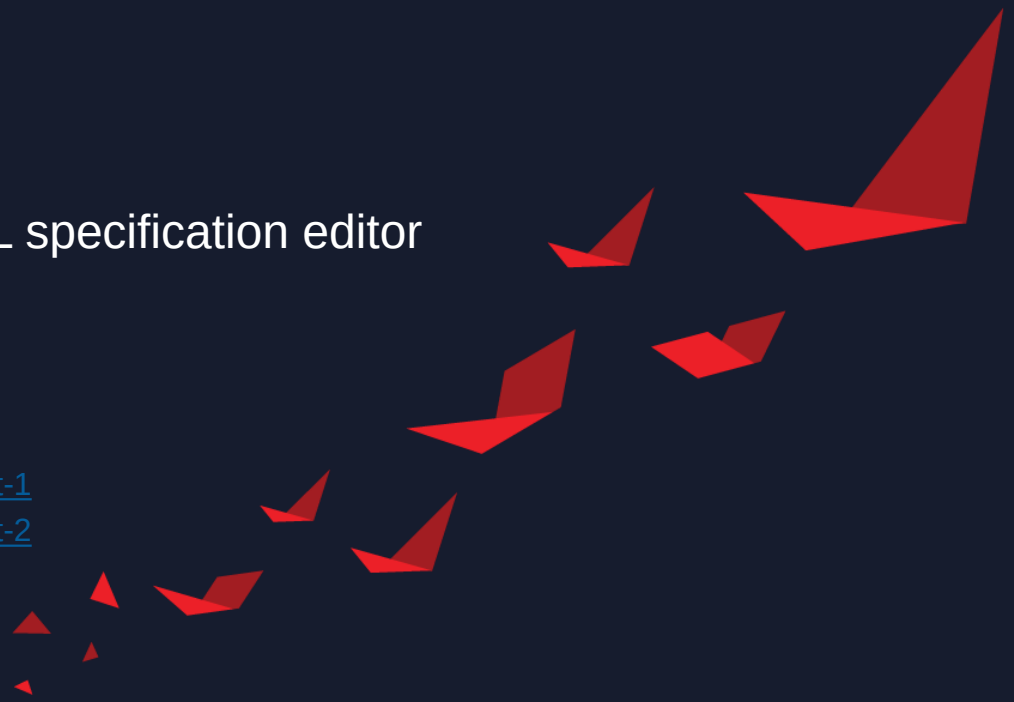
XILINX

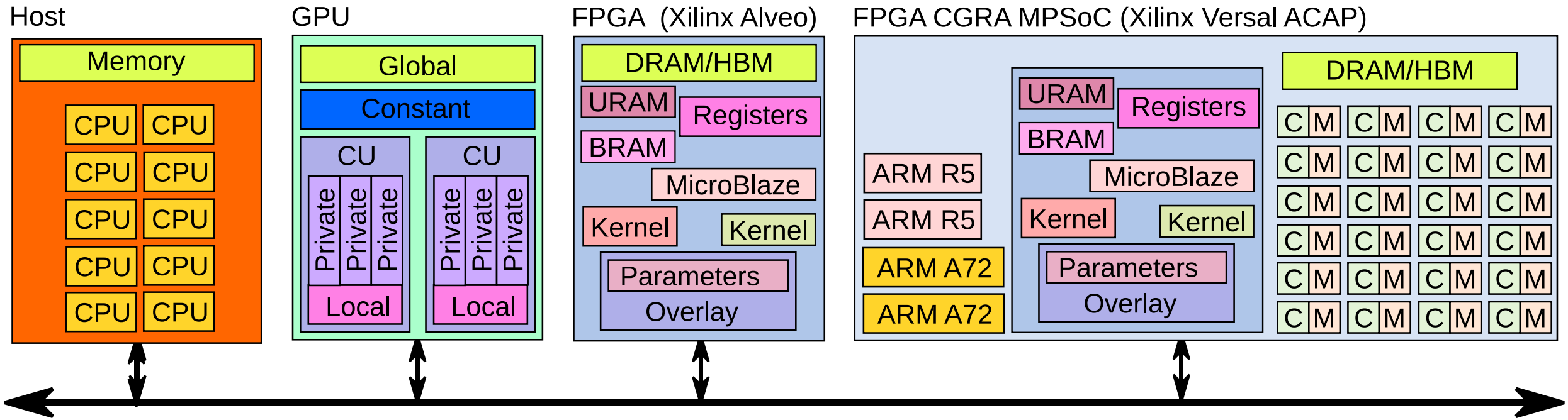
SYCL 2020, C++20 & PiM, In-Memory-Computing, Near-Memory-Computing…

Ronan Keryell (rkeryell@xilinx.com)
Xilinx Research Labs (San José, California) & Khronos SYCL specification editor
2021/02/15
Journée thématique du GDR SoC²:
In-Memory-Computing: from Device to Programming Model
https://replay.ec-lyon.fr/video/0819-gdr-soc2-journee-thematique-in-memory-computing-part-1
https://replay.ec-lyon.fr/video/0818-gdr-soc2-journee-thematique-in-memory-computing-part-2



SYCL

Programing a *full* modern/future system…



- Add your own accelerator to this picture...
- Scale this at the data-center/HPC level too…
- Need a programming model…
- Tim Mattson's law (Intel): no new language! ☺

C++20: is this C++ for Python programmers?

- Modern Python (3.9)

```
for e in [ 1,2,3,5,7 ] :
  print(e)
```

- Modern C++ (C++20)

```
for (auto e : { 1,2,3,5,7 })
  std::cout << e << std::endl;
```

- C (also usable in C++)

```
int a[] = { 1,2,3,5,7 };
 for (int i = 0;
       i < sizeof(a)/sizeof(a[0]);
      ++i)
   printf("%d ", a[i]);
```

- Old C++ (C++98/C++03)

```
std::vector<int> v;
v.push_back(1);
v.push_back(2);
v.push_back(3);
v.push_back(5);
v.push_back(7);
for (std::vector<int>::iterator i =
      v.begin(); i != v.end(); ++i)
  std::cout << *i << std::endl;
```

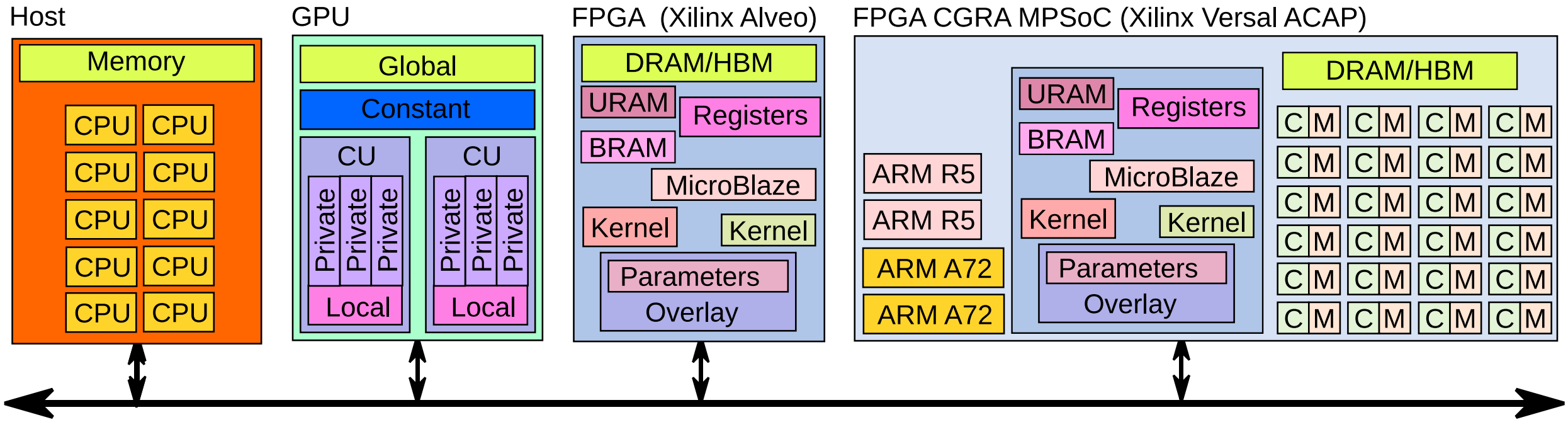
C++20 down to x86 assembly code specialization

```
#include <cassert>
#include <type_traits>
constexpr auto sum = [](auto arg, auto... args) {
  if (std::is_constant_evaluated()) {
    return (arg + ... + args);
  } else {
    ([&](auto value) {
      asm("leal (%1, %2), %0"
        : "=r" (arg)
        : "r"  (arg),
          "r"  (value));
    }(args), ...);
    return arg;
  }
};
static_assert(6 == sum(1, 2, 3));
int main(int argc, char *[]) {
  assert(15 == sum(1, 2, 3, 4, 5));
}
```

- **Remember inline assembly???**
- **Now with compile-time specialization without code change!!!**
- **https://godbolt.org/z/CJsKZT**
- **C is just *le passé*… ☺**

- **Inspired by Kris Jusiak @krisjusiak Oct 24 2019**
  - https://twitter.com/krisjusiak/status/1187387616980635649
  - C++20: Folding over inline-assembly in constexpr

Programing a *full* modern/future system… with C++???



- C++20 might be good for direct programing…
  - Handle both high-level and low-level
  - Zero cost abstraction
- …but nothing for heterogeneous computing yet! ☹

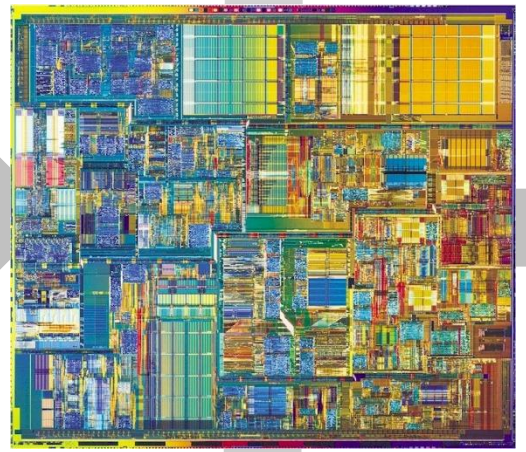
Khronos standards for heterogeneous systems

Connecting Software to Silicon

COLLADA

3D for the Web
- Real-time apps and games in-browser
- Efficiently delivering runtime 3D assets

WebGL glTF



NNEF OpenVX

Vision and Neural Networks
- Tracking and odometry
- Scene analysis/understanding
- Neural Network inferencing

OpenGL SC OpenVG OpenGL ES Vulkan OpenGL OpenXR

Real-time 2D/3D
- Virtual and Augmented Reality
- Cross-platform gaming and UI
- CG Visual Effects
- CAD and Product Design
- Safety-critical displays

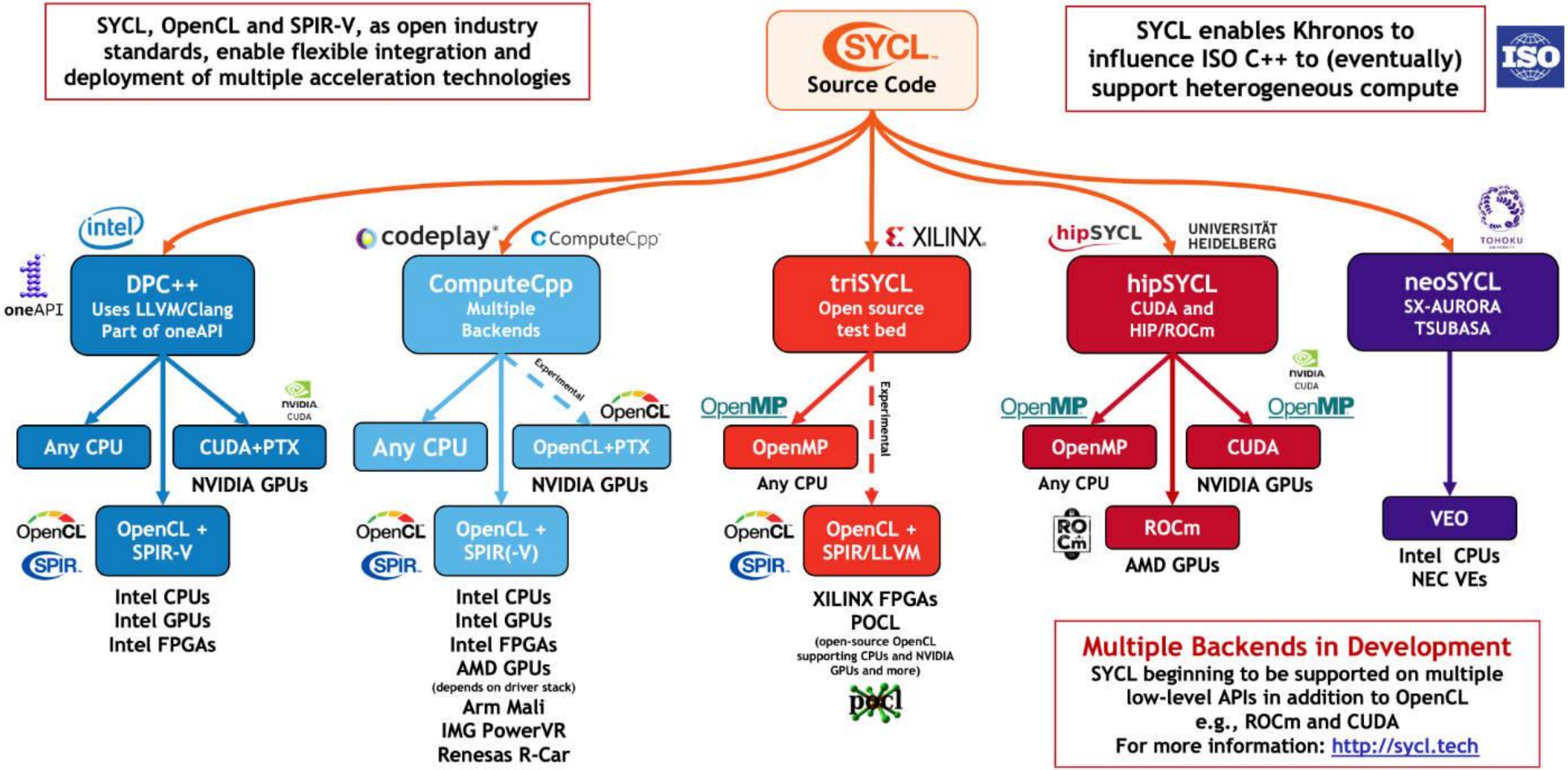
Parallel Computation
- Machine Learning acceleration
- Embedded vision processing
- High Performance Computing (HPC)

OpenCL SPIR SYCL

XILINX

Khronos Group SYCL C++ DSL — current public ecosystem

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



Multiple Backends in Development
SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL
e.g., ROCm and CUDA
For more information: http://sycl.tech

https://www.khronos.org/blog/sycl-2020-what-do-you-need-to-know

+ Celerity: SYCL on MPI+SYCL

SYCL 2020 ≡ heterogeneous simplicity with modern C++

```
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
constexpr int N = 32;

int main () {
  buffer<int> b { N };
  queue {}.submit([&](auto &h) {
     accessor a { b, h, write_only, no_init };
     h.parallel_for(N, [=](auto i) { a[i] = i; });
  });
  for (host_accessor a { B }; auto e : a)
     std::cout << e << std::end;
}
```

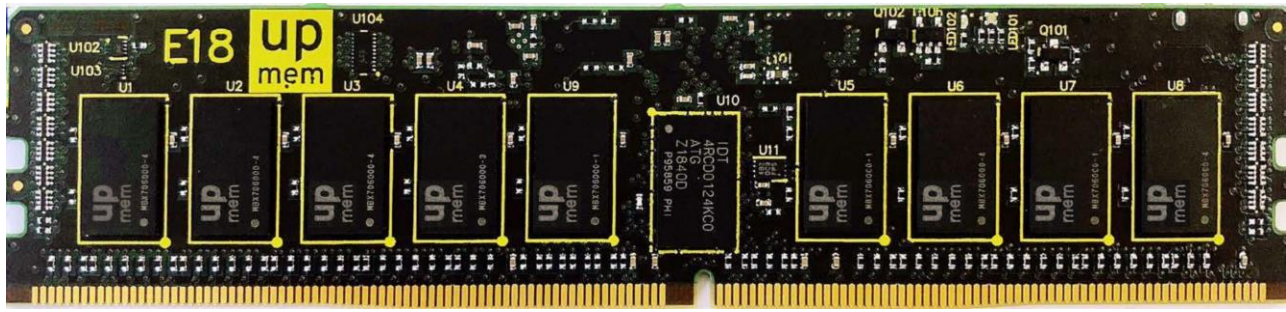
- Abstract storage
- Code executed on device (“kernel”)
- “Single-source”
  - Seamless integration in host code
  - Type-safety
- Accessor
  - Express access intention
  - Implicit data flow graph
  - Automatic data transfers across devices
  - Overlap computation & communication

SYCL 2020 with unified shared memory (USM)

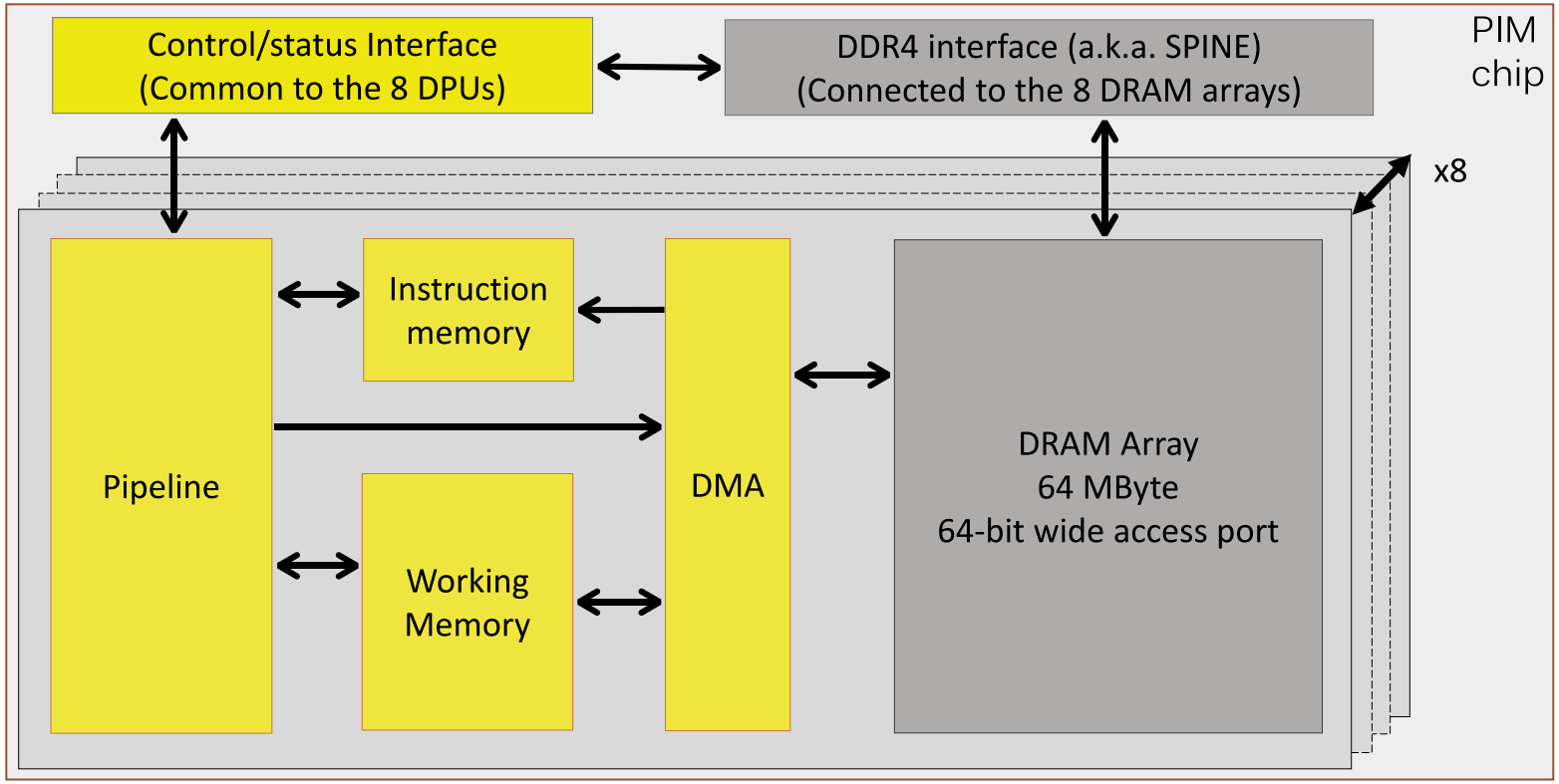
```
// Using buffers and accessors
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
constexpr int N = 32;

int main () {
  buffer<int> b { N };
  queue {}.submit([&](auto &h) {
     accessor a { b, h, write_only, no_init };
     h.parallel_for(N, [=](auto i) { a[i] = i; });
  });
  for (host_accessor a { B }; auto e : a)
     std::cout << e << std::end;
}
```

```
// Using USM only
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
constexpr int N = 32;

int main () {
  queue q;
  int *a = malloc_shared<int>(N, q);
  q.parallel_for(N, [=](auto i) { a[i] = i; });
  q.wait();
  for (int i = 0; i < N; i++)
     std::cout << a[i] << std::endl;
}
```

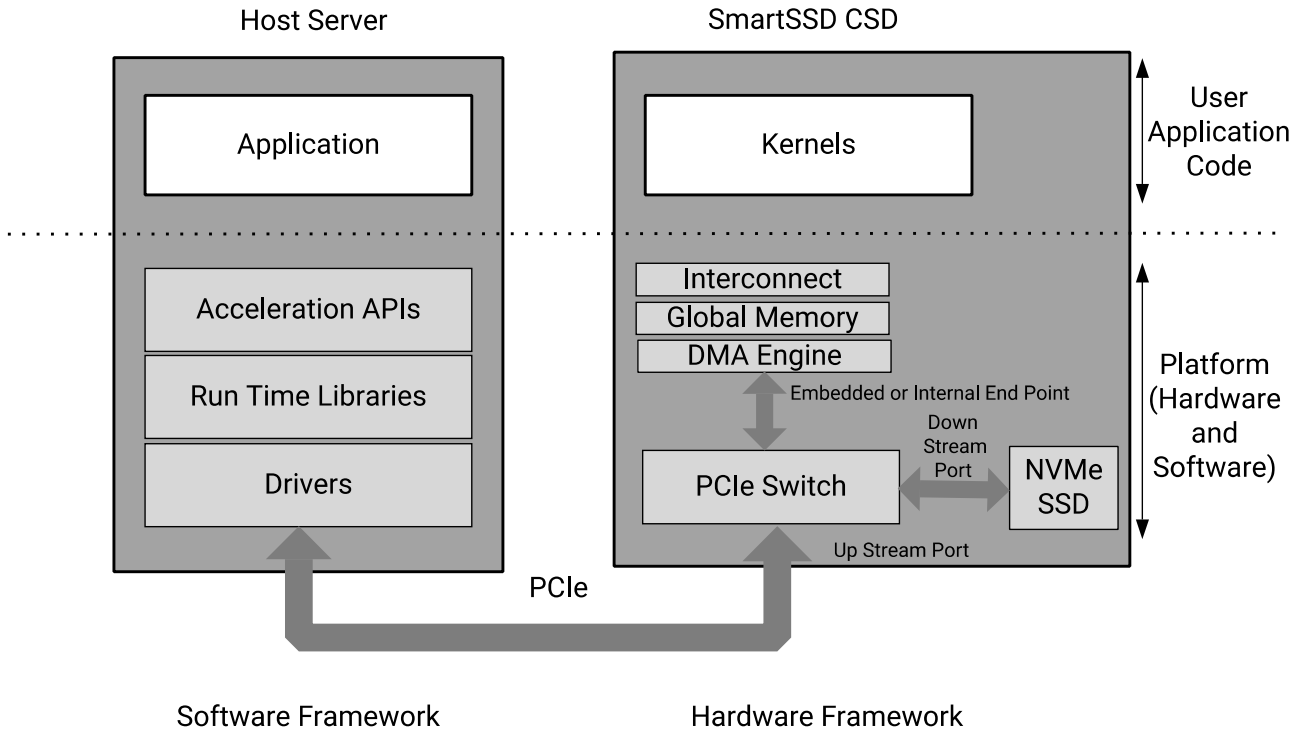
UPMEM: Processing In Memory accelerator @ HotChips-31 2019



PIM chip Block Diagram



Samsung SmartSSD® computational storage drive (CSD)



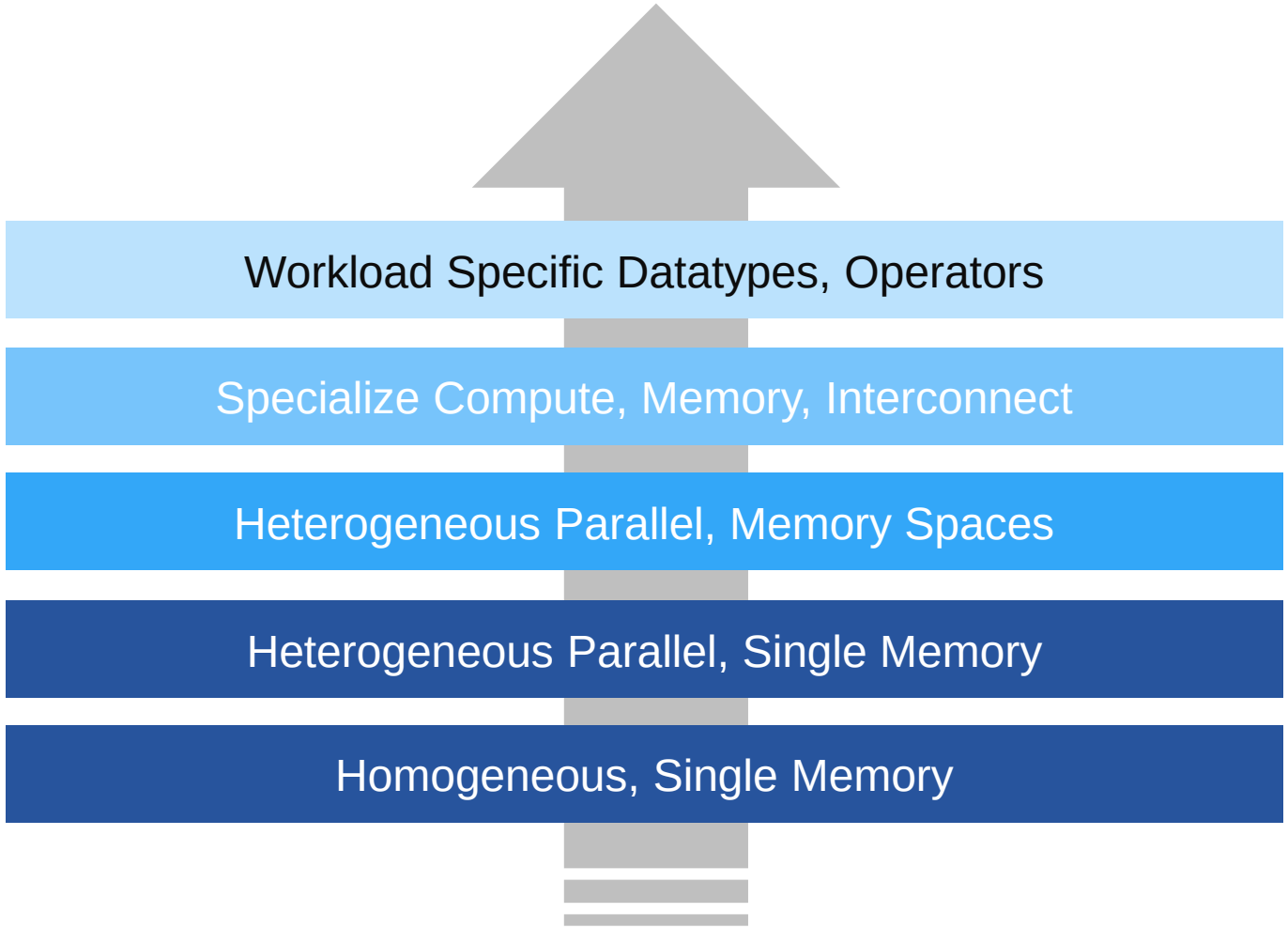
- 60 x 100 x 15 mm, 400 grams
- Samsung V-NAND Flash SSD 2.5” (U.2) 3.84TB
  - NVMe spec rev. 1.3, PCIe v3
  - Up to 3,300 MB/sec Sequential Read (128KB)
  - Up to 2,000 MB/sec Sequential Write (128KB)
- Xilinx KintexTM Ultrascale+ KU15P FPGA
  - 1.143M System Logic Cells
  - 1,968 DSP Slices
  - 34.6 Mbit Internal Distributed RAM
  - 36.0 Mbit Internal UltraRAM
  - 4 Gbyte DDR4 SDRAM Accelerator-dedicated RAM

Samsung SmartSSD® typical applications

- AI/ML Inference
- Big Data Analytics
- Business Intelligence
- Data Lake/DB Acceleration
- Data Warehousing
- Encryption/Decryption
- Financial Services
- Genomics
- Search Queries
- Storage & Virtualization
- Transparent Compression
- Video Analytics
- Video File Transcoding

https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html
To get access to the Xilinx FPGA shell for Vitis:
https://www.xilinx.com/member/storage-shell.html#smartssd

SYCL ≡ plain C++ → refinement levels with plain C++ libraries



Domain Lib

Architecture Lib

C++SYCL

C++Parallel Lib

C++20

XILINX

Use SYCL 2020 buffers with properties for PiM/IMC

```
// Using buffers
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
constexpr int N = 32;

int main () {
  buffer<int> b { N };
  queue {}.submit([&](auto &h) {
     accessor a { b, h, write_only, no_init };
     h.parallel_for(N, [=](auto i) { a[i] = i; });
  });
  for (host_accessor a { B }; auto e : a)
     std::cout << e << std::end;
}
```

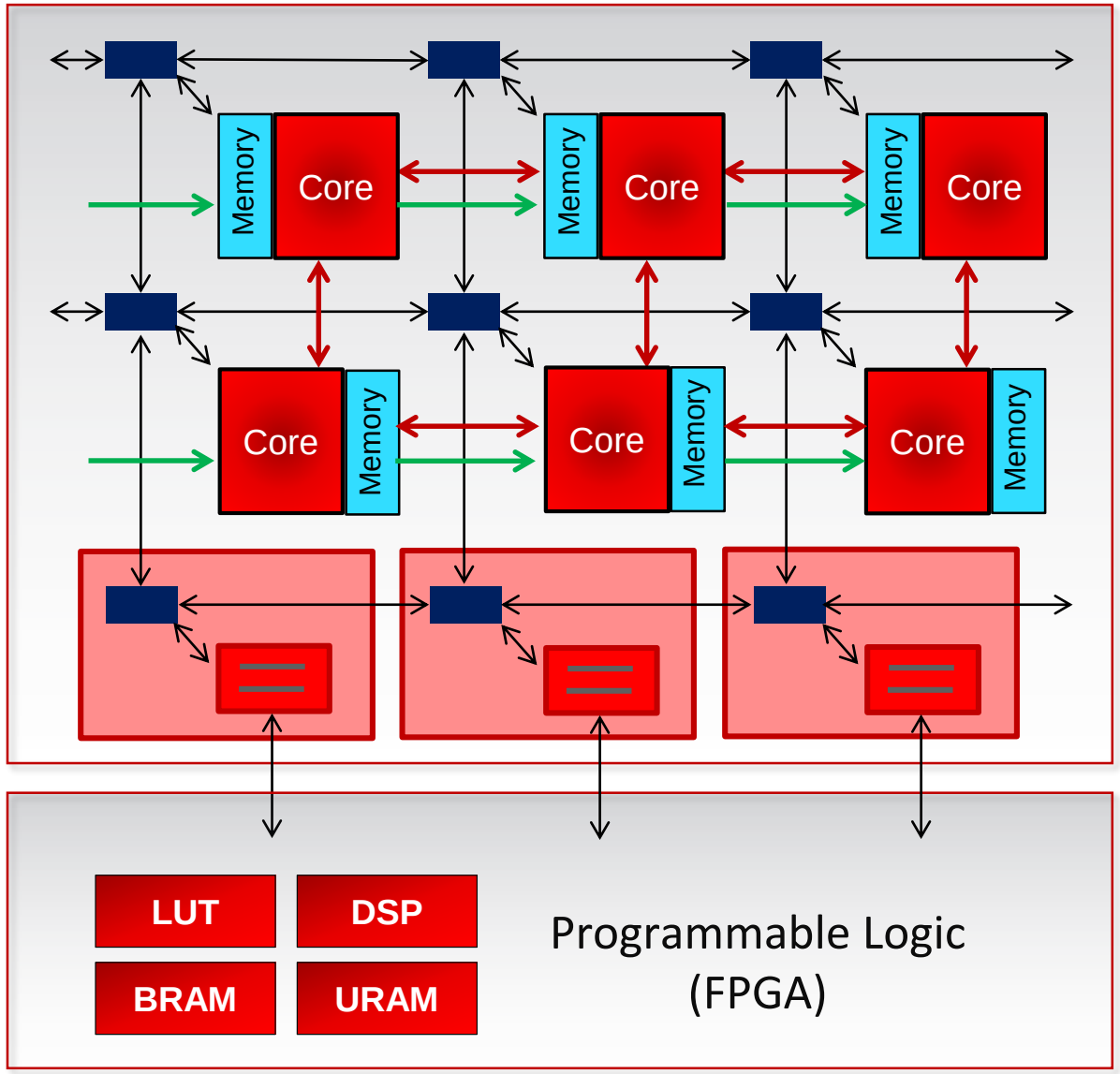
```
// Using buffers + IMC properties
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
constexpr int N = 32;

int main () {
  device ssd { [](auto& d) { return
    d.get_info<info::device::name>() == "SmartSSD";
    } };
  buffer<int> b { N, imc::persistent { ssd }, imc::ns { 1 },
                     imc::lba { 0x1000 } };
  queue { ssd }.submit([&](auto &h) {
     accessor a { b, h, write_only, no_init };
     h.parallel_for(N, [=](auto i) { a[i] = i; });
  });
  for (host_accessor a { B }; auto e : a)
     std::cout << e << std::end;
}
```

Use SYCL 2020 USM with properties for PiM/IMC

```
// Using buffers + IMC properties
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
constexpr int N = 32;

int main () {
  device ssd { [](auto& d) { return
    d.get_info<info::device::name>() == "SmartSSD";
    } };
  buffer<int> b { N, imc::persistent { ssd }, imc::ns { 1 },
                     imc::lba_start { 0x1000 } };
  queue { ssd }.submit([&](auto &h) {
     accessor a { b, h, write_only, no_init };
     h.parallel_for(N, [=](auto i) { a[i] = i; });
  });
  for (host_accessor a { B }; auto e : a)
     std::cout << e << std::end;
}
```

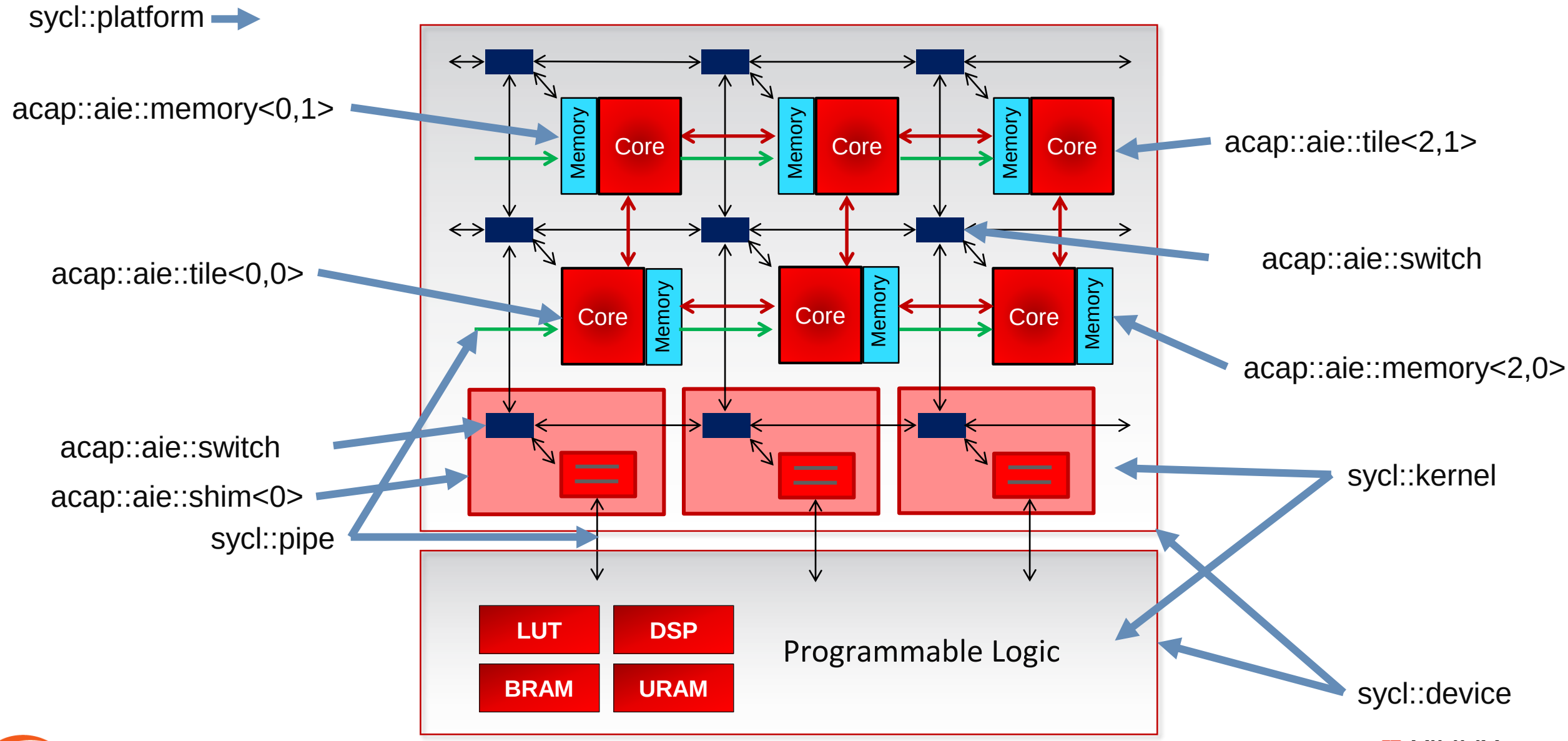
```
// Using USM + IMC properties
#include <sycl/sycl.hpp>
#include <iostream>
using namespace sycl;
constexpr int N = 32;

int main () {
  device ssd { [](auto& d) { return
    d.get_info<info::device::name>() == "SmartSSD";
    } };
  queue q { ssd };
  int *a = malloc_shared<int>(N, q, imc::ns { 1 },
                                    imc::persistent { ssd },
                                    imc::lba_start { 0x1000 });
  q.parallel_for(N, [=](auto i) { a[i] = i; });
  q.wait();
  for (int i = 0; i <  N; i++)
     std::cout << a[i] << std::endl;
}
```

CGRA inside Xilinx Versal VC1902 (37B tr, 7nm)



- Scalar Engines:
  - ARM dual-core Cortex-A72 (Host)
  - ARM dual-core Cortex-R5
- Programmable Logic (FPGA)
- Coarse-Grain Reconfigurable Array
  - 400 AI Engine (AIE) cores (tiles)
  - Each AIE Tile contains:
    - 32-bit Scalar RISC processor
    - 512-bit VLIW SIMD vector processor
    - 32KB RAM
    - Each tile can access neighbor's memory
      - 128KB shared
    - 16KB program memory
- Fast I/O
- Etc…

ACAP++: SYCL abstractions templated by 2D coordinates



Mandlebrot set computation (≈ cryptocurrency POW too…)

- Start simple: no neighborhood communication ☺

```
using namespace sycl::vendor::xilinx;

static auto constexpr image_size = 229;
graphics::application a;

// All the memory modules are the same
template <typename AIE, int X, int Y>
struct pixel_tile : acap::aie::memory<AIE, X, Y> {
  // The local pixel tile inside the complex plane
  std::uint8_t plane[image_size][image_size];
};

// All the tiles run the same Mandelbrot program
template <typename AIE, int X, int Y>
struct mandelbrot : acap::aie::tile<AIE, X, Y> {
  using t = acap::aie::tile<AIE, X, Y>;
  // Computation rectangle in the complex plane
  static auto constexpr x0 = -2.1, y0 = -1.2, x1 = 0.6, y1 = 1.2;
  static auto constexpr D = 100; // Divergence norm
  // Size of an image tile
  static auto constexpr xs = (x1 - x0)/t::geo::x_size/image_size;
  static auto constexpr ys = (y1 - y0)/t::geo::y_size/image_size;
  void run() {
    // Access to its own memory
    auto& m = t::mem();
    while (!a.is_done()) {
      for (int i = 0; i < image_size; ++i)
        for (int k, j = 0; j < image_size; ++j) {
          std::complex c { x0 + xs*(X*image_size + i),
                           y0 + ys*(Y*image_size + j) };
          std::complex z { 0.0 };
          for (k = 0; k <= 255; k++) {
            z = z*z + c;
            if (norm(z) > D)
              break;
          }
          m.plane[j][i] = k;
        }
      a.update_tile_data_image(t::x, t::y, &m.plane[0][0], 0, 255);
    }
  }
};

int main(int argc, char *argv[]) {
  acap::aie::device<acap::aie::layout::size<2,3>> aie;
  // Open a graphic view of a AIE array
  a.start(argc, argv, decltype(aie)::geo::x_size, decltype(aie)::geo::y_size,
          image_size, image_size, 1);
  a.image_grid().palette().set(graphics::palette::rainbow, 100, 2, 0);

  // Launch the AI Engine program
  aie.run<mandelbrot, pixel_tile>();
  // Wait for the graphics to stop
  a.wait();
}
```

Wave propagation sequential PDE reference code (mdspan!)

```
/// Compute a time-step of wave propagation
  void compute() {
    for (int j = 0; j < size_y; ++j)
      for (int i = 0; i < size_x - 1; ++i) {
        // dw/dx
        auto up = w(j,i + 1) - w(j,i);
        // Integrate horizontal speed
        u(j,i) += up*alpha;
      }
    for (int j = 0; j < size_y - 1; ++j)
      for (int i = 0; i < size_x; ++i) {
        // dw/dy
        auto vp = w(j + 1,i) - w(j,i);
        // Integrate vertical speed
        v(j,i) += vp*alpha;
      }
    for (int j = 1; j < size_y; ++j)
      for (int i = 1; i < size_x; ++i) {
        // div speed
        auto wp = (u(j,i) - u(j,i - 1)) + (v(j,i) - v(j - 1,i));
        wp *= side(j,i)*(depth(j,i) + w(j,i));
        // Integrate depth
        w(j,i) += wp;
        // Add some dissipation for the damping
        w(j,i) *= damping;
      }
  }
```

Moving lines and columns around…

```
void compute() {
  auto& m = t::mem();

  for (int j = 0; j < image_size; ++j)
    for (int i = 0; i < image_size - 1; ++i) {
      // dw/dx
      auto up = m.w[j][i + 1] - m.w[j][i];
      // Integrate horizontal speed
      m.u[j][i] += up*alpha;
    }

  for (int j = 0; j < image_size - 1; ++j)
    for (int i = 0; i < image_size; ++i) {
      // dw/dy
      auto vp = m.w[j + 1][i] - m.w[j][i];
      // Integrate vertical speed
      m.v[j][i] += vp*alpha;
    }

  t::barrier();
[…]

  if constexpr (t::is_memory_module_up()) {
    auto& above = t::mem_up();
    for (int i = 0; i < image_size; ++i)
      above.w[0][i] = m.w[image_size - 1][i];
  }

  t::barrier();

  // Transfer last line of w to next memory module on the right
  if constexpr (Y & 1) {
    if constexpr (t::is_memory_module_right()) {
      auto& right = t::mem_right();
      for (int j = 0; j < image_size; ++j)
        right.w[j][0] = m.w[j][image_size - 1];
    }
  }
  if constexpr (!(Y & 1)) {
    if constexpr (t::is_memory_module_left()) {
      auto& left = t::mem_left();
      for (int j = 0; j < image_size; ++j)
        m.w[j][0] = left.w[j][image_size - 1];
    }
  }
```

Multi-level implementation/emulation for codesign & debug

Different types of implementations

- Full SYCL compiler & runtime implementation
  - Run on real hardware on hardware simulator
- Pure SYCL C++ implementation
  - No specific compiler required!
  - Run on (laptop) host CPU at full C++ speed, standard debugging, thread-sanitizer of hardware features…
    - 1 thread per host… thread, 1 thread per AIE tile, 1 thread per GPU work-item, 1 thread per FPGA work-item
  - Easy code instrumentation for statistics by adapting SYCL C++ classes
  - Use normal debugger
    - Gdb is scriptable in Python to expose new features ☺
- Mix-and-match
  - Run some parts of the hardware remotely or in simulators
  - Allow kernels on host CPU while using memory-mapped real hardware (DMA, AXI streams, NoC…)
  - Distribute execution across datacenter (Celerity SYCL for MPI+SYCL)

Documentation & resources

- https://www.khronos.org/sycl
- SYCL-related community resource web site http://sycl.tech
- https://tech.io/playgrounds/48226/introduction-to-sycl/introduction-to-sycl-2
  - playground on “Introduction to SYCL” with on-line code execution
- Code samples from Codeplay
  - https://github.com/codeplaysoftware/computecpp-sdk/tree/master/samples
- Parallel Research Kernels implemented with different frameworks, including SYCL
  - Useful to compare or translate between different programming models https://github.com/ParRes/Kernels
- Intel open-source to be up-streamed, aka oneAPI DPC++
  - https://github.com/intel/llvm
- Open-source (mainly Xilinx)
  - https://github.com/triSYCL/triSYCL
- Open-source fusion triSYCL+Intel to Xilinx FPGA
  - https://github.com/triSYCL/sycl

Conclusion

- Do not invent a new language for new hardware! (But at least use the latest version ☺)
  - Simpler programming & legacy compatibility
  - Only open-standard and open-source matter
- Focus on direct programming the **whole** system: SYstem-wide Compute Language
  - Single-source → address the full application, simpler, type-safe & enable generic libraries
  - Adaptable interoperability with any device: FPGA, ACAP, CUDA, HIP, OpenMP, OpenCL, Vulkan, MLIR, NEC…
    - Heterogeneous programming CPU+GPU+CGRA+FPGA+DSP+Vector+AI+PiM/IMC…
- SYCL 2020 ≡ pure modern C++ DSL
  - Extensible with abstractions for any device: FPGA, ACAP++, PiM/IMC
  - Target emulation, debug & co-design on CPU for free → 256 cores AMD EPYC ≡ fast ACAP/GPU emulation ! ☺
- For a teaching audience: C++ is now C++20
  - Generic & functional programming
  - Forget about old C++ (and C?)
  - Only teach modern C++!
  - Always looking for good master/PhD-level interns ☺

Thank You