

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"

() codeplay[®]

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 (nonvectorised) 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











Ninine.	
Number of Street	
for the contract of the contra	
Contraction of the second s	Carrier and an and a second seco
building and the second s	and the stand of the second state of the secon
These	study, hart
No. of the Work	
Seat 1	
Contractions for the local spin of the same	
of the design of the second seco	the second se
MARCH 18	and a Red Sector St. Dates, Million St. & Bart
18"+ 8 800 \$18,75,854	C Alamber 1 Control of
construction descendence of the second se	state and a second second second second
derfrequences and	
tala ar dat inferencede beirgentingenen	the second se
the start and real states but	second of the second second from a filler
allocation and the form	
A DESCRIPTION OF A	parties had a set on the second set of a second set of a
of a fear of double babiet	and the second sec
bd gewan state	
the second react	The first sectors a back on the
the Clarge Haranat	stage bei the difference in the stage of the second s
segulation took too ad	Pileant
Countries of the party of	and the strength of the streng
Advent Road Classed Road	
Aparted and interactional	
Approximate and the second methods	Cond Transport
General sheat manufacture	Country of the state of the local field from the state
dans when another see	
Gently Dimensional adjustments	Standard Contraction of Contractiono
Gam Qi Chananana an ta Banda barta	day to the second se
A REAL TRACTORISES AND ADD.	the second se
field and services and the	
of the factor of the first of the factor of	calle - bits
40.4	and the second sec
ALCONE'S MARK	
Sector and the first an	Advertising the Deserve
A de la constante de la consta	A with the bullet as with the
Contrast Include, Bri	The second se
	profile a first is not complicate of the first of the
These arrests are to a	And a state of the
And and all the	
	Charles Transfer B. angles and Record and
AT utradit daritation	The same the began during the
to all filles with a state of the	by indicated a property of the style of the second frame
	A second s
and the property of the last on the property of	
and have a second se	Films - Provided you got Balance and
	A CONTRACTOR OF A CONTRACTOR O
Address of the second sec	1 Auto All Auto Auto Auto
talks/fileinblat	
Local A. P. Muse Report of Serve Public Intellig	prefike how to mention manufall
Tars Applied work has set	
and the state of t	
schooled, Minist	(Report and Fulling)
The second s	15/19 Appendix Inclassification
A CONTRACT OF A	course and and any segment to course and the second s
Print	and the hadry of the set of the
	the state of the s
and the different difficult lines in fam.	on hits for an inging all in condex part of a firmed
114-00-04FE	Fach Appendix

GPGPU programming was once a niche technology

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

curve



with a further and inter



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 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

alle in the second s		
AND AND AND		and the bold is all the improvemented if
	Characterization of the Contract of Contra	weißt
and the driver of the		
and the second sec	and the state of the second state of the secon	
	shafed water	ittamis is known Production if and in set & had
adv. Star F1		Sixtupfichered interview Transformers
#1 ⁻		pitch and
	constant de la const	en soften en differentiere besen de in alle 1.6 int Apiele it an 8.6 on 8 ant
and some first and an and some first and some first and		7m
A Amod han	1 Bud an apply and d	
A/ 81 18	and of an R and and a 18 A. Min on A.A. Market have	and The a field to an advise of 1.4 def engels and
And BERRINS, BRIDE, MALER AND	A familie (abatt, batt
	and The a Total is and an experimental inflations, whereas	
القرعمتق تعامرون بطراع فحاصت	shade have	a fed he i fe a a fine fra conchiferen
for all the second s		an auffinite contraction of a start of the first of a second of the seco
		7 m 1 4 3 48 48 1
far Wil BBI Hitsessells falsgassisassel	I Bud an owned when	weather both and advances being a street
he shifts and shaft disheaded	and a first for the second second from the first second seco	
	P(massis)	
and a part from:		
	and the fail is addressed as a different of the second state of th	and the second sec
a Consult of and free Reality of	study was	maint
or Comme scate		Tel les
and the state of the state		
and the second second	The second and s	have be closed and
and the second sec	the second se	
superior and Constructions' and		here Table T His Table (Jacob Hand I do Bis halo
		hards half Transchiller days of such that the factor of a set
	and the second sec	Terfinal W ** Francis and the feature and
the second s		
and a second second second second		adart
and and and a linearly and and	1949	and the Wild WILD WILd WILd To Call (Brittan Call)
and the lineshance	the second se	
And Boat Hannah had	Company and Address	
and it is a second with and a second	an fire	figure.com
and it is a second to be and the large	(A DOMESTIC ASSOCIATION OF A DOMESTIC ASSOCIATIONO ASSOCIATORIA ASSOCIATIONO ASSOCIATIONO ASSOCIATIONO ASSOCIATIONO ASSOCIATICO ASSOCIA	post 7 for \$40.0. We shall all the surf owners are highly assored assored.
and the second second second second		
and in the state of the family in the	and The a had been deeped and the block on advanced	Tolana Ta Follan
	official and the second s	affeder affere Mar (Kar)
Trive house & a Franks for also	and the state	affect of the March (A)
A	wheeld state	affente affent Mar 18,0
charaf a share		and the second sec
the Burnard and		affectuary and the feature of
Tarihand (Faithing his	C Read of Proceedings Come Programs	and a starting all
A delivered i Part New York, New .	No. 2010 All and a second s	("his dos wells werden owers b)
Table and Collections, Mar	A Depart 1 and a Department	offente admenta (merical)
	and The a Total to and cannot charaft all informations	
Noting of Personal West Read	shaded stars	
Contractifier team		
	Fland of the part is the segment of the concerning	
Taffander fylfam.	The same for the production of the last	
Contraction in the second s	and the second se	
a s a parter i s a s a s a s a s a s a s a s a s a s		
	the second se	
shaded, Start	a Read of Frankel and a State and an	
RECEIVED AND A CONTRACT OF	To an it with the second state with the second state to be the base	
		Jacksf all
Brideldow	Construction of the second sec	and the second s
Calling & Robinstein School		galante .
and Pales Barrel (Into Pales 1 and	and The a field in the block have as an of all	and the second s
in the appropriate for the first state of the	100 C	and surged at and [
		A CONTRACTOR OF
and The a field in pit Franklow in References in a		an age (and) and (
shall hart	Efference in an Photoson	Art and
Proventie Contraction of the Con	a face is a second to see an an interest of the second sec	6/ as 15 \$3
	the difference in the second sec	
Trans Ma	and the ball of the ball of the section of the sect	
and a first and a fame and	and a station in party with a read (grow We)	
	and in the Barrier State of State and Canada May	
second in the Statement of State 2 and the State	to be the New Yorks, while conducting all all facents	
The second se		
/month_laffith	A Second Second	

Typical OpenCL hello world application

Inchesting Sport		

alarmit.		
distant with	and of Count St.	
off cale fast a dis	and all frequently in	
Weaterfall allo	and of Falsal Til	
A CONTRACTOR OF		
Advantación & Scient	adult adult	
Actor Solds & New	Adda (and and)	
Ballar Ballet Ballion	dida (and and)	
transition.		
stand and a state of a	manner and to	
100 10 10 100 100 100 100 100 100 100 1		
and the balance with	and the set of some line and the set of the	
water water	(alter)	
1		
10.		
122.02		

Typical SYCL hello world application

- SYCL provides highlevel abstractions over common boilerplate code
 - Platform/device selection
 - Buffer creation
 - Kernel compilation
 - Dependency management and scheduling



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 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 nonstandard backends

SYCL implementations





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

codeplay[®]



- 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{}};
```

<pre>buffer<float,< pre=""></float,<></pre>	1>	<pre>bufA(dA.data(),</pre>	<pre>range<1>(dA.size()));</pre>
<pre>buffer<float,< pre=""></float,<></pre>	1>	<pre>bufB(dB.data(),</pre>	range<1>(dB.size()));
<pre>buffer<float,< pre=""></float,<></pre>	1>	<pre>bufO(d0.data(),</pre>	range<1>(d0.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> buf0(d0.data(), range<1>(d0.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_	<pre>access<access::mode::read>(cgh);</access::mode::read></pre>
auto	inB	=	bufB.get_	_access <access::mode::read>(cgh);</access::mode::read>
auto	out	=	buf0.get	_access <access::mode::write>(cgh);</access::mode::write>

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 = buf0.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> buf0(d0.data(), range<1>(d0.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 = buf0.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> buf0(d0.data(), range<1>(d0.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 = buf0.get access<access::mode::write>(cgh);
      cqh.parallel for<add>(range<1>(dA.size()),
                                                                       This is the code which is
        [=] (id<1> i) { out[i] = inA[i] + inB[i]; });
                                                                        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

codeplay[®]



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 workgroups





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

codeplay[®]



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

codeplay[®]





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
 - 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



Basic GPU programming principles



Optimizing GPU programs means maximizing throughput





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();
```

🜔 codeplay®



ξ

🌔 codeplay®

a[globalId] = 0;	***	\$ }	₹ ↓	***	**	**	\$ }	***
if (globalId < 4) {	***	\$ \$ \$	\$	***	***	*	¥ ₹	*
a[globalId] = x();	****	\$	*	***	\$	S F	\$	\$
} else {	*	****	*	****	***	****	***	~~~
a[globalId] = y();	~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~	~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~	*	↓	*	***
}	***	***	***	***	*	***	*	***

🌔 codeplay®







...











...

...





codeplay^{*}

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];




• • •

f(a[globalId]);



f(a[globalId]);





• • •

f(a[globalId]);



100% global access utilisation



. . .

f(a[globalId * 2]);





• • •

50% global access utilisation

float data[size];

. . .

f(a[globalId * 2]);







Row-major

auto id0 = get global id(0);

auto id1 = get global id(1);

a[linearId] = f();

auto linearId = (id1 * 4) + id0;

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



<pre>auto id0 = get_global_id(0);</pre>
<pre>auto id1 = get_global_id(1);</pre>
auto linearId = $(id1 * 4) + id0;$
a[linearId] = f();

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

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



() codeplay[®]

Row-major



<pre>auto id0 = get_global_id(0);</pre>
<pre>auto id1 = get_global_id(1);</pre>
<pre>auto linearId = (id0 * 4) + id1;</pre>
a[linearId] = f();

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

Global memory access is coalesced again



Column-major

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

1	7	5	8	2	3	8	3	4	6	2	2	4	5	8	3	
1	3	4	3.	-2 _	4	3	4	5	6	1	6	5	7	8	5	
9	2	1	8	1	4	6	9	-5 -	1	4	5	1	9-	_4_	7	
3	6	2	0	2	2	9	8	2	7	9	4	2-	6	1	5	
1	7	2	2	8	4	6	8	4	7	6	8	3	2	4	1	
4	9	9	5	1	3	7	3	8	1	7	4	1	5	9	4	
4	0	6	-3-	- 6_	9	9	6	- 8	- 5_	9	9	0	2	1	5	
3	8	1	2	4	7	1	-7-	6	7	7	2	-6-	-3	6	7	
6	7	5	4	3	1	4	4	2	6	3	-0-	5	0	7	0	
1	3	4	2	2	8	1	6	4	9	5	3	7	1	2.	4	
7	5	4	3	7	0	4	0	3	0	4	4	2	8	9	0	
0	9	9	8	0	2	9	8	2	1	6	0	6	3	4	1	
6	4	0	1	9	1	7	4	8	3	0	5	0	2	0	6	
1	5	7	6	3	0	6	5	4	6	0	4	1	8	7	0	
3	3	0	5	9	8	2	4	7	1	5	2	0	4	9	7	
1	9	0	4	0	3	0	6	1	2	8	7	0	1	2	9	



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 workgroup 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

codeplay^{*}



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

© codeplay®

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 opensource SYCL implementation

Has various extensions to the SYCL 1.2.1 API

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



Ocodeplay*

Preliminary performance results



BabelStream FP32 MB/s

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

Platform: CUDA 10.1 on GeForce GTX 980



Preliminary performance results



BabelStream FP32 MB/s

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:
 - <u>https://github.com/intel/llvm/releases</u>
 - Detailed introductions:
 - <u>https://github.com/intel/llvm/blob/sycl/sycl/doc/GetStartedGuide.md</u>

Pre-release > 20200424 -• cece82e Verified

DPC++ daily 2020-04-24

🐮 bb-sycl released this 2 days ago



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

Compare 🔻

...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>

() codeplay[®]

• 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









Indicative only, still subject to change!


Unified Shared Memory





🌔 codeplay®



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

() codeplay[®]

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

```
#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

```
() codeplay<sup>*</sup>
```

```
#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 = qpuQueue.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(d0.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(d0.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

```
© codeplay<sup>®</sup>
```

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





Thank you



We're Hiringe coleptar.com/careers



@codeplaysoft

/codeplaysoft



www

codeplay.com