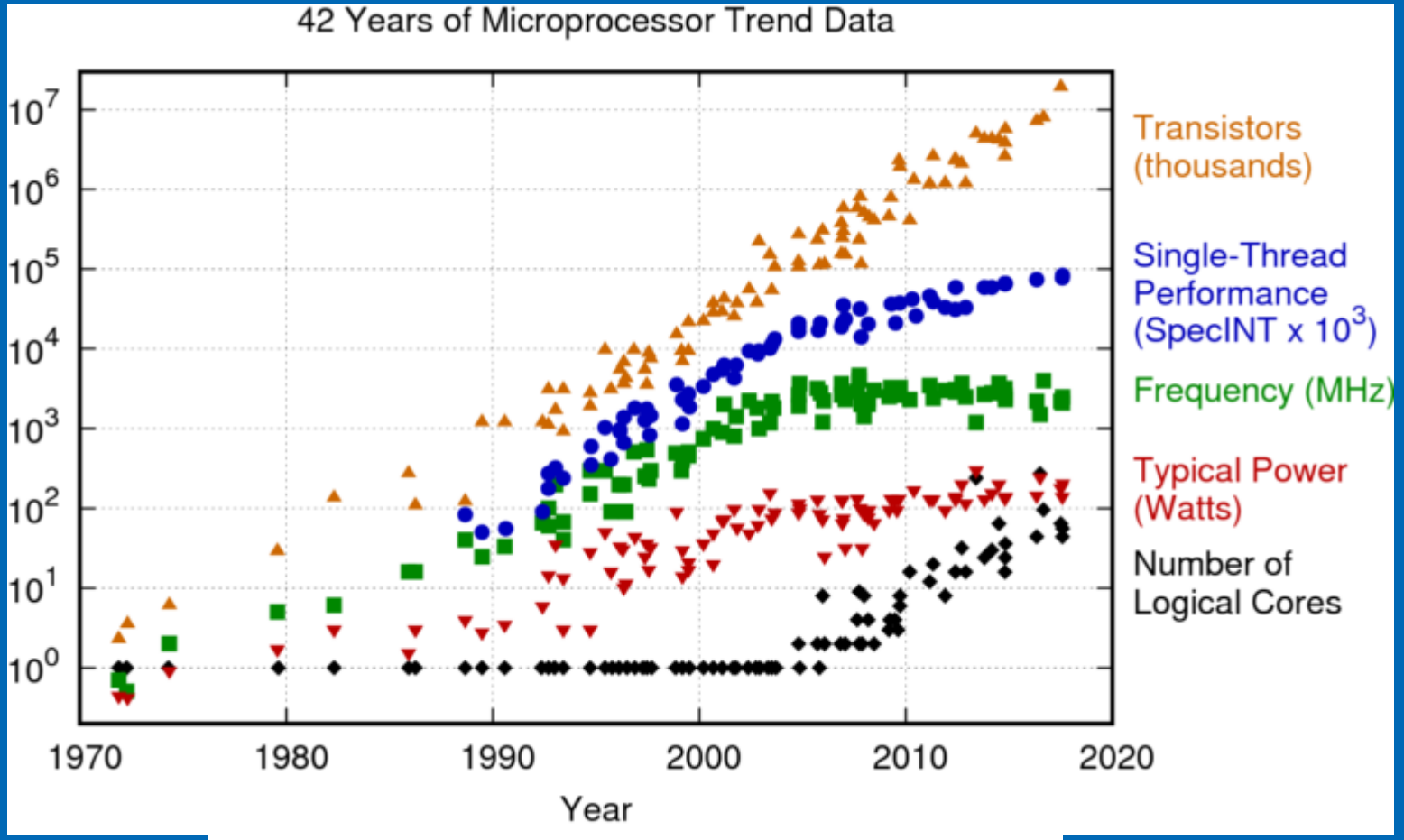
- "nodes" have become much fatter
  - Multicore, multisocket, and heterogeneous compute
- nodes require parallel programming of all kinds – distributed, share memory, offload

Heterogeneous Systems – Programming them

My talk today:

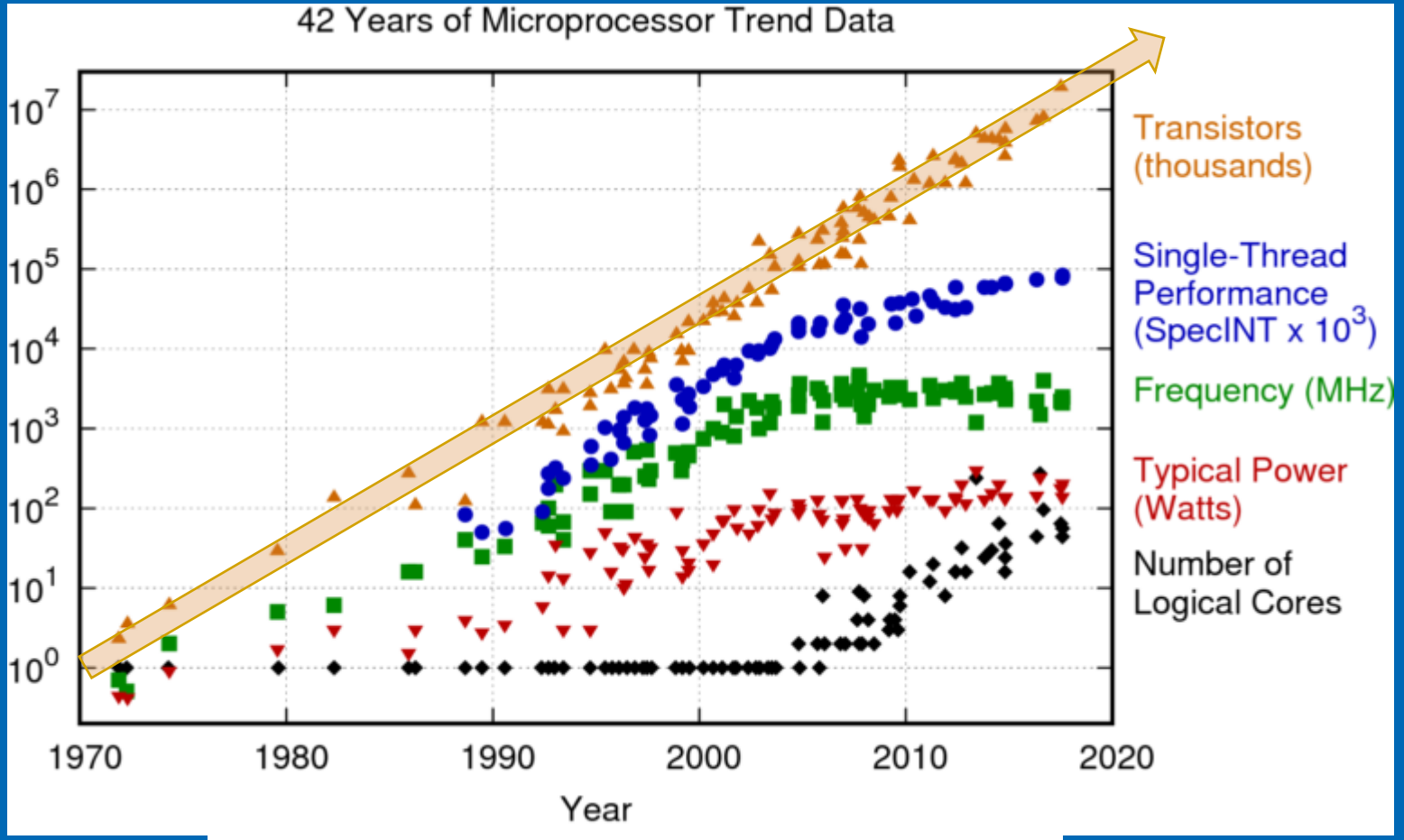
1. Heterogeneous Systems are here to stay
   and will be ubiquitous (like parallelism)
2. Standardizing support is HARD, and we keep getting it WRONG
3. Look at the essentials of SYCL (this is just for C++)
4. We need
   open, multivendor, multiarchitecture support
   that spans programming languages
5. This is OUR problem – let's solve it together

Our quest for more performance is eternal; how we obtain it adapts to the times



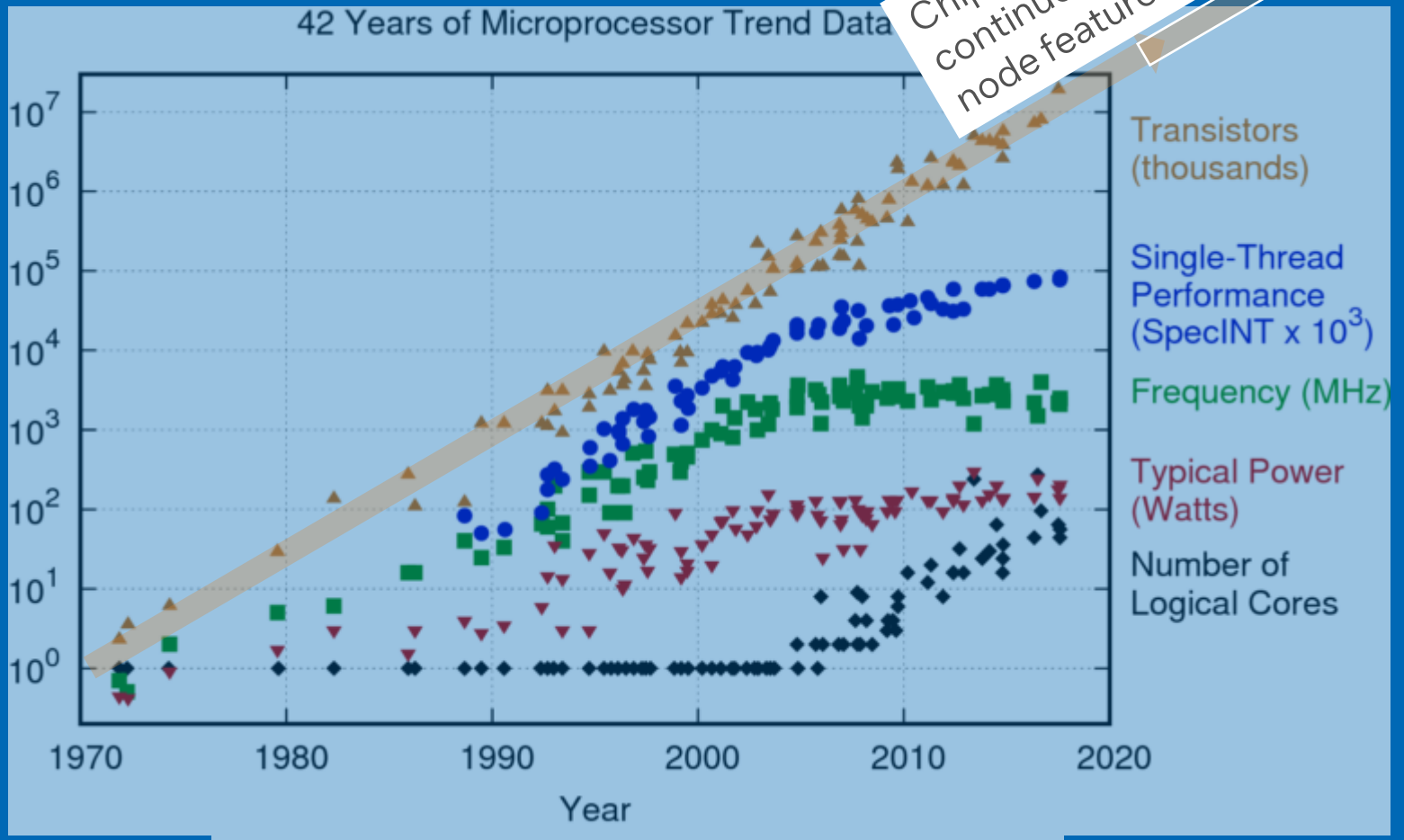
Source: tinyurl.com/karlruppdata (CC BY 4.0 license)

Our quest for more performance is eternal; how we obtain it adapts to the times



Source: tinyurl.com/karlruppdata (CC BY 4.0 license)

Our quest for more performance is eternal; how we obtain it adapts to the times



Source: tinyurl.com/karlruppdata (CC BY 4.0 license)

Computer trends: Parallel and Heterogeneous

Why Parallel?

Desire to get more work done, by having more workers.

Workers = compute units, devices, processing units, etc.
(e.g., CPU, GPU, FPGA, ASIC, AI chip)

Computer trends: Parallel and Heterogeneous

Why Parallel?

Desire to get more work done, by having more workers.

Why Heterogeneous?

Desire to get more work done, by having different types of workers.

Workers = compute units, devices, processing units, etc.
(e.g., CPU, GPU, FPGA, ASIC, AI chip)

Computer trends: Parallel and Heterogeneous

Why Parallel?
Desire to get more work done, by having more workers.

Why Heterogeneous?
Desire to get more work done, by having different types of workers.
And… well planned specialization can be more power efficient.

Workers = compute units, devices, processing units, etc.
(e.g., CPU, GPU, FPGA, ASIC, AI chip)

A New Golden Age for Computer Architecture

"The next decade will see a **Cambrian explosion of novel computer architectures**, meaning exciting times for computer architects in academia and industry."

ACM Turing Award laureates
John Hennessy and David Patterson (CACM, Feb 2019, Vol 62, No 2, pp 48-60)

https://tinyurl.com/HPcambrian <<< *HIGHLY RECOMMENDED READING*

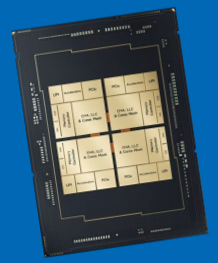
A New Golden Age for Computer Architecture

"The next decade will see a **Cambrian explosion of novel computer architectures**, meaning exciting times for computer architects in academia and industry."

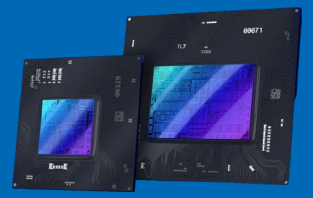
ACM Turing Award laureates
John Hennessy and David Patterson (CACM, Feb 2019, Vol 62, No 2, pp 48-60)

https://tinyurl.com/HPcambrian

Products



Novel On-Die Accelerators



GPU/Data Parallel

Spatial/ Dataflow

Deep Learning Optimized

Blockchain

UCIe
Universal Chiplet Interconnect Express

Mix & Match Nearly Endless Combinations

Research

Neuromorphic

???

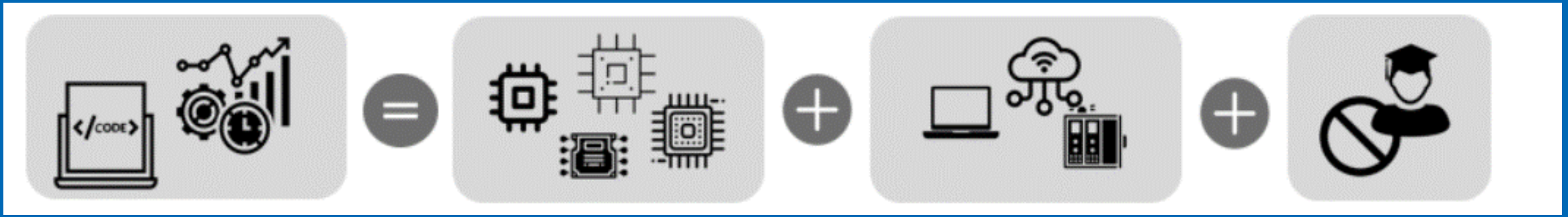
Graph Analytics and More...

the future must be

open, multivendor, multiarchitecture, multilanguage

the future must be

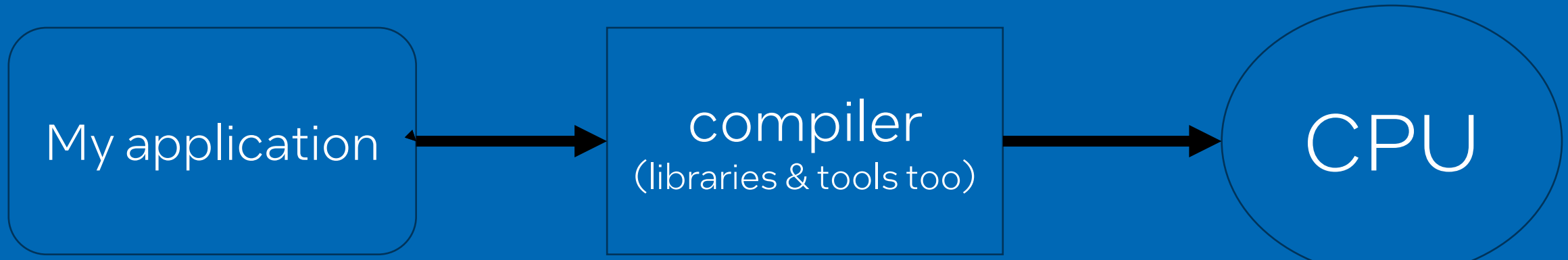
open, multivendor, multiarchitecture, multilanguage



Observation

- When a computer was homogeneous – we could program it with any tool, even if it was unique or proprietary.
- When a computer is heterogeneous – we need tools to work together.

Before heterogeneous systems



Portability was
a function
of the
language used.

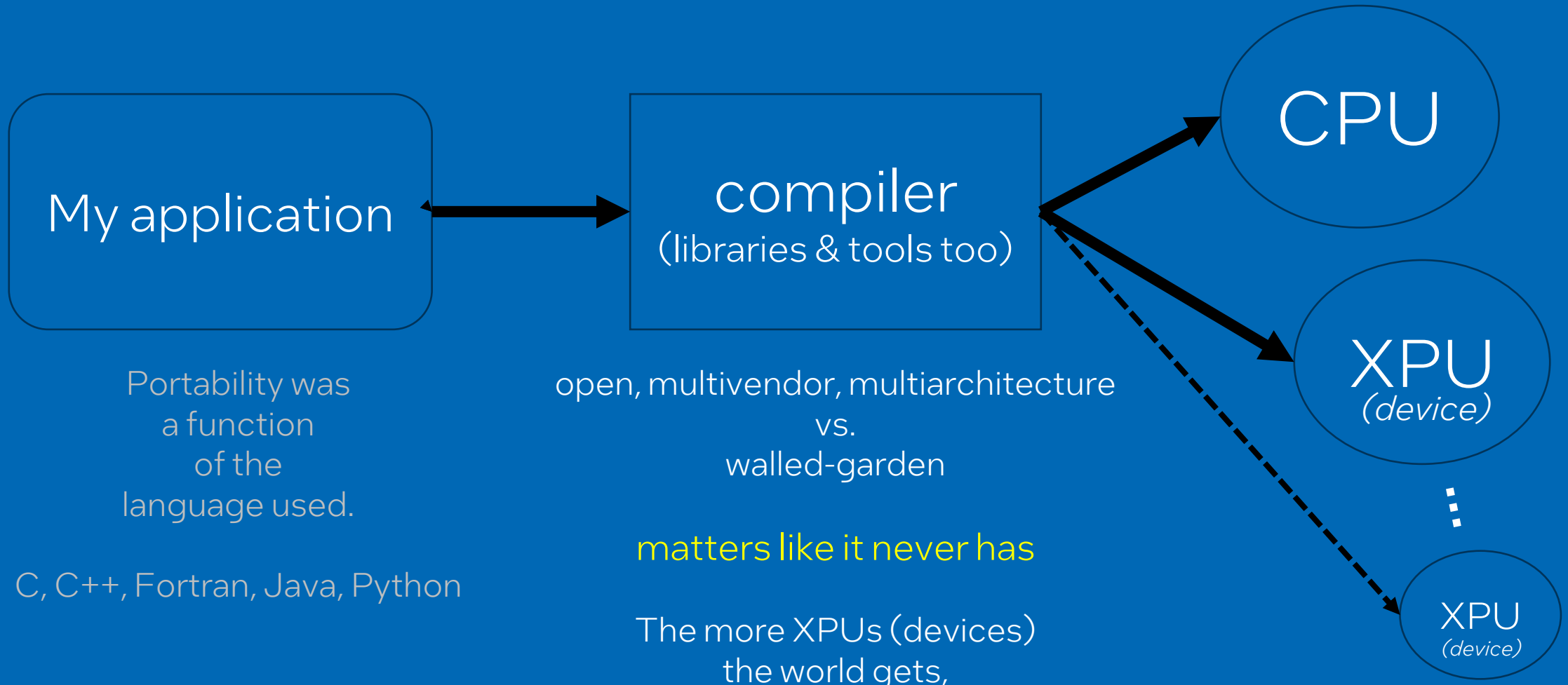
C, C++, Fortran, Java, Python

I didn't care if
the compiler, etc.,
was proprietary or not –
since the target system was
single vendor, single architecture.

Observation

- When a computer was homogeneous – we could program it with any tool, even if it was unique or proprietary.
- When a computer is heterogeneous – we need tools to work together.

Now, with heterogeneous systems



Portability was
a function
of the
language used.

C, C++, Fortran, Java, Python

open, multivendor, multiarchitecture
vs.
walled-garden

matters like it never has

The more XPUs (devices)
the world gets,
the more this matters.

Can we survive the diversity?

Do we have a choice?

make it much easier with
“open, multivendor, multiarchitecture”

A List of the…

Effective Programming of Heterogeneous Systems needs:

- be open, multivendor, and multiarchitecture – always
  - Pass three tests:
    1. Freedom to use any device (regardless of vendor or architecture)
    2. Ability to access maximum performance
    3. A future for my investments in coding
- support across many programming languages
- performance portability
- commonality for developers
- commonality under the covers

Heterogeneous Systems – Programming them

My talk today:

1. Heterogeneous Systems are here to stay
   and will be ubiquitous (like parallelism)
2. Standardizing support is HARD, and we keep getting it WRONG
3. Look at the essentials of SYCL (this is just for C++)
4. We need
   open, multivendor, multiarchitecture support
   that spans programming languages
5. This is OUR problem – let's solve it together

C++ p2300r4 – section 1.1

Let me illustrate how hard this is… by drawing from the C++ experience.

My only point: it is really hard.

C++ p2300r4 – section 1.1

While the C++ Standard Library has a rich set of concurrency primitives (std::atomic, std::mutex, std::counting_semaphore, etc) and lower level building blocks (std::thread, etc), we lack a Standard vocabulary and framework for asynchrony and parallelism that C++ programmers desperately need.

C++ p2300r4 – section 1.1

While the C++ Standard Library has a rich set of concurrency primitives (std::atomic, std::mutex, std::counting_semaphore, etc) and lower level building blocks (std::thread, etc), we lack a Standard vocabulary and framework for asynchrony and parallelism that C++ programmers desperately need.

std::async/std::future/std::promise, C++11's intended exposure for asynchrony, is inefficient, hard to use correctly, and severely lacking in genericity, making it unusable in many contexts.

C++ p2300r4 – section 1.1

While the C++ Standard Library has a rich set of concurrency primitives (std::atomic, std::mutex, std::counting_semaphore, etc) and lower level building blocks (std::thread, etc), we lack a Standard vocabulary and framework for asynchrony and parallelism that C++ programmers desperately need.

std::async/std::future/std::promise, C++11's intended exposure for asynchrony, is inefficient, hard to use correctly, and severely lacking in genericity, making it unusable in many contexts.

We introduced parallel algorithms to the C++ Standard Library in C++17, and while they are an excellent start, they are all inherently synchronous and not composable.

C++ p2300r4 – section 1.1

While the C++ Standard Library has a rich set of concurrency primitives (std::atomic, std::mutex, std::counting_semaphore, etc) and lower level building blocks (std::thread, etc), we lack a Standard vocabulary and framework for asynchrony and parallelism that C++ programmers desperately need.

std::async/std::future/std::promise, C++11's intended exposure for asynchrony, is inefficient, hard to use correctly, and severely lacking in genericity, making it unusable in many contexts.

We introduced parallel algorithms to the C++ Standard Library in C++17, and while they are an excellent start, they are all inherently synchronous and not composable.

This paper proposes a Standard C++ model for asynchrony, based around three key abstractions: schedulers, senders, and receivers, and a set of customizable asynchronous algorithms.

C++ p2300r4 – section 1.1

My point:

It's hard.

Be CAREFUL what you standardize.

History suggests we standardize too soon.

We need more proposals, criticism, failures, and refinement.

Portability is not enough in a heterogeneous world.
Performance Portability- Definition and Metric:

"A measurement of an application's performance efficiency for a given problem that can be executed correctly on all platforms in a given set."

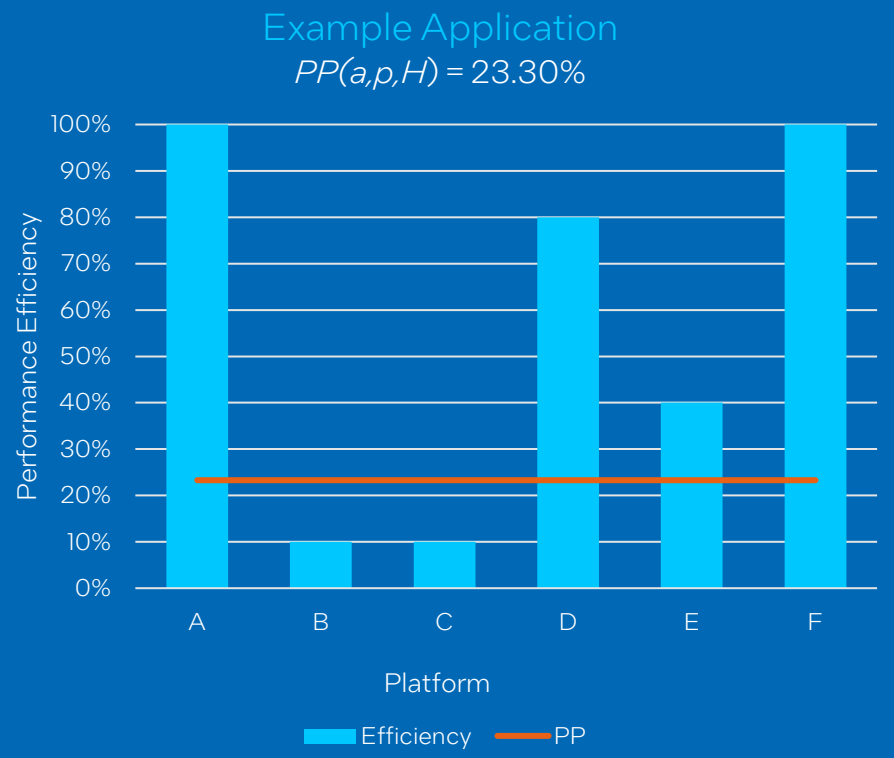
$$\mathcal{P}(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i \in H} \dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

Anything portable *is* "performance portable".
The question becomes: "How performance portable is it?"

- Yes/No answer for "is it PP?"
- Captures "average" performance in H
- Architectural and Application Efficiency

Recommended reading:

Navigating Performance, Portability and Productivity
https:// tinyurl.com / NavigatePerf



S. J. Pennycook, J. D. Sewall and V. W. Lee, "*A Metric for Performance Portability*", PMBS 2017

Heterogeneous Systems – Programming them

My talk today:

1. Heterogeneous Systems are here to stay
   and will be ubiquitous (like parallelism)

2. Standardizing support is HARD, and we keep getting it WRONG

3. Look at the essentials of SYCL (this is just for C++)

4. We need
   open, multivendor, multiarchitecture support
   that spans programming languages

5. This is OUR problem – let's solve it together

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

int main() {

    sycl::queue Q;
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    int sum;
    std::vector<int> data{1, 1, 1, 1, 1, 1, 1, 1};

    sycl::buffer<int> sum_buf(&sum, 1);
    sycl::buffer<int> data_buf(data);

    Q.submit([&](sycl::handler& h)
    {
        sycl::accessor buf_acc{data_buf, h, read_only};

        h.parallel_for(sycl::range<1>{8},
                       sycl::reduction(sum_buf, h, std::plus<>()),
                       [=](sycl::id<1> idx, auto& sum)
        {
            sum += buf_acc[idx];
        });
    });

    sycl::host_accessor result{sum_buf, read_only};
    std::cout << "Sum equals " << result[0] << std::endl;

    return 0;
}
```

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

int main() {

    sycl::queue Q;
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    int sum;
    std::vector<int> data{1, 1, 1, 1, 1, 1, 1, 1};

    sycl::buffer<int> sum_buf(&sum, 1);
    sycl::buffer<int> data_buf(data);

    Q.submit([&](sycl::handler& h)
    {
        sycl::accessor buf_acc{data_buf, h, read_only};

        h.parallel_for(sycl::range<1>{8},
                       sycl::reduction(sum_buf, h, std::plus<>()),
                       [=](sycl::id<1> idx, auto& sum)
        {
            sum += buf_acc[idx];
        });
    });

    sycl::host_accessor result{sum_buf, read_only};
    std::cout << "Sum equals " << result[0] << std::endl;

    return 0;
}
```

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

int main() {

    sycl::queue Q;
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    int sum;
    std::vector<int> data{1, 1, 1, 1, 1, 1, 1, 1};

    sycl::buffer<int> sum_buf(&sum, 1);
    sycl::buffer<int> data_buf(data);

    Q.submit([&](sycl::handler& h)
    {
        sycl::accessor buf_acc{data_buf, h, read_only};

        h.parallel_for(sycl::range<1>{8},
                       sycl::reduction(sum_buf, h, std::plus<>()),
                       [=](sycl::id<1> idx, auto& sum)
        {
            sum += buf_acc[idx];
        });
    });

    sycl::host_accessor result{sum_buf, read_only};
    std::cout << "Sum equals " << result[0] << std::endl;

    return 0;
}
```

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

int main() {

    sycl::queue Q;
    std::cout << "Running on: " << Q.get_device().get_info<sycl::info::device::name>() << std::endl;

    int sum;
    std::vector<int> data{1, 1, 1, 1, 1, 1, 1, 1};

    sycl::buffer<int> sum_buf(&sum, 1);
    sycl::buffer<int> data_buf(data);

    Q.submit([&](sycl::handler& h)
    {
        sycl::accessor buf_acc{data_buf, h, read_only};

        h.parallel_for(sycl::range<1>{8},
                       sycl::reduction(sum_buf, h, std::plus<>()),
                       [=](sycl::id<1> idx, auto& sum)
        {
            sum += buf_acc[idx];
        });
    });

    sycl::host_accessor result{sum_buf, read_only};
    std::cout << "Sum equals " << result[0] << std::endl;

    return 0;
}
```

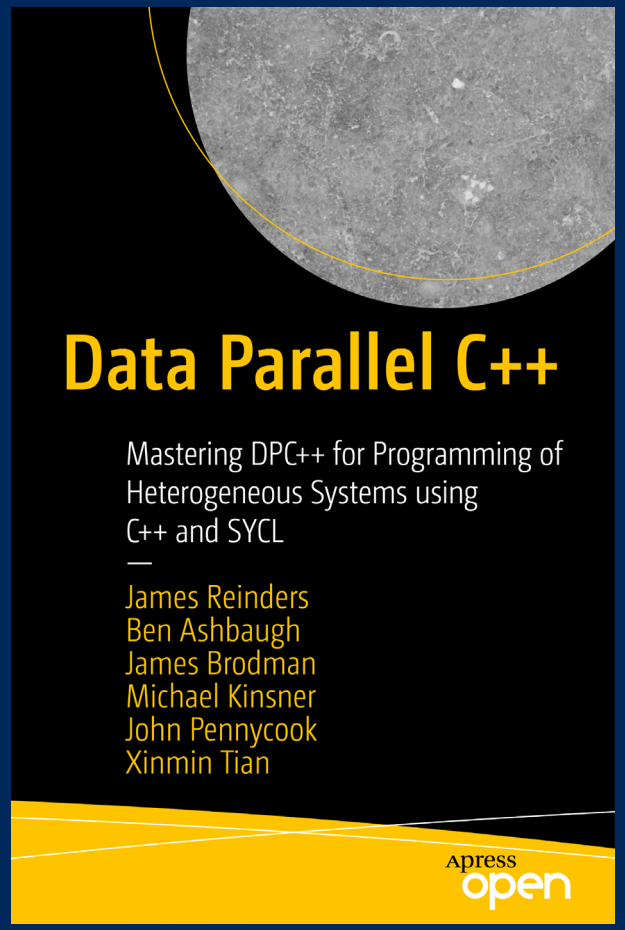
SYCL is expressive & exposes control

COMMON NEEDS FOR PROGRAMMERS

- Device queries
- Queue & context control
- OpenCL-like buffers and unified shared memory
- Optional asynchrony & task DAG
- Generic groups & group algorithms
- SPMD-to-SIMD interoperability (InvokeSIMD)
- JIT & Specialization Constants
- Interoperability with OpenMP

SYCL is expressive
exposes control

- Device queries
- Queue & context cor
- OpenCL-like buffers
- Optional asynchrony
- Generic groups & gr
- SPMD-to-SIMD inte
- JIT & Specialization
- Interoperability with



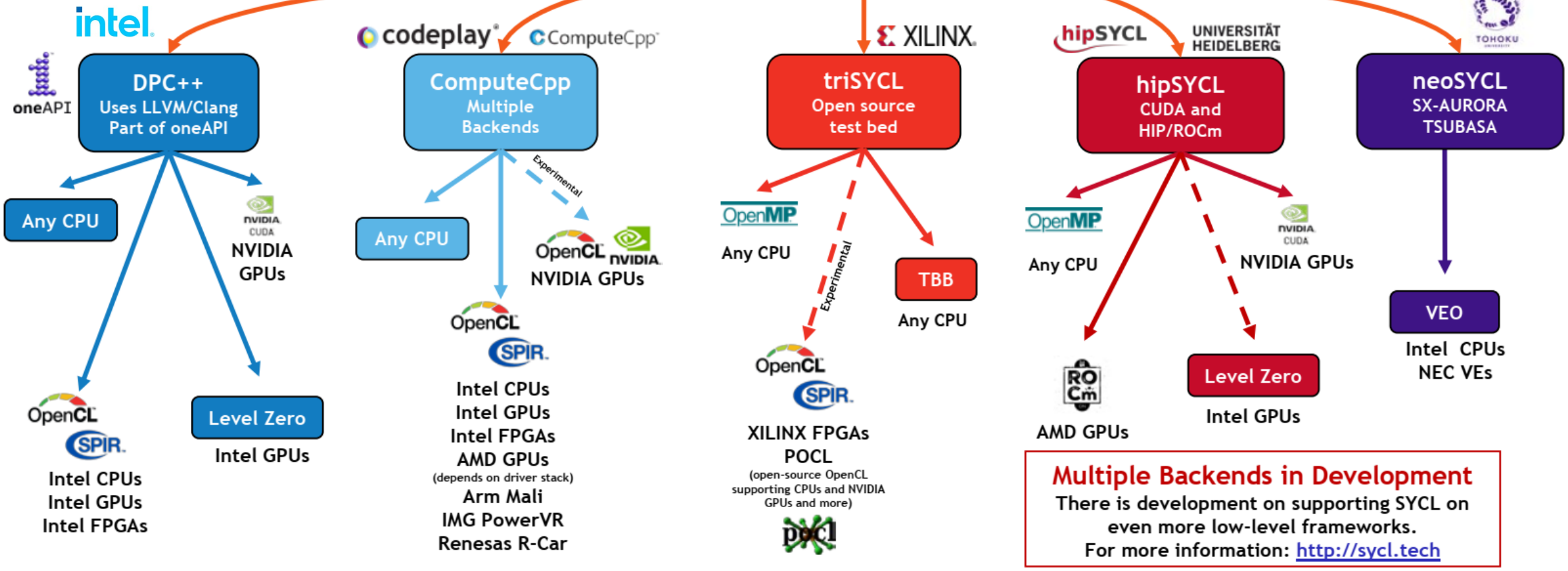
Book (PDF) Download
tinyurl.com/DataParallelCpp

SYCL Implementations in Development

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

SYCL Source Code

SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute



Multiple Backends in Development

There is development on supporting SYCL on even more low-level frameworks.
For more information: http://sycl.tech

https://www.iwocl.org/wp-content/uploads/k04-iwocl-syclcon-2021-wong-slide

Heterogeneous Systems – Programming them

My talk today:

1. Heterogeneous Systems are here to stay
   and will be ubiquitous (like parallelism)

2. Standardizing support is HARD, and we keep getting it WRONG

3. Look at the essentials of SYCL (this is just for C++)

4. We need
   open, multivendor, multiarchitecture support
   that spans programming languages

5. This is OUR problem – let's solve it together

C++ programming is just one piece

- LIBRARIES are KEY
- Supporting MANY languages is IMPORTANT
  e.g., Python, Fortran, C, Julia, …

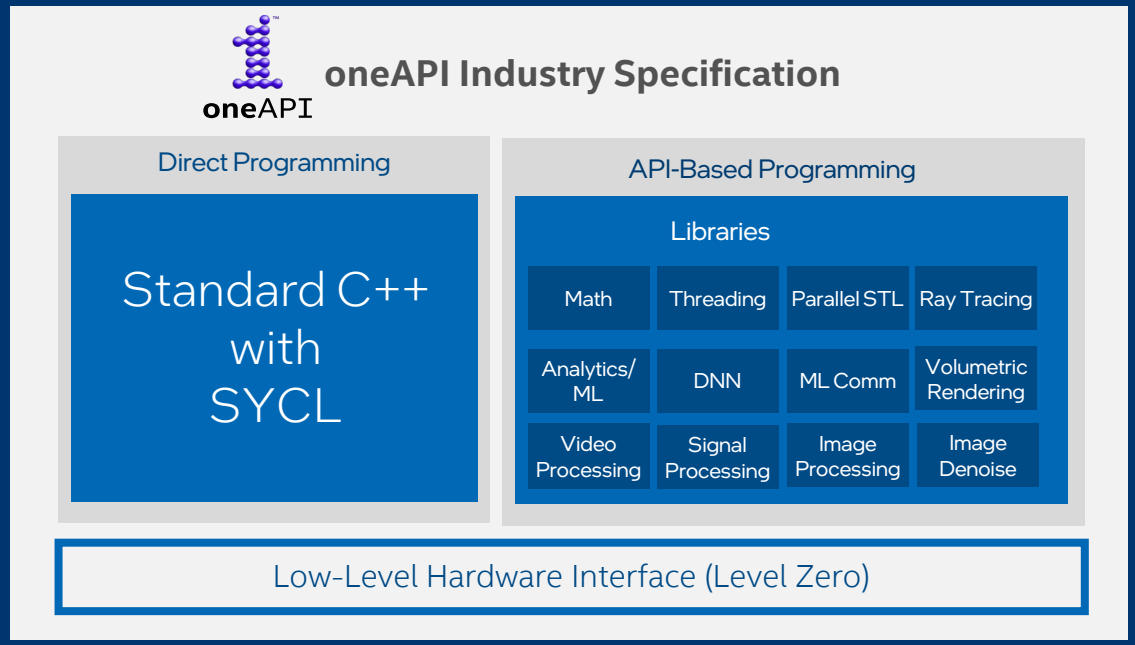
oneAPI: One Name, Two Distinct Objectives

oneAPI

- Open industry specification
- Open-source repo and development
- Community driven
- Multivendor implementations

oneAPI

- Intel's implementation
- Toolkits optimized for Intel HW
- Free to download and use

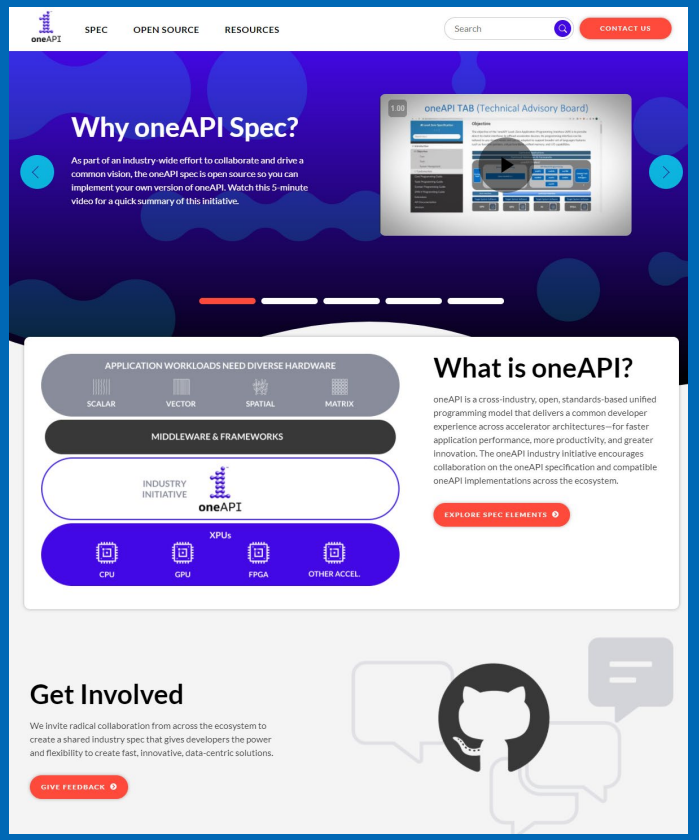


- Standard C++ with SYCL
- Standardized interfaces for common libraries
- Standardized hardware interface

41

oneAPI

An open specification and initiative to standardize programming of accelerated processing units (XPUs)

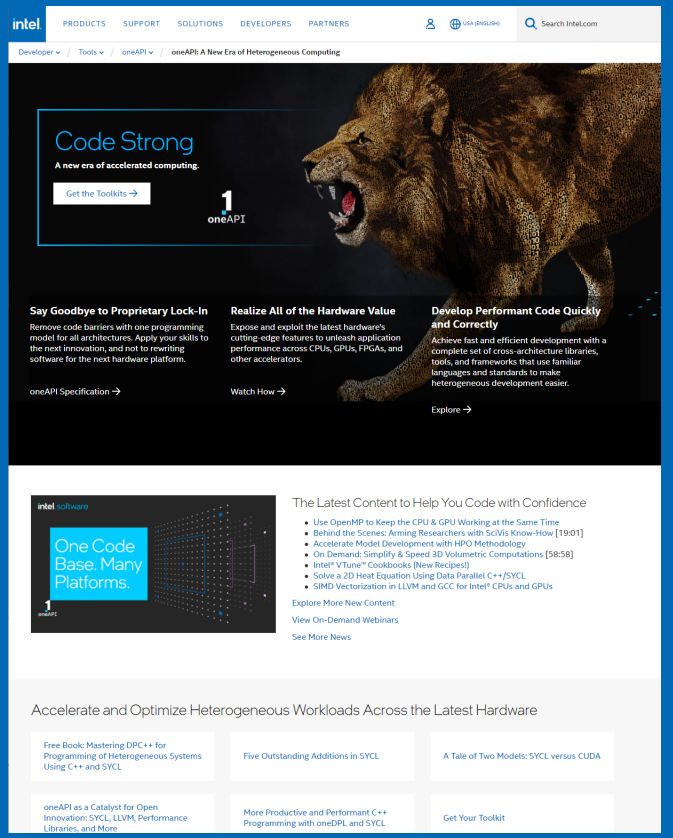


oneapi.io

oneAPI

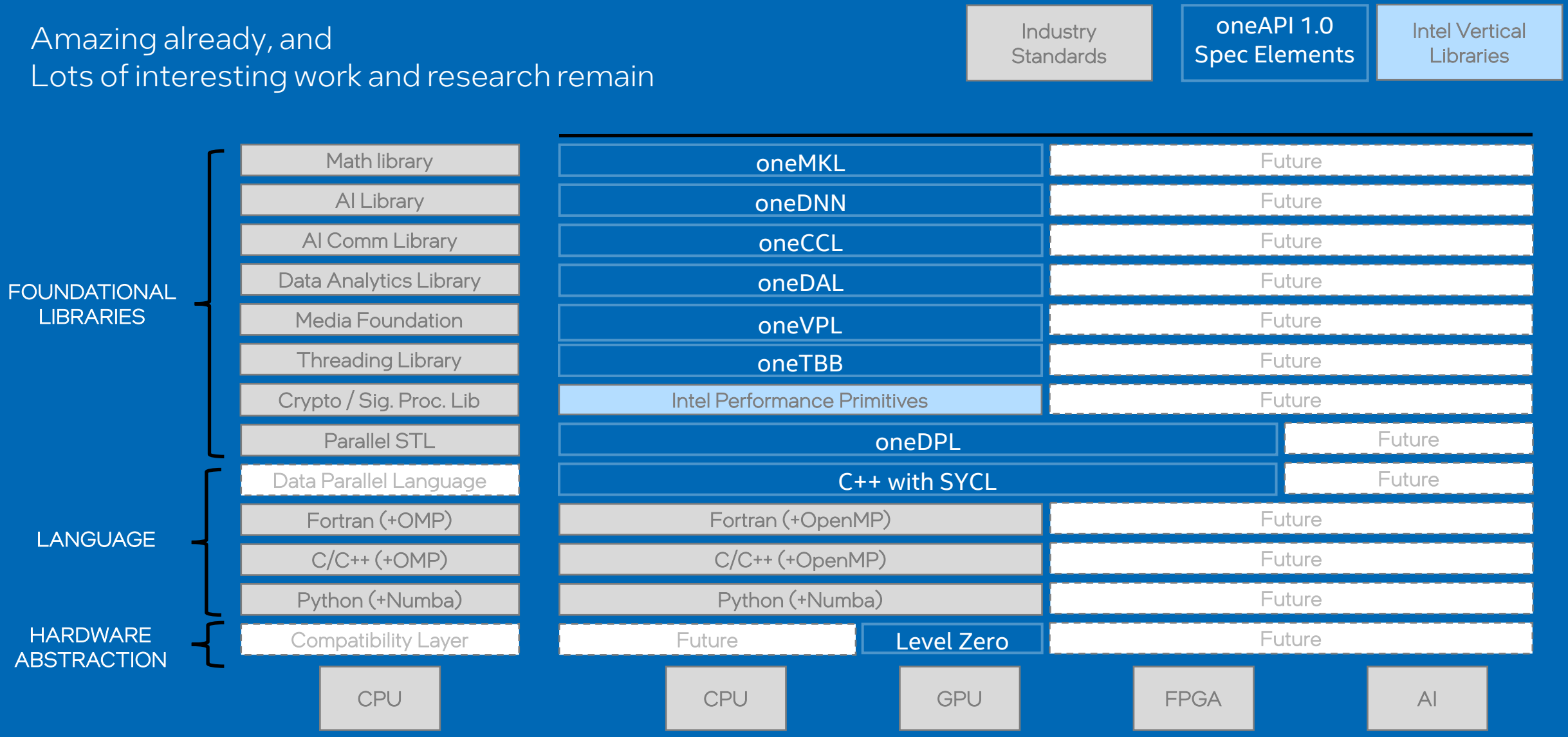
Intel's product implementation of the oneAPI specification

free



software.intel.com/oneAPI

Amazing already, and
Lots of interesting work and research remain



Common “under the covers” - lots of work to do!

Composability It’s important. ☺

Heterogeneous is leading to mix-and-match like nothing before, therefore… composibility matters even more.

Heterogeneous Systems – Programming them

My talk today:

1. Heterogeneous Systems are here to stay
   and will be ubiquitous (like parallelism)

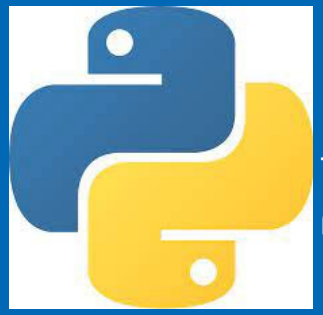
2. Standardizing support is HARD, and we keep getting it WRONG

3. Look at the essentials of SYCL (this is just for C++)

4. We need
   open, multivendor, multiarchitecture support
   that spans programming languages

5. This is OUR problem – let's solve it together

oneAPI is not alone



Python

DSLs

OpenMP

FORTRAN

Kokkos

RAJA

FleCSI

DOE-led efforts

oneAPI is not alone

We do STRESS our belief in the need to bring us all together to create an

open, multivendor, multiarchitecture, multilanguage

future

A List of the…

Effective Programming of Heterogeneous Systems needs:

- be open, multivendor, and multiarchitecture – always
  - Pass three tests:
    1. Freedom to use any device (regardless of vendor or architecture)
    2. Ability to access maximum performance
    3. A future for my investments in coding
- support across many programming languages
- performance portability
- commonality for developers
- commonality under the covers

It’s a Journey

We started oneAPI with a good idea

We knew enough to propose initial specifications

We are rapidly iterating and refining through community feedback

oneAPI has evolved

It's a Journey

We started oneAPI with a good idea

We knew enough to propose initial specifications

We are rapidly iterating and refining through community feedback

oneAPI has evolved

Much work remains – join us in creating an
open, multivendor, multiarchitecture, multilanguage future

https:// oneapi.io
https:// software.intel.com/oneAPI

Thank you

Have a GREAT conference!

We started oneAPI with a good idea

We knew enough to propose initial specifications

We are rapidly iterating and refining through community feedback

oneAPI has evolved

Much work remains – join us in creating an
open, multivendor, multiarchitecture, multilanguage future

https:// oneapi.io
https:// software.intel.com/oneAPI

Disclaimers & Notices

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 www.intel.com/benchmarks.

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. Check with your system manufacturer or retailer or learn more at intel.com.

Khronos® is a registered trademark and SYCL™ and SPIR™ are trademarks of The Khronos Group Inc. OpenCL™ is a trademark of Apple Inc. used by permission by Khronos.