

Untangling Modern Parallel Programming Models

Presenter: Mike Kinsner

Co-authors: Ben Ashbaugh, James Brodman, Greg Lueck, John Pennycook and Roland Schulz



Notices & Disclaimers

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details.
No product or component can be absolutely secure.

Your costs and results may vary.

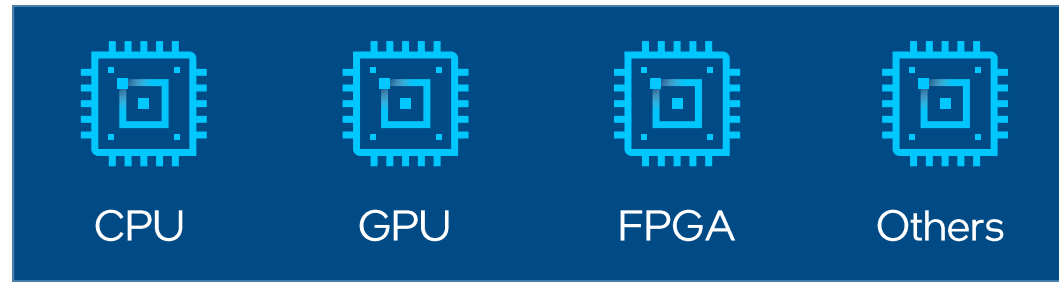
Intel technologies may require enabled hardware, software or service activation.

Intel does not control or audit third-party data. You should consult other sources to evaluate accuracy.

© Intel Corporation. Intel, the Intel logo, Xeon, Core, VTune, OpenVINO, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Khronos and the Khronos Group logo are trademarks of the Khronos Group Inc. in the U.S. and/or other countries. OpenCL and the OpenCL logo are trademarks of Apple Inc. in the U.S. and/or other countries. SYCL and the SYCL logo are trademarks of the Khronos Group Inc. in the U.S. and/or other countries. SPIR and the SPIR logo are trademarks of the Khronos Group Inc. in the U.S. and/or other countries. Other names and brands may be claimed as the property of others.

Motivation

- Today's workloads target a rich diversity of hardware:



- Modern hardware exposes multiple types of parallelism
- Programming across architectures is a challenge
 - Language abstractions aim to ease the burden

Topics

1. Scope
2. Mapping to hardware
 - Programming model
 - Toolchain
3. Languages/frameworks
4. Composing models
5. Takeaways

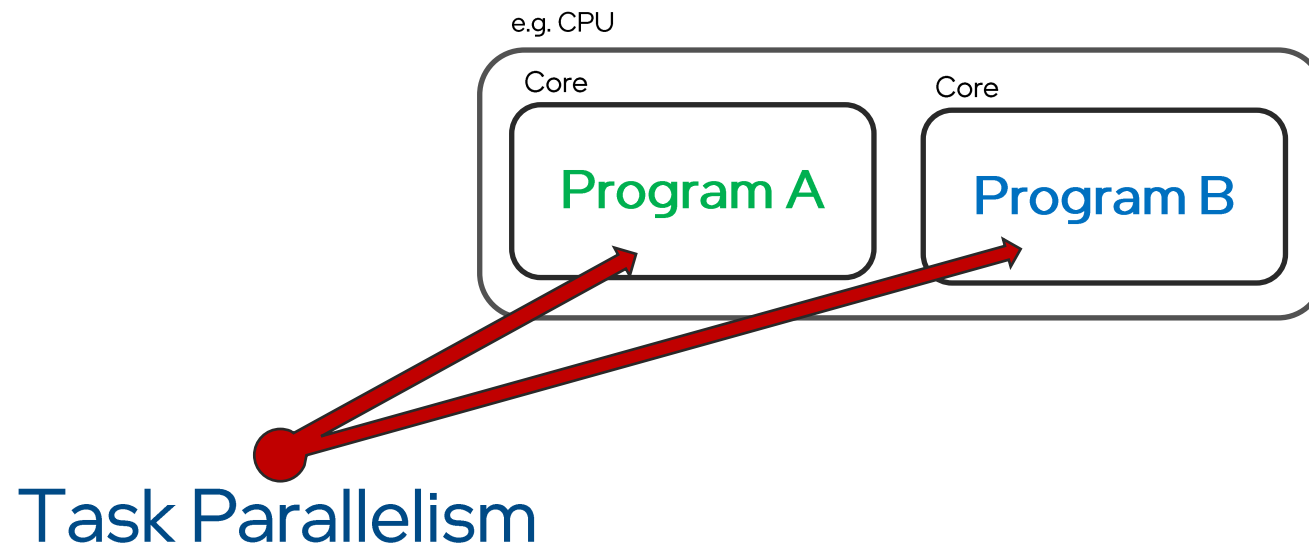
Getting on the same page

Focus of presentation



Broad Breakdown of Parallelism*

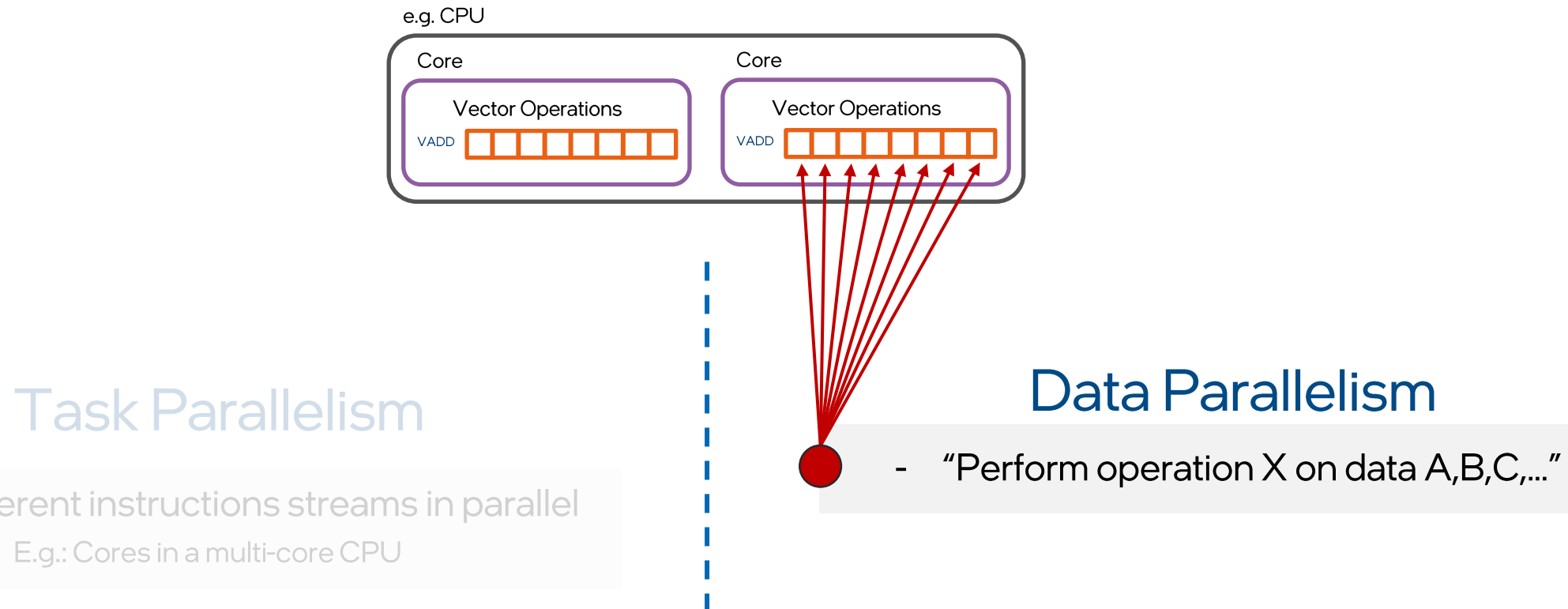
- First, think in terms of streams of instructions (or programs)
 - **Program A** and **Program B** running on different processor cores



- Different instructions streams in parallel
 - E.g.: Cores in a multi-core CPU

Broad Breakdown of Parallelism* (2)

- Then, think of parallelism inside a **single** instruction stream (program)
 - e.g. vector instructions



Broad Breakdown of Parallelism (3)

- Data parallelism includes loop vectorization, pipeline parallelism, others

This talk covers expression of data (not task or other) parallelism



Task Parallelism

- Different instructions streams in parallel
 - E.g.: Cores in a multi-core CPU

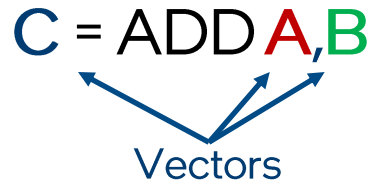
Data Parallelism

- "Perform operation X on data A,B,C,..."

Reminder: Vector hardware instructions

$$\mathbf{C} = \text{ADD } \mathbf{A}, \mathbf{B}$$

Vectors



$C_0 = A_0 + B_0$	$C_1 = A_1 + B_1$	$C_2 = A_2 + B_2$	$C_3 = A_3 + B_3$	$C_4 = A_4 + B_4$	$C_5 = A_5 + B_5$	$C_6 = A_6 + B_6$	$C_7 = A_7 + B_7$
-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------	-------------------

- Vector instructions exist in many throughput-centric architectures today
 - Multiple components / elements in a single operation
 - May appear as vector instructions in assembly/machine code
- **Question:** How do we write programs to leverage data parallelism hardware?

Mapping to hardware

Part 1: The programming model



Using vector hardware

Single Instruction Multiple Data (SIMD):

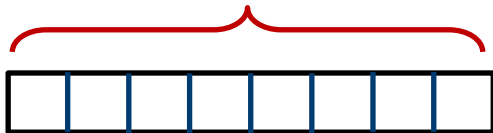
Explicit SIMD

```
// Vector addition  
c8 = a8 + b8;
```

Vector variables, each with
8 elements / components

Often ~direct
mapping

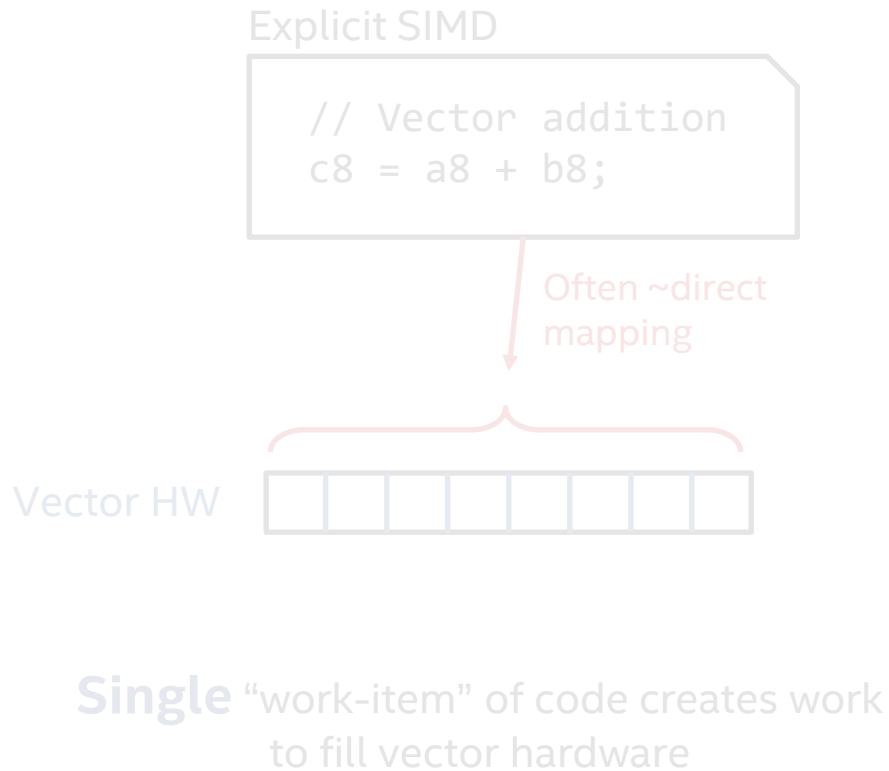
Vector HW



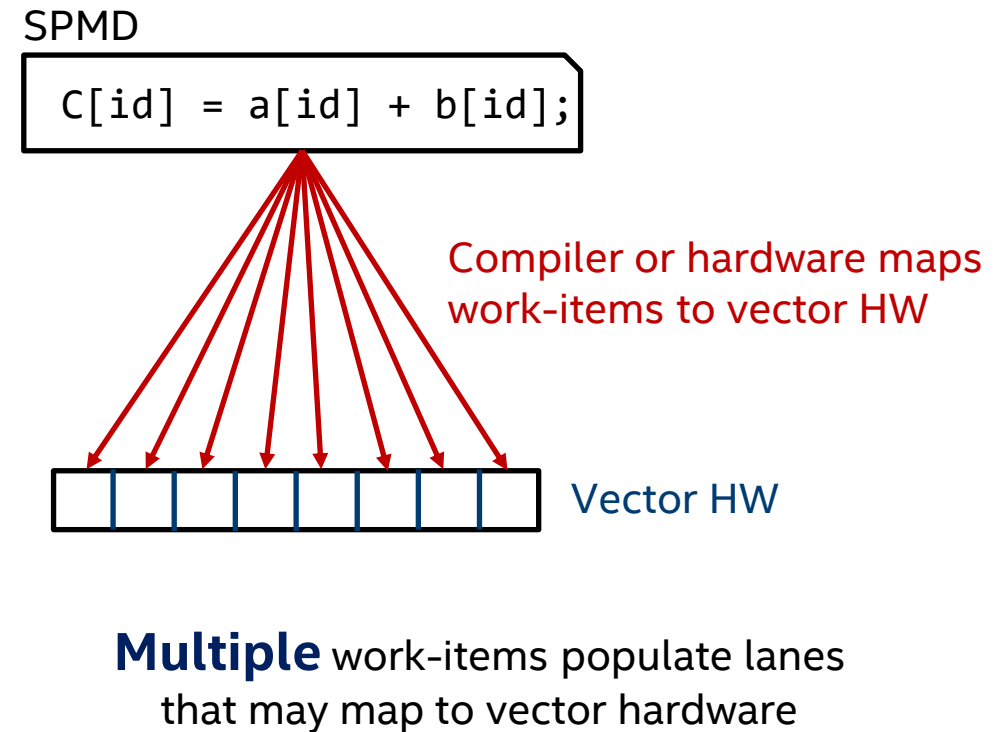
Single “work-item” of code creates work
to fill vector hardware

Using vector hardware (2)

Single Instruction Multiple Data (SIMD):



Single Program Multiple Data (SPMD*):



* Also called SIMT and other names

Using vector hardware (3)

Add in Single Instruction Single Data (SISD)



SISD

```
for (i=0; i<8; i++) {  
    c[i] = a[i] + b[i];  
}
```

Vectorizer

Explicit SIMD

```
// Vector addition  
c8 = a8 + b8;
```

Often ~direct
mapping

Vector HW



Single “work-item” of code creates work to fill vector hardware

Single Program Multiple Data (SPMD):

SPMD

```
C[id] = a[id] + b[id];
```

Compiler or hardware maps
work-items to vector HW



Vector HW

Multiple work-items populate lanes that may map to vector hardware

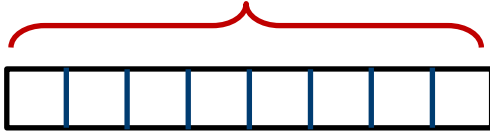
Common ways to access vector hardware

SISD

```
for (i=0; i<8; i++) {  
    c[i] = a[i] + b[i];  
}
```

Vectorizer

Vector HW



Explicit SIMD

```
// Vector addition  
c8 = a8 + b8;
```

Often ~direct mapping

Explicit SIMD:

- Direct control over hardware
- Often 1:1 mapping
- Direct cross-lane operations

SPMD

```
C[id] = a[id] + b[id];
```

Compiler or hardware maps work-items to vector HW



Vector HW

SPMD:

- More portable across architectures / generations
- Easier to reason about how to write correct code
- Details like masking handled by toolchain

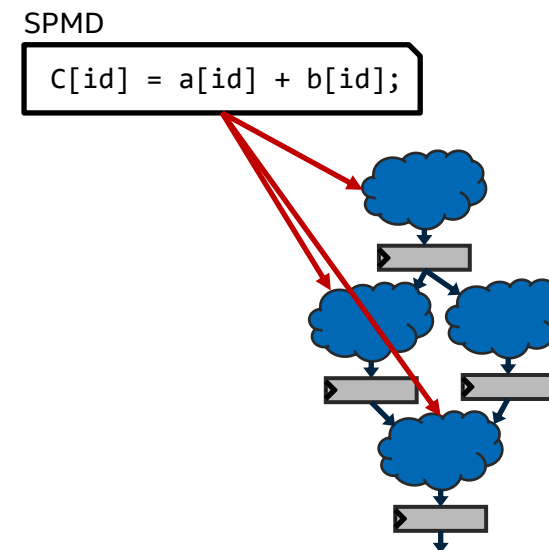
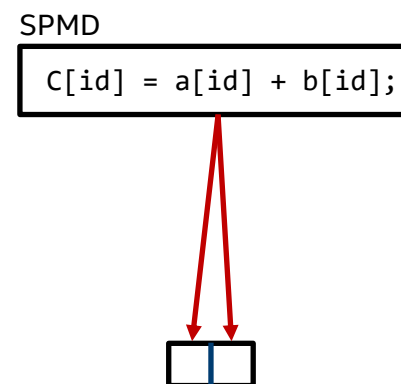
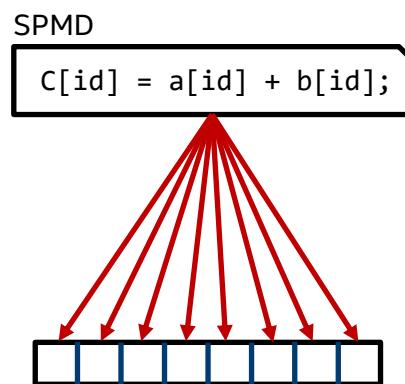
SISD:

- Easy for sequential coders
- Automaton or directives fit in well

Takeaway #1

SPMD code is often more portable across architectures/generations, and for many is an easier mental model (independent work-items)

SPMD even maps cleanly to pipeline parallelism (e.g. FPGA)

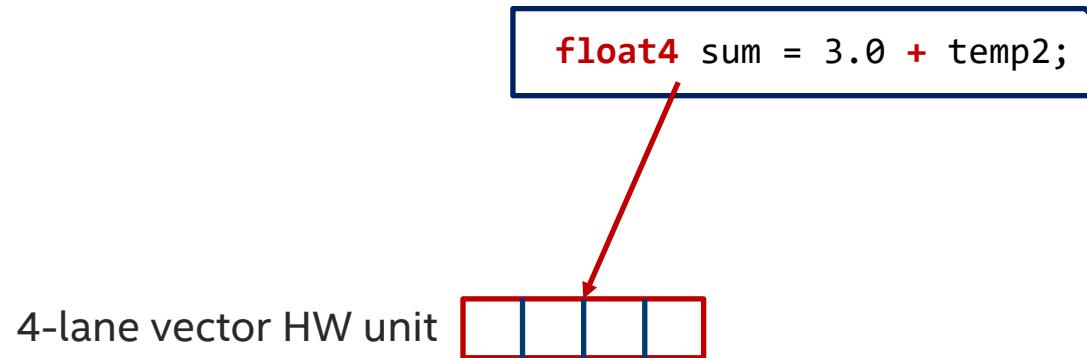


Mapping to hardware

Part 2: The toolchain's mapping



Two competing vector mapping options

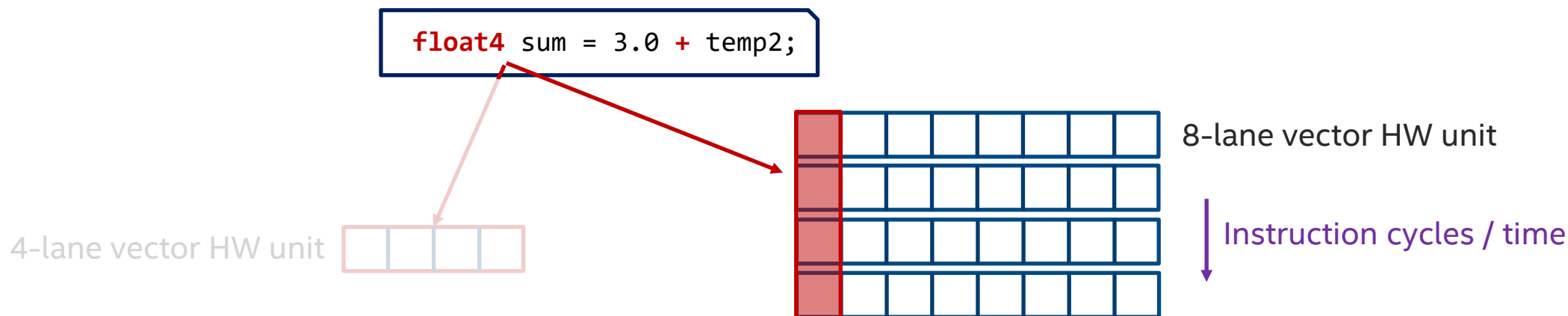


An option: Vector in code maps to SIMD hardware

- Asked for by expert developers targeting specific HW
- Like assembly/intrinsics in higher level language
- Sometimes called “explicit SIMD”

Does not integrate trivially with multi-work-item
SPMD ND-range model 😞

Two competing vector mapping options (2)



An option: Vector in code maps to SIMD hardware

- Asked for by expert developers targeting specific HW
- Like assembly in higher level language
- Sometimes called "explicit SIMD"

Does not integrate trivially with multi-work-item SPMD ND-range model 😞

An option: Vector in code maps to single lane over "time"

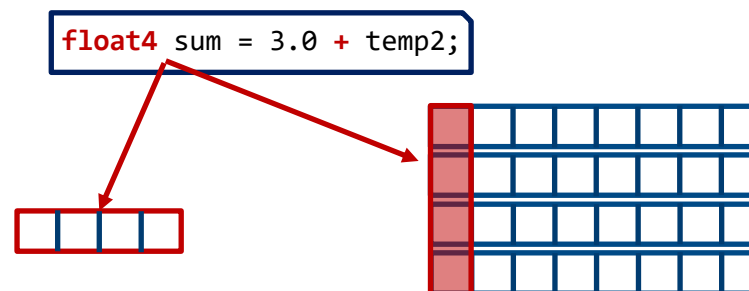
- Not tied to specific hardware
 - Potentially portable to all hardware
 - Ordering between work-items not specified = freedom
 - Compiler deals with masking for inactive lanes

Sub-groups in OpenCL/SYCL enable the other (horizontal) view for performance tuning/reasoning 😎

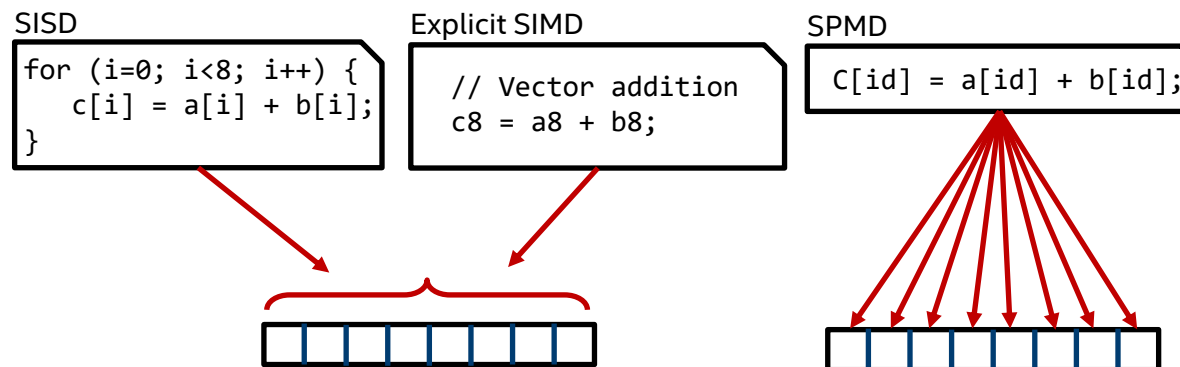
Takeaway #2

Languages/standards often don't define mappings to hardware (intentionally). To achieve performance, need to understand how the tools interpret code.

Tools decide what a vector means

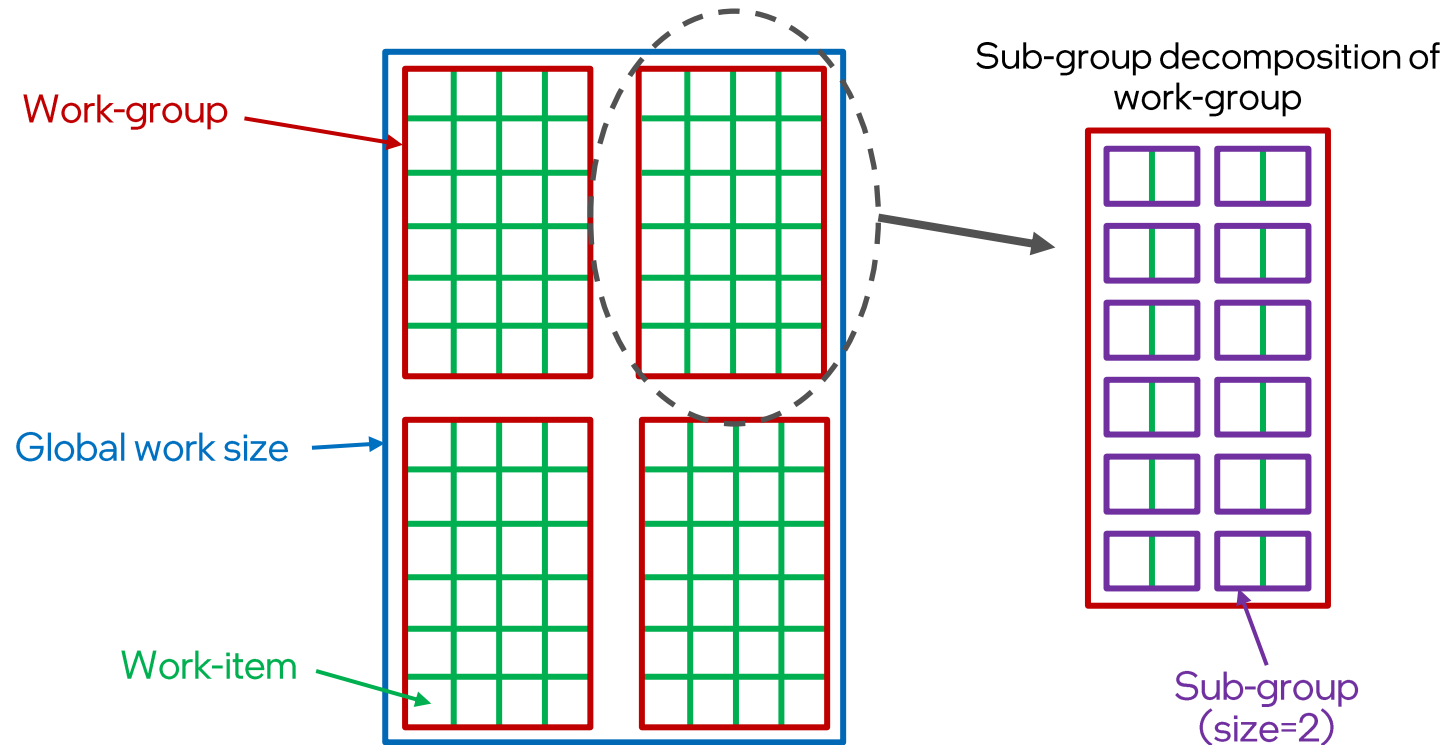


Even the difference between SPMD and explicit SIMD is an interpretation detail



Sub-groups in SYCL / OpenCL

- Pure SPMD abstraction missing expressibility around cross-lane hardware
 - Work-items in isolated lanes by design
 - SGs map to what **can** be vector hardware – enables cross-lane collective operations



SYCL 2020 sub-group algorithms

```
sycl::sub_group sg = it.get_sub_group();  
  
auto a = sycl::shift_group_left(sg, x, 1);  
auto b = sycl::shift_group_right(sg, x, 1);  
auto c = sycl::select_from_group(sg, x, id);  
auto d = sycl::permute_group_by_xor(sg, x, mask);
```

Some common programming
languages/frameworks



Languages/frameworks vary in model and mapping

	Common Model	Interpretation of vector types	Cross-lane abstraction	Cross-lane communication
SYCL	SPMD	<ul style="list-style-type: none">• <code>sycl::marray</code> for intra work-item• Future <code>std::simd</code> alignment for SIMD	Sub-group	Group algorithms
OpenCL	SPMD	Not spec mandated. Implementers interpret	Sub-group	Sub-group functions
CUDA	SPMD (SIMT)	Intra work-item*	Warp	Cooperative group, intrinsics
OpenMP	SISD + SPMD	No vector types defined in specification	Implementation defined	N/A
Kokkos	SPMD	Math convenience type, SIMD math wrappers	Nested parallelism	Algorithms

* Some explicit SIMD ops with data type re-interpretation

Composing programming models



Experimental: Stepping across execution models

- Occasionally, experts want to code directly to an architecture (small % of code)
 - More control over compiler, working around bugs, or use of new hardware features
- Working on proving ground extension to compose programming models
 - Composition of models with clear boundaries / semantics / type safety
 - Coherently incorporates concepts from other programming models (e.g. ISPC, OpenMP)



Enabling composition: “Invoke SIMD” extension

```
namespace ex = sycl::ext::oneapi::experimental;

ex::simd<float, 8> scale(ex::simd<float, 8> x, float n) {
    return x * n;
}

q.parallel_for(..., sycl::nd_item<1> it) [[sycl::reqd_sub_group_size(8)]] {
    sycl::sub_group sg = it.get_sub_group();
    float x = ...;
    float n = ...;

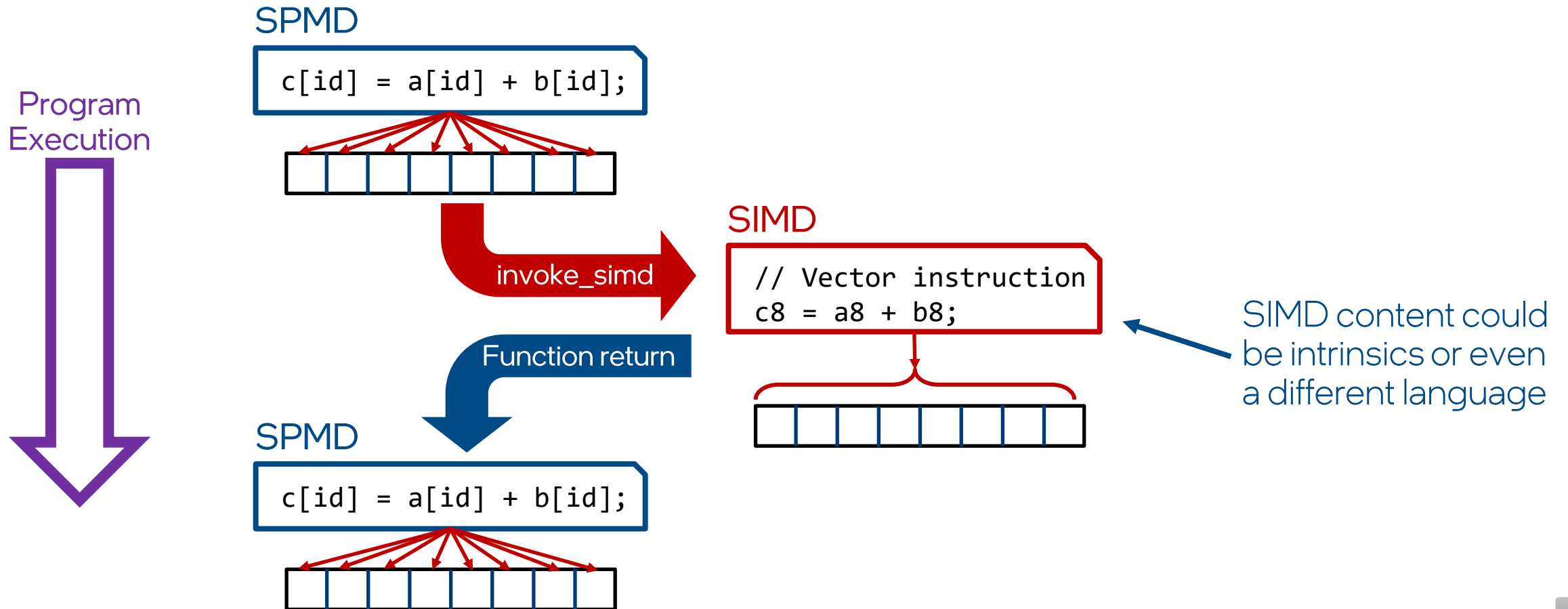
    // Invoke SIMD function - x values from each work-item are combined into a simd<float, 8>
    float y = ex::invoke_simd(sg, scale, x, ex::uniform(n));
});
```

In callee, switch from SPMD to explicit SIMD execution context (SIMD with respect to sub-group `sg`)

https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/proposed/sycl_ext_oneapi_invoke_simd.asciidoc

Enabling composition: “Invoke SIMD” extension (2)

```
// Invoke SIMD function - x values from each work-item combined into simd<float, 8>  
float y = ex::invoke_simd(sg, scale, x, ex::uniform(n));
```

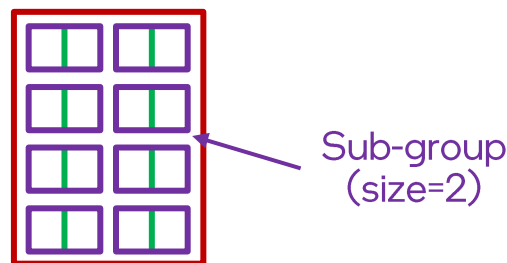


Takeaway #3

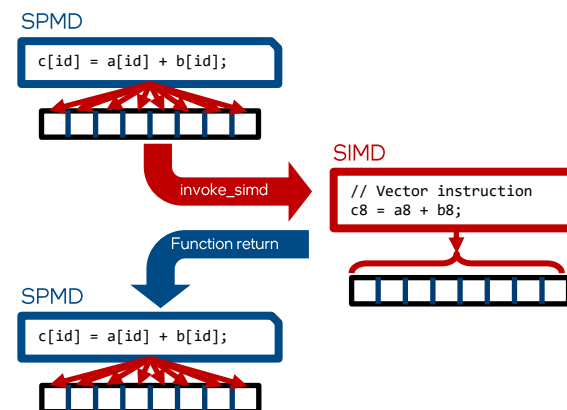
Pure models can have gaps in portability or performance.
Step outside the model in consistent, well-defined ways.

Sub-groups add concept of ganged lanes and collective operations to SPMD model

Sub-group decomposition of work-group



Invoke_simd extension enables coding directly to architecture in small % of code, when needed

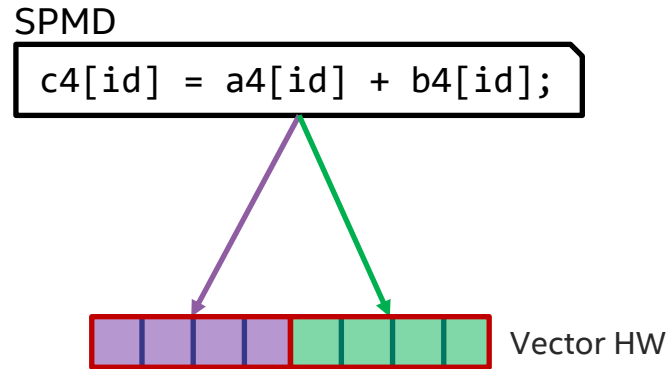


Non-scalar lanes and recap

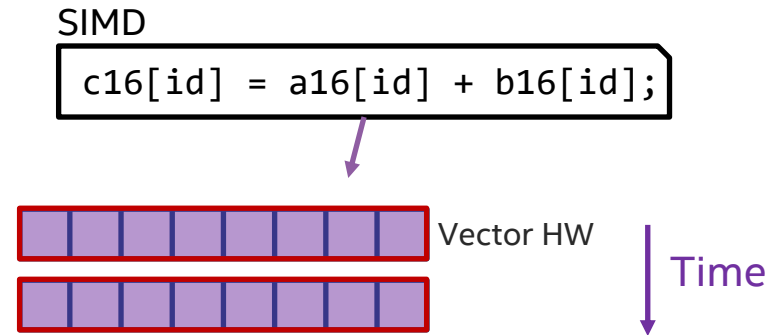


Reality: Vector “lanes” are not always scalar

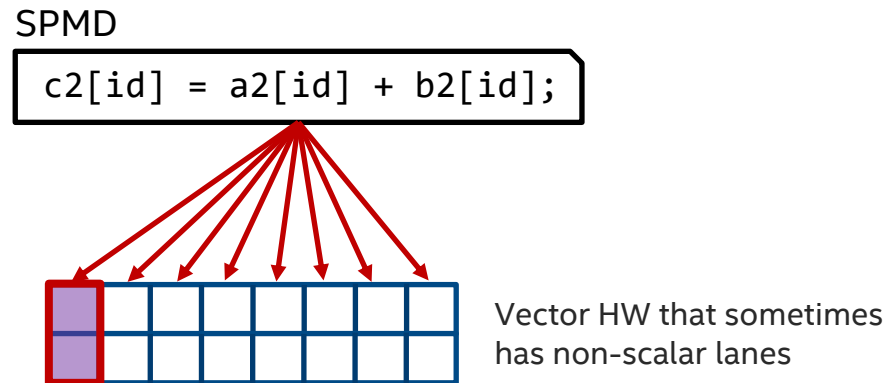
Hybrid: Work-item uses >1 vector lane



Hybrid: Explicit vector width > #vector lanes

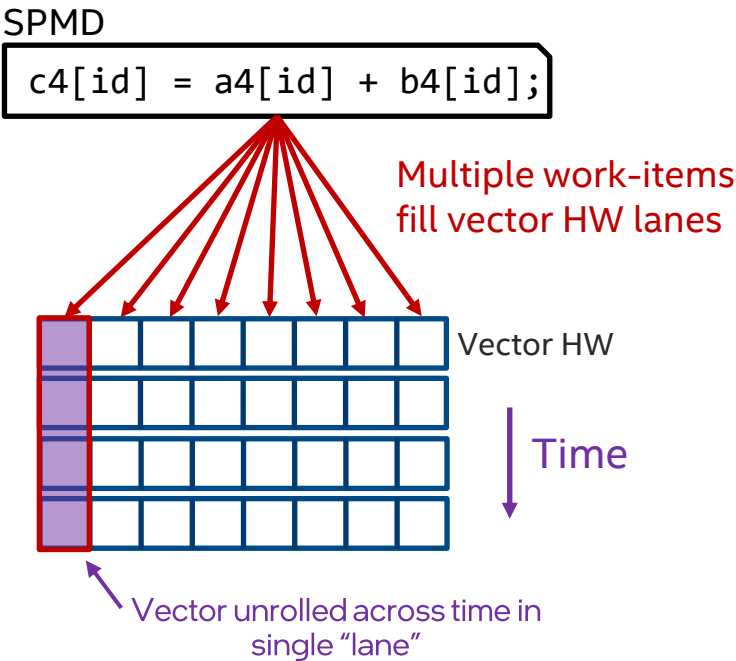


Special case: Some instructions do vector operations in a “lane”



Recap – key mappings to hardware

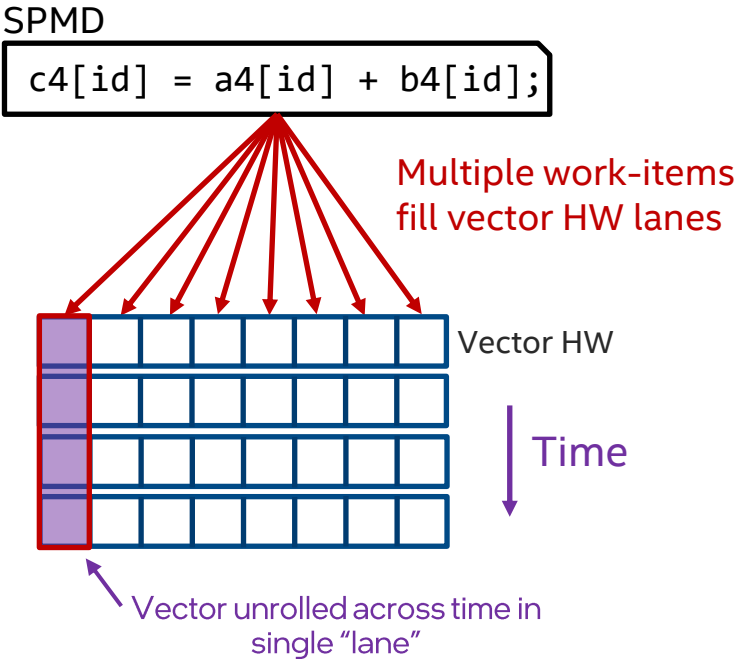
Style 1: SPMD mapping



marray added to SYCL 2020 for this!

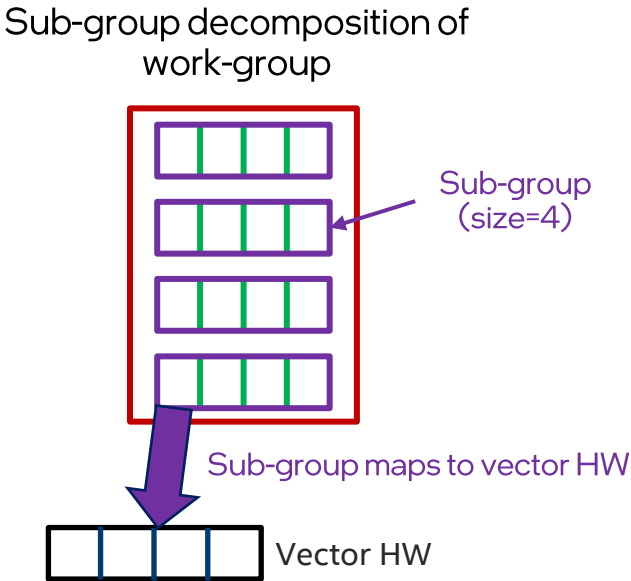
Recap – key mappings to hardware (2)

Style 1: SPMD mapping



marray added to SYCL 2020 for this!

Style 2: SPMD with sub-groups

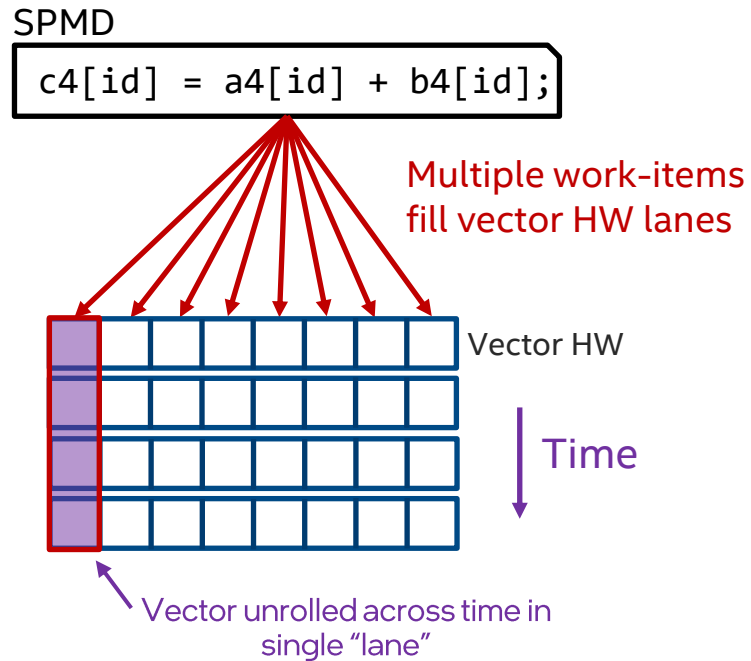


Sub-group represents SIMD width, gives representation for collective operations

This is why SG exist in SYCL 2020

Recap – key mappings to hardware (3)

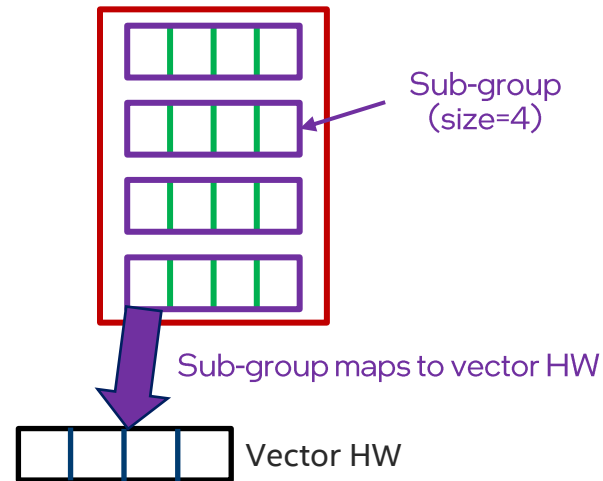
Style 1: SPMD mapping



marray added to SYCL 2020 for this!

Style 2: SPMD with sub-groups

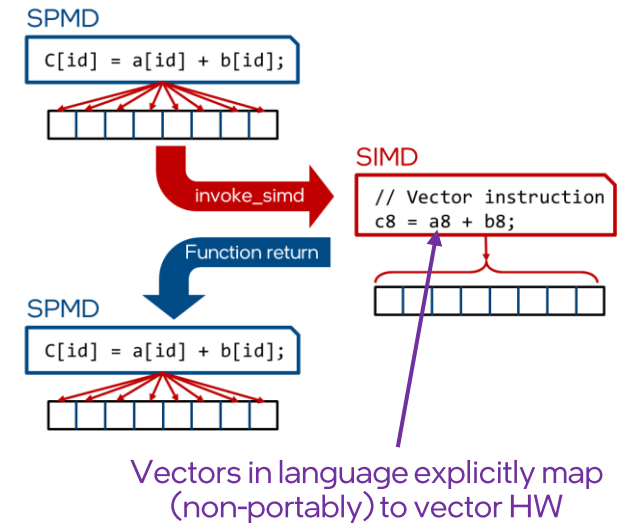
Sub-group decomposition of work-group



Sub-group represents SIMD width, gives representation for collective operations

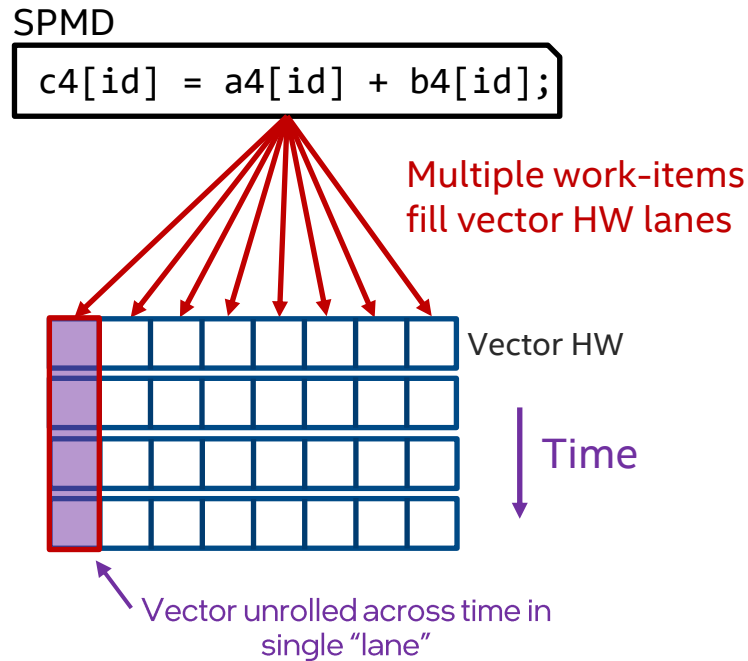
This is why SG exist in SYCL 2020

Style 3a: SPMD with explicit SIMD coding in special code regions



Recap – key mappings to hardware (4)

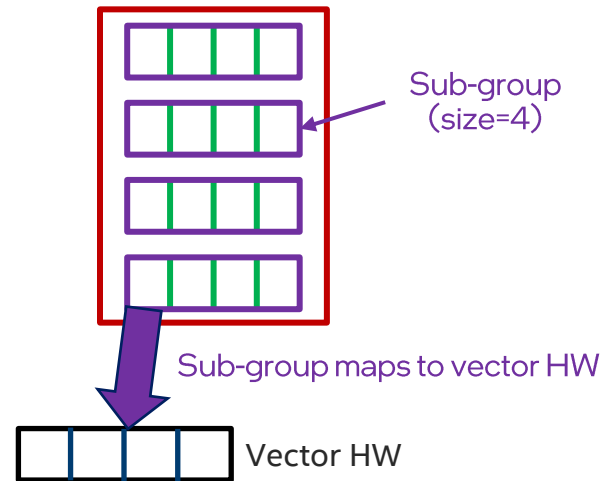
Style 1: SPMD mapping



marray added to SYCL 2020 for this!

Style 2: SPMD with sub-groups

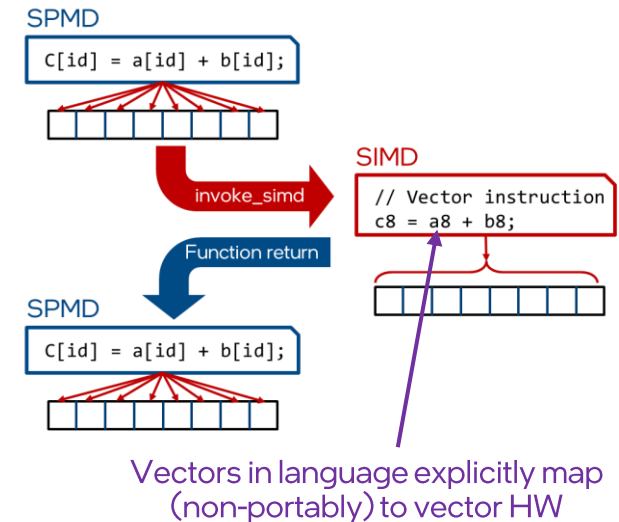
Sub-group decomposition of work-group



Sub-group represents SIMD width, gives representation for collective operations

This is why SG exist in SYCL 2020

Style 3a: SPMD with explicit SIMD coding in special code regions



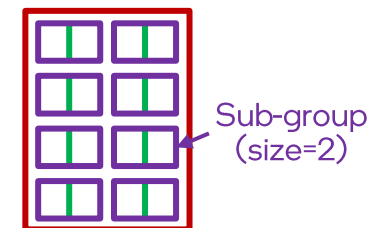
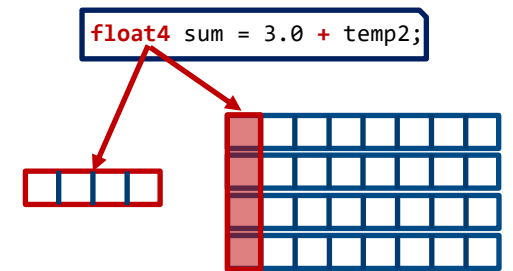
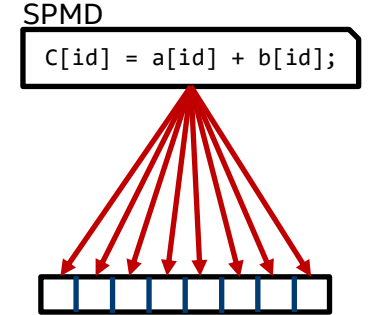
Style 3b: Full kernel explicit SIMD

SIMD

```
my_explicit_SIMD_kernel (...) {  
    ...  
    c8 = a8 + b8; }  
}
```

Takeaways

1. **SPMD**: Improves portability + an easier mental model
2. **Performance** =
Programming model + Toolchain interpretation
3. **Pure models** create gaps in portability or performance
 - Must occasionally step outside model. Do so in a consistent, well-defined framework



 intel[®]

