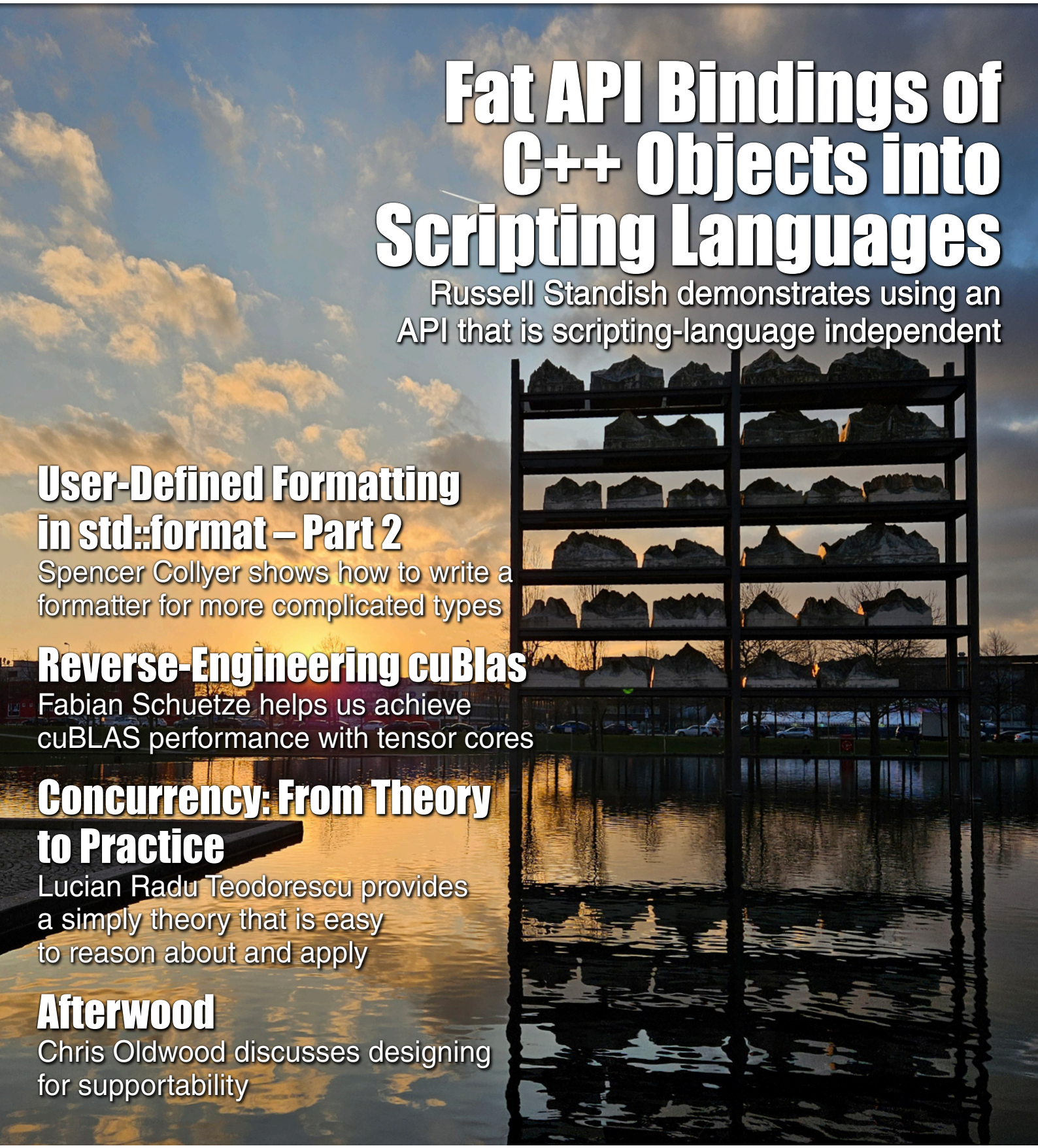
overload 181

JUNE 2024 £4.50



Fat API Bindings of C++ Objects into Scripting Languages

Russell Standish demonstrates using an API that is scripting-language independent

User-Defined Formatting in std::format – Part 2

Spencer Collyer shows how to write a formatter for more complicated types

Reverse-Engineering cuBlas

Fabian Schuetze helps us achieve cuBLAS performance with tensor cores

Concurrency: From Theory to Practice

Lucian Radu Teodorescu provides a simply theory that is easy to reason about and apply

Afterwood

Chris Oldwood discusses designing for supportability

A magazine of ACCU

ISSN: 1354-3172

June 2024
ISSN 1354-3172

Printing and distribution
Parchment (Oxford) Ltd

Cover design
Original design by Pete Goodliffe
pete@goodliffe.net
Cover photo by Fran Buontempo: taken outside the Internationales Congress Center München when attending the OOP Conference in February 2024.

ACCU
ACCU is an organisation of programmers who care about professionalism in programming. We care about writing good code, and about writing it in a good way. We are dedicated to raising the standard of programming.

Many of the articles in this magazine have been written by ACCU members – by programmers, for programmers – and all have been contributed free of charge.

Overload is a publication of the ACCU For details of the ACCU, our publications and activities, visit the ACCU website: www.accu.org

Copy deadlines
All articles intended for publication in Overload 182 should be submitted by 1st July and those for Overload 183 by 1st September 2024.

Breaking Bad (Habits)

Trying to make a change can be difficult. Frances Buontempo considers how to start forming new, better, habits.

As you may have noticed, I have fallen into a pattern of never writing an editorial for *Overload* and always making excuses. This time is no different. However, I have been contemplating my usual approaches to tasks, and more besides. Breaking bad habits is difficult, but the first step is usually spotting them. Pause for a moment, and ask yourself if you have any tendencies you to fall into on autopilot? Some of these might be useful, like brushing your teeth or going to bed at a specific time. Some may be relatively harmless, for example glancing at Stack Overflow or Reddit when a build or similar is taking a while. Both are possibly better than sitting doing nothing, though it might be more sensible to stand up and stretch for a bit, or having a sword fight [xkcd]. Sitting still for too long doesn't do us any good. Bad posture is a horrible habit, as we all know, and even using a mouse for too long or typing badly can set off repetitive strain injury. Many people have opinions on potential fixes [reddit], but a standing desk, track ball, or some form of mixed martial arts might not solve all your problems. Changing how you do things might help though.

Waking up on time may or may not be a habit for you. I do wake up, but my preferred time is about half eight or so. Setting an alarm helps. I noticed recent news saying that people's iPhones are failing to ring when the morning alarm is supposed to go off [Vigliarolo24]. Oops. I remember getting a swanky new alarm clock which plugged straight into the wall socket when I was young. I was paranoid about potential power cuts stopping it working, so always had a back-up hand wound clock. I now rely on my husband as back up, because he's definitely a morning person, so is usually awake a couple of hours before me. It's odd to think back and notice how my day-to-day habits have changed over time. This probably means I am getting old(er).

Do you have any good habits? Maybe focus on programming related areas, rather than everything. Do you practice? Read books or articles? Listen to podcasts? Do you always write tests first? Or, like me, do you claim to do TDD, but know full well you have a few 'scripts' dotted around that you never tested. Or shovel lots of code in `main`, which has no tests? Another bad habit I have fallen into is not bothering to use a library to parse arguments in a C++ project. I spent time learning to do this properly in Python, but never got around to picking and learning a C++ approach. I have tried several, so my excuse is too much choice, but not enough time. A very poor excuse, I know. I usually resort to a small hack to try to parse numbers or strings I pass in, but can never remember the order I set up. So, now I have confessed in public, I really must do something about this. Perhaps you can own up to something too, and use that as motivation to change. Acknowledging a problem is the first step to fixing it, after all.

I suspect my argument-parsing laziness is based on feeling it's a small thing and I don't have time to do it properly. I feel like my small hack will be quicker. However, we all know the quick workaround often turns out to be a time sink in the long run. Sometimes, I notice other problematic approaches, and after a few times limping along with a 'bodge' I created, I get annoyed enough to re-create something better. Annoyance can be a motivating factor, but there are other ways to help yourself change track. I attended Phil Nash's session 'Rewiring your brain – with Test Driven Thinking' at *MeetingCpp* last year. [Nash23]. He's given several variations of this talk; do take time to listen to one. He talked about the reward of seeing the green of passing tests being habit forming. If you know you will get a reward for something, you might be more likely to do it. Eventually, you no longer need the reward itself, hence his title 'rewiring your brain'. His abstract [ACCU24] addresses the idea of spending time doing the right thing seeming wasteful:

> We all say we should write tests, or at least we should write more of them.
>
> But we never seem to have the time, and our focus is on the actual problems we're trying to solve. Nobody wants to be bogged down by busy work.
>
> What if all of that was wrong?
>
> What if tests could save you time, improve your focus – and even be fun!

Maybe the thing to do is promise yourself you will start with just one test first next time you have code to write. Once you have one in place, it's easier to add others. I have added a single test to a few projects in previous jobs, and it never takes long for others to add more. One small change is all you need to get started.

Now, you might notice something isn't ideal, but not be sure what to do instead. I don't know the solution to this, but often talking to others helps. Don't suffer in silence. Or perhaps, you don't realise you have a problem. An example might be accidently relying on undefined behavior. If the code appears to behave on one machine, you may never notice. As soon as you switch or upgrade compilers, things blow up. It's worth throwing an undefined behavior sanitizer at your code once in a while [Clang]. Sometimes code does work, but may be confusing for someone else to read. Code reviews can pick out potential areas for improvement like this. If you have been deep in something for a while, managing to get it working seems like success. However, as we know, code might need to be read at some point in the future, so ensuring it is readable is sensible. This often requires someone else to look with fresh eyes. If you don't have anyone to hand, for example if this is a personal project, don't be shy about asking the accu-general email group, or other community.

In general, if you don't notice something is a bit broken, you are unlikely to fix it. Stepping back might help you notice the bigger picture though. That's why I enjoy the conferences. Even if I go to a talk that I think I know

Frances Buontempo has a BA in Maths + Philosophy, an MSc in Pure Maths and a PhD using AI and data mining. She's written a book about machine learning: *Genetic Algortithms and Machine Learning for Programmers*. She has been a programmer since the 90s, and learnt to program by reading the manual for her Dad's BBC model B machine. She can be contacted at frances.buontempo@gmail.com.

a lot about, I always come away with new things to think about, or realise I have slightly misunderstood or forgotten something. Not everyone is fortunate enough to be able to attend a conference, but several now seem to be hybrid, allowing people who can't afford to travel or even take all the time off the chance to join in. I'm pleased to see some conferences have offered free online tickets to anyone who has a poster submission accepted. Many students have to prepare a poster for a final year project. All we need now is to let Universities know about this opportunity. Let's change the world, one step at a time.

Change can be unnerving. Once you find a way to do something, it can be very difficult to adopt a new approach. They say you can't teach an old dog new tricks. However, if the dog is willing to change, then anything might be possible. If someone tries to force you to drop old habits, you might be more likely to defiantly stick to your current ways. However, as Phil's talk points out, if there is a reward for changing, ranging from a treat, to a warm fuzzy feeling from a notification saying "Tests passed", even all the way through to saved time or confusion, change can happen. I am experimenting with a small handful of personal mantras to help motivate myself and do the 'right thing'. For example, I enjoy going to the gym, but keep allowing other jobs to crowd in and stop me. I tell myself, "Go to the gym first, you'll feel better." I'm right, but often argue with myself for a bit first. "But, this task needs doing today"… "So, do it later, after the gym." I'm gradually ending up just needing to say "Go to the gym." Whether I can talk myself round from all bad habits this way remains to be seen.

Programmers are often caricatured as arguing over silly things, such as brace placement, or tabs versus spaces. We do often end up disagreeing over seemingly simple things, but coming to an agreement with others who have different experiences to you can be hard. I tend to give my variables full names, but if I spend time reading maths code or books, I often fall back to single letter variable names. If I have just read up on a model, and the paper or book uses an *x* for a variable name, then you will often find `x` in my code. Don't at me… I don't think this is actually bad, or a habit. It just illustrates that current context often influences behaviour. I have to consciously swivel my head back to a fuller variable name, say `horizontal_distance`, if I am collaborating with others who don't like the more terse approach. Trying to be consistent and respect conventions when appropriate is sensible. If you find a particular coding style really difficult, maybe you can find an automatic code formatter that will do this for you? Save the arguments for important issues, like potential production crashes or incorrectly implemented algorithms. And of course, automating compliance with rules, so you don't have to do it yourself, is what might be expected of a proper programmer ☺

Now, automation and AI might not be the solution to every problem out there. And sometimes, you just can't manage to change what you are doing at the moment. Yes, this might be personal issues like your posture or similar, as well as writing hacky arg parsing code. If you can't manage to make that change now, don't beat yourself up over it. I have taken the first step, by acknowledging my terrible code. Next time I need to read arguments to main, I will say "I don't know how to do better, Yet. But I will one day." And one day I will make the change. If you have a similar problem, be kind to yourself. Another approach might be finding someone to pair with, or even simply delegating the task. If you can't do differently, let go. Maybe try an actual person rather than AI though? Just a suggestion.

It seems appropriate to end with the relatively well-known Serenity prayer:

> *God, grant me the serenity to accept the things I cannot change,*
> *the courage to change the things I can,*
> *and the wisdom to know the difference.*

However, I was going to find a reference in case a reader hasn't come across this before. I now have yet another tab open [Buontempo24], and notice Wikipedia says it needs help. The page for the Serenity prayer [Wikipedia] has a banner at the top saying in bold it needs "attention from an expert in history". Apparently, the specific problem is "internally discrepant conclusions", among other problems. I am not sure a history expert can fix that. Maybe a logician or programmer is required? Another section possibly contains "original research". Shocking. I thought I got a PhD because I had undertaken original research. I suppose my thesis did have a literature review first, sharing references for the state of the art at the time, and when I tried to do something original, I did have references at least to the maths and machine learning techniques I was using. Perhaps I have just owned up to another bad habit – getting distracted by being very literal when I read something. If anyone out there does know the history of this prayer, please go fix the internet for me. I don't have time, I need to learn an arg parsing library.

References

[ACCU24] Phil Nash 'Rewiring your brain – with Test Driven Thinking' (abstract) available at https://accuconference.org/session/rewiring-your-brain-with-test-driven-thinking

[Buontempo24] Frances Buontempo 'Editorial: I Don't Believe It', in *Overload* 180, available at https://accu.org/journals/overload/32/180/buontempo/

[Clang] UndefinedBehaviorSanitizer: https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html

[Nash23] Phil Nash 'Rewiring your brain with test driven thinking in C++', presented at *Meeting C++ 2023*, available at: https://www.youtube.com/watch?v=Hx-1Wtvhvgw

[reddit] 'Fixing the Developer Posture' (comments on the document, although the link to the original PDF no longer works): https://www.reddit.com/r/programming/comments/6dcs7s/fixing_the_developer_posture_pdf/

[Vigliarolo24] Brandon Vigliarolo, 'Miss your morning iPhone alarm? It's not just you, and Apple is looking into it', *The Register*, posted 1 May 2024, https://www.theregister.com/2024/05/01/miss_your_morning_iphone_alarm/

[Wikipedia] Serenity prayer: https://en.wikipedia.org/wiki/Serenity_Prayer

[xkcd] 'Compiling': https://xkcd.com/303/

User-Defined Formatting in std::format – Part 2

Last time, we saw how to provide formatting for a simple user-defined class. Spencer Collyer builds on this, showing how to write a formatter for more complicated types.

In the previous article in this series [Collyer24], I showed how to write a class to format user-defined classes using the `std::format` library. In this article I will describe how this can be extended to container classes or any other class that holds objects whose type is specified by the user of your class.

A note on the code listings: The code listings in this article have lines labelled with comments like `// 1`. Where these lines are referred to in the text of this article, it will be as 'line `1`' for instance, rather than 'the line labelled `// 1`'.

Nested formatter objects

The objects created from the `formatter` template structs are just ordinary C++ objects – there is nothing special about them[1]. In particular, there is nothing to stop you including an object of a `formatter` template type inside one of your user-defined `formatter` structs.

You might wonder why you would want to do that. One simple case is if you have a templated container class, and want to create a `formatter` that can output the container in one go, rather than having to write code to iterate over the container and output each value in turn. Having a nested `formatter` for the contained value type allows you to do this and allow the values to be formatted differently to the default, as the following examples will show. Other uses will no doubt come to mind for your own classes.

A formatter for std::vector

The first example we will look at is a simple `formatter` for `std::vector`. The code is given in Listing 1, and sample output is in Listing 2.

The format specification we will use has the following form:

```
[ 'w' lc rc ] [ 's' sep ] [ '/'
[ value-fmt-spec ] ]
```

The element starting with `w` allows the user to specify characters to wrap the vector values in the output. The `w` must be followed by exactly two characters. The first character, *lc*, is written before the value, and the second, *rc*, is written after the value. If not given, no wrapper characters are output.

The element starting with `s` allows the user to specify a single character to act as a separator between the individual vector element values. If given, the `s` must be followed by exactly one character, which will be used as the separator. If not given, it defaults to the space character. If a separator is given it will be followed by a space in the output.

1 Other than being called automatically by the various `std::format` functions that is, obviously.

Spencer Collyer Spencer has been programming for more years than he cares to remember, mostly in the financial sector, although in his younger years he worked on projects as diverse as monitoring water treatment works on the one hand, and television programme scheduling on the other.

```
#include <format>
#include <iostream>
#include <vector>

using namespace std;

template<typename T>
struct std::formatter<vector<T>>
{
  constexpr auto
    parse(format_parse_context& parse_ctx)
  {
    auto iter = parse_ctx.begin();
    auto get_char = [&]() { return iter
      != parse_ctx.end() ? *iter : 0; };
    char c = get_char();
    if (c == 0 || c == '}')   // 1
    {
      m_val_fmt.parse(parse_ctx);   // 2
      return iter;
    }
    auto get_next_char = [&]() {   // 3
      ++iter;
      char vc = get_char();
      if (vc == 0)
      {
        throw format_error(
          "Invalid vector format specification");
      }
      return vc;
    };
    if (c == 'w')   // 4
    {
      m_lc = get_next_char();
      m_rc = get_next_char();
      ++iter;
    }
    if ((c = get_char()) == 's')   // 5
    {
       m_sep = get_next_char();
       ++iter;
    }
    if ((c = get_char()) == '/' || c == '}') // 6
    {
      if (c == '/')   // 7
      {
        ++iter;
      }
      parse_ctx.advance_to(iter);   // 8
      iter = m_val_fmt.parse(parse_ctx);   // 9
    }
    if ((c = get_char()) != 0 && c != '}')  // 10
    {
      throw format_error(
        "Invalid vector format specification");
    }
    return iter;
  }
  auto format(const vector<T>& vec,
    format_context& format_ctx) const
```

Listing 1

The objects created from the formatter template structs are just ordinary C++ objects – there is nothing special about them

```
  {
    auto pos = format_ctx.out();    // 11
    bool need_sep = false;
    for (const auto& val : vec)
    {
      if (need_sep)   //  12
      {
        *pos++ = m_sep;
        if (m_sep != ' ')
        {
          *pos++ = ' ';
        }
      }
      if (m_lc != '\0')   //  13
      {
        *pos++ = m_lc;
      }
      format_ctx.advance_to(pos); //  14
      pos = m_val_fmt.format(val,
        format_ctx);   // 15
      if (m_rc != '\0')   //  16
      {
        *pos++ = m_rc;
      }
      need_sep = true;
    }
    return pos;
  }

private:
  char m_lc = '\0';
  char m_rc = '\0';
  char m_sep = ' ';
  formatter<T> m_val_fmt; // 17
};

int main()
{
  vector<int> vec{1, 2, 4, 8, 16, 32};
  cout << format("{}\n", vec);             // a
  cout << format("{:w[]}\n", vec);         // b
  cout << format("{:s,}\n", vec);          // c
  cout << format("{:w[]s,}\n", vec);       // d
  cout << format("{:w[]/3}\n", vec);       // e
  cout << format("{:s;/+0{}}\n", vec, 5); // f
  vector<vector<int>> vec2{ {1, 2, 3},
    {40, 50, 60}, {700, 800, 900} };
  cout << format("{}\n", vec2);            // g
  cout << format("{:w[]}\n", vec2);        // h
  cout << format("{:s,}\n", vec2);         // i
  cout << format("{:w[]s,}\n", vec2);      // j
  cout << format("{:w[]/s,}\n", vec2);     // k
  cout << format("{:s;/s,/03}\n", vec2);   // l
}
```

Listing 1 (cont'd)

The `/` delimits the start of the *format-spec* for the vector's value type. This will be read by the member variable `m_val_fmt`, defined in line 17, to set up the formatting for the vector values. If not given, it will use the default formatting for the value type. It is allowable – although not really useful – to give a `/` character with no following *format-spec*.

```
a: 1 2 4 8 16 32
b: [1] [2] [4] [8] [16] [32]
c: 1, 2, 4, 8, 16, 32
d: [1], [2], [4], [8], [16], [32]
e: [  1] [  2] [  4] [  8] [ 16] [ 32]
f: +0001; +0002; +0004; +0008; +0016; +0032
g: 1 2 3 40 50 60 700 800 900
h: [1 2 3] [40 50 60] [700 800 900]
i: 1 2 3, 40 50 60, 700 800 900
j: [1 2 3], [40 50 60], [700 800 900]
k: [1, 2, 3] [40, 50, 60] [700, 800, 900]
l: 001, 002, 003; 040, 050, 060; 700, 800, 900
```

Listing 2

The parse function

The first few lines of the `parse` function, up to line 1, are the same as the ones for the `Point` class described in my previous article.

The first notable change is line 2. This calls the `parse` function on the nested `m_val_fmt` object, which is the `formatter` for the vector's value type. Doing this allows the `m_val_fmt` object to set up its formatting for the default case where no *format-spec* is given.

The `get_next_char` function defined starting at line 3 is used to read the next character from the *format-spec*. It throws an exception if there are no more characters to read, as indicated by getting 0 back from the `get_char` function. As with the `get_char` function, when this function is done it leaves the `iter` variable pointing at the character read.

The `if`-statement starting at line 4 simply processes any `w` element to read the wrapper characters. It should be obvious what it is doing. Similarly, the code starting at line 5 just processes any `s` element to read the separator character.

The `if`-statement starting at line 6 holds the code to initialise the `m_val_fmt` object when we don't have an empty *format-spec*. The `if`-statement condition has to check for both the `/` character that indicates the value type has a *format-spec*, and also for the `}` character that indicates the end of the *format-spec*, i.e. the case where there is no specific *format-spec* for the value type.

Line 7 checks for the `/` character and, if present, increments `iter`. This is because the `/` character is not part of the value type's *format-spec* so seeing it would confuse the `m_val_fmt.parse` function.

Line 8 is important because, by calling the `advance_to` function on `parse_ctx`, it resets `parse_ctx`'s idea of where in the *format-spec* the start point is located. When line 9 then calls `m_val_fmt.parse`, it will start the processing at the correct position, i.e. the start of the value type's embedded *format-spec*, not the `vector`'s *format-spec*.

When the `m_val_fmt.parse` function returns, it should have processed everything up to the `}` that terminates the *format-spec*. Note that in this case the `}` is doing double duty, as it terminates both the `vector` *format-spec* and the embedded value type *format-spec*. Line 10 carries out our normal check for correct termination of the *format-spec*.

The majority of the function is just a loop over the vector's values

The format function

Line `11` puts the current output iterator from **`format_ctx`** into the **`pos`** variable. This indicates where the next data is written to in the output.

The majority of the function is just a loop over the vector's values. The interesting parts are described below.

Line `12` checks if we need to output a separator character. The first time through the loop this will be false, but on subsequent iterations it will be true. The body of the **`if`**-statement just outputs the separator character, then if it is not a space it outputs a space character as well. As we are just outputting single characters each time we can use the **`*pos++ = c`** form to write them to the output.

Lines `13` and `16` write the wrapper characters, if they are defined.

Line `14` sets up the **`format_ctx`** variable correctly for the output in the next line. By calling **`advance_to`** on **`format_ctx`** we set its output iterator to match the position we have reached up to this point in the function.

Line `15` outputs the current value by calling the **`format`** function on the **`m_val_fmt`** object. Because we have updated the output iterator on **`format_ctx`** in the line above, the value will be written to the correct position in the output. The **`format`** function returns the new value of the output iterator.

Test cases

The first set of test cases in the **`main`** function use a simple vector-of-ints as the value to output.

Test case `a` checks that the default formatting works for the **`vector`** and its contained values.

Test cases `b`, `c`, and `d` just check that the various parts of the **`vector`** *format-spec* work, but with no value *format-spec*, so the values will just use the default output.

Test case `e` checks that using a *format-spec* for the value works correctly. Using wrapper characters lets us check that the output values are indeed all output in fields three characters wide.

Test case `f` shows that you can use nested format specifiers in the value *format-spec*, in this case picking up the width from the argument list.

The second set of test cases use a vector-of-vectors-of-ints as the value to output.

Test case `g` checks that the default formatting works.

Note that in the output for case `g`, there is no way to tell where one nested vector ends and the next one starts. Test cases `h`, `i`, and `j` use the various parts of the **`vector`** *format-spec* to delimit the nested vectors in various ways.

Test case `k` checks that the nested vectors are output using the value *format-spec*, as can be seen from each value in them being separated by the comma specified by the *format-spec*.

Test case `l` checks that the nested vector's *format-spec* can handle a *format-spec* for their values – in this case indicating a three character wide, zero-padded field.

A formatter for std::map

The next example we will look at is a **`formatter`** for **`std::map`**. This is more complicated because we want to allow *format-specs* for both the key type and value type of the map. The code is given in Listing 3, and sample output is in Listing 4.

The format specification we will use has the following form:

```
[ 'w' lc rc ] [ 'c' conn ] [ 's' sep ]
[ '/' '{' key-fmt-spec '}' '{' value-fmt-spec '}'
]
```

The elements starting with **`w`** and **`s`** have identical purposes and default to the ones we used for **`std::vector`**.

The element starting with **`c`** allows you to specify the connecting character that is output between the key and the value. The **`c`** must be followed by exactly one character. If not specified, the default value is **`=`**.

The **`/`** character introduces the *format-specs* for the key and value types of the map. Unlike the case for **`std::vector`**, these *format-specs* are mandatory if you have a **`/`** character. Unsurprisingly, *key-fmt-spec* is the one for the key type, and the *value-fmt-spec* is the one for the value type. You can use a default **`{}`** for either of these if you don't want to change that particular item's format.

Note that these two nested *format-specs* are surrounded by **`{`** and **`}`** characters. This breaks one of the guidelines I gave in the previous article for format specification mini-languages (see the appendix 'Simple Mini-Language Guidelines' in that article). The reason for this is as follows. The **`parse`** functions in **`formatter`**s need to see a **`}`** character terminating the *format-spec* they are processing. This means when processing the *key-fmt-spec*, we need a **`}`** character at the end of the *key-fmt-spec*, before the *value-fmt-spec* starts. This could be confusing as it might look like it is the **`}`** that terminates the **`std::map`**'s *format-spec*. Using a **`{`** at the start of the *key-fmt-spec* helps to make it clear it is a single unit. As for the *value-fmt-spec*, that could use the **`}`** at the end of the **`std::map`** *format-spec* as its terminator, just like we do for **`std::vector`** above, but for consistency between the two *format-specs* it made more sense to also surround it with **`{`** and **`}`** characters.

The parse function

Much of the **`parse`** function is similar to the one for **`std::vector`** shown previously. Lines `1` and `2` handle the case where we have a default *format-spec*, calling the respective **`parse`** functions on the nested **`formatter`**s for the key and value types. Note that we assume here that the **`m_key_fmt.parse`** function doesn't alter the **`parse_ctx`** value passed to it. If you are concerned that it might do, you can take a copy of **`parse_ctx`** and pass that copy to the **`m_val_fmt.parse`** function instead.

these two nested format-specs are surrounded by { and } characters... breaks one of the guidelines I gave in the previous article for format specification mini-languages

```
#include <format>
#include <iostream>
#include <map>
using namespace std;

template<typename K, typename V>
struct formatter<map<K,V>>
{
  constexpr auto
    parse(format_parse_context& parse_ctx)
  {
    auto iter = parse_ctx.begin();
    auto get_char = [&]() { return
      iter != parse_ctx.end() ? *iter : 0; };
    char c = get_char();
    if (c == 0 || c == '}')
    {
      m_key_fmt.parse(parse_ctx); // 1
      m_val_fmt.parse(parse_ctx); // 2
      return iter;
    }
    auto get_next_char = [&]() {
      ++iter;
      char vc = get_char();
      if (vc == 0)
      {
        throw format_error(
          "Invalid map format specification");
      }
      return vc;
    };
    if (c == 'w')   // 3
    {
      m_lc = get_next_char();
      m_rc = get_next_char();
      ++iter;
    }
    if ((c = get_char()) == 'c')    // 4
    {
      m_con = get_next_char();
      ++iter;
    }
    if ((c = get_char()) == 's')    // 5
    {
      m_sep = get_next_char();
      ++iter;
    }
    if ((c = get_char()) == '/')    // 6
    {
      //  Next char must be '{' at start of key
      // format spec
      if ((c = get_next_char()) != '{')   // 7
      {
        throw format_error(
          "Invalid map format specification");
      }
      parse_ctx.advance_to(++iter);       // 8
      iter = m_key_fmt.parse(parse_ctx);  // 9
      // Iter should point to '}' at end of key
      // format spec
```

Listing 3

```
      if ((c = get_char()) != '}')        // 10
      {
        throw format_error(
          "Invalid map format specification");
      }
      // Next char must be '{' at start of value
      // format spec
      if ((c = get_next_char()) != '{')   // 11
      {
        throw format_error(
          "Invalid map format specification");
      }
      parse_ctx.advance_to(++iter);
      iter = m_val_fmt.parse(parse_ctx);
      // Iter should point to '}' at end of
      // value format spec
      if ((c = get_char()) != '}')
      {
        throw format_error(
          "Invalid map format specification");
      }
      // Advance past the '}' at end of value
      // format spec
      ++iter;
    }
    else if (c == '}')  // 12
    {
      parse_ctx.advance_to(iter);
      m_key_fmt.parse(parse_ctx);
      m_val_fmt.parse(parse_ctx);
    }
    if ((c = get_char()) != 0 && c != '}')  // 13
    {
      throw format_error(
        "Invalid map format specification");
    }
    return iter;
  }
  auto format(const map<K,V>& vals,
    format_context& format_ctx) const
  {
    auto pos = format_ctx.out();    // 14
    bool need_sep = false;
    for (auto val : vals)
    {
      if (need_sep)   // 15
      {
        *pos++ = m_sep;
        if (m_sep != ' ')
        {
          *pos++ = ' ';
        }
      }
      if (m_lc != '\0')   // 16
      {
        *pos++ = m_lc;
      }
      format_ctx.advance_to(pos);     // 17
      pos = m_key_fmt.format(val.first,
        format_ctx);
```

Listing 3 (cont'd)

```
      *pos++ = m_con;                  // 18
      format_ctx.advance_to(pos);      // 19
      pos = m_val_fmt.format(val.second,
        format_ctx);
      if (m_rc != '\0')    // 20
      {
        *pos++ = m_rc;
      }
      need_sep = true;
    }
    return pos;
  }

private:
  char m_lc = '\0';
  char m_rc = '\0';
  char m_sep = ' ';
  char m_con = '=';
  formatter<K> m_key_fmt;
  formatter<V> m_val_fmt;
};

int main()
{
  map<int, string> map1{ {1, "a"}, {2, "bc"},
    {3, "def"} };
  cout << format("{}\n", map1);            // a
  cout << format("{:w[]}\n", map1);        // b
  cout << format("{:s,}\n", map1);         // c
  cout << format("{:c:}\n", map1);         // d
  cout << format("{:w[]c:s,}\n", map1);    // e
  cout << format("{:w[]/{}{5}}\n", map1); // f
  cout << format("{:s;/{3}{5}}\n", map1); // g
  cout << format("{:s;/{3}{}}\n", map1);  // h
}
```

Listing 3 (cont'd)

The **if**-statements starting at lines 3 and 5 read the **w** and **s** elements, just as the corresponding lines do for **std::vector**. The **if**-statement starting at line 4 reads the **c** element, which must have a single character following it.

The **if**-statement starting at line 6 handles any nested *format-specs* defined. As mentioned previously, they are mandatory if the **/** character is present.

Line 7 checks for the **{** that indicates the start of the *key-fmt-spec*, and if not present throws a **format_error**. We just report a generic error text here, but obviously a more expressive text would help the user find the error quicker.

Line 8 uses the **advance_to** function to set up the iterator in **parse_ctx**. Note that we increment the value passed in as we need to skip the **{** detected in the previous line, which is not part of the *key-fmt-spec*. Line 9 then calls **m_key_fmt.parse** so the **formatter** for the key type can parse the *key-fmt-spec*. Finally, line 10 checks that the *key-fmt-spec* is correctly terminated with a **}** character.

The code starting at line 11 then does the same work, but for the value type, using the **m_val_fmt** member variable.

If the condition in line 6 is false it means we don't have format specifications for the key or value types. Line 12 checks if we have reached the end of the *format-spec* for the map, and if so the controlled lines call the **parse** functions on **m_key_fmt** and **m_val_fmt** to set them to their defaults.

```
a: 1=a 2=bc 3=def
b: [1=a] [2=bc] [3=def]
c: 1=a, 2=bc, 3=def
d: 1:a 2:bc 3:def
e: [1:a], [2:bc], [3:def]
f: [1=a    ] [2=bc   ] [3=def  ]
g:   1=a    ;   2=bc   ;   3=def
h:   1=a;   2=bc;   3=def
```

Listing 4

Finally, line 13 does the usual check to make sure we have reached the end of the *format-spec*.

The format function

The **format** function for **std::map** is similar to the one for **std::vector** given previously.

Line 14 picks up the current output iterator from **format_ctx**. The function then enters a loop over all the values in the map.

Line 15 checks if we need to output a separator character, and if so the controlled block does that work. Line 16 then does the same for the left-hand wrapper character.

Line 17 then sets the output iterator in **format_ctx** to the now-current value, and the following line uses **m_key_fmt.format** to output the key, returning the new value of the output iterator. Line 18 then outputs the onnector character.

Line 19 updates the **format_ctx** output iterator again so the following line can output the value using **m_val_fmt.format**.

Line 20 then outputs the right-hand wrapper character, if required.

Test cases

Test case **a** checks that the default formatting works for **map** and its contained key-value pairs.

Test cases **b**, **c**, **d**, and **e** check that the various parts of the **map**'s *format-spec* work correctly, singly and in combination.

Test cases **f**, **g**, and **h** test that using *format-specs* for the key and value parts works, including that using default *format-specs* is allowed.

Summary

In this article we have shown how you can write a formatter for a container type, or any other class where the types of some elements are unknown to you when writing the formatter because they are specified by the user of the class.

In the next and final article of this series I will show you how to create format wrappers, special purpose classes that allow you to apply specific formatting to existing classes. ■

References

[Collyer24] Spencer Collyer 'User-Defined Formatting in **std::format**: Part 1', *Overload* 180, April 2024, available at https://accu.org/journals/overload/32/180/collyer/

Reverse-Engineering cuBLAS

It's possible to achieve cuBLAS performance with tensor cores by mimicking SASS instructions. Fabian Schuetze guides us through the process.

Importance of GEMM and GPUs

Matrix multiplication is at the heart of linear algebra and the core of scientific, engineering, and statistical computation. Many variants of matrix multiplication can be expressed to interface with the Basic Linear Algebra Subprograms (BLAS). The BLAS is the de facto standard low-level interface for matrix multiplications, and its influence is hard to overstate. For example, *Nature* named the BLAS one of ten computer codes that transformed science [Perkel21]. Moreover, Jack J. Dongarra received the Turing Award in 2021 [ACM21] as:

> the primary implementor or principal investigator for [...] BLAS. [...] The libraries are used, practically universally, for high performance scientific and engineering computation on machines ranging from laptops to the world's fastest supercomputers.

Finally, with C++26, programmers can interface with the BLAS directly from C++ (under the `std::linalg` namespace), thanks to P1637.

Because the low-level interface for matrix multiplication adheres to a de facto standard and its importance, hardware vendors offer dedicated implementations. These libraries are highly optimized, but their source code is often undisclosed. Matrix multiplications comprise many small and independent computations and are well-suited for GPUs. Consequently, AMD, ARM, Nvidia, and Intel offer libraries for their GPUs. GPUs are, in essence, vector processors. They have simple (compared to modern CPUs) but enormous numbers of cores. Their memory units are also simple but provide huge throughput. To attain maximum performance, programmers commonly explicitly control data loading into caches.

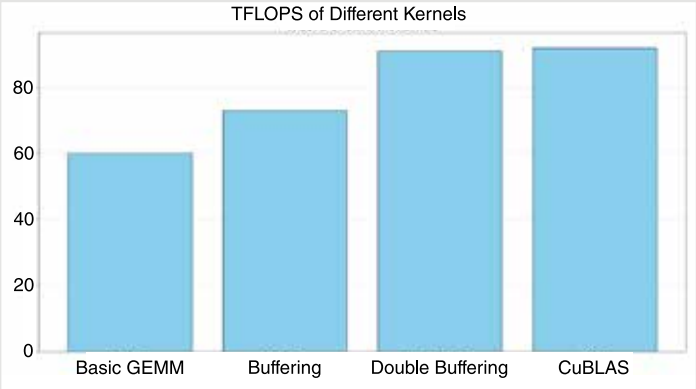


Figure 1

This article extracts the essence of such computations by reverse-engineering a matrix multiplication with Nvidia's BLAS library (cuBLAS). The implementation is simple yet instructive and attains performance almost on par with the cuBLAS variant. Re-engineering the cuBLAS kernel is not too difficult when using good abstractions as building blocks. The kernels provided with cuBLAS are heavily tuned, and the best-performing kernel gets selected at runtime. The runtime chooses among many kernels. One can count ~5000 kernels containing GEMM in its name, and cuBLAS ships a whopping 100MB. In comparison, the BLAS library provided by Ubuntu, libblas, ships 600KB.

The performance of three different handwritten CUDA kernels and the cuBLAS version is shown in Figure 1.

The three versions differ in their use of PTX (which can be understood as a mid-level IR for Nvidia GPUs) primitives and the degree of instruction-level parallelism (ILP) attained. A high ILP can be achieved by writing efficient abstractions and placing them well in the code to permit prefetching and avoiding pipeline stalls. PTX Modern PTX instructions need to be used to permit asynchronous and highly efficient loading of global memory. This efficiency is documented by the kernels ILP, which is shown in Figure 2 (overleaf).

Note, for users used to CPU optimization, the ILP is extremely high, which is explained by the extensive parallelism GPUs offer.

This article proceeds in the following stages: First, the basic GEMM implementation using Tensor cores is shown. Second, the SASS (CUDA assembly) code for the highly optimized CUDA kernel is analyzed, and

Glossary

A5000 (GPU): A GPU produced by Nvidia. The A5000 is based on the Ampere microarchitecture. The article uses specialized instructions introduced with Ampere. The subsequent microarchitecture (Hopper) introduced new instructions to attain maximum performance on these types of GPUs.

BLAS (and **GEMM**): GEMM stands for General Matrix Multiplication. Refers to a group of operations (called Level 3) of the Basic Linear Algebra Subprograms (BLAS) too. A standardized interface to BLAS will become part of C++ 26 (std::linalg) as proposed by P1673.

cuBLAS: Nvidia's variant of the BLAS library. It contains highly optimized and specialized code for all GPU variants and matrix sizes. Its source code is not publicly accessible.

CUDA: An extension of the C language to write programs for Nvidia GPUs. CUDA affords programmers the ability to control the L1 cache of such GPUs.

PTX: PTX (Parallel Thread Execution) describes an idealized virtual machine depicting an archetypical Nvidia GPU and its corresponding instruction set architecture (ISA). Cuda code also compiles to PTX, which gets further translated to (undocumented) SASS code. Programmers can also write PTX code.

SASS: An undocumented assembly language for Nvidia GPUs. It translates to binary microcode that gets executed on an actual target.

Fabian Schuetze Fabian works on computer vision and AI in the automotive and robotics industry. When not working, he's enjoying running or drinking wine, though not at the same time. Fabian can be contacted at fschuetze0@gmail.com

Re-engineering the cuBLAS kernel is not too difficult when using good abstractions as building blocks

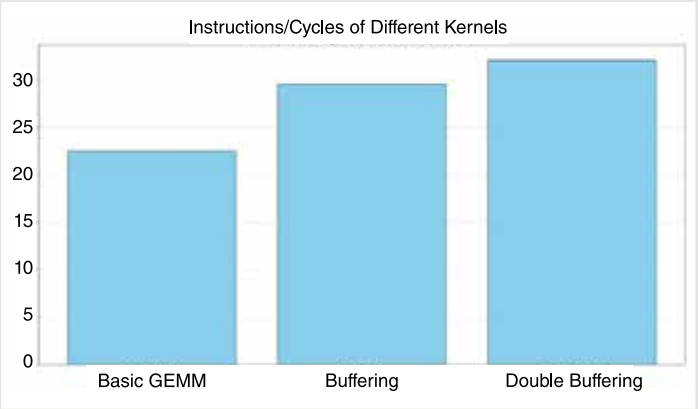


Figure 2

differences between the instructions of the basic implementation are identified. The basic implementation is refined in two steps to reach performance parity with cuBLAS.

Basic GEMM Implementation

The main loop of the basic implementation of a GEMM kernel with tensor cores is in Listing 1. This documents the basic structure of a decent GEMM kernel with tensor cores: Looping along the K (inner) dimension of the matrix product in blocks, the kernel loads blocks of the matrices A and B into shared memory. The load function is named `load_blocking` (which already provides a glimpse at future optimizations). The kernel then uses a nested loop to compute the matrix product over these blocks. Smaller blocks of the shared memory get loaded into local register files, and their matrix product gets calculated. The kernel reaches about 60TFLOPS on an A5000, or ⅔ of the GPU limit.

The code in Listing 1 gets compiled to the following SASS assembly:

```
...
LDG.E.128.CONSTANT R72, [R72.64]
...
WARPSYNC 0xffffffff
...
STS.128 [R143], R52
...
BAR.SYNC 0x0
LDSM.16.M88.4 R80, [R80]
...
HMMA.16816.F16 R18, R80, R68, R18
...
BAR.SYNC 0x0
```

The assembly reveals the inner workings of the code above: First, `load_blocking` stores 128 bits from global memory into thread-local registers. After the global loads, all threads in the warp wait at a barrier. Then, the threads store the loaded data in shared memory, and all threads in a block sync. Furthermore, data from shared memory is loaded as a matrix for processing by the tensor cores. Then, a tensor core matrix multiplication with half-floats ensues. Finally, all threads in the block

```
for (size_t block = 0; block < K; block +=
Threadblock::kK) {
  LoaderA.load_blocking();
  LoaderB.load_blocking();
  LoaderA.next(Threadblock::kK);
  LoaderB.next(Threadblock::kK * N);
  __syncthreads();
  constexpr size_t wmma_steps
    = Threadblock::kK / WMMAblock::kK;
  for (size_t wmma_step = 0;
       wmma_step < wmma_steps; wmma_step++) {
    RegisterLoaderA.load();
    RegisterLoaderB.load();
    RegisterLoaderA.step(WMMAblock::kK);
    RegisterLoaderB.step
      (Bs.cols_ * WMMAblock::kN);
    matmul.compute();
  }
  RegisterLoaderA.reset(0);
  RegisterLoaderB.reset(0);
  __syncthreads();
}
```

Listing 1

wait at a barrier before the loop starts again. The way data is loaded is pictured in the graph in Figure 3 (overleaf).

From the very right, 255MB are loaded from device memory to the L2 Cache before landing in the L1 Cache. As can be seen in the top left of the figure, there are 3.41M instructions used to load data into the local registers. From the local registers, the data is stored again in the shared memory (a portion of the L1 cache) in 3.15M requests. From the shared memory, the data gets accessed in 11.53M requests.

SASS code for cuBLAS assembly code

The SASS code for the cuBLAS kernel is interesting. An abbreviated version reads as follows:

```
HMMA.16816.F32 R0, R152, R184, R0
LDSM.16.MT88.4 R168, [R137+UR8+0x800]
LDGSTS.E.BYPASS.LTC128B.128.CONSTANT
[R129+UR4+0x3000], [R130.64+0x180]
...
HMMA.16816.F32 R4, R152, R186, R4
HMMA.16816.F32 R8, R152, R188, R8
...
HMMA.16816.F32 R120, R164, R196, R120
DEPBAR.LE SB0, 0x1
...
```

The assembly code highlights several aspects: The main loop starts with a matrix multiplication instead of a memory load. The global load `LDGSTS.E.BYPASS.LTC128B.128.CONSTANT` differs from the load in the basic GEMM implementation, `LDG.E.128.CONSTANT R72`: Firstly, it bypasses the register and stores the data directly in shared memory. Furthermore, it is an asynchronous load and does not block the threads. Non-blocking requires a separate memory fence to signal when the data is ready. Such a barrier is the dependency barrier `DEPBAR.LE`.

The gift of asynchronous copy operations is that one can overlay computation with memory transfers and avoid pipeline stalls

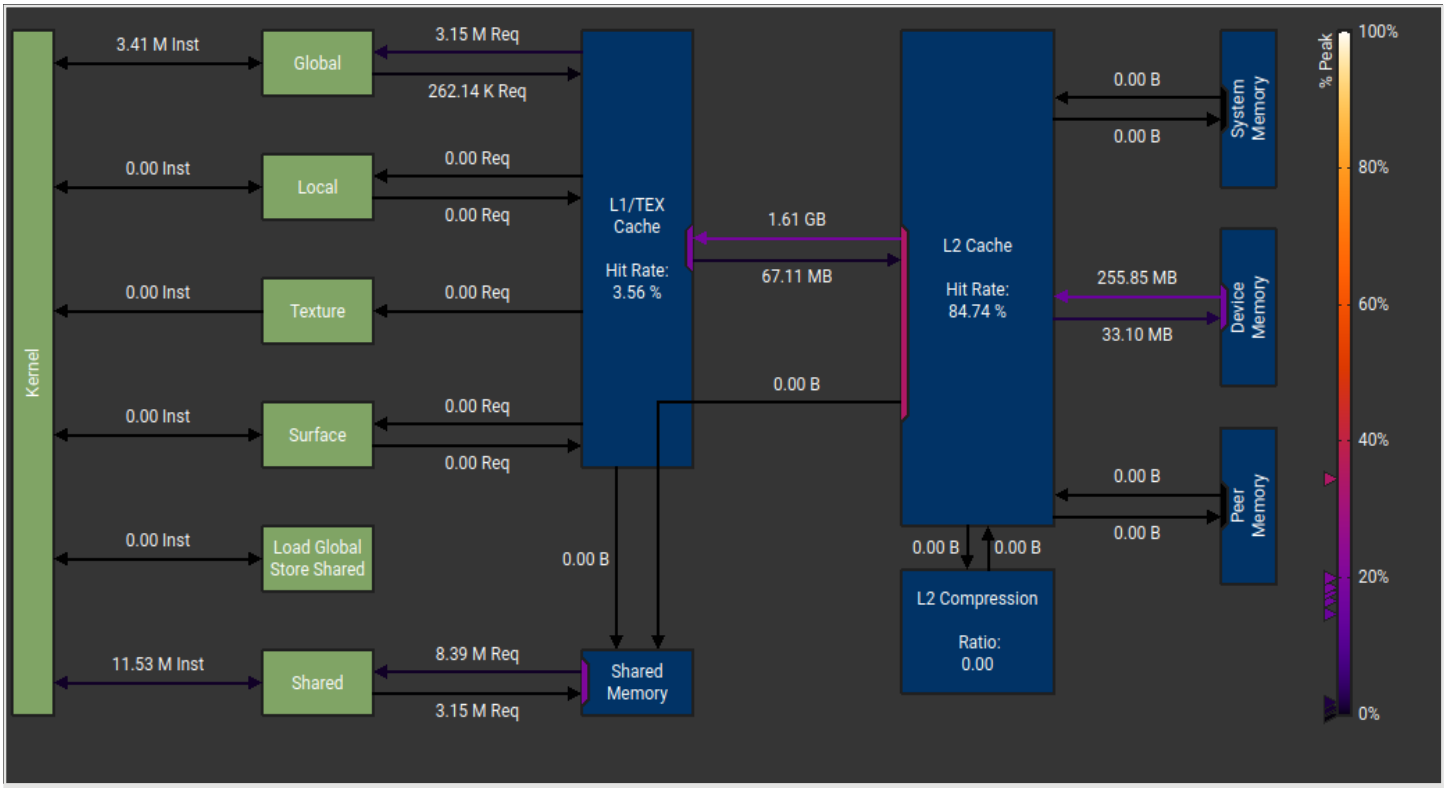


Figure 3

Finally, the instructions are interleaved: There is no linear separation between loading data and operating on it, but a heavy mixture of instructions. The cuBLAS kernel achieves ~90TFLOPS. The following two kernels describe how to write code that produces similar SASS and attains the same performance.

Improvement I: buffering

Asynchronous load instructions

Starting with PTX Version 7.0 [PTX-1], CUDA provides instructions to copy data asynchronously from global to shared memory. The copy bypasses local registers and stores data directly to the shared memory (L1 cache). As identified above, asynchronous loading is one of the differences between the simple GEMM code and the cuBLAS version.

Two changes are necessary for asynchronous loading. First, the new load function is in Listing 2. What is was before is shown in Listing 3 (overleaf).

The `load_blocking` function loads 128bit by casting eight half floats as an int4 and loads it. In contrast, the `load` function uses the macro `CP_ASTNC_CG` comprising the PTX instructions in Listing 4 (also overleaf).

The compiler converts it into the same SASS instruction as can be seen in the cuBLAS code:

```
LDGSTS.E.BYPASS.LTC128B.128 [R11], [R2.64]
```

Because the load is non-blocking, a separate memory fence is needed to synchronize the threads. As stated in the PTX manual [PTX-2],

```
__device__ void load(size_t counter) {
  const size_t global_idx =
    offset_.row * ld_ + offset_.col;
  for (size_t row = 0; row < rows;
       row += stride_) {
    const T *src =
      global_ptr_ + row * ld_ + global_idx;
    T *dst =
      &shmem_(counter * rows + offset_.row + row,
              offset_.col); // + row * cols;
    constexpr size_t load_bytes = 16;
    uint32_t pos_in_ss = __cvta_generic_to_shared
      (reinterpret_cast<int4 *>(dst));
    CP_ASYNC_CG(pos_in_ss, src, load_bytes);
  }
}
```

Listing 2

```
__device__ void load_blocking() {
  const size_t global_idx =
    offset_.row * ld_ + offset_.col;
  for (size_t row = 0; row < rows;
       row += stride_) {
    const T *src =
      global_ptr_ + row * ld_ + global_idx;
    T *dst = &shmem_(offset_.row + row,
      offset_.col); // + row * cols;
    const int4 t =
      reinterpret_cast<const int4 *>(src)[0];
    reinterpret_cast<int4 *>(dst)[0] = t;
  }
}
```

Listing 3

```
#define CP_ASYNC_CG(dst, src, Bytes)          \
  asm volatile(                               \
    "cp.async.cg.shared.global.L2::128B [%0],"  \
    "[%1], %2;\n" ::"r"(dst), "l"(src),       \
    "n"(Bytes))
```

Listing 4

asynchronous copies need to be committed to a group and waited for. The following two macros, comprising PTX instructions, do exactly that:

```
CP_ASYNC_COMMIT_GROUP();
CP_ASYNC_WAIT_GROUP(0);
```

These two macros get compiled into the following SASS code:

```
LDGDEPBAR
DEPBAR.LE SB0, 0x0
```

These two SASS instructions are found in the cuBLAS code too. The slight difference between the two is covered in the next section. Visualizing the new load instruction `LDGSTS.E.BYPASS.LTC128B.128` is very instructive (see Figure 4). The data goes directly from the L2 Cache through the shared memory (a portion of the L1 cache).

Overlapping memory loads with computation

The gift of asynchronous copy operations is that one can overlay computation with memory transfers and avoid pipeline stalls. The kernel can be expressed as shown in Listing 5 (overleaf).

The computation starts by loading data from global to shared memory. The class loading data from shared to global memory manages two buffers. Data gets read from one buffer and stored in the other buffer. The main loop begins by initiating a global memory load. The matrix elements are then computed. Afterward, the threads block until the previously fetched memory has been loaded. In the loop's epilogue, the last outstanding matrix computation is conducted.

This kernel attains 73 TFLOPS, a 20 percent increase to the first kernel.

Improvement II: double buffering

The code above already improves the throughput of the kernel. However, it is still below the cuBLAS version, and the assembly instructions do not match. In particular, the memory barrier in the code above is `DEPBAR.LE SB0, 0x0`, but the memory barrier in the cuBLAS code is `DEPBAR.LE SB0, 0x1`. The SASS instructions are undocumented, but one can assume that LE stands for less or equal. Furthermore, the PTX docs for the memory barrier [PTX-3] state that the PTX instruction `cp.async.wait_group N` is:

> cp.async.wait_group instruction will cause the executing thread to wait till only N or fewer of the most recent cp.async-groups are pending and all the prior cp.async-groups committed by the executing threads are complete.

Besides the difference in instructions, the kernel above also regularly stalled because data was unavailable. The warps stalled for almost two cycles for each issued instruction because data was unavailable (long scoreboard stall). To avoid such stalls and replicate the SASS code for the cuBLAS kernel, the kernel below does "double buffering": Always have two shared memory operations in flight and await only the oldest one. Register loads are buffered too. The kernel has one register file loaded, loads the next one, and computes the matrix operation on the previous register file. The code for the kernel is in Listing 6 (opposite).

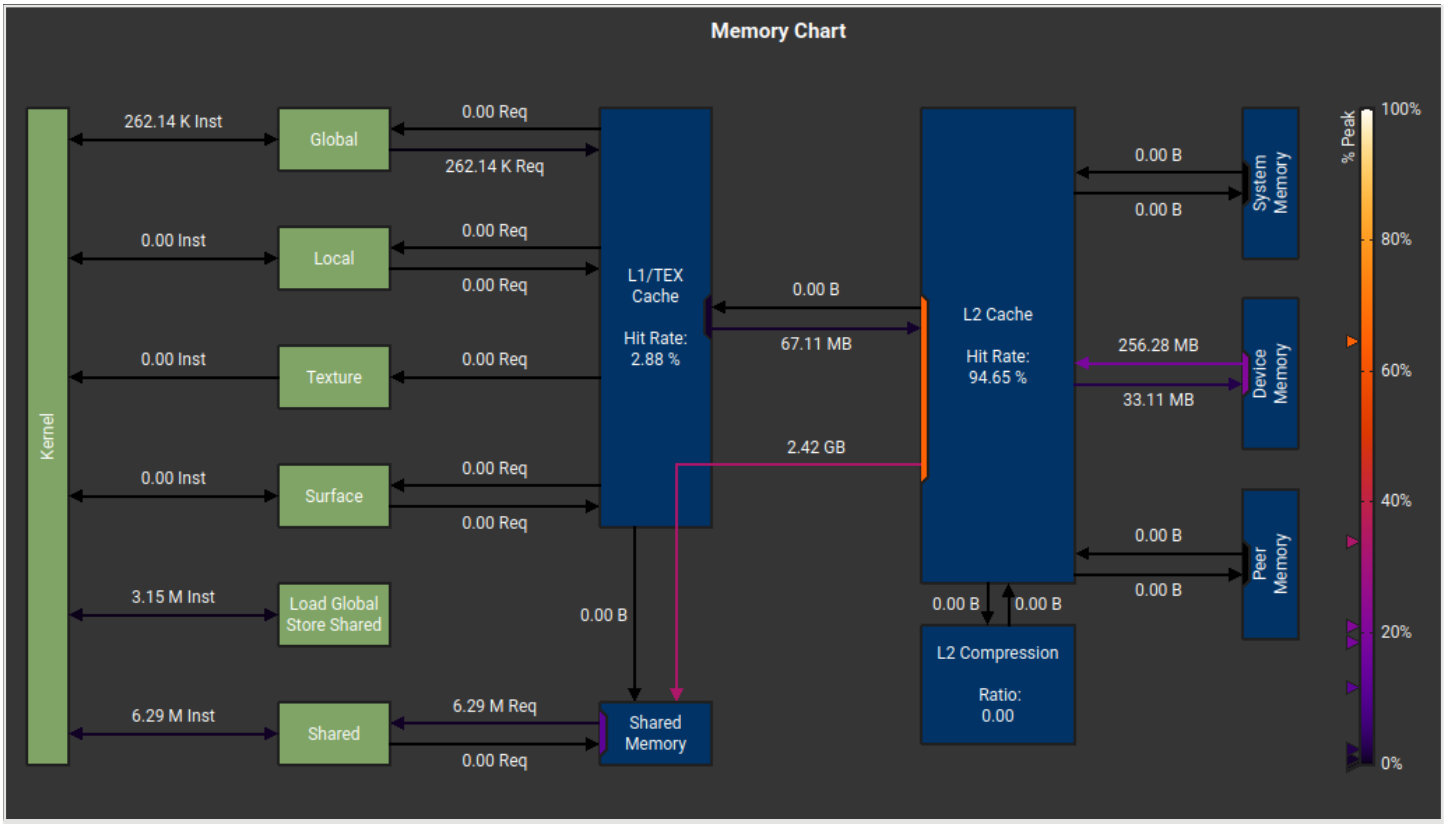


Figure 4

```
size_t counter = 0;
LoaderA.load(counter);
LoaderB.load(counter);
LoaderA.next(Threadblock::kK);
LoaderB.next(Threadblock::kK * N);
CP_ASYNC_COMMIT_GROUP();
CP_ASYNC_WAIT_GROUP(0);
__syncthreads();
for (size_t block = 0;
  block < K - Threadblock::kK;
  block += Threadblock::kK) {
  LoaderA.load(counter ^ 1);
  LoaderB.load(counter ^ 1);
  LoaderA.next(Threadblock::kK);
  LoaderB.next(Threadblock::kK * N);
  constexpr size_t wmma_steps =
    Threadblock::kK / WMMAblock::kK;
  for (size_t wmma_step = 0;
      wmma_step < wmma_steps; ++wmma_step) {
    RegisterLoaderA.load();
    RegisterLoaderB.load();
    RegisterLoaderA.step(WMMAblock::kK);
    RegisterLoaderB.step
      (Bs.cols_ * WMMAblock::kN);
    matmul.compute();
  }
  counter ^= 1;
  RegisterLoaderA.reset(counter *
    Threadblock::kM * (Threadblock::kK + skew));
  RegisterLoaderB.reset(counter *
    Threadblock::kK * (Threadblock::kN + skew));
  CP_ASYNC_COMMIT_GROUP();
  CP_ASYNC_WAIT_GROUP(0);
  __syncthreads();
}
for (size_t bk = 0; bk < Threadblock::kK;
     bk += WMMAblock::kK) {
  RegisterLoaderA.load();
  RegisterLoaderB.load();
  RegisterLoaderA.step(WMMAblock::kK);
  RegisterLoaderB.step(Bs.cols_ * WMMAblock::kN);
  matmul.compute();
}
```

Listing 5

The prologue to the main loop begins by issuing two shared memory loads. The threads block until the first load is completed, while the second one remains in flight. Then, the first register file is loaded. The main loop begins by loading a further fragment of shared memory, and the tensor cores operate on the previous fragment. When all local registers are filled, the shared memory of the first block has been exhausted. No computation can be overlaid over the memory copies anymore. Another load is issued, and the warps wait until the previous load is completed.

With these advances, the throughput of the kernel advances to 89 TFLOPS and reaches within 95% of cuBLAS performance. Further gains can be reaped by writing the result of the multiplication through shared memory back to global memory. The kernel throughput then advances to 91 TFLOPS, 1 TFLOP behind the cuBLAS kernel. ■

References

[ACM21] ACM Turing Award 2021: available at https://awards.acm.org/about/2021-turing

[Perkel21] Jeffrey M. Perkel 'Ten computer codes that transformed science', published on Nature website 20 January 2021 (last updated 8 April 2021), available at https://www.nature.com/articles/d41586-021-00075-2.

[PTX-1] PTX Version 7.0 documentation, 'Changes in PTX ISA Version 7.0', published by NVIDIA, available at https://docs.nvidia.com/cuda/parallel-thread-execution/index.html#changes-in-ptx-isa-version-7-0


```
LoaderA.load(0);
LoaderB.load(0);
LoaderA.next(Threadblock::kK);
LoaderB.next(Threadblock::kK * N);
CP_ASYNC_COMMIT_GROUP();
LoaderA.load(1);
LoaderB.load(1);
LoaderA.next(Threadblock::kK);
LoaderB.next(Threadblock::kK * N);
CP_ASYNC_COMMIT_GROUP();
CP_ASYNC_WAIT_GROUP(1); // 1 = Wait until 1
         // recent async groups are pending
__syncthreads();
RegisterLoaderA.load(0);
RegisterLoaderB.load(0);
RegisterLoaderA.step(WMMAblock::kK);
RegisterLoaderB.step
  (SpanTypeB::cols_ * WMMAblock::kN);
size_t counter = 1;
for (size_t block = 0; block < K - 2 *
Threadblock::kK;
  block += Threadblock::kK) {
    constexpr size_t wmma_steps =
      Threadblock::kK / WMMAblock::kK;
    for (size_t i = 0; i < wmma_steps; ++i) {
      size_t current = i % 2;
      size_t next = (i + 1) % 2;
      RegisterLoaderA.load(next);
      RegisterLoaderB.load(next);
      RegisterLoaderA.step(WMMAblock::kK);
      RegisterLoaderB.step
        (SpanTypeB::cols_ * WMMAblock::kN);
      matmul.compute(current);
      if (i == 0) {
        LoaderA.load(counter ^ 1);
        LoaderB.load(counter ^ 1);
        LoaderA.next(Threadblock::kK);
        LoaderB.next(Threadblock::kK * N);
        CP_ASYNC_COMMIT_GROUP();
        CP_ASYNC_WAIT_GROUP(1);
        __syncthreads();
        RegisterLoaderA.reset
          (counter * MemLoaderA::size_);
        RegisterLoaderB.reset
          (counter * MemLoaderB::size_);
        counter ^= 1;
      }
    }
  __syncthreads();
}
```

Listing 6

[PTX-2] PTX Version 7.0 documentation, 'Data Movement and Conversion Instructions: Asynchronous copy', published by NVIDIA, available at https://docs.nvidia.com/cuda/parallel-thread-execution/index.html?highlight=async#data-movement-and-conversion-instructions-asynchronous-copy

[PTX-3] PTX Version 7.0 documentation, 'Data Movement and Conversion Instructions: cpl.async.wait_group/cp.async.wait_all', published by NVIDIA, available at https://docs.nvidia.com/cuda/parallel-thread-execution/index.html?highlight=async#data-movement-and-conversion-instructions-cp-async-wait-group-cp-async-wait-all

This article was previously published on github by Fabian on 14 March 2024, and is available at https://fabianschuetze.github.io/category/articles.html


World-class conference
Professional development
Printed journals
Local groups
Email discussion groups
Reviews of technical books

Individual membership from £35

Corporate membership from £175

Visit accu.org for details

Fat API Bindings of C++ Objects into Scripting Languages

How do you expose a C++ object to a TypeScript layer or other scripting language? Russell K. Standish demonstrates an approach using a RESTService API that is scripting-language independent.

A *fat* API exposes nearly all of a C++ object's public attributes and methods to a consuming environment, such as a scripting language, or web client. This can be contrasted with a conventional, or *thin* API, where the API is defined up front, and the C++ object provides the implementation, most of which is private to the C++ layer.

Obviously, reflection is required to expose C++ objects to a consuming layer like this – this paper explores using the Classdesc system to implement reflection of C++ objects into a JavaScript/TypeScript environment via a REST service, and also via a Node.js API module.

Introduction

Minsky [Standish] is a *systems dynamics* [Forrester07] simulation package, with an orientation towards economics, that has been under continual development since 2011. It is implemented in C++, and historically the user interface was implemented using the TCL/Tk toolkit [Ousterhout94], with C++ bindings provided by the EcoLab [Standish01, EcoLab] library.

From 2019–2021, the TCL/Tk layer was completely reimplemented in TypeScript [Cherny19, Goldberg22], on top of the Angular [Green13] and Electron [Kredpattanakul19] toolkits, running in the Node.js [Ihrig14] interpreter. The advantages to doing this include accessing a much larger ecosystem of 3rd party components, a much larger pool of programmers (JavaScript is consistently in the top 10 of programming languages according to the Tiobe index [Tiobe]), and potentially longer term an in-browser version of the code could be enabled via technologies such as WebASM [Haas17].

This paper reports on the subtask of exposing the Minsky's C++ core to the TypeScript layer, allowing C++ objects to be manipulated in a seamless manner in TypeScript code. The approach is quite general, and could be readily adapted to other language binding APIs, or even without an explicit binding API by means of a REST service that can be accessed with an HTTP client implementation.

REST service

REST (REpresentational State Transfer) [Fielding00] is based on web technologies. The part of a URL after the domain, such as

```
http://www.somewhere.com/path/to/page
```

is called the URL's *pathinfo*. In REST terminology, it is called an *endpoint*, and represents a resource. What to do with the resource is given by the HTTP verb of the request. A web browser typically performs a GET request when you type a URL into its address bar, but there are verbs covering all of the *CRUD* operations (create, read, update and delete):

- **POST** create an object at the resource location
- **GET** read an object at the resource location
- **PUT** update the object
- **DELETE** destroy the object

In something like an EcoLab model, or the Minsky project, there is a global static object that holds the state of the model. In the C++ code, this is accessible via a Meyer singleton pattern, ie the `minsky()` function. So for example, a REST GET call on `/minsky/t` returns the value of the current timestep of the Minsky model, and performing a PUT, with floating point data in the HTTP request body, updates the timestep to the supplied value. For convenience, the Minsky REST service ignores whether a PUT or GET is used, using the presence or absence of HTTP body data to determine whether the operation is an update or a read.

One can also map method calls into the same schema. For example `/minsky/reset` calls the reset method, which has no arguments. The above schema for reading or updating an attribute could be considered an example of calling an implied overloaded getter/setter method, with overload resolution determined by the presence or absence of data in the request body. Since we're targeting the JavaScript ecosystem, it is natural to use JSON [ECMA13] to encode the parameters being passed, and the return value. Compound objects can be serialised to/from JSON using Classdesc's existing JSON serialiser into a JSON object (delimited by braces). Calling a method with more than one parameter can be achieved by placing the JSON representation of the arguments in a JSON array, which conveniently are allowed to be of different types. So the command to export a LATEX document describing the model's differential equation, which has signature

```
void latex(const std::string& fileName,
  bool wrapLaTeXLines)
```

can be called through the REST service as

```
/minsky/latex ["foo.tex", true]
```

where the first space delineates the pathinfo and request body.

Whilst JSON is used for data encoding in this example, it is perfectly possible to use alternate encodings. The `RESTProcess_t`[1] *descriptor*[2] object has a method:

```
REST_PROCESS_BUFFER RESTProcess_t::process
  (const std::string& pathinfo,
   const REST_PROCESS_BUFFER& body);
```

where `REST_PROCESS_BUFFER` is a macro representing the 'buffer' concept, which defaults to `json_pack_t`. A buffer implements:

- `REST_PROCESS_BUFFER::operator>>(T&)` for deserialisation to an arbitrary type

Russell K. Standish Russell gained a PhD in Theoretical Physics, and has had a long career in computational science and high performance computing. Currently, he operates a consultancy specialising in computational science and HPC, with a range of clients from academia and the private sector. You can contact him at hpcoder@hpcoders.com.au

1 Released in Classdesc 3.43, available from https://classdesc.sourceforge.net, or https://github.com/highperformancecoder/classdesc.

2 In the Classdesc reflection system [Madina01], a *descriptor* is an overloaded set of function definitions that is mostly automatically generated by the Classdesc processor for each type used in the program.

C++ provides for overload resolution based on types as well as number of arguments; JavaScript does not provide for overloaded functions at all

- `REST_PROCESS_BUFFER::operator<<(const T&)` for serialisation of an arbitrary type
- `RESTProcessType REST_PROCESS_BUFFER::type()` which refers to the type of the object serialised in the buffer
- `REST_PROCESS_BUFFER::Array REST_PROCESS_BUFFER::array() const` returns a sequence concept object (eg `std::vector` or `std::deque`) if called on a `REST_PROCESS_BUFFER` that is an array, or usually an empty sequence if not.

  `REST_PROCESS_BUFFER::Array::operator[](size_t)` returns a `REST_PROCESS_BUFFER`.

The `RESTProcess_t` type is a map, where the keys are the endpoints of the fat API, and the values are wrappers around the C++ object, or method. These wrappers are polymorphic, with different implementations depending on whether it is an object or a method, smart pointer or container type. The interface is shown in Listing 1.

The reason `REST_PROCESS_BUFFER` is a macro rather than a template argument, is because `RESTProcessBase` is polymorphic, and C++ does not allow templated virtual functions.

The methods `signature`, `list` and `type` provide a modicum of introspection to allow exploration of the fat API from the calling side. `signature` returns an array containing the return type and types of all arguments.

Node.js API

Minsky's C++ layer renders directly to a native window for performance reasons. Electron's BrowserWindow class has a native window handle getter method that can be used to pass the native window to the C++ layer. The strategy described in the previous section of making the C++ implementation a REST service worked well for Windows, where the native window handles are system wide, and X-Windows system, which is distributed by design, but unfortunately failed for the MacOSX architecture. It turns out that Mac native window handles are actually pointers which are, of course, only meaningful within the same process address space.

```
class RESTProcessBase
{
public:
virtual ~RESTProcessBase() {}
/// perform the REST operation, with \a remainder
being the
/// query string and \a arguments as body text
virtual REST_PROCESS_BUFFER process(const string&
remainder,
const REST_PROCESS_BUFFER& arguments)=0;
/// return signature(s) of the operations
virtual REST_PROCESS_BUFFER signature() const=0;
/// return list of subcommands to this
virtual REST_PROCESS_BUFFER list() const=0;
/// return type name of this
virtual REST_PROCESS_BUFFER type() const=0;
};
```

Listing 1

So the C++ layer needed to be implemented as a dynamic library, and linked within the Node.js process using the Node.js API. Conceptually, this is quite simple, implementing a single Node.js API endpoint (call) that takes the pathinfo and body arguments as above. Of course, it hasn't stayed simple – the Node.js API allows for callbacks into the JavaScript world from C++, which is important for some interactive functionality; as well as also allowing offloading of C++ processing to a separate thread, and returning the results via a JavaScript *promise*, which is important for not blocking the user interface during long-running backend operations.

Attributes and methods

We map C++ public attributes to an implied pair of overloaded setter/getter methods. If an argument is provided to the method, a setter is called, and the argument assigned to the attribute. For the Minsky project, JSON encoding of the attribute is performed, using the existing `json_pack` and `json_unpack` descriptors.

This is a very simple example of a method overload. However, C++ provides for overload resolution based on types as well as number of arguments. JavaScript does not provide for overloaded functions at all, but with type introspection built into the language, it is possible to write a method that can dispatch to different implementations based on types and number of arguments. However, with an impoverished set of types compared with C++, this leaves us with the problem of how to match a particular JavaScript call with a C++ method.

The approach taken in this work is to walk the C++ argument list for each overloaded C++ method (Classdesc has been able to address overloaded methods since version 3.37 [Standish19]), and add a penalty for each argument that doesn't quite match. For instance, if the JavaScript environment passes a number with a non-zero fractional part, then an integer argument C++ will receive a small penalty, but a float or double parameter does not. If there are fewer arguments passed than the arity of the function, or no meaningful conversion is possible, then an infinite penalty is applied. Default C++ arguments are not supported as is, but a default argument can be reimplemented as an overloaded method with fewer argument calls, delegating to the method with the full number of arguments.

Finally, the method with lowest finite penalty is called, if it is unique. Otherwise, an exception is thrown back to the JavaScript environment.

Modern C++ variadic templates are used to walk the C++ type arguments to determine the penalty values. Then to call the C++ method, *currying* is used. The JSON arguments are converted to the relevant C++ type, starting from the last argument, currying the bound method to an n-1 argument functor, where the last argument has been fixed by the converted JSON argument. It takes one walk through the C++ argument list to generate the curry functors, then the final zero argument curried functor is called, which in turn calls the curried functors up into the final bound method. The technique works well, except that each of these curried functors need

to be linked, blowing up the build time. In 'Build time optimisation' on page 18, I describe a number of techniques to reduce the build times.

TypeScript

JavaScript, being a dynamic language, only checks numbers and types of arguments at runtime. TypeScript [Cherny19, Goldberg22] is an extension of JavaScript with type annotations that are checked at compile time. For larger more complex projects like Minsky, the TypeScript compile step is an invaluable means of eliminating logic errors.

The JavaScript interface to C++ is of the form

```
call("method.name", args...);
```

which performs type checking at runtime. For Minsky, we created another *descriptor* that outputs a series of TypeScript definitions. This is not the only viable method. The REST API has sufficient introspection built in that it should be possible to build a TypeScript script that queries the REST API and emits the TypeScript definitions. However, doing it as a C++ process for the Minsky project was chosen due to greater familiarity with that environment.

For example, the **`Minsky`** class has a **`t`** double precision attribute, a complex attribute **`model`** of type **`Group`** and **`classifyOp`** method, amongst others. The custom TypeScript descriptor outputs a definition like that shown in Listing 2.

The TypeScript class **`CppClass`** provides a number of features, including the **`$prefix()`** accessor and the **`$callMethod()`** method that arranges for the named C++ method to be called on a separate thread, and returns a *promise* that is *resolved* or *rejected* with the return value or exception from the C++ method. Calling into C++ asynchronously in this way prevents the C++ code from blocking the GUI interface if the C++ method takes a long time to run (as some do). There is also a **`$callMethodSync()`** which calls into C++ directly on the Node.js thread, which is useful when you need to call C++ from a non-asynchronous function – such as at application startup. Note the use of the **`$`** character in the identifier, which is a valid character in JavaScript identifiers, but not C++, so preventing any possibility of a name clash with C++ identifiers.

To use the class definition for any object, you just have to declare:

```
let minsky=new Minsky("minsky");
```

Then you can access the time attribute via **`minsky.t()`** or set the time attribute via **`minsky.t(10.2)`**. For the complex object **`model`** above, because one can call methods on it (eg **`minsky.model.numItems()`**), and in TypeScript identifiers cannot be both attributes and methods at the same time, setting and getting that object has to be done via the special **`$properties()`** method, ie **`minsky.model.$properties()`** returns a JavaScript object containing the public attributes of **`minsky.model`**, and **`minsky.model.$properties(object)`** sets the public attributes of **`minsky.model`** using the data contained in **`object`**.

Since **`minsky`** is a global object, this definition is already provided in the backend module. But for example, the attribute **`minsky.canvas.item`** is a polymorphic type with base type Item – it can be cast to the correct type in TypeScript via (eg)

```
export class Minsky extends CppClass {
  model: Group;
  constructor(prefix: string){
    super(prefix);
    this.model
      =new Group(this.$prefix()+'.model');
    ...
  }
  async classifyOp(a1: string): Promise<string>
    {return this.$callMethod('classifyOp',a1);}
  async t(...args: number[]): Promise<number>
    {return this.$callMethod('t',...args);}
  ...
}
```

Listing 2

```
let variable=
  new VariableBase(minsky.canvas.item);
```

then **`variable`** gets all of the additional attributes and methods of the **`VariableBase`** subclass.

Python

A Python API descriptor already exists [Standish19]. However, it has a couple of serious downsides. The first is that it requires the boost-python library, which is not available currently for the MXE cross compiler [MXE], and may never be, as it depends on the Python library being available, the codebase of which is not friendly towards cross compilation.

The second issue is just calling the Python descriptor on the minsky global object was not sufficient to create all the types required, and that additional explicit descriptor calls were required to generate all the types. This is not insurmountable – something like this approach was done with the TypeScript descriptor, but given the full fat API was available through the RESTService descriptor, it was decided to use the existing RESTService API descriptor, and write a Python interface using the low level Python C API. That way, we should be able to load the built Python module dynamic library into an unmodified running Python interpreter on Windows. As well as that, there would be no inconsistencies between the TypeScript API and the Python API.

It was relatively straightforward, following online tutorials, to implement a 'call' function that takes one or two arguments, the first being the REST function name, and the second being a JSON5 string for arguments. The second step involved creating a **`REST_PROCESS_BUFFER`** object (called a PythonBuffer) that directly marshals Python objects into their C++ counterparts without going via JSON serialisation. Of course, for simplicity, and to avoid creating yet another descriptor, complex objects (structs, classes etc) will always go via JSON serialisation. Unfortunately, this exposed a weakness in the macro approach outlined above, and the explicit instantiation of templates, which meant that at link time there was a definitional conflict between **`REST_PROCESS_BUFFER`** being a JSONBuffer and a PythonBuffer. So for now, the PythonBuffer containing the arguments is serialised to JSON before being passed to the RESTProcess, and the returned JSON string used to instantiate a PythonBuffer. Another attempt at implementing a template solution of the RESTProcess descriptor is planned.

Finally, for return values, the PythonBuffer stores the value as an appropriate Python object (PyObject) for the type, whether number, string, array or so on. For objects, a custom object is returned that has the JSON string returned by the RESTProcess stored as the attribute **`_properties`** (**`$`** is not a valid character in Python identifiers), and also new callable attributes for each method, allowing usage like:

```
r=container._elem(2).method()
```

within Python code.

Build time optimisation

As previously alluded, extensive use of variadic templates for processing overloaded functions caused a dramatic impact on compile times for the Minsky project, which went from circa 2 minutes for the TCL/Tk version (which doesn't support overloaded methods) to around 20 minutes for the JavaScript build. Profiling the build times indicated a massive increase in the time taken to link the 'executable' – in this case a dynamic library with a `.node` extension that Node.js loads as an 'add on'.

One of the identified reasons for the slowdown in linking speeds is the large number of generated template helper functions to handle introspection of functional objects. The number grows as the square of the number of arguments of the method, and linking objects is $O(n^2)$, so the link time grows as the 4th power of the number of method arguments. As noted later, the link times for standard Linux linkers is not actually too bad – in the few years since this work was started, Linux linkers have improved remarkably.

Strategy	GCC	Clang
None	1048	377
Explicit instantiation	445	287
Unrolled templates	427	291
Arity reduction	409	284

Build times for the different build time optimisations for the two different compiler toolchains.

Table 1

Linker	Version	Time (seconds)
GNU ld	2.41	4
LLVM ld (lld)	15.07	3.9
Mold	2.3	0.7
MXE ld.bfd	2.37	791

Link times for various linkers tested.

Table 2

In some way, the link strategy is quite stupid, as these helper functions only need to be used on one place in one object file, and so resolved at compile time. This suggested a strategy of privately declaring the variadic templates and explicitly instantiating them within just a single object file where they were used – unfortunately, the compiler still emitted symbols for each and every helper template, even if they're not linked to from other object files, and this technique didn't help.

So the next thing was to remove the RESTProcess `.rcd` definition files from the include headers, and include them in just one compilation unit, and explicitly instantiate the template within that compilation unit. This improved the build time quite significantly.

The next strategy tried was to do things the old-fashioned way. Instead of recursively defined variadic templates, explicit templates created by means of a shell script that creates explicit support functions for 0, 1, 2 etc arity functions up to some predefined maximum value (6 was found to be the maximum arity function present, with the renderWindow method being one of the biggest).

The final strategy was to reduce the maximum arity of the exposed methods. The simplest way to do this, given that one could pass a Javascript object which is packed and then unpacked into the C++ object via JSON, is to rollup several of the arguments into a compound object. In this way, the maximum arity was reduced to 4.

Finally, it turned out that the clang ecosystem had a much more performant compiler and linker for these purposes than the GCC ecosystem, and that template unrolling gave negligible benefit in the clang case.

Table 1 shows the build times for the various build time optimisations described in the text above, displayed graphically in figure 1. The optimisations were applied consecutively from top to bottom, so that the unrolled template method was applied to explicitly instantiated code, and so on.

The final test was to try the extremely performant *mold* linker [Ueyama]. As per Mold's README, adding the flag **`-fuse_ld=mold`** is sufficient to delegate the link step to mold. Link times were measured by building the target (minskyRESTService.node), removing just the target, leaving all the object files present, and timing how long it takes to build the target again.

As can be seen from table 2, for Linux builds, the linking time is inconsequential, well within noise, so even though Mold is blazingly fast, there is no particular advantage for this project. What isn't inconsequential is the link time for generating Windows versions of the Node.js addon, which takes over 13 minutes. Just quite why the linker is so slow for Windows is unclear, however a neat trick discovered whilst doing this benchmarking is to symbolically link the LLVM linker **`ld.lld`** to the MXE linker **`x86_64-w64-mingw32.shared-ld`**. It works just as well, and only takes around 4 seconds.

Methods

Build times were recorded using the inbuilt **`time`** command, running on a quad-core Intel(R) Core(TM) i5-1135G7, at 3.8GHz, with a Samsung 970 EVO 500GB NVMe M.2 SSD. The operating system was OpenSUSE Leap 15.5, and the compilers used: GCC 13.2.1 and Clang 15.0.7.

The codebase used was Minsky 3.3.2[3], except for the 'none' strategy above.

In explicitly instantiating the templates that define the descriptor, it is not feasible to put the code change behind a feature flag. Going back to the

3 Available from https://minsky.sourceforge.net, or https://github.com/highperformancecoder/minsky

Build times for the different build time optimisations for the two different compiler toolchains.

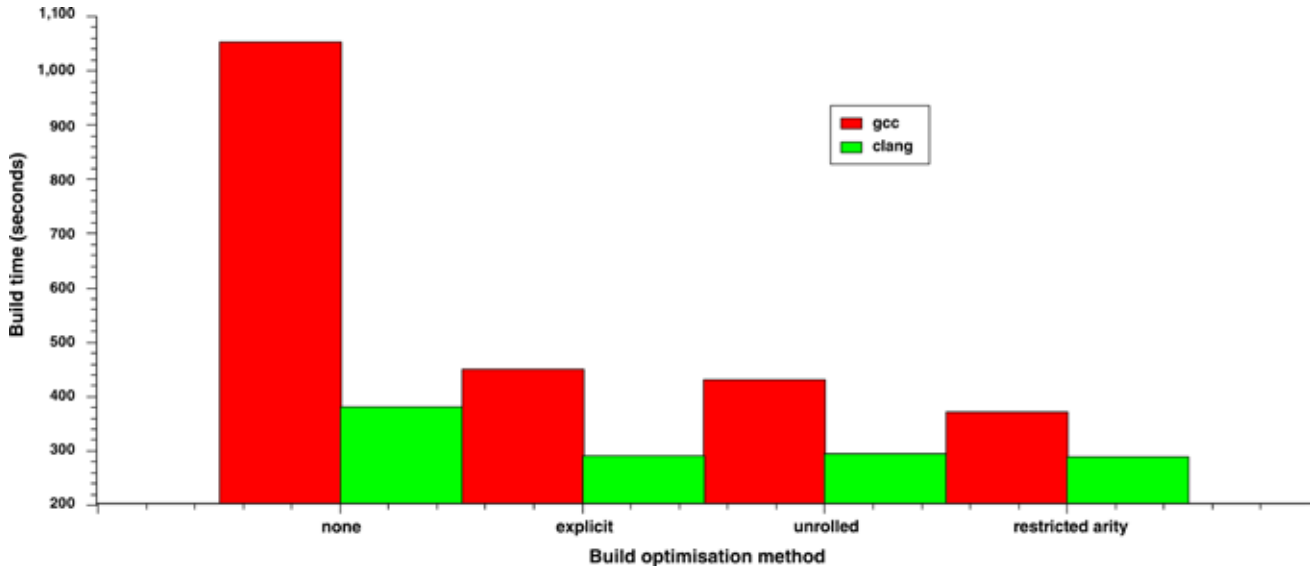


Figure 1

earlier version of the code will not be comparing apples with apples, as about a year's worth of development has occurred since that change. So the particular optimisations were backed out from the 3.3.2 codebase: the explicit instantiations removed (they were implemented in a macro, so this was easy), then the inlined descriptor definitions included back in the header files. The code changes were committed to the branch compile-optimisations-undone[4].

Particular optimisation feature flags can be turned on via Makefile flags, as shown in table 3. The command was run after an initial `make -j9` to ensure all prerequisites were built, to avoid including the prerequisites' build time. One can measure the overhead time required for `make` to start up via `make -n`, which proved to be about 1.3 seconds, so well within experimental noise.

Toolchain, Strategy	Command
GCC,none4	`rm *.i; time make -j9 GCC=1 CLASSDESC_ARITIES=`
Clang,none4	`rm *.i; time make -j9 GCC= CLASSDESC_ARITIES=`
GCC,explicit	`rm *.i; time make -j9 GCC=1 CLASSDESC_ARITIES=`
Clang,explicit	`rm *.i; time make -j9 GCC= CLASSDESC_ARITIES=`
GCC,unrolled	`rm *.i; time make -j9 GCC=1 CLASSDESC_ARITIES=0xffff`
Clang,unrolled	`rm *.i; time make -j9 GCC= CLASSDESC_ARITIES=0xffff`
GCC,arity reduction	`rm *.i; time make -j9 GCC=1 CLASSDESC_ARITIES=0xf`
Clang,arity reduction	`rm *.i; time make -j9 GCC= CLASSDESC_ARITIES=0xf`
Link time	`rm gui-js/node-addons/minskyRESTService.node; \`
GCC link time	`time make -j9 GCC=1`
Clang link time	`time make -j9 GCC=`
Mold link time	`time make -j9 OPT=-fuse_ld=mold`

Commands for timing different optimisation strategies.

Table 3

Conclusion

The RESTService API descriptor provides a scripting-language-independent fat API interface to C++ code. Method arguments and return values can be marshaled using a custom native type 'buffer' object, or using JSON5 encoding with the preexisting Classdesc json descriptor. In practice, JSON5 encoding tends to be sufficiently performant. Both a Javascript and Python bindings were generated automatically for the Minsky systems dynamics simulator, and furthermore, TypeScript binding were generated automatically though a custom descriptor, leading to easier-to-read scripting code, and relatively more type-safe use in Minsky's front end code.

Using the RESTService descriptor comes at additional build cost, compared with the original TCL bindings used for the EcoLab package, which is ameliorated via a number of C++ coding techniques, the use of the Clang toolchain over the GCC one, and the use of modern Linux linkers. ■

References

[Cherny19] Boris Cherny. *Programming TypeScript: making your JavaScript applications scale*. O'Reilly, 2019.

[ECMA13] ECMA. 'The JSON data interchange format. Technical Report ECMA-404', *ECMA International*, 2013. https://ecma-international.org/publications-and-standards/standards/ecma-404/ accessed March 2024.

[EcoLab] EcoLab. https://ecolab.sourceforge.net accessed March 2024.

[Fielding00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[Forrester07] Jay W Forrester. 'System dynamics|a personal view of the first fifty years' *System Dynamics Review: The Journal of the System Dynamics Society*, 23(2-3):345–358, 2007.

[Goldberg22] Josh Goldberg. *Learning TypeScript*. O'Reilly, 2022.

4 compile-optimisations-undone branch, available from https://github.com/highperformancecoder/minsky

[Green13] Brad Green and Shyam Seshadri. *AngularJS*. O'Reilly, 2013.

[Haas17] Andreas Haas, Andreas Rossberg, Derek L Schu, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 'Bringing the web up to speed with WebAssembly'. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

[Ihrig14] Colin J Ihrig. *Pro Node.js for developers*. Apress, 2014.

[Kredpattanakul19] Kitti Kredpattanakul and Yachai Limpiyakorn. 'Transforming JavaScript-based web application to cross-platform desktop with Electron'. In *Information Science and Applications 2018: ICISA 2018*, pages 571–579. Springer, 2019.

[Madina01] Duraid Madina and Russell K. Standish. 'A system for reflection in C++'. In *Proceedings of AUUG2001: Always on and Everywhere*, page 207. Australian Unix Users Group, 2001.

[MXE] MXE (M cross environment). https://mxe.cc/, accessed March 2024.

[Ousterhout94] J. K. Ousterhout. *TCL and the Tk Toolkit*. Addison-Wesley, 1994.

[Standish] Russell K. Standish and Steven L. Keen. Minsky. https://sourceforge.net/projects/minsky/, accessed March 2024.

[Standish01] Russell K. Standish. Ecolab4. *Complexity International*, 8, 2001.

[Standish19] Russell K. Standish. 'C++ reflection for Python binding' in *Overload*, 27(152):11–18, 2019, available at https://accu.org/journals/overload/27/152/standish_2682/.

[Tiobe] Tiobe index. https://www.tiobe.com/tiobe-index/, accessed March 2024.

[Ueyama] Rui Ueyama. Mold: A modern linker. https://github.com/rui314/mold accessed March 2024.

Concurrency: From Theory to Practice

Concurrency is a complicated topic. Lucian Radu Teodorescu provides a simple theory of concurrency which is easy to reason about and apply.

One of the big challenges with concurrency is the misalignment between theory and practice. This includes the goals of concurrency (e.g., improving the performance of the application) and the means we use to achieve that goal (e.g., blocking primitives that slow down the program). The theory of concurrency is simple and elegant. In practice, concurrency is often messy and strays from the good practices of enabling local reasoning and using structured programming.

We present a concurrency model that starts from the theory of concurrency, enables local reasoning, and adheres to the ideas of structured programming. We show that the model can be put into practice and that it yields good results.

Most of the ideas presented here are implemented in a C++ library called concore2full [concore2full]. The library is still a work in progress. The original goal for this model and for this library was its inclusion in the Hylo programming language [Hylo]. For Hylo, we want a concurrency model that allows local reasoning and adheres to the structured programming paradigm. We also wanted a model in which there is no function colouring [Nystrom15], in which concurrency doesn't require a different programming paradigm.

This article is based on a talk I gave at the ACCU 2024 conference [Teodorescu24]. The conference was great! The programme selection was great; there was always something of interest to me. With many passionate C++ engineers and speakers, the exchange of information between participants was excellent; as they say, the best track was the hallway track. I highly encourage all C++ enthusiasts (and not just C++) to participate in future ACCU conferences.

What is concurrency?

Before we actually define concurrency, it's important to draw a distinction between what the program expresses at design-time and its run-time behaviour. There might be subtle differences between the two. For example, even though the program expresses instruction `A` before instruction `B`, at run-time, the two instructions might be executed in reverse order (if there is no dependency between them) [Wikipedia-2]. In this respect, at program design-time we express a range of run-time possibilities, without prescribing a precise run-time behaviour.

Another example, more appropriate to our article: the code may specify that there needs to be two threads that execute some work, but we don't know at run-time if the two work items are executed in parallel or whether the execution hardware somehow sequences them. It may happen that at run-time we have only one core available to execute the two work items, and thus we execute them serially. The original program expresses more possibilities than the actual execution.

More formally, we say that the execution of the program is a *refinement* of the program description written in the code. The execution is more *determinate* than the original program; it is more predictable and more controllable, and adds further decisions compared to the original program. See [Hoare14] for a more formal description of refinement, and how this can be applied to concurrency.

From a run-time perspective, we can define **concurrent execution** as the *partial ordering of work execution* (as opposed to non-concurrent execution, which is a total ordering of work execution). If we denote this ordering relation with $\leq$, then the following rules apply:

$$A \leq A$$
$$A \leq B, B \leq C \Rightarrow A \leq C$$
$$A \leq B, B \leq A \Rightarrow A = B$$

This means that, for two work items $A \neq B$, there are only three ways in which execution can happen:

- $A < B$
- $B < A$
- neither $A < B$, nor $B < A$; we donate this by $A \parallel B$.

I urge the reader to pause for a moment and reflect on the significance of this. There is no other way in which concurrent execution can happen at run-time. From a run-time perspective, concurrency is elementary.

From a design-time perspective, things are slightly more involved, but still simple. At design-time, we want to express constraints that would limit the behaviour at run-time. There are four simple constraints that immediately follow from the run-time possibilities:

- $A < B$
- $B < A$
- neither $A < B$, nor $B < A$; we donate this by $A \parallel B$,
- either $A < B$, or $B < A$; we call this *mutually exclusive* and we donate this by $A \nparallel B$.

Besides these simple constraints, we should also define *conditional concurrency*, $\mathbb{C}(c, A, B)$, which expands to either $A \parallel B$ or $A \nparallel B$, depending on the run-time evaluation of the expression c. And, of course, we need to expand our schema to include more than two work items; this expansion is trivially achievable.

If we want local reasoning or structured programming, then we should strive to make the concurrency constraints local to different functions. In this case, we would use the simple constraints, and conditional concurrency is not as useful.

In the context of programming languages, the goal of concurrency is to allow the programmer to express concurrency constraints on how different work items need to be executed. These constraints just put bounds on the execution; they still allow a multitude of ways in which work items can be executed.

Lucian Radu Teodorescu has a PhD in programming languages and is a Staff Engineer at Garmin. He likes challenges; and understanding the essence of things (if there is one) constitutes the biggest challenge of all. You can contact him at lucteo@lucteo.ro

Before we actually define concurrency, it's important to draw a distinction between what the program expresses at design-time and its run-time behaviour

Expressing concurrency in C++

Let's now try to express these rules in C++. Let's assume that *A* and *B* are local work items (i.e., that needs to be executed in the body of a function). We will encode them by using function calls.

For the first two cases, it's easy, as we already have support from the language:

```
A() ; B()
```

or

```
B() ; A()
```

There is nothing special here; we just sequence the work items in the order we execute them.

Expressing mutual exclusion with local work items is trivial. We choose which one we want to be before the other, and just code it like that. Thus, both `A() ; B()` and `B(); A()` are good forms of mutual exclusion between *A* and *B*.

To express concurrent execution, we introduce a new abstraction that can be implemented as a function taking a lambda as an argument. The code in Listing 1 shows an example.

In this example, we express the following concurrency constraints: $A < B$, $A < C$, $B \parallel C$, $B < D$, and $C < D$.

The code behaves as if we spawn a new thread to execute `C`, and then join that thread when awaiting. Of course, we are not doing this, but having that as a mental model might help.

The `spawn` function returns a future object that is neither movable nor copyable. We will discuss this restriction and alternatives later; for now, it's important to note that it implies that we only represent local concurrency constraints.

This `spawn`/`await` model is similar to other async/await models [Wikipedia-1], but the implementation details differ.

This forms the basis of the concurrency we need.

More examples

Let's start with an example showing that this model can be used to encode more complex graphs. Please refer to Listing 2 for an implementation of the graph expressed in Figure 1 (overleaf).

To build a concurrent sort with this `spawn` primitive, we can write something similar to Listing 3 (overleaf). In this example, we partition the array that needs to be sorted into two parts so that all the elements

```
A();
auto future = spawn([] { C(); });
B();
future.await();
D();
```

Listing 1

```
int run_work() {
  auto sum = 0;
  // T1 is run before anything else.
  sum += run_task(1);

  // Flow that executes T2, T6, T13, T17.
  auto f1 = concore2full::spawn([] {
    auto local_sum = 0;
    local_sum += run_task(2);

    // T6 and T7 are run concurrently.
    auto f = concore2full::spawn([] {
      return run_task(7); });
    local_sum += run_task(6);
    local_sum += f.await();

    local_sum += run_task(13);
    local_sum += run_task(17);
    return local_sum;
  });

  // Flow that executes T3, T8.
  auto f2 = concore2full::spawn([] {
    return run_task(3) + run_task(8);
  });

  // Flow that executes T4, T9, T10, T14.
  auto f3 = concore2full::spawn([] {
    auto local_sum = 0;
    local_sum += run_task(4);

    // T9 and T10 are run concurrently.
    auto f = concore2full::spawn([] {
      return run_task(10); });
    local_sum += run_task(9);
    local_sum += f.await();

    local_sum += run_task(14);
    return local_sum;
  });

  // Flow that executes T5, T11, T12, T15, T16.
  auto f4 = concore2full::spawn([] {
    auto local_sum = 0;
    local_sum += run_task(5);

    // T11+T15 and T12+T16 are run concurrently.
    auto f = concore2full::spawn([] {
      return run_task(12) + run_task(16); });
    local_sum += run_task(11) + run_task(15);
    local_sum += f.await();
    return local_sum;
  });

  // Everything must finish before executing T18.
  sum += f1.await() + f2.await() + f3.await()
    + f4.await();
  sum += run_task(18);
  return sum;
}
```

Listing 2

The purpose of structured programming is to enable local reasoning. As Dijkstra puts it, the human mind is limited.

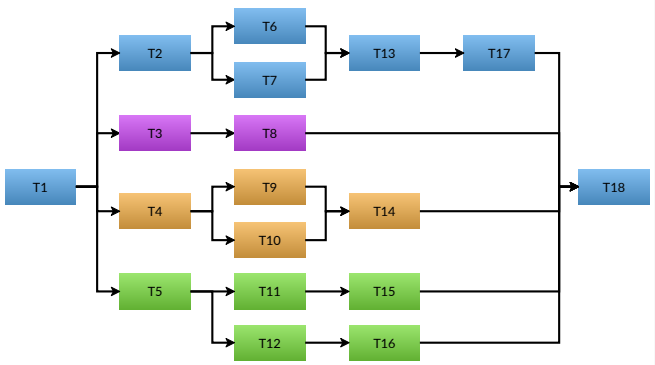


Figure 1

on the left side are smaller than the elements on the right side. Then, we can sort the left side and the right side concurrently. This process is recursively applied until the array that needs to be sorted is small enough that a serial sort is more efficient.

Listing 4 shows how one can use **bulk_spawn** to run the computation for a Mandelbrot set concurrently. This primitive spawns multiple work items concurrently, where the number of work items is known at run-time.

This section shows that, using this model, one can build concurrency into slightly more complex problems in an intuitive manner.

Structured concurrency

Let us start by reminding the reader about structured programming. Following the *Structured Programming* book by Dahl, Dijkstra, and

```
template <std::random_access_iterator It>
void concurrent_sort(It first, It last) {
  auto size = std::distance(first, last);
  if (size_t(size) < size_threshold) {
    // Use serial sort under a certain threshold.
    std::sort(first, last);
  } else {
    // Partition the data, such as elements
    // [0, mid) < [mid] <= [mid+1, n).
    auto p = sort_partition(first, last);
    auto mid1 = p.first;
    auto mid2 = p.second;

    // Spawn work to sort the right-hand side.
    auto handle = spawn([=] { concurrent_
sort(mid2, last); });
    // Execute the sorting on the left side,
    // on the current thread.
    concurrent_sort(first, mid1);
    // We are done when both sides are done.
    handle.await();
  }
}
```

Listing 3

```
void concurrent_mandelbrot(int* vals, int max_x,
     int max_y, int depth) {
  concore2full::bulk_spawn(max_y, [=](int y) {
    for (int x = 0; x < max_x; x++) {
      vals[y * max_x + x] =
        mandelbrot_core(transform(x, y), depth);
    }
  }).await();
}
```

Listing 4

Hoare [Dahl72], we extract two important characteristics of structured programming (there are more, but we are just going to focus on these two).

The first one is the idea that every operation needs to have a single entry and a single exit point. All the basic operations have this shape; the **if**, **while**, and **for** blocks all share this as well. Function calls also have this shape. This makes all the operations in the program have the same shape.

A second significant idea in structured programming is that of recursive decomposition. Complex functionalities can be decomposed into smaller functionalities, which may be further divided into even smaller functionalities. The entire program can be divided into small operations that will ultimately reach the basic operations of the language (variable declaration, assignment, arithmetic operations, etc.).

It's not enough to just be able to decompose programs into smaller operations; these operations also need to be (to a large degree) independent. That is, one can look at one function and reason about it independently without needing to know how other (unrelated) functions in the program are implemented. Of course, there are interactions between the functions, but these interactions should be reduced as much as possible.

The purpose of structured programming is to enable local reasoning. As Dijkstra puts it, the human mind is limited. Having a linear flow in the program, in which every operation has the same shape, and being able to recursively decompose the program into smaller, mostly independent chunks, helps our mind reason about the code.

Let's now turn our attention to the properties of the future. A future can be of four types, based on the movability and copyability traits:

- not movable and not copyable (what we've seen above)
- movable but not copyable
- movable and copyable
- not movable but copyable

The last option doesn't make much sense, and we can drop it. Thus, we have only three options to analyse. The most restrictive one is for the future to be not movable and not copyable.

Not being able to move the future implies that the **await** call (we always assume that there will be an **await** call) needs to be in the same scope as the **spawn**. This means that the pair **spawn**/**await** can behave like a

sometimes, adding restrictions in a language may provide additional guarantees, improving it

block (there are some exceptions, but we can safely ignore those). Such a block has one entry and one exit point. This means that using `spawn`/`await` blocks follows the idea of structured programming.

With this type of future, we can say that we obtain *structured concurrency*. It makes it easy to reason about concurrency, localising the concurrency concerns, and allowing for their encapsulation.

Now, because the `await` is in the same scope as `spawn`, it means that the stack used at the `spawn` point is kept alive until `await`. However, because the spawned work needs to be completed before `await`, it follows that the `spawn` work can safely access the stack available at the `spawn` point. In the example for Listing 1, both `B()` and `C()` can access the stack that was available at the call of `A()`.

Furthermore, we can store directly in the future object all the data needed to synchronise between the two work items that need to be executed concurrently. This helps performance, as there is no need for a heap allocation.

While this future type is more restrictive than the others, it clearly provides advantages.

Let us now look at the future that is movable (but still not copyable). Listing 5 provides an example of using such a future.

In this example, we use a different abstraction, called `escaping_spawn`, as we need to produce a different type of future. We see that the spawn point and the await point happen at different points, and for that reason, the concurrency model is not fully structured. We call this model *weakly structured concurrency*.

While the guarantees for this style are weaker, one can still reason about the concurrency being handled between the two functions. The declaration of the `spawn_work()` function indicates that we are escaping a future. Reasoning about such an escaped function is similar to that needed for returned functors.

If we look at the stack access, we notice that, in this case, `C()` cannot access the stack at point `A()` (for example, access the `data` variable). The `spawn_work` function might exit before `C()` gets a chance to execute. The spawned work can only access stack data that is kept alive by the `await` call. However, because we require global reasoning to understand where the `await` point is, in most cases the spawned work cannot access the stack from which it was spawned.

Similarly, we cannot put the data required for the synchronisation on the stack, as the stack may shortly disappear. Thus, we need to have a heap allocation for `escaping_spawn`.

Thus, weakly structured concurrency is less restrictive, but is not as efficient. This is another example that shows that, sometimes, adding restrictions in a language may provide additional guarantees, improving it. In this case, not being able to move a future allows us to use the stack at the `spawn` point, and allows improved performance.

```
auto spawn_work() {
  A();
  std::vector<int> data;

  return escaping_spawn([] {
    C();
  });
}

void weakly_structured_concurrency() {
  auto future = spawn_work();
  B();
  future.await();
  D();
}
```

Listing 5

The `bulk_spawn` abstraction that we've seen in Listing 4, can work both in strict structured concurrency but also in weak concurrency cases. For `bulk_spawn` we allocate the frame object on the heap, as the size of the frame depends on run-time parameters.

At the point of writing this article, we haven't yet implemented copyable futures; their implementation is more involved, as one spawned work item can potentially continue multiple flows that await the result of the original work item.

Implementing spawn

Now that we have described the expectations around using this model, let us describe how this can be implemented. We are going to focus on the implementation of `spawn`, but the implementation of `escaping_spawn` and `bulk_spawn` is similar. We use the code from Listing 1 as our running example.

First, we are using a task pool to handle the spawned work. This is a pretty common technique.

Now, if the work `B()` takes more time than the work `C()`, then the execution follows the expected pattern; please see Figure 2. There would be no blocking wait. The original thread would execute `A()`, then `B()`, then `D()`, while a worker thread would execute `C()`. The work corresponding to `C()` would finish before the `await` point, thus all the concurrent constraints are satisfied. All good.

The problem appears if executing `C()` takes longer than executing `B()`. The original thread arrives at the `await` point before the work corresponding to `C()` is complete. This means that the original thread cannot continue executing `D()`; see Figure 3 (overleaf).

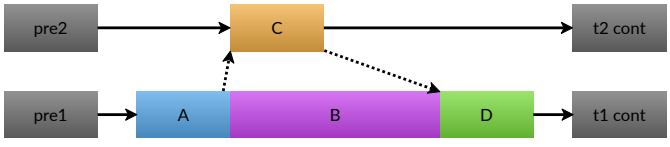


Figure 2

this may be a bit counterintuitive ... but an adequate option is to go to the thread pool and continue the work there

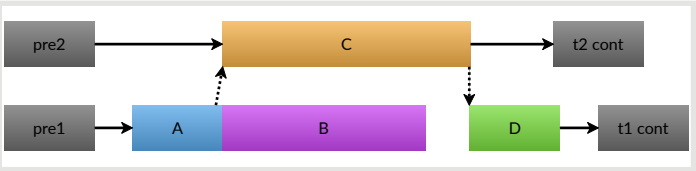


Figure 3

A common strategy is to block the thread until `C()` is done. However, this has negative performance implications. We cannot go this route (at least, not for the general case).

Another strategy is for the original thread to steal some other work from the system and execute it while waiting for `C()` to complete. However, while this strategy keeps throughput of the application high, it has negative implications in terms of latency. For example, if `C()` is 500µs longer than `B()`, we might start another work item that takes 1s to execute. So, we introduce a latency of 1s into this thread. This is not a good strategy either.

A better strategy is to let the worker thread continue executing work `D()`, instead of executing it on the original thread. A similar strategy is employed by the `when_all()` algorithm from senders/receivers [P2300R9]. If we do this, a new challenge arises: what can the original thread do in the meantime?

Well, this may be a bit counterintuitive to the reader, but an adequate option is to go to the thread pool and continue the work there. That is, we essentially *switch the threads*. The original thread will continue to execute whatever the worker thread has, while the worker thread will continue to execute everything on the main flow after the `await` point. We also call this behaviour *thread hopping*.

A simplified view of a thread is that it consists of a set of registers (most importantly an instruction pointer, `IP`, and a stack pointer, `SP`) plus a stack memory region associated with it. During the lifetime of the thread, the stack pointer register keeps changing within the stack region. Thread hopping essentially swaps important registers between threads, allowing a thread to point to the stack region created by another thread.

A good technique to implement thread hopping is to use stackful coroutines [Moura09]. Indeed, for my implementation, I've used the boost::context library [context]. A stackful coroutine is created to execute the spawned work; the worker thread doesn't do much work on its stack, as it immediately jumps to the coroutine stack.

Figure 4 shows how thread hopping works. On the left side of the figure, we depict the stack regions; we have three of them: two for the threads and one for the coroutine that was created. After executing `B()` thread 1 jumps and continues execution on the stack created for thread 2. After executing `C()`, thread 2 continues to execute the continuation on the stack created for thread 1. At the end of the work, the two threads are essentially swapped.

There is another case that needs to be discussed. It might happen that, during the entire execution of the work, there isn't a worker thread available to execute the `C()` work item. If, when reaching the await point, the task corresponding to `C()` has not been taken by a worker thread for execution, we execute it inline, on the original thread.

It is important to note that this execution is consistent with the concurrency constraints. That is, $B < C$ can be a valid execution of $B \parallel C$, and we still have $B < D$ and $C < D$.

In this case, we don't create a new coroutine for executing `C()`. We are doing the most reasonable thing to do in the case where we don't have enough hardware resources.

Allowing thread switching, we ensure that in any scenario, the system will not block, and we always execute work items as soon as possible, within the bounds of the given concurrency constraints.

A direct consequence of thread hopping is that a function may enter on one thread and exit on a different thread. Please note that this still respects the principles of structured programming.

Similar to `spawn`, we can implement `escaping_spawn` (to create weakly structured futures) and `bulk_spawn` (to start executing multiple work items at the same time).

Early measurements

The ideas presented here are still a work-in-progress. But, even in this case, a few measurements would help to understand whether the direction in which this is moving is promising or not.

Skynet

Let's start with the Skynet micro-benchmark [Skynet]. We create a task, which creates 10 more tasks, each creating 10 more tasks, etc. At the final level, we would be creating 10 million tasks (original benchmark was going up to 1 million, but we increased it to 10 million). The tasks at the final level are returning their ordinal number, while the other tasks are just summing up the values returned from the children. In total, there are five quadrillion five trillion tasks created.

The purpose of this micro-benchmark is to check if the task model scales for a massive number of tasks. We check whether the program deadlocks

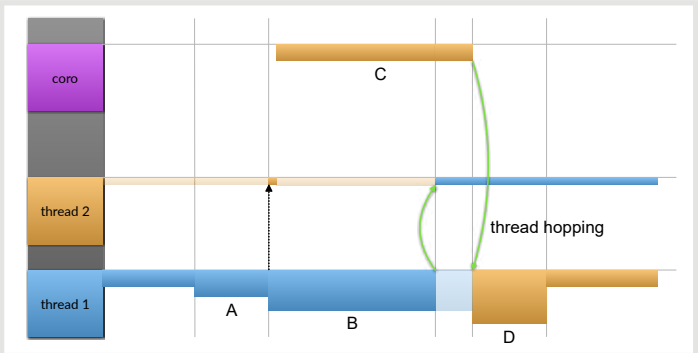


Figure 4

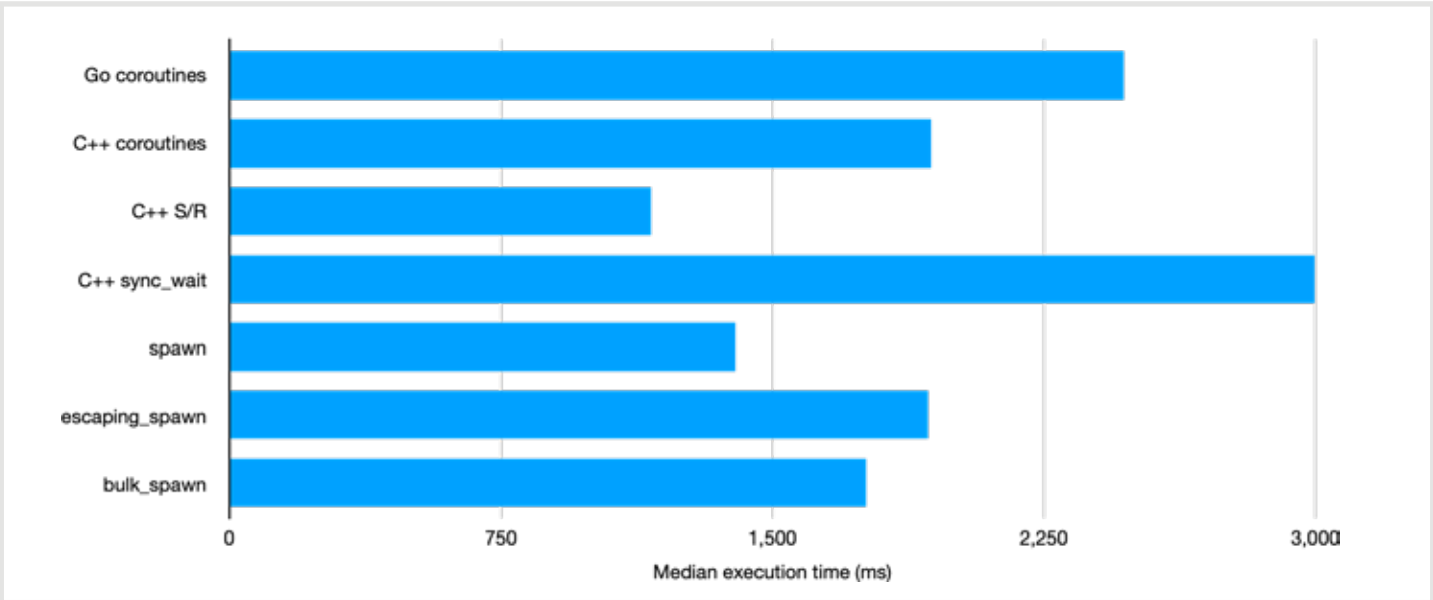


Figure 5

or we run out of stack or other resources. In terms of performance, this will measure the overhead of creating and joining tasks, and it's not very representative of real-world workloads (where we would do more useful work, and create fewer tasks).

The results of running this micro-benchmark are presented in Figure 5. First, we present the 'reference' measurements, that is, the implementation of the micro-benchmark in Go, which uses concurrency with goroutines.

Next, we present the measurements corresponding to three C++ implementations: one that just uses coroutines, one that uses senders/receivers and coroutines, and one that uses senders/receivers and the `sync_wait` algorithm. The coroutine version is single-threaded. The version that uses senders/receivers with a coroutine task uses a thread pool and fully utilises all the cores on the machine; it achieves the best performance from all our measurements. It is important to note that the version with `sync_wait` deadlocks as soon as it creates more tasks than there are threads in the thread pool.

Then we show the measurements made for our concurrency framework in three different scenarios: using structured concurrency (`spawn`), using weakly structured concurrency (`escaping_spawn`), and spawning 10 items at once (`bulk_spawn`). All three measurements corresponding to our implementation are faster than the Go implementation. The `spawn` execution is 20% slower than the senders/receivers execution. As expected, the structured concurrency program is faster than the other two versions. In the weakly structured concurrency, we are doing a heap allocation for each work item we spawn, while in the case of bulk spawning items, we are making a heap allocation for spawning the work for 10 children.

The results from running this micro-benchmark are overall positive. Firstly, we did not deadlock (unlike the `sync_wait` version), and we did not consume a large amount of stack. In terms of performance, we are 20% slower than the fastest version measured. This result is not that bad, considering the overhead is relatively small, and that the number of `spawn`/`await` points in a typical application is relatively small.

Speedup

Another micro-benchmark worth doing is checking the scale-up of a somewhat more realistic problem (computing the Mandelbrot values for a 4K image, one task per row). This time, we try to specify the number of threads that the library can use and measure the total runtime.

Figure 6 shows the speedup results for running this on my Apple MacBook Pro, M2 Pro, 16 GB, with 12 cores (8 performance, 4 efficiency); the total execution time for a test is between 868 ms and 9319 ms. The speedup looks really good; for up to 8 threads, it is really close to the ideal numbers. Going between 8 and 12 threads, the speedup is not that great, as we are utilising the efficiency cores for performance tasks. Going past 12, the number of cores on my machine, doesn't help; there are simply no extra resources to speed up the computation.

For people familiar with speedup calculations, the numbers are excellent.

Analysis

Expressing concurrency

The concurrency model presented here is very good at expressing concurrency. With just a few primitives, we are able to represent many concurrency problems. While we don't have conditional concurrency implemented yet, many problems do not need it directly (expressing non-local concurrency constraints is not best practice).

The model provides a forward progress guarantee. Once a work item starts executing, it will complete and, eventually, all work items are started. Thus, all the spawned work items are executed. This means that the program will always make progress and never be stuck.

Safety

The model assumes that the user ensures proper constraints between work items. That is, there are no two concurrent tasks that access the same memory location such that at least one of them is writing to it. This forms a basic precondition of writing a concurrent relationship between

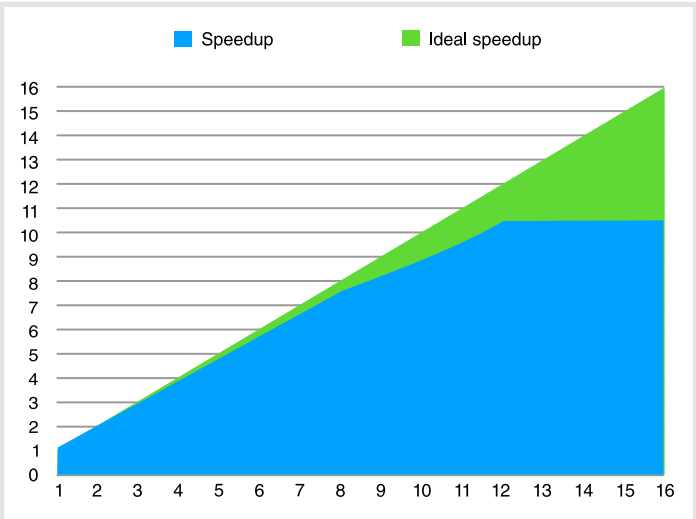


Figure 6

work items. If this precondition is met, the model doesn't have any race conditions.

The model allows directly expressing concurrency constraints, so there is no need for extra synchronisation; this eliminates an entire category of safety issues. In particular, there are no deadlocks.

To conclude, if the constraints are correctly set, the model ensures concurrency safety.

Performance

The concurrency model that we presented doesn't require blocking waits at the user level. This is a huge performance advantage compared to many other models found in practice. The only performance costs that the model incurs are localised in the calls to `spawn` and `await`. As we've seen, early measurements indicate that this makes the model about 20% slower compared to the implementation on top of senders/receivers. This is a good number in itself.

In real-world applications, the time spent in `spawn`/`await` is tiny compared to the useful work. This means that this 20% will not affect the overall performance of these applications. This can be seen from the speedup measurements we've presented.

To conclude, the performance appears to be good, but not necessarily the best.

Stack usage

In general, there is a concern that models based on stackful coroutines are bad because of their stack usage. That is, one cannot spawn too many coroutines as it would require many stack allocations, each coroutine needing a full stack. The results from the Skynet micro-benchmark proved that our model doesn't have this problem.

An important factor that influences stack consumption is the way we create a coroutine stack for spawning new work: we only do that after creating a task in our thread pool. This means that the number of coroutine stacks used for spawning work is limited by the number of threads in the thread pool.

At this point, the implementation of the model also creates a coroutine inside `await`, to be able to swap continuations. The stack requirements for this one are small, and, with a bit of extra work, can be avoided (e.g., by reusing the caller's stack).

Furthermore, the worker threads don't need a lot of stack space. They would only jump to executing on coroutine stacks.

All these, with some extra tuning, can make the stack usage of this concurrency model to be small. It can be smaller than the stack required for an application that uses the threads-and-locks model and creates more threads than necessary.

Interoperability

Here, the model doesn't fare that well. The main reason is that, with thread hopping, a function execution can start on one thread and end on a different one. This may break the assumptions of the surrounding code.

If external code calls into our code that uses thread hopping, it may need to restore the original thread each time it calls a function into our code. This potentially involves a blocking wait (the original thread may be doing something else, and we need to wait for it to finish). This is not great.

Additionally, the code cannot use thread-local storage in the way people are accustomed to.

These interoperability challenges are present in all asynchronous models (senders/receivers, other async/await models). In each of these models, there needs to be a *synchronous-wait* operation so that synchronous code can call asynchronous code.

More to explore

The current implementation of the model is still young. More features need to be added to it. We need copyable futures, so that multiple parties can await the completion of a work item. Then, we have to add cancellation to the entire model.

To be able to easily encode non-local concurrency constraints, we also need more support for what we call conditional concurrency: that is, sometimes work items are executed concurrently, sometimes they are mutually exclusive, depending on some other conditions.

Another important aspect that we should consider is the integration with I/O, timers, running work on GPUs, and custom execution contexts.

All these are in the plan for the future of the model.

Conclusions

We presented a model for concurrency that starts from the theory and tries to put it into practice in a simple, easy-to-reason-about, and efficient way.

The theory of concurrency is surprisingly simple: just partial ordering on the execution of work items. Instead of modelling this concurrency with mutexes, semaphores, and other synchronisation primitives, we can directly try to express the possible constraints in the code. We introduce the `spawn`/`await` model, which can model the most common concurrency constraints.

Using `spawn`/`await` will keep us in the realm of structured programming. The `spawn`/`await` block can be considered an operation with one entry and one exit point, so it has a similar shape to the rest of the operations. We can still use recursive decomposition, and we can encapsulate concurrency constraints inside functions. For example, we might add concurrent execution to a function that previously did not have any, without the callers being affected by it.

All this makes the model give us *reasonable concurrency*. That is, something that we can easily reason about, and something that is not out-of-ordinary, something that is not unexpected, outrageous, or excessively costly. One doesn't need to use dark arts to master concurrency. ■

References

[concore2full] Lucian Radu Teodorescu, concore2full library, https://github.com/hylo-lang/concore2full, accessed April 2024.

[context] Oliver Kowalke, boost::context library, https://www.boost.org/doc/libs/1_85_0/libs/context/.

[Dahl72] O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Structured Programming, Academic Press Ltd., 1972.

[Hoare14] Tony Hoare, Stephan van Staden, The laws of programming unify process calculi, Science of Computer Programming 85, 2014.

[Hylo] The Hylo Programming Language, https://www.hylo-lang.org/.

[Moura09] Ana Lúcia De Moura, Roberto Ierusalimschy, Revisiting coroutines, ACM Transactions on Programming Languages and Systems (TOPLAS), 2009, https://dl.acm.org/doi/pdf/10.1145/1462166.1462167.

[Nystrom15] Bob Nystrom, What Color is Your Function?, https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/.

[P2300R9] Michał Dominiak, Georgy Evtushenko, Lewis Baker, Lucian Radu Teodorescu, Lee Howes, Kirk Shoop, Michael Garland, Eric Niebler, Bryce Adelstein Lelbach, std::execution, 2024, http://wg21.link/P2300R9.

[Skynet] Alexander Temerev, Skynet 1M concurrency microbenchmark, https://github.com/atemerev/skynet.

[Teodorescu24] Lucian Radu Teodorescu, Concurrency Hylomorphism, ACCU Conference, April 2024.

[Wikipedia-1] Wikipedia, Async/await, https://en.wikipedia.org/wiki/Async/await, 2024.

[Wikipedia-2] Wikipedia, Out-of-order execution, https://en.wikipedia.org/wiki/Async/await, 2024.

Afterwood

What do you do when a software system goes wrong? Chris Oldwood discusses designing for supportability.

One of the books which had a profound impact on me early on in my programming career was *Writing Solid Code* by Steve Maguire. In Chapter 4 (Step Through Your Code) he introduces the practice of stepping through any new code you write, in the debugger, to see the code in action so that you can check the data flow, such as loop variables, to help avoid the perennial programmer nemesis – the off-by-one error. One of the side-effects of this practice is that it forces you to think about how to make it easy to get to that point in code in a debugger. If the code is many layers deep in the application then you might be tempted to create an explicit test harness that allows you to invoke the code more easily, along with the added benefit of giving you more control over the inputs. In turn, that thought process can have an effect on the design of the code as you make it more 'debuggable' in the first place.

Although he didn't use the term in his book back in 1993, this notion of shaping the code to make it easier to test is now known as 'Design for Testability' and has a history in the hardware world that dates back to at least the early half of the 20th century. Black Box Testing, while useful, can only get you so far in the hardware world and, as complexity grew, they started to add additional features to help ensure the product was working correctly internally. In the software world, White Box Testing has materialized under the guise of Unit Testing, with Mocking in particular being a realization of how the desire to make code more testable can affect the design of components.

I continued to use the practice of stepping through my code in the debugger as my primary means of testing for the better part of a decade. What brought it to an end was being introduced to the newfangled practice of automated unit testing, along with the realization that the computer was so much more reliable at repetitive tasks like regression testing than a human. (More details on my eventual fall from grace and subsequent epiphany can be found in my *ACCU 2017* conference talk 'A Test of Strength'.)

Being able to easily and reliably test my code was definitely a big win, but it also had another side-effect that I hadn't anticipated until I started working on more complex systems – supportability. I got my first glimpse of this when I discovered that a test harness I wrote to make development of a back-end scheduling engine easier was being bundled with the application, for when bugs in the front-end made it impossible to fix-up the schedule. My test harness, while very raw from a GUI point of view (the sea of database IDs felt a bit like staring at The Matrix) allowed direct access to the back-end code so the schedule could be fixed-up by manually driving it using the real business logic. This was considered far safer than hacking about directly in the database as it minimised the chances of corrupting the state. (Debugging through the front-end, the default practice up to that point, cost you 8 minutes just waiting for it to load before you could invoke any back-end logic.)

That experience taught me that there was more value in test harnesses than simply being able to make a developer's life easier. As I started to interact with more support engineers, I began to see how hard their life was supporting applications and systems because they were so far removed from the developers building the system. In the intervening three decades since that episode took place, the industry as a whole has started to empathise more with those outside the development team and have recognised that other areas such as InfoSec and Ops are also valid stakeholders in the system and their needs have to be listened to and addressed alongside those from the end users. This culminated in the creation of the DevOps movement and a 'you build it, you run it' mentality, although it has since grown so much wider as the realisation dawned that only a holistic approach to building and running systems works in practice over the long term.

While perhaps somewhat easier now, in the past I have had to fight for my belief in what appears to be only informally known today as Design for Supportability. One project manager back in the late 2000s even suggested that any time spent creating custom tooling should be my own time, as it was not part of The Deliverables. When the 'Business as usual' (BAU) and Analysis teams discovered a testing tool I wrote to help me create custom test data sets, they openly thanked me, and then I felt my approach and time was vindicated.

When I moved to another organization in the same industry to work on a similar system, I put supportability front and centre, letting it drive the design and architecture to such an extent that for production it ran as a bunch of distributed services, but the same code could also be hosted in a single command line tool using local instead of remote procedure calls. I called it a 'gig-in-a-box' because the entire distributed calculation engine was essentially running as a monolithic process which allowed us to easily debug, test, profile, and hence support the majority of the system's codebase. We even had a formal database schema called 'support' so our ad hoc SQL snippets could become first class citizens.

For sure, wasting time on speculative requirements and gold-plating are a concern, but there are ways to make that visible and, more importantly, discover what is driving that behaviour. Any team probably already has a bunch of half-baked, stale, duplicated support scripts and tools, so formalising them by adding them to the codebase can only be a good thing as then they will get the care and attention they deserve. Production incidents are stressful enough as it is, having a good toolkit can reduce the chances of that turning into a full-on disaster. ■

Chris Oldwood is a freelance programmer who started out as a bedroom coder in the 80s writing assembler on 8-bit micros. These days it's enterprise grade technology from ~~plush corporate offices~~ the comfort of his breakfast bar. He also commentates on the Godmanchester duck race and is easily distracted by emails and DMs to gort@cix.co.uk and @chrisoldwood

Join ACCU

Run by programmers for programmers,
join ACCU to improve your coding skills

- A worldwide non-profit organisation
- Journals published alternate months:
  - *CVu* in January, March, May, July, September and November
  - *Overload* in February, April, June, August, October and December
- Annual conference
- Local groups run by members

Join now!
Visit the website

professionalism in programming

www.accu.org

"The magazines"

The ACCU's *C Vu* and *Overload* magazines are published every two months, and contain relevant, high quality articles written by programmers for programmers.

"The conferences"

Our respected annual developers' conference is an excellent way to learn from the industry experts, and a great opportunity to meet other programmers who care about writing good code.

"The community"

The ACCU is a unique organisation, run by members for members. There are *many* ways to get involved. Active forums flow with programmer discussion. Mentored developers projects provide a place for you to learn new skills from other programmers.

"The online forums"

Our online forums provide an excellent place for discussion, to ask questions, and to meet like minded programmers. There are job posting forums, and special interest groups.

Members also have online access to the back issue library of ACCU magazines, through the ACCU web site.

ACCU | JOIN: IN

PROFESSIONALISM IN PROGRAMMING
WWW.ACCU.ORG

Invest in your skills. Improve your code. Share your knowledge.

Join a community of people who care about code. Join the ACCU.

Use our online registration form at **www.accu.org**.

Design: Pete Goodliffe