codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

GPU programming in C++ with SYCL

Gordon Brown
Principal Software Engineer, SYCL & C++

C++ Europe 2020 – June 2020

Agenda

Why use the GPU?

Brief introduction to SYCL

SYCL programming model

Optimising GPU programs

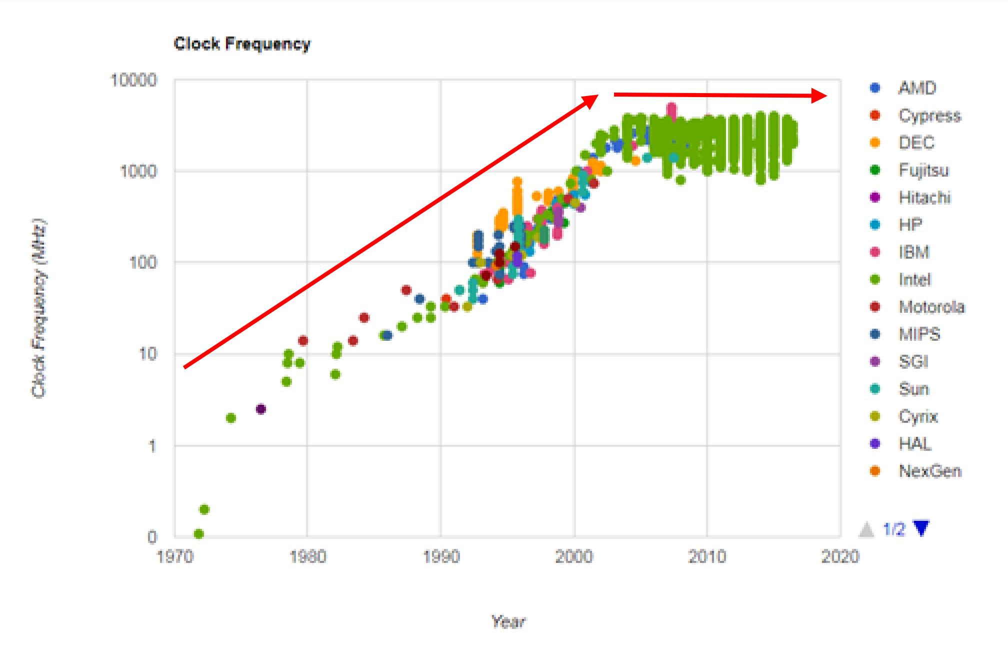
SYCL for Nvidia GPUs

SYCL 2020 preview

Why use the GPU?

“The end of Moore’s Law”

“The free lunch is over”

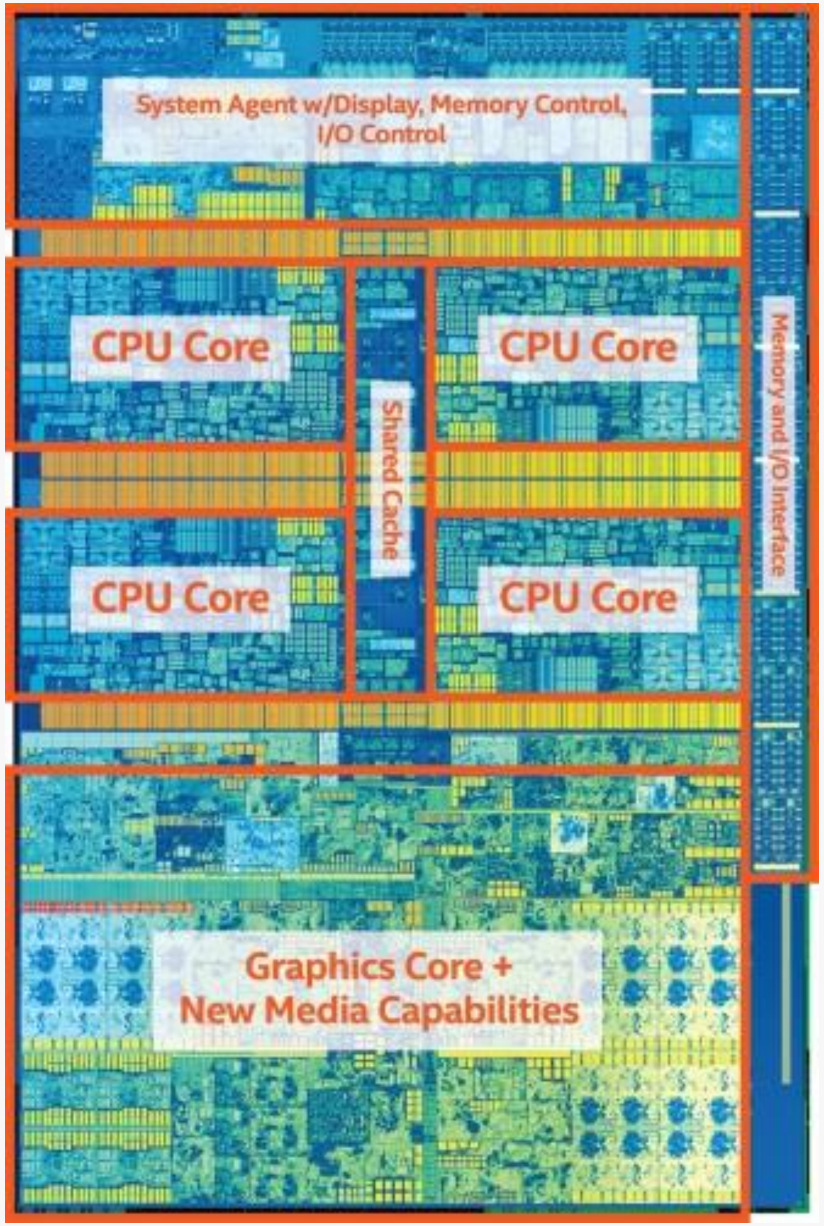


“The future is parallel”

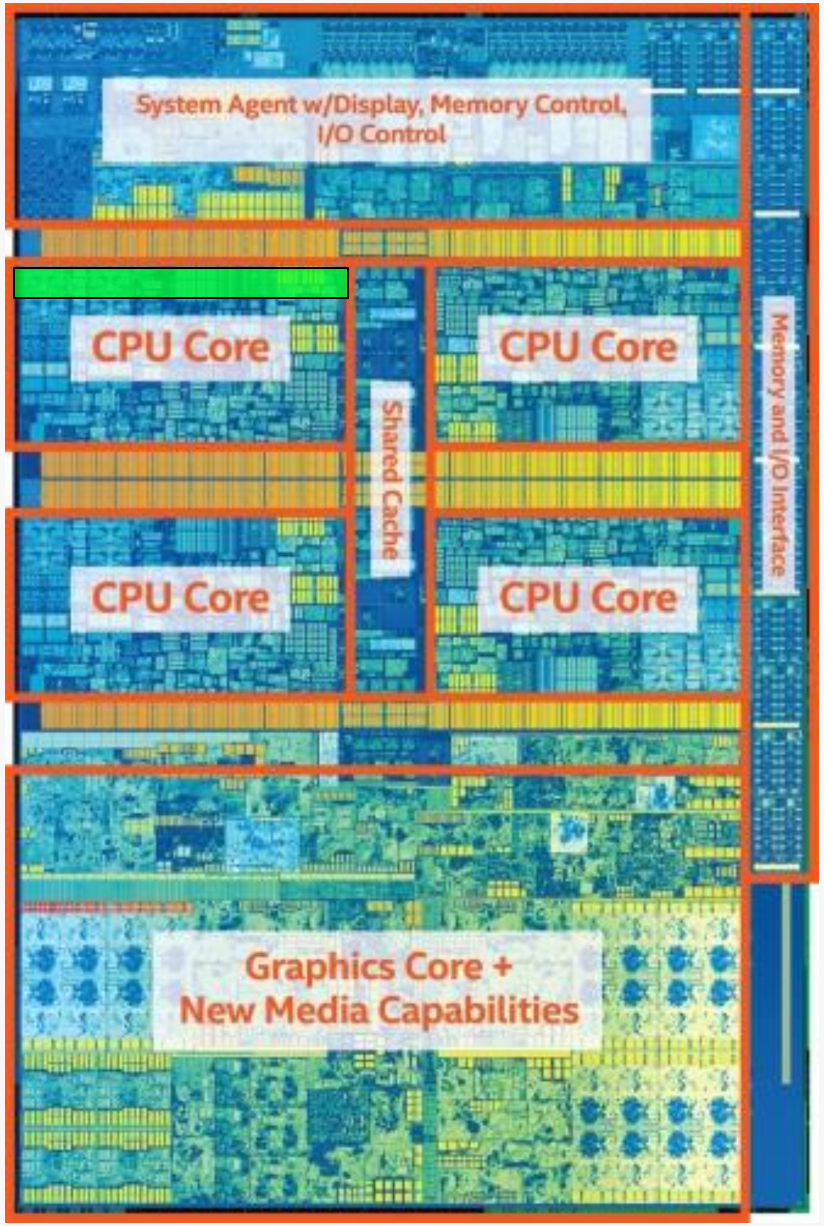
Take a typical Intel chip

Intel Core i7 7th Gen

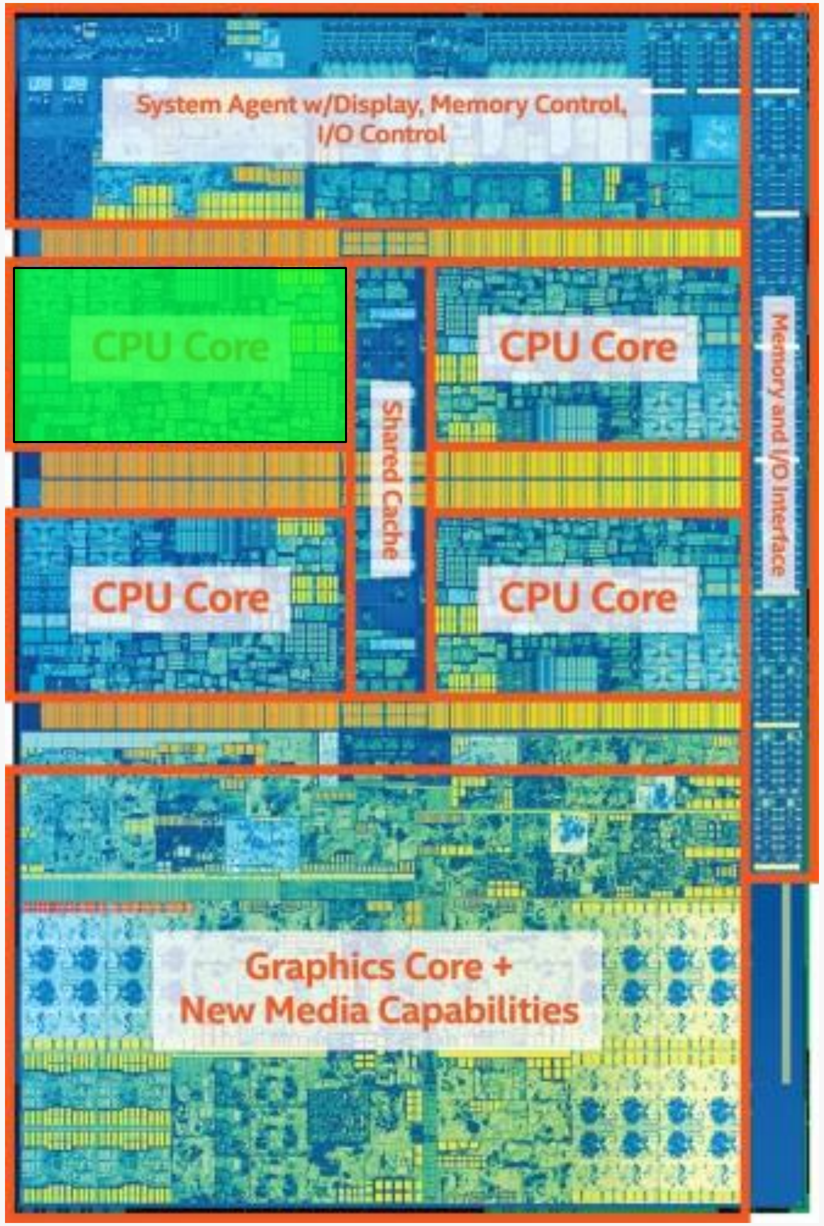
- 4x CPU cores
  - Each with hyperthreading
  - Each with support for 256bit AVX2 instructions
- Intel Gen 9 GPU
  - With 1280 processing elements



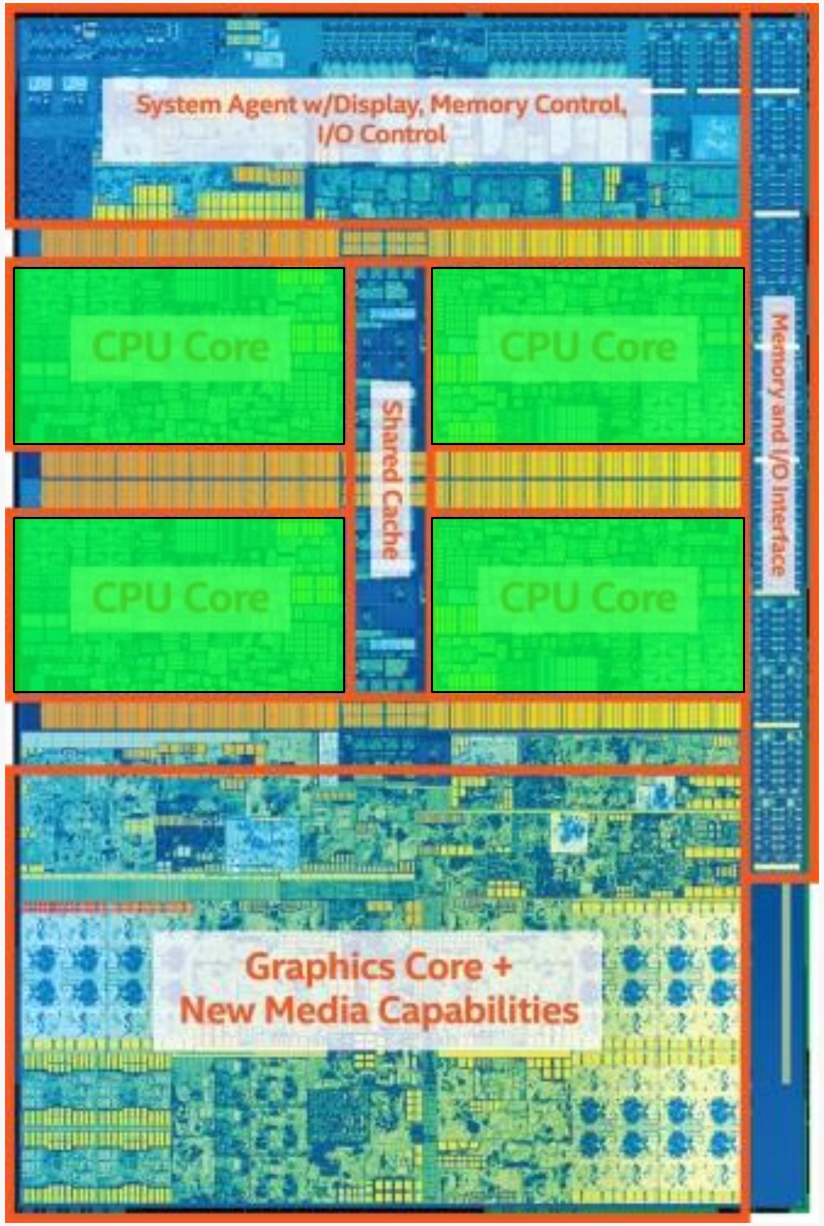
Regular sequential C++ code (non-vectorised) running on a single thread only takes advantage of a very small amount of the available resources of the chip



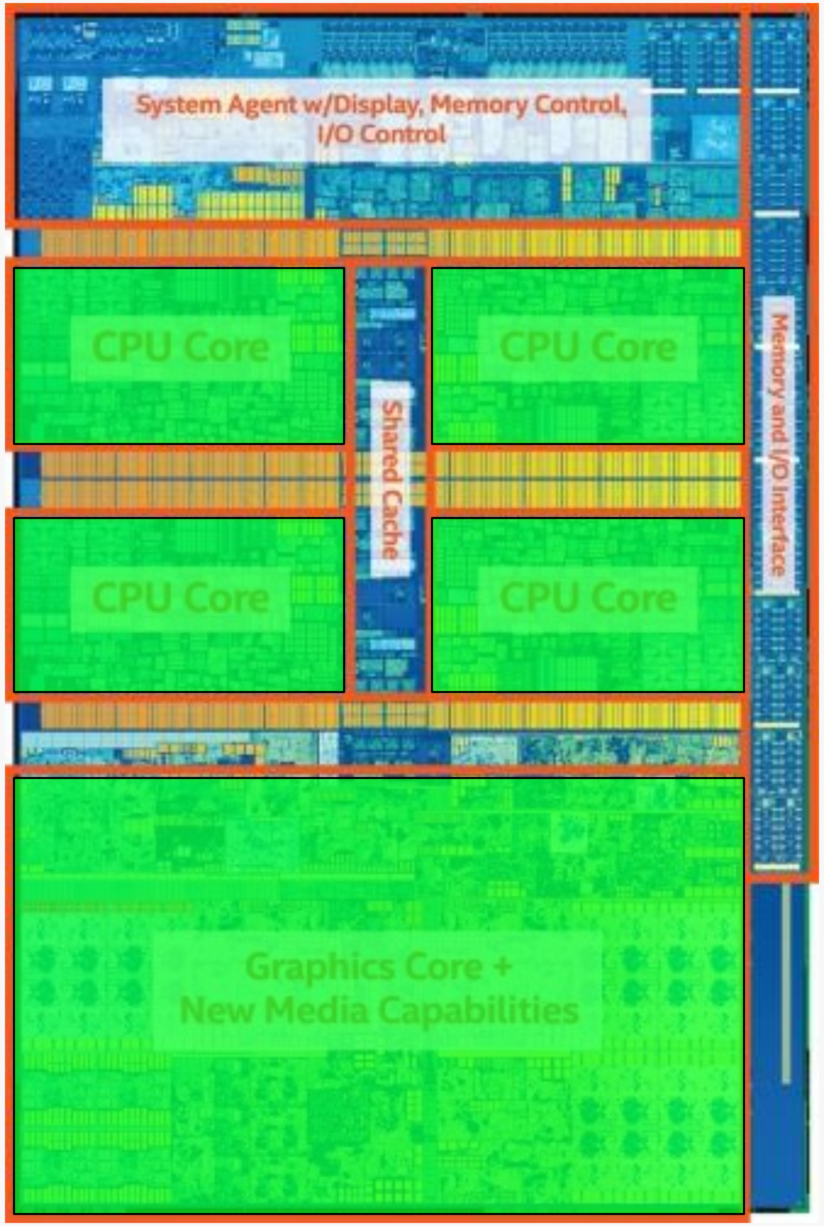
Vectorisation allows you to fully utilise a single CPU core



Multi-threading allows you to fully utilise all CPU cores



Heterogeneous dispatch allows you to fully utilise the entire chip



GPGPU programming was once a niche technology

- Limited to specific domain
- Separate source solutions
- Verbose low-level APIs
- Very steep learning curve

C++AMP

SYCL

CUDA Agency

Kokkos

HPX

Raja

This is not the case anymore

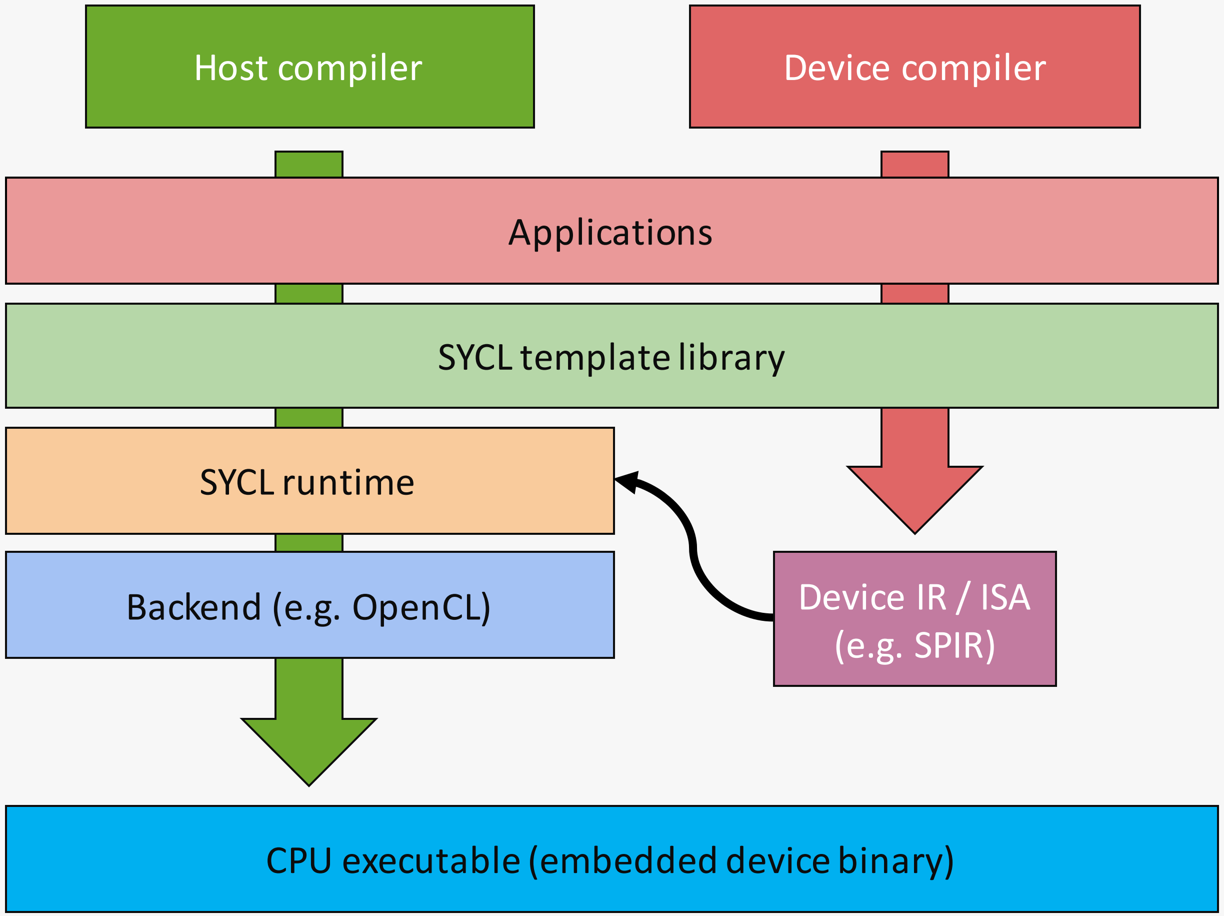
- Almost everything has a GPU now
- Single source solutions
- Modern C++ programming models
- More accessible to the average C++ developer

Brief introduction to SYCL

SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

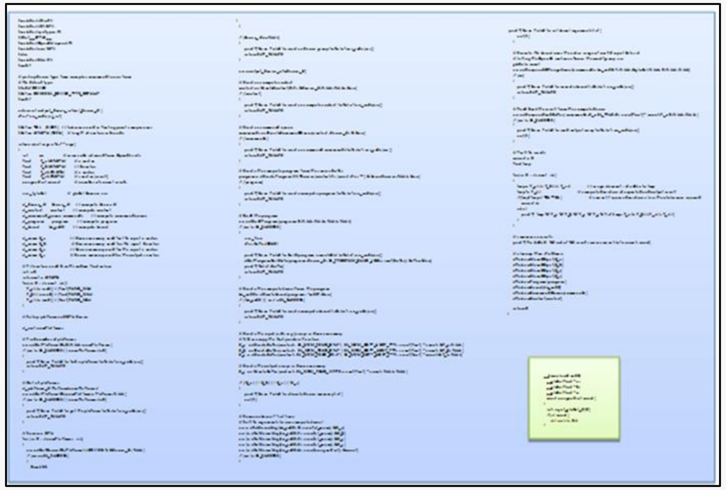
SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms



- SYCL allows you write both host CPU and device code in the same C++ source file
  - This requires two compilation passes; one for the host code and one for the device code

SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms



Typical OpenCL hello world application

Typical SYCL hello world application

- SYCL provides high-level abstractions over common boiler-plate code
  - Platform/device selection
  - Buffer creation
  - Kernel compilation
  - Dependency management and scheduling

SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

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

<2> idx) restrict(amp) {
```

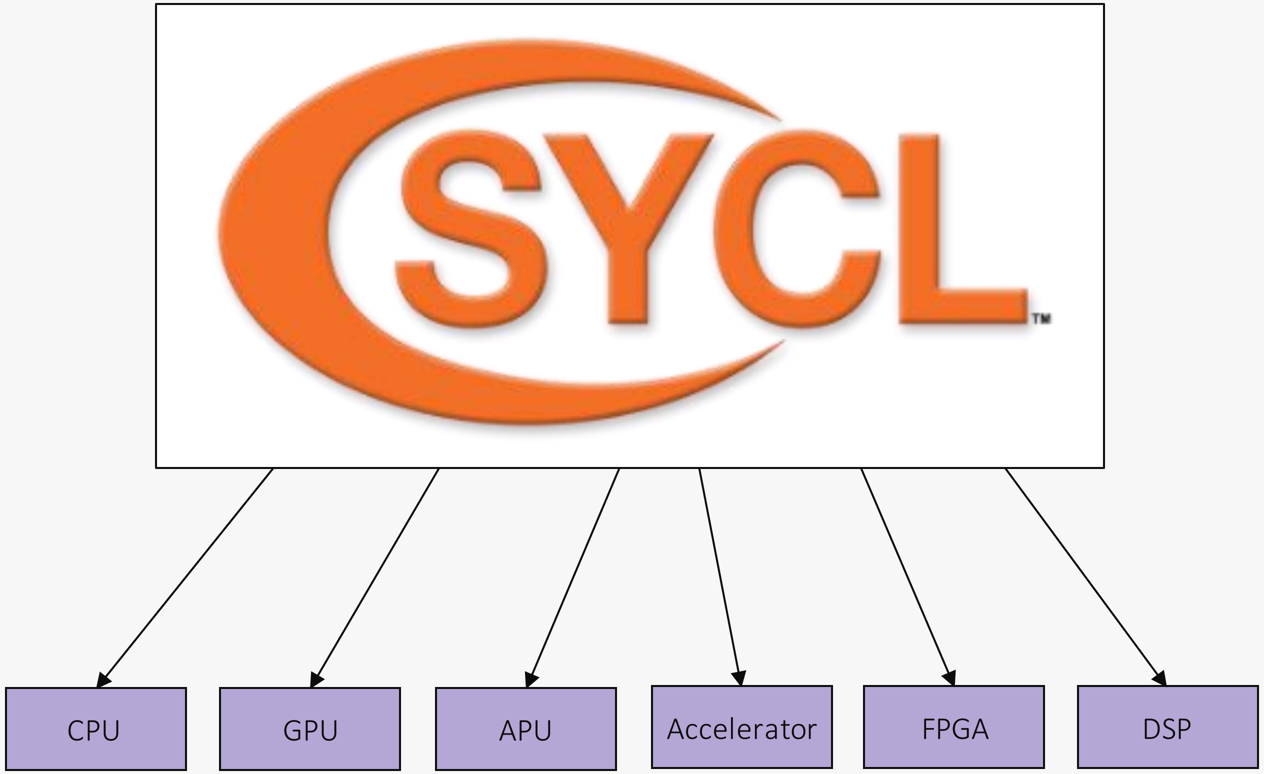
```
std::vector<float> a, b, c;

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

```
__global__ vec_add(float *a, float *b, float *c) {
  return c[i] = a[i] + b[i];
}

float *a, *b, *c;
vec_add<<<range>>>(a, b, c);
```

```
cgh.parallel_for<class vec_add>(range, [=](cl::sycl::id<2> idx) {
  c[idx] = a[idx] + c[idx];
}));
```

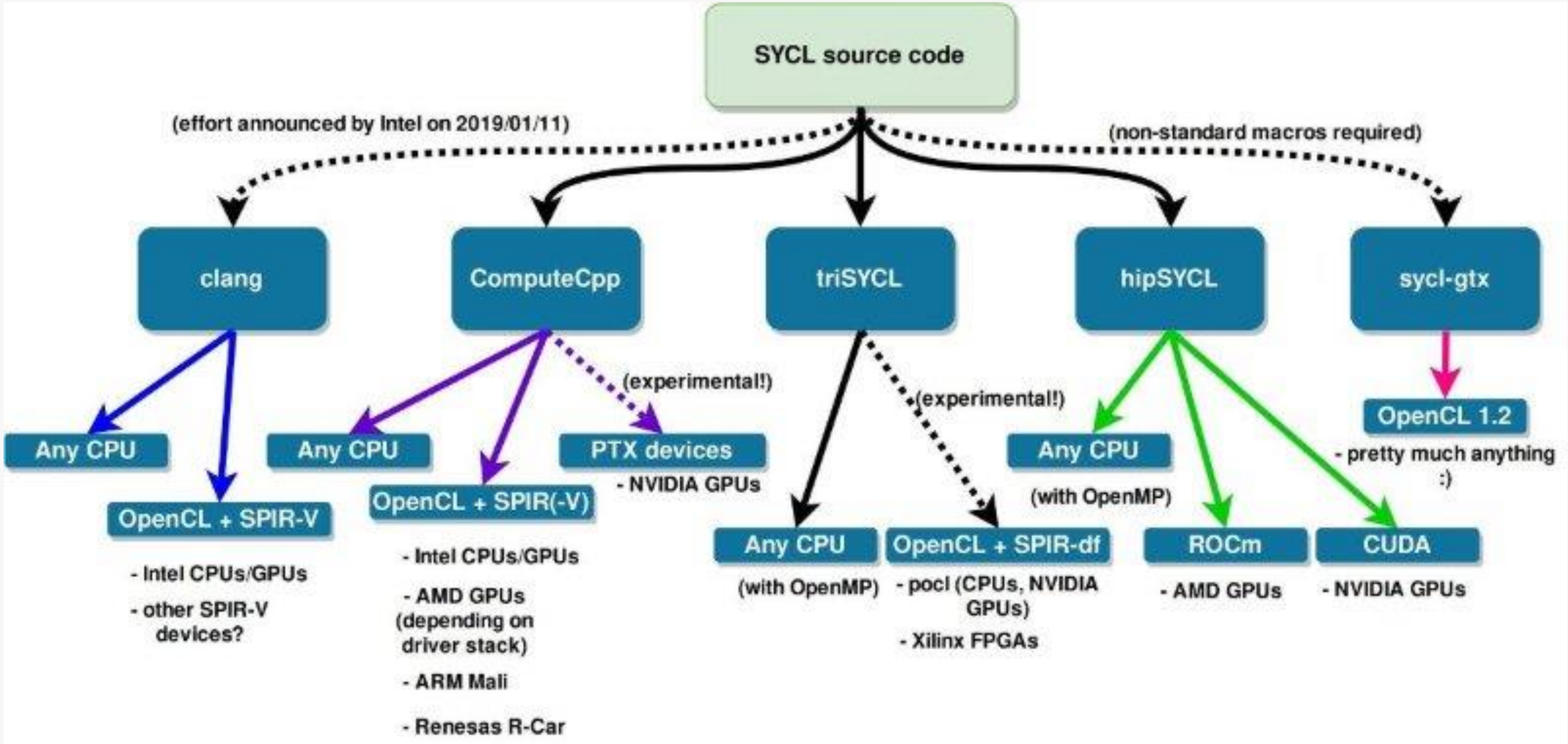
- SYCL allows you to write standard C++
  - No language extensions
  - No pragmas
  - No attributes

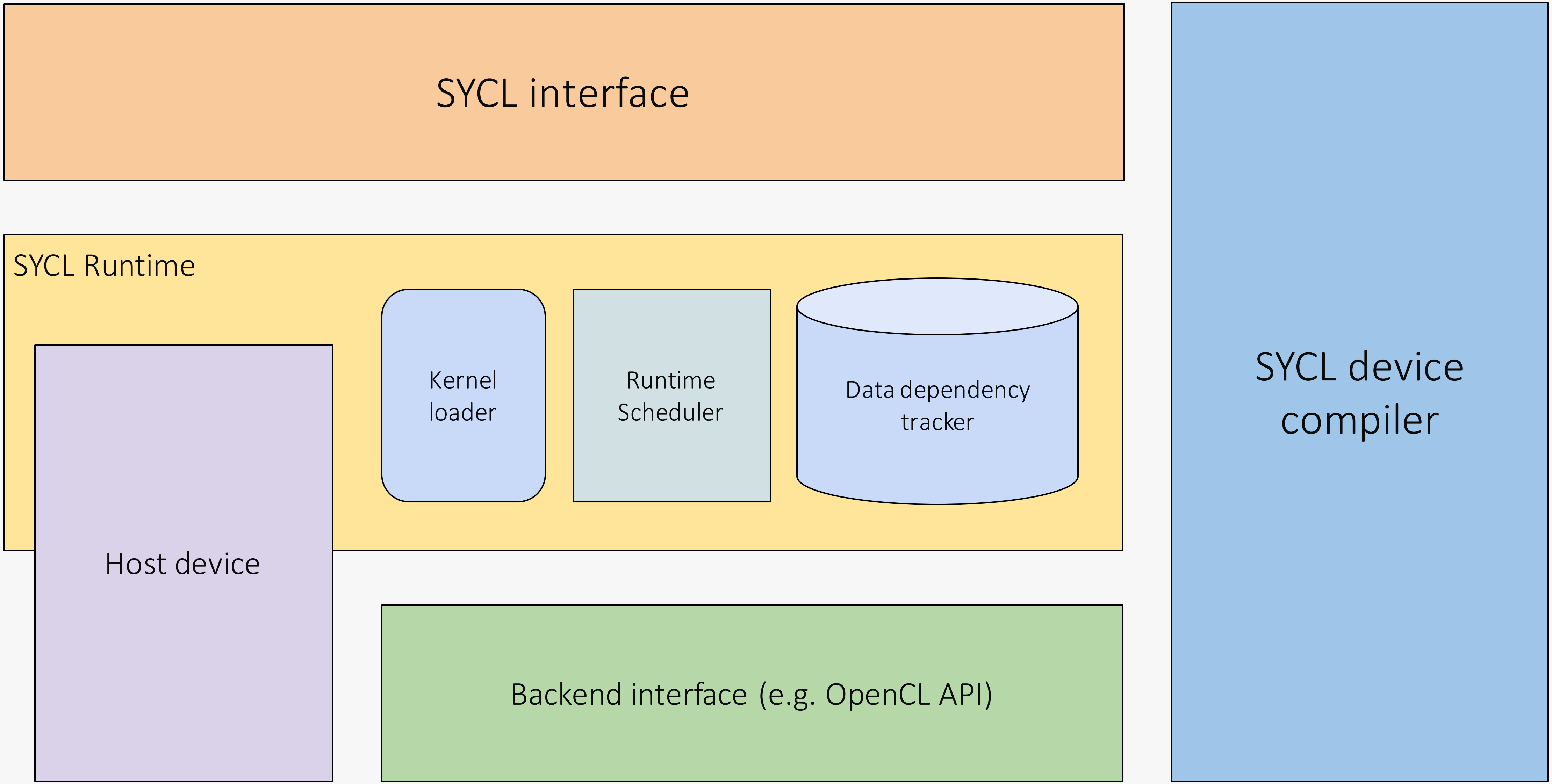
SYCL is a single-source, high-level, standard C++ programming model, that can target a range of heterogeneous platforms

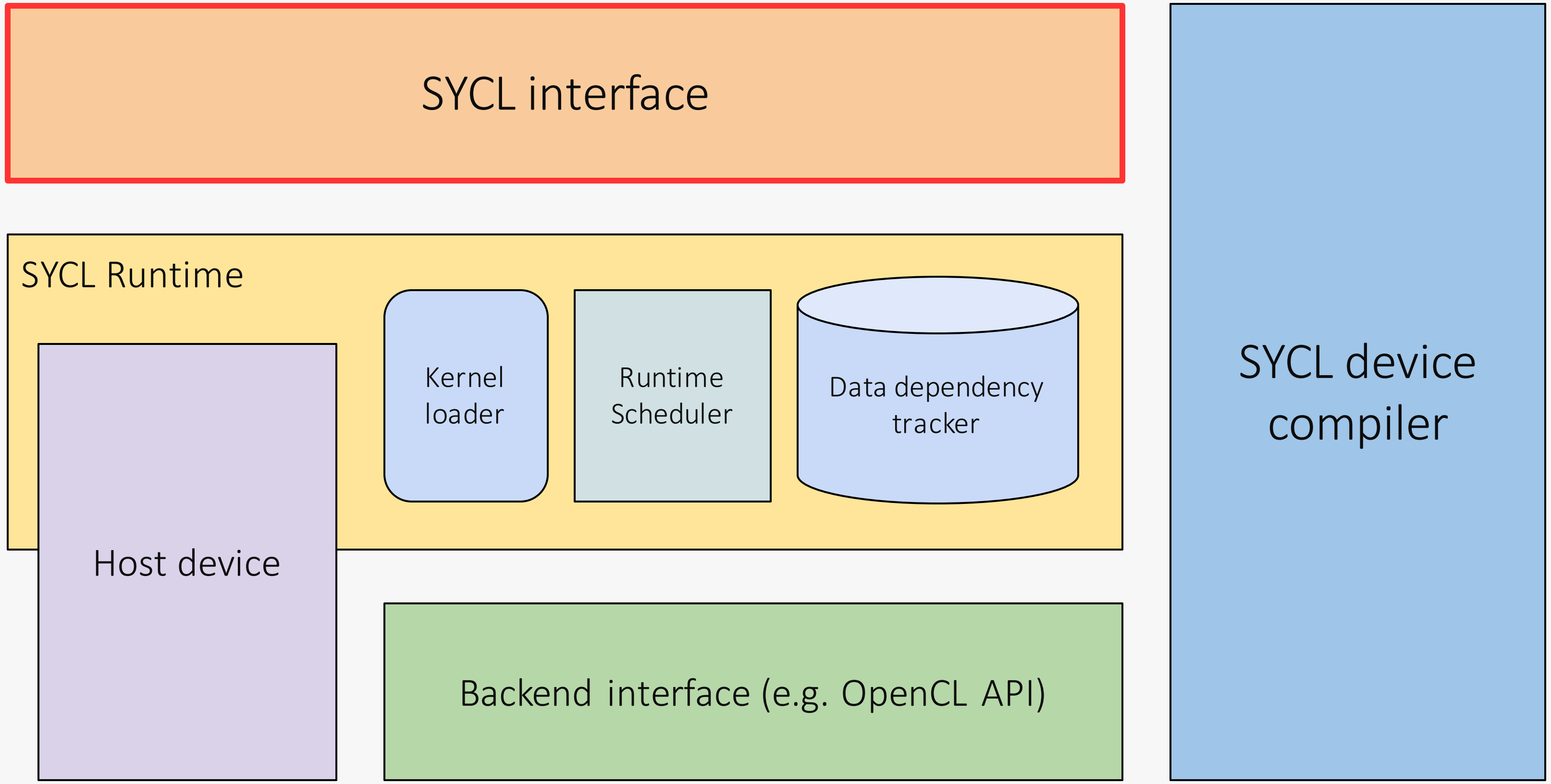


- SYCL can target any device supported by its backend
- SYCL can target a number of different backends
  - Currently the specification is limited to OpenCL
  - Some implementations support other non-standard backends

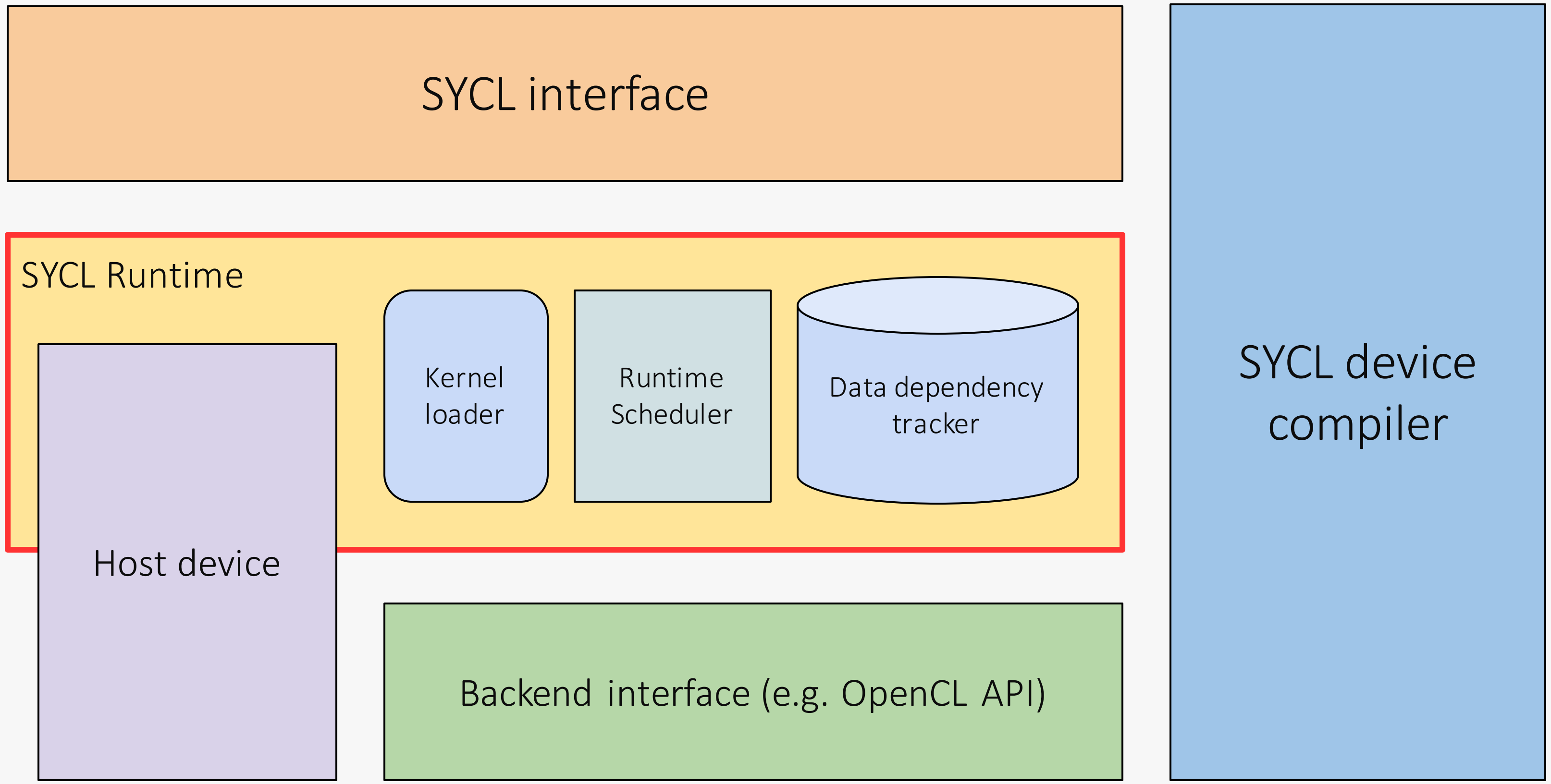
SYCL implementations


SYCL source code
(effort announced by Intel on 2019/01/11)
(non-standard macros required)
clang
ComputeCpp
triSYCL
hipSYCL
sycl-gtx
Any CPU
OpenCL + SPIR-V
- Intel CPUs/GPUs
- other SPIR-V devices?
Any CPU
OpenCL + SPIR(-V)
- Intel CPUs/GPUs
- AMD GPUs (depending on driver stack)
- ARM Mali
- Renesas R-Car
(experimental!)
PTX devices
- NVIDIA GPUs
Any CPU
(with OpenMP)
(experimental!)
OpenCL + SPIR-df
- pocl (CPUs, NVIDIA GPUs)
- Xilinx FPGAs
Any CPU
(with OpenMP)
ROCm
- AMD GPUs
CUDA
- NVIDIA GPUs
OpenCL 1.2
- pretty much anything :)

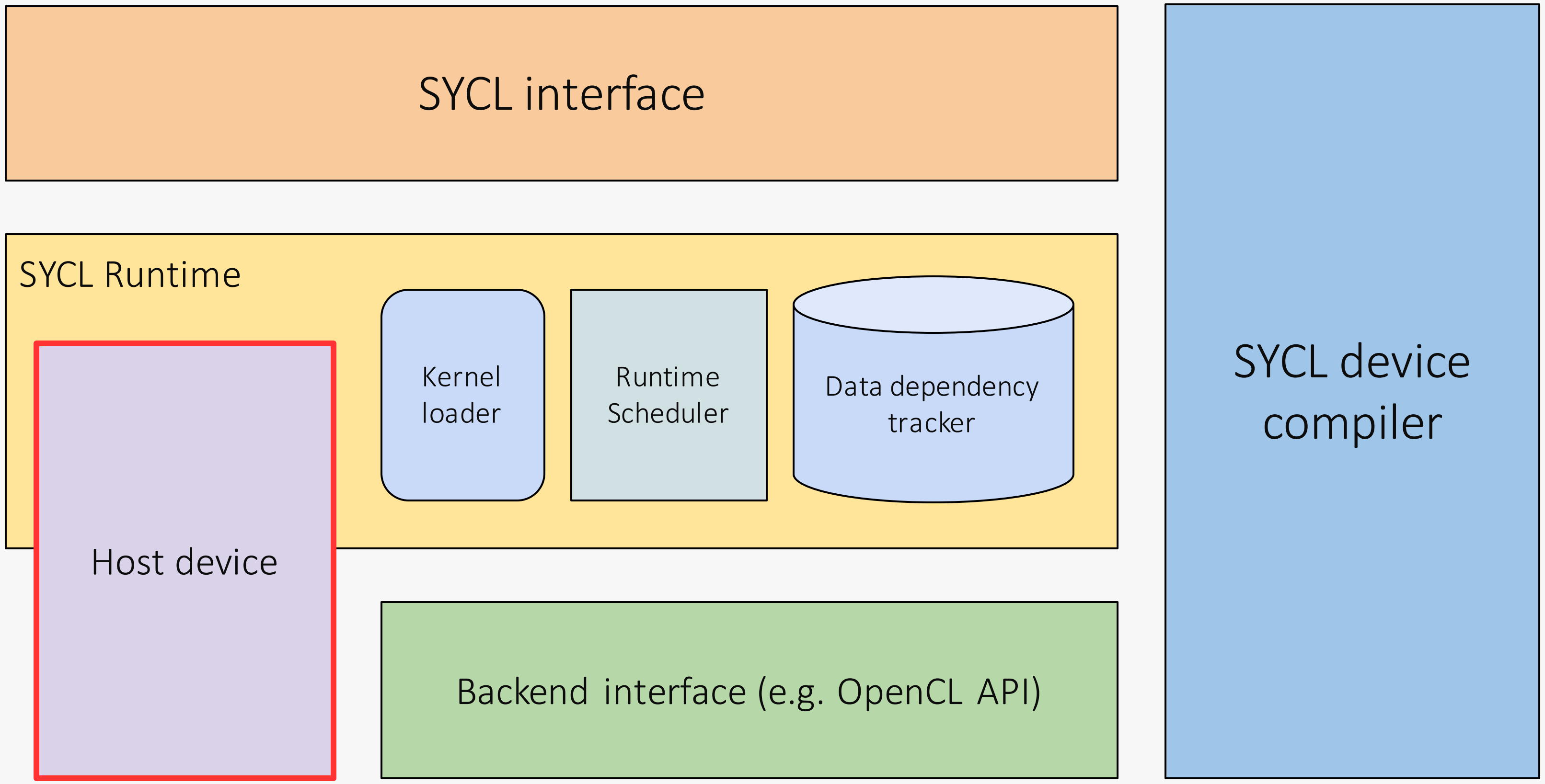

SYCL interface
SYCL Runtime
Kernel loader
Runtime Scheduler
Data dependency tracker
Host device
Backend interface (e.g. OpenCL API)
SYCL device compiler



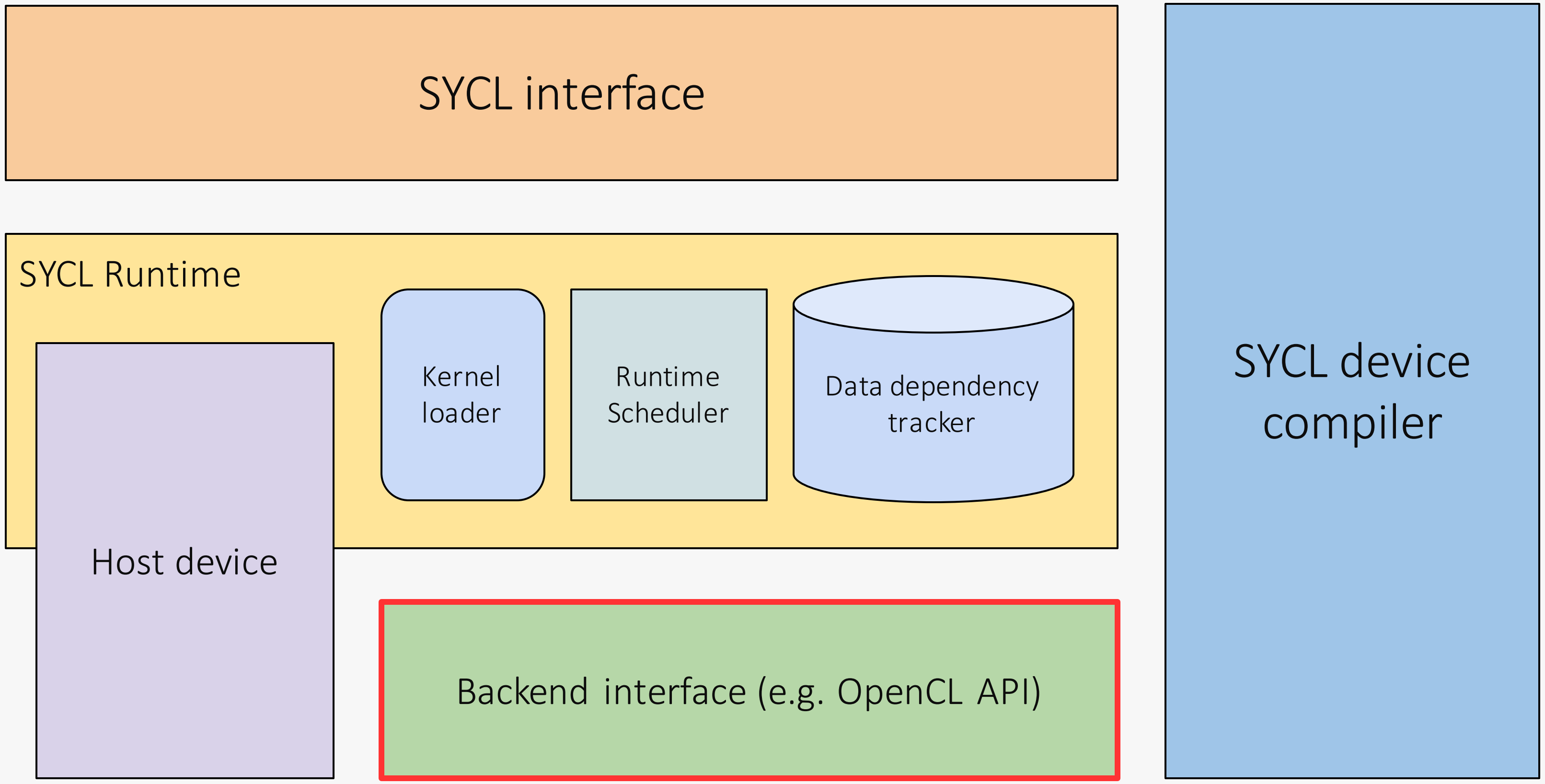
- The SYCL interface is a C++ template library that users and library developers program to
  - The same interface is used for both the host and device code



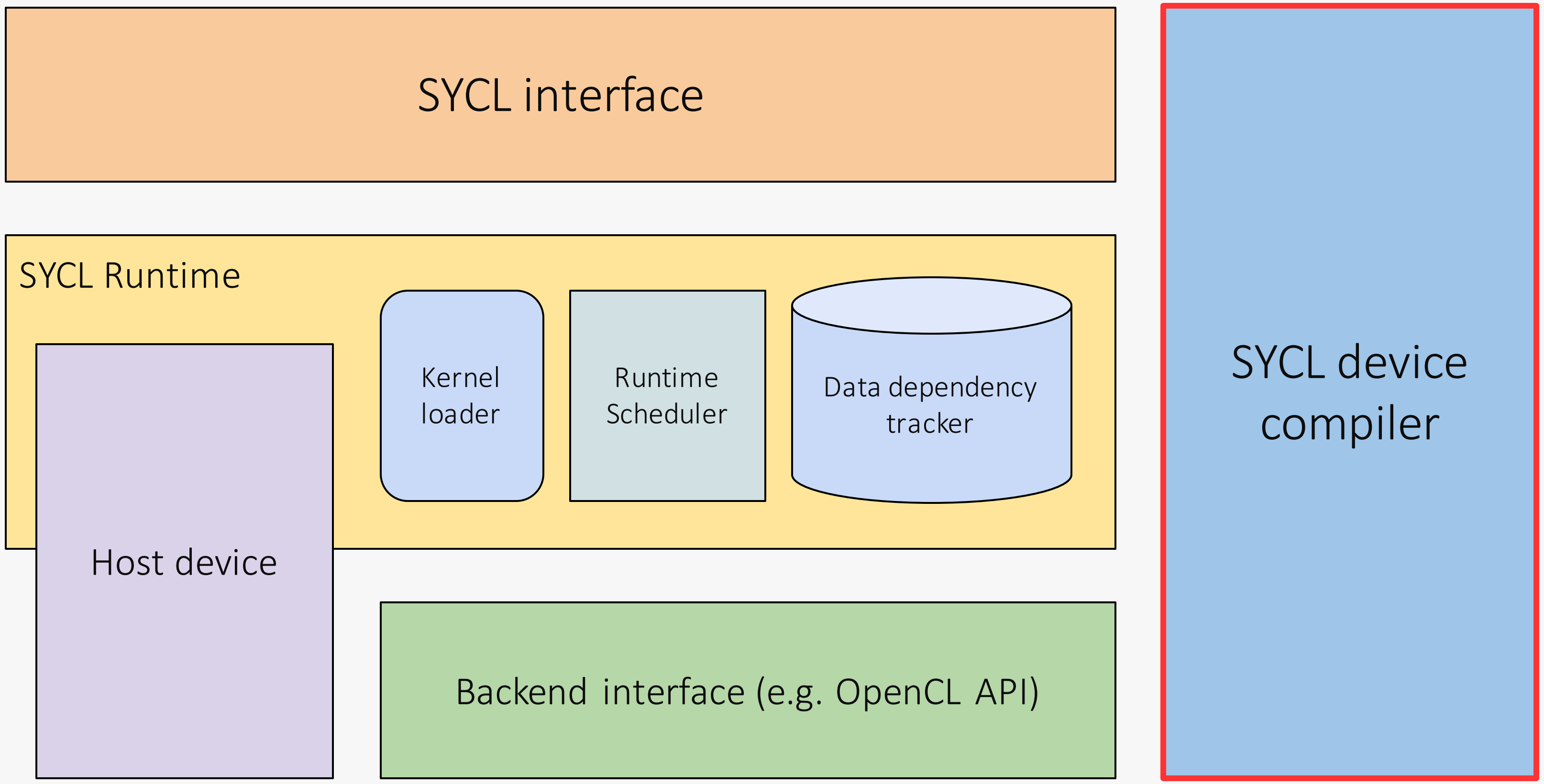
- The SYCL runtime is a library that schedules and executes work
  - It loads kernels, tracks data dependencies and schedules commands



- The host device is an emulated backend that is executed as native C++ code and emulates the SYCL execution and memory model
  - The host device can be used without backend drivers and for debugging purposes



- The backend interface is where the SYCL runtime calls down into a backend in order to execute on a particular device
  - The standard backend is OpenCL but some implementations have supported others



- The SYCL device compiler is a C++ compiler which can identify SYCL kernels and compile them down to an IR or ISA
  - This can be SPIR, SPIR-V, GCN, PTX or any proprietary vendor ISA

Example SYCL application

```
int main(int argc, char *argv[]) {











}
```

```
#include <CL/sycl.hpp>
using namespace cl::sycl;


int main(int argc, char *argv[]) {



















}
```

The whole SYCL API is included in the CL/sycl.hpp header file

```
#include <CL/sycl.hpp>
using namespace cl::sycl;


int main(int argc, char *argv[]) {


  queue gpuQueue{gpu_selector{}};










}
```

A queue is used to enqueue work to a device such as a GPU

A device selector is a function object which provides a heuristic for selecting a suitable device

```
#include <CL/sycl.hpp>
using namespace cl::sycl;


int main(int argc, char *argv[]) {


  queue gpuQeueue{gpu_selector{}};




  gpuQeueue.submit([&](handler &cgh){




  });

}
```

A command group describes a unit work of work to be executed by a device

A command group is created by a function object passed to the submit function of the queue

```
#include <CL/sycl.hpp>
using namespace cl::sycl;


int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector{}};




  gpuQeueue.submit([&](handler &cgh){




  });

}
```

We initialize three vectors, two inputs and an output

```
#include <CL/sycl.hpp>
using namespace cl::sycl;


int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector{}};

  buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
  buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
  buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

  gpuQeueue.submit([&](handler &cgh){







  });

}
```

Buffers take ownership of data and manage it across the host and any number of devices

```
#include <CL/sycl.hpp>
using namespace cl::sycl;


int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector{}};
  {
    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

    gpuQeueue.submit([&](handler &cgh){







    });
  }
}
```

Buffers synchronize on destruction via RAII waiting for any command groups that need to write back to it

```
#include <CL/sycl.hpp>
using namespace cl::sycl;


int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector{}};
  {
    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

    gpuQeueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);



    });
  }
}
```

Accessors describe the way in which you would like to access a buffer

They are also use to access the data from within a kernel function

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector{}};
  {
    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

    gpuQeueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for<add>(range<1>(dA.size()),
        [=](id<1> i){ out[i] = inA[i] + inB[i]; });
    });
  }
}
```

Commands such as parallel_for can be used to define kernel functions

The first argument here is a range, specifying the iteration space

The second argument is a function object that represents the entry point for the SYCL kernel

The function object must take an id parameter that describes the current iteration being executed

```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector{}};
  {
    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

    gpuQeueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for<add>(range<1>(dA.size()),
        [=](id<1> i){ out[i] = inA[i] + inB[i]; });
    });
  }
}
```

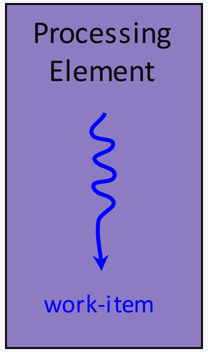
Kernel functions defined using lambdas have to have a typename to provide them with a name

The reason for this is that C++ does not have a standard ABI for lambdas so they are represented differently across the host and device compiler

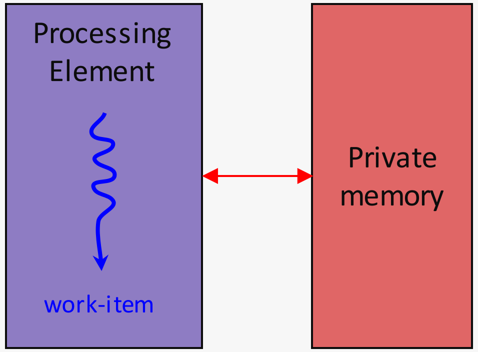
```
#include <CL/sycl.hpp>
using namespace cl::sycl;
class add;

int main(int argc, char *argv[]) {
  std::vector<float> dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector{}};
  {
    buffer<float, 1> bufA(dA.data(), range<1>(dA.size()));
    buffer<float, 1> bufB(dB.data(), range<1>(dB.size()));
    buffer<float, 1> bufO(dO.data(), range<1>(dO.size()));

    gpuQeueue.submit([&](handler &cgh){

      auto inA = bufA.get_access<access::mode::read>(cgh);
      auto inB = bufB.get_access<access::mode::read>(cgh);
      auto out = bufO.get_access<access::mode::write>(cgh);

      cgh.parallel_for<add>(range<1>(dA.size()),
        [=](id<1> i){ out[i] = inA[i] + inB[i]; });
    });
  }
}
```

This is the code which is executed on the GPU

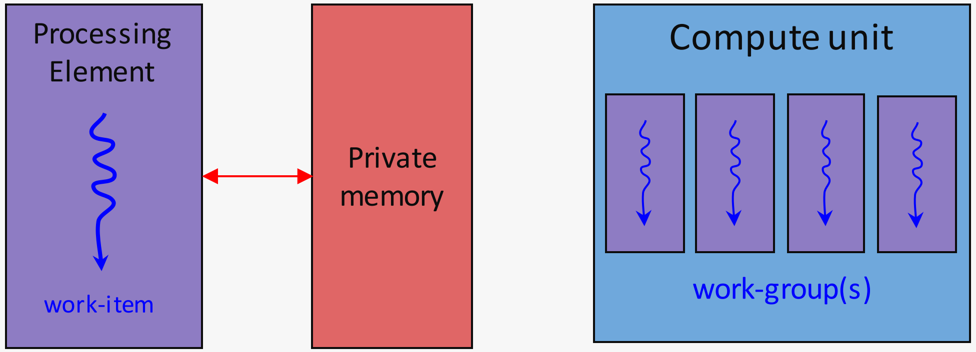
SYCL programming model



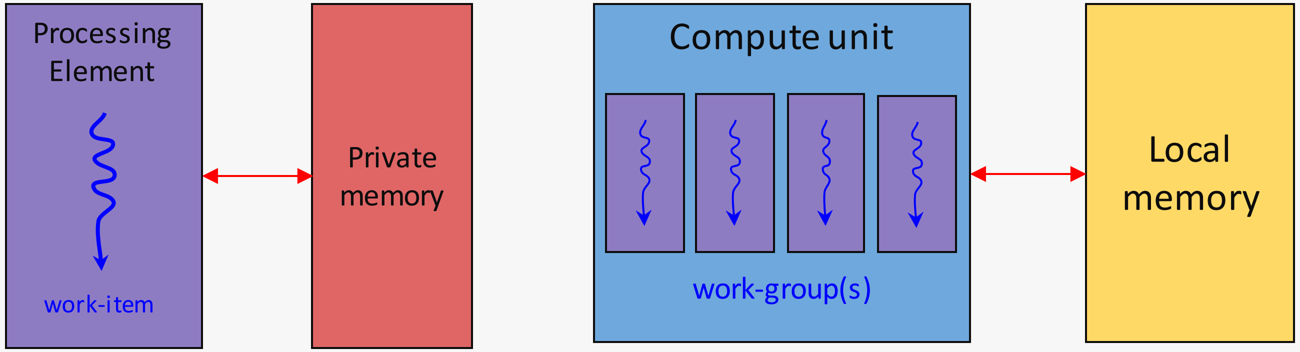
A processing element executes a single work-item



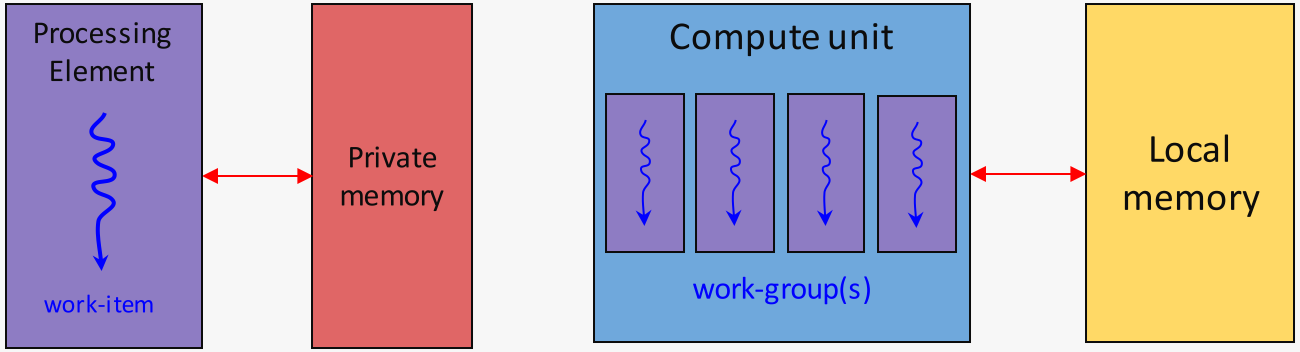
Each work-item can access private memory, a dedicated memory region for each processing element

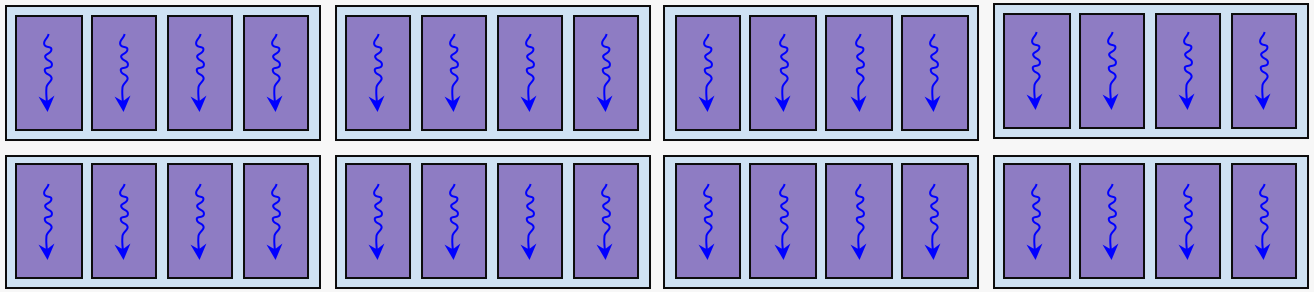


A compute is composed of a number of processing elements and executes one or more work-group which are composed of a number of work-items

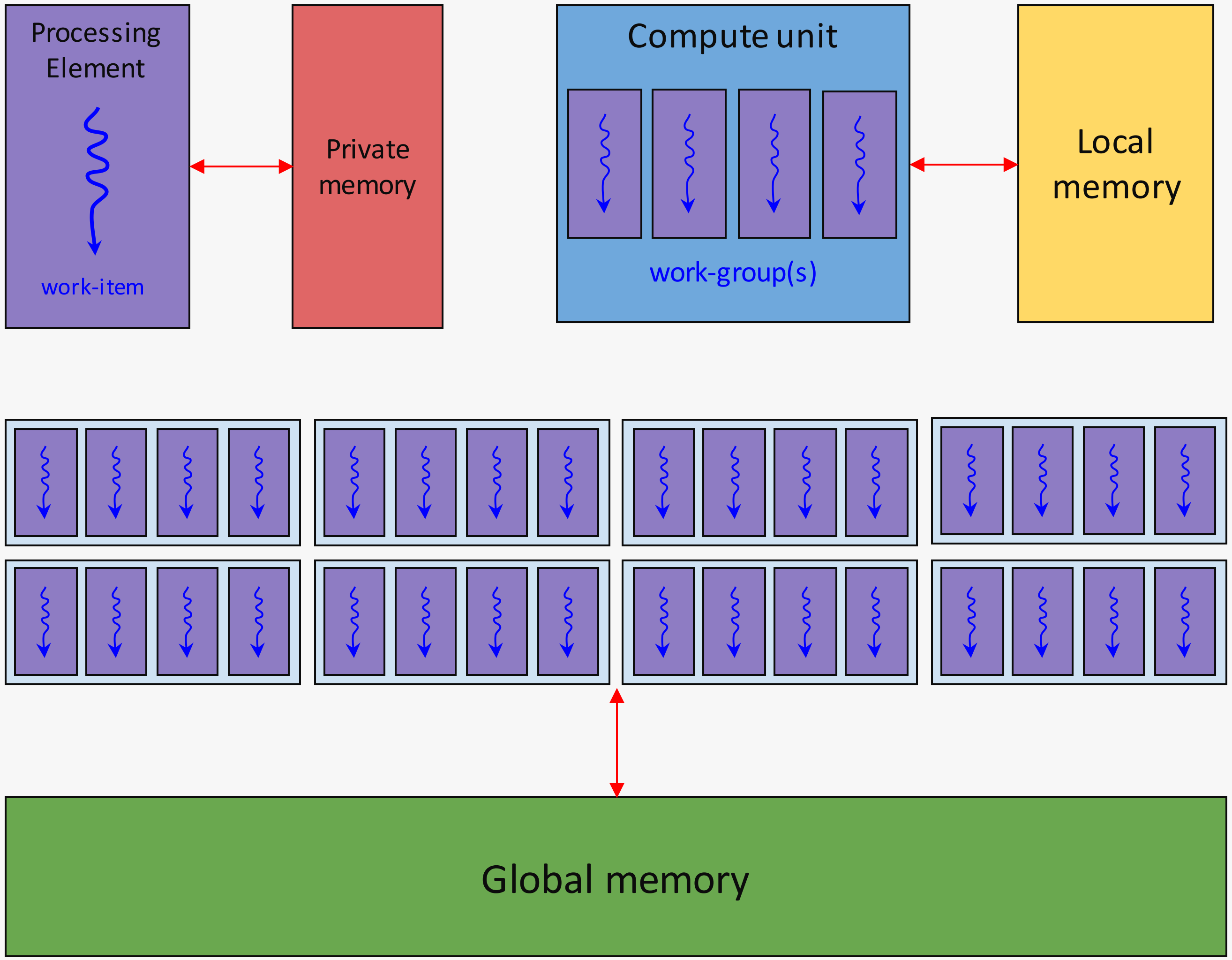


Each work-item can access the local memory of their work-group, a dedicated memory region for each compute unit

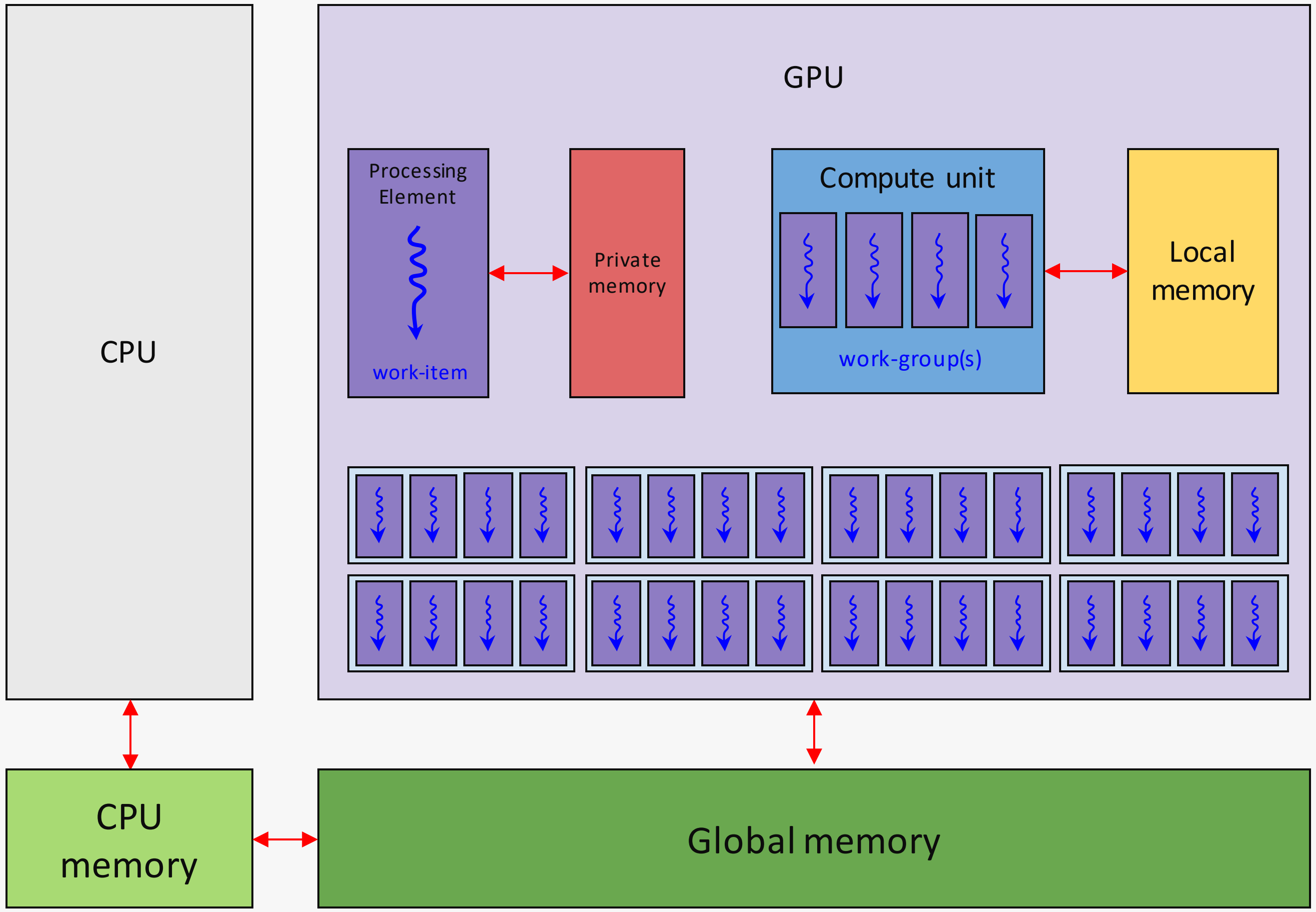




A device can execute multiple work-groups



Each work-item can access global memory, a single memory region available to all processing elements

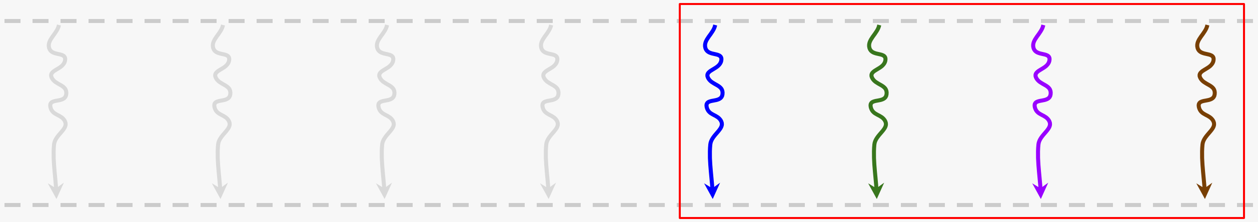


Data must be copied or mapped between the host CPU memory and the GPU's global memory

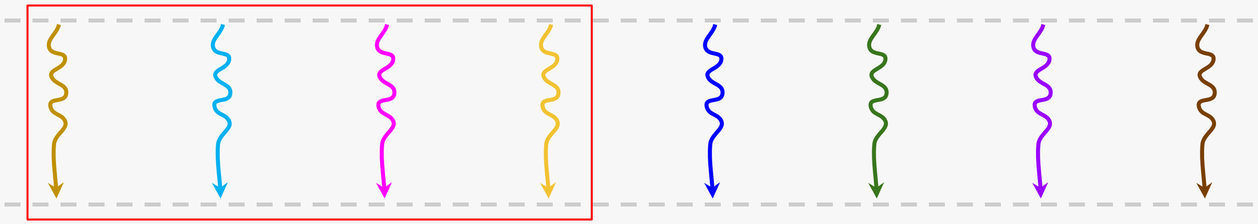
This is can be very expensive depending on the architecture

Private memory < Local memory < Global memory

GPUs execute a large number of work-items



They are not all guaranteed to execute concurrently, most GPUs do execute a number of work-items uniformly (lock-step)



The number that are executed concurrently varies between different GPUs

There is no guarantee as to the order in which they execute

What are GPUs good at?

- Highly parallel
  - *GPUs can run a very large number of processing elements in parallel*
- Efficient at floating point operations
  - *GPUs can achieve very high FLOPs (floating-point operations per second)*
- Large bandwidth
  - *GPUs are optimised for throughput and can handle a very large bandwidth of data*

Optimising GPU programs

There are different levels of optimisations you can apply

- Choosing the right algorithm
  - *This means choosing an algorithm that is well suited to parallelism*
- Basic GPU programming principles
  - *Such as coalescing global memory access or using local memory*
- Architecture specific optimisations
  - *Optimising for register usage or avoiding bank conflicts*
- Micro-optimisations
  - *Such as floating point dnorm hacks*

There are different levels of optimisations you can apply

- Choosing the right algorithm
  - *This means choosing an algorithm that is well suited to parallelism*
- Basic GPU programming principles
  - *Such as coalescing global memory access or using local memory*
- Architecture specific optimisations
  - *Optimising for register usage or avoiding bank conflicts*
- Micro-optimisations
  - *Such as floating point dnorm hacks*

This talk will focus on these two

Choosing the right algorithm

What to parallelise on a GPU

- Find hotspots in your code base
  - *Looks for areas of your codebase that are hit often and well suited to parallelism on the GPU*

- Follow an adaptive optimisation approach such as APOD
  - ***A**nalyse -> **P**arallelise -> **O**ptimise -> **D**eploy*

- Avoid over-optimisation
  - *You may reach a point where optimisations provide diminishing returns*

What to look for in an algorithm

- Naturally data parallel
  - *Performing the same operation on multiple items in the computation*
- Large problem
  - *Enough work to utilise the GPU's processing elements*
- Independent progress
  - *Little or no dependencies between items in the computation*
- Non-divergent control flow
  - *Little or no branch or loop divergence*

Basic GPU programming principles

Optimizing GPU programs means maximizing throughput

$$\frac{\text{Compute}}{\text{Memory}}$$

Maximize compute operations

Reduce time spent on memory

Optimizing GPU programs means maximizing throughput

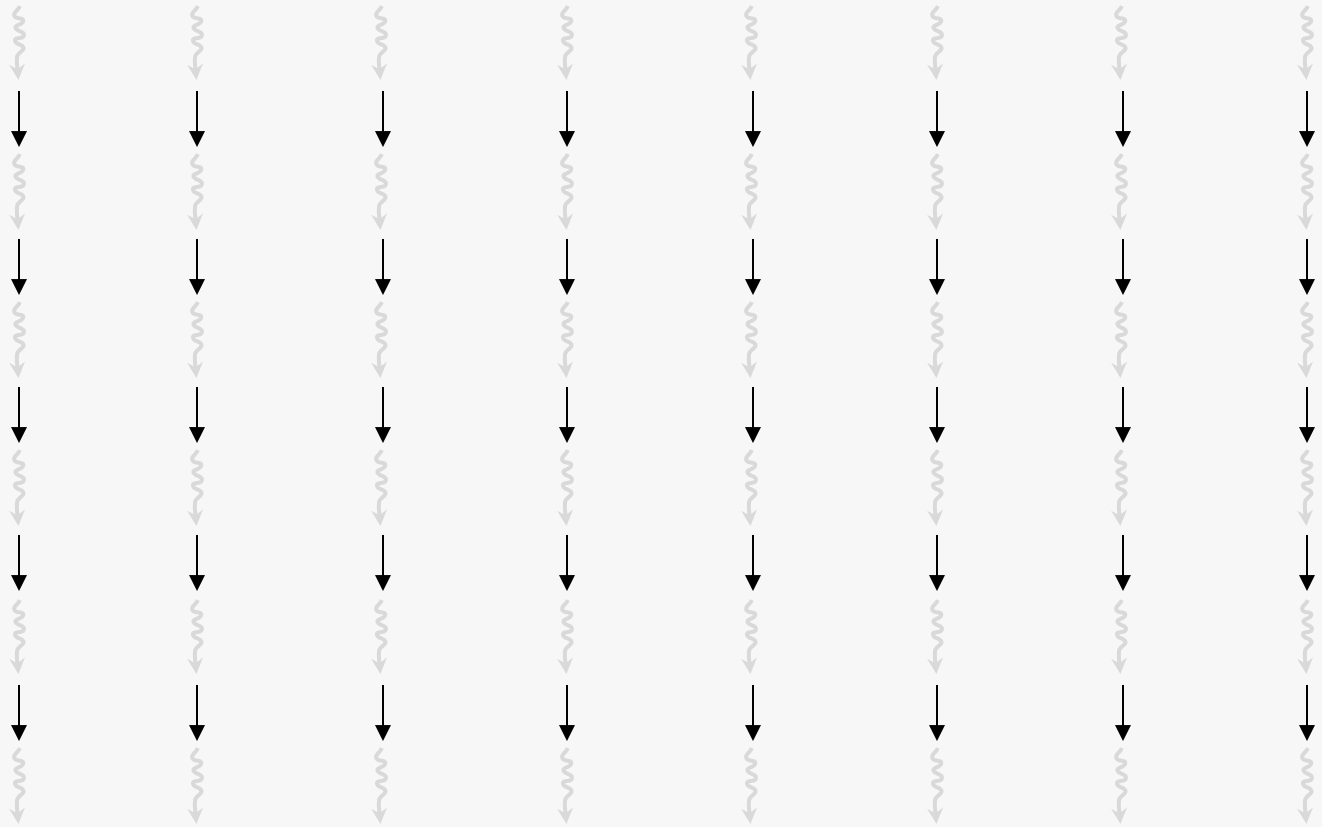
- Maximise compute operations per cycle
  - *Make effective utilisation of the GPU's hardware*

- Reduce time spent on memory operations
  - *Reduce latency of memory access*

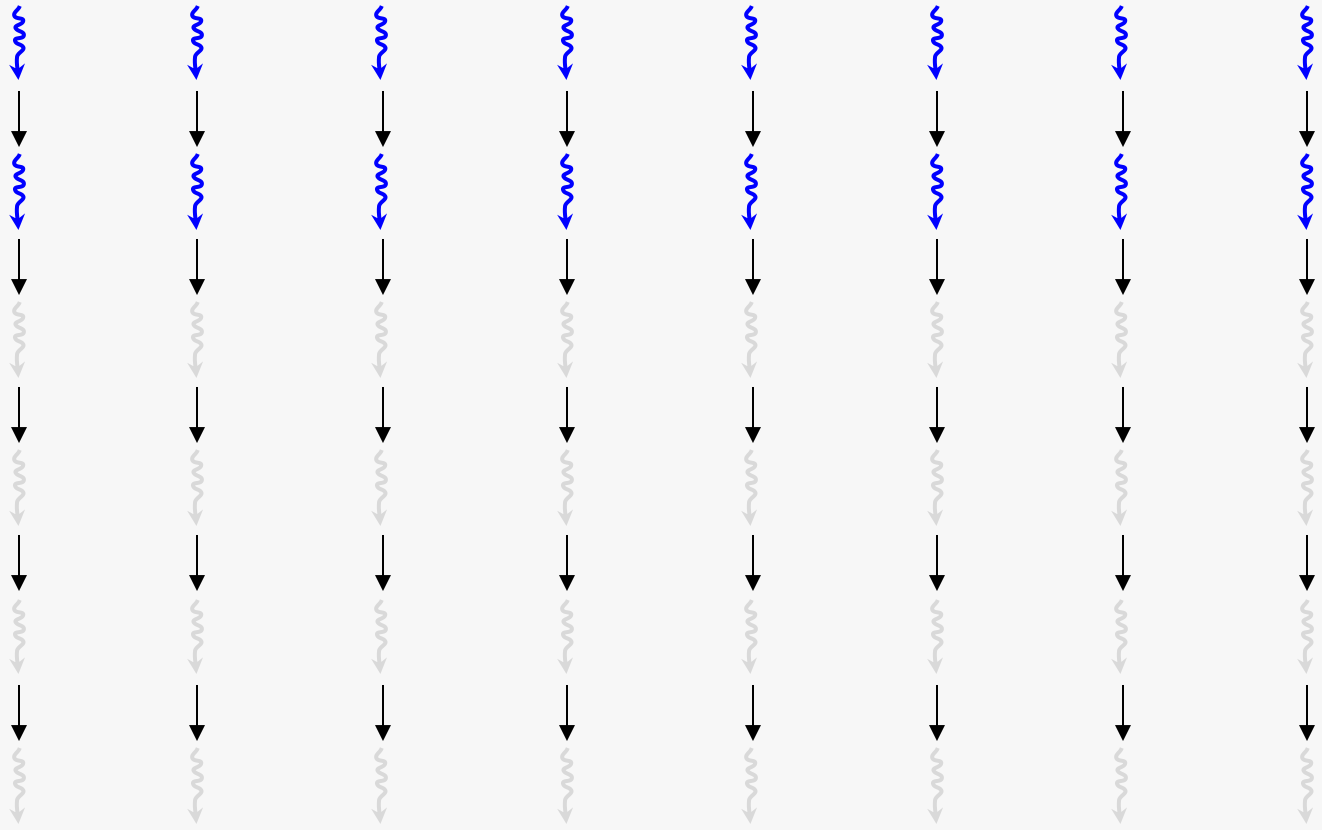
Avoid divergent control flow

- Divergent branches and loops can cause inefficient utilisation
  - *If consecutive work-items execute different branches they must execute separate instructions*
  - *If some work-items execute more iterations of a loop than neighbouring work-items this leaves them doing nothing*

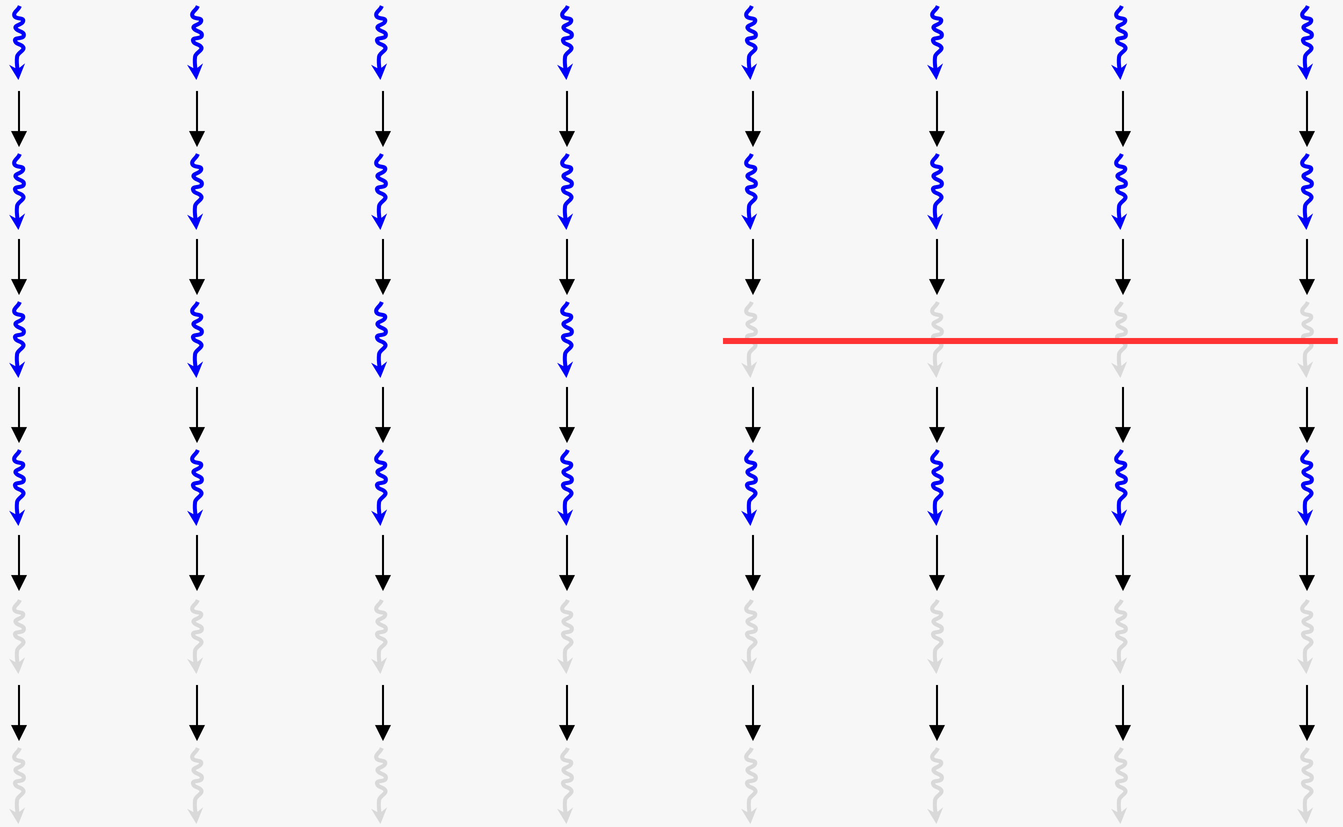
```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



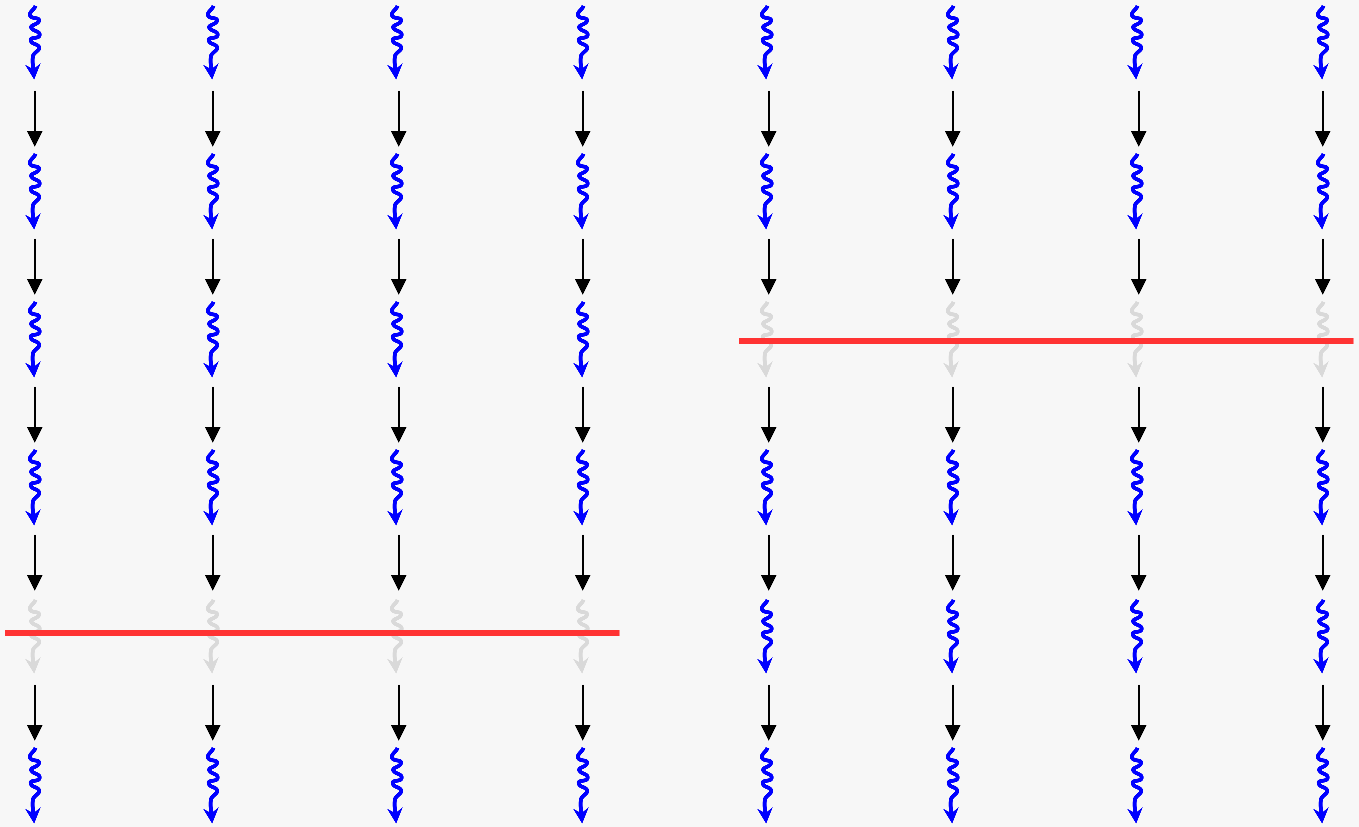
```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



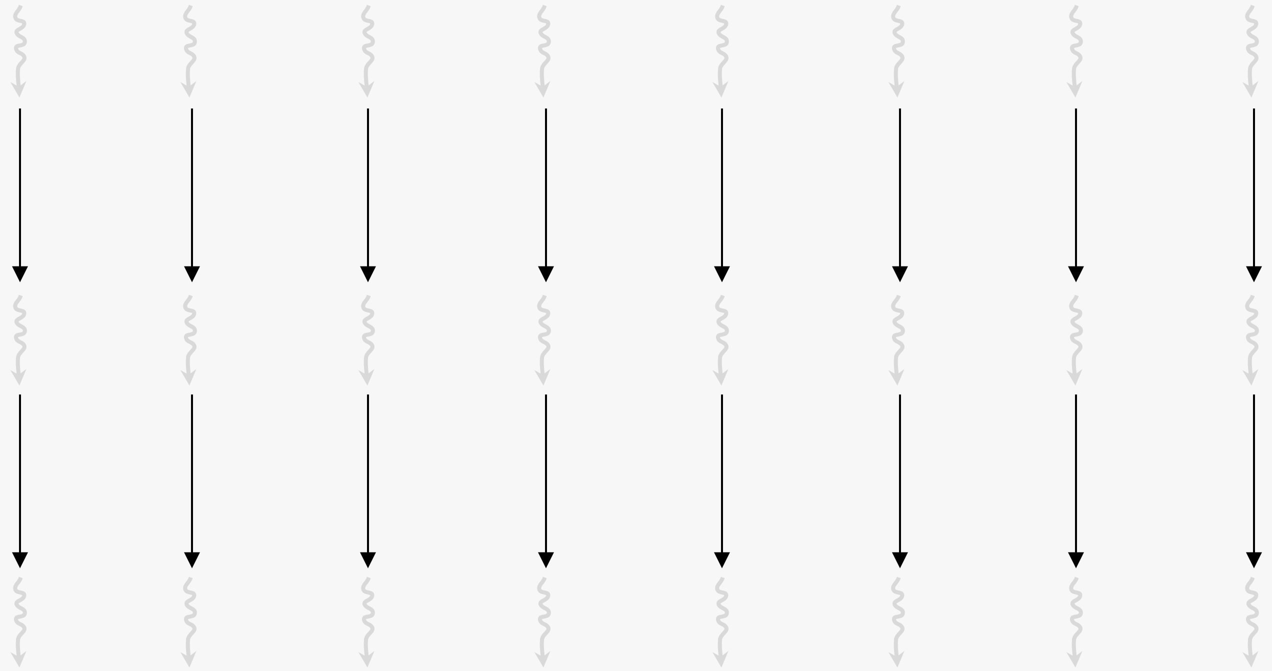
```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



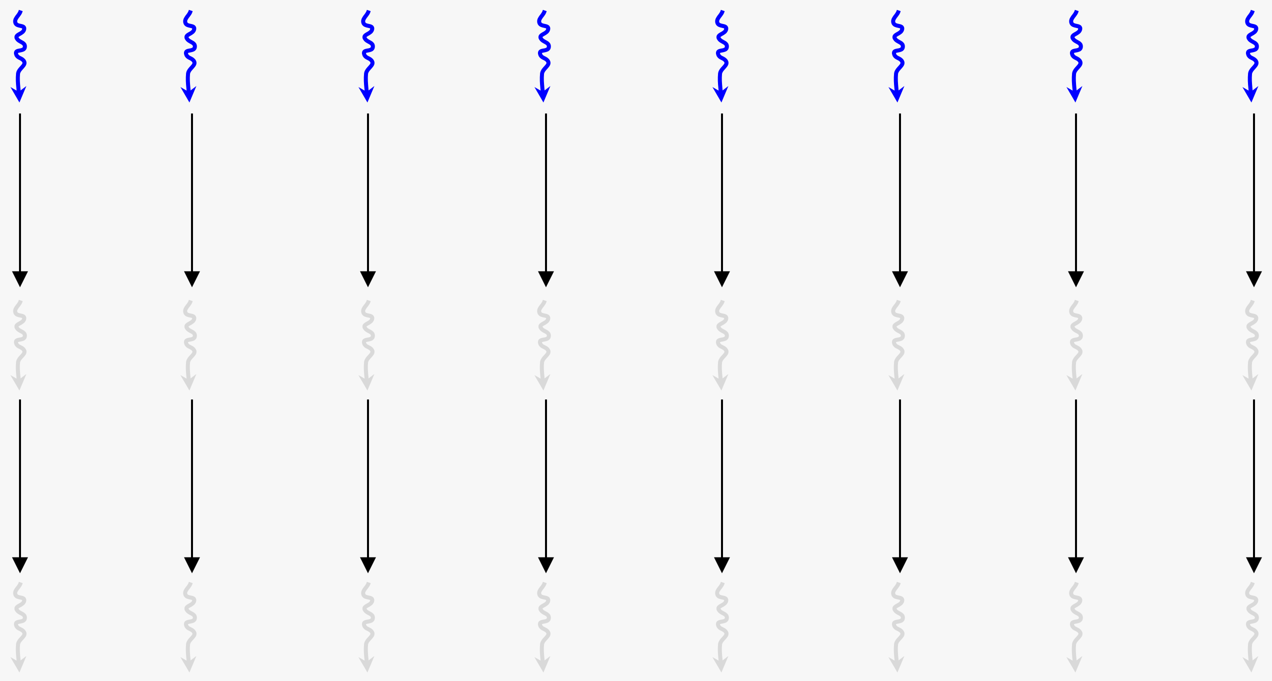
```
a[globalId] = 0;

if (globalId < 4) {

  a[globalId] = x();

} else {

  a[globalId] = y();

}
```



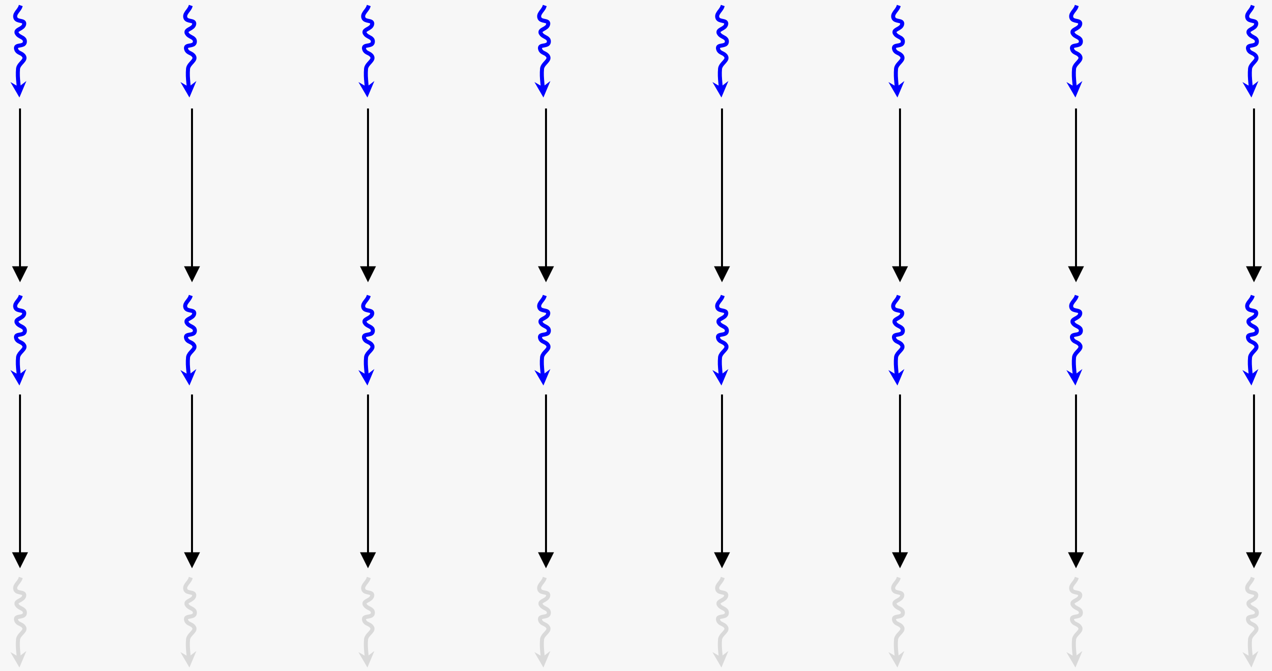
```
…

for (int i = 0; i < 
  globalId; i++) {
  do_something();
}

…
```



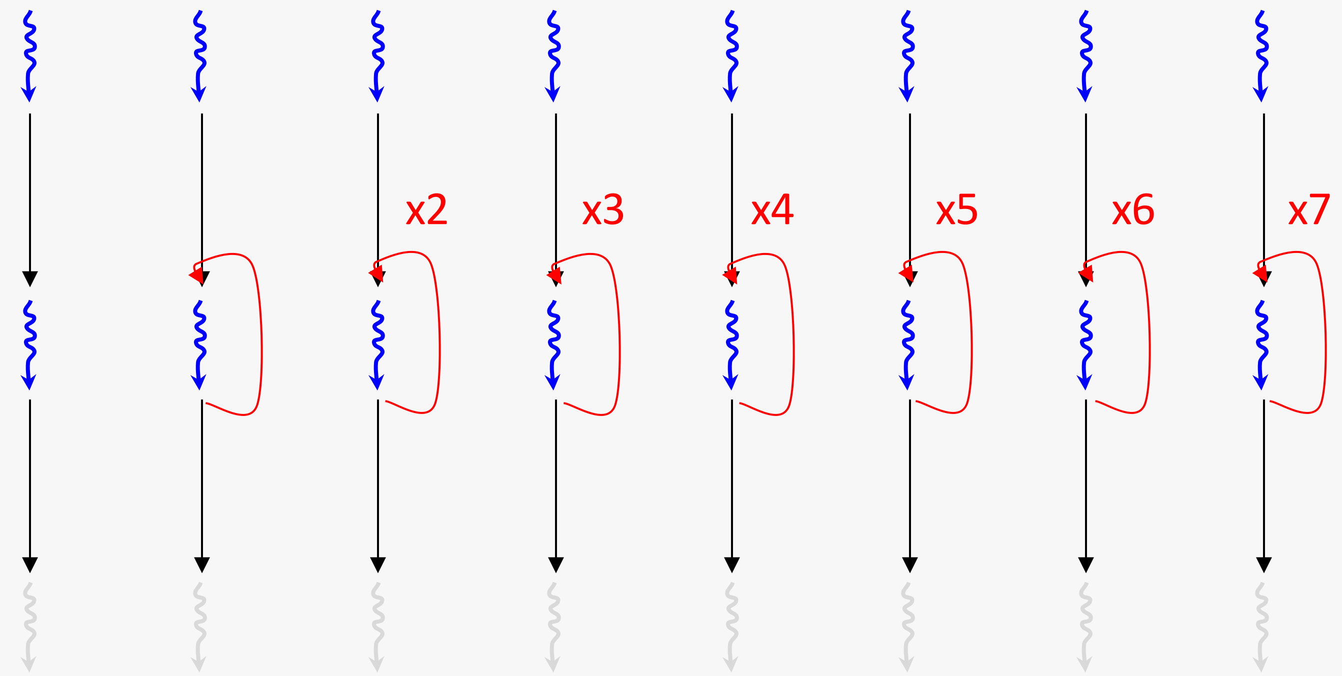
```
…

for (int i = 0; i <
  globalId; i++) {
  do_something();
}

…
```



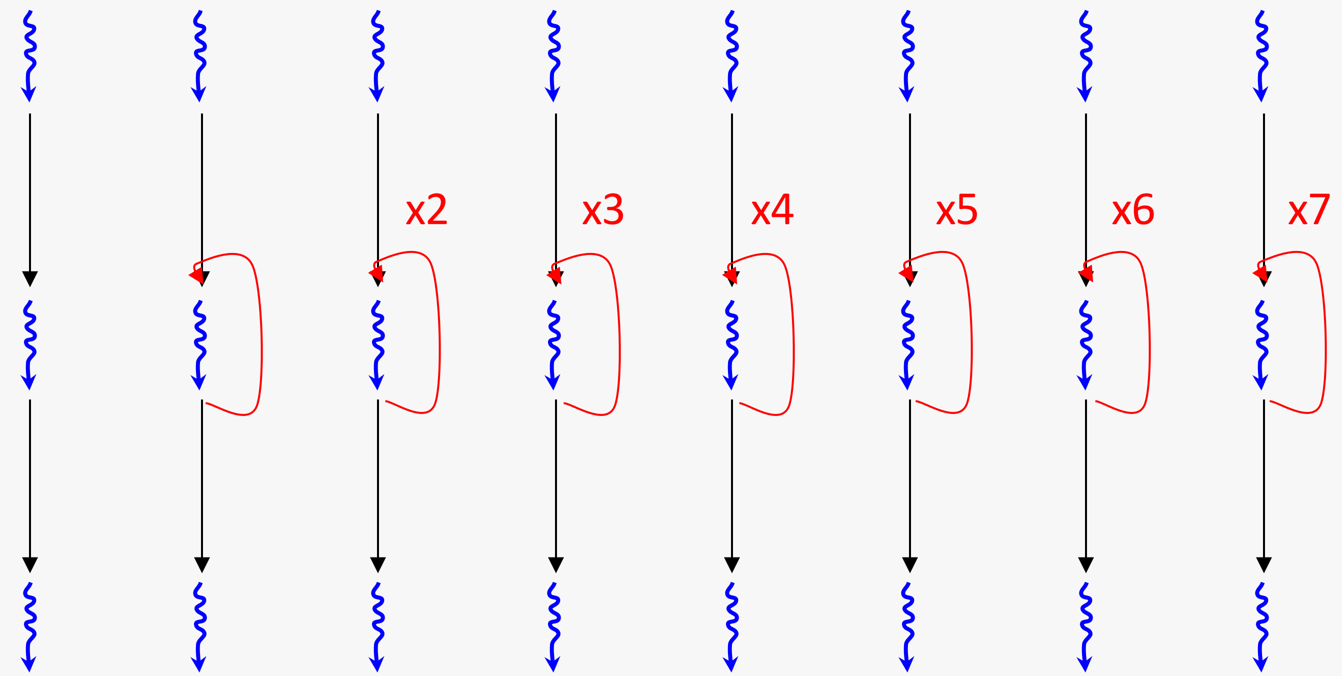
```
…

for (int i = 0; i <
  globalId; i++) {
  do_something();
}

…
```



```
...

for (int i = 0; i <
   globalId; i++) {
   do_something();
}

...
```



```
…

for (int i = 0; i <
  globalId; i++) {
  do_something();
}

…
```

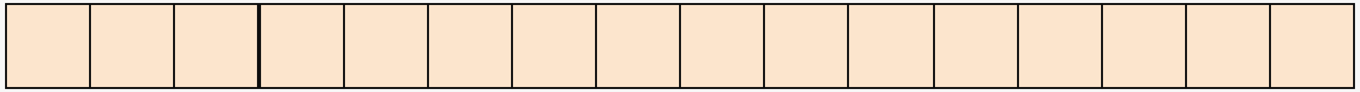


Coalesced global memory access

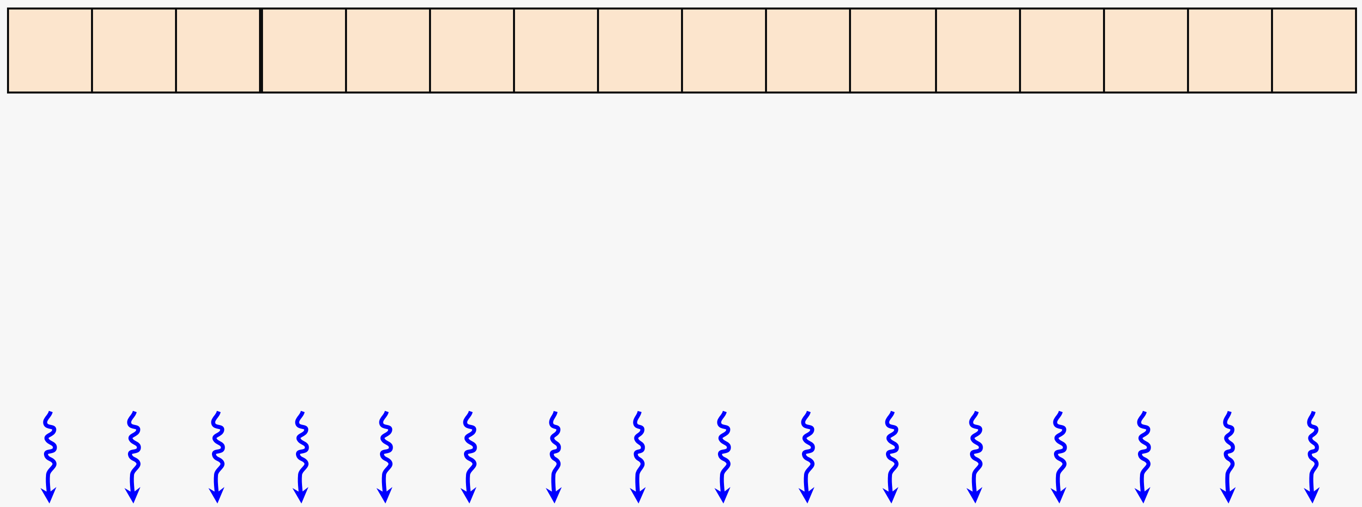
- Reading and writing from global memory is very expensive
  - *It often means copying across an off-chip bus*

- Reading and writing from global memory is done in chunks
  - *This means accessing data that is physically close together in memory is more efficient*

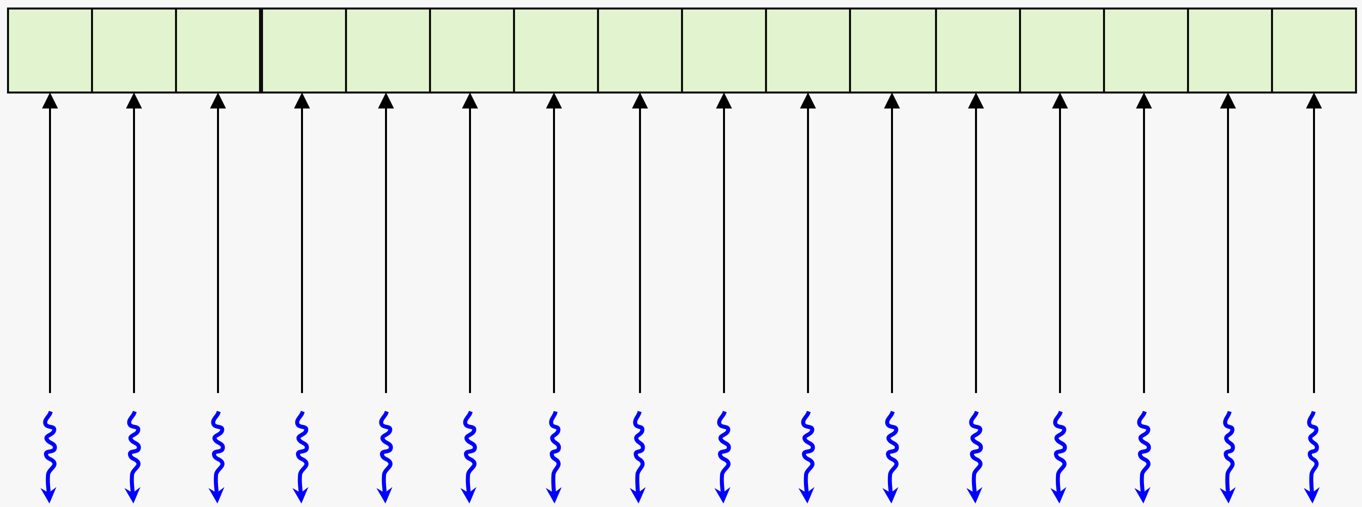
```
float data[size];
```



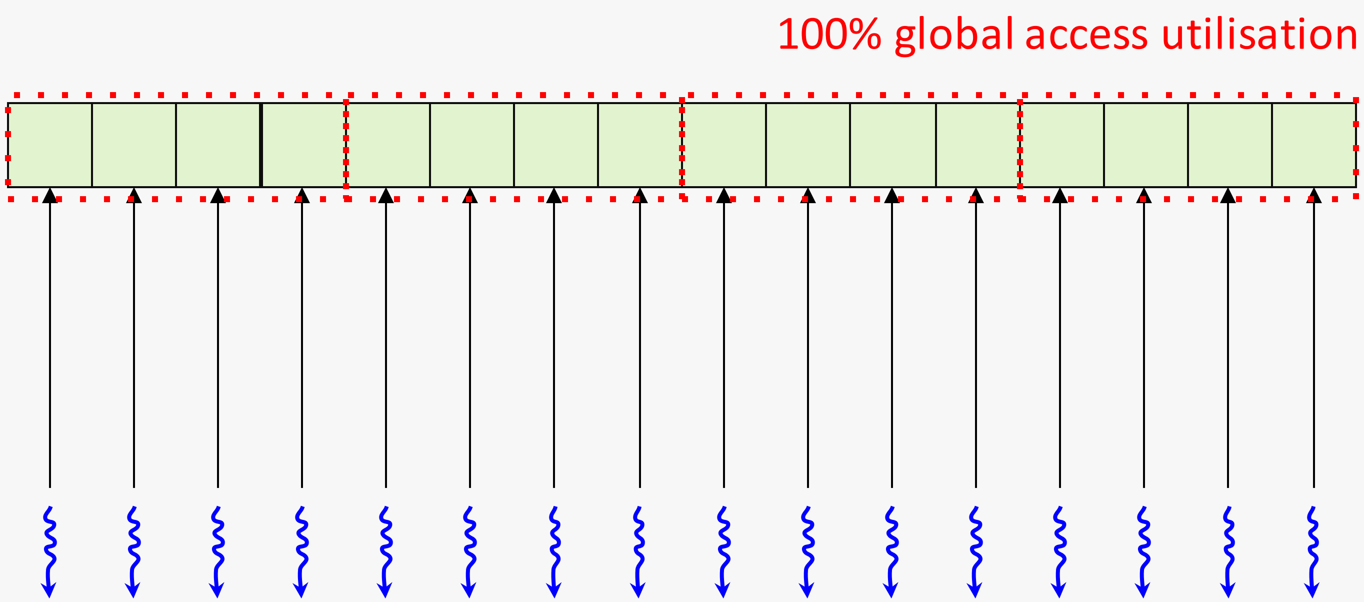
```
float data[size];

...

f(a[globalId]);
```



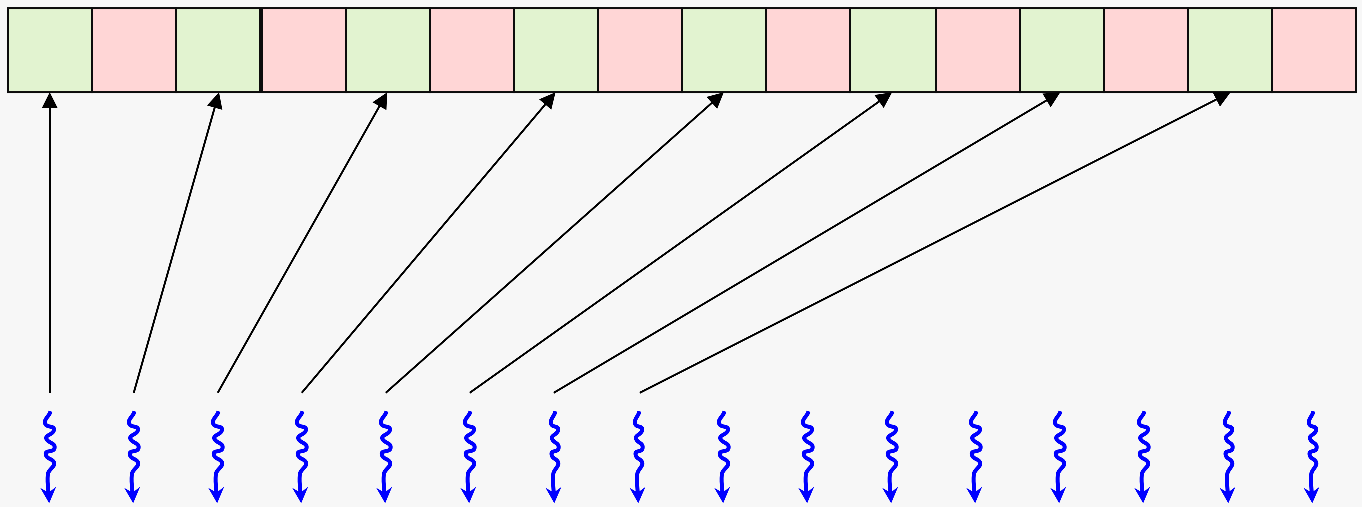
```
float data[size];

...

f(a[globalId]);
```



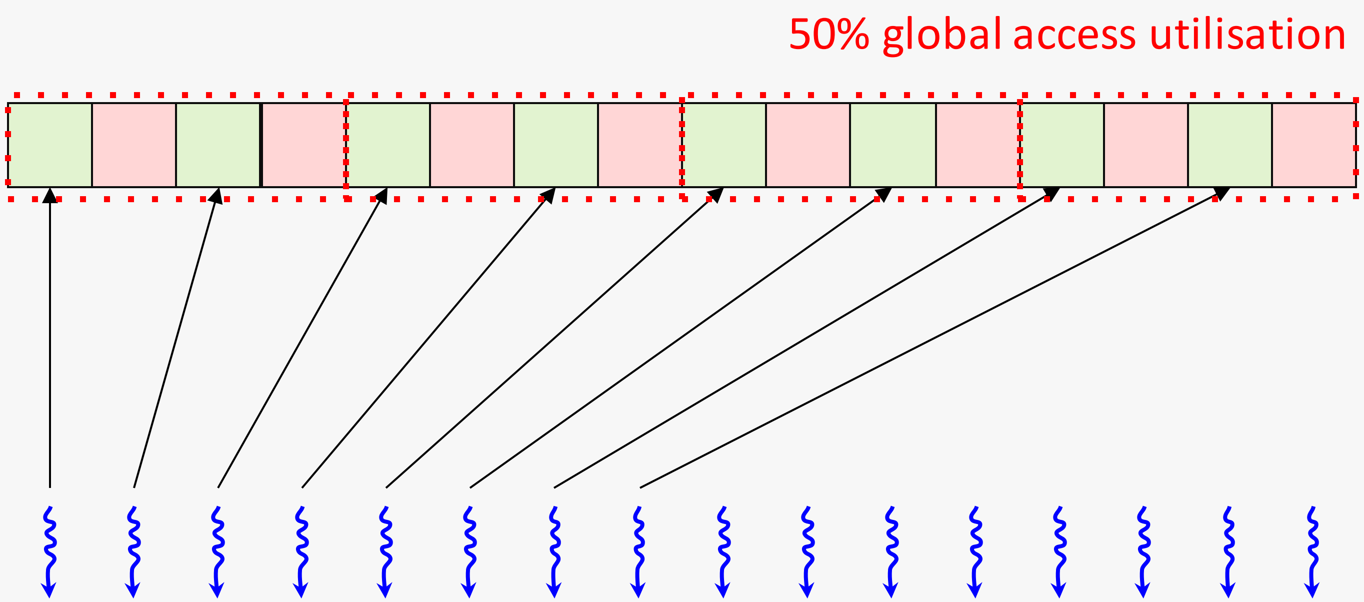
```
float data[size];

...

f(a[globalId]);
```

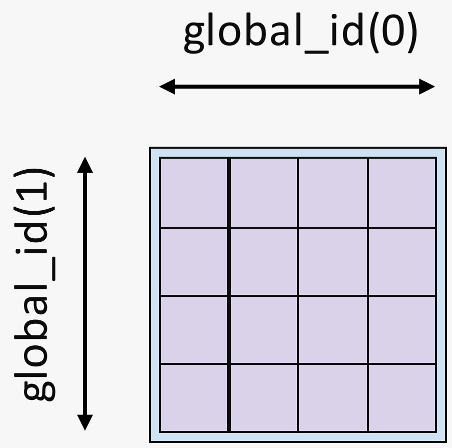


```
float data[size];

...

f(a[globalId * 2]);
```



```
float data[size];

...

f(a[globalId * 2]);
```





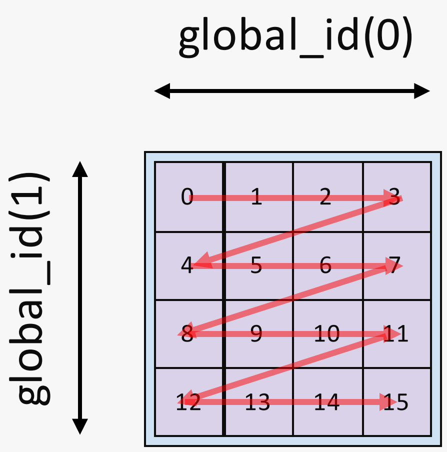
Row-major

```
auto id0 = get_global_id(0);

auto id1 = get_global_id(1);

auto linearId = (id1 * 4) + id0;

a[linearId] = f();
```

This becomes very important when dealing with multiple dimensions

It’s important to ensure that the order work-items are executed in aligns with the order that data elements that are accessed

This maintains coalesced global memory access



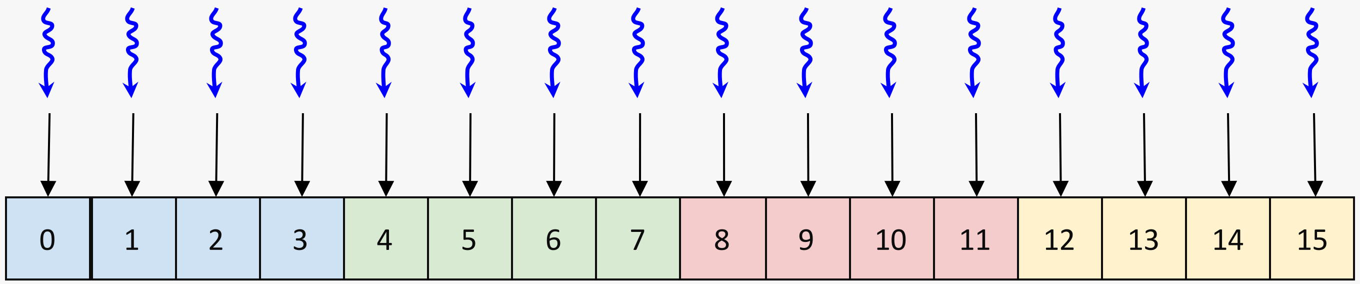
Row-major

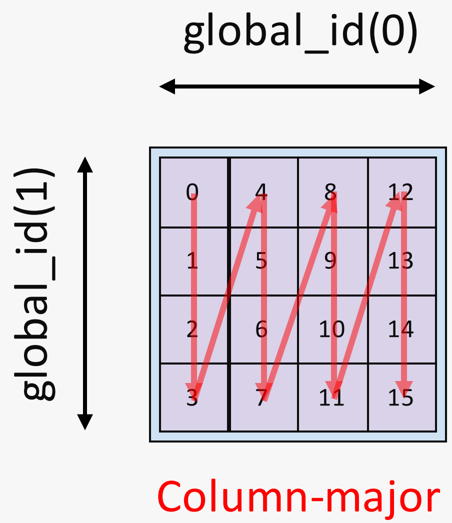
Row-major

```
auto id0 = get_global_id(0);

auto id1 = get_global_id(1);

auto linearId = (id1 * 4) + id0;

a[linearId] = f();
```

Here data elements are accessed in row-major and work-items are executed in row-major

Global memory access is coalesced



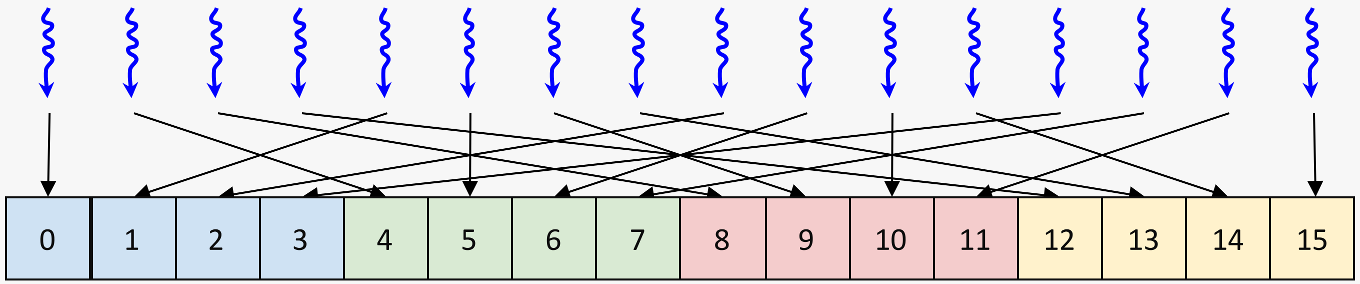


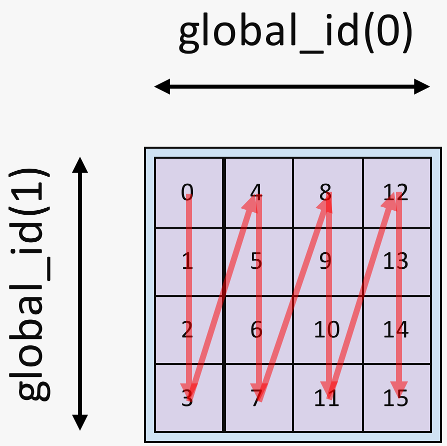
Row-major

```
auto id0 = get_global_id(0);

auto id1 = get_global_id(1);

auto linearId = (id1 * 4) + id0;

a[linearId] = f();
```

If the work-items were executed in column-major

Global memory access is no longer coalesced





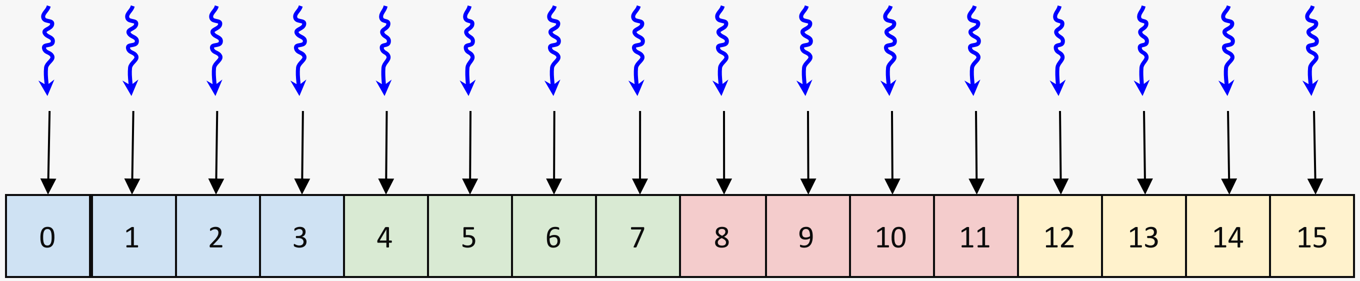
Column-major

Column-major

```
auto id0 = get_global_id(0);

auto id1 = get_global_id(1);

auto linearId = (id0 * 4) + id1;

a[linearId] = f();
```

However if you were to switch the data access pattern to column-major

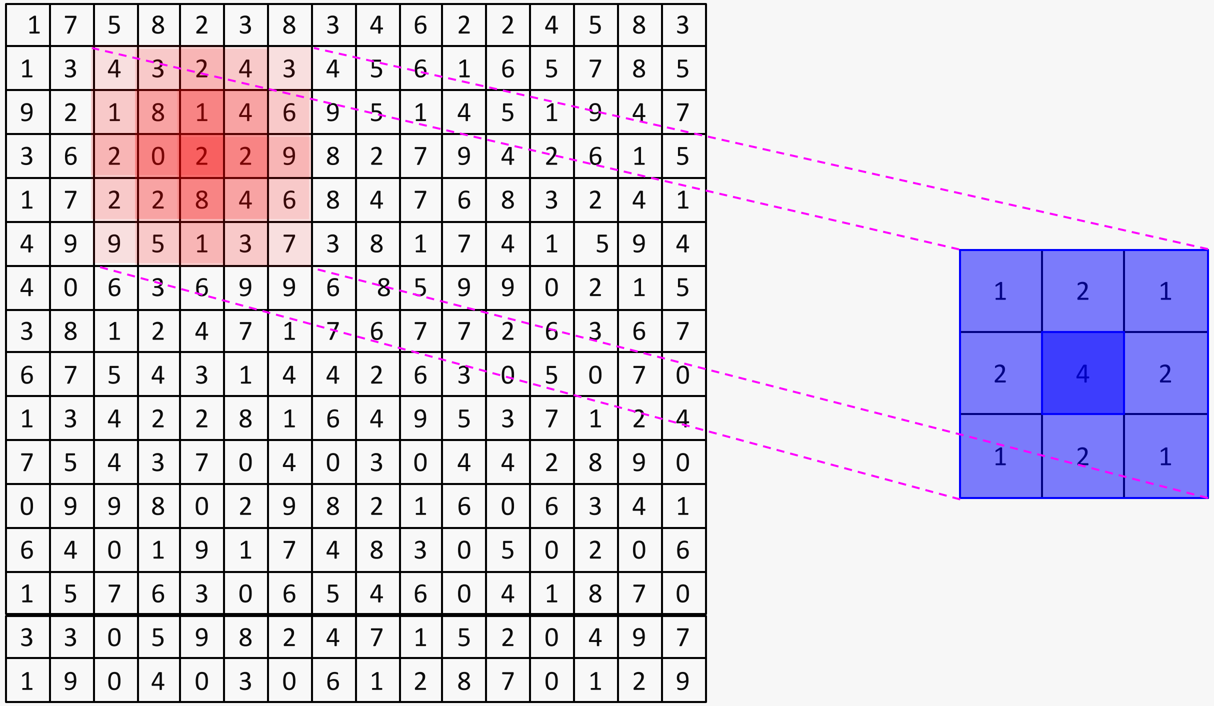
Global memory access is coalesced again



Make use of local memory

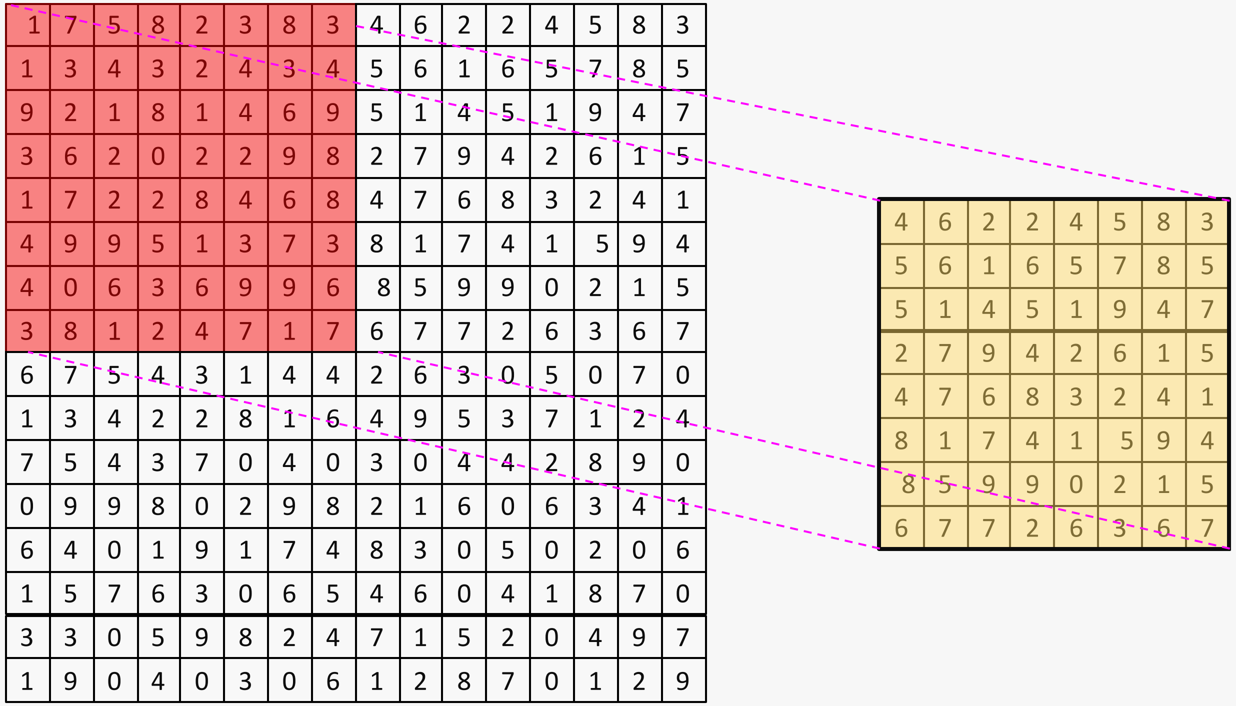
- Local memory is much lower latency to access than global memory
  - *Cache commonly accessed data and temporary results in local memory rather than reading and writing to global memory*

- Using local memory is not necessarily always more efficient
  - *If data is not accessed frequently enough to warrant the copy to local memory you may not see a performance gain*



If each work-item needs to access a number of neighbouring elements

And each of these operations loads directly from global memory this is can be very expensive

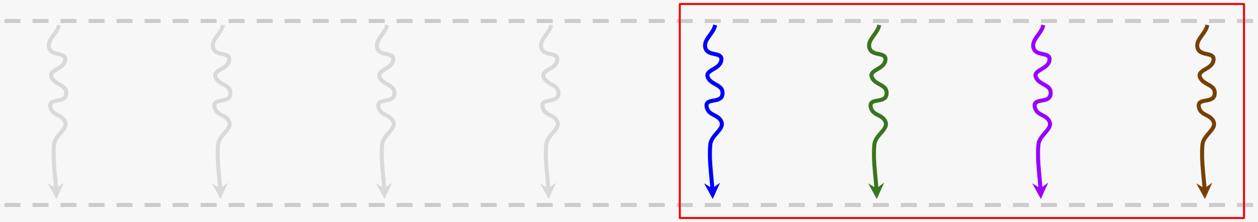


A common technique to avoid this is to use local memory to break up your data into tiles

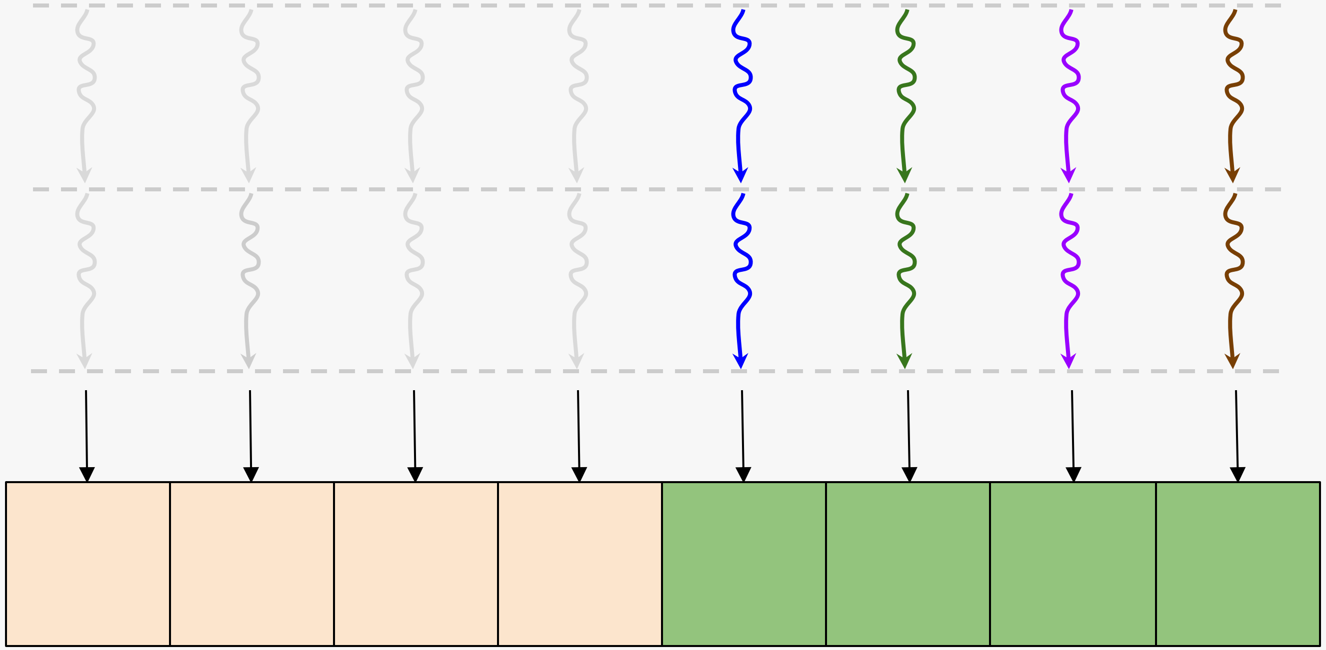
Then each tile can be moved to local memory while a work-group is working on it

Synchronise work-groups when necessary

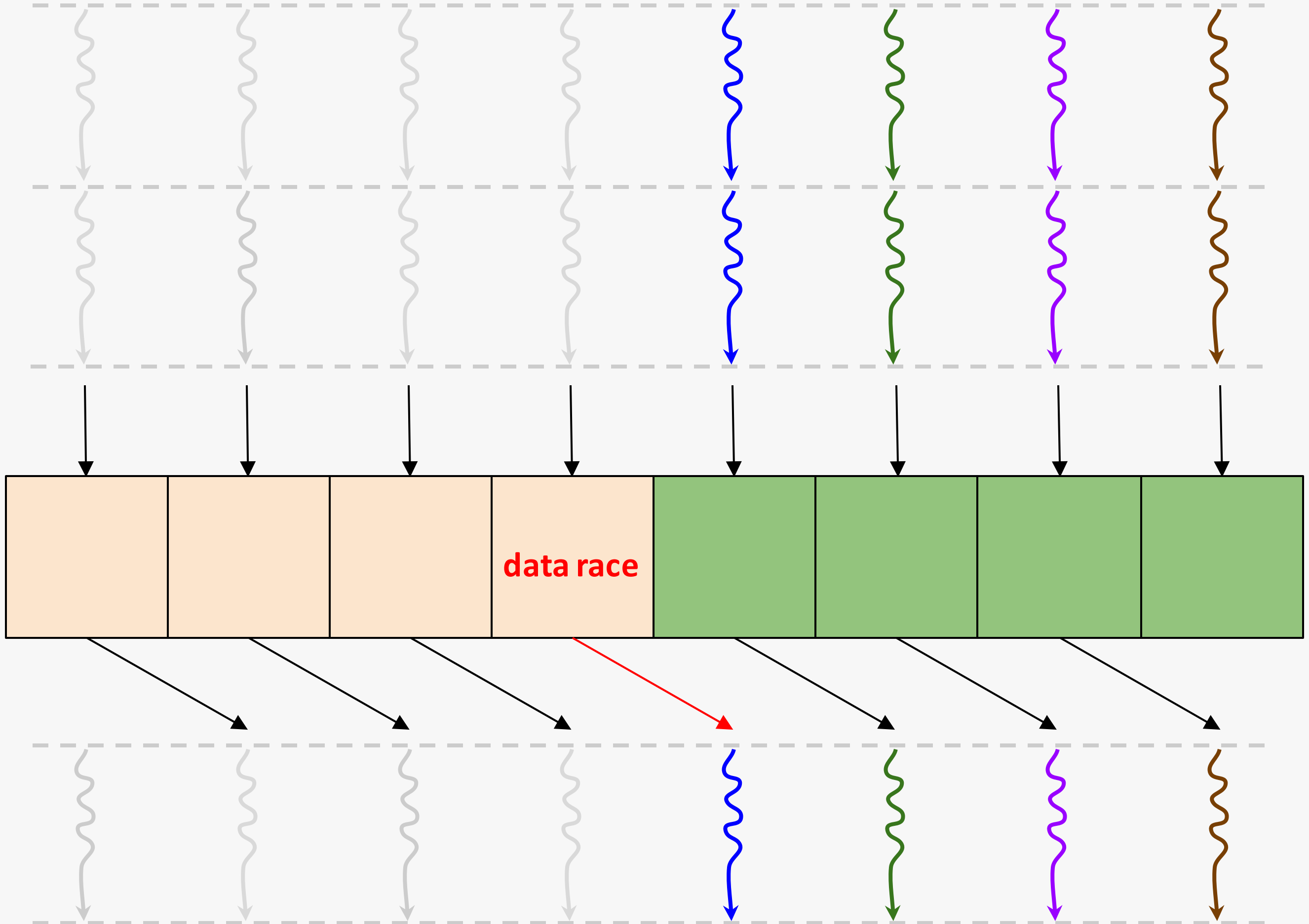
- Synchronising with a work-group barrier waits for all work-items to reach the same point
  - *Use a work-group barrier if you are copying data to local memory that neighbouring work-items will need to access*
  - *Use a work-group barrier if you have temporary results that will be shared with other work-items*



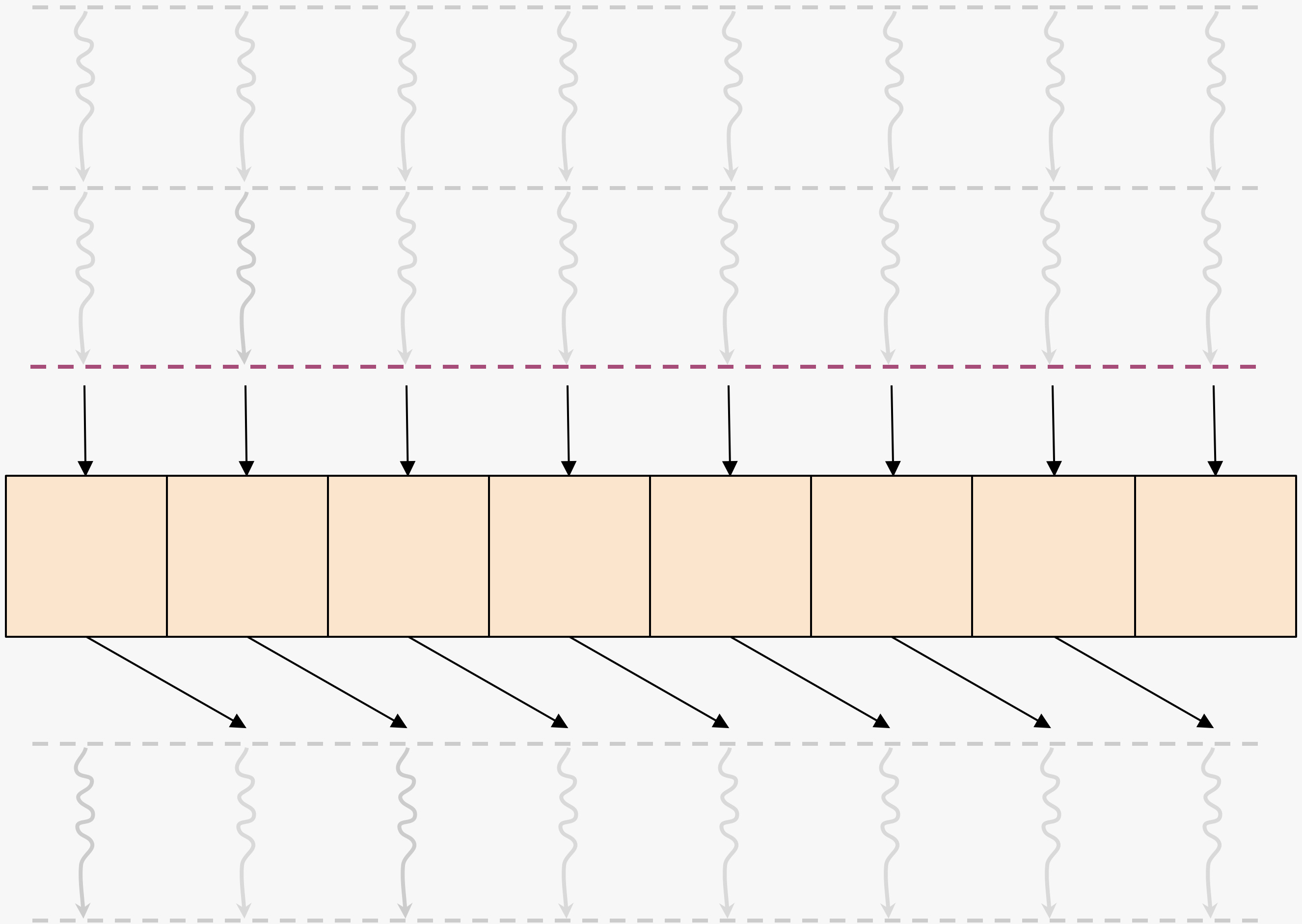
Remember that work-items are not all guaranteed to execute concurrently



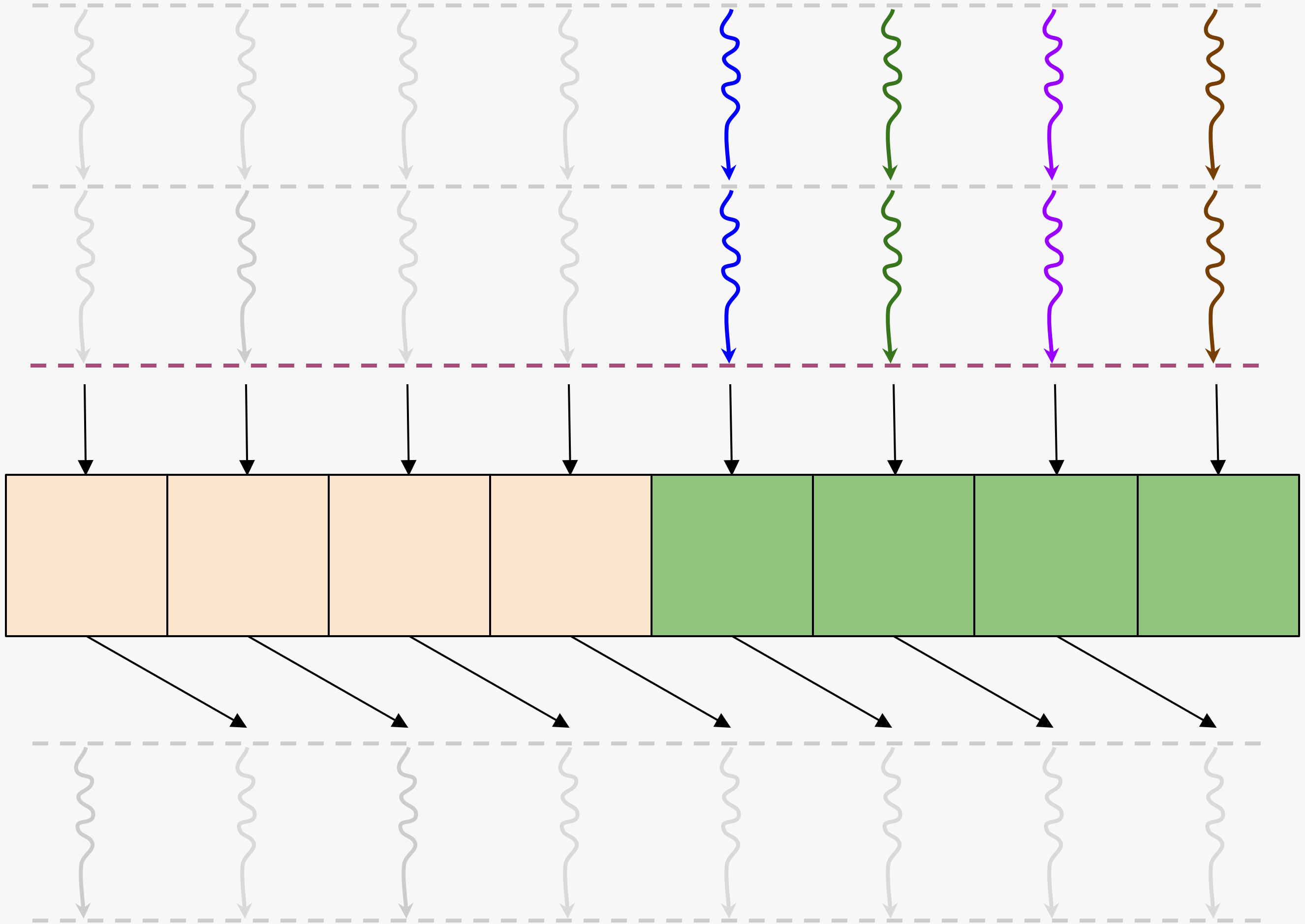
A work-item can share results with other work-items via local and global memory



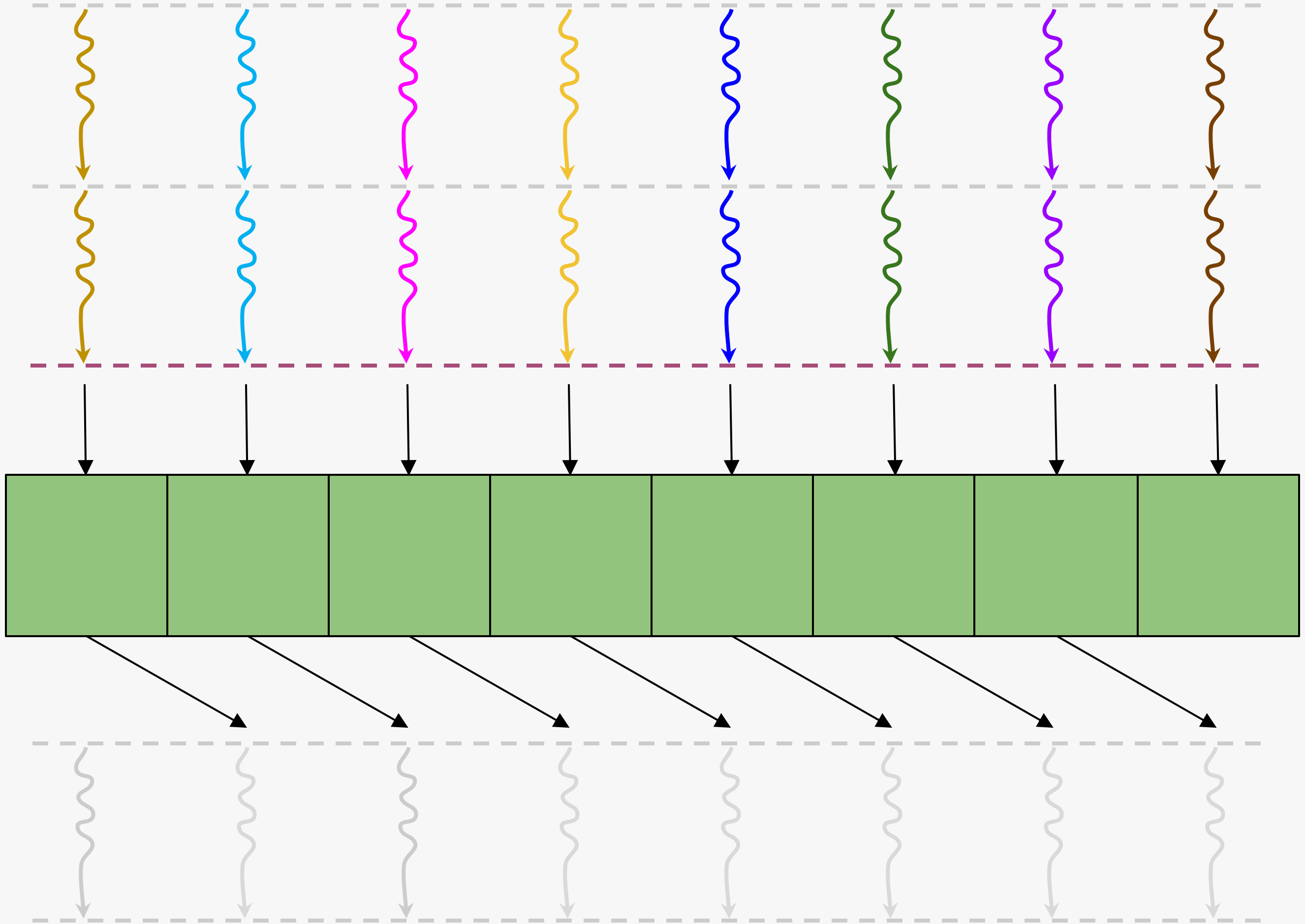
This means that it’s possible for a work-item to read a result that hasn’t yet been written to yet, you have a data race



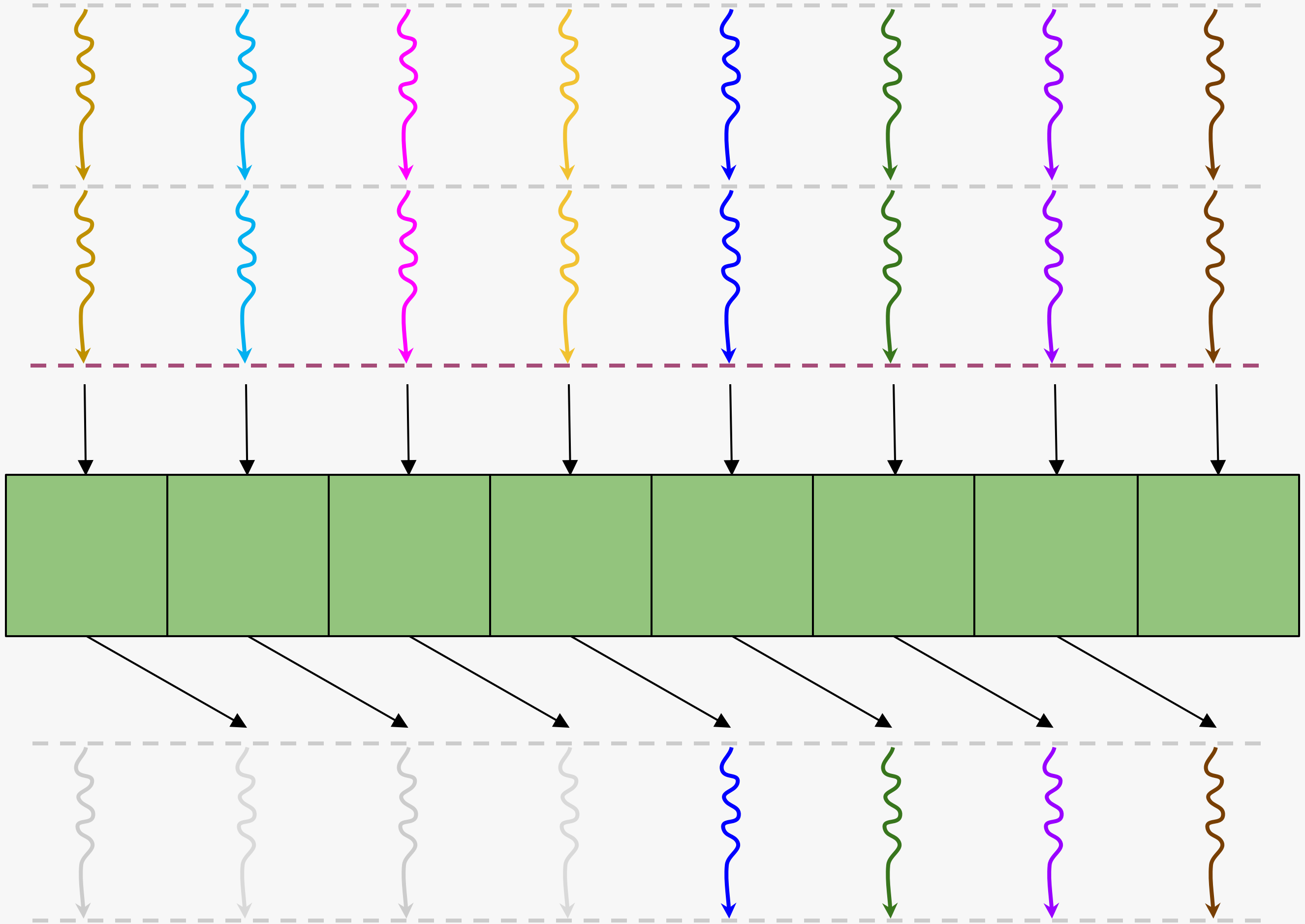
This problem can be solved by a synchronisation primitive called a work-group barrier



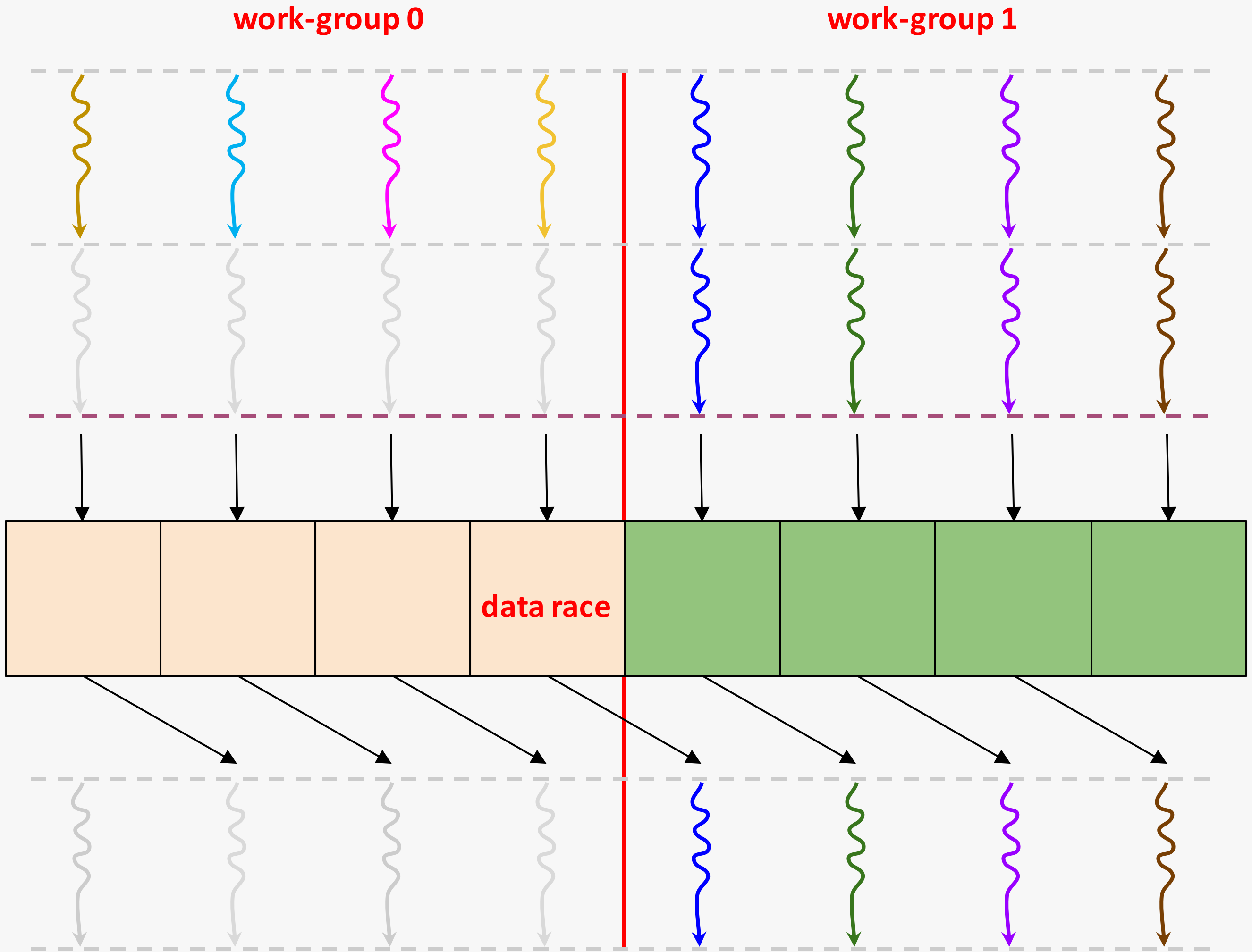
Work-items will block until all work-items in the work-group have reached that point



Work-items will block until all work-items in the work-group have reached that point



So now you can be sure that all of the results that you want to read from have been written to



However this does not apply across work-group boundaries, and you have a data race again

Choosing an good work-group size

- The occupancy of a kernel can be limited by a number of factors of the GPU
  - *Total number of processing elements*
  - *Total number of compute units*
  - *Total registers available to the kernel*
  - *Total local memory available to the kernel*
- You can query the preferred work-group size once the kernel is compiled
  - *However this is not guaranteed to give you the best performance*
- It's good practice to benchmark various work-group sizes and choose the best

Conclusions

Takeaways

- Identify which parts of your code to offload and which algorithms to use
  - *Look for hotspots in your code that are bottlenecks*
  - *Identify opportunity for parallelism*
- Optimising GPU programs means maximising throughput
  - *Maximize compute operations*
  - *Minimise time spent on memory operations*
- Use profilers to analyse your GPU programs and consult optimisation guides

Further tips

- Use profiling tools to gather more accurate information about your programs
  - *SYCL provides kernel profiling*
  - *Most OpenCL implementations provide proprietary profiler tools*

- Follow vendor optimisation guides
  - *Most OpenCL vendors provide optimisation guides that detail recommendations on how to optimise programs for their respective GPU*

SYCL for Nvidia GPUs

SYCL on non-OpenCL backends?

- SYCL 1.2/1.2.1 was designed for OpenCL 1.2
- Some implementations are supporting non-OpenCL backends (ROCm, OpenMP)
- So what other backends could SYCL be a high level model for?

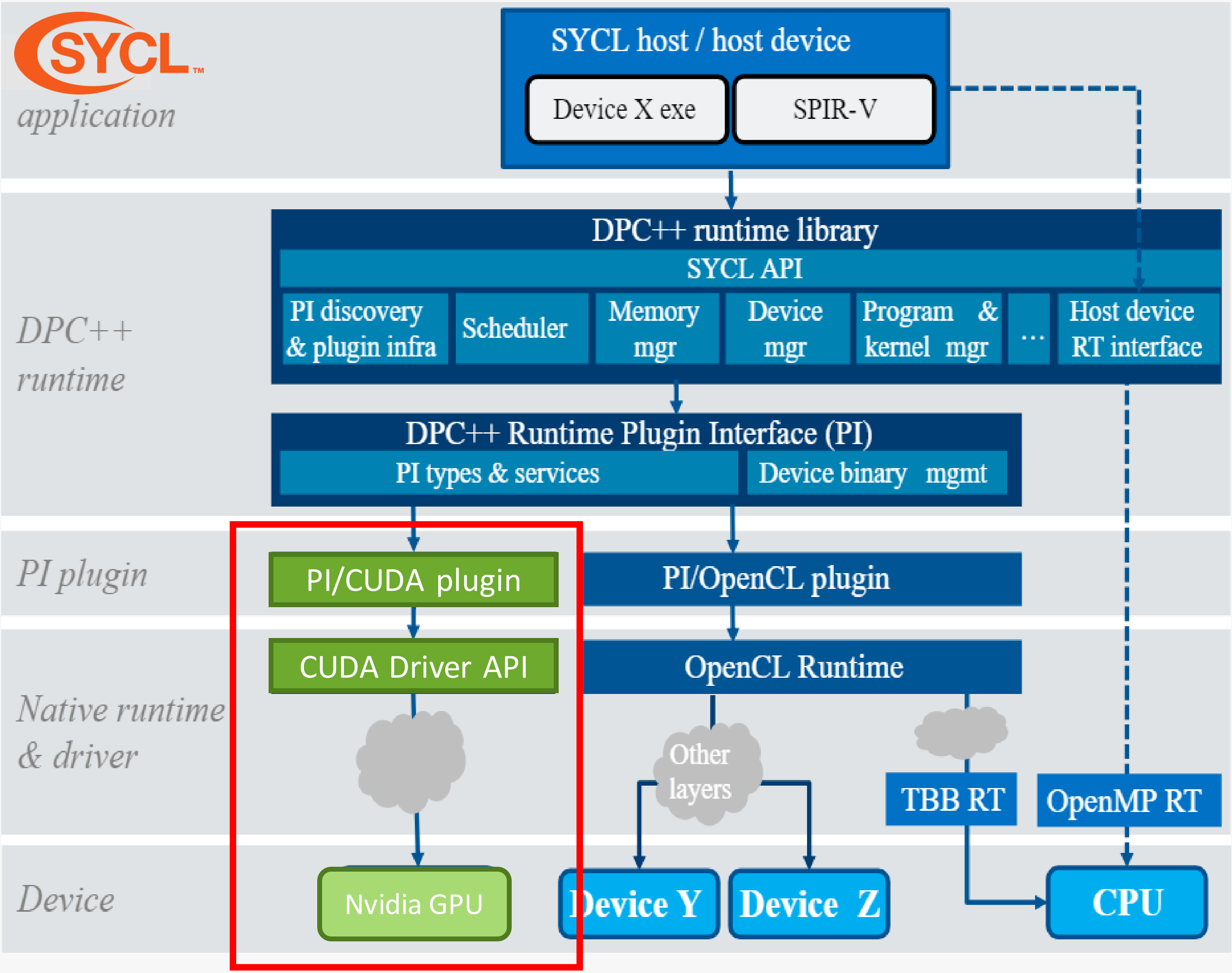
What about CUDA?

- Support for Nvidia GPUs is probably one of the most requested features from SYCL application developers
- Existing OpenCL + PTX path for Nvidia GPUs in ComputeCpp (still experimental)
- Native CUDA support is better for expanding the SYCL ecosystem

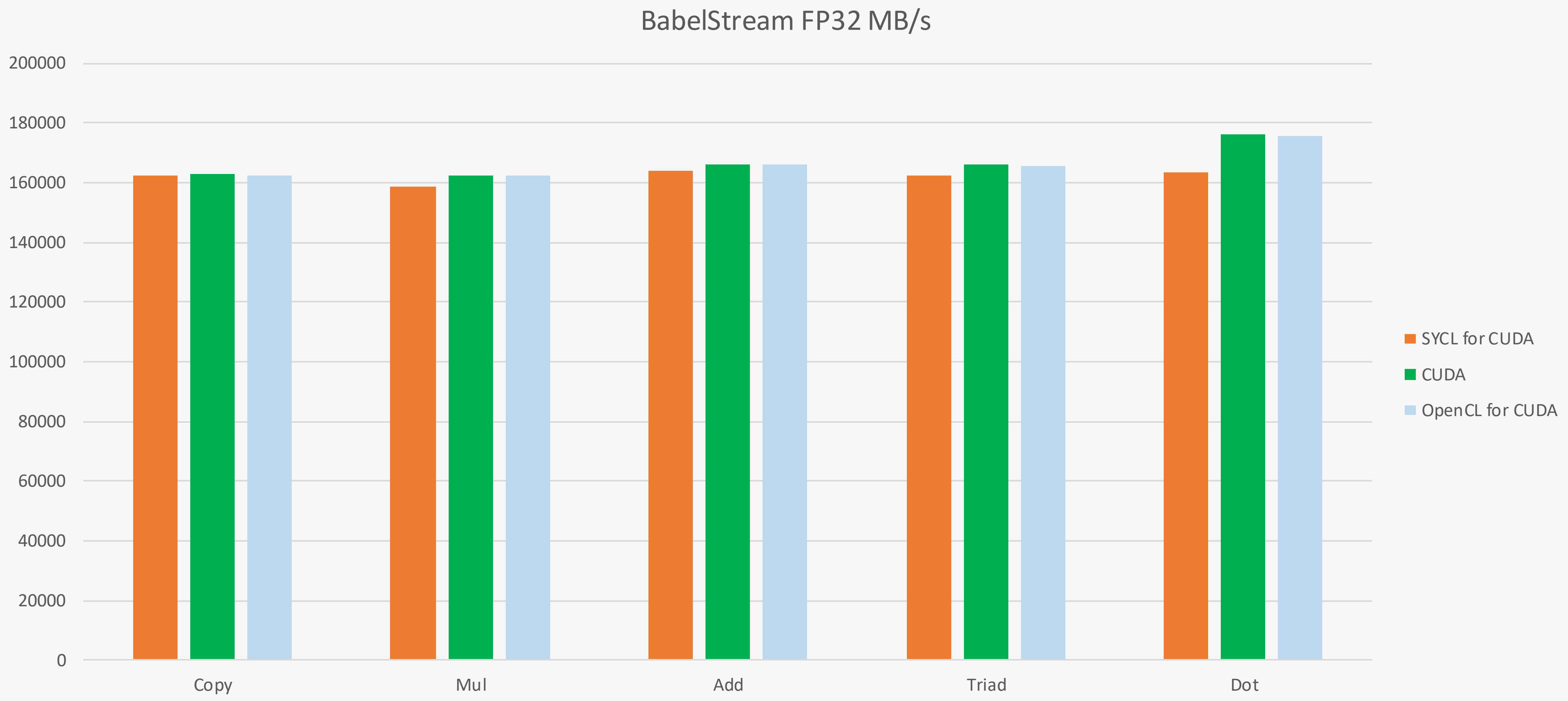
DPC++ is an open-source SYCL implementation

Has various extensions to the SYCL 1.2.1 API

Also provides a plugin interface (PI) to extend it for other backends



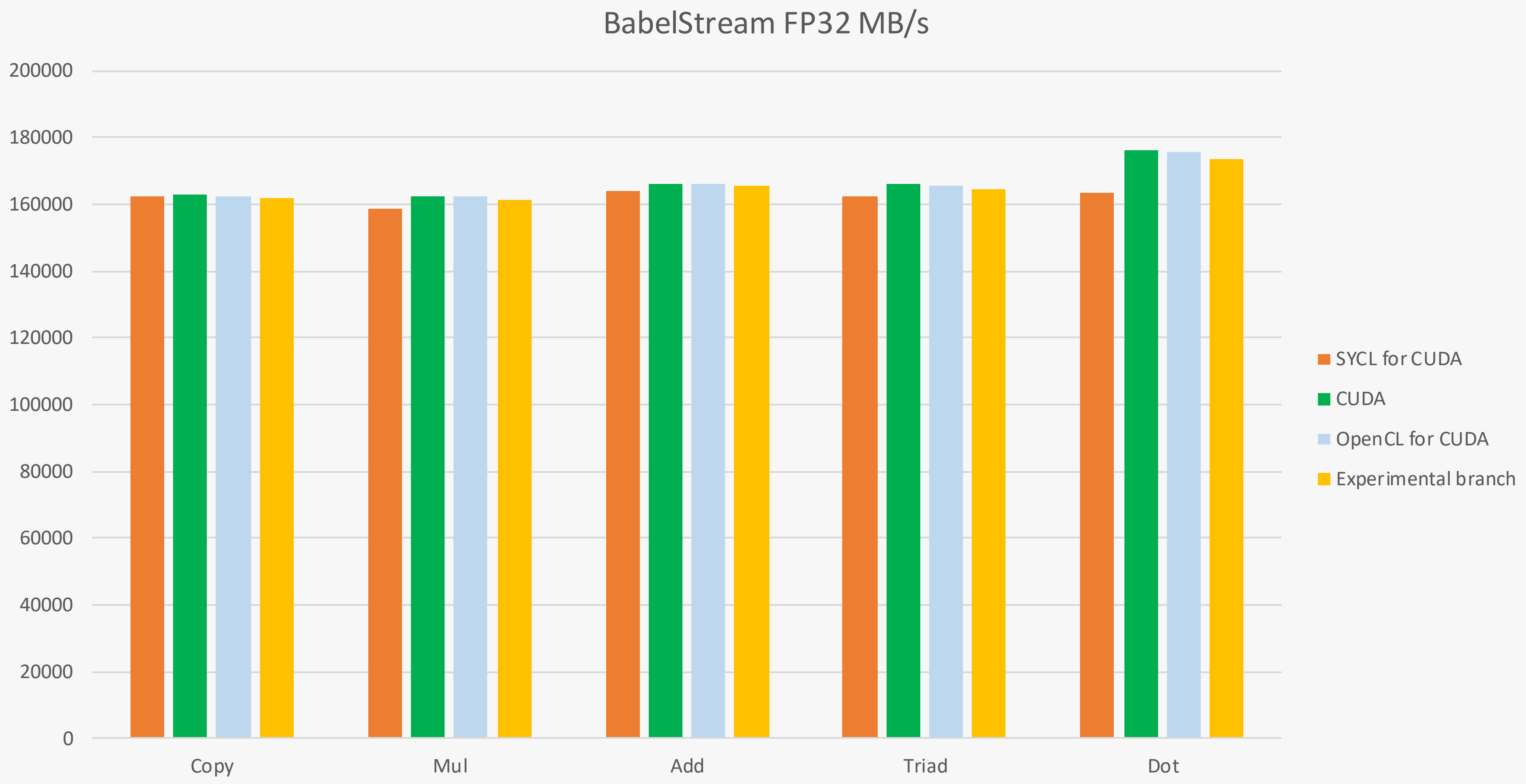
Preliminary performance results



http://uob-hpc.github.io/BabelStream

Platform: CUDA 10.1 on GeForce GTX 980

Preliminary performance results

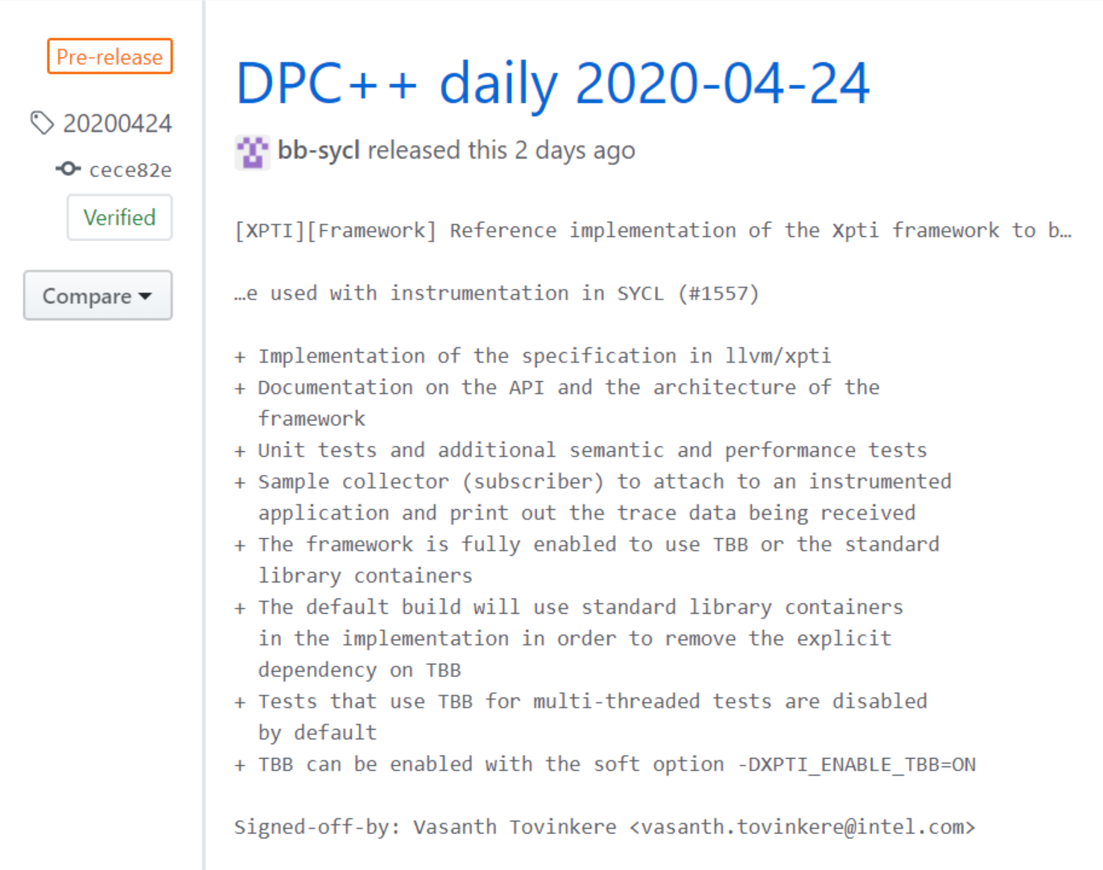


http://uob-hpc.github.io/BabelStream

Platform: CUDA 10.1 on GeForce GTX 980

How to use it?

- First build or download a binary package of DPC++
  - Nvidia support is now available in DPC++
  - There daily and more stable monthly releases
  - Release packages:
    - https://github.com/intel/llvm/releases
  - Detailed introductions:
    - https://github.com/intel/llvm/blob/sycl/sycl/doc/GetStartedGuide.md



Pre-release

20200424

cece82e

Verified

Compare

DPC++ daily 2020-04-24

bb-sycl released this 2 days ago

```
[XPTI][Framework] Reference implementation of the Xpti framework to b…

…e used with instrumentation in SYCL (#1557)

+ Implementation of the specification in llvm/xpti
+ Documentation on the API and the architecture of the
  framework
+ Unit tests and additional semantic and performance tests
+ Sample collector (subscriber) to attach to an instrumented
  application and print out the trace data being received
+ The framework is fully enabled to use TBB or the standard
  library containers
+ The default build will use standard library containers
  in the implementation in order to remove the explicit
  dependency on TBB
+ Tests that use TBB for multi-threaded tests are disabled
  by default
+ TBB can be enabled with the soft option -DXPTI_ENABLE_TBB=ON

Signed-off-by: Vasanth Tovinkere <vasanth.tovinkere@intel.com>
```

- Then compile you SYCL application with the DPC++ compiler using the CUDA triple

```
clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda-sycldevice sample.cpp -o sample
```

- Then enable the CUDA backend in the SYCL runtime by setting the environment variable

```
SYCL_BE=PI_CUDA ./sample
```

- And that’s it…
- Make sure to use a device selector in your application that will choose an Nvidia device
- Using both the OpenCL backend and the CUDA backend at the same time is currently not supported

SYCL 2020 preview

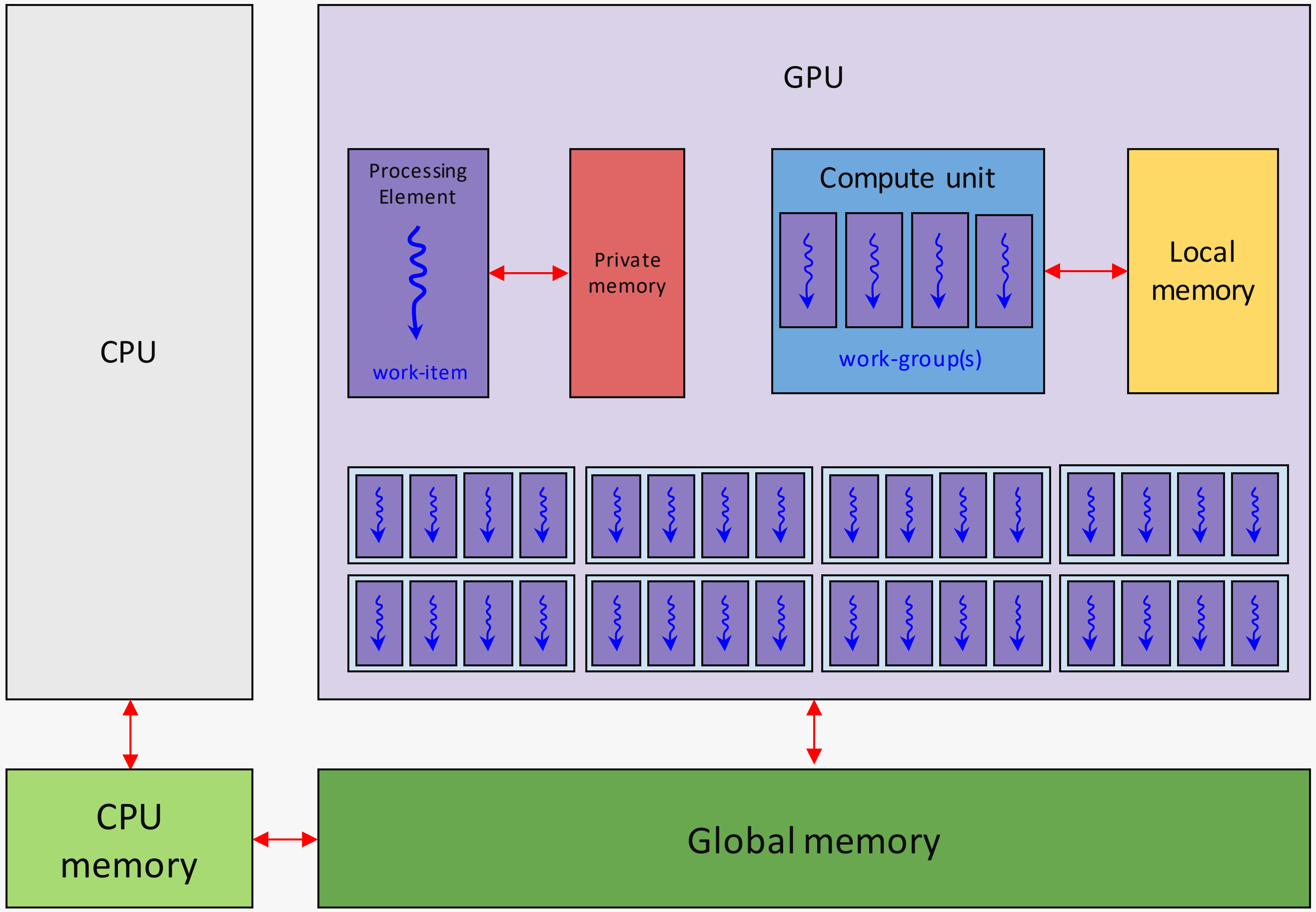

UNDER
CONSTRUCTION

SYCL™ 2020

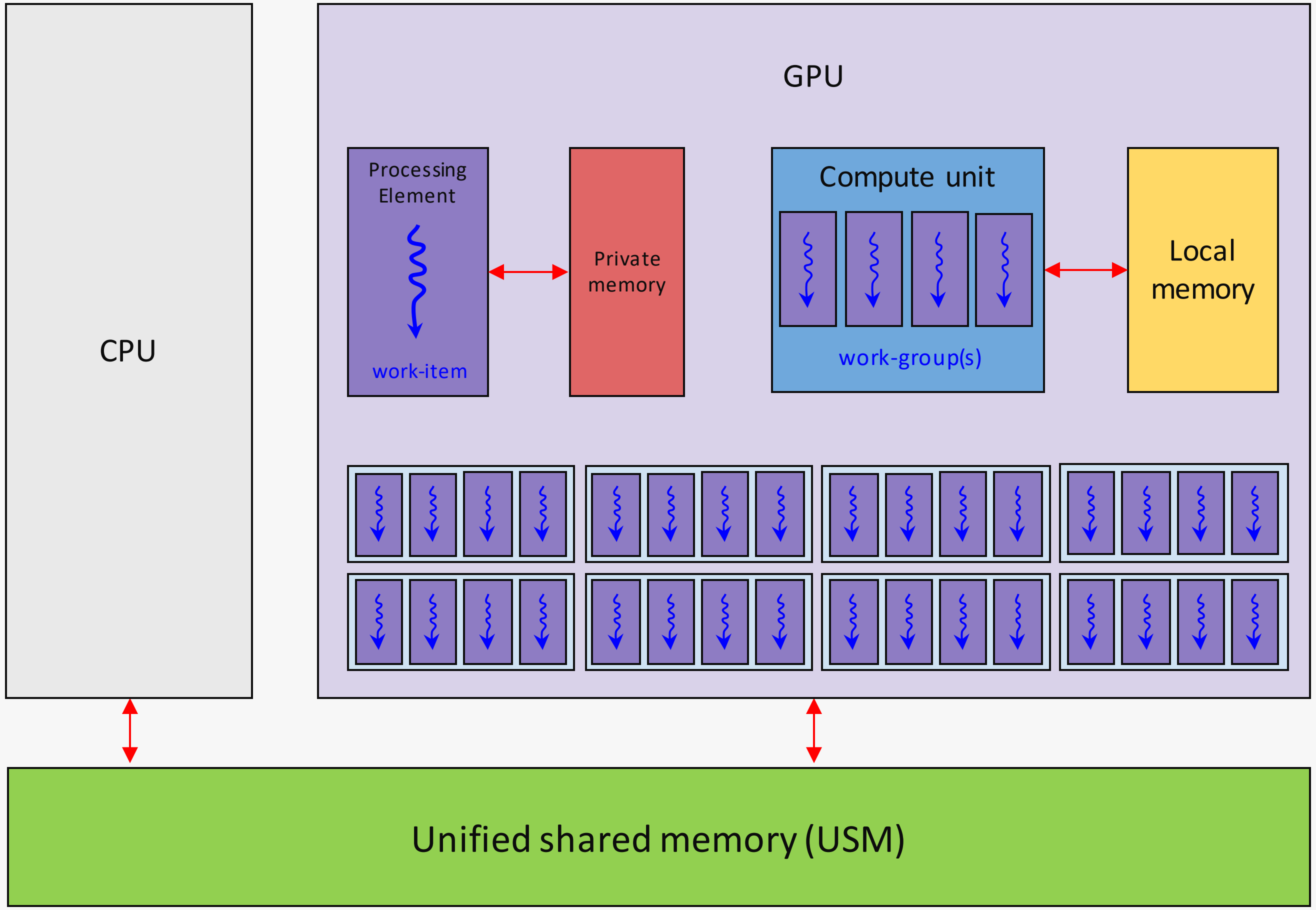
- Backend generalization
- Modules
- Specialization constants
- Unified shared memory
- In-order queues
- Sub-groups
- Group algorithms
- Host tasks
- Improved address space inference

Indicative only, still subject to change!

Unified Shared Memory



CPU
GPU
Processing Element
work-item
Private memory
Compute unit
work-group(s)
Local memory
CPU memory
Global memory



Unified shared memory allows the host CPU and the GPU to access a shared address space

This means a pointer allocated on the host CPU can be dereferenced on the GPU

	Explicit USM (minimum)	Restricted USM (optional)	Concurrent USM (optional)	System USM (optional)
Consistent pointers	✓	✓	✓	✓
Pointer-based structures	✓	✓	✓	✓
Explicit data movement	✓	✓	✓	✓
Shared access	✗	✓	✓	✓
Concurrent access	✗	✗	✓	✓
System allocations (malloc/new)	✗	✗	✗	✓

```
#include <SYCL/sycl.hpp>
using namespace sycl;

int main(int argc, char *argv[]) {
  std::vector dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector_v};
  auto context = gpuQueue.get_context();













}
```

If we take our example from earlier

```
#include <SYCL/sycl.hpp>
using namespace sycl;

int main(int argc, char *argv[]) {
  std::vector dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector_v};
  auto context = gpuQueue.get_context();

  auto inA = malloc_device<float>(dA.size(), gpuQeueue);
  auto inB = malloc_device<float>(dA.size(), gpuQeueue);
  auto out = malloc_device<float>(dA.size(), gpuQeueue);

}
```

With the USM explicit data movement model we can allocate memory on the device by calling malloc_device

This pointer will be consistent across host and device, but only dereferenceable on the device

```
#include <SYCL/sycl.hpp>
using namespace sycl;

int main(int argc, char *argv[]) {
  std::vector dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector_v};
  auto context = gpuQueue.get_context();

  auto inA = malloc_device<float>(dA.size(), gpuQeueue);
  auto inB = malloc_device<float>(dA.size(), gpuQeueue);
  auto out = malloc_device<float>(dA.size(), gpuQeueue);

  auto bytes = dA.size() * sizeof(float);

  gpuQueue.memcpy(inA, dA.data(), bytes).wait();
  gpuQueue.memcpy(inB, dB.data(), bytes).wait();

}
```

Now using the queue we can copy from the input std::vector objects initialized on the host to the device memory allocations by calling memcpy

Since these are asynchronous operations they return events, which can be used to synchronise with the completion of the copies

In this case we just wait immediately by calling wait

```
#include <SYCL/sycl.hpp>
using namespace sycl;

int main(int argc, char *argv[]) {
  std::vector dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector_v};
  auto context = gpuQueue.get_context();

  auto inA = malloc_device<float>(dA.size(), gpuQeueue);
  auto inB = malloc_device<float>(dA.size(), gpuQeueue);
  auto out = malloc_device<float>(dA.size(), gpuQeueue);

  auto bytes = dA.size() * sizeof(float);

  gpuQueue.memcpy(inA, dA.data(), bytes).wait();
  gpuQueue.memcpy(inB, dB.data(), bytes).wait();

  gpuQueue.parallel_for(range(dA.size()),
    [=](id i){ out[i] = inA[i] + inB[i]; });
  }).wait();

}
```

We can invoke a SYCL kernel function in the same way as before using command groups

However, here we are using one the new shortcut member functions of the queue

Again this operation is asynchronous so we wait on the returned event

```
#include <SYCL/sycl.hpp>
using namespace sycl;

int main(int argc, char *argv[]) {
  std::vector dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector_v};
  auto context = gpuQueue.get_context();

  auto inA = malloc_device<float>(dA.size(), gpuQeueue);
  auto inB = malloc_device<float>(dA.size(), gpuQeueue);
  auto out = malloc_device<float>(dA.size(), gpuQeueue);

  auto bytes = dA.size() * sizeof(float);

  gpuQueue.memcpy(inA, dA.data(), bytes).wait();
  gpuQueue.memcpy(inB, dB.data(), bytes).wait();

  gpuQueue.parallel_for(range(dA.size()),
    [=](id i){ out[i] = inA[i] + inB[i]; });
  }).wait();

  gpuQueue.memcpy(dO.data(), out, bytes).wait();

}
```

Finally we can copy from the device memory allocation to the output std::vector by again calling memcpy

And just as we did for the copies to the device we call wait on the returned event

```
#include <SYCL/sycl.hpp>
using namespace sycl;

int main(int argc, char *argv[]) {
  std::vector dA{ … }, dB{ … }, dO{ … };

  queue gpuQeueue{gpu_selector_v};
  auto context = gpuQueue.get_context();

  auto inA = malloc_device<float>(dA.size(), gpuQeueue);
  auto inB = malloc_device<float>(dA.size(), gpuQeueue);
  auto out = malloc_device<float>(dA.size(), gpuQeueue);

  auto bytes = dA.size() * sizeof(float);

  gpuQueue.memcpy(inA, dA.data(), bytes).wait();
  gpuQueue.memcpy(inB, dB.data(), bytes).wait();

  gpuQueue.parallel_for(range(dA.size()),
    [=](id i){ out[i] = inA[i] + inB[i]; });
  }).wait();

  gpuQueue.memcpy(dO.data(), out, bytes).wait();

  free(inA, context);
  free(inB, context);
  free(out, context);
}
```

Once we are finished with the device memory allocations we can free them

There is also a usm_allocator available

Getting started with SYCL

SYCL specification: khronos.org/registry/SYCL

SYCL news: sycl.tech

SYCL Academy: github.com/codeplaysoftware/syclacademy

ComputeCpp: computecpp.com

DPC++: github.com/intel/llvm/releases

hipSYCL: https://github.com/illuhad/hipSYCL

We're Hiring!
codeplay.com/careers/

codeplay®
THE HETEROGENEOUS SYSTEMS EXPERTS

Thank you

@codeplaysoft

/codeplaysoft

codeplay.com