Argonne NATIONAL LABORATORY

GPU Hardware Characteristics

How does that even work? (ATPESC edition)

Thomas Applencourt(apl@anl.gov)

July 27, 2026

Outline

Who I'm ? (bias)

- Part of the Performance Enginiring Group (so we will discuss some tinming, and performance)
- Worked a lot, and for a long time, on Aurora (so we will use it sorry, and we will use Intel Hardware, but everything should (is?) transferable)
- Part of the SYCL Working-Group (so yeah, I like open stantard)

Outline

Key Question

- What is the difference between a CPU and a GPU?
- Why we can use CUDA only on NVIDIA Hardware?
- Should I use OpenMP, SYCL or Kokkos?

Roadmap for my two sessions

1. **This morning — What is a GPU?** Hardware in a nutshell: parallelism, the memory wall, the roofline.
2. **This afternoon — Climbing the ladder**: from the low-level model up to SYCL / OpenMP / Kokkos, and why they are all the same idea.

In between, the low-level ecosystem (CUDA / HIP / OpenCL).

Outline

At the beginning was a loom[1]

Figure 1: Late 1800s, Jacquard Machine: Threads and Warps

[1] and then the Canuts, the Luddite, Engels' pause...

NVIDIA in their wisdom reused some weaving terms

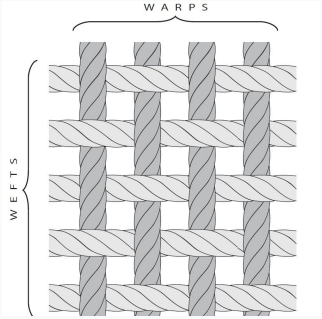


Figure 2: Warp and weft threads on a loom

Don’t worry, later we will teach some less confusing terminology...

And then CPU

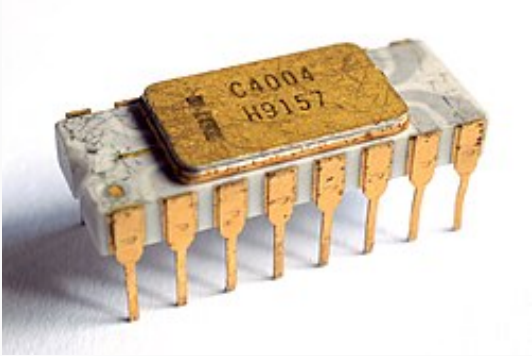


Figure 3: 1970, Intel 4004, 740 kHz, 2300 transistor

How to get more performance?

- **Be Smarter**
  - Branch prediction[2], ILP (instruction level parallelism – aka don't stall)
- **Increase frequency!** Thread go faster!
  - Core i9-14900K (released in 2023) is 6 GHz
  - One problem is power: $P \propto CV^2f$, and higher f needs higher $V \Rightarrow$ heat grows.
  - But still, still will not give you 2000x speed-up...
- **What about doing more stuff in parallel?**
  - SIMD (single instruction multiple data)
  - More thread!
  - This is Smart. We can always put "more" of stuff. "horizontal scaling"

[2] Speculative execution was a really good idea security wise...

Parallelism isn't new: vector computing is 50 years old

- The **Cray-1** (1976) was a **vector** machine: one instruction over a whole vector of data.
- And then lots of CPUs work in parallel
- GPU is the same thing! [a]

[a] Kind of... with more limited synchronization

Figure 4: Cray-1, Science Museum London (CC BY-SA)

And then come GPU...

Let's just put more parallelism. No sync between them. Just for "embarrassingly-parallel" applications. Like... drawing pixels on a screen?
Graphics Processing Unit

It's just

- A GPU $\approx$ a very "stupid" CPU with a huge number of wide SIMD threads.
- And then people realised that GPU can be used to do some Linear Algebra. So why not... GP-GPU General-Purpose Graphics Processing Unit (means nothing already...)
- CPU followed a similar pattern, just added more threads, but smart threads. (All those ooo) and fancy synchronization (software threads can "time-slice / hyper-threading"), they can be stopped, and restarted (preemption).

And the lines are blurring: Top500 #1 is a **pure-CPU** machine with 304 cores per socket[3]

[3] LineShine, Shenzhen (Top500, June 2026).

Some Terminology

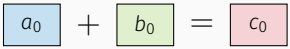
SIMD lane	Work on one data element
SIMD instruction	Work on multiple data at once
Hardware thread	Execute a list of SIMD Instructions

- Each Hardware Thread runs some **SIMD instructions**
- For example Intel PVC GPU has 3,584 hardware threads concurrently on 512 bits wide data [4]

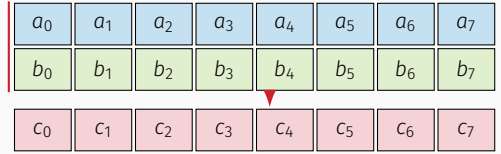
[4] Pro tips: All the HW have some register to know "where" you are running in the hardware. Not super public, but not secret either: *HW_REG_HW_ID2* for AMD and *intel_get_slice_id* for example

One instruction, many lanes

Scalar: 1 op → 1 result



SIMD: 8 ops → ONE instruction



one instruction,
one cycle

- Same opcode, same cycle — the **lane count is the SIMD width** (here 8; PVC is 16 floats wide).
- A **hardware thread** just streams a list of these wide instructions.

How do you get so many hardware threads? Replication (same as HPC)

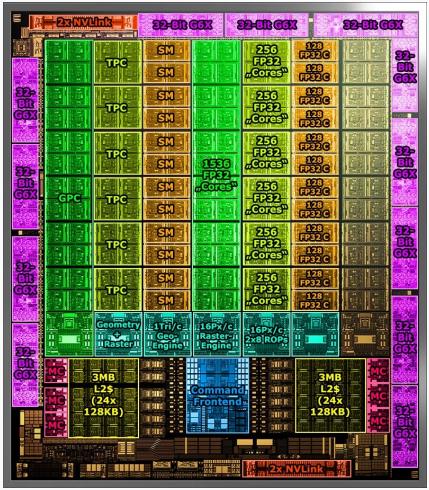


Figure 5: Annotated NVIDIA GA102 die. Image credit: Locuza

PVC Slice Example (not a real picture)

Terminology

No, I really don't want to talk about different vendors naming things differently for fun. It's not fun.

Just some stuff share a L1, then an L2, and then nothing. And then they are interconnected. That's it (SM, Slice, GCD, whatever, doesn't matter. What matters is the topology)

Let's still try...

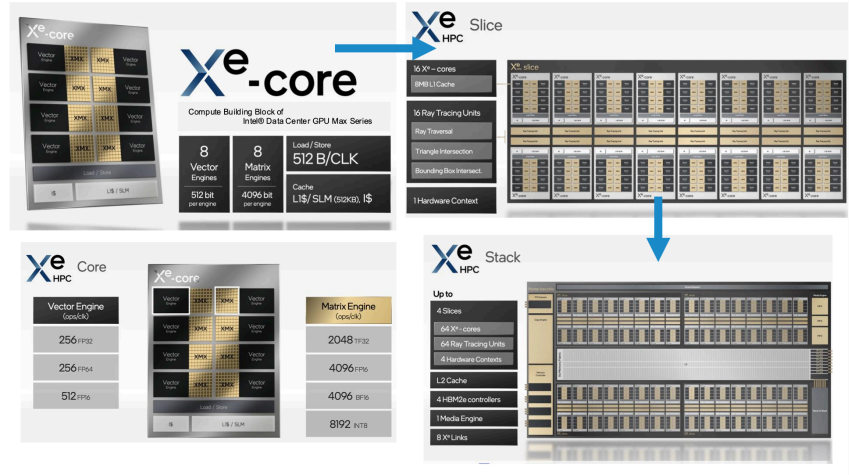


Figure 6: Intel Terminology

I prefer the “logical” view

- Forget the names. What matters: **what do they share**, and **can they synchronize**?
- **Inside one Xe Core**: shared L1 / shared-local-memory, and lanes can sync — this is the compute unit (the SM / CU equivalent).
- **Across Xe Cores / tiles**: only an L2, then the interconnect — no sync. Same story on every vendor.

Same thing, three vendors' names

Concept	NVIDIA	AMD	Intel
SIMD group (lanes lock-step)	warp (32)	wavefront (64)	SIMD8/16/32
Compute unit (shares L1, can sync)	SM	CU	Xe Core
Fast local memory	shared mem	LDS	SLM
Whole die / tile (no sync across)	GPU	GCD	Xe Stack

- The sync boundary is the row that matters: inside the compute unit you can sync + share fast memory; above it you cannot.
- Everything else is marketing.

The one big restriction: limited synchronization

- Inside **one hardware thread**: lanes can sync[5].
- Across **hardware threads** running on the same core: you can sync, and have access to some fast “shared-local-memory” aka “Managed L1 cache”
- Between cores, it’s not ok. No SYNC. There is no “global synchronization” on GPU. [6]

[5] Obviously as most of the time they executed a lock-step

[6] I mean you can always be non-portable and do so. But threads are not “preemptable” so they cannot hit the barrier, and “make room” for other threads. They are not, because then it’s not a GPU, it’s a CPU. And it would be too complex / expensive, submission overhead would be huge, etc etc. Trade-off overhead

Why that restriction is a gift

Because the hardware threads are independent, the scheduler is trivial:

- If the HW has nothing to execute, just take some work.
- A HW thread is never **preempted** — its registers stay live for its whole lifetime, so a context switch costs nothing (there is nothing to save/restore).
- So when one HW stalls (on memory for example), another runs. That is how latency is hidden, for free.

The restriction (no global sync) is exactly what buys the parallelism. Freedom vs. performance — the GPU takes performance. Fast to schedule.

Recap

- Lots of Threads running Concurrently
- Not that much synchronization possible
- The hard thing is not really programming the GPU, it's designing an algorithm that is "embarrassingly-parallel"
- But at the bottom of it, this is what HPC is all about: Extracting concurrency from your algorithm/problem

Outline

Let's talk about performance

Before we get started, just in case:

- **FLOP** = one floating-point operation (a count). One FMA $a{=}b{\cdot}c{+}d$ is **2 FLOP** (but one instruction). **FLOP/S**, is FLOP per Second[7]

We will assess the Peak FP32 of your favorite Hardware: One tile of PVC

[7] and don't use FLOPS... just confusing. And I will make this mistake many times, sorry.

Just need to have some data

- how many HW Threads we have,
- The size of the SIMD instruction (aka how many FLOP they can do in 1 instruction)
- And the Frequency
- Then just multiply

Step 1 — ask the hardware (`ze_info`)

- Get a node
- Then run `ZE_AFFINITY_MASK=0.0 ze_info`
- ...
- Profit (give me a number on slack). The people who have it correctly (and can explain) win a sticker!

Step 1 — ask the hardware (`ze_info`): Solution

One tile (*ZE_AFFINITY_MASK=0.0*):

```
Core Clock Rate             1500MHz
Physical EU SIMD Width      16      # FP32 lanes (8 for FP64)
Number EU Per SubSlice      8       # vector engines / Xe-Core
Number SubSlices Per Slice  56      # Xe-Cores per tile
Number Slices               1       # one tile
```

Yeah, enjoy different terminology. Fun isn't it?

Step 2 — derive the theoretical peak

Nothing magic — just multiply the fields (FP32, one tile):

$$\underbrace{16}_{\substack{\text{SIMD width}\\(\text{FP32 lanes})}} \times \underbrace{8}_{\substack{\text{vector eng.}\\/\ \text{Xe-Core}}} \times \underbrace{56}_{\text{Xe-Cores}} \times \underbrace{2}_{\text{FMA}} \times \underbrace{1.5}_{\text{GHz}} = \mathbf{21{,}504}\ \text{GFLOP/s}$$

Next question: How much of this we can reach? Any guess?

Step 3 — measure it

- I give you 3 binaries, each to measure one key characteristic.
- By default they use one HW thread, you can pass as argument the number you want
- Enjoy compiling it. It's always one of the most fun parts...

How much? With how many HW threads used?

Extra credit (a gift): can you write one kernel that reaches **both** the compute and bandwidth peak at once? If somebody can do it, win a 1 dollar Cray Coin!

Step 3 — measure all three roofs

Run the three provided benchmarks; compare to what the hardware promised:

Benchmark	Roof	Measured
flops	compute (21,504 theo.)	21,427 GFLOP/s (∼99.6%)
triad	HBM bandwidth	∼1,000 GB/s
pci	host ↔ device (PCIe)	∼75 GB/s

- 99% for HBM. Easy!
- **Feel the gap**: HBM is ∼10–100× slower than compute, PCIe another ∼10–100× below HBM.
- So yeah, compute fast, data-transfer slow.

Memory wall: data movement, not FLOPs

- Compute is cheap, **data movement is expensive**: the bottleneck is almost always memory, not FLOPs.
- **Discrete GPU**: separate memory, so you must ship data over PCIe. Slow — minimize transfers, keep data resident, recompute over reload.

Side story: discrete accelerator → integrated

The same cycle plays out every time, e.g. the floating-point unit:

- **Discrete first.** The x87 math coprocessor was a separate chip you socketed: 8087 (with 8086/8088), 80287 (286), 80387 (386).
- **Then integrated.** The 486DX (1989) put the FPU on the CPU die. Nobody thinks of "the FPU" as a separate thing anymore.

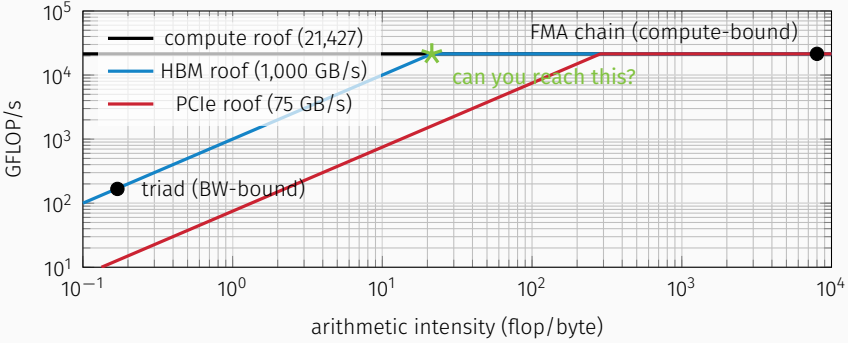
GPUs are following the exact same path: discrete PCIe cards today → integrated tomorrow (APUs, MI300A, Grace-Hopper…).

No silver bullet. Now we have NUMA effects. Maybe better than PCIe but not "free" either. Locality is, always will be, key for Performance

Arithmetic intensity: flop per byte

Arithmetic intensity = FLOP done / bytes moved. It decides whether you are compute- or memory-bound.

The roofline (one PVC tile, Single Precision)



Below AI ≈20 you are HBM memory-bound; below ∼286 you are PCIe-bound.

Just to be super clear

AI is per byte. So for each byte you read from main memory you need to do at least 20 FLOP:

- so you need at least 80 FLOP per float! Enjoy being compute bound guys.
- And you have at least 7168 simd-lanes to use. If you use less, you are not fully utilizing the full compute capability of one Tile.
- Oh, and we have 63744 GPUs on Aurora, so you just need to use 913_833_984 vector lines every cycle [8]

[8] So if your "problem space" is lower than that, it's mathematically impossible to use Aurora at max capacity

Conclusion: GPU in a nutshell

- A wide-SIMD, latency-hiding throughput machine.
- The naming does matter, it's just hierarchical. You have some levels where sync is free, some where it's expensive, some where it's impossible.
- Fast because it is simple: no big caches, no branch prediction, no fancy per-lane cleverness.
- Gap between Memory and Flops is huge. You need to compute, compute, compute
- GPUs are big. And we have many of them. So please extract the maximum concurrency of your simulation. And avoid reading from main memory[9]

[9] cache -> data locality, re-computing

Thanks!

I think we have a Q&A now.

Argonne NATIONAL LABORATORY

Higher Level Layer: SYCL/OpenMP/Kokkos

How does that even work? (ATPESC edition)

Thomas Applencourt(apl@anl.gov)

July 27, 2026

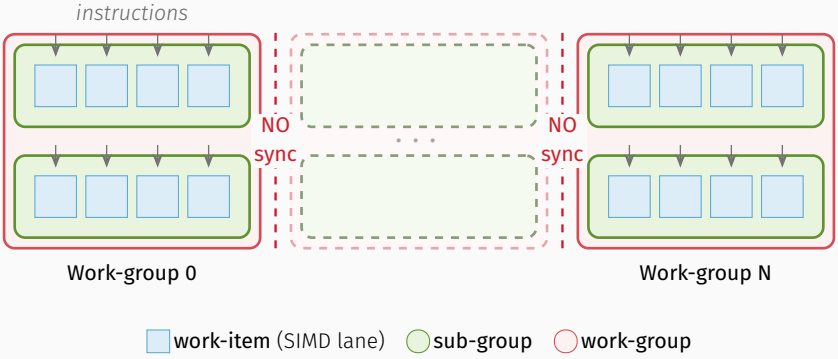
Outline

Before we start: some terminology so we speak the same language

- Work-Item, Sub-Group, Work-group

Now we can name it: the software hierarchy

Same picture as the hardware sync boundary — this time in **SYCL** words. Each **work-item** runs its own stream of instructions (top arrows).



Now we can name it: the software hierarchy

- **Work-item** inside a **sub-group**: sync is fast, lanes share registers (*`shuffle`* / cross-lane ops).
- **Sub-group** inside a **work-group**: share Shared Local Memory (SLM), can *`barrier()`*.
- Between **work-groups**: no synchronization possible — they are independent, may not even be co-resident.
- The model only guarantees **forward progress within one work-group** (conceptually “one at a time”). The hardware may run many at once — but you cannot rely on it.

Recap to previous session (hopefully)

- A **host program**, with a C API driving it: find device, load the binary, make a queue, allocate memory, copy, launch kernel, synchronize.
- The kernel is written into a separate file.

This afternoon: the same thing, but with syntactic sugar[10]

[10] And we have enough diabetes... Please don't make your own candy factory

[One C API to] in the darkness bind them

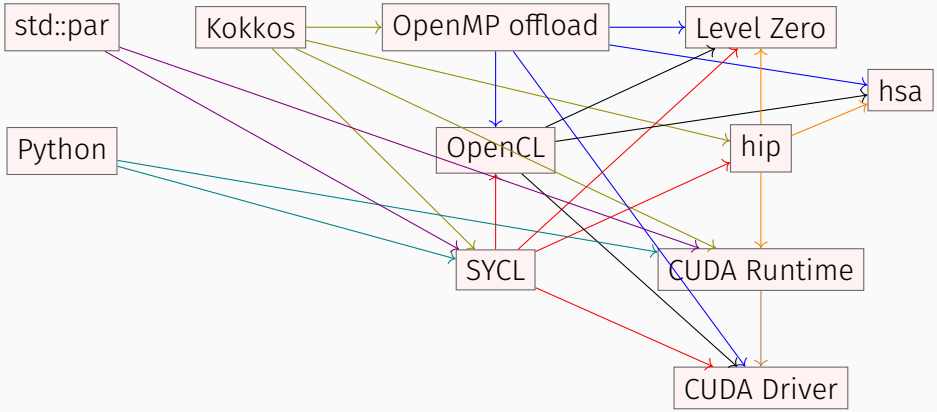
create/load/launch dance:

```
1  // Intel / OpenCL                          // NVIDIA / CUDA Driver
2  clCreateProgramWithIL(spirv,...);     |    cuModuleLoad(&m, "hello.ptx");
3  clCreateKernel(prog,"saxpy");         |    cuModuleGetFunction(&f,m,"saxpy");
4  clSetKernelArg(...); ...              |    cuLaunchKernel(f, grid, block, ...);
5  clEnqueueNDRangeKernel(q,k,...);
```

- I will not discuss the ‘«</»>‘ cuda syntax. [11]
- Explicit, just a C compiler and … **tedious**.
- Every high-level programming model is “this, wrapped.” Nothing more.
- Layers, layers everywhere

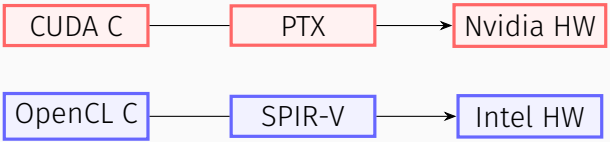
[11] It’s not valid C. You need a special compiler pass to parse that… Thanks NVIDIA

Talking about higher-level programming models[12]



[12] "Where is my Vulkan?" Sorry gamer people... OpenCL → Vulkan exists. Rusticl!

Note on compilation: two compilers



- You can generate PTX, or SPIR-V, from multiple “source languages”
- Rust, Fortran, Python, whatever.

Single-source: let's build our own. It's not magic

Let's build a 40-line version of this. Your own little single-source, higher-level language![13]

- Tell the compiler what is a kernel
- It will generate the SPIR-V for you
- Wrap everything in a tiny runtime

Let me introduce you to THOMAS:

- **T**iny — **H**eterogeneous — **O**penCL — **M**acro — **A**TPESC — **S**PIR-V

[13] Please don't do that at home, just for illustration purposes, to remove the magic. It's not magic, just painful to maintain

Single-source: let's build our own — THOMAS

Our own little single-source wrapper.[14]

```
#include "thomas.hpp"
KERNEL(triad, (float a, gptr<const float> x, gptr<const float> y, gptr<float> z),
       { z[i] = x[i] + a*y[i]; })
int main() {
  thomas::init();                          // pick the GPU
  float *x = thomas::alloc<float>(N);      // USM: one pointer, host + device
  float *y = thomas::alloc<float>(N), *z = thomas::alloc<float>(N);
  for (size_t i = 0; i < N; ++i) { x[i] = 2.0f; y[i] = 1.0f; }
  thomas::parallel_for(N, triad, 2.0f, x, y, z);   // bind args, launch on GPU
}
```

[14] THOMAS: **T**iny **H**eterogeneous **O**penCL **M**acro, **A**TPESC + **S**PIR-V.

How it works: one source, compiled twice

The same file is compiled twice — the *KERNEL* macro expands differently each time:

- **Device pass**: *icpx -x cl -cl-std=clc++ -DTHOMAS_DEVICE* turns the lambda into a *__kernel* and lowers it to SPIR-V.
- **Host pass**: plain *icpx* keeps only the kernel name, embeds the SPIR-V, and drives OpenCL.
- *gptr<T>* is *__global T** on the device, plain *T** on the host — one signature, both passes.

```
1  // DEVICE pass -- lambda body becomes a __kernel
2  #define KERNEL(name, params, body) \
3    __kernel void name params { auto _k=[&](ulong i) body; _k(get_global_id(0)); }
4
5  // HOST pass -- keep only the name to find the compiled kernel
6  #define KERNEL(name, params, body) ::thomas::Kernel name{#name};
```

Not magic (2/2): the whole “runtime” is a few calls

The device pass emits SPIR-V; the host side just loads it and launches. That is the entire “runtime”:

```
// load our compiled SPIR-V and JIT it for whatever device we picked
prog = clCreateProgramWithIL(ctx, thomas_spv, thomas_spv_len, &e);
clBuildProgram(prog, 1, &dev, ...);
// parallel_for: bind args, launch N work-items
clSetKernelArg(...);  clEnqueueNDRangeKernel(q, k, 1, 0, &N, ...);
```

Built twice (device→SPIR-V, host), the triad ran on one PVC tile — right on top of SYCL, because it is the same thing (resident USM + SPIR-V):

```
THOMAS    N=1048576  min_ms=0.013  BW=960 GB/s  residual=0
SYCL      N=1048576  min_ms=0.013  BW=960 GB/s  residual=0
```

∼40 lines over OpenCL + SPIR-V. That is all SYCL/Kokkos are, plus a lot of bug fixes, syntactic sugar, and runtime optimization

I will now show you some programming models

Comparative literature!

I will show you

- SYCL (the best)
- Kokkos (the best)
- OpenMP (the best)
- std-par (the best)
- Python[15]

[15] not the best, sorry, I'm just old. Using a scripting language for HPC just seems wrong. I don't like Jupyter notebooks either, so this one is maybe on me

The models, one by one

The next few slides: who backs each model, what it looks like, and what it runs on. Same kernel everywhere — only the packaging changes.

Model	Kind	Backed by
SYCL	open standard, pure C++	Khronos
Kokkos	C++ portability library	Sandia / HPSF
OpenMP	directive (pragma) standard	OpenMP ARB
std::par	ISO C++ standard library	ISO C++ vendors
Python	array libraries / bindings	

SYCL — the open C++ standard

- Backed by **Khronos**[16] — an open standard, not one vendor's product.
- **Pure C++**: a light abstraction over the native backend, with full control when you want it.
- Two main implementations:
  - **Intel oneAPI DPC++** — github.com/intel/llvm
  - **AdaptiveCpp** (higher-level, ex-hipSYCL) — github.com/AdaptiveCpp/AdaptiveCpp
- Backends: **CUDA, OpenCL, Level Zero, FPGA**.

Spec & overview: khronos.org/sycl

[16] The OpenCL / Vulkan / SPIR-V people.

Kokkos — the C++ portability library

- Developed at **Sandia National Laboratories**; now governed by the **High Performance Software Foundation** (part of the Linux Foundation).
- **C++** with a portability abstraction on top (*`View`*, execution/memory spaces).
- Ships as a **library** — super easy to install (*`module load`* or CMake).
- Backends: **SYCL, CUDA, HIP**, and an **OpenMP host** backend.

Home & docs: kokkos.org

OpenMP — directives (pragmas)

- **Pragma-based**: you annotate a loop, the compiler offloads it — no library API to learn.
- Works from **C, C++, and Fortran**[17] — the only model here that covers Fortran.
- Descriptive: you still write the loop; the pragma says “run this on the device.”

Spec: openmp.org

[17] Sad face? Happy face? I let you decide.

`std::par` — it's just ISO C++

- **No pragma, no library**: parallel algorithms are in the C++ standard itself (C++17).
- Pass an **execution policy** (`std::execution::par_unseq`) to `std::for_each`, `transform`, …
- The compiler/runtime decides where it runs — on Intel, that's the GPU.

Reference: cppreference: execution policies

Python — multiple GPU array bindings

- Not one thing: several **NumPy-style array libraries** that run on the GPU.
- **dpnp** (Intel, SYCL/Level Zero) — github.com/IntelPython/dpnp
- **CuPy** (NVIDIA/AMD) — cupy.dev
- Same `a*x + y` you'd write in NumPy — it just lands in a GPU kernel.

Same kernel, three models — C++ (all run on PVC)

SYCL (Khronos standard)

```
sycl::queue Q;
auto x = malloc_shared<float>(N,Q);
auto y = malloc_shared<float>(N,Q);
Q.parallel_for(N, [=](auto i){
  y[i] = a*x[i] + y[i];
}).wait();
```

Kokkos (portability library)

```
Kokkos::View<float*> x("x",N), y("y",N);
Kokkos::parallel_for(N,
  KOKKOS_LAMBDA(size_t i){
    y(i) = a*x(i) + y(i);
});
Kokkos::fence();   // async; sync to
↪  time/read
```

OpenMP target (directives)

```
#pragma omp target teams \
  distribute parallel for \
  map(to:x[0:N]) map(tofrom:y[0:N])
for (int i=0;i<N;i++)      // explicit
  y[i] = a*x[i] + y[i];    // loop!
```

Squint — they are the same kernel. Same index space, same lambda body, same $y = ax + y$.

Same kernel, higher still — standard languages & Python

C++ **`std::par`** (ISO C++, no pragma)

```
std::for_each(std::execution::par_unseq,
  idx.begin(), idx.end(),
  [=](size_t i){ y[i]=a*x[i]+y[i]; });
```

Fortran **`do concurrent`** (ISO Fortran)

```
do concurrent (i = 1:N)
   y(i) = a*x(i) + y(i)
end do
```

Python (dpnp)

```
import dpnp
x = dpnp.ones(N, dtype=dpnp.floa
y = dpnp.full(N, 2.0, dtype=dpnp
y = a*x + y           # runs on t
```

Some difference

	Kernel is	Data	Default submit
SYCL	lambda (C++)	USM pointers / buffers	**async**
Kokkos	lambda (C++)	*`View`* + spaces	**async**
OpenMP	**explicit loop**	*`map`* clauses	**blocking**
`std::par`	lambda (C++)	normal containers	**blocking**

OpenMP *`target`* is blocking by default; add *`nowait`*+*`depend`* for async. SYCL's default queue is the only **out-of-order** one; the rest are effectively in-order.

- **OpenMP is descriptive**: you write the loop, the pragma offloads it. The others hide the loop inside *`parallel_for`*.
- **Async default** (SYCL/Kokkos) vs **blocking default** (OpenMP) — this drives performance (we come back to it in Act 4).

Where does the data live? Three styles

The kernel is only half the story — you also decide how memory moves.

1. **Raw pointer, explicit.** You allocate on the device and copy yourself. SYCL USM, OpenMP 6 allocation, our THOMAS. Most control, most boilerplate.
2. **OpenMP map clauses.** *`map(to:/from:/tofrom:)`* — the runtime tracks what is already present and moves the rest. Depends on the surrounding *`target data`* region.
3. **Implicit objects.** *`sycl::buffer`*[18], Kokkos *`View`* / *`DualView`*, the object owns the data and the runtime moves it where a kernel needs it.

[18] We will not talk about those, nobody uses those

One pointer, three flavors

Khronos[19]	CUDA / HIP	What it means
device memory	device memory	only accessible by the **device**
shared memory	managed memory	accessible by **host and device**; data may migrate between them
host memory	pinned memory	accessible by **both**; data stays on the host (can speed up transfers)

[19] In Khronos land we call all three **Unified Shared Memory** (USM); it just means "pointer access." *`shared`* and *`host`* are a subset of USM

Kokkos adds data abstractions: `View` & `DualView`

SYCL/OpenMP move data with pointers. Kokkos provides some abstractions:

```
1  Kokkos::View<float*, MemSpace> d("d", N);   // layout + space chosen for the arch
2  Kokkos::DualView<float*> v("v", N);         // a host mirror AND a device copy
3  v.modify_device();                          // lazy H2D: just marks dirty, 0 bytes
4  v.sync_host();                              // sync, copy if required
```

- *`View`*: memory **space** (host/device) + **layout** (row/col-major) picked per architecture — portability of data, not just kernels.
- *`DualView`*: manage the host↔device copy explicitly, only sync when a side is dirty. (USM hides this; Kokkos gives you control.)
- Some of these abstractions were **upstreamed into the C++ standard** — e.g. *`std::mdspan`* (C++23). It's super nice.

`std::mdspan`: a Kokkos `View` that made it into C++23

A non-owning, multi-dimensional **view** over a flat buffer — index it like *`a(i,j)`*, no manual *`i*ncols+j`*:

```
#include <mdspan>
float buf[6];                                  // plain, flat storage
std::mdspan a{buf, 2, 3};                       // view it as 2 x 3

for (std::size_t i = 0; i < a.extent(0); ++i)  // extent(0)=2, extent(1)=3
  for (std::size_t j = 0; j < a.extent(1); ++j)
    a[i, j] = i * 10 + j;                        // C++23 multidim subscript
```

- Owns **nothing** — just shape + a pointer. Same idea as Kokkos *`View`*, now standard.
- Swap the **layout** (row- vs column-major) without touching the loop — exactly the portability trick Kokkos does per architecture.
- **Fortran** has had native N-D arrays since day one (1957); C++ only gets the view now, in 2023.[20]

[20] Better late than never.

Lambda vs pragma: what the compiler has to know

Everyone here needs a device compiler (our THOMAS used one too: `-x cl -cl-std=clc++`). The real split is **how the kernel is represented**.

- **SYCL / Kokkos / `std::par`: the kernel is a lambda** — a plain C++ object the library can capture, store and pass to `parallel_for`. So the layer above is ordinary library code (AdaptiveCpp even has a library-only path). The `parallel_for` we wrote in THOMAS is exactly that.
- **OpenMP / `do concurrent`: the kernel is a pragma / language construct.** `#pragma omp target` is not a C++ object — a library cannot see it. The **compiler** must recognize it and emit the device code.

Both need a toolchain to reach the device. A lambda is a value you can hand around; a pragma only the compiler can act on.

Let's run!

- I prepared a few code examples that do the triad
- Let's run them. And let's see who is faster
- And let's discuss their differences, similarities. Answer any question

Standards are good (please use them)

The **open, vendor-neutral** column, mostly from Khronos:

Layer	Open standard	Proprietary cousins
IR	**SPIR-V**	PTX
Low-level	**OpenCL, HSA**	CUDA Driver, Level Zero
High-level	**OpenMP, SYCL, Kokkos**	Whatever NVIDIA and AMD are pushing for[21].

- Break free from the vendor-provided ecosystem!
- The only advantage is that they work...

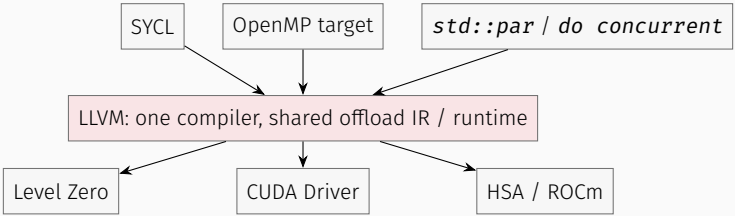
[21] Don't get me started on OpenACC...

Please don’t reinvent the abstraction layer

We just saw a layer is “only” ∼40 lines. And then a few C++ abstractions. Easy!

- Everyone builds their own ⇒ **dilution of effort**: N half-finished runtimes, each thin, each buggy.
- Each new layer carries **workarounds for the bugs** of the layer below
- If you find a gap in the abstraction, or a bug, please **contribute to a shared one**.
- Kokkos is the good example: they needed some library, they pushed it into the C++ main spec (thanks mdspan!)

It is already converging: one compiler, one offload runtime



- SYCL is being **upstreamed into mainline LLVM** (Intel's *intel/llvm* tracks LLVM *main*, 1–2 weeks behind).
- SYCL and OpenMP increasingly share the same LLVM offload plumbing[22] which route to the native backend: **Level Zero**, **CUDA Driver**, **HSA**.

[22] liboffload

Which one should I use?

- **Fortran?** → **OpenMP** (or *`do concurrent`* for standard-language offload). SYCL/Kokkos are C++-only.
- **C++, want close to the metal / an open standard?** → **SYCL**. Lower-level, maps almost 1:1 to the native runtime.
- **C++, want data abstractions (`View`, layouts, `DualView`)?** → **Kokkos**. Higher-level; sits on CUDA / HIP / SYCL. (Portability is a wash — all three already cover the three vendors.)
- **Prototyping / data science / ML?** → **Python** (PyTorch, dpnp) — same runtime underneath (SYCL on Intel, CUDA on NVIDIA; it does not reinvent either).

No wrong answer. Just use what you have support for.

Exp 2: benchmarks are always misleading

- I have 2 experiments. Watch, SYCL is the best!
- Let's see if we can flip the conclusion, the result
- Always take benchmarks with a grain of salt

Conclusion

- “same same, but different”
- If a gap exists, most probably it can be fixed in your code
- If not, it’s probably a bug. Please report it
- My personal opinion:
  - DON’T REINVENT ONE MORE
  - Use the one where you have more support

Last word

The offload model is genuinely portable: between models and between vendors, migration is cheap. The hard part is getting into the model. You must extract your compute kernels:

- Isolate the kernel; hopefully an “embarrassingly-parallel” pure function
- Port the memory management

Acknowledgments

ARGONNE TRAINING PROGRAM ON EXTREME-SCALE COMPUTING

Produced by Argonne National Laboratory, a U.S. Department of Energy Laboratory managed by UChicago Argonne, LLC under contract DE-AC02-06CH11357.

Special thanks to the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the use of their resources during the training event.

extremecomputingtraining.anl.gov