Efficient GPU Programming in Modern C++

Gordon Brown
Principal Software Engineer, SYCL & C++

CppCon 2019 – Sep 2019
A Modern C++ Programming Model for GPUs using Khronos SYCL

Gordon Brown, Michael Wong

CppCon 2018
This talk is based on the SYCL programming model

Terminology may differ for other programming models
Agenda

Why use the GPU?

Brief introduction to SYCL

SYCL programming model

Optimising GPU programs

- Choosing the right algorithm
- Basic GPU programming principles
- Ideas for further optimisations
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 9.5 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
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

C++AMP
SYCL
CUDA Agency
Kokkos
HPX
Raja
Brief introduction to SYCL
Cross-platform, single-source, high-level, C++ programming layer
Built on top of OpenCL and based on standard C++11
Delivering a heterogeneous programming solution for C++
Applications

C++ template libraries

SYCL for OpenCL

OpenCL

OpenCL-enabled devices

Device compiler

Device IR (SPIR, SPIR-V, etc)

Host compiler
__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);

vector<float> a, b, c;
#pragma parallel_for
dataAGMA::parallel_for_each(e, [=](index<2> idx) restrict(amp) {
    c[idx] = a[idx] + b[idx];
});
int main(int argc, char *argv[]) {

}
```cpp
#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 gpuQueue(gpu_selector{});

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

    });

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

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

    queue gpuQueue(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()));

    defaultQueue.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 gpuQueue{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()));

        defaultQueue.submit([&](handler &cgh) {
            ...
        });
    }
}

Buffers synchronize on destruction via RAII waiting for any command groups that need to write back to it
```cpp
#include <CL/sycl.hpp>
using namespace cl::sycl;

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

    queue gpuQueue{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()));

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

class add;

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

    queue gpuQueue(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()));

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

class add;

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

    queue gpuQueue = 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()));

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

class add;

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

    queue gpuQueue(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()));

        defaultQueue.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]; });
        });
    }
}
```

The rest of this talk will focus on kernels and how to optimize them.
SYCL programming model
1. A processing element executes a single work-item
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. 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
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. 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
4. Each work-item can access the local memory of their work-group, a dedicated memory region for each compute unit
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. 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
4. Each work-item can access the local memory of their work-group, a dedicated memory region for each compute unit
5. A device can execute multiple work-groups
1. A processing element executes a single work-item
2. Each work-item can access private memory, a dedicated memory region for each processing element
3. 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
4. Each work-item can access the local memory of their work-group, a dedicated memory region for each compute unit
5. A device can execute multiple work-groups
6. Each work-item can access global memory, a single memory region available to all processing elements
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 mostly 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
  ○ *Analyse -> Parallelise -> Optimise -> Deploy*

➢ 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
As a motivational example we will be looking at an image convolution

- The image convolution algorithm is “embarrassingly parallel”
  - Each item in the computation can be calculated entirely independently

- The image convolution algorithm is very computation heavy
  - A large number of operations have to be calculated for each item in the computation, particularly when using larger filters

- Image processing requires a large bandwidth
  - A lot of data must be passed through the GPU to process an image, particularly if the image is very high resolution
Approximate gaussian blur 3x3
Basic GPU programming principles
Optimizing GPU programs means maximizing throughput

- 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();
}
...
cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange),
  [=](cl::sycl::nd_item<2> item) {

  int rowOffset = item.get_global_id(0) * WIDTH * NUM_CHANNELS;
  int my = NUM_CHANNELS * item.get_global_id(1) + rowOffset;

  int fIndex = 0;
  float sumR = 0.0f, sumG = 0.0f, float sumB = 0.0f, float sumA = 0.0f;

  for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++) {
    int curRow = my + r * (WIDTH * NUM_CHANNELS);
    for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE;
        c++, fIndex += NUM_CHANNELS) {
      int offset = c * NUM_CHANNELS;

      sumR += inputAcc[curRow + offset] * filterAcc[fIndex];
      sumG += inputAcc[curRow + offset + 1] * filterAcc[fIndex + 1];
      sumB += inputAcc[curRow + offset + 2] * filterAcc[fIndex + 2];
      sumA += inputAcc[curRow + offset + 3] * filterAcc[fIndex + 3];
    }
  }

  outputAcc[my] = sumR;
  outputAcc[my + 1] = sumG;
  outputAcc[my + 2] = sumB;
  outputAcc[my + 3] = sumA;
});

First we calculate the linear position of the data element within global memory relative to the current work-item.
cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange), [=](cl::sycl::nd_item<2> item) {

    int rowOffset = item.get_global_id(0) * WIDTH * NUM_CHANNELS;
    int my = NUM_CHANNELS * item.get_global_id(1) + rowOffset;

    int fIndex = 0;
    float sumR = 0.0f, sumG = 0.0f, float sumB = 0.0f, float sumA = 0.0f;

    for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++) {
        int curRow = my + r * (WIDTH * NUM_CHANNELS);
        for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE; c++, fIndex += NUM_CHANNELS) {
            int offset = c * NUM_CHANNELS;

            sumR += inputAcc[curRow + offset] * filterAcc[fIndex];
            sumG += inputAcc[curRow + offset + 1] * filterAcc[fIndex + 1];
            sumB += inputAcc[curRow + offset + 2] * filterAcc[fIndex + 2];
            sumA += inputAcc[curRow + offset + 3] * filterAcc[fIndex + 3];
        }
    }

    outputAcc[my] = sumR;
    outputAcc[my + 1] = sumG;
    outputAcc[my + 2] = sumB;
    outputAcc[my + 3] = sumA;
});

Then we loop over each element in the filter, incrementing an offset as we go
cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange),
  [=](cl::sycl::nd_item<2> item) {

  int rowOffset = item.get_global_id(0) * WIDTH * NUM_CHANNELS;
  int my = NUM_CHANNELS * item.get_global_id(1) + rowOffset;

  int fIndex = 0;
  float sumR = 0.0f, sumG = 0.0f, float sumB = 0.0f, float sumA = 0.0f;

  for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++) {
    int curRow = my + r * (WIDTH * NUM_CHANNELS);
    for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE;
         c++, fIndex += NUM_CHANNELS) {
      int offset = c * NUM_CHANNELS;

      sumR += inputAcc[curRow + offset] * filterAcc[fIndex];
      sumG += inputAcc[curRow + offset + 1] * filterAcc[fIndex + 1];
      sumB += inputAcc[curRow + offset + 2] * filterAcc[fIndex + 2];
      sumA += inputAcc[curRow + offset + 3] * filterAcc[fIndex + 3];
    }
  }

  outputAcc[my] = sumR;
  outputAcc[my + 1] = sumG;
  outputAcc[my + 2] = sumB;
  outputAcc[my + 3] = sumA;
});
cgh.parallel_for<native>(cl::sycl::nd_range<2>(globalRange, localRange),
 [=](cl::sycl::nd_item<2> item) {

    int rowOffset = item.get_global_id(0) * WIDTH * NUM_CHANNELS;
    int my = NUM_CHANNELS * item.get_global_id(1) + rowOffset;

    int fIndex = 0;
    float sumR = 0.0f, sumG = 0.0f, float sumB = 0.0f, float sumA = 0.0f;

    for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++) {
        int curRow = my + r * (WIDTH * NUM_CHANNELS);
        for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE;
             c++, fIndex += NUM_CHANNELS) {

            int offset = c * NUM_CHANNELS;

            sumR += inputAcc[curRow + offset] * filterAcc[fIndex];
            sumG += inputAcc[curRow + offset + 1] * filterAcc[fIndex + 1];
            sumB += inputAcc[curRow + offset + 2] * filterAcc[fIndex + 2];
            sumA += inputAcc[curRow + offset + 3] * filterAcc[fIndex + 3];
        }
    }

    outputAcc[my] = sumR;
    outputAcc[my + 1] = sumG;
    outputAcc[my + 2] = sumB;
    outputAcc[my + 3] = sumA;
});

Finally we write out the sums to global memory again
Kernel time (ms)

Image conv
512x512 source image
Intel HD Graphics 530

- Naive
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]);
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
Here data elements are accessed in row-major and work-items are executed in row-major.

Global memory access is coalesced.
If the work-items were executed in column-major:

Global memory access is no longer coalesced.

```
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id1 * 4) + id0;
a[linearId] = f();
```
However if you were to switch the data access pattern to column-major:

```
auto id0 = get_global_id(0);
auto id1 = get_global_id(1);
auto linearId = (id0 * 4) + id1;
a[linearId] = f();
```

Global memory access is coalesced again.
cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange), [=](cl::sycl::nd_item<2> item) {

    int rowOffset = item.get_global_id(1) * WIDTH * NUM_CHANNELS;
    int my = NUM_CHANNELS * item.get_global_id(0) + rowOffset;

    int fIndex = 0;
    float sumR = 0.0f, sumG = 0.0f, float sumB = 0.0f, float sumA = 0.0f;

    for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++) {
        int curRow = my + r * (WIDTH * NUM_CHANNELS);
        for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE;
            c++, fIndex += NUM_CHANNELS) {
            int offset = c * NUM_CHANNELS;

            sumR += inputAcc[curRow + offset] * filterAcc[fIndex];
            sumG += inputAcc[curRow + offset + 1] * filterAcc[fIndex + 1];
            sumB += inputAcc[curRow + offset + 2] * filterAcc[fIndex + 2];
            sumA += inputAcc[curRow + offset + 3] * filterAcc[fIndex + 3];
        }
    }

    outputAcc[my] = sumR;
    outputAcc[my + 1] = sumG;
    outputAcc[my + 2] = sumB;
    outputAcc[my + 3] = sumA;
});

Reversing the global ids will flip the linearization from row-major to column-major

Whether column-major or row-major linearization is more efficient depends on the device you are on
Kernel time (% of base)

Image conv
512x512 source image
Intel HD Graphics 530

Naive Coalesced

<table>
<thead>
<tr>
<th>Kernel Size</th>
<th>Naive</th>
<th>Coalesced</th>
</tr>
</thead>
<tbody>
<tr>
<td>3x3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5x5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7x7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9x9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11x11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Make use of vector operations

- GPUs are vector processors
  - Each processing element is capable of wide instructions which can operate on multiple elements of data at once

- Many compilers can auto-vectorise
  - This can affect the amount of performance gain you may see in vectorising your kernels
float rS, gS, bS, aS;
float r1, g1, b1, a1;
float r2, g2, b2, a2;

...  

rS = r1 + r2;
gS = g1 + g2;
bS = b1 + b2;
aS = a1 + a2;
```cpp
cl::sycl::float4 vS;
cl::sycl::float4 v1;
cl::sycl::float4 v2;
...

rS = v1 + v2;
```
cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange),
[](cl::sycl::nd_item<2> item) {

    int rowOffset = item.get_global_id(1) * WIDTH;
    int my = item.get_global_id(0) + rowOffset;

    int fIndex = 0;
    cl::sycl::float4 sum = cl::sycl::float4(0.0f);

    for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++) {
        int curRow = my + r * WIDTH
        for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE; c++) {
            sum += inputAcc[curRow + c] * filterAcc[fIndex];

            fIndex++;
        }
    }

    outputAcc[my] = sum;
});

To vectorise the kernel define all accessors in terms of SYCL vector types

This allows us to remove the calculations to factor in the number of channels

This also allows us to reduce the multiplications and assignments to single vector operators
Kernel time (% of base)

- Coalesced
- Vectorised

Image conv
512x512 source image
Intel HD Graphics 530
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*
Each item in the computation needs to read neighbouring elements

This means each element of data is read multiple times

- 3x3 filter: up to 9 ops
- 5x5 filter: up to 25 ops
- And so on...

If each of these operations loads from global memory this is can be very expensive
A common technique for using local memory is to break up your input into tiles.

Then each tile can be moved to local memory while the 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.
cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange), [=](cl::sycl::nd_item<2> item) {

    int globalRowOffset = item.get_global_id(1) * WIDTH;
    int global = item.get_global_id(0) + globalRowOffset;

    int localRowOffset = item.get_local_id(1) * WIDTH;
    int local = item.get_local_id(0) + localRowOffset;

    int fIndex = 0;
    cl::sycl::float4 sum = cl::sycl::float4{0.0f};

    copy_tile(scratchpad, inputAcc, local, global);
    item.barrier(cl::sycl::access::fence_space::local);

    for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++) {
        int curRow = local + r * WIDTH
        for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE; c++) {
            sum += scratchpad[curRow + c] * filterAcc[fIndex];

            fIndex++;
        }
    }

    outputAcc[global] = sum;
});

To use local memory we need to also calculate the linear position in the current work-group

We can then use this to copy a tile from global memory into the local memory of the current work-group

Now the multiply operators within the loop are reading from local memory
cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange),
[=](cl::sycl::nd_item<2> item) {

  int globalRowOffset = item.get_global_id(1) * WIDTH;
  int global = item.get_global_id(0) + globalRowOffset;

  int localRowOffset = item.get_local_id(1) * WIDTH;
  int local = item.get_local_id(0) + localRowOffset;

  int fIndex = 0;
  cl::sycl::float4 sum = cl::sycl::float4{0.0f};

  copy_tile(scratchspace, inputAcc, local, global);

  item.barrier(cl::sycl::access::fence_space::global_and_local);

  for (int r = -HALF_FILTER_SIZE; r <= HALF_FILTER_SIZE; r++) {
    int curRow = local + r * WIDTH
    for (int c = -HALF_FILTER_SIZE; c <= HALF_FILTER_SIZE; c++) {
      sum += scratchspace[curRow + c] * filterAcc[fIndex];

      fIndex++; 
    }
  }

  outputAcc[global] = sum;
});

Since we’re moving a tile into local memory and then performing operations on it there we need a barrier to ensure all elements of the tile are copied
Kernel time (% of base)

- **Image conv**
- **512x512 source image**
- **Intel HD Graphics 530**

---

- **Vectorised**
- **Local mem**

Dimensions:
- 3x3
- 5x5
- 7x7
- 9x9
- 11x11
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
Kernel time (% of base)

- Vectorised
- Local mem (8x8)
- Local mem (16x16)
Ideas for further optimisations
Use constant memory

➢ Some GPUs provide a region of global memory that is read-only
  ➢ *This can be faster to access as it doesn’t require caching*
Use texture memory

➢ Most GPUs have texture memory
  ➢ *This can be faster to access for data that is represented as pixels*
  ➢ *This also provides sampling operations*
Batch work together

➢ Hitting occupancy limitations of a GPU can lead to drops in performance gain
  ➢ *This is because single work-items are having to do more chunks of work*

➢ Batching work for each work-item allows reusing cached data
  ➢ *Batching work that share neighbouring data allows you to further share local memory and registers*
Use double buffering

➢ If you hit occupancy limitations you will have more tiles than can be computed at once
➢ *This means each work-group will compute more than one tile*
Overlapping copy and compute within kernels allows for better utilisation of GPU processing elements and therefore better throughput.
Loop unrolling

cgh.parallel_for<naive>(cl::sycl::nd_range<2>(globalRange, localRange),
[=](cl::sycl::nd_item<2> item) {

    int rowOffset = item.get_global_id(1) * WIDTH;
    int my = item.get_global_id(0) + rowOffset;

    int fIndex = 0;
    cl::sycl::float4 sum = cl::sycl::float4{0.0f};

    sum += inputAcc[(my - 1 * WIDTH) - 1] * filterAcc[0];
    sum += inputAcc[(my - 1 * WIDTH)] * filterAcc[1];
    sum += inputAcc[(my - 1 * WIDTH) + 1] * filterAcc[2];
    sum += inputAcc[(my * WIDTH) - 1] * filterAcc[3];
    sum += inputAcc[(my * WIDTH)] * filterAcc[4];
    sum += inputAcc[(my * WIDTH) + 1] * filterAcc[5];
    sum += inputAcc[(my + 1 * WIDTH) - 1] * filterAcc[6];
    sum += inputAcc[(my + 1 * WIDTH)] * filterAcc[7];
    sum += inputAcc[(my + 1 * WIDTH) + 1] * filterAcc[8];

    outputAcc[my] = sum;
});

➢ Here we unroll the loop over the filter
➢ *This allows the compiler more freedom in how it vectorises and allocates registers*

➢ However this does make the code more obfuscated and less flexible
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*
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
Thank you for listening