Efficient GPGPU programming

Ashot Vardanian

Who am I?

Ashot Vardanian, 24
In software >10 years

Working on:

- High Performance Computing
- AI Research

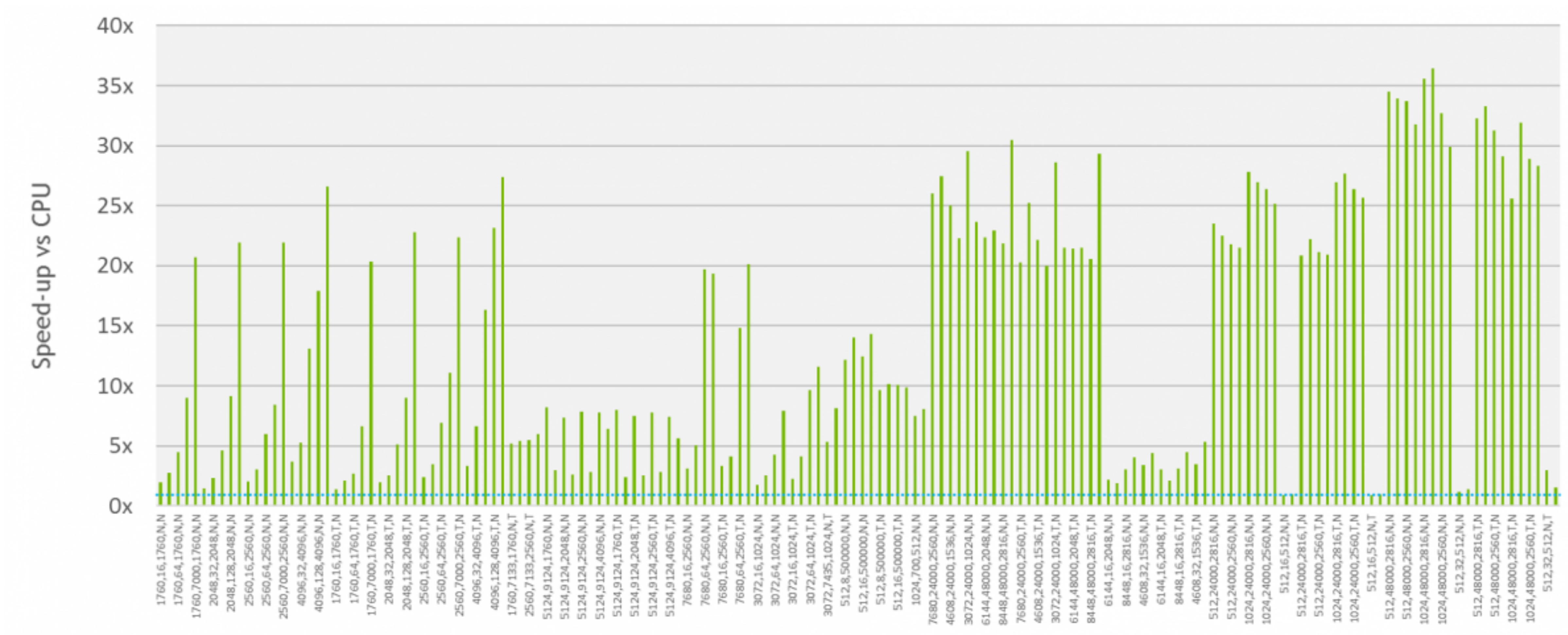
Worked on:

- Web
- Mobile
- Desktop
- Scientific Computing

github.com/ashvardanian
fb.com/ashvardanian

Why GPUs?

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



What we want?

Write code once
Run everywhere
Max performance
Minimal code size

What we want?

Write code once	Unified language
Run everywhere	Modular comilers
Max performance	Tools for tuning
Minimal code size	Clean libs & tools

What we want?

Write code once	Unified language
Run everywhere	Modular comilers
Max performance	Tools for tuning
Minimal code size	Clean libs & tools

Comparison of recipes

...we will fill this table:

	Simple	Unified	Flexible	Clean
Technology	?	?	?	?
Write code once	?	?	?	?
Run everywhere	?	?	?	?
Max performance	?	?	?	?
Minimal code size	?	?	?	?

Plan

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

Plan

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

Plan

1. Existing Standards
2. Writing Low-level code
3. **Existing Libraries & Tools:**
    1. **Linear Algebra,**
    2. **Lazy Evaluation,**
    3. **Symbolic Graphs**.
4. Optimal Recipes

Plan

1. Existing Standards
2. Writing Low-level code
3. Existing Libraries & Tools
4. **Optimal Recipes**
    1. **SyCL,**
    2. **Halide,**
    3. **Custom.**

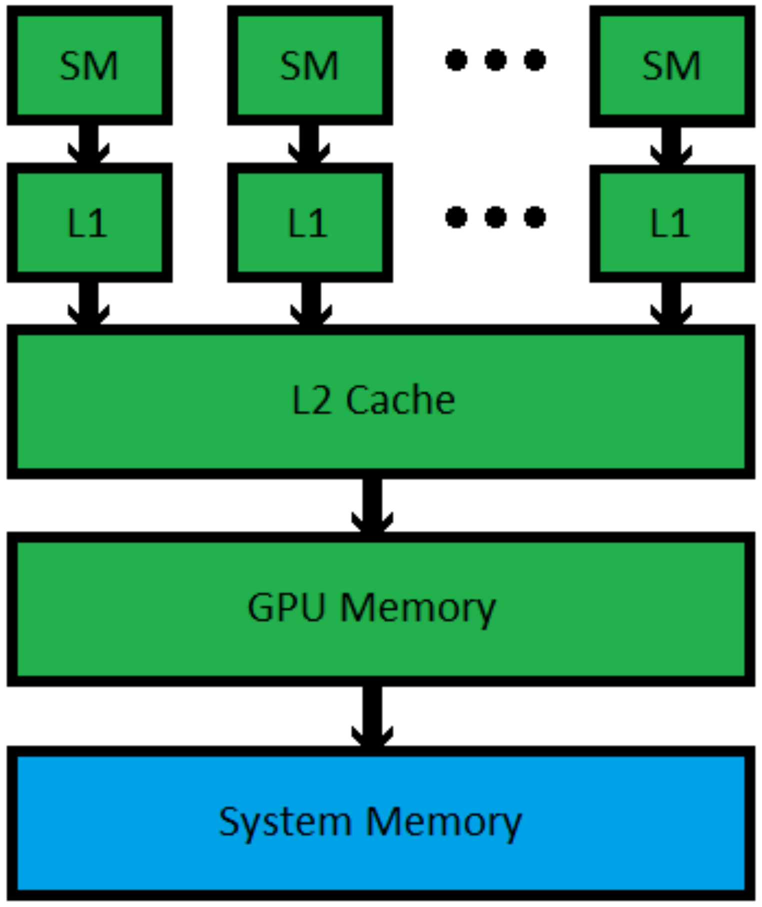
Whats a GPU?

A typical GPU

	ARM (Mali), Qualcomm (Adreno)	Intel (Integrated), AMD (Integrated)	NVidia (Discrete), AMD (Discrete)
Threads*	400	500	4000
Frequency	600 MHz	900 MHz	1,4 GHz
Memory	6* Gb	1 Gb	12 Gb
Energy Consumption	<5 W	<15 W	250 W

Memory Pools

...on GPU are physically similar
to CPU memory!



	CPU	GPU
L1 / Core	1 MB	32 Kb
L2	20 Mb	2 Mb
RAM	32 Gb	8 Gb
Bandwidth	40 Gb/s	600 Gb/s

Existing APIs

For CPU-GPU communication

API Support

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

API Support

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

API Support

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

API 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	💻	No	💻	**Yes**
Android	**Yes**	No	Depends	No

API 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	💻	No	💻	**Yes**	MoltenVK
Android	**Yes**	No	Depends	No	**Yes**

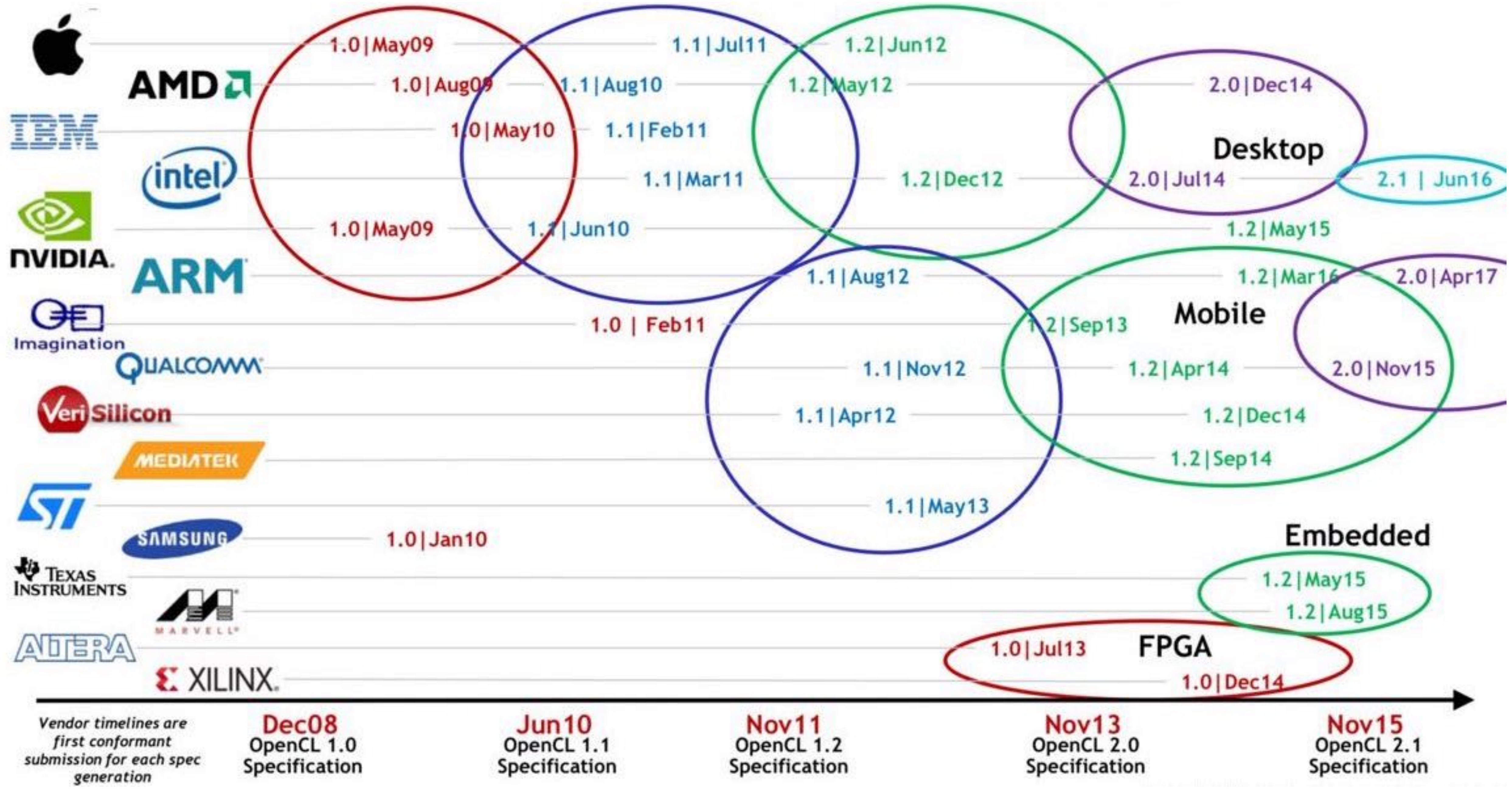
API Comparison

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

API Comparison

		CUDA	OpenCL		Vulkan
Primary Purpose		**Compute**	**Compute**		Graphics
Base Input Language		**C++**	C		**Any**
Complexity		**Easy**	**Easy**		Hard ...in every way
Targets Flexibility		Low ...only Nvidia	**Extreme ...FPGA**		**High**
API Flexibility*		**High****	Average		**High**

Nvidia vs Everybody



Language Syntax

CUDA vs OpenCL

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

	CUDA	OpenCL
All Threads	Global	Global
Group of Threads	Shared	Local
Single Thread	Local, Register (faster)	Private
Other	Constant, Texture	Constant

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 $\boxed{\mathbb{N}^1}$	<1024 threads $\boxed{\mathbb{N}^3}$	< 256 threads $\boxed{\mathbb{N}^3}$
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 $\mathbb{N}^1$	<1024 threads $\mathbb{N}^3$	< 256 threads $\mathbb{N}^3$
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
Local Memory	32 Kb		32 Kb

1 Mb L2

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

Terminology

CUDA	OpenCL
Stream Multiprocessor	Compute Unit
Thread	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()
blockIdx * blockDim + threadIdx	get_global_id()
gridDim * blockDim	get_global_size()

Kernels Synchronization

	CUDA	OpenCL	
group	__syncthreads()	barrier()	
all	__threadfence()	~	
group mem	__threadfence_block()	mem_fence()	
	~	read_mem_fence()	**?**
	~	write_mem_fence()	**?**

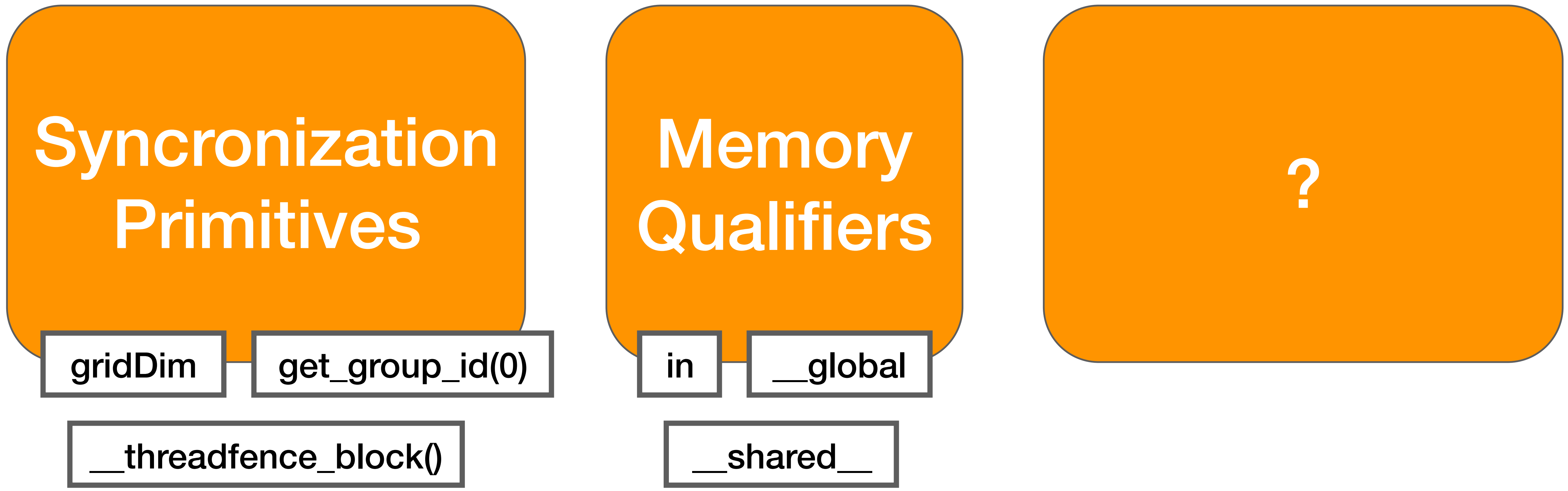
Host API

...even here everything is identical!

CUDA	OpenCL
cudaGetDeviceProperties()	clGetDeviceInfo()
cudaMalloc()	clCreateBuffer()
cudaMemcpy()	clEnqueueRead(Write)Buffer()
cudaFree()	clReleaseMemObj()
kernel<<<...>>>()	clEnqueueNDRangeKernel()

Parallelism in Language

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

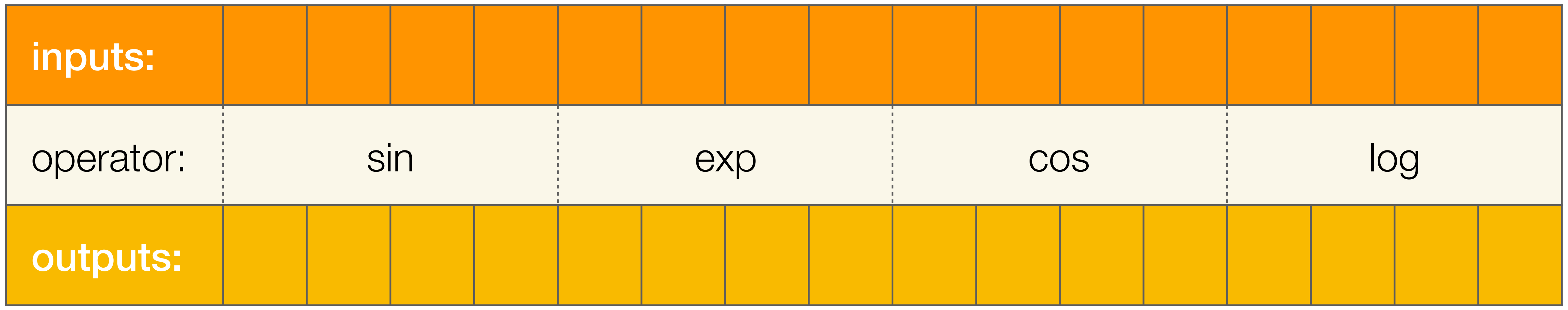


Code Examples

Why would you want to write low-level kernels?

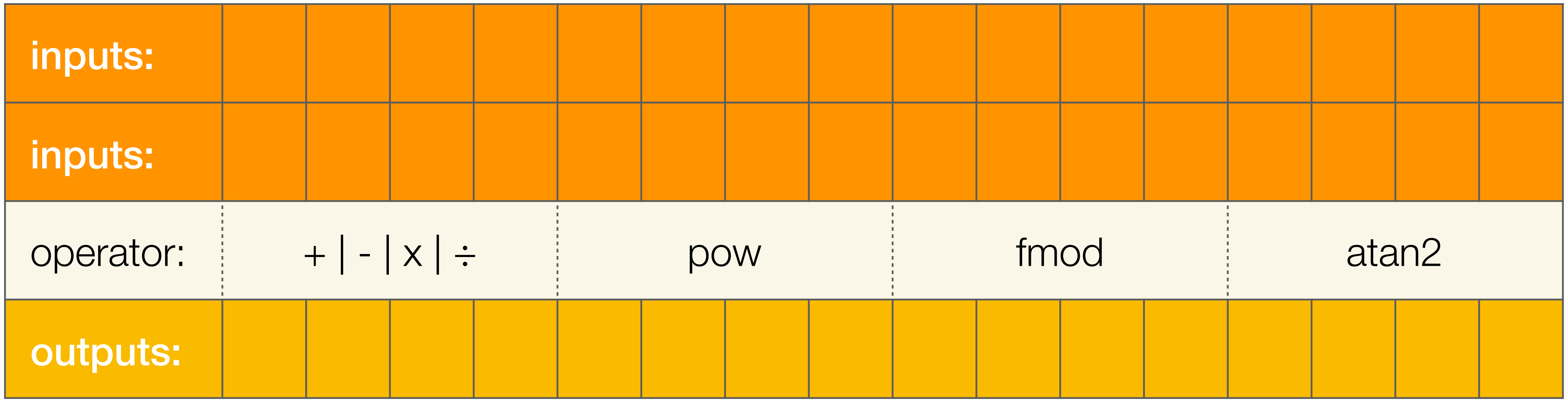
Data-Parallel Tasks

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



Data-Parallel Tasks

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



Vector Sum: C

```c
void gArithmAddArr(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

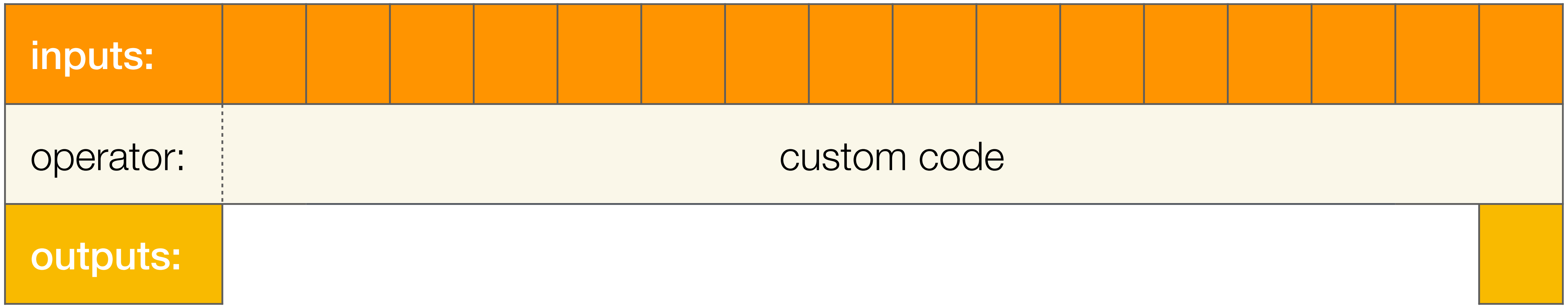
```
kernel void gArithmAddArr(global float const * xA,
                          global float const * xB,
                          global float * y) {
    int i = get_global_id(0);
    y[i] = xA[i] + xB[i];
}
```

Vector Sum: GLSL

```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!

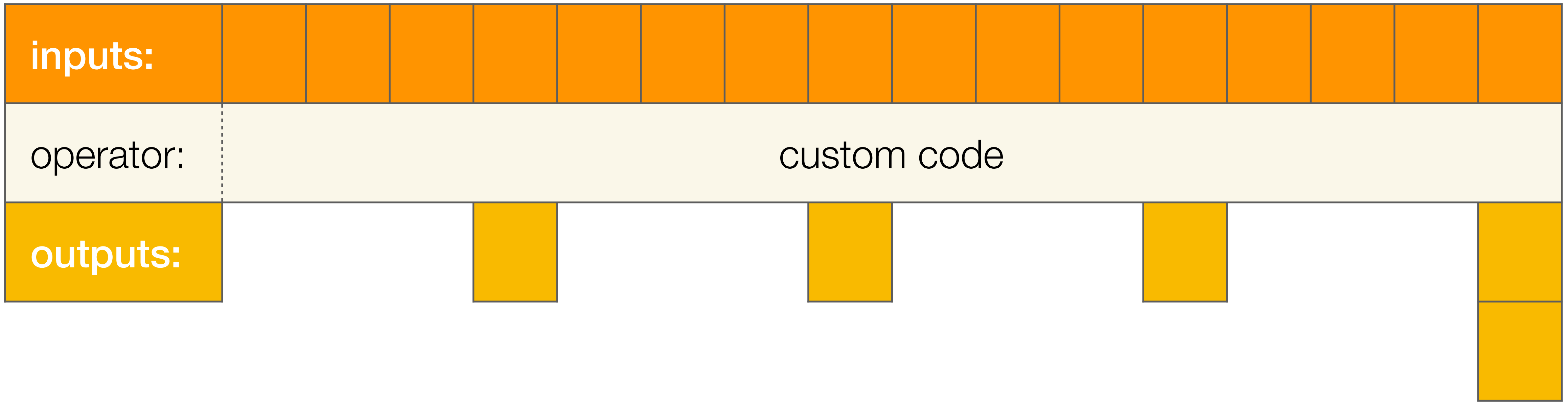


Reduction: C

```
void gReduce(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 gReduceSimple(__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 gReduceUnrolled(__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];
}
```

Data-Parallel Tasks

Reductions in OpenCL

Matrix Multiplications

- Eigen on CPU
- Nvidia mixed precision for Tensor Cores
-

Existing Libs & Tools

The complexity of Choice

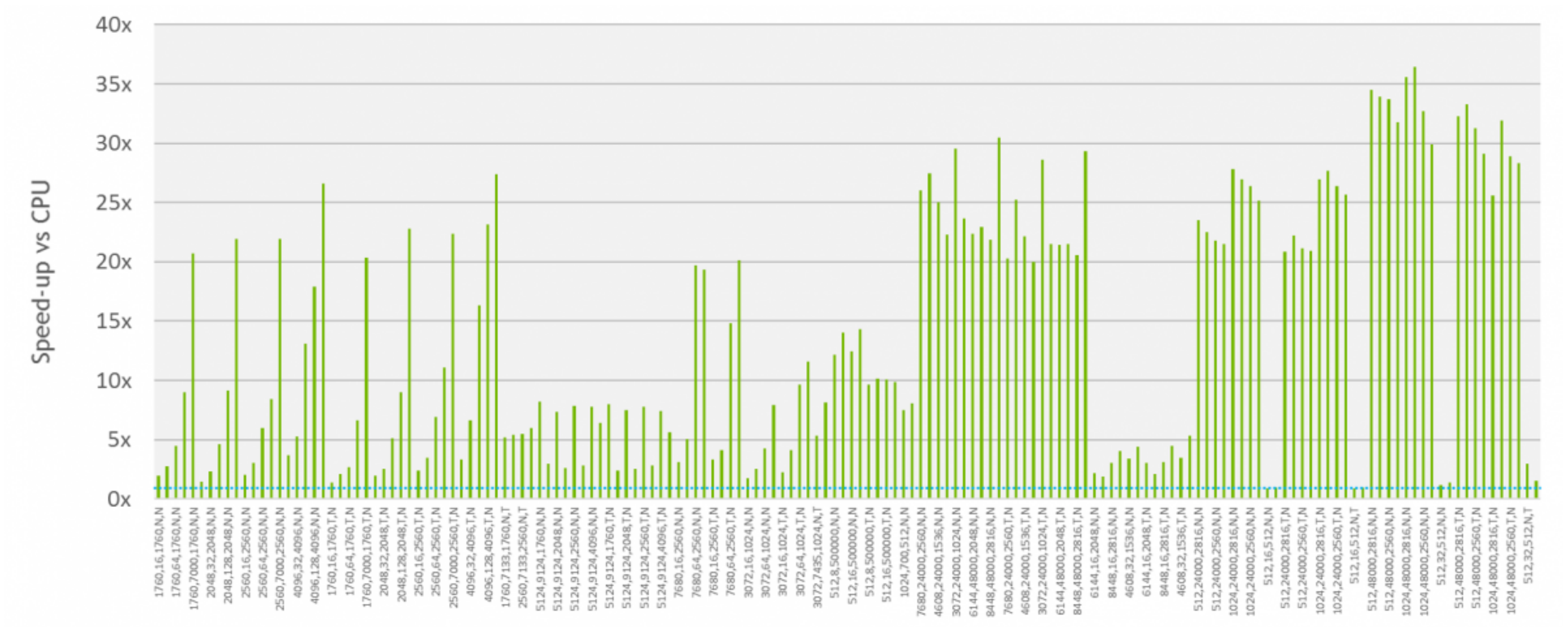
Linear Algebra

	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!



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

Lazy Evaluation

Lazy	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.

Symbolic Graphs

	TensorFlow	PyTorch	MxNet
LOC Code	2,251,532	710,449	406,488
LOC Comments	555,516	100,223	119,447
LOC C++	53%	56%	35%
CUDA/OpenCL	0%	13%	4%

Almost no internal optimizations for GPUs.
Building symbolic computations graphs is very inefficient for small jobs.
Huge number of complex dependencies!

Data-Parallel Tasks

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

inputs:																
operator:	sin				exp				cos				log			
outputs:																

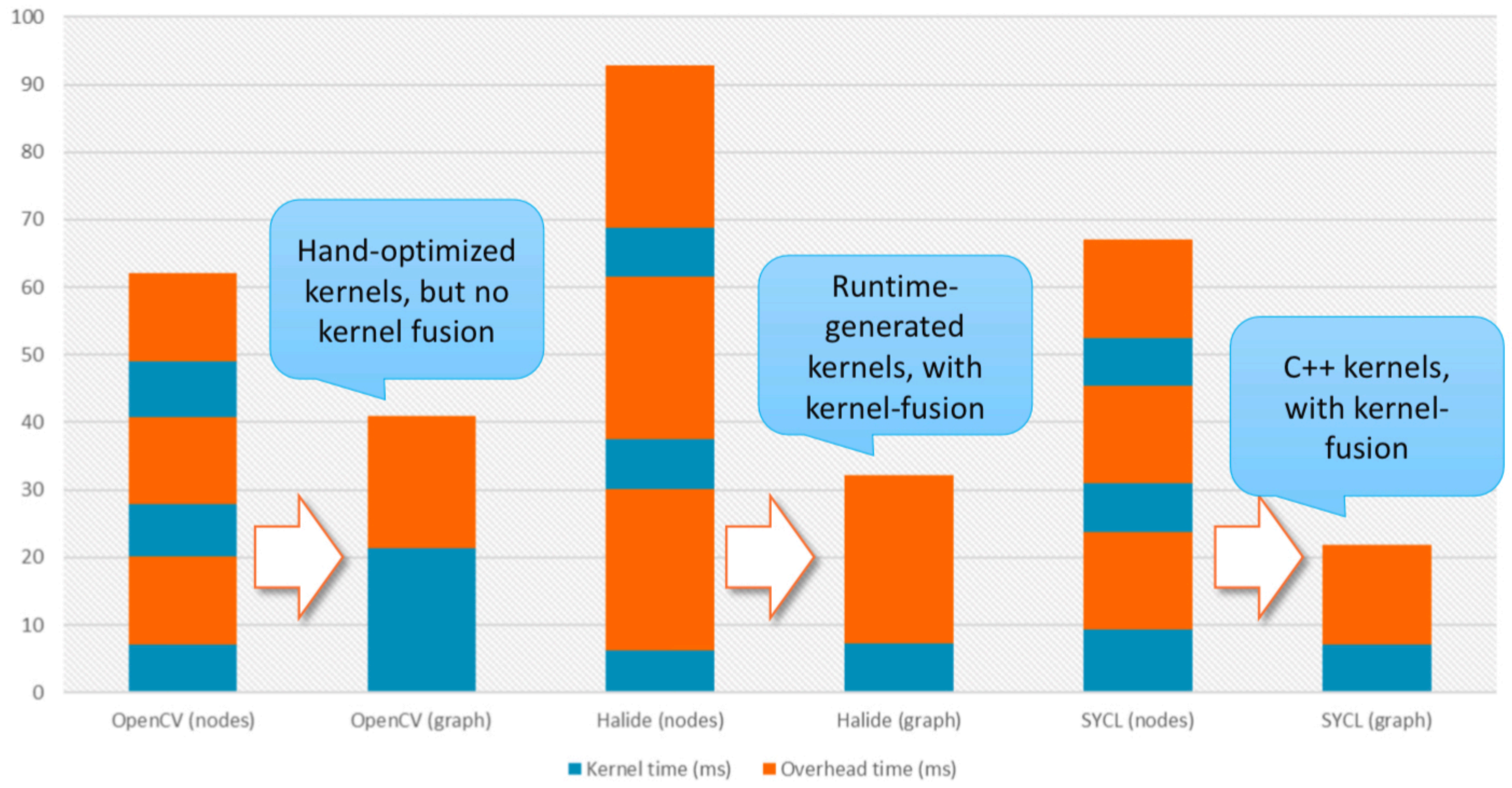
Cost of Memory Access

...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 slower**
Send over LTE	10 μJ	**10,000,000x slower**

Kernel Fusion

...the key to high-performance <u>pipelines!</u>



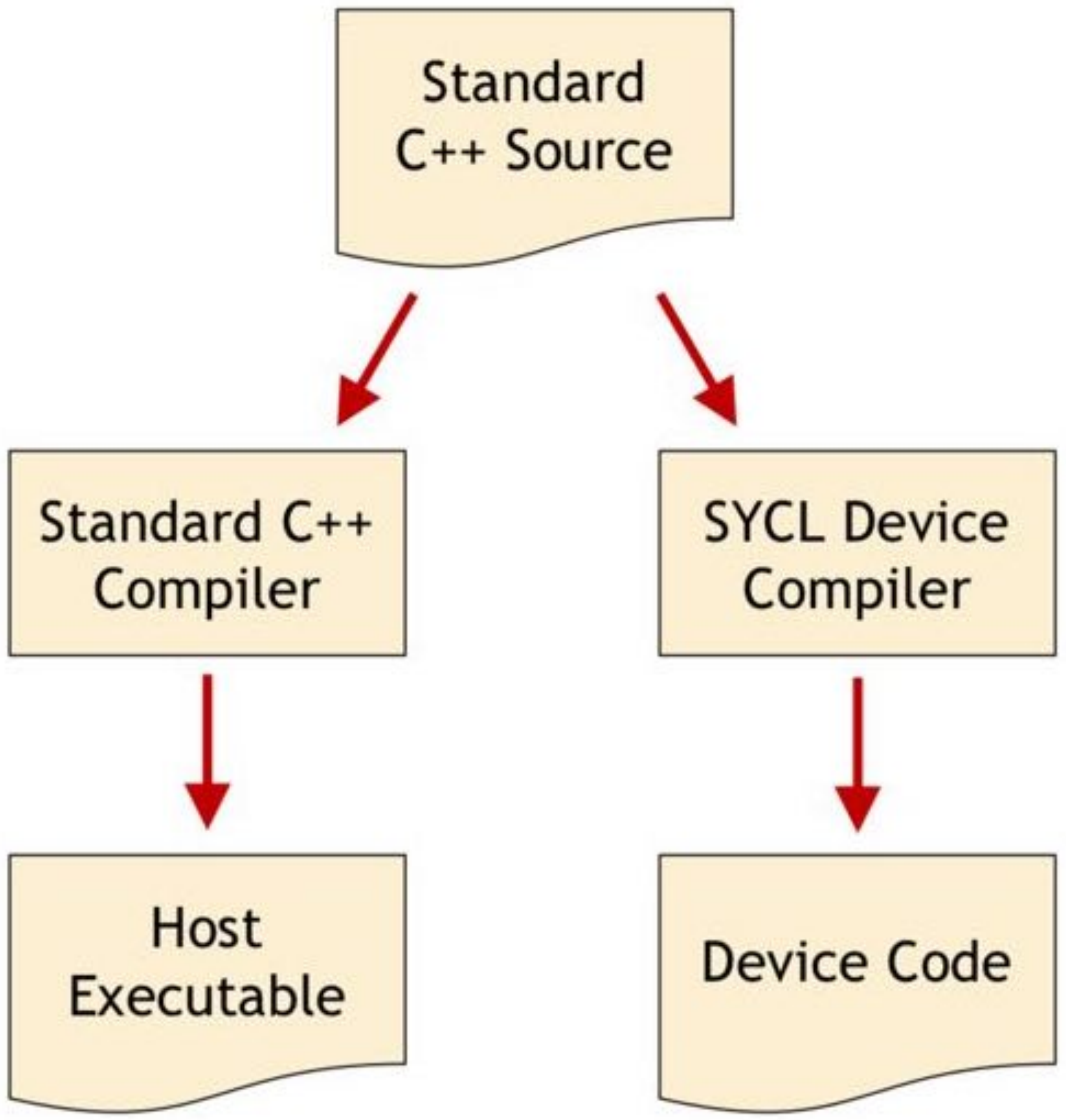
Vector Sum: SyCL Today

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

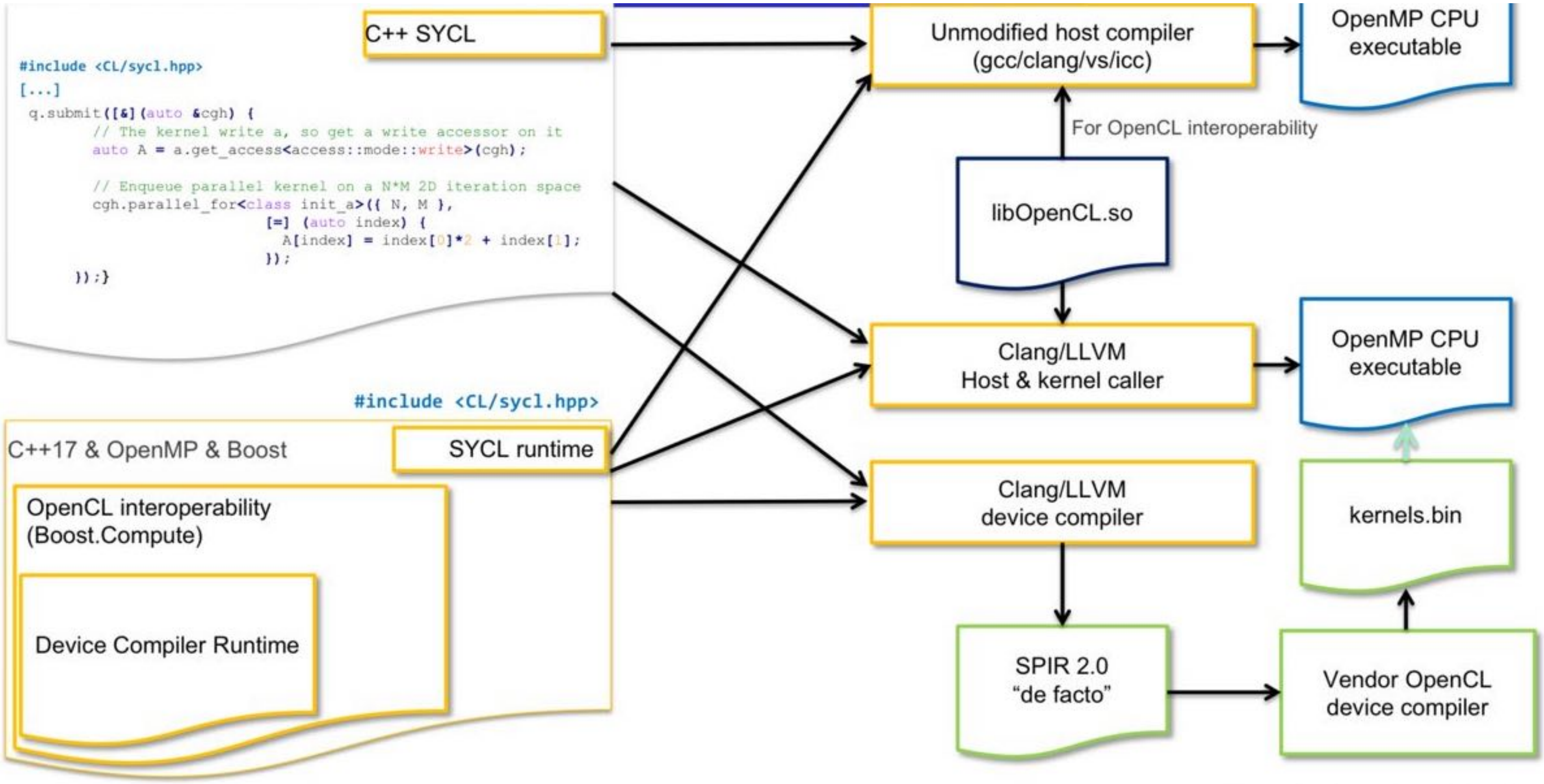
Vector Sum: SyCL STL

```cpp
void gArithmAddArr(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?



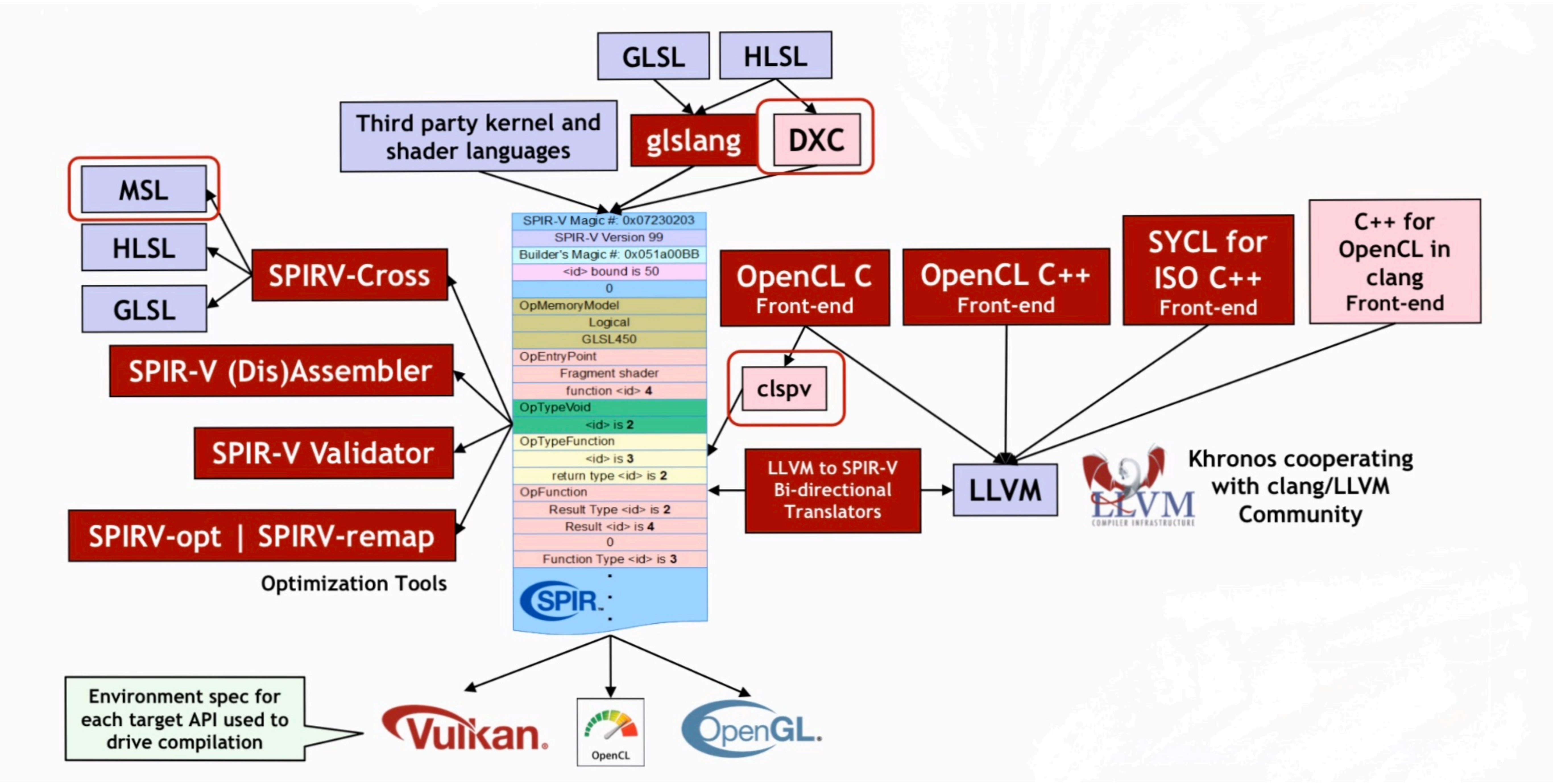
How SyCL works?



```
#include <CL/sycl.hpp>
[...]
 q.submit([&](auto &cgh) {
       // The kernel write a, so get a write accessor on it
       auto A = a.get_access<access::mode::write>(cgh);

       // Enqueue parallel kernel on a N*M 2D iteration space
       cgh.parallel_for<class init_a>({ N, M },
                                   [=] (auto index) {
                                     A[index] = index[0]*2 + index[1];
                                   });
     });}
```

C++ SYCL
#include <CL/sycl.hpp>
C++17 & OpenMP & Boost
SYCL runtime
OpenCL interoperability (Boost.Compute)
Device Compiler Runtime
Unmodified host compiler (gcc/clang/vs/icc)
OpenMP CPU executable
For OpenCL interoperability
libOpenCL.so
Clang/LLVM Host & kernel caller
OpenMP CPU executable
Clang/LLVM device compiler
kernels.bin
SPIR 2.0 "de facto"
Vendor OpenCL device compiler

How SyCL works?



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

Order Descriptors

To simplify loops optimization

How Halide works?

...by making loops implicit!

```
void gArithmAddArr(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 gArithmAddArr(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);
}
```

Parallel "for" loop

How Halide works?

```
void gArithmAddArr(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);
}
```

Vectorized "for" loop with "float8"

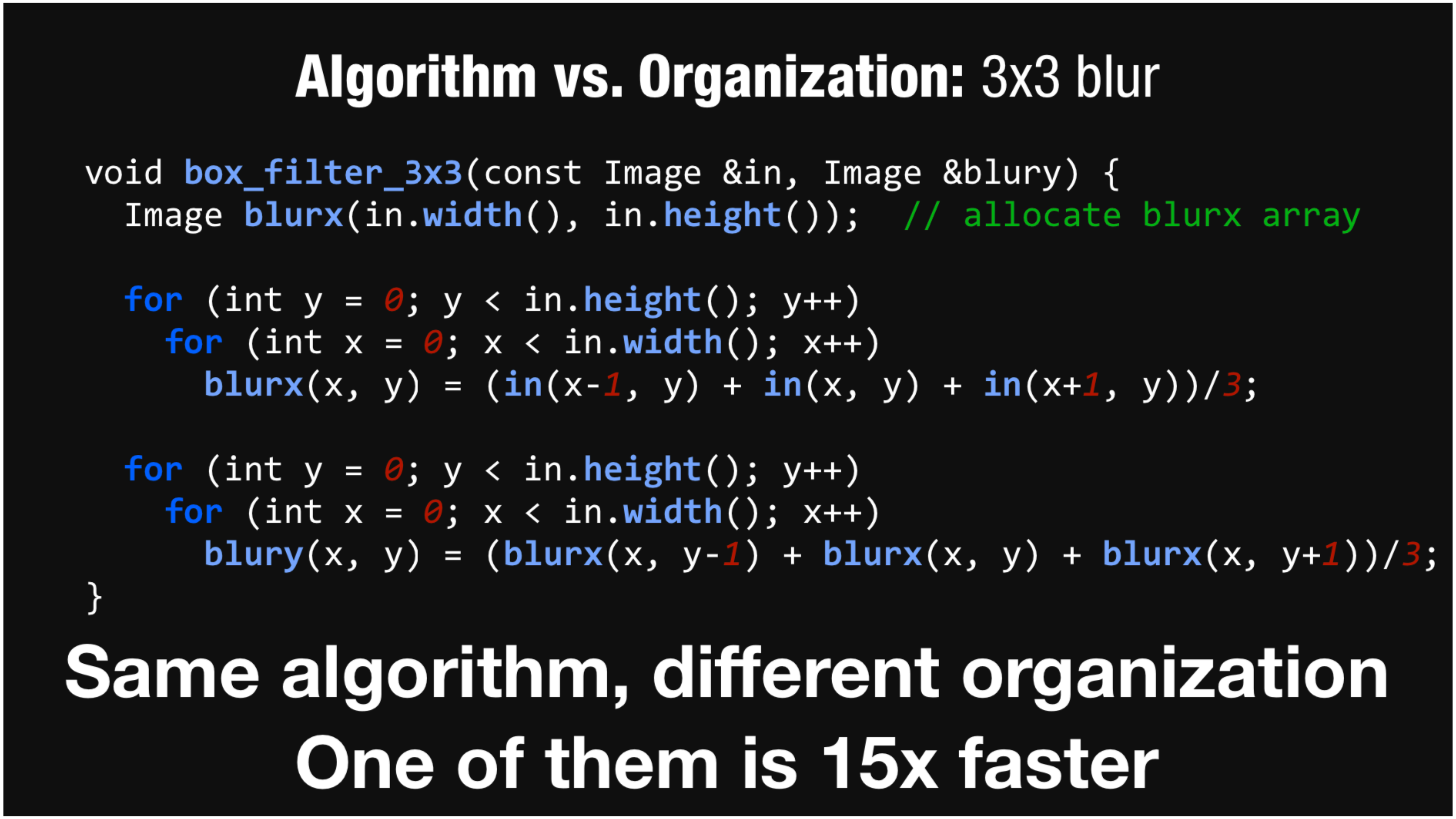
How Halide works?

```
void gArithmAddArr(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++



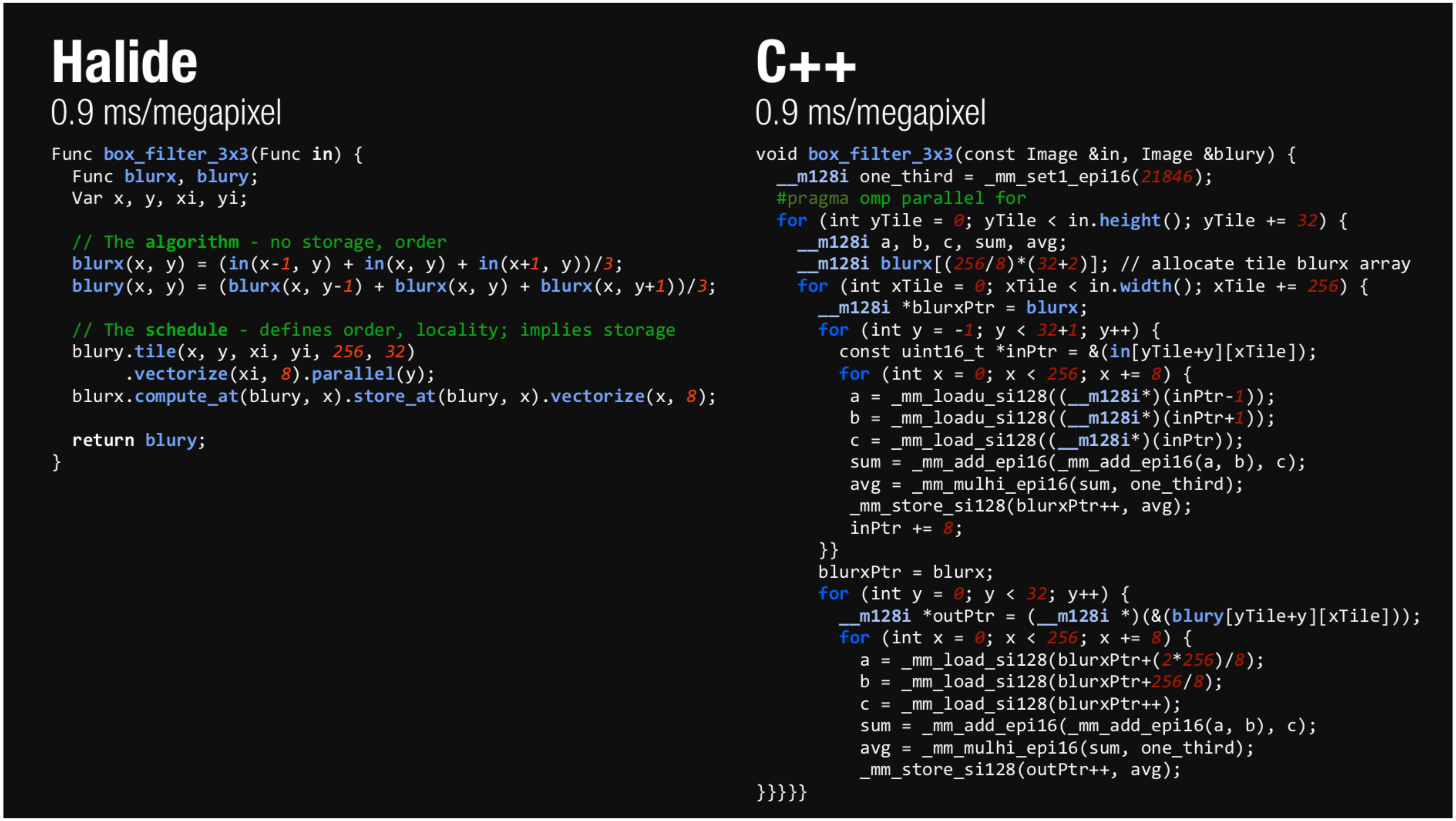
Algorithm vs. Organization: 3x3 blur

```
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;
}
```

Same algorithm, different organization

One of them is 15x faster

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;
}
```

C++

0.9 ms/megapixel

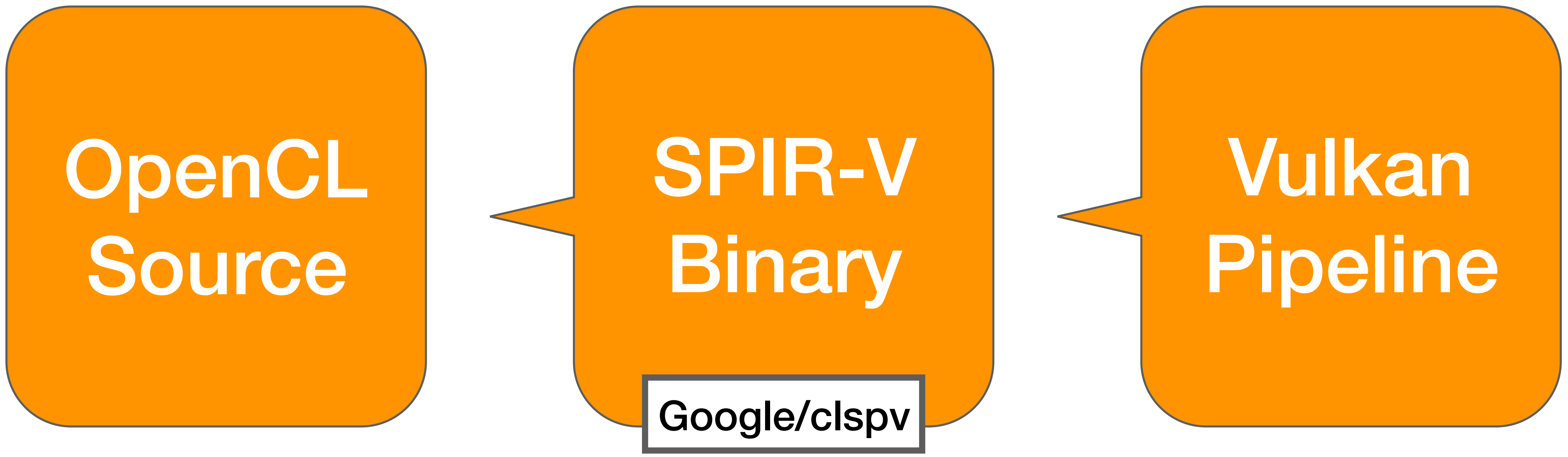
```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 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);
}}}}}
```

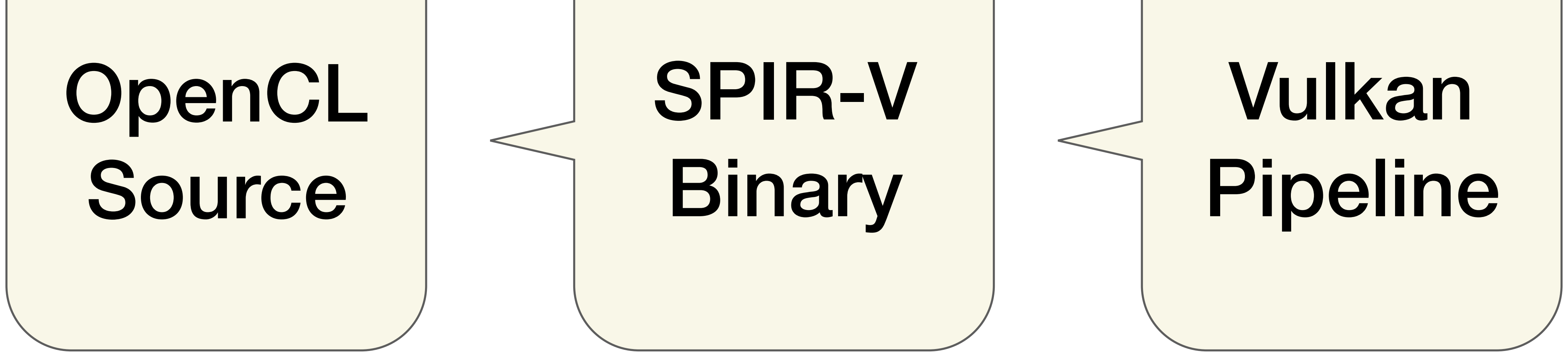
What to choose?

Compromises

Unified solution

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



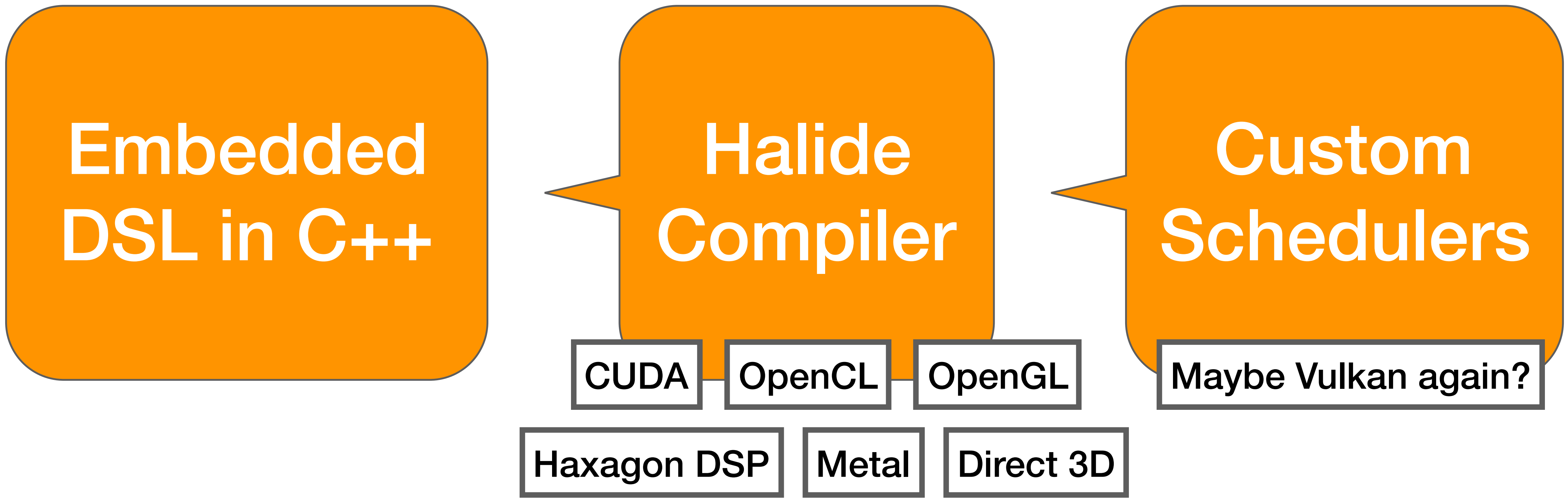


Pros
Same binary runs everywhere, easy to debug
Concurrent queues
Logical devices can represent SLI groups
Same ecosystem for both graphics and compute

Cons
Separate codebases
Experimental stage compilers
No modern C++ support until version 2.2
No support for CUDA features: warp shuffles, hardware-specific Instructions, L1 cache tuning

Flexible solution

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



Embedded DSL in C++

Halide Compiler

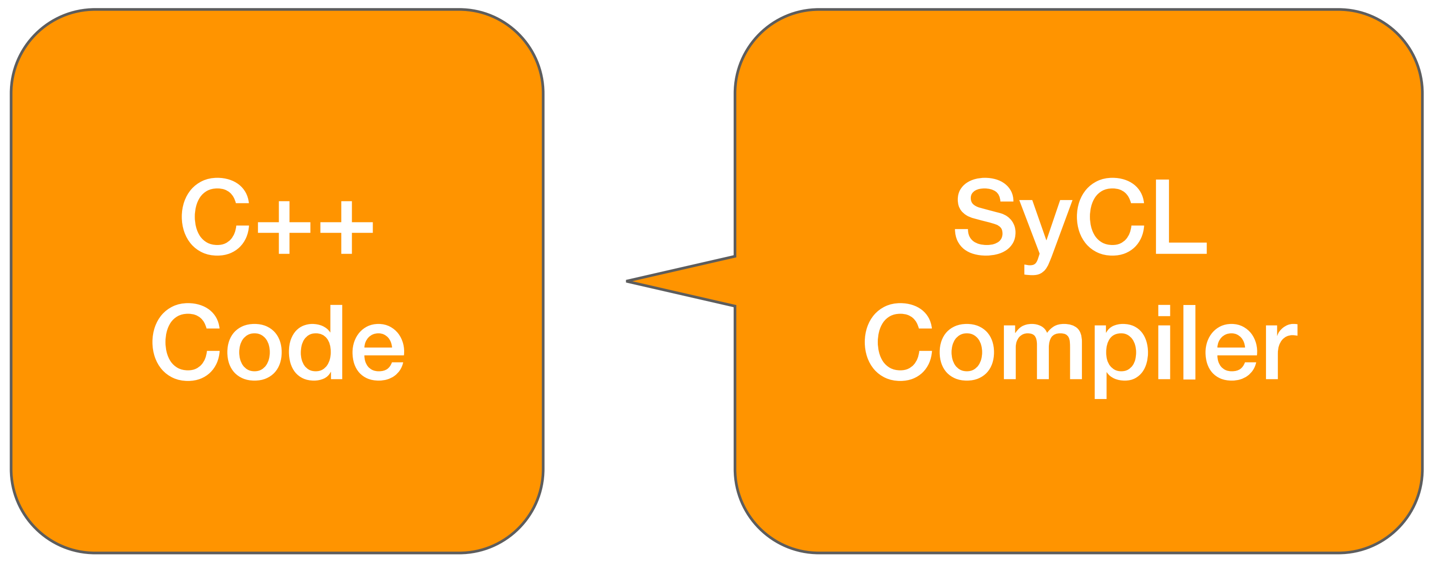
Custom Scheduler

Pros
Great for prototyping and benchmarking
Generate binaries for every platform
Easy to debug algorithms
Easy to export computational graphs into other libs
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
Use C++ templates and lambda functions for host & device code - just pass "sycl" policy
SYCL will not create C++ language extensions, but instead add features via C++ library
Does kernel fusion
Layered over OpenCL

Cons
Very immature, stability and C++17 adoption is expected closer to 2020
Underlying implementation requires compiler support
Kernel fusion may be weak

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
Already packed with binaries for multiple backends
Minimal coding required

Cons
Weak kernel fusion

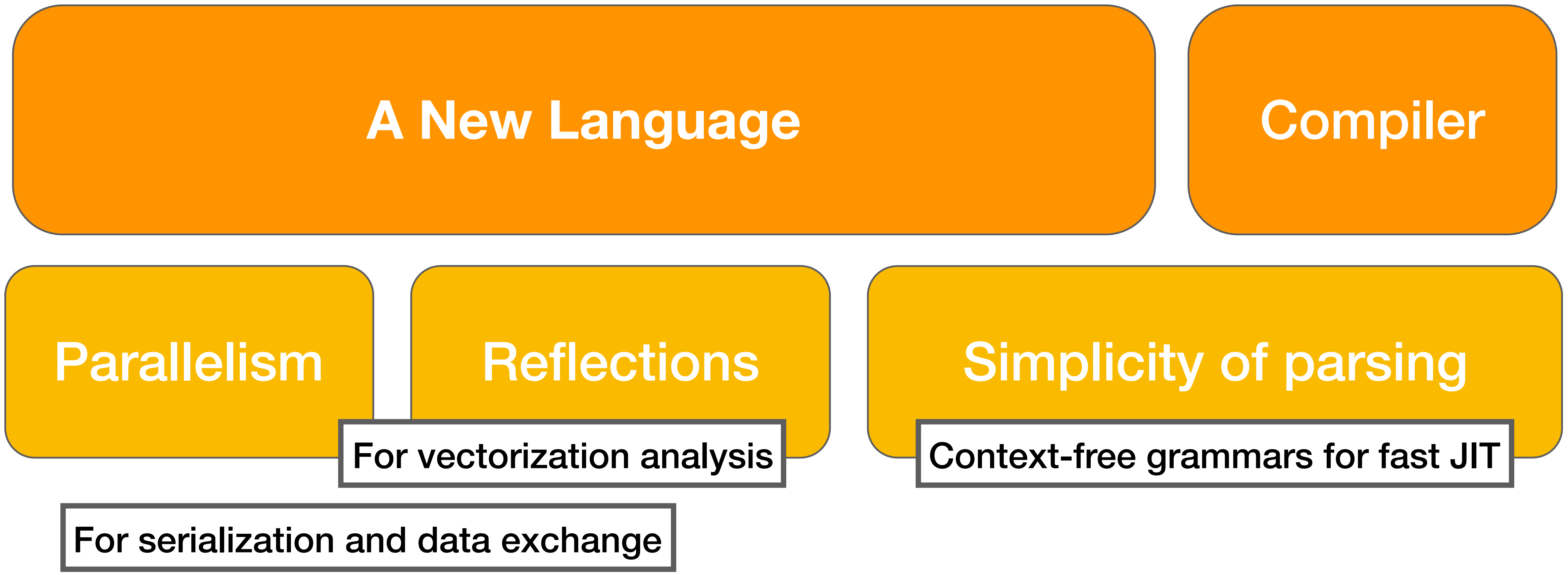
Comparison of recipes

...lets summarize our results!

	Simple	Unified	Flexible	Clean
Technology	ArrayFire	CL & SPIR-V	Halide	C++ SyCL
Write code once	Yes	Yes	T-Incomplete	Yes
Run everywhere	Almost	Yes	Yes	Eventually
Max performance	Average	High	Highest	High
Minimal code size	Smallest	Average	Large	Small

Bonus: Crazy solution

...if only you are as obsessed with parallel computing as I am!


A New Language
Compiler
Parallelism
Reflections
Simplicity of parsing
For vectorization analysis
Context-free grammars for fast JIT
For serialization and data exchange

Bonus: Hierarchy

...approximation of high-performance C++ GPGPU solutions

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

Q & A

github.com/ashvardanian/SandboxGPUs

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