Receiving Data

- You receive (read) data from a UART by reading from its URXBUF register.
- You can read any time you want, but you might not get anything useful.
- The RDR (Receive Data Ready) bit indicates whether the URXBUF register contains valid data.
- Reading a character from URXBUF when RDR is 0 doesn't interfere with the current receive operation.
- Reading a character from URXBUF when RDR is 1 resets RDR to 0 and readies URXBUF to receive another transmission.

Receiving Data

- The `UART` class provides one input function:
  - `get()` returns an `int` whose value is either a character read from the UART, or -1 is no character is available.
- For example, if `port` is a reference to a `UART`, then:

```
int c;
while ((c = port.get()) < 0)
    ;
```

  waits for a valid character from the UART.
- Only the low-order byte of the URXBUF contains data.
  - The remaining bytes are unused.

Sending Data

- The `UART` class provides two output functions:
  - `ready_for_put()` returns true if the UART is ready to accept more output.
  - `put(c)` sends character `c` to the UART.
- For example, if `port` is a reference to a `UART`, then:

```
while (!port.ready_for_put())
    ;
port.put(c);
```

  waits for the UART to complete the previous transmission, and then initiates another.
- The UART transmits the low-order byte of UTXBUF.
  - It ignores the remaining bytes.

A UART Class in C++

```
class UART {
public:
    ~~~     // see the next few slides
private:
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    device_register UTXBUF;
    device_register URXBUF;
    device_register UBRDIV;
};
```

Controlling Transmission Speed

- These public members are for enabling and disabling the UART:

```
class UART {
public:
    ~~~
    enum mode { RXM = 1, TXM = 8 };
    void disable() { UCON = 0; }
    void enable() { UCON = RXM | TXM; }
    void init() { enable(); ~~~ }
    ~~~
};
```

- And these are the three i/o functions…

Controlling a UART

```
class UART {
public:
    ~~~
    int get() {
        return (USTAT & RDR) != 0 ?
            static_cast<int>(URXBUF) : -1;
    }
    bool ready_for_put() {
        return (USTAT & TBE) != 0;
    }
    void put(int c) {
        UTXBUF = static_cast<device_register>(c);
    }
    ~~~
};
```

Better I/O Functions

- UARTs are often used for sending and receiving character data.
- Unfortunately, our current UART interface requires that we use a cast to operate in terms of characters, as in:

```
int temp;
char c;
if ((temp = port.get()) != -1) {
    c = static_cast<char>(temp);
    ~~~
}
```

Better I/O Functions

- Here's a better set of UART functions:

```
bool get(char &c) {
    if ((USTAT & RDR) == 0)
        return false;
    c = static_cast<char>(URXBUF);
    return true;
}

void put(char c) {
    UTXBUF = c;
}
```

Better I/O Functions

- This `get` function mimics `std::istream::get`:
  - If it gets a character, it returns true and places the character value in a `char` (not an `int`) argument passed by reference.
  - Otherwise, it returns false.

```
UART &port = *new UART;
~~~
char c;
if (port.get(c)) {
    ~~~
}
```

Ensuring Proper Alignment

- Again, the UART 0 group consists of six device registers:

Offset	Register	Description
0xD000	ULCON	line control register
0xD004	UCON	control register
0xD008	USTAT	status register
0xD00C	UTXBUF	transmit buffer register
0xD010	URXBUF	receive buffer register
0xD014	UBRDIV	baud rate divisor register

- How can you be sure that each structure member is at the correct offset?
- Most compilers offer extensions to help you control structure layout…

Layout Guarantees

- For example, some compilers offer pragmas, as in:

```
#pragma pack(push, 4)
struct UART {
    ~~~
};
#pragma pack(pop)
```

- Some dialects of GNU C and C++ support type attributes such as:

```
struct UART __attribute__(packed) {
    ~~~
};
```

Using Static Assertions to Enforce Layout

- You can catch misalignments at compile time using static assertions, as in:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    ~~~
};
static_assert(
    offsetof(UART, UCON) == 0x04,
    "UCON register must be at offset 4 in UART"
);
~~~
```

Guaranteed Initialization

- A `UART` must be initialized by calling `init` before it can be used.
  - A user could easily forget to do so.

✓ *Interfaces should be:*

✓ ***easy to use correctly***

✓ ***hard to use incorrectly.***

- How can we make sure that initialization for a `UART` is guaranteed?

Constructors

- We could turn `init` into a constructor to ensure that it's always called when a `UART` is created:

```
class UART {
public:
    ~~~
    UART() { enable(); ~~~ }
    ~~~
};
```

- Unfortunately, this doesn't quite work with our definitions…

Constructors

- Memory-mapped objects aren't normal objects.
  - You don't (or at least shouldn't) define any objects of the `UART` type.
  - You just set up pointers or references to existing memory-mapped registers using `reinterpret_cast`.
- This leaves the compiler no opportunity to generate code for constructor calls as it normally would.
- However, you can construct a `UART` object at a memory-mapped location by using a placement new-expression…

Constructors and New-Expressions

- An expression such as:

```
p = new T (v);  // (v) is optional
```

translates into something (sort of) like:

```
p = static_cast<T *>(operator new(sizeof(T)));
p->T(v);    // explicit constructor "call"
```

- The expression `p->T(v)` is a notation for "apply the `T` constructor that accepts argument `v` to the storage addressed by `p`".
  - It doesn't actually compile.
- If the new-expression lacks an initializer such as `(v)`, it uses `T`'s default constructor.

Constructors and Placement New

- The C++ Standard Library provides a version of `operator new` that you can use to "place" an object at a specified location.
- The function is defined as just:

```
void *operator new(std::size_t, void *p) throw () {
    return p;
}
```

- It ignores its first parameter and simply returns its second.
- The empty exception-specification, `throw()`, indicates that the function won't propagate any exceptions.
- This `operator new` is often an inline function, so the compiler can optimize calling the function down to just copying a pointer.

Constructors and Placement New

- A *placement new-expression* such as:

```
p = new (region) T (v);      // (v) is optional
```

translates into something along the lines of:

```
p = static_cast<T *>(operator new(sizeof(T), region));
p->T(v);
```

- In effect, it just constructs a T object in the storage addressed by region.
  - The first line of the translation assigns p the address returned from operator new, namely the value of region.
  - The second line constructs a T object at that address.

UART with Placement New

- Using a placement new-expression, the definition for `UART0` now looks like:

```
UART *const UART0 =
    new(reinterpret_cast<void *>(0x03FFD000)) UART;
```

- As a reference, it looks like:

```
UART &UART0 =
    *new(reinterpret_cast<void *>(0x03FFD000)) UART;
```

Class-Specific New

- C++ lets you declare operator new as a class member.
- If T is a class type with a member operator new, then a new-expression such as:

```
p = new T (v);  // (v) is optional
```

  uses T's operator new rather than the global operator new.
- New-expressions for all other types continue to use the global operator new.

Class-Specific New

- You can implement an operator new as a class member that places a device at a specified memory-mapped address:

```
class UART {
public:
    void *operator new(std::size_t) {
        return reinterpret_cast<void *>(UART0_address);
    }
    ~~~
private:
    enum { UART0_address = 0x03FFD000 };
    ~~~
};
```

Class-Specific New

- Then you can create a fully constructed UART object at the desired memory-mapped location using just:

```
UART &UART0 = *new UART;
```

- This new-expression:
  - uses the UART's operator new to "place" the UART object in its memory-mapped location, and
  - uses the UART's default constructor to initialize the object.
- A member operator new is always a static member, even if not declared so explicitly.
- Unfortunately, this only works for UART 0…

Class-Specific New

- How can we make it work for UART 1 as well?
- You can overload `operator new` with a parameter that specifies the UART number:

```
class UART {
public:
    void *operator new(std::size_t, int n) {
        return reinterpret_cast<void *>(
            UART0_address + n * 0x1000
        );
    ~~~
};
```

Class-Specific New

- Using this operator new, you can write:

```
UART &port = new (0) UART;        // use UART0
```

- Unfortunately, this works only when the placement argument is 0 or 1, but it still compiles for other values:

```
UART &port = new (42) UART;       // compiles, but fails
```

- You can't prevent this with a static assertion.
- If you want to restrict the argument to 0 or 1, you must use a run-time check, or…
- You can use an enumeration as the placement parameter type…

Class-Specific New

```
class UART {
public:
    enum uart_number { zero, one };
    void *operator new(std::size_t, uart_number n) {
        return reinterpret_cast<void *>(
            UART0_address + n * 0x1000
        );
    }
    ~~~
private:
    enum { UART0_address = 0x03FFD000 };
    ~~~
};
```

Class-Specific New

- Using this operator new, you can write:

```
UART &port = new (UART::zero) UART;
```

- Now your choice of UART number is limited to only zero (= 0) and one (= 1).
- You can't place a UART somewhere else unless you go way out of your way:

```
UART &port = new (static_cast<UART::number>(42)) UART;
```

- Yet another reason to avoid casts.

Sooner Rather Than Later

- Static (compile-time) checking has advantages over run-time checking:
  - Fixing compile-time errors is almost always faster and easier than finding and fixing run-time errors.
  - Whereas you might be reluctant to pay for a run-time check, compile-time checks are free.
  - Whereas you can ship a program that might fail a run-time check, you can't ship a program that fails a compile-time check.
- Obviously, you can't detect all errors at compile time, but C++ offers lots of ways to turn would-be run-time errors into compile-time errors.

Modeling Devices More Accurately

- Most of the E7T's special registers support both read and write operations.
  - Class members of type `device_register` are read/write by default.
- But not all UART registers are read/write:
  - USTAT and URXBUF are read-only.
  - UTXBUF is write-only.

Modeling Devices More Accurately

- Writing to a read-only register can produce unpredictable misbehavior that can be hard to diagnose.
  - You're much better off enforcing read-only semantics at compile time.
- Fortunately, declaring a member as read-only is easy — just declare it `const`.
- Reading from a write-only register can also produce unpredictable misbehavior that can be hard to diagnose.
  - Again, you're better off catching this at compile time, too.
- Unfortunately, C++ doesn't have a write-only qualifier.
- However, you can enforce write-only semantics by using a class template…

A Write-Only Class Template

- `write_only<T>` is a simple class template for write-only types.
- For any type `T`, a `write_only<T>` object is just like a `T` object, except that it doesn't allow any operations that read the object's value.
- For example,

```
write_only<int> m = 0;
write_only<int> n;
n = 42;
m = n;  // error: attempts to read the value of n
```

A Write-Only Class Template

- The class template definition is:

```
template <typename T>
class write_only {
public:
    write_only() { }
    write_only(T const &v): m (v) { }
    void operator =(T const &v) { m = v; }
private:
    T m;
    // disallow copy operations...
    write_only(write_only const &);
    write_only &operator =(write_only const &);
};
```

Modeling Devices More Accurately

- Using the const qualifier and the `write_only<T>` template, the `UART` class data members look like:

```
class UART {
    ~~~
private:
    device_register ULCON;
    device_register UCON;
    device_register const USTAT;
    write_only<device_register> UTXBUF;
    device_register const URXBUF;
    device_register UBRDIV;
};
```

- However, compilers will complain about this, as they should…

Constructors and Const Members

- In C++, any class with non-static data members of const type must have at least one user-declared constructor:

```
class UART {
public:
    UART();     // no definition needed
    ~~~
    device_register const USTAT;
    ~~~
};
```

- You need not define the constructor provided you never actually define any UART objects.

Constructors and Const Members

- The compiler will complain if you define the constructor like this:

```
struct UART {
    UART() { }  // error (maybe just a warning)
    device_register const ULCON;
    ~~~
};
```

- Every constructor in a class with non-static data members of const scalar type must have a member-initializer for each such member.

Constructors and Const Members

- The const UART members are USTAT and URXBUF.
- If you define a default UART constructor, it should look like:

```
class UART {
public:
    UART(): USTAT (v1), URXBUF (v2) { }
    ~~~
};
```

- Whatever you use for v1 and v2, this is not good:
  - Constructing a memory-mapped UART will write values into read-only registers USTAT and URXBUF.
  - Even if you omit v1 and v2 (which you can do), the constructor will write zeros into those registers.

Constructors and Const Members

- If you don't want your program calling that constructor, then you should use access control to prevent the call:

```
class UART {
    ~~~
private:
    UART();     // no definition needed
    ~~~
    device_register ULCON;
};
```

- However, you won't be able to use member `operator new`.

A Read-Only Class Template

- There's another, better way to enforce read-only semantics.
- You can define a class template `read_only<T>` much like `write_only<T>`, and use that instead of `const`, as in:

```
class UART {
    ~~~
    read_only<device_register> USTAT;
    write_only<device_register> UTXBUF;
    read_only<device_register> URXBUF;
    ~~~
};
```

A Read-Only Class Template

- A `read_only<T>` object is not const:
  - You need not initialize a `read_only<T>` data member.
- Other than the default constructor, `read_only<T>` provides only two operations:
  - A conversion operator that lets you inspect the object, as in:

```
read_only<int> r;
~~~
int i = r;              // OK
```

  - An address-of operator that yields the address of the object as a "pointer to const".

```
int const *p = &r;  // OK
```

A Read-Only Class Template

- The class template definition is:

```
template <typename T>
class read_only {
public:
    read_only() { }
    operator T const &() const { return m; }
    T const *operator &() const { return &m; }
private:
    T m;
    // disallow copy operations...
    read_only(read_only const &);
    read_only &operator =(read_only const &);
};
```

Summary

- You can control memory-mapped i/o using only standardized language features (but they will have platform-specific behavior).
- Put your effort into writing data declarations that model the hardware as precisely as possible.
  - If you do that well, writing the code to manipulate the hardware will be much easier.
- Whatever you do, isolate language and hardware dependencies inside abstract types, either as structures in C or classes in C++.

2017 CPP-Summit

Is Parallel Programming still Hard or just OK

Michael Wong, Codeplay Software
VP of Research and Development
Chair of SYCL Heterogeneous Programming Language
ISOCPP.org Director, VP http://isocpp.org/wiki/faq/wg21#michael-wong
Head of Delegation for C++ Standard for Canada
Chair of Programming Languages for Standards Council of Canada

Chair of WG21 SG5 Transactional Memory
Chair of WG21 SG14 Games Dev/Low Latency/Financial Trading/Embedded
Editor: C++ SG5 Transactional Memory Technical Specification
Editor: C++ SG1 Concurrency Technical Specification
http:://wongmichael.com/about

Acknowledgement Disclaimer

Numerous people internal and external to the original C++/Khronos group, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk.
I even lifted this acknowledgement and disclaimer from some of them.
But I claim all credit for errors, and stupid mistakes. **These are mine, all mine!**

Legal Disclaimer

This work represents the view of the author and does not necessarily represent the view of Codeplay.

Other company, product, and service names may be trademarks or service marks of others.

Codeplay - Connecting AI to Silicon

Products

ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR™, HSA™ and Vulkan™

Addressable Markets

Automotive (ISO 26262)
IoT, Smartphones & Tablets
High Performance Compute (HPC)
Medical & Industrial

Technologies: Vision Processing
Machine Learning
Artificial Intelligence
Big Data Compute

Company

High-performance software solutions for custom heterogeneous systems

Enabling the toughest processor systems with tools and middleware based on open standards

Established 2002 in Scotland

~70 employees

Customers

BROADCOM
RENESAS
Imagination
QUALCOMM
Movidius
intel Partners AMD

Agenda

- Use the Proper abstractions?
- Why the rush to Massive Parallelism
- What Now?
- Hello World from C++11/14/17 Parallelism
- SYCL: C++ Heterogeneous (GPU) Programming
- Bonus: Executors

Use the Proper Abstraction

- Cores
- HW Threads
- Vectors
- Offload
- Heterogeneous
- Cloud
- Caches
- NUMA

- Tasks, C++11/14/14
- Tasks, C++11/14/17
- SIMD, Parallelism TS2
- OpenCL or SYCL
- OpenCL or SYCL
- OpenCL or SYCL
- Context, executors
- Context, executors

If you have to remember 2 things

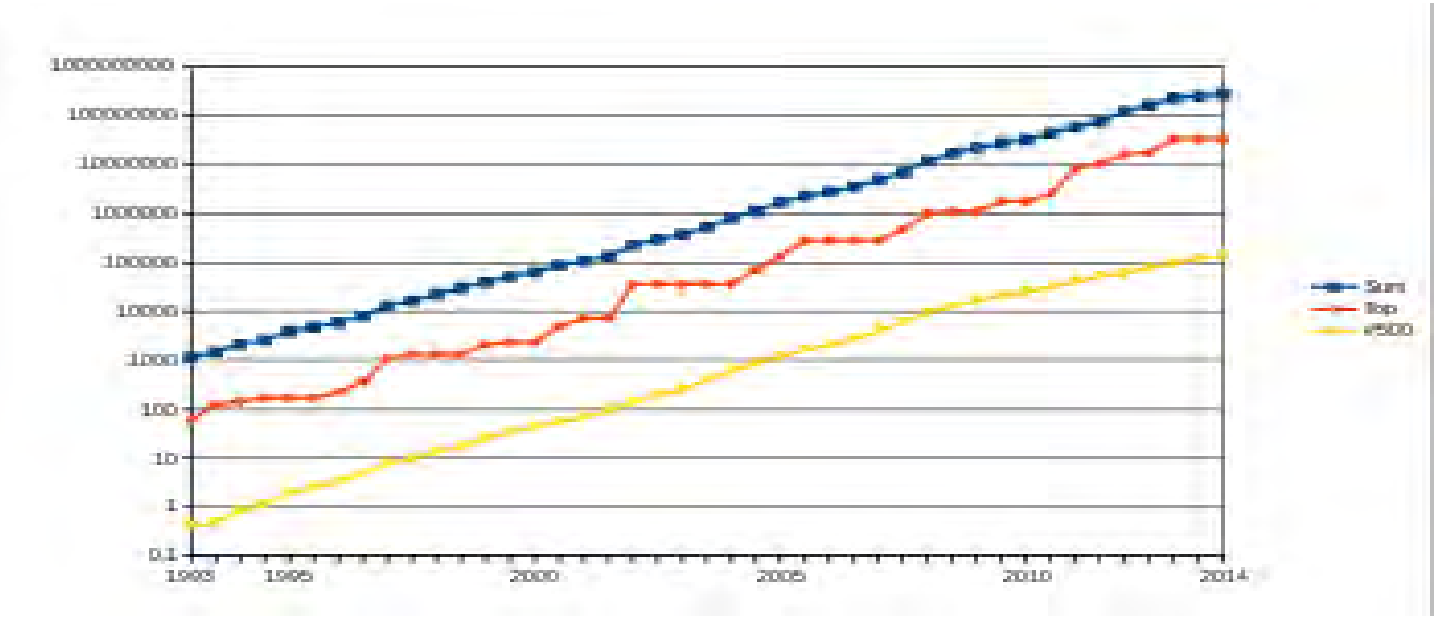
- Expose more parallelism
- Increase Locality of reference

Agenda

- Use the Proper abstractions?
- Why the rush to Massive Parallelism
- What Now?
- Hello World from C++11/14/17 Parallelism
- SYCL: C++ Heterogeneous (GPU) Programming
- Bonus: Executors

Why is GPU important now?

- Or is it a flash in the pan?
- The race to exascale computing .. 10 ^{18} flops
- Vertical scale is in GFLOPS



Top500 contenders

Internet of Things

- All forms of accelerators, DSP, GPU, APU, GPGPU
- Network heterogenous consumer devices
  - Kitchen appliances, drones, signal processors, medical imaging, auto, telecom, automation, not just graphics engines







WORLD OF WARSHIPS
WARGAMING.NET

Beautiful and elegant Lambdas

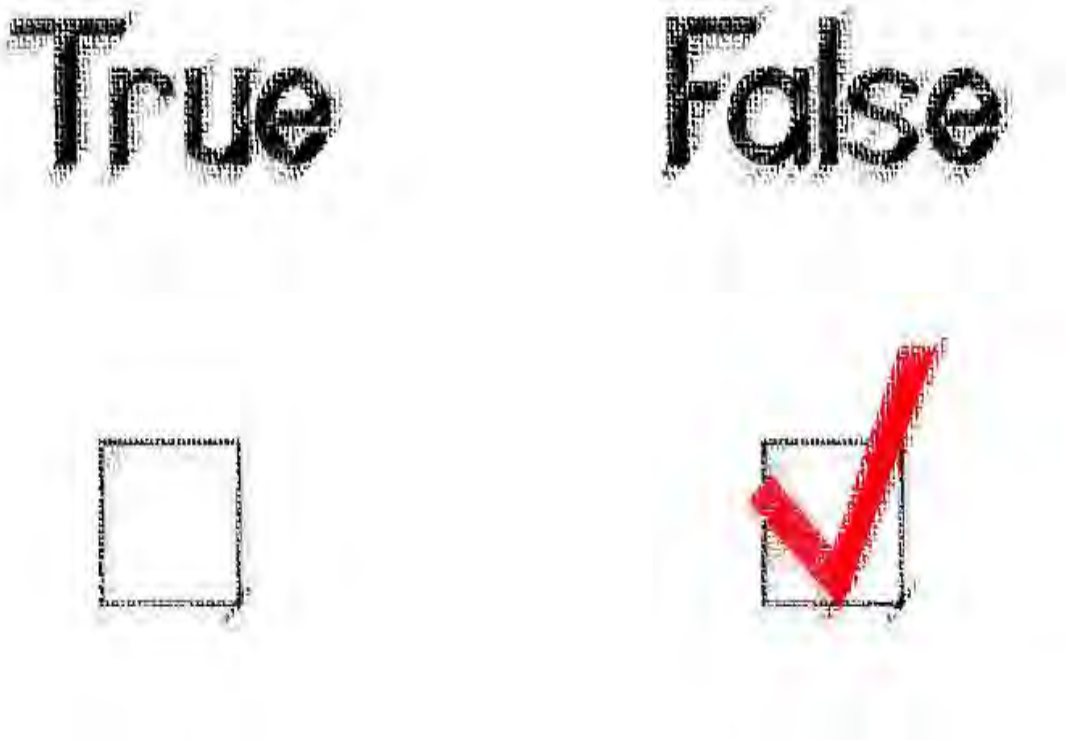
C++98	C++11/14/17

vector<int>::iterator i =
v.begin();
for(; i != v.end(); ++i) {
  if(*i > x && *i < y)
  break;
}
	auto i = **find_if**(begin(v), end(v),
	[=](int i) {
	  return i > x && i < y;
	});

- "Lambdas, Lambdas Everywhere"
  http://vimeo.com/23975522
- *Full Disclosure: I love C++ and have for many years*
- *But … What is wrong here*

The Truth

- Q: Does your language allow you to access all the GFLOPS of your machine?



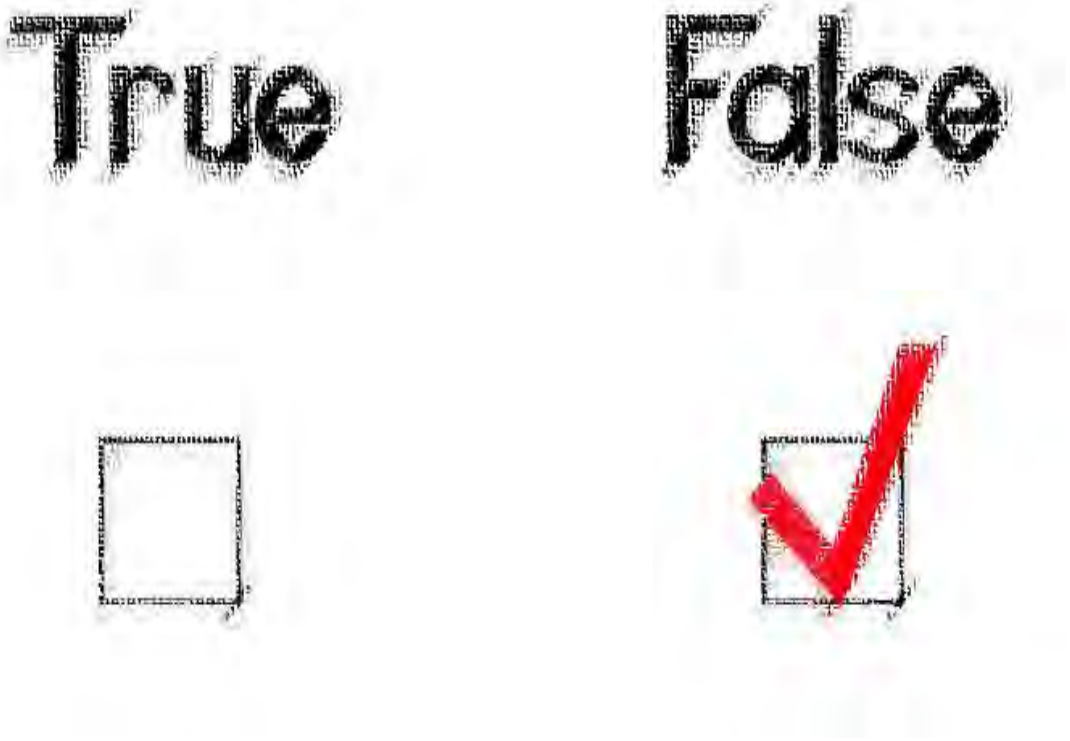
“Is there in Truth No Beauty?”

by George Herbert *from Jordan*

- Q: Does your language allow you to access all the GFLOPS of your machine?
- A: What a quaint concept!
  - I thought its natural to drop out into OpenCL, CUDA, OpenGL, DirectX, C++AMP, Assembler …. to get at my GPU
  - Why? I just use my language as a cool driver, it’s a great scripting language too. But for real kernel computation, I just use Fortran
  - I write vectorized code, so my vendor offers me intrinsics, they also tell me they can auto-vectorize, though I am not sure how much they really do, so I am looking into OpenCL
  - Well, I used to use one thread, but now that I use multiple threads, I can get at it with C++11, OpenMP, TBB, GCD, PPL, MS then, Cilk
  - I know I may have a TM core somewhere, so my vendor offers me intrinsics
  - No I like using a single thread, so I just use C, or C++

The Question

- Q: Is it true that there is a language that allows you to access all the GFLOPS of your machine?



Power of Computing

- 1998, when C++ 98 was released
  - Intel Pentium II: 0.45 GFLOPS
  - No SIMD: SSE came in Pentium III
  - No GPUs: GPU came out a year later
- 2011: when C++11 was released
  - Intel Core-i7: 80 GFLOPS
  - AVX: 8 DP flops/HZ*4 cores *4.4 GHz= 140 GFlops
  - GTX 670: 2500 GFLOPS
- Computers have gotten so much faster, how come software have not?
  - Data structures and algorithms
  - latency

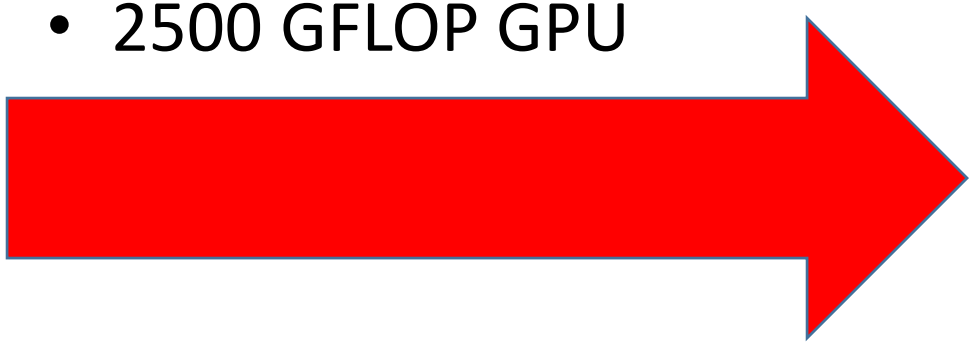
In 1998, a typical machine had the following flops

- .45 GFLOP, 1 core

- Single threaded C++98/C99/Fortran dominated this picture

In 2011, a typical machine had the following flops

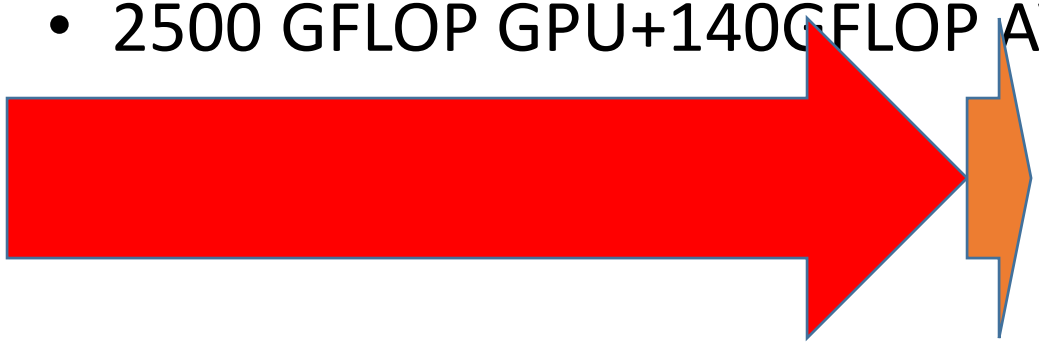
- 2500 GFLOP GPU



- To program the GPU, you have to use CUDA, OpenCL, SYCL OpenGL, DirectX, Intrinsics, C++AMP

In 2011, a typical machine had the following flops

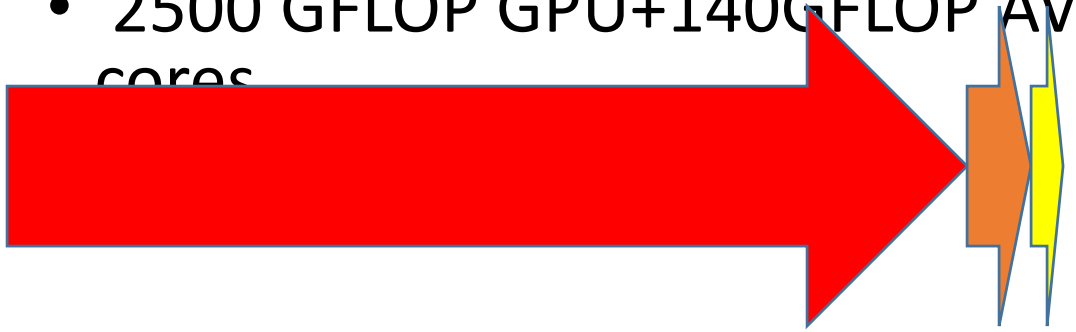
- 2500 GFLOP GPU+140GFLOP AVX



- To program the GPU, you have to use CUDA, OpenCL, SYCL, OpenGL, DirectX, Intrinsics, C++AMP
- To program the vector unit, you have to use SYCL, Intrinsics, OpenCL, or auto-vectorization

In 2011, a typical machine had the following flops

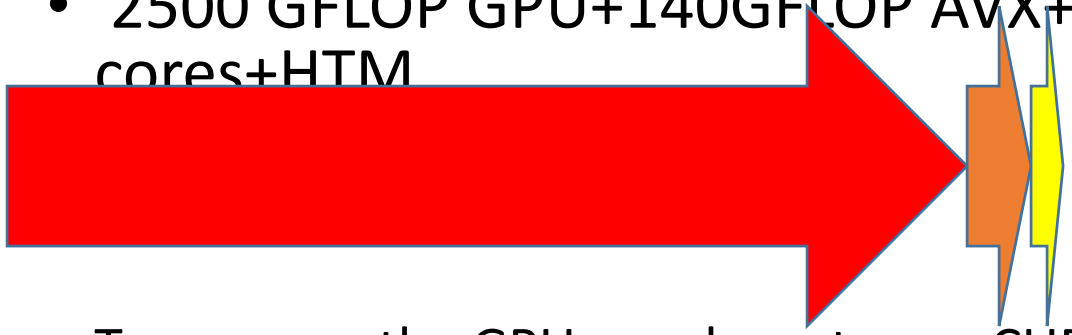
- 2500 GFLOP GPU+140GFLOP AVX+80GFLOP 4 cores



- To program the GPU, you have to use CUDA, OpenCL, SYCL, OpenGL, DirectX, Intrinsics, C++AMP
- To program the vector unit, you have to use SYCL, Intrinsics, OpenCL, or auto-vectorization
- To program the CPU, you might use C/C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors

In 2011, a typical machine had the following flops

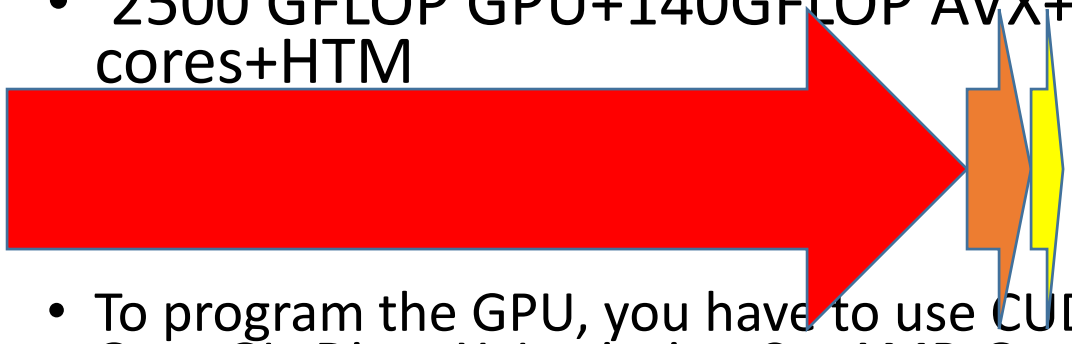
- 2500 GFLOP GPU+140GFLOP AVX+80GFLOP 4 cores+HTM



- To program the GPU, you have to use CUDA, OpenCL, SYCL, OpenGL, DirectX, Intrinsics, C++AMP
- To program the vector unit, you have to use SYCL, Intrinsics, OpenCL, or auto-vectorization
- To program the CPU, you might use C/C++11, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors
- To program HTM, you have?

In 2014, a typical machine had the following flops

- 2500 GFLOP GPU+140GFLOP AVX+80GFLOP 4 cores+HTM



- To program the GPU, you have to use CUDA, OpenCL, SYCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenMP
- To program the vector unit, you have to use SYCL, Intrinsics, OpenCL, or auto-vectorization, OpenMP
- To program the CPU, you might use C/C++11/14, SYCL, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors
- To program HTM, you have the upcoming C++ TM TS

In 2017, a typical machine had the following flops

- 4600 GFLOP GPU+560 GFLOP AVX+140 GFLOP

- To program the GPU, you have to use SYCL, CUDA, OpenCL, OpenGL, DirectX, Intrinsics, C++AMP, OpenMP
- To program the vector unit, you have to use SYCL, Intrinsics, OpenCL, or auto-vectorization, OpenMP
- To program the CPU, you might use C/C++11/14/17, SYCL, OpenMP, TBB, Cilk, MS Async/then continuation, Apple GCD, Google executors, OpenMP, parallelism TS, Concurrency TS

Agenda

- Use the Proper abstractions?
- Why the rush to Massive Parallelism
- What Now?
- Hello World from C++11/14/17 Parallelism
- SYCL: C++ Heterogeneous (GPU) Programming
- Bonus: Executors
-

What now?

- C++11 Std is
  - 1353 pages compared to 817 pages in C++03
- C++14 Std is
  - 1373 pages (N3937), n3972 (free)
- The new C++17 CD is
  - N4606: 1572 pages
- C99
  - 550 pages
- C11 is
  - 701 pages compared to 550 pages in C99

- OpenMP 3.1 is
  - 160 pages and growing
- OpenMP 4.0 is
  - 320 pages
- OpenMP 4.5 is
  - 359 pages
- OpenCL 2.0
  - 288 pages
- OpenCL 2.1
  - 300 pages
- OpenCL 2.2
  - 304 pages