

Data Parallelism

Gordon Brown & Michael Wong

CppCon 2020 – Sep 2020

- Learning objectives:
 - Learn the differences between task and data parallelism
 - Learn about Flynn's taxonomy
 - Learn the differences between multi-core CPU and many-core CPU
 - Learn about the current state of SIMD programming
 - Learn about auto-vectorization
 - Learn about the characteristics of several Heterogeneous programming languages

Task vs data parallelism



Task parallelism:

- Few large tasks with different operations / control flow
- Optimized for latency

Data parallelism:

- Many small tasks with same operations on multiple data
- Optimized for throughput

Review of Latency, bandwidth, throughput

- **Latency** is the amount of time it takes to travel through the tube.
- **Bandwidth** is how wide the tube is.
- The amount of water flow will be your **throughput**



Definition and examples

Latency is the time required to perform some action or to produce some result. Latency is measured in units of time -- hours, minutes, seconds, nanoseconds or clock periods.

Throughput is the number of such actions executed or results produced per unit of time. This is measured in units of whatever is being produced (cars, motorcycles, I/O samples, memory words, iterations) per unit of time. The term "memory bandwidth" is sometimes used to specify the throughput of memory systems.

bandwidth is the maximum rate of data transfer across a given path.

Example

An assembly line is manufacturing cars. It takes eight hours to manufacture a car and that the factory produces one hundred and twenty cars per day.

The latency is: 8 hours.

The throughput is: 120 cars / day or 5 cars / hour.



Flynn's Taxonomy

- Distinguishes multi-processor computer architectures along the two independent dimensions
 - *Instruction* and *Data*
 - Each dimension can have one state: *Single* or *Multiple*
- SISD: Single Instruction, Single Data
 - Serial (non-parallel) machine
- SIMD: Single Instruction, Multiple Data
 - Processor arrays and vector machines
- MISD: Multiple Instruction, Single Data (weird)
- MIMD: Multiple Instruction, Multiple Data

Assuming power is the constraint

What kind of processors are we building?

- CPU
 - complex control hardware
 - Increasing flexibility + performance
 - Expensive in power
- GPU
 - Simpler control structure
 - More HW per computation
 - Potentially more efficient in ops/watt
 - More restrictive programming model

Multicore CPU vs Manycore GPU

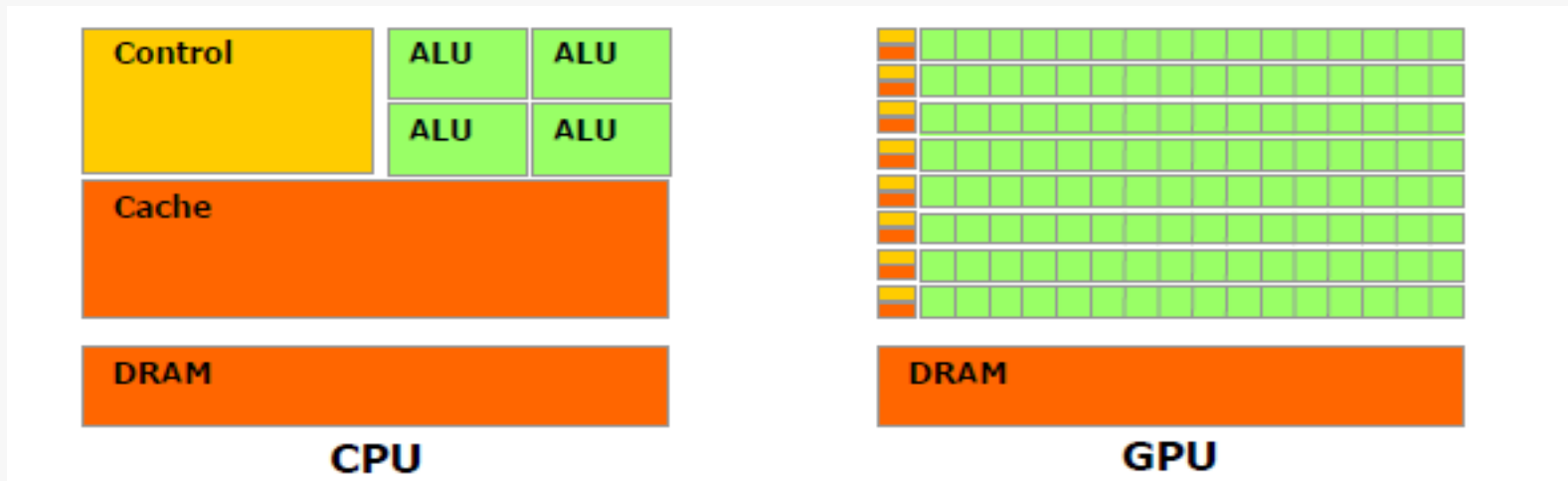
- Each core optimized for a single thread
 - Fast serial processing
 - Must be good at everything
 - Minimize latency of 1 thread
 - Lots of big on chip caches
 - Sophisticated controls
- Cores optimized for aggregate throughput, deemphasizing individual performance
 - Scalable parallel processing
 - Assumes workload is highly parallel
 - Maximize throughput of all threads
 - Lots of big ALUs
 - Multithreading can hide latency, no big caches
 - Simpler control, cost amortized over ALUs via SIMD

SIMD hard knocks

- SIMD architectures use data parallelism
- Improves tradeoff with area and power
 - Amortize control overhead over SIMD width
- Parallelism exposed to programmer & compiler
- Hard for a compiler to exploit SIMD
- Hard to deal with sparse data & branches
 - C and C++ Difficult to vectorize, Fortran better
- So
 - Either forget SIMD or hope for the autovectorizer
 - Use compiler intrinsics

Memory

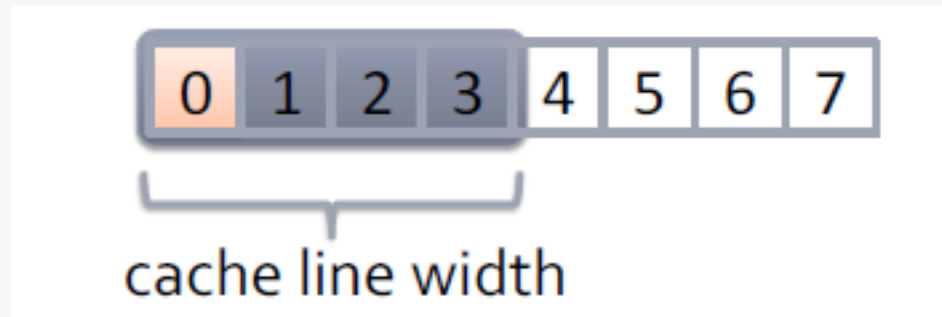
- Many core gpu is a device for turning a compute bound problem into a memory bound problem



- Lots of processors but only one socket
- Memory concerns dominate performance tuning

Memory is SIMD too

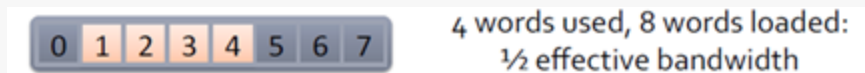
- Virtually all processors have SIMD memory subsystems



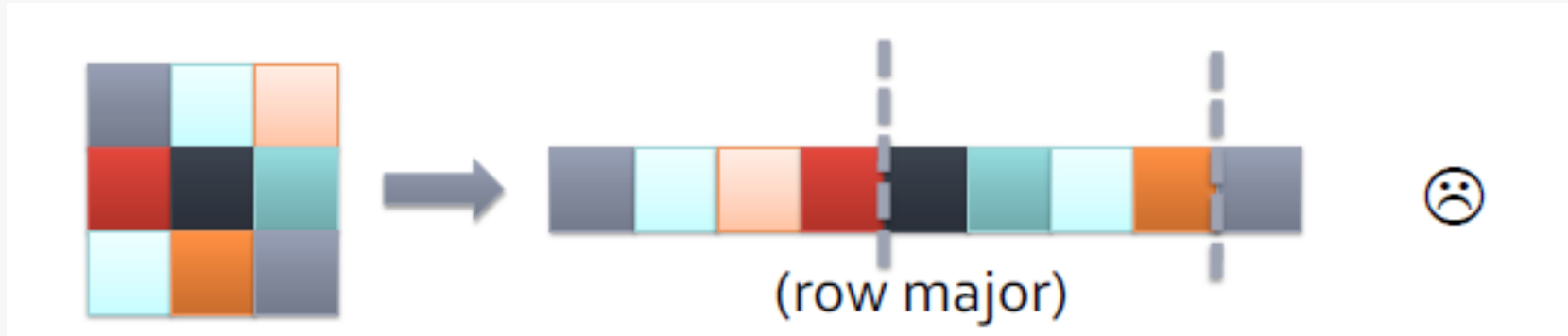
- This has 2 effects
 - Sparse access wastes bandwidth



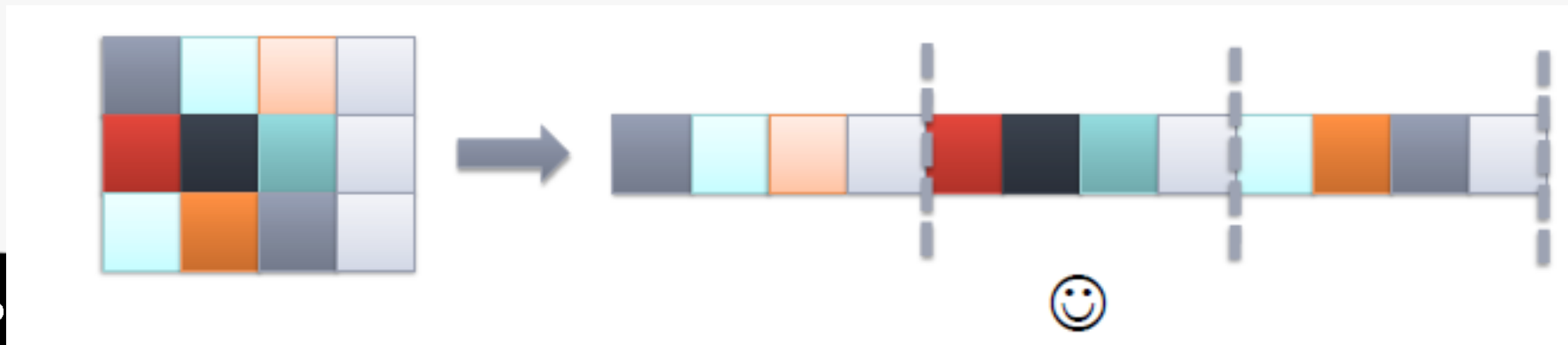
- Unaligned access wastes bandwidth



Data Structure Padding



- Multidimensional arrays are usually stored as monolithic vectors in memory
- Care should be taken to assure aligned memory accesses for the necessary access pattern



Coalescing

- GPUs and CPUs both perform memory transactions at a larger granularity than the program requests (cache line)
- GPUs have a coalescer which examines memory requests dynamically and coalesces them
- To use bandwidth effectively, when threads load, they should
 - Present a set of unit strided loads (dense accesses)
 - Keep sets of loads aligned to vector boundaries

SIMD Language Extensions

- IBM currently has 7 SIMD architectures
 - VMX, VMX128, VSX, SPE, BGL, BGQ, QPX
 - Each has its own proprietary language extension
 - Code written for one language extension can't be moved without a rewrite
 - We don't even have compatibility within our own company
- Intel has 7 SIMD architectures
 - MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX-512/MIC
 - MMX, SSEx, AVX each have a different language extension
 - Code written for one language extension can't be moved without a rewrite

“The Great Hope” - Auto-Vectorizing compilers

- SIMD floating point entered the market 15 years ago
 - (Intel Pentium III, Motorola G4, AMD K6-2)
- Software industry held its breath waiting for a “magic” auto-vectorizing compiler (including Microsoft)
- Despite 15 years of research and development the industry still doesn’t have a good auto-vectorizing compiler
- Industry instead ended up with primitive language support
 - Multiple non-compatible language extensions
 - Compiler intrinsics
- Using intrinsics humans still produce superior vector code but at great pain

Why autovectorization is hard?

- SIMD register width has increased from 128-256-512, 1024 soon
- Instructions are more powerful and complex
 - Hard for compiler to select proper instruction
 - Code pattern needs to be recognized by the compiler
 - Precision requirements often inhibit SIMD codegen

What sort of loops can be vectorized?

- Countable
- Single entry, single exit
- Straight-line code
- Innermost loop of a nest
- No function calls
- Certain non-contiguous memory access
- Some Data dependencies
- Efficient Alignment
- Mixed data types
- Non-unit stride between elements

• Loop body too complex (register pressure)

Industry needs better language support for SIMD

- 80% of C++ programmers time spent vectorizing code
- Need to reduce programming effort
 - Fewer code modifications to vectorize
 - Rapid conversion of scalar to vector code
- Code portability
 - Don't rewrite for every SIMD architecture
- Less code maintenance
 - Intrinsics impossible to maintain
 - Easier to rewrite than figuring out what the code is doing
- Support required vendor-specific extensions

C++ 20 Parallelism

- No standard!
- Boost.SIMD
- Proposal N3571 by Mathias Gaunard et. al., based on the Boost.SIMD library.
- Proposal N4184 by Matthias Kretz, based on Vc library.
- Unifying efforts and expertise to provide an API to use SIMD portably
- Within C++
- P0193 status report
- P0203 design considerations
- P0214 latest SIMD paper

SIMD from Matthias Kretz

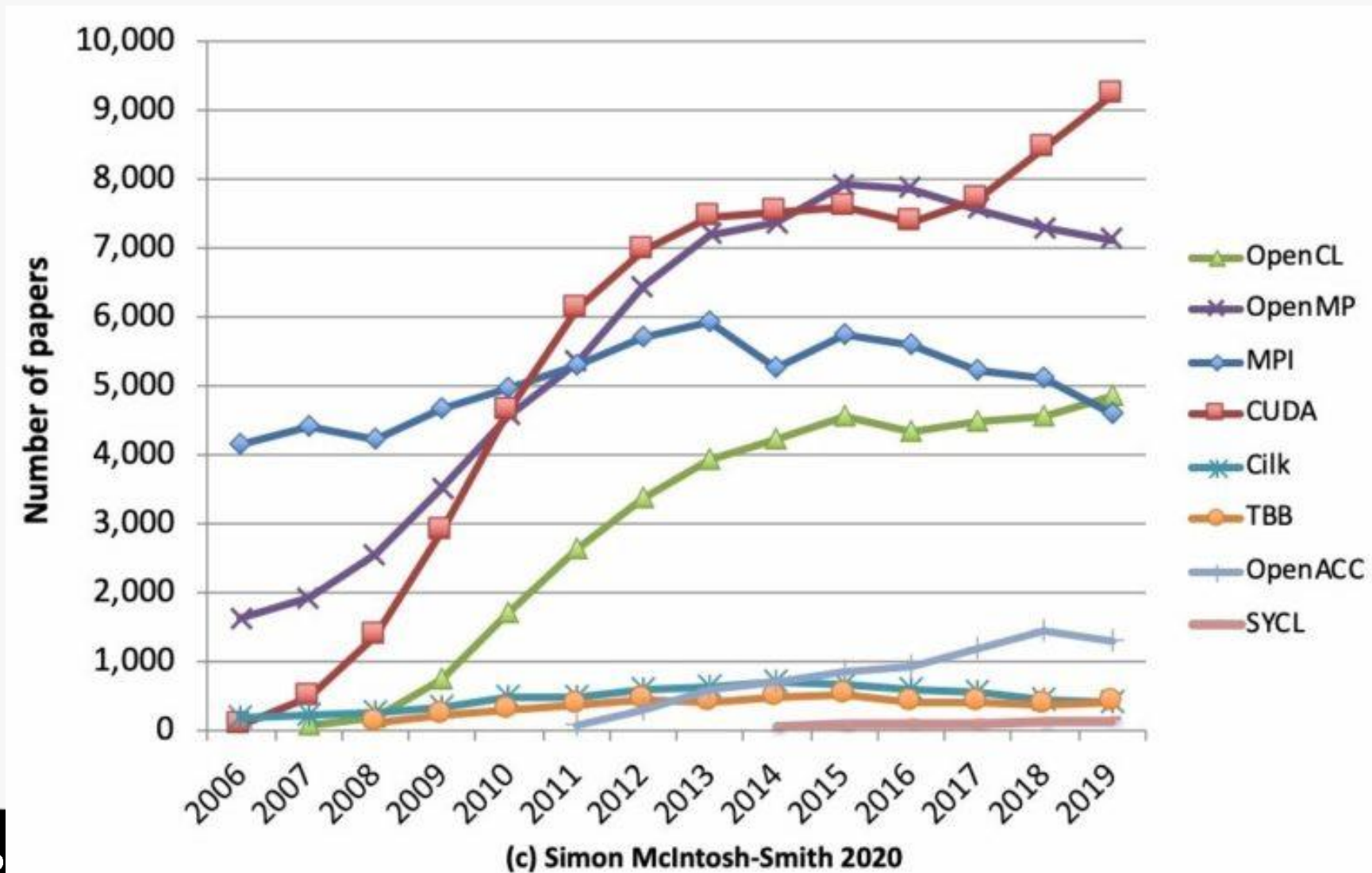
- `std::simd<T, N, Abi>`
 - `simd<T, N>` SIMD register holding N elements of type T
 - `simd<T>` same with optimal N for the currently targeted architecture
 - Abi Defaulted ABI marker to make types with incompatible ABI different
 - Behaves like a value of type T but applying each operation on the N values it contains, possibly in parallel.
- Constraints
 - T must be an integral or floating-point type (tuples/struct of those once we get reflection)
 - N parameter under discussion, probably will need to be power of 2.

Operations on SIMD

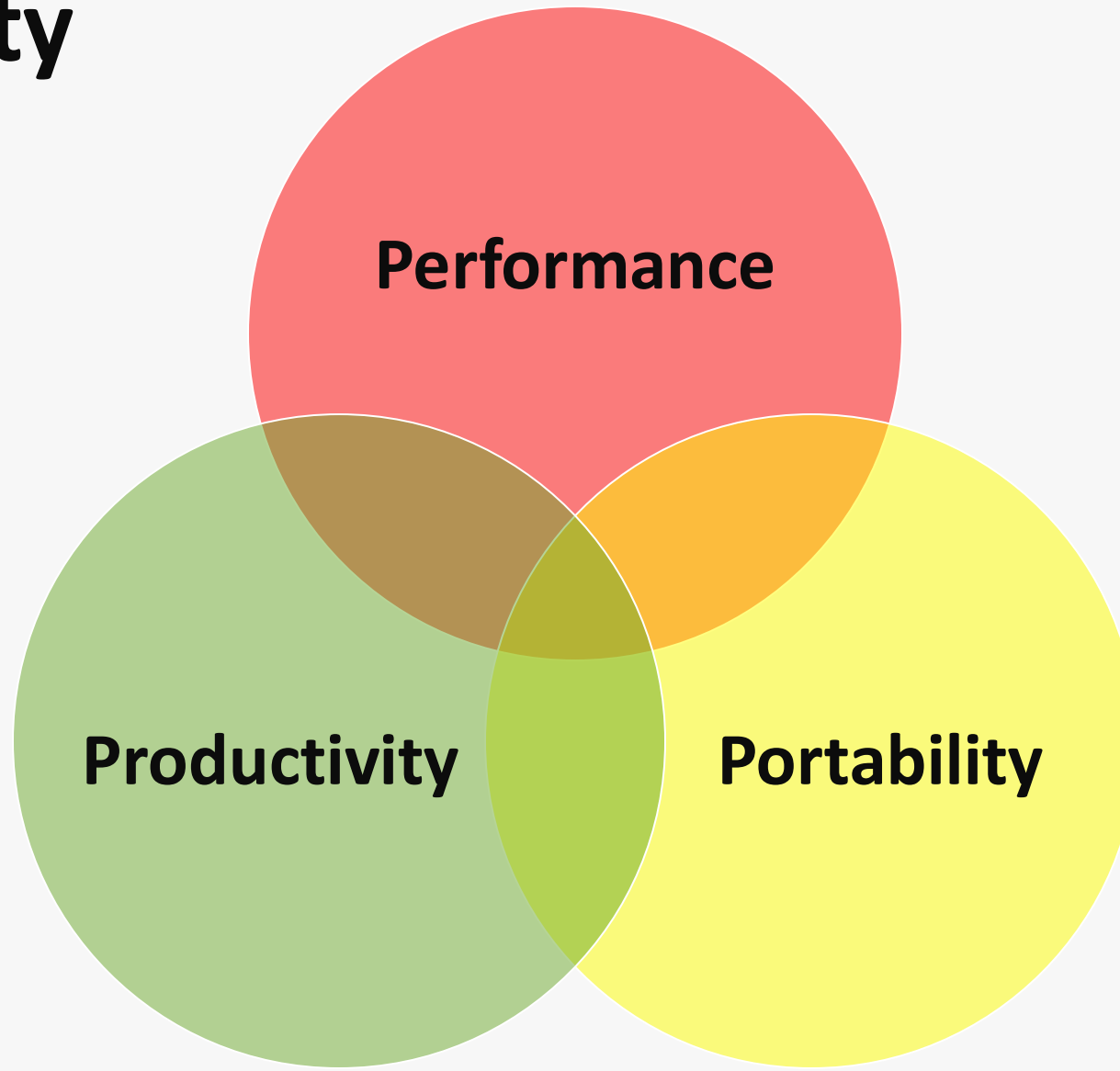
- Built-in operators
- All usual binary operators are available, for all:
 - `simd<T, N> simd<U, N>`
 - `simd<T, N> U, U simd<T, N>`
- Compound binary operators and unary operators as well
 - `simd<T, N>` convertible to `simd<U, N>`
 - `simd<T, N>(U)` broadcasts the value
- No promotion:
 - `simd<uint8_t>(255) + simd<uint8_t>(1) == simd<uint8_t>(0)`
- Comparisons and conditionals:
 - `==`, `!=`, `<`, `<=`, `>` and `>=` perform element-wise comparison return `mask<T, N, Abi>`
 - `if(cond) x = y` is written as `where(cond, x) = y`
 - `cond ? x : y` is written as `if_else(cond, x, y)`

Heterogenous Programming Landscape

Simon McIntosh-Smith annual language citations



The Reality



Long Answer



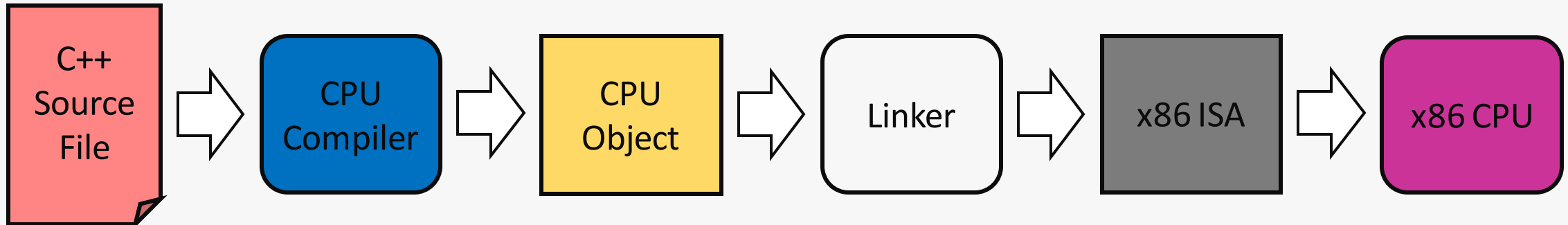
- The right programming model can allow you to express a problem in a way which adapts to different architectures

Problem composition

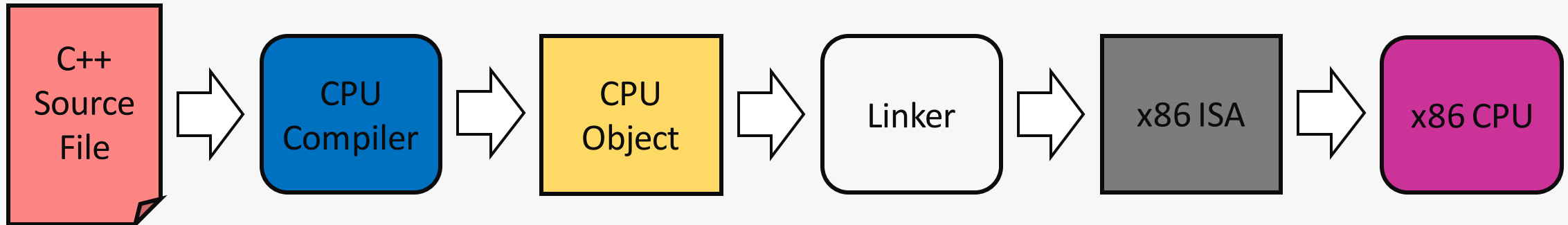
So if you can't write a single program to run everywhere

- You need a programming model which allows you to compose your problem in different ways

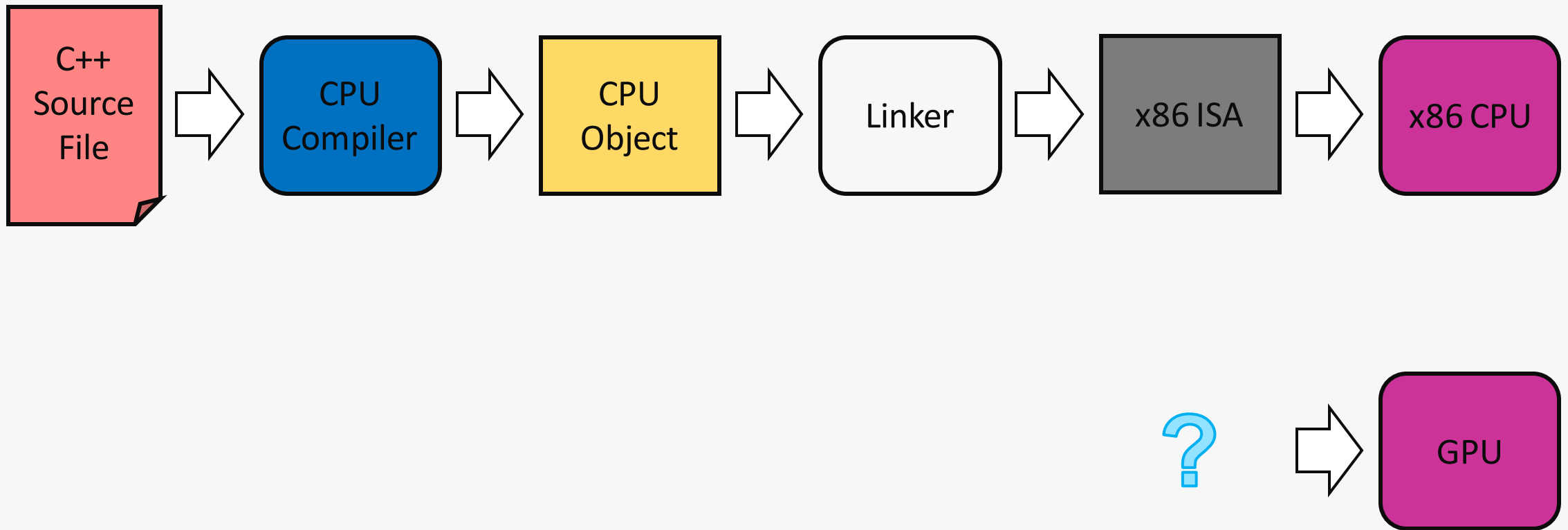
C++ Compilation Model



C++ Compilation Model



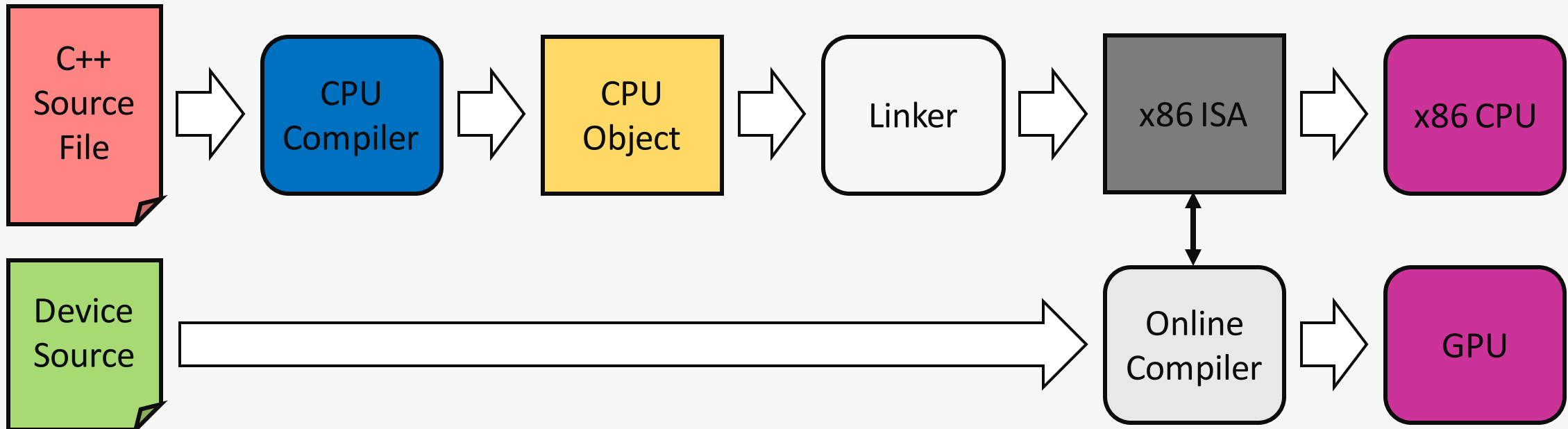
C++ Compilation Model



How can we compile source code for a sub architectures?

- Separate source (OpenCL C, OpenCL C++, GLSL)
- Single source (SYCL, C++, CUDA, OpenMP, C++ AMP)
- Embedded DSLs (RapidMind, Halide)

Separate Source Compilation Model

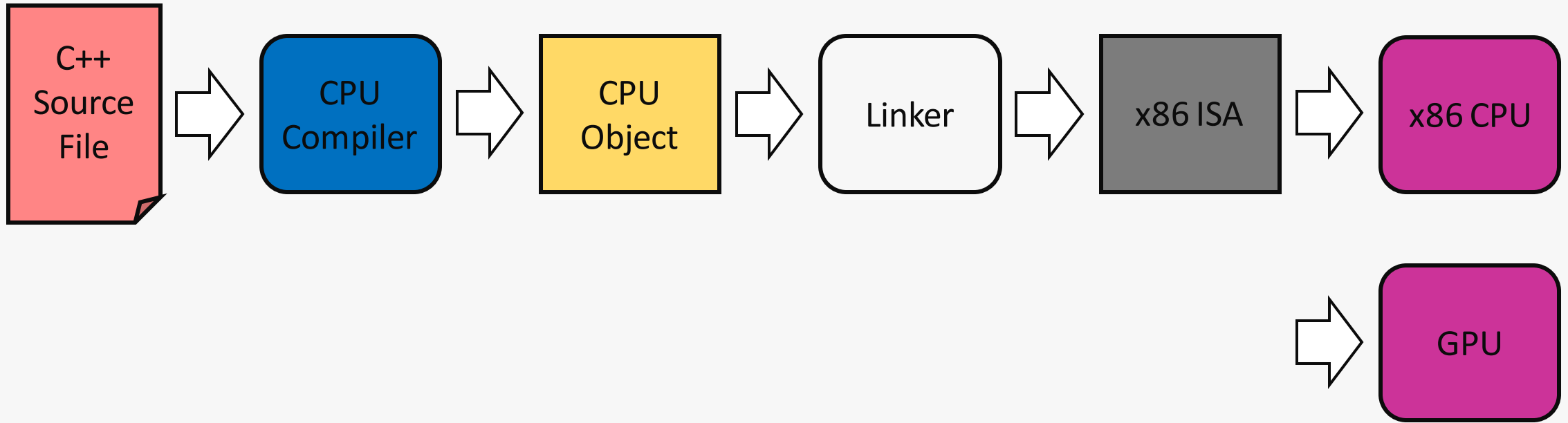


```
float *a, *b, *c;
...
kernel k = clCreateKernel(..., "my_kernel", ...)
clEnqueueWriteBuffer(..., size, a, ...);
clEnqueueWriteBuffer(..., size, a, ...);
clEnqueueNDRange(..., k, 1, {size, 1, 1}, ...);
clEnqueueWriteBuffer(..., size, c, ...);
```

Here we're using OpenCL as an example

```
void my_kernel(__global float *a, __global float *b,
               __global float *c) {
    int id = get_global_id(0);
    c[id] = a[id] + b[id];
}
```

Single Source Compilation Model

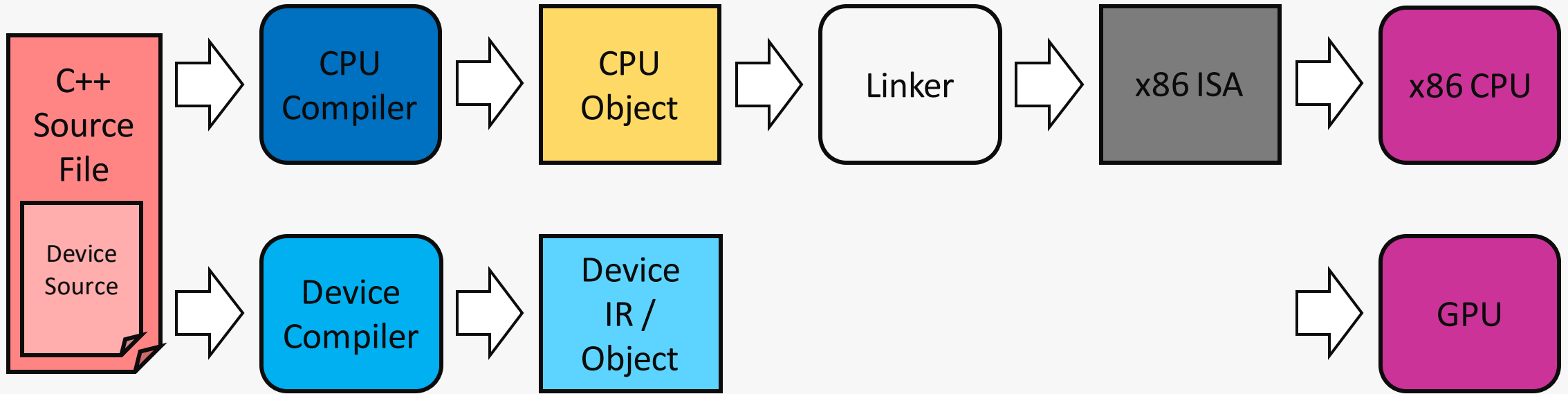


```
array_view<float> a, b, c;  
extent<2> e(64, 64);
```

```
parallel_for_each(e, [=](index<2> idx) restrict(amp) {  
    c[idx] = a[idx] + b[idx];  
});
```

Here we are using C++ AMP as an example

Single Source Compilation Model

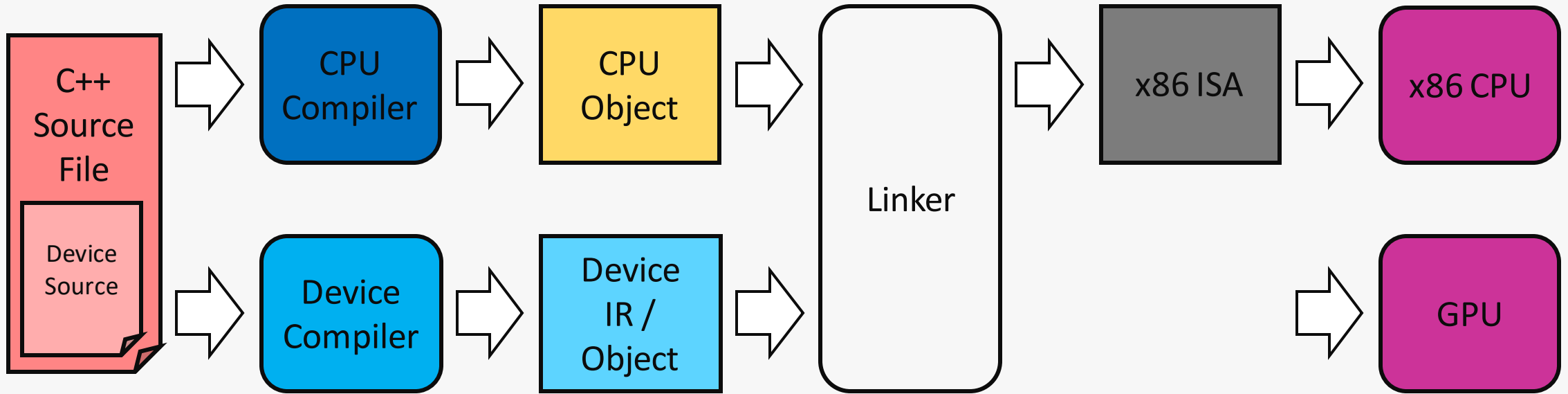


```
array_view<float> a, b, c;  
extent<2> e(64, 64);
```

```
parallel_for_each(e, [=](index<2> idx) restrict(amp) {  
    c[idx] = a[idx] + b[idx];  
});
```

Here we are using C++ AMP as an example

Single Source Compilation Model

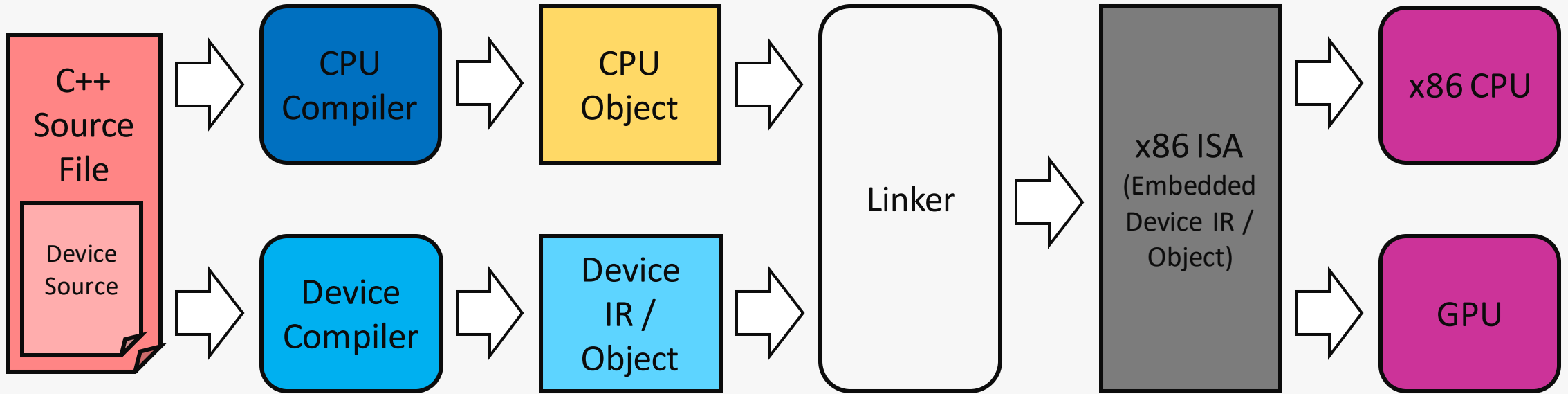


```
array_view<float> a, b, c;  
extent<2> e(64, 64);
```

```
parallel_for_each(e, [=](index<2> idx) restrict(amp) {  
    c[idx] = a[idx] + b[idx];  
});
```

Here we are using C++ AMP as an example

Single Source Compilation Model



```
array_view<float> a, b, c;  
extent<2> e(64, 64);
```

```
parallel_for_each(e, [=](index<2> idx) restrict(amp) {  
    c[idx] = a[idx] + b[idx];  
});
```

Here we are using C++ AMP as an example

Benefits of Single Source

- Device code is written in C++ in the same source file as the host CPU code
- Allows compile-time evaluation of device code
- **Supports type safety across host CPU and device**
- **Supports generic programming**
- Removes the need to distribute source code

Describing Parallelism

How do you represent the different forms of parallelism?

- Directive vs explicit parallelism
- Task vs data parallelism
- Queue vs stream execution

Directive vs Explicit Parallelism

Examples:

- OpenMP, OpenACC

Implementation:

- Compiler transforms code to be parallel based on pragmas

Here we're using OpenMP as an example

```
vector<float> a, b, c;

#pragma omp parallel for
for(int i = 0; i < a.size(); i++) {
    c[i] = a[i] + b[i];
}
```

Examples:

- SYCL, CUDA, TBB, Fibers, C++11 Threads

Implementation:

- An API is used to explicitly enqueue one or more threads

Here we're using C++ AMP as an example

```
array_view<float> a, b, c;
extent<2> e(64, 64);
parallel_for_each(e, [=](index<2> idx)
restrict(amp) {
    c[idx] = a[idx] + b[idx];
});
```

Task vs Data Parallelism

Examples:

- OpenMP, C++11 Threads, TBB

Implementation:

- Multiple (potentially different) tasks are performed in parallel

Examples:

- C++ AMP, CUDA, SYCL, C++17 ParallelSTL

Implementation:

- The same task is performed across a large data set

Here we're using TBB as an example

```
vector<task> tasks = { ... };  
  
tbb::parallel_for_each(tasks.begin(),  
    tasks.end(), [=](task &v) {  
    task();  
});
```

Here we're using CUDA as an example

```
float *a, *b, *c;  
cudaMalloc((void **)&a, size);  
cudaMalloc((void **)&b, size);  
cudaMalloc((void **)&c, size);  
  
vec_add<<<64, 64>>>(a, b, c);
```

Queue vs Stream Execution

Examples:

- C++ AMP, CUDA, SYCL, C++17 ParallelSTL

Implementation:

- Functions are placed in a queue and executed once per enqueueer

Examples:

- BOINC, BrookGPU

Implementation:

- A function is executed on a continuous loop on a stream of data

Here we're using CUDA as an example

```
float *a, *b, *c;
cudaMalloc((void **)&a, size);
cudaMalloc((void **)&b, size);
cudaMalloc((void **)&c, size);

vec_add<<<64, 64>>>(a, b, c);
```

Here we're using BrookGPU as an example

```
reduce void sum (float a<>,
                 reduce float r<>) {
    r += a;
}
float a<100>;
float r;
sum(a, r);
```

Data Locality & Movement

One of the biggest limiting factor in parallel and heterogeneous computing

- Cost of data movement in time and power consumption

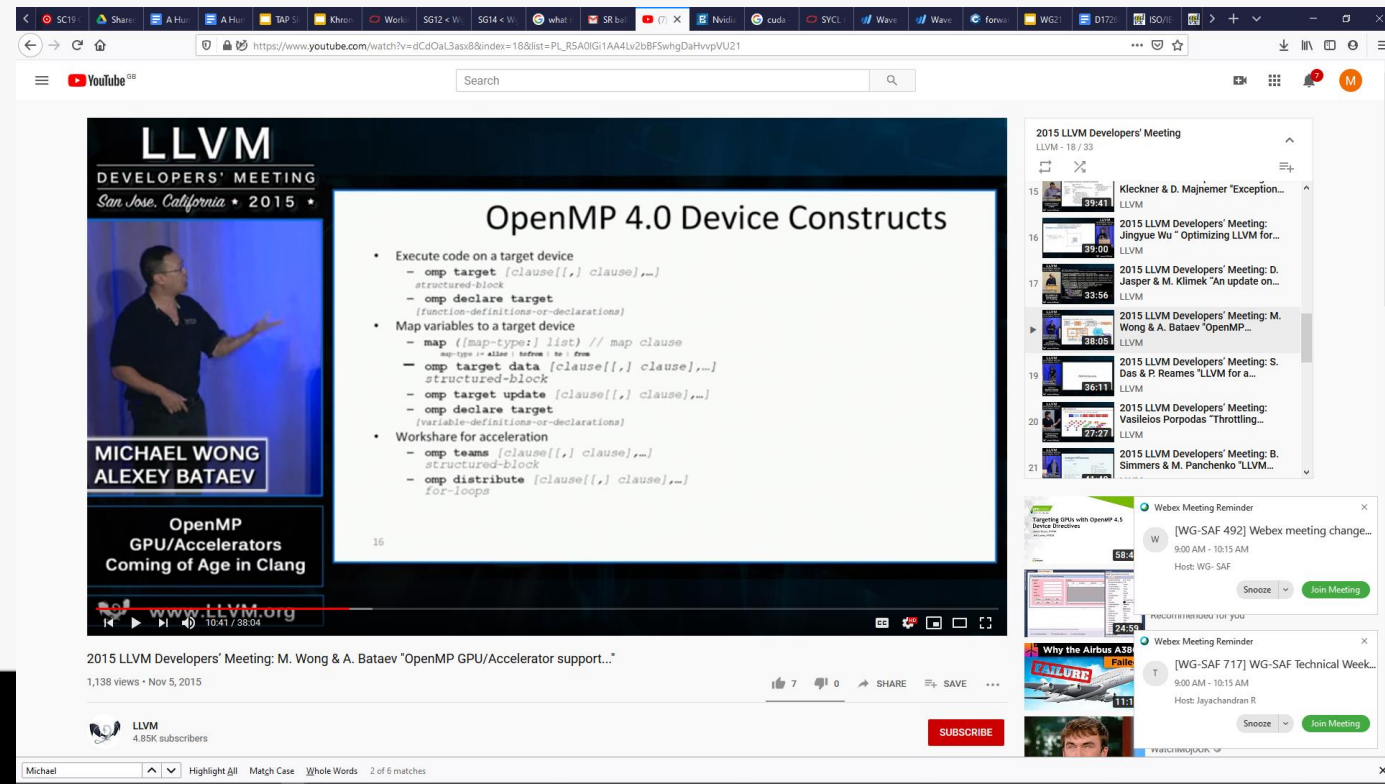
Cost of Data Movement

- It can take considerable time to move data to a device
 - This varies greatly depending on the architecture
- The bandwidth of a device can impose bottlenecks
 - This reduces the amount of throughput you have on the device
- Performance gain from computation $>$ cost of moving data
 - If the gain is less than the cost of moving the data it's not worth doing
- Many devices have a hierarchy of memory regions
 - Global, read-only, group, private
 - Each region has different size, affinity and access latency
 - Having the data as close to the computation as possible reduces the cost

https://www.youtube.com/watch?v=dCdOaL3asx8&index=18&list=PL_R5A0IGi1AA4Lv2bBFSwhgDaHvvpVU21

How do you move data from the host CPU to a device and back?

- Implicit vs explicit data movement



The screenshot shows a YouTube video player with the following details:

- Video Title:** OpenMP 4.0 Device Constructs
- Presenters:** MICHAEL WONG, ALEXEY BATAEV
- Event:** 2015 LLVM Developers' Meeting, San Jose, California • 2015 •
- Slide Content:**
 - Execute code on a target device**
 - `- omp target [clause[[],] clause],...`
 - `structured-block`
 - `- omp declare target`
 - `[function-definitions-or-declarations]`
 - Map variables to a target device**
 - `- map ([map-type:] list) // map clause`
 - `map-type: in-alias: before: to: from`
 - `- omp target data [clause[[],] clause],...`
 - `structured-block`
 - `- omp target update [clause[[],] clause],...`
 - `- omp declare target`
 - `[variable-definitions-or-declarations]`
 - Workshare for acceleration**
 - `- omp teams [clause[[],] clause],...`
 - `structured-block`
 - `- omp distribute [clause[[],] clause],...`
 - `for-loops`
- Bottom Text:** OpenMP GPU/Accelerators Coming of Age in Clang
- URL:** www.LLVM.org

Below the video player, the video title is repeated: "2015 LLVM Developers' Meeting: M. Wong & A. Bataev 'OpenMP GPU/Accelerator support...'. It shows 1,138 views and a date of Nov 5, 2015. There are 7 likes and 0 dislikes. The video is from the channel "LLVM" which has 4.85K subscribers. A "SUBSCRIBE" button is visible.

On the right side of the screenshot, there is a list of other videos from the "2015 LLVM Developers' Meeting" series, including talks by Kleckner & D. Majnemer, Jingyue Wu, D. Jasper & M. Klimek, M. Wong & A. Bataev, S. Das & P. Reames, Vasileios Porpodas, and B. Simmers & M. Panchenko.

At the bottom of the screenshot, there are two "Webex Meeting Reminder" notifications for "WG-SAF 492" and "WG-SAF 717".

Implicit vs Explicit Data Movement

Examples:

- SYCL, C++ AMP

Implementation:

- Data is moved to the device implicitly via cross host CPU / device data structures

Here we're using C++ AMP as an example

```
array_view<float> ptr;  
extent<2> e(64, 64);  
parallel_for_each(e, [=](index<2> idx)  
restrict(amp) {  
    ptr[idx] *= 2.0f;  
});
```

Examples:

- OpenCL, CUDA, OpenMP

Implementation:

- Data is moved to the device via explicit copy APIs

Here we're using CUDA as an example

```
float *h_a = { ... }, d_a;  
cudaMalloc((void **)&d_a, size);  
cudaMemcpy(d_a, h_a, size,  
            cudaMemcpyHostToDevice);  
vec_add<<<64, 64>>>(a, b, c);  
cudaMemcpy(d_a, h_a, size,  
            cudaMemcpyDeviceToHost);
```


How do you address memory between host CPU and device?

- Multiple address space
- Non-coherent single address space
- Cache coherent single address space

Comparison of Memory Models

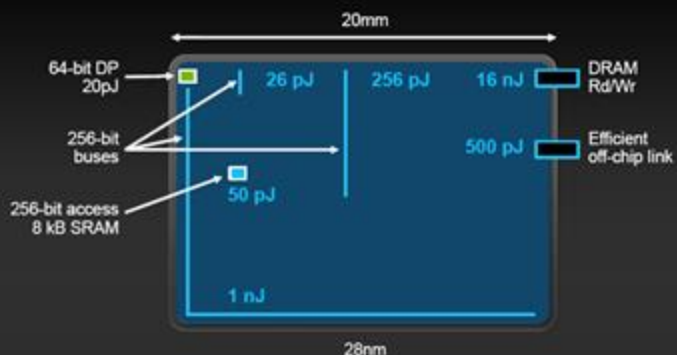
- Flat
 - C++
- Multiple address space
 - SYCL 1.2.1, C++AMP, OpenCL 1.x, CUDA, SYCL 2020
 - Pointers have keywords or structures for representing different address spaces
 - Allows finer control over where data is stored, but needs to be defined explicitly
- Non-coherent single address space
 - HSA, OpenCL 2.x , CUDA 4, SYCL 2020 usm
 - Pointers address a shared address space that is mapped between devices
 - Allows the host CPU and device to access the same address, but requires mapping
- Cache coherent single address space
 - HSA, OpenCL 2.x, CUDA 6
 - Pointers address a shared address space (hardware or cache coherent runtime)
 - Allows concurrent access on host CPU and device, but can be inefficient for large data



The Four Horsemen

The High Cost of Data Movement

Fetching operands costs more than computing on them



Socket 0

Core 0

Core 1

0

1

2

3

0

1

2

3

0

4

1

5

Socket 1

Core 0

Core 1

0

1

2

3

0

1

2

3

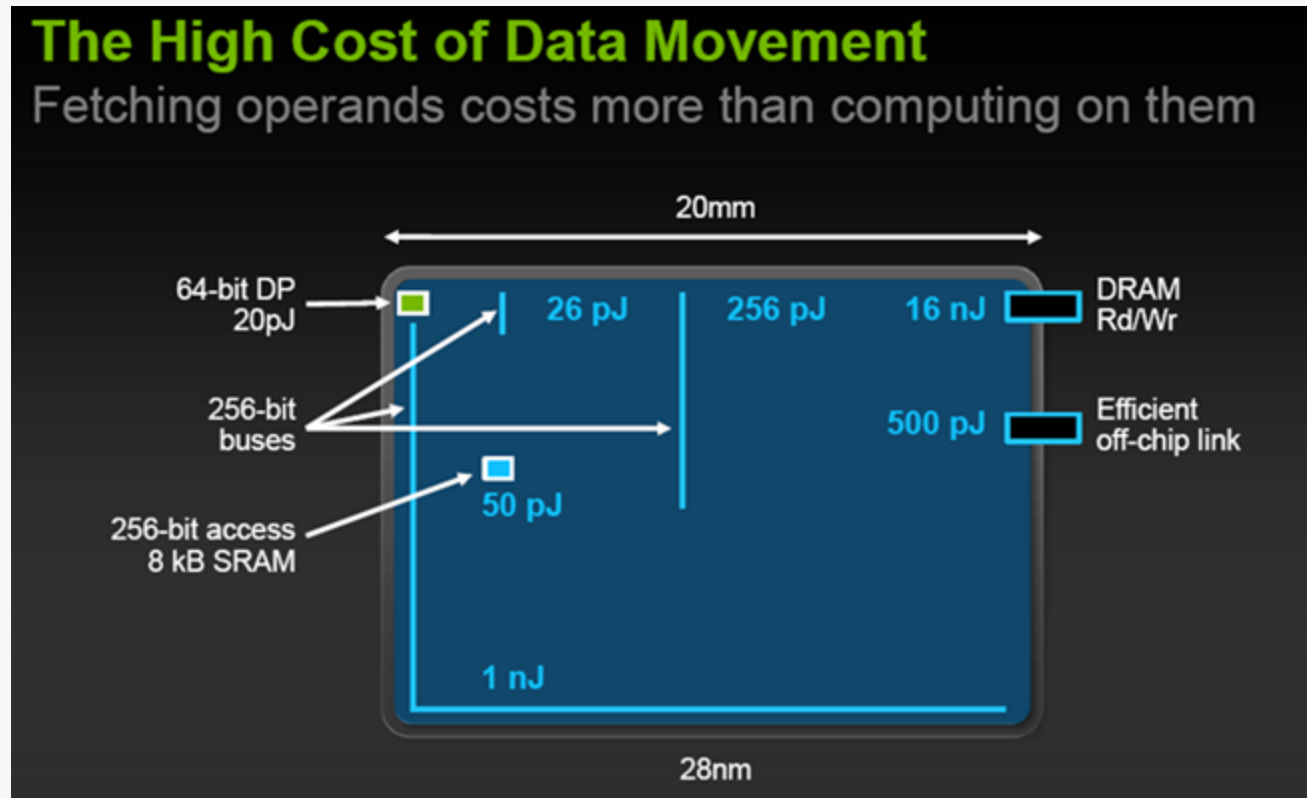
2

6

3

7

Cost of Data Movement



**Credit: Bill Dally, Nvidia,
2010**

- 64bit DP Op:
 - 20pJ
- 4x64bit register read:
 - 50pJ
- 4x64bit move 1mm:
 - 26pJ
- 4x64bit move 40mm:
 - 1nJ
- 4x64bit move DRAM:
 - 16nJ

Implicit vs Explicit Data Movement

Examples:

- SYCL, C++ AMP

Implementation:

- Data is moved to the device implicitly via cross host CPU / device data structures

Here we're using C++ AMP as an example

```
array_view<float> ptr;  
extent<2> e(64, 64);  
parallel_for_each(e, [=](index<2> idx)  
restrict(amp) {  
    ptr[idx] *= 2.0f;  
});
```

Examples:

- OpenCL, CUDA, OpenMP

Implementation:

- Data is moved to the device via explicit copy APIs

Here we're using CUDA as an example

```
float *h_a = { ... }, d_a;  
cudaMalloc((void **)&d_a, size);  
cudaMemcpy(d_a, h_a, size,  
            cudaMemcpyHostToDevice);  
vec_add<<<64, 64>>>(a, b, c);  
cudaMemcpy(d_a, h_a, size,  
            cudaMemcpyDeviceToHost);
```

Key takeaways

Task vs Data Parallelism

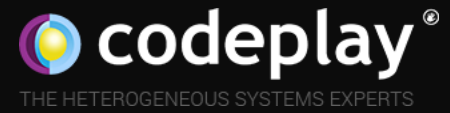
Multicore CPU vs Manycore GPU

Auto vectorisation (implicit SIMD) is not as reliable as explicit SIMD

We need a standard to converge the many SIMD formats

But it still only covers CPUs. There are a lot more SIMD lanes in GPUs.

Understand Characteristics of Heterogeneous programming languages



Questions?