Approaches to GPU programming

Ashot Vardanian

github.com/ashvardanian/SandboxGPUs

Approaches to GPU programming for AI

Ashot Vardanian

github.com/ashvardanian/SandboxGPUs

Approaches to GPU programming for AI using OpenCL, Halide, SyCL

Ashot Vardanian

github.com/ashvardanian/SandboxGPUs

Who am I?

Ashot Vardanian, 24 y.o.
First OpenGL line at the age of 15

Worked on:

- Web
- Mobile
- Desktop
- Scientific Computing

Working on:

- High Performance Computing
- Artificial Intelligence Research

linkedin.com/in/ashvardanian
fb.com/ashvardanian

Who is this talk for?

You are familiar with C/C++.

You know what a GPU is.

You want to do number-crunching:

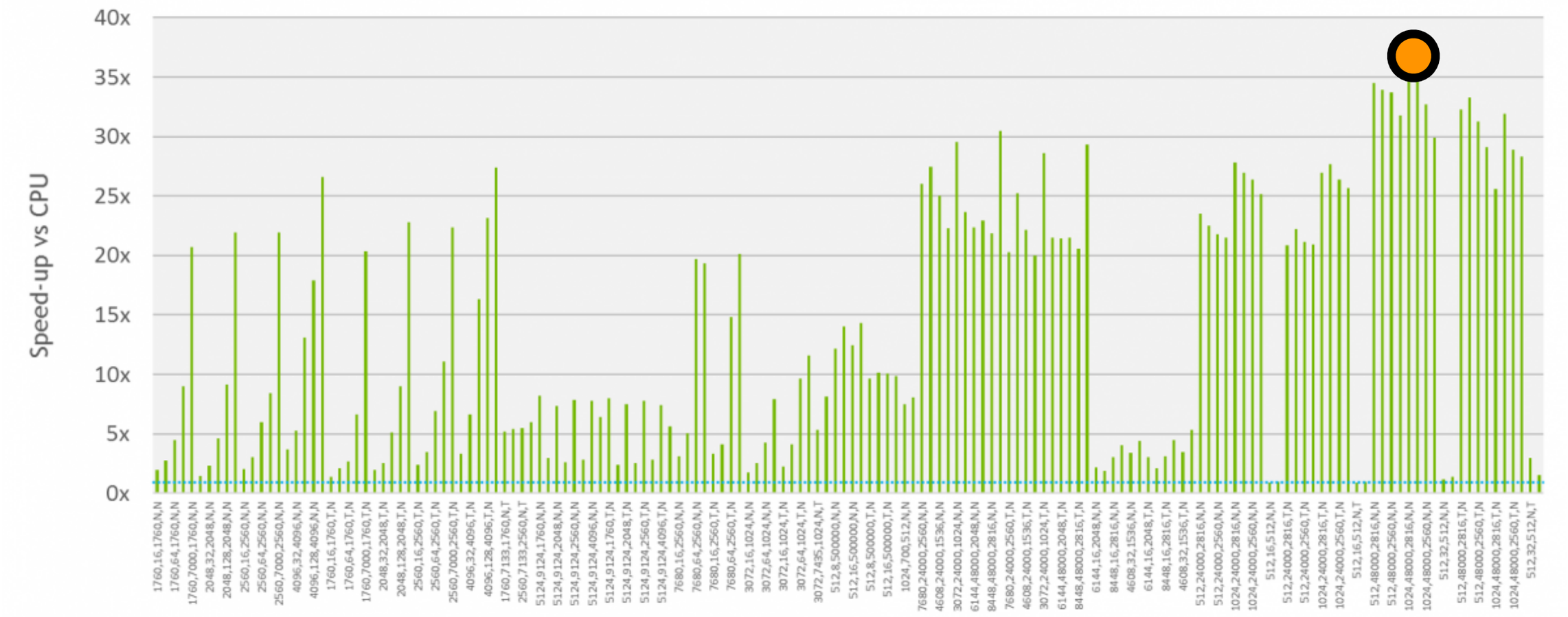
- AI & Datascience,
- Video Processing,
- Physics & Bio Simulations.

Not about benchmarking,

but about architecture!

Why GPUs?

...I have heard we can get a 35x performance increase...



ond: 241
e: 4144 us
e: 2512 us
5.9 mb
7.8 mb
41616
256

[Hold

Ground Truth

Our Method

ond: 2

Frames Per Second: 3861

[Hold

Frames Per Second:	114
CPU Frame Time:	8704 us
GPU Frame Time:	7680 us
CPU Memory:	0.3 mb
GPU Memory:	1.8 mb
Total Vertices:	640768
PCA Basis:	64

What we want?

Write code once ,

but deploy everywhere!

Optimise performance ,

but avoid boilerplate!

What we want?

Write code once , .cpp

but deploy everywhere!

Intel, Nvidia GPUs, AMD, Xilinx FPGA

Optimise performance ,

but avoid boilerplate!

What we want?

Unified Language

Write code once ,

Modular Compilers

but deploy everywhere!

Tune code without rewriting logic

Optimise performance ,

but avoid boilerplate!

Clean APIs

Comparison of recipes

...we will fill this table:

	Simple	Unified	Flexible	Clean
Technology	?	?	?	?
Write code once	?	?	?	?
Deploy everywhere	?	?	?	?
Optimise performance	?	?	?	?
Avoid boilerplate	?	?	?	?

We have a plan!

The Plan

1. **Popular APIs & Langs:**
    1. **OpenGL,**
    2. **OpenCL.**
2. Writing Low-level code
3. Existing Libraries & Tools
4. Optimal Recipes

The Plan

1. Popular APIs & Langs
2. **Writing Low-level code:**
    1. **OpenCL Language,**
    2. **CUDA Language,**
    3. **GLSL.**
3. Existing Libraries & Tools
4. Optimal Recipes

The Plan

1. Popular APIs & Langs
2. Writing Low-level code
3. **Existing Libraries & Tools:**
    1. **Linear Algebra,**
    2. **Lazy Evaluation,**
    3. **Halide,**
    4. **SyCL.**
4. Optimal Recipes

The Plan

1. Popular APIs & Langs
2. Writing Low-level code
3. Existing Libraries & Tools
4. **Optimal Recipes.**

CPU - GPU communication

Platform Support

	OpenGL
Release	1992, SGI
Intel	**Yes**
AMD	**Yes**
Nvidia	**Yes**
Apple	Deprecated
Android	**Yes**

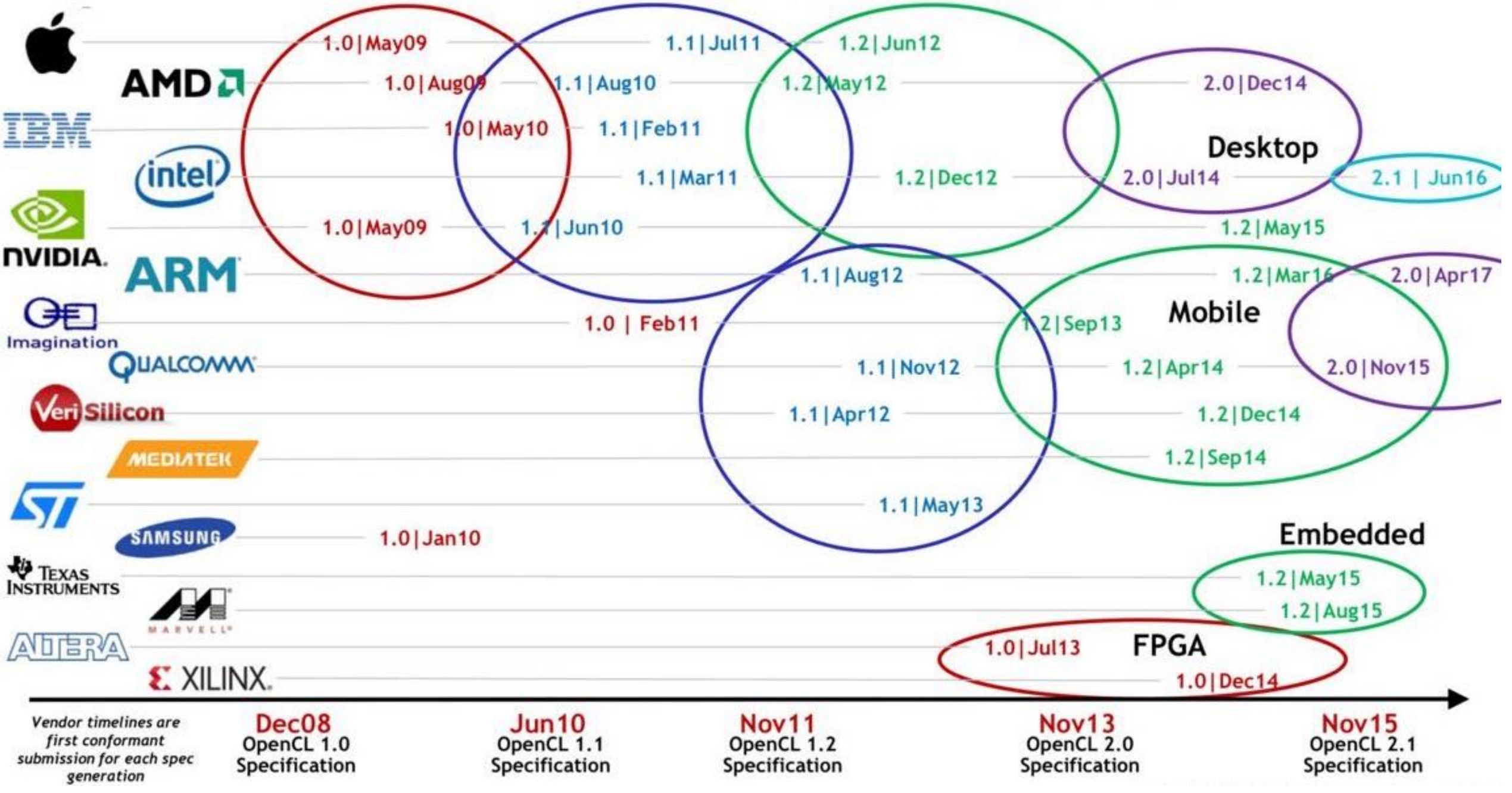
Platform Support

	OpenGL	CUDA
Release	1992, SGI	2007, Nvidia
Intel	**Yes**	No
AMD	**Yes**	No
Nvidia	**Yes**	**Yes**
Apple	Deprecated	No
Android	**Yes**	No

Platform Support

	OpenGL	CUDA	OpenCL
Release	1992, SGI	2007, Nvidia	2009, Apple
Intel	**Yes**	No	**Yes**
AMD	**Yes**	No	**Yes**
Nvidia	**Yes**	**Yes**	**Yes**
Apple	Deprecated	No	MacOS
Android	**Yes**	No	Depends

OpenCL Support (2015)



Platform Support

	OpenGL	CUDA	OpenCL	Metal
Release	1992, SGI	2007, Nvidia	2009, Apple	2014, Apple
Intel	**Yes**	No	**Yes**	No
AMD	**Yes**	No	**Yes**	No
Nvidia	**Yes**	**Yes**	**Yes**	No
Apple	Deprecated	No	MacOS	**Yes**
Android	**Yes**	No	Depends	No

Platform Support

	OpenGL	CUDA	OpenCL	Metal	Vulkan
Release	1992, SGI	2007, Nvidia	2009, Apple	2014, Apple	2016, AMD
Intel	**Yes**	No	**Yes**	No	**Yes**
AMD	**Yes**	No	**Yes**	No	**Yes**
Nvidia	**Yes**	**Yes**	**Yes**	No	**Yes**
Apple	Deprecated	No	MacOS	**Yes**	MoltenVK
Android	**Yes**	No	Depends	No	**Yes**

Comparison

	OpenGL	CUDA	OpenCL	Metal	Vulkan
Primary Purpose	Graphics	**Compute**	**Compute**	Graphics	Graphics
Base Input Language	C	**C++**	C	**C++**	**Any**
Learning Curve	Hard ...on Device	**Easy**	**Easy**	Average	Very Hard
Targets Flexibility	Average	Low ...only Nvidia	**Extreme ...FPGA**	Low ...only Apple	**High**
API Flexibility*	Average	**High****	Average	Average	**High**

Comparison

		CUDA	OpenCL		Vulkan
Primary Purpose		**Compute**	**Compute**		Graphics
Base Input Language		**C++**	**C**		**Any** SPIR-V
Learning Curve		**Easy**	**Easy**		**Very Hard**
Targets Flexibility		**Low ...Nvidia**	**Extreme ...FPGA**		**High**
API Flexibility*		**High****	Average		**High**

CUDA <=> OpenCL

```
void sum_2_vecs(float const * xA,
                float const * xB,
                float * y,
                int const xLen);
```

```
kernel
void sum_2_vecs(global float const * xA,
                global float const * xB,
                global float * y);
```

Parallelism in Language

Which keywords and features must a language have to make parallel programming easy?

Syncronization Primitives

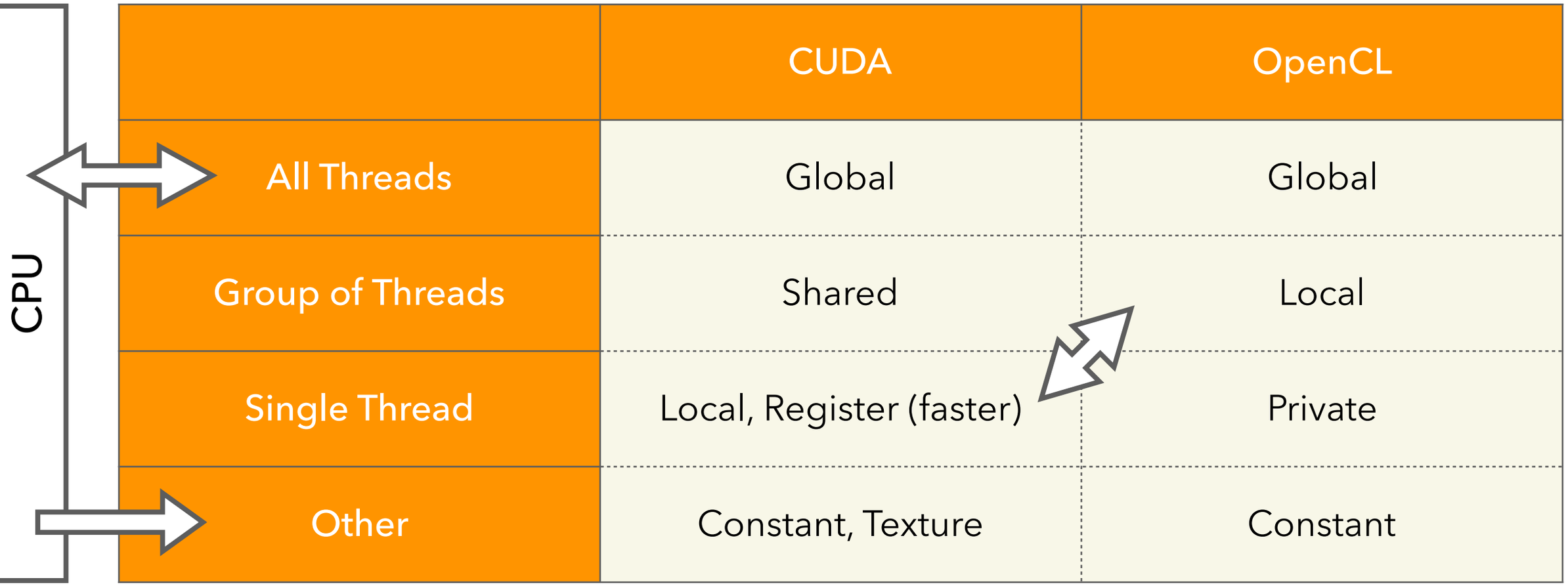
To help threads understand their role

Memory Qualifiers

To limit data visibility

?

Memory Types



OpenGL	Vertex Buffer	Frame Buffer	Texture	Local

Actual Memory Types

...have little to do with physical capabilities of the device! At least from OpenCL perspective!

	i7-7820HQ	Titan V	Radeon Pro 560
Compute Units	8 cores	80 cores	16 cores
Sync-able Group	<1024 threads $\mathbb{N}^1$	<1024 threads $\mathbb{N}^3$	< 256 threads $\mathbb{N}^3$
Constant Buffer	64 Kb	? Kb	64 Kb
Local Memory	32 Kb	? Kb	32 Kb

1 Mb L2

16 Kb L1 per CU

Actual Memory Types

...have little to do with physical capabilities of the device! At least from OpenCL perspective!

	i7-7820HQ	Titan V	Radeon Pro 560
Compute Units	8 cores	80 cores	16 cores
Sync-able Group	<1024 threads ℕ1	<1024 threads ℕ3	< 256 threads ℕ3
Constant Buffer	64 Kb	"In Volta the L1 cache, texture cache, and shared memory are backed by a combined 128 KB data cache."	64 Kb 1 Mb L2
Local Memory	32 Kb		32 Kb 16 Kb L1 per CU

Nvidia GPUs have one real "constant" buffer (64-128 Kb) and allocate rest in global memory.

AMD GPUs often have multiple "constant" buffers (64 Kb each) and allocate rest in global memory.

Memory Qualifiers

	CUDA	OpenCL
All Threads	__device__	__global
Group of Threads	__shared__	__local
Single Thread	~	~
Other	__constant__	__constant

CPU

```
void sum_2_vecs(float const * xA,
                float const * xB,
                float * y,
                int const xLen);
```

GPU version

```
kernel
void sum_2_vecs(global float const * xA,
                global float const * xB,
                global float * y);
```

Terminology

CUDA		OpenCL
Stream Multiprocessor	**Core on CPU**	Compute Unit
Thread	**Thread on CPU**	Work-Item
Block		Work-Group
__global__ function		__kernel function
__device__ function		~

Kernels Indexing

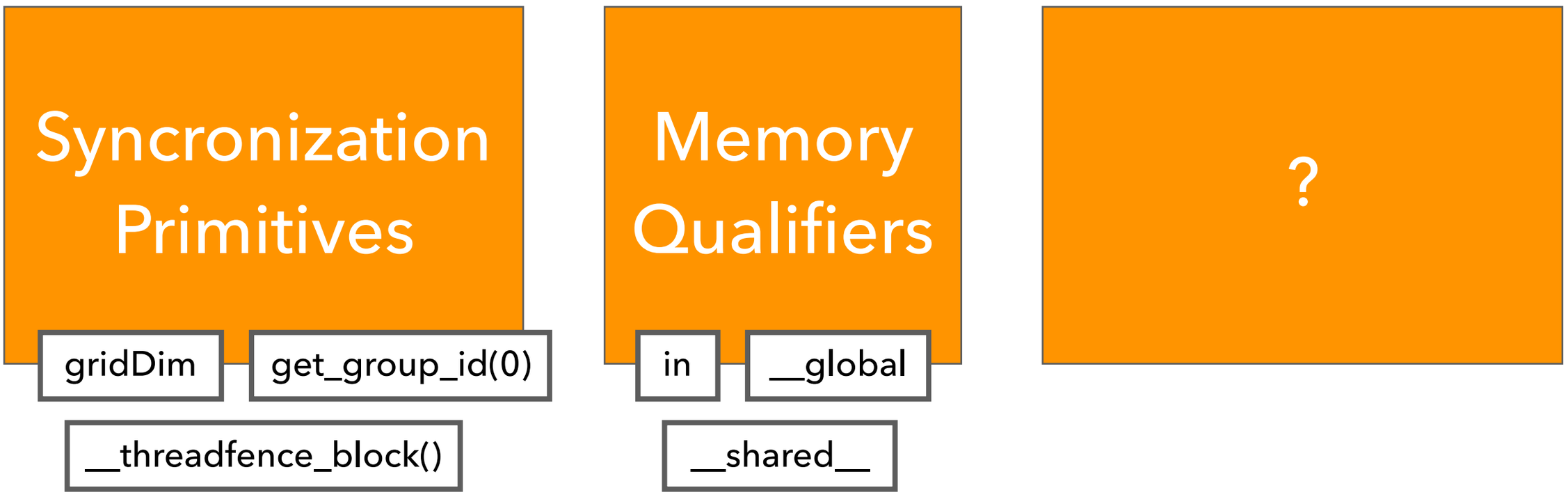
CUDA	OpenCL
gridDim	get_num_groups()
blockDim	get_local_size()
blockIdx	get_group_id()
threadIdx	get_local_id()
Ugly blockIdx * blockDim + threadIdx	get_global_id()
Ugly gridDim * blockDim	get_global_size()

Kernels Synchronization

	CUDA	OpenCL
Group	__syncthreads()	barrier(...)
All	__threadfence()	~
Group Memory	__threadfence_block()	mem_fence(...)
	~	read_mem_fence(...) ?
	~	write_mem_fence(...) ?

Parallelism in Language

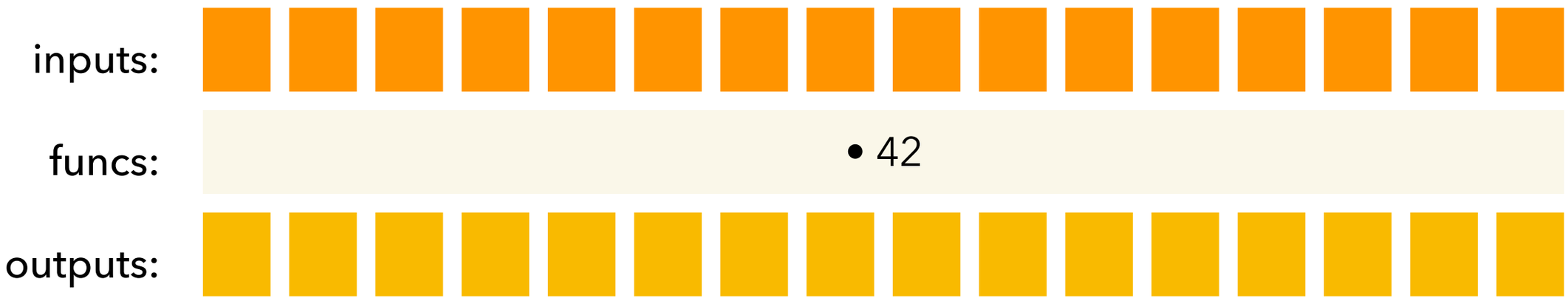
Which keywords and features must a language have to make parallel programming easy?



Bare Metal Examples

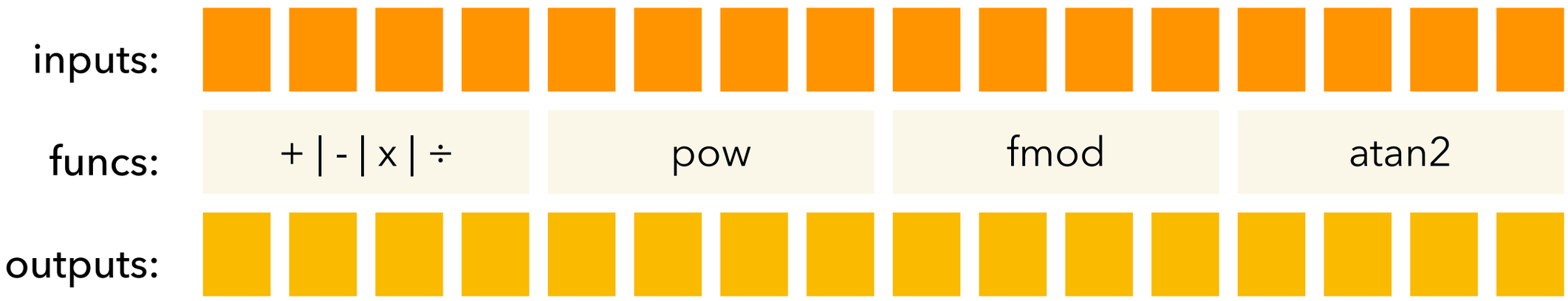
Data-Parallel Tasks

...brute-force scaling of simple
non-concurrent problems



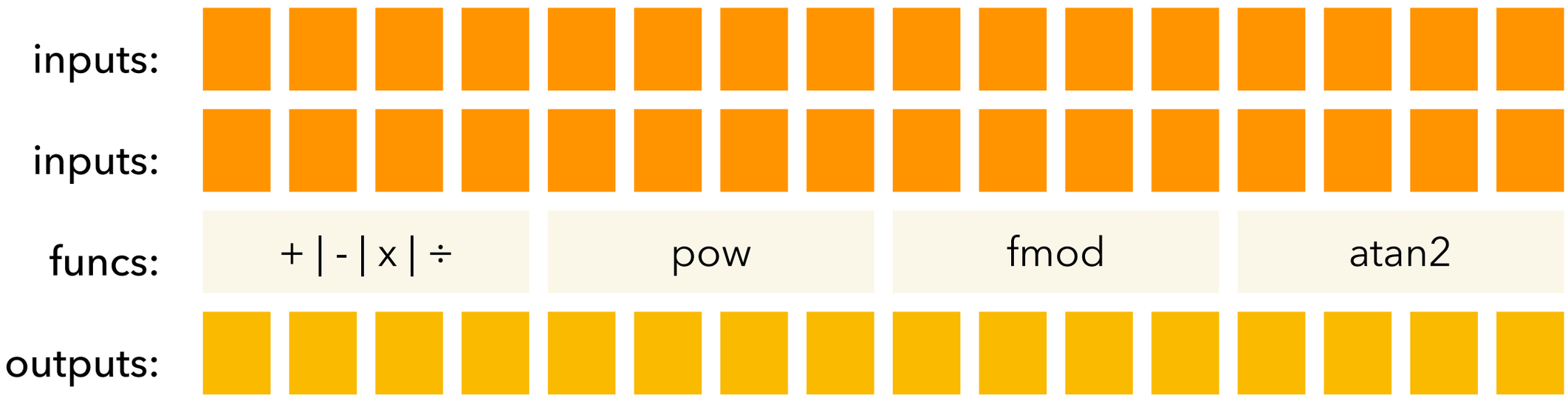
Data-Parallel Tasks

...brute-force scaling of simple
non-concurrent problems



Data-Parallel Tasks

...brute-force scaling of simple
non-concurrent problems



Vector Sum: C

```
void sum_2_vectors(float const * xA,
                   float const * xB,
                   float * y,
                   int const xLen) {
    for (int i = 0; i < xLen; i ++)
        y[i] = xA[i] + xB[i];
}
```

Vector Sum: OpenCL

```
kernel void sum_2_vectors(global float const * xA,
                          global float const * xB,
                          global float * y) {
    int i = get_global_id(0);
    y[i] = xA[i] + xB[i];
}
```

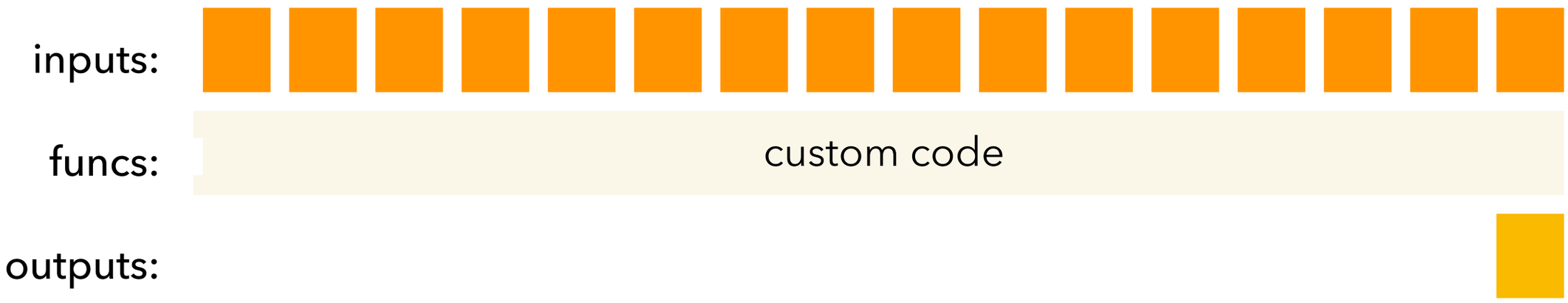
Vector Sum: GLSL

```
#version 450
layout(binding = 0) in buffer lay0 { float xA[]; };
layout(binding = 1) in buffer lay1 { float xB[]; };
layout(binding = 2) out buffer lay2 { float y[]; };

void main() {
    uint const i = gl_GlobalInvocationID.x;
    y[i] = xA[i] + xB[i];
}
```

Concurrent Tasks

...synchronization nightmare
and benchmarks heaven!

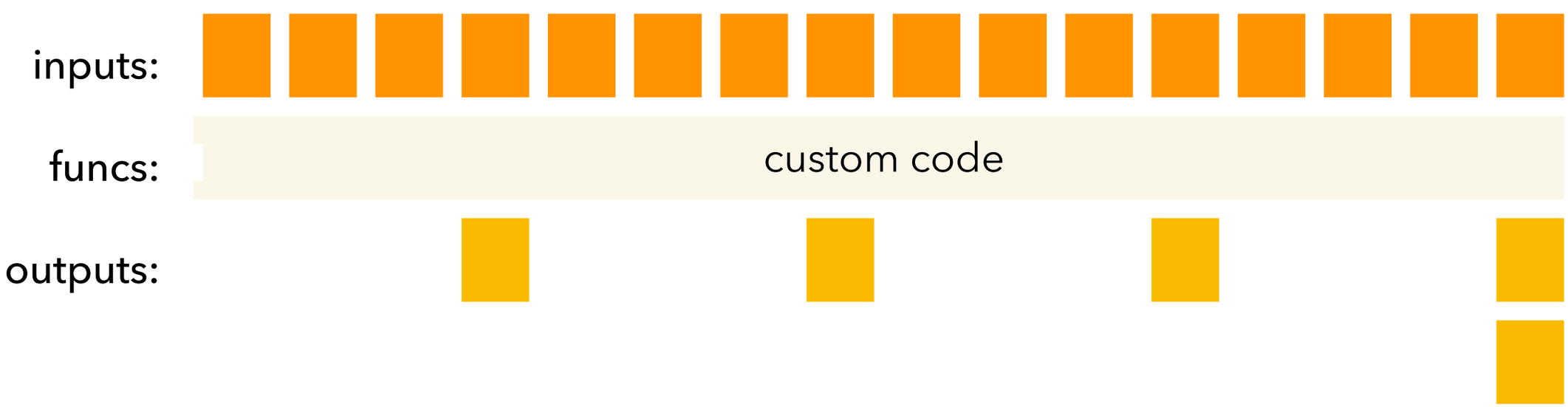


Reduction: C

```
void reduce(float const * x,
            float * y,
            int const xLen) {
    *y = 0;
    for (int i = 0; i < xLen; i ++)
        *y += x[i];
}
```

Concurrent Tasks

...force us to inject memory synchronization barriers and loops, that compiler won't unroll!



Reduction: OpenCL (1)

```
__kernel
void reduce_simple(__global float const * xArr, __global float * yArr,
                   int const xLen, __local float * mBuffer) {
    int const lIdxGlobal = get_global_id(0);
    int const lIdxInBlock = get_local_id(0);
    mBuffer[lIdxInBlock] = (lIdxGlobal < xLen) ? xArr[lIdxGlobal] : 0;

    barrier(CLK_LOCAL_MEM_FENCE);
    int lBlockSize = get_local_size(0);
    int lBlockSizeHalf = lBlockSize / 2;
    while (lBlockSizeHalf > 0) {
        if (lIdxInBlock < lBlockSizeHalf) {
           mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock + lBlockSizeHalf];
           if ((lBlockSizeHalf * 2) < lBlockSize) {
                if (lIdxInBlock == 0)
                    mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock + (lBlockSize - 1)];
            }
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        lBlockSize = lBlockSizeHalf;
        lBlockSizeHalf = lBlockSize / 2;
    }

    if (lIdxInBlock == 0) yArr[get_group_id(0)] = mBuffer[0];
}
```

Reduction: OpenCL (2)

```
__kernel
void reduce_unrolled(__global float const * xArr, __global float * yArr,
                     int const xLen, __local float * mBuffer) {
    int const lIdxInBlock = get_local_id(0);
    int const lIdxGlobal = get_group_id(0) * (get_local_size(0) * 2) + get_local_id(0);
    int const lBlockSize = get_local_size(0);
    mBuffer[lIdxInBlock] = (lIdxGlobal < xLen) ? xArr[lIdxGlobal] : 0;

    if (lIdxGlobal + get_local_size(0) < xLen)
        mBuffer[lIdxInBlock] += xArr[lIdxGlobal + get_local_size(0)];
    barrier(CLK_LOCAL_MEM_FENCE);

    #pragma unroll 1
    for (int lTemp = get_local_size(0) / 2; lTemp > 32; lTemp >>= 1)  {
        if (lIdxInBlock < lTemp)
            mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock + lTemp];
        barrier(CLK_LOCAL_MEM_FENCE);
    }

    if (lIdxInBlock < 32) {
        if (lBlockSize >=  64) { mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock + 32]; }
        if (lBlockSize >=  32) { mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock + 16]; }
        if (lBlockSize >=  16) { mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock +  8]; }
        if (lBlockSize >=   8) { mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock +  4]; }
        if (lBlockSize >=   4) { mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock +  2]; }
        if (lBlockSize >=   2) { mBuffer[lIdxInBlock] += mBuffer[lIdxInBlock +  1]; }
    }

    if (lIdxInBlock == 0) yArr[get_group_id(0)] = mBuffer[0];
}
```

Higher Layers

Linear Algebra

	Intel MKL	cuBLAS	CLBlast
Types	Basic	**Basic, FP16, INT8**	Basic, FP16
Performance	+	**+++**	++
APIs	BLAS, LAPACK…	BLAS +	BLAS
BLAS Levels	Vector-Vector	Matrix-Vector	Matrix-Matrix
LAPACK	Least Squares	Eigenvalues	Factorization

Optimized kernels are chained into slow pipelines!

Linear Algebra

	Intel MKL	cuBLAS	CLBlast
Types	Basic	**Basic, FP16, INT8**	Basic, FP16
Performance	+	**+++**	++
APIs	BLAS, LAPACK...	BLAS +	BLAS
BLAS Levels	Vector-Vector	Matrix-Vector	Matrix-Matrix
LAPACK	Least Squares	Eigenvalues	Factorization

152 Ops!

Vectors Sum?

Array Sum Reduction?

Optimized kernels are chained into slow pipelines!

Linear Algebra

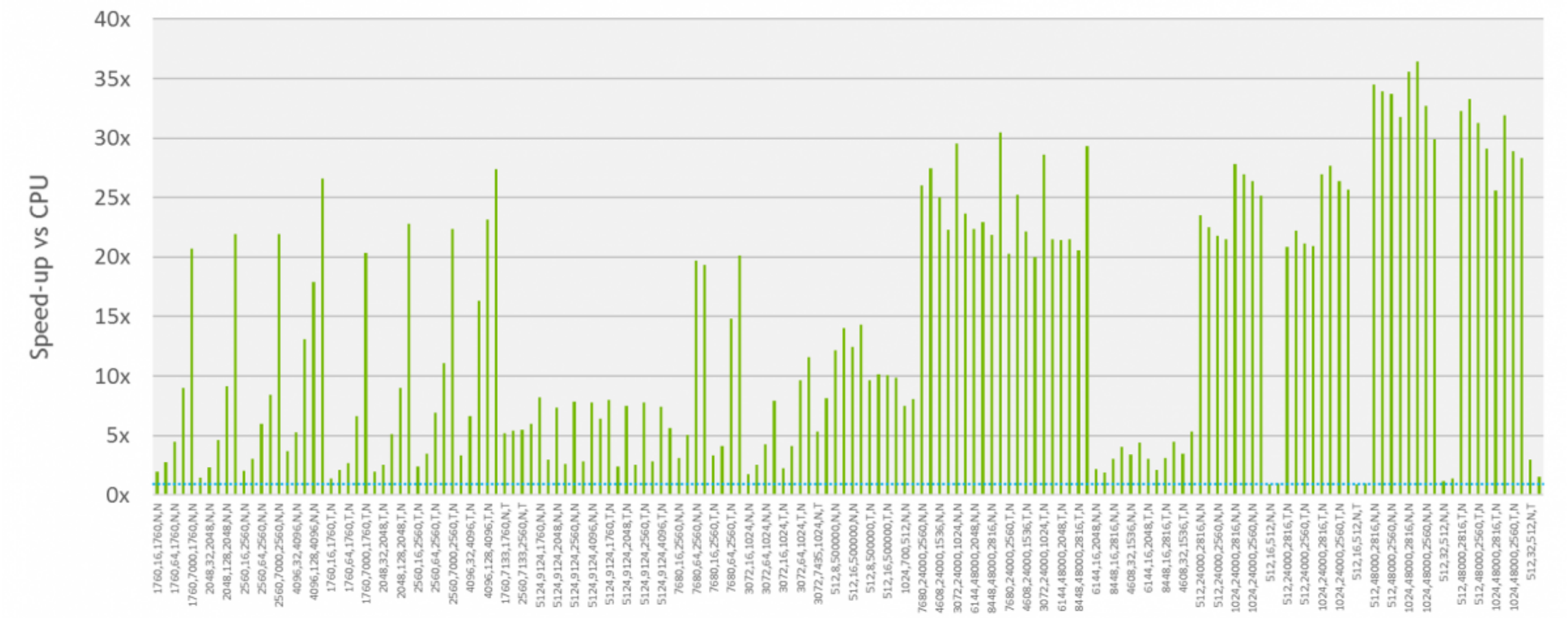
	Intel MKL	cuBLAS	CLBlast
Types	Basic	**Basic, FP16, INT8**	Basic, FP16
Performance	+	**+++**	++
APIs	BLAS, LAPACK…	BLAS +	BLAS
BLAS Levels	Vector Vector	Matrix Vector	Matrix Matrix
LAPACK	Least Squares	Eigenvalues	Factorization

152 Ops!

Vectors Sum?
SAXPY with a=1

Array Sum Reduction?
SDOT with unit vector

Optimized kernels are chained into slow pipelines!



	E5-2690v4	Gold 6262V	V100
Float Performance	+	++	**+++**
Cores	14	24	14
Year	2016	2019	2017
Price	2,000-2,500 USD	**3,000 USD**	8,000 USD

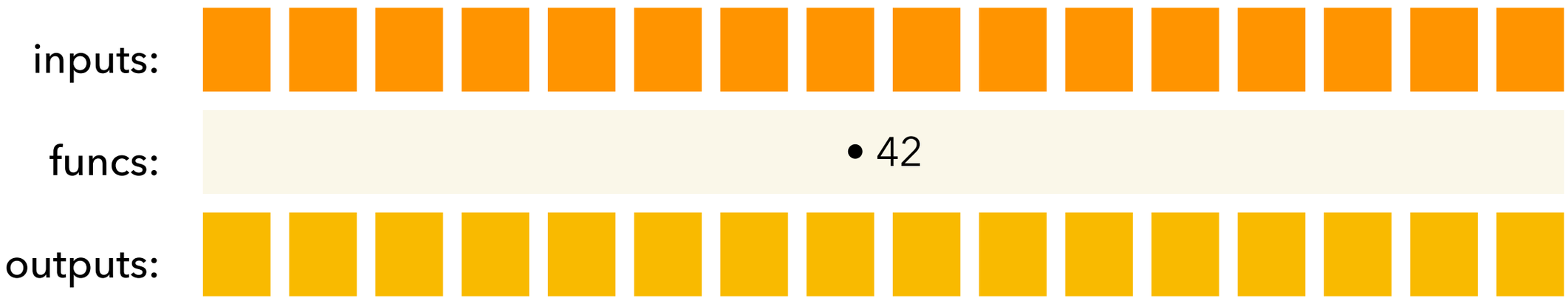
Lazy Evaluation Graph

	Eigen	ArrayFire	Boost. Compute	Thrust	VexCL
Stars	10k	2.8k	1K	2.5k	565
Type-Safe	**Yes**	No	Yes	Yes	Yes
Backends	OpenMP, CUDA?	**OpenCL, CUDA, etc.**	OpenCL	CUDA, OpenMP	OpenCL, CUDA, OpenMP

Very different functionality and inconsistent APIs.
Potential Licensing issues.

Data-Parallel Tasks

...again, but now with higher level heterogeneous computing tools!



Cost of Memory Access

...is much higher, than cost of compute, so we need **kernel fusion!**

	Power	
ALU	1 pJ	
Load from SRAM	3 pJ	
Move 10 mm on-chip	30 pJ	
Send off-chip	500 pJ	
Send to DRAM	1 nJ	1,000x more
Send over LTE	10 µJ	10,000,000x more

Parallelism in Language

...we want to separate the inner part of
the "for" loop and the enumeration order

Syncronization Primitives

To help threads understand their role

Memory Qualifiers

To limit data visibility

Order Descriptors

To simplify loops optimization

How Halide works?

...by separating the inner loop logic!

```
func(i) = lA(i) + lB(i);
```

How Halide works?

...and by making loops implicit!

```
void sum_2_vectors(float const * xA,
                   float const * xB,
                   float * y,
                   int const xLen) {
    Halide::Buffer<bFlt32> lA { const_cast<float *>(xA), xLen, "xA" };
    Halide::Buffer<bFlt32> lB { const_cast<float *>(xB), xLen, "xB" };
    Halide::Var i { "i" };
    Halide::Func func;

    func(i) = lA(i) + lB(i);

    Halide::Buffer<bFlt32> lOut = func.realize(xLen);
    std::copy_n(lOut.data(), xLen, y);
}
```

Function body

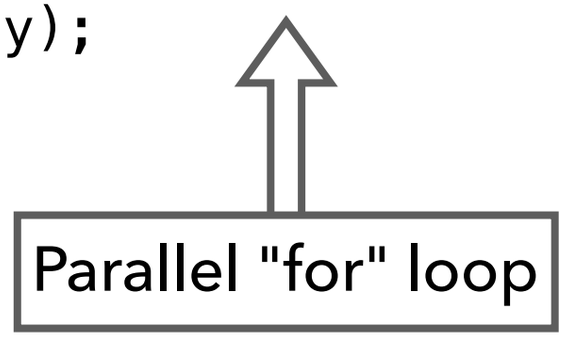
The "for" loop

How Halide works?

```
void sum_2_vectors(float const * xA,
                   float const * xB,
                   float * y,
                   int const xLen) {
    Halide::Buffer<bFlt32> lA { const_cast<float *>(xA), xLen, "xA" };
    Halide::Buffer<bFlt32> lB { const_cast<float *>(xB), xLen, "xB" };
    Halide::Var i { "i" };
    Halide::Func func;

    func(i) = lA(i) + lB(i);

    Halide::Buffer<bFlt32> lOut = func.parallel(i).realize(xLen);
    std::copy_n(lOut.data(), xLen, y);
}
```

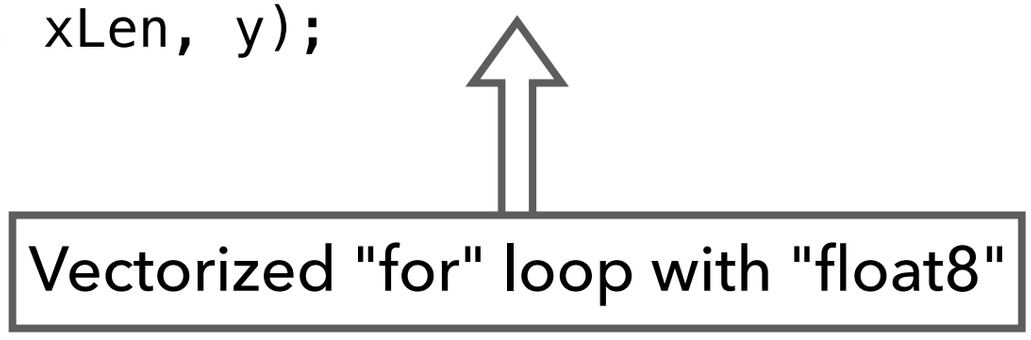


How Halide works?

```
void sum_2_vectors(float const * xA,
                   float const * xB,
                   float * y,
                   int const xLen) {
    Halide::Buffer<bFlt32> lA { const_cast<float *>(xA), xLen, "xA" };
    Halide::Buffer<bFlt32> lB { const_cast<float *>(xB), xLen, "xB" };
    Halide::Var i { "i" };
    Halide::Func func;

    func(i) = lA(i) + lB(i);

    Halide::Buffer<bFlt32> lOut = func.vectorize(i, 8).realize(xLen);
    std::copy_n(lOut.data(), xLen, y);
}
```



How Halide works?

```
void sum_2_vectors(float const * xA,
                   float const * xB,
                   float * y,
                   int const xLen) {
    Halide::Buffer<bFlt32> lA { const_cast<float *>(xA), xLen, "xA" };
    Halide::Buffer<bFlt32> lB { const_cast<float *>(xB), xLen, "xB" };
    Halide::Var i { "i" }, j { "j" }, k { "k" };
    Halide::Func func;

    func(i) = lA(i) + lB(i);

    func.vectorize(i, j, k, 8);
    Halide::Buffer<bFlt32> lOut = func.parallel(j).unroll(k).realize(xLen);
    std::copy_n(lOut.data(), xLen, y);
}
```

Transforming a 1 dimensional "for"-loop into 2D loop

Unroll the inner loop!

Blur Filter: C++

...the baseline for comparison!

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate blurx array

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```

Slow

Fast

Blur Filter: Halide

Halide

0.9 ms/megapixel

```
Func box_filter_3x3(Func in) {
  Func blurx, blury;
  Var x, y, xi, yi;

  // The algorithm - no storage, order
  blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
  blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;

  // The schedule - defines order, locality; implies storage
  blury.tile(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
  blurx.compute_at(blury, x).store_at(blury, x).vectorize(x, 8);

  return blury;
}
```

Sugar: tiling!

C++

With platform-specific SIMD!

0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
          a = _mm_loadu_si128((__m128i*)(inPtr-1));
          b = _mm_loadu_si128((__m128i*)(inPtr+1));
          c = _mm_load_si128((__m128i*)(inPtr));
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(blurxPtr++, avg);
          inPtr += 8;
        }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Laplacian Filter

...real life example!

	Reference C++
LOC	300
Time	?
Performance	1x

Laplacian Filter

...real life example!

	Reference C++	Adobe CPU
LOC	300	1,500
Time	?	3 months
Performance	1x	10x

Laplacian Filter

...real life example!

	Reference C++	Adobe CPU	Halide CPU
LOC	300	1,500	**60**
Time	?	3 months	**1 day**
Performance	1x	10x	**20x**

Laplacian Filter

...real life example!

	Reference C++	Adobe CPU	Halide CPU	Halide GPU
LOC	300	1,500	**60**	
Time	?	3 months	**1 day**	
Performance	1x	10x	**20x**	**70x**

Vector Sum: SyCL Today

```
void sum_2_vectors(float const * xA,
                   float const * xB,
                   float * y,
                   int const xLen) {

    cl::sycl::queue q;
    cl::sycl::buffer<float, 1> lA { xA, xLen };
    cl::sycl::buffer<float, 1> lB { xB, xLen };
    cl::sycl::buffer<float, 1> lOut { y, xLen };

    q.submit([&](cl::sycl::handler & h) {
        auto hA = lA.get_access<cl::sycl::access::mode::read>(h);
        auto hB = lB.get_access<cl::sycl::access::mode::read>(h);
        auto hOut = lOut.get_access<cl::sycl::access::mode::write>(h);
        h.parallel_for<class kernel_name>(xLen, [=] (cl::sycl::id<1> i) {
            hOut[i] = hA[i] + hB[i];
        });
    });
    q.wait();
}
```

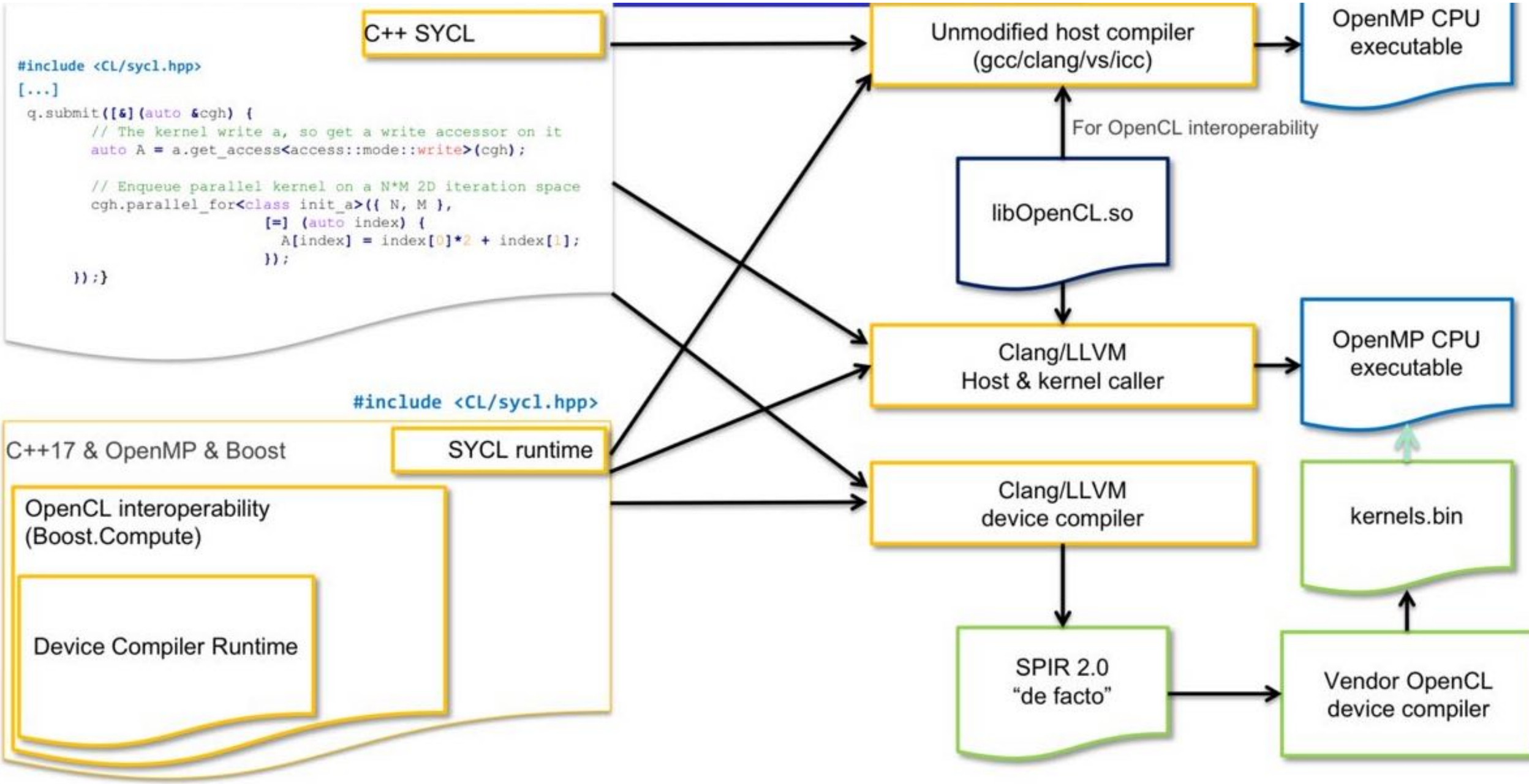
A lonely meaningful line, surrounded by boilerplate!

Vector Sum: SyCL STL

```
void sum_2_vectors(float const * xA,
                   float const * xB,
                   float * y,
                   int const xLen) {

    std::transform(cl::sycl::uniform_policy,
                   std::span(xA, xLen), std::span(xB, xLen),
                   std::span(y, xLen),
                   std::plus<float> { });
}
```

How SyCL works?



How SyCL works?

...here is what you actually need to know

It's not a language!

It's not a library!

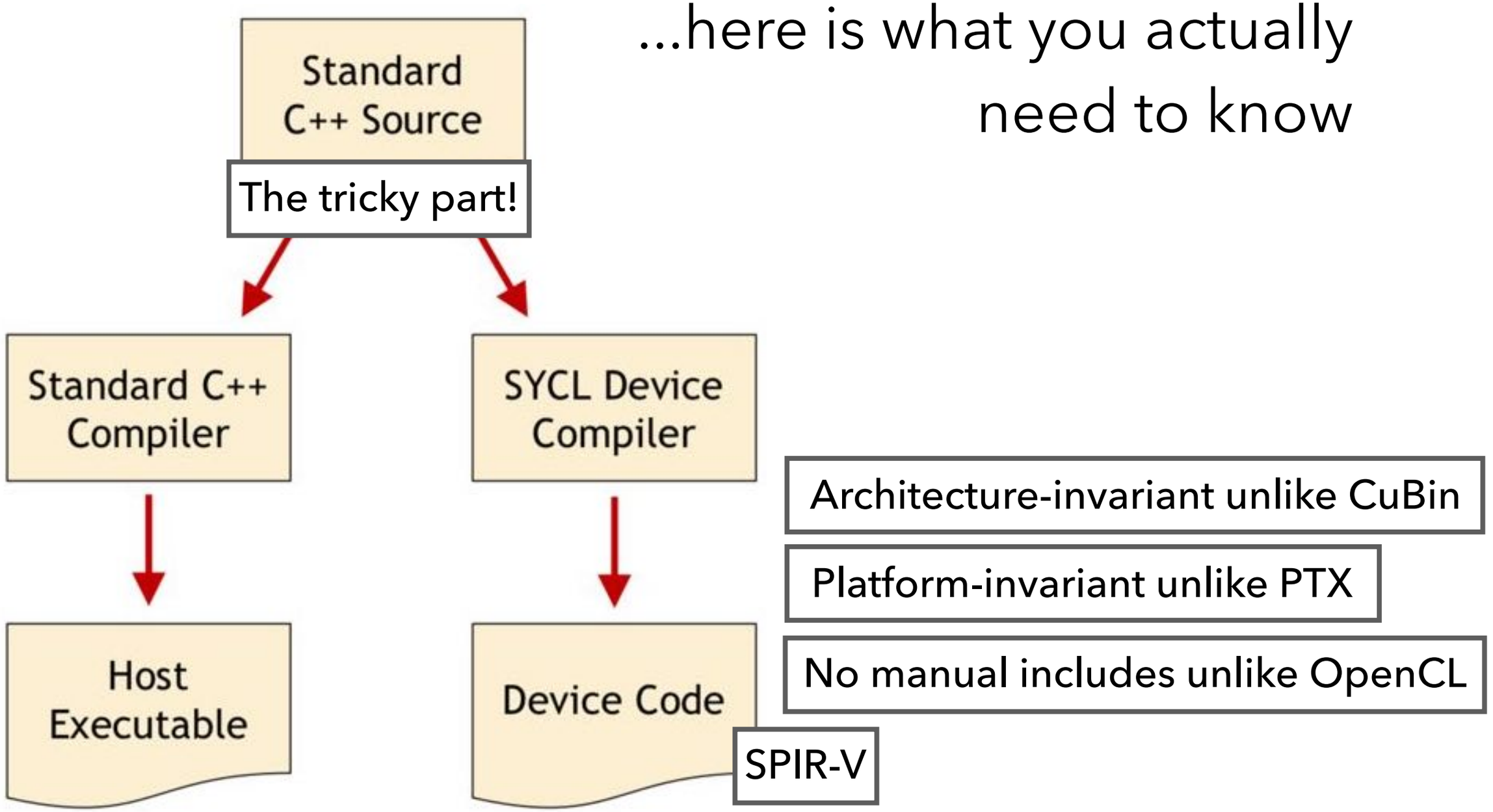
It's not a compiler!

It's a combination of:

- library,
- compiler extensions!

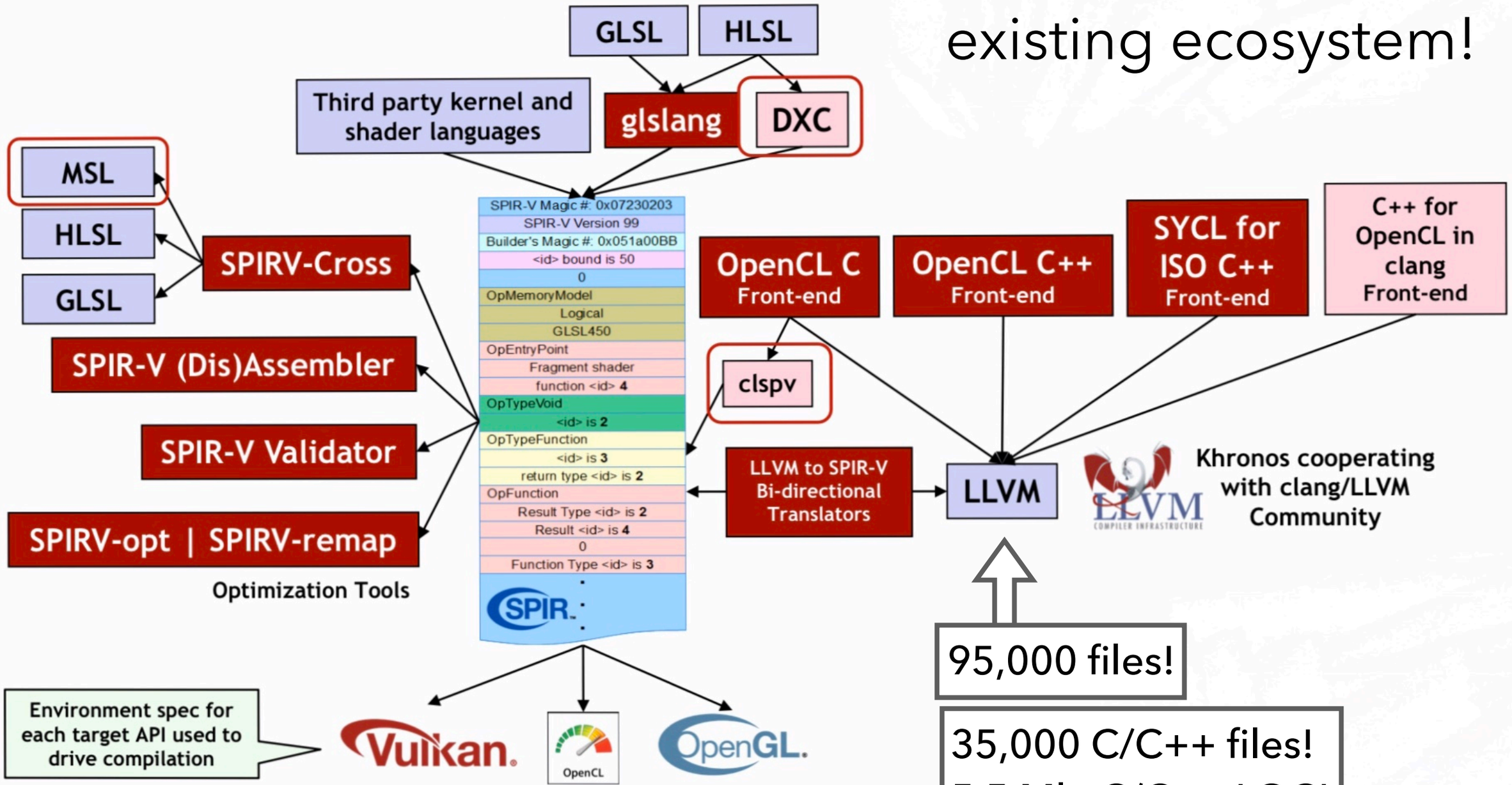
How SyCL works?

...here is what you actually need to know



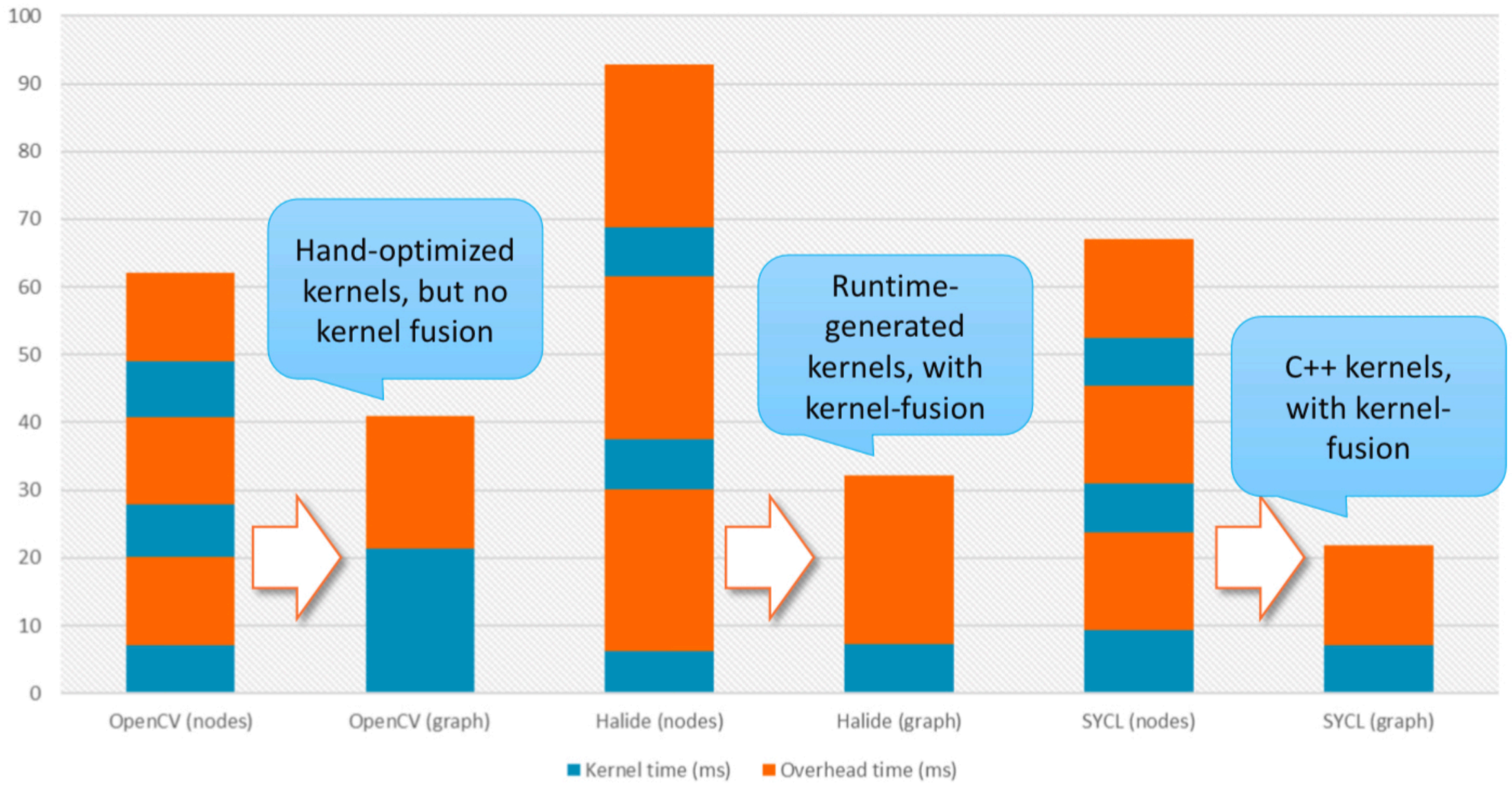
How SyCL works?

...it grows on top of the existing ecosystem!



Kernel Fusion

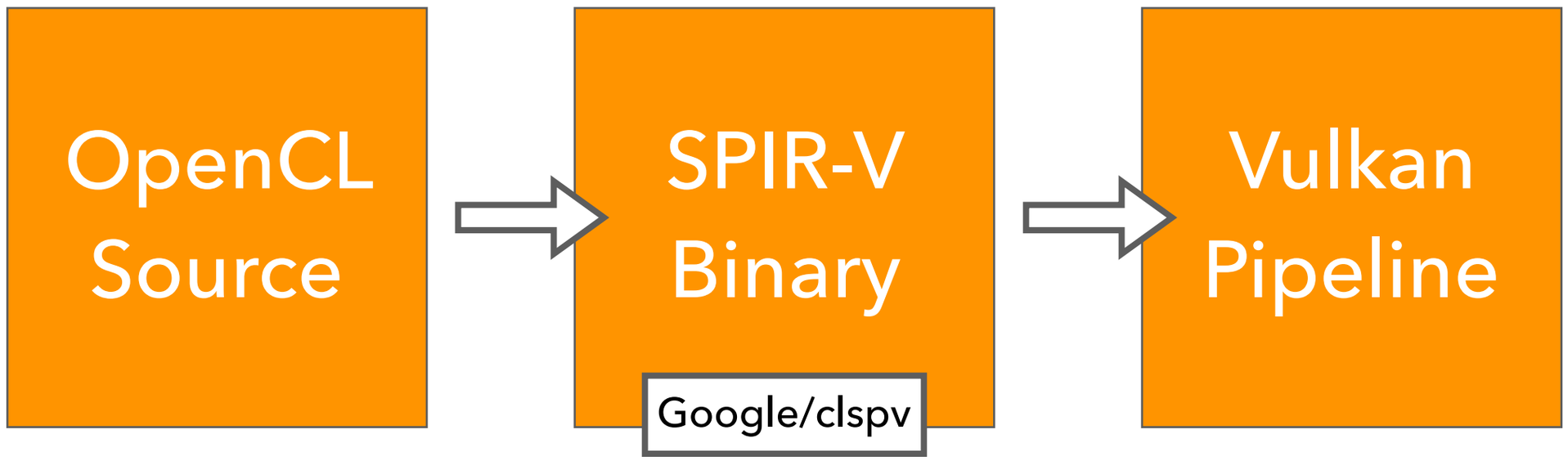
...impact on pipeline performance!

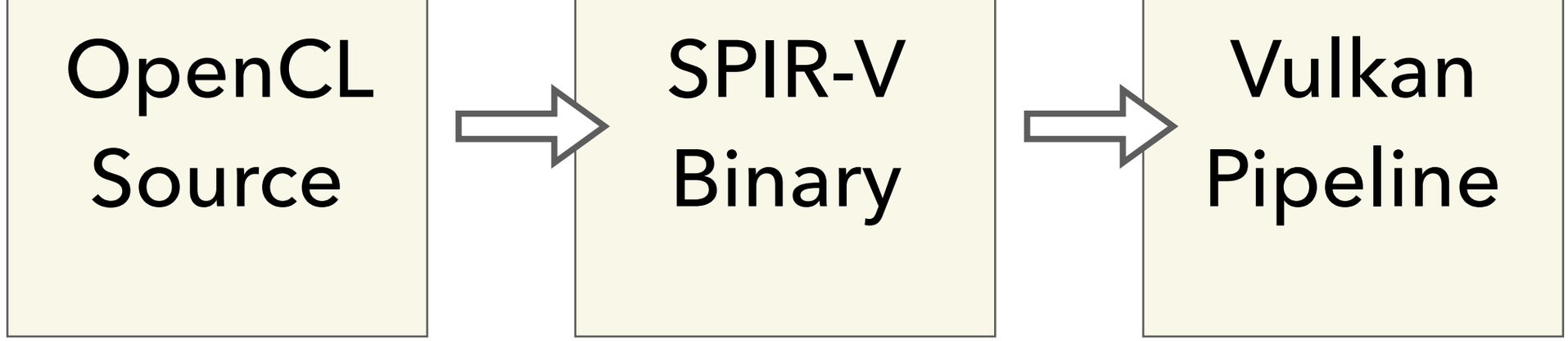


Recipes

Unified solution

...if you want cross platform binaries for your custom hand-made kernels!





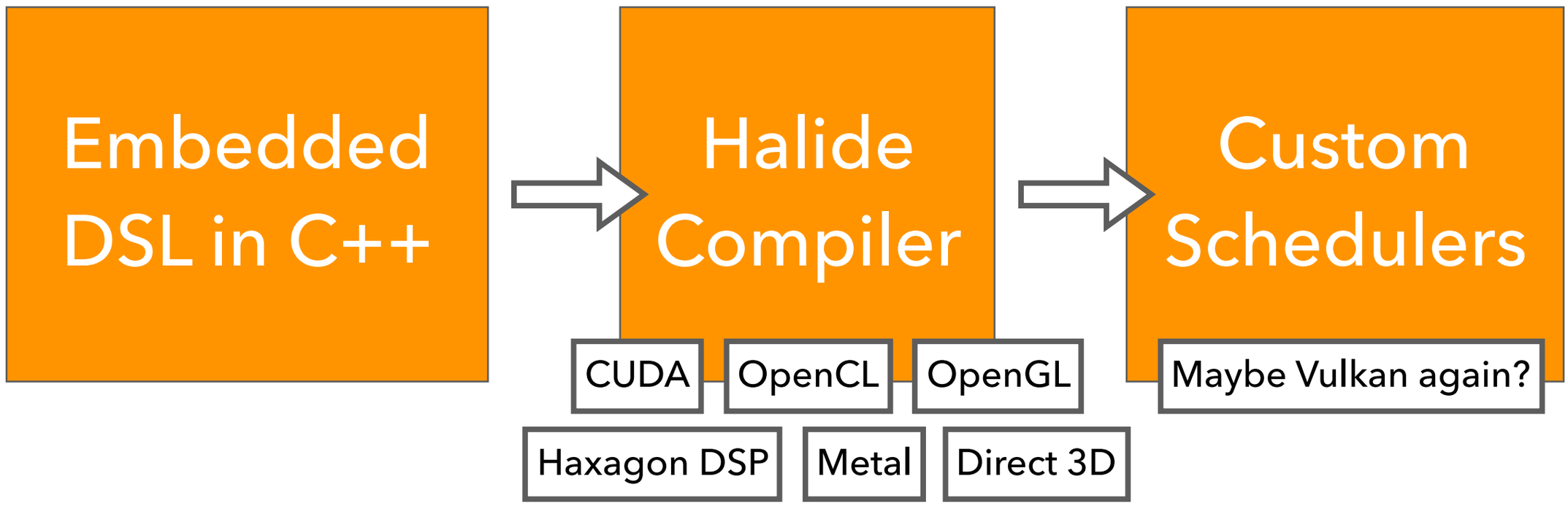
Pros	Cons
Same binary runs everywhere, easy to debug	**Separate CL/C++ codebases**
Concurrent queues	**Experimental stage compilers**
Logical devices can represent SLI groups	No support for CUDA features: warp shuffles, hardware-specific instructions *, L1 cache tuning
Same ecosystem for both graphics and compute	No modern C++ support until version 2.2

No work-arounds?

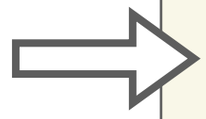
100 TFlops?!

Flexible solution

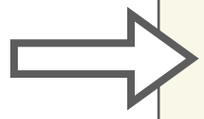
...if you want a flexible tool
here and now!



Embedded DSL in C++



Halide Compiler



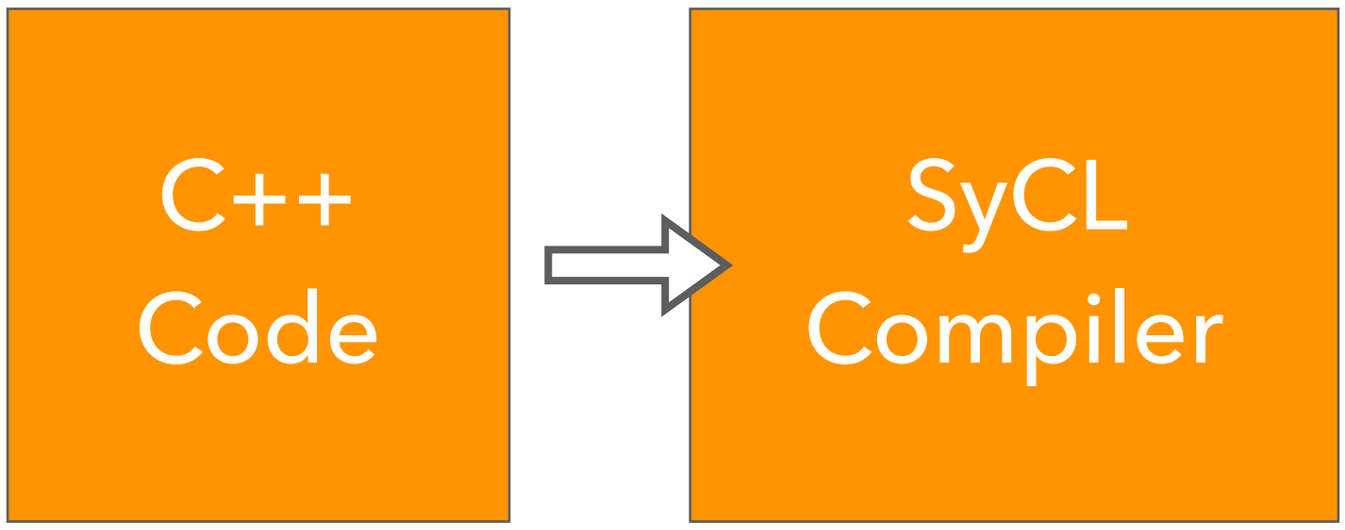
Custom Scheduler

Pros
Great for prototyping and benchmarking
Generate binaries for every platform
Easy to export computational graphs into other libs
Easy to debug algorithms
Built-in tools for image procssing

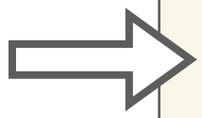
Cons
Turing-incomplete
Limited number of supported types
Huge LLVM dependency
Non-standard C++
Bad error messages

Clean solution

...if you want some classical
type-safe C++!



C++ Code



SyCL Compiler

Pros	Cons
Use C++ templates and lambda functions for host & device code - just pass "sycl" policy	**Very immature, stability and C++17 adoption is expected closer to 2020-2021**
SYCL will not create C++ language extensions, but instead add features via C++ library	**Underlying implementation requires compiler support**
Does kernel fusion	Kernel fusion may be weak
Layered over OpenCL	Boost.Compute dependency

Simple solution

...if you want a brute-force accelerator for simple data-parallel number-crunching!

High level GPGPU library of choice like ArrayFire

High level GPGPU library of choice like ArrayFire

Pros	Cons
Already packed with binaries for multiple backends	Weak kernel fusion
Minimal coding required	

Comparison of recipes

...lets summarize our results!

	Simple	Unified	Flexible	Clean
Technology	ArrayFire	CL & SPIR-V	Halide	C++ SyCL
Write code once	Yes	Separate Files	T-Incomplete	Yes
Deploy everywhere	Almost	Yes	Yes	Eventually
Optimise performance	Average	High	Highest	High
Avoid boilerplate	YES!	Depends	Depends	Yes

Comparison of recipes

...lets summarize our results!

	Max Performance Today	
Technology	CL & SPIR-V	Halide
Write code once	Yes	T-Incomplete
Deploy everywhere	Yes	Yes
Optimise performance	High	Highest
Avoid boilerplate	Depends	Depends

Comparison of recipes

...lets summarize our results!

	Sometime in the Future
Technology	C++ SyCL
Write code once	Yes
Deploy everywhere	Eventually
Optimise performance	High
Avoid boilerplate	Yes

Tips & Tricks

OpenCL Tips

1. Unroll loops & inline functions manually!
2. Violate the IEEE 754 standard:
   - Accuracy: `-cl-unsafe-math-optimizations, -cl-mad-enable`
   - Zeroes: `-cl-finite-math-only, -cl-no-signed-zeros`
3. **Use special functions:**
   - Work with fractions: `remainder(), remqo(), fract()`
   - Perform multiple actions: `fma(), sincos(), expm1(), mad()`
   - OpenCL knows Pi: `tanpi(), sinpi(), atanpi(), atan2pi()`
4. Don't create too many command buffers!

CUDA Tips

1. Similarly, use fast math functions:
   - Half precision: `__hadd()`, `__hadd_sat()`, `__hmul()`, `__hneg()`
   - Control rounding: `__fadd_rd()`, `__fadd_rz()`, `__fadd_rn()`
   - Perform multiple actions: `__fmaf_rd()`, `__sincosf()`
2. **Use the ecosystem!**
   - **Mixed Precision Math libraries**.
   - cuBLAS & cuDNN.
3. Transform CUDA stream code into multi-GPU reusable dependency graph.

Existing Ecosystem

Symbolic Graph	**TF, PyTorch**, **cuDNN**, MKL-DNN
Lazy Evaluation	Eigen, **TF**, VexCL, ArrayFire
Linear Algebra	Eigen, MKL, VexCL, **cuBLAS**, **ArrayFire**, Boost.Compute
Scheduling	Intel TBB, Vulkan, **OpenMP**, SyCL
Language & Extensions*	**CUDA**, **OpenCL**, GLSL, OpenMP, OpenACC
Compilers*	**LLVM**, **TVM**, GCC

Where to apply?

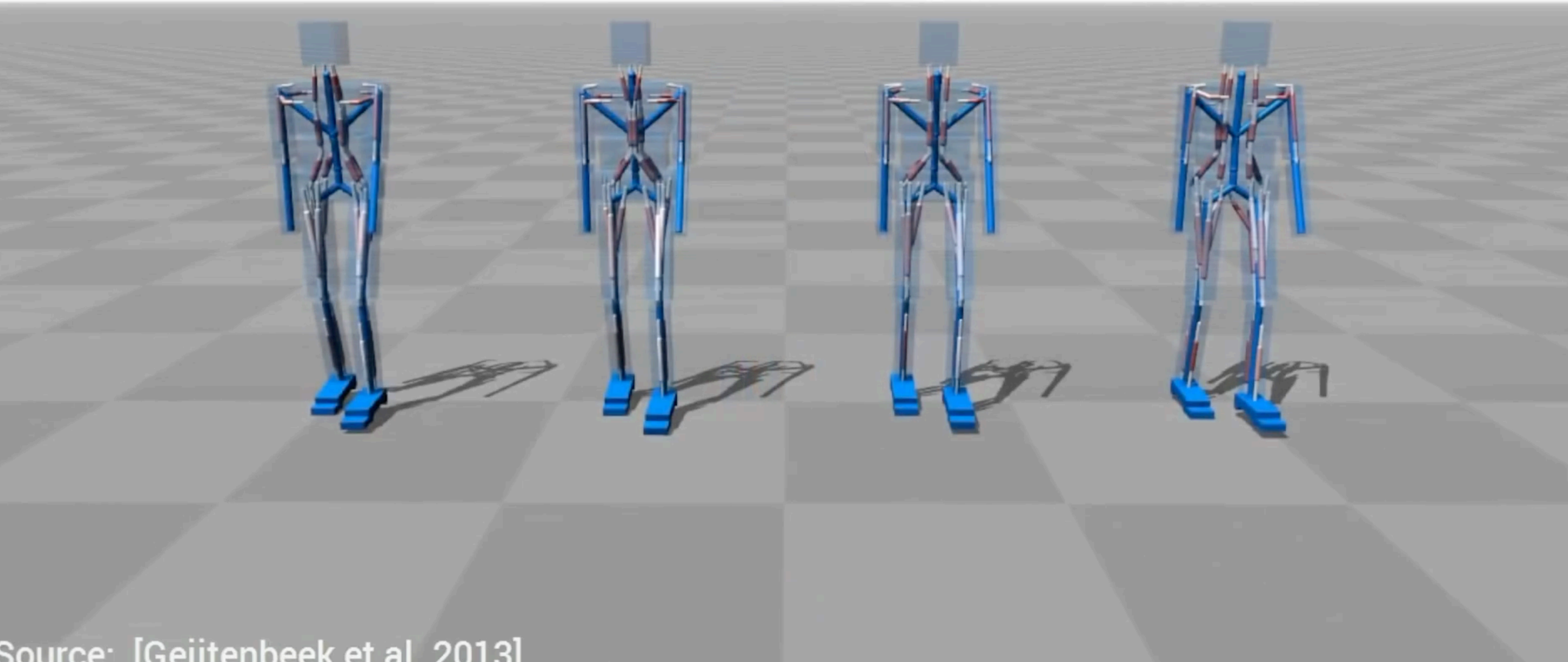
Unum

Scalable & cheap analytics with modern ML techniques.

Built on top of:

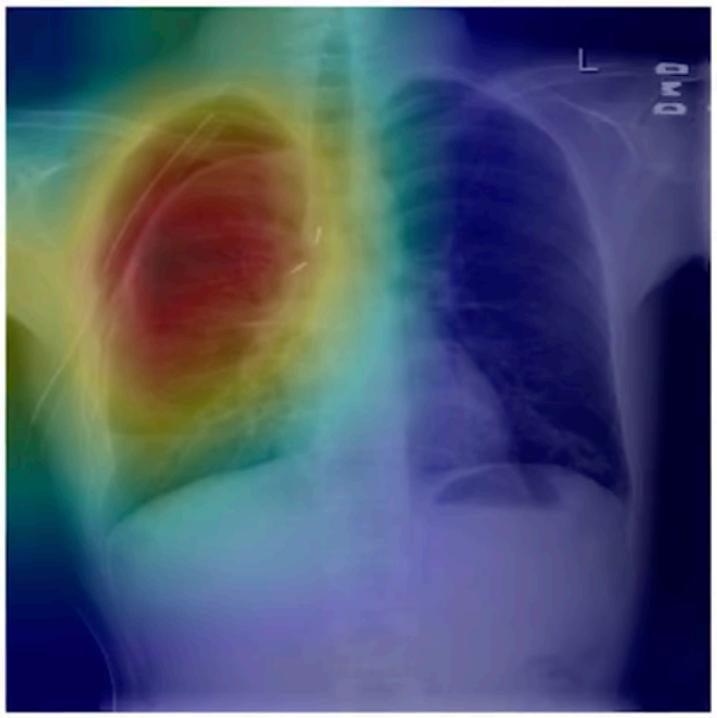
- C++14 since 2015,
- C++17 since 2017,
- OpenCL, LLVM,
- Eigen, ArrayFire.

Why AI? Robotics

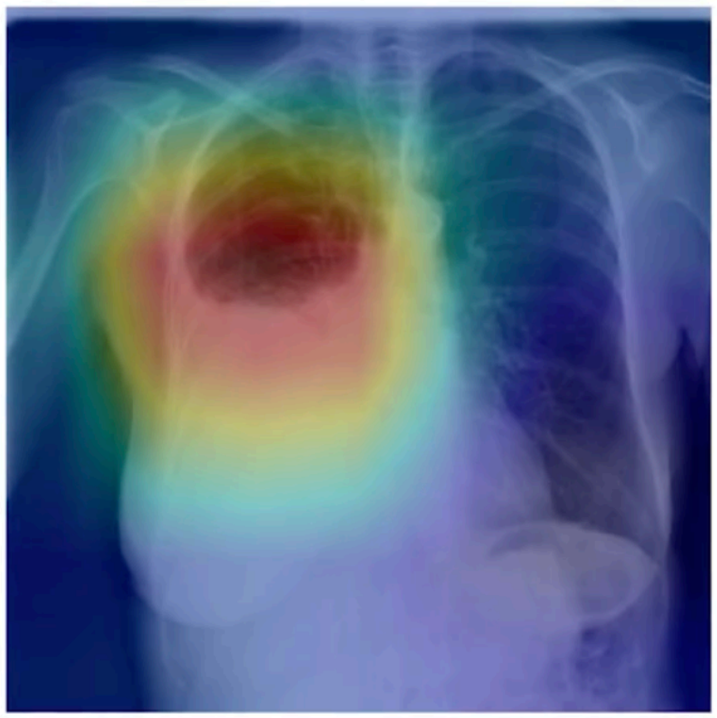


Source: [Geijtenbeek et al. 2013]

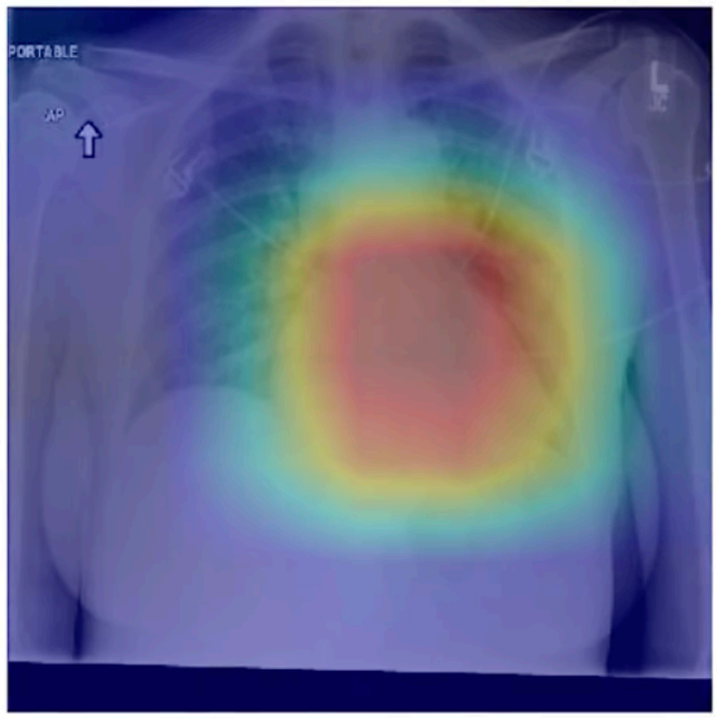
Why AI? Cancer



(d) Patient with a right-sided pneumothroax and chest tube. The model detects the abnormal lung to correctly predict the presence of pneumothorax (collapsed lung).



(e) Patient with a large right pleural effusion (fluid in the pleural space). The model correctly labels the effusion and focuses on the right lower chest.



(f) Patient with congestive heart failure and cardiomegaly (enlarged heart). The model correctly identifies the enlarged cardiac silhouette.

Figure 3. CheXNet localizes pathologies it identifies using Class Activation Maps, which highlight the areas of the X-ray that are most important for making a particular pathology classification. The captions for each image are provided by one of the practicing radiologists.

Why AI? Knowledge

GPT-2 BERT TRANSFORMER XL

Sentence:

The best programming language is

Predictions:

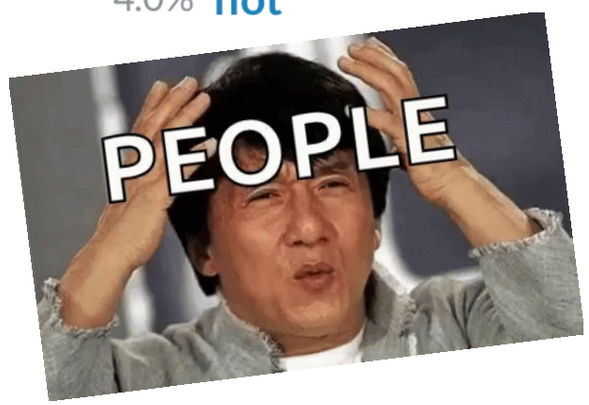
5.7% **Python**

5.7% **Java**

5.3% **the**

4.0% **a**

4.0% **not**



https://demo.allennlp.org/next-token-lm

Ecosystem Problems; C*

Symbolic Graph	**TF, PyTorch**, Too big to navigate & embed!
Lazy Evaluation	Eigen, **TF**, VexCL, Limited serialisation beyond ONNX!
Linear Algebra	Eigen, MKL, VexCL, **cuBLAS**, **ArrayFire**, Boost.Compute
Scheduling	Intel TBB, Vulkan, **OpenMP**, SyCL Verbose!
Language & Extensions*	**CUDA**, **OpenCL**, Same, but fragmented!
Compilers*	**LLVM**, Single point of failure!

Goals for C*

Performance higher than in C++.

Kernel fusion > Inlining

As simple as Common LISP.

Interpretable into C, OpenCL, CUDA

Flexible like Halide.

Tuning is separate from logic

Q & A

The talk:

- github.com/ashvardanian/SandboxGPUs

The project:

- unum.xyz
- github.com/unumxyz

Me:

- a@unum.xyz

- linkedin.com/in/ ashvardanian
- github.com/ ashvardanian
- fb.com/ ashvardanian
- vk.com/ ashvardanian