C++11 threads, affinity and hyperthreading (http://eli.thegreenplace.net/2016/c11-threads-affinity-and-hyperthreading/)

January 17, 2016 at 16:38

Background and introduction

For decades, the C and C++ standards treated multi-threading and concurrency as something existing outside the standard sphere - in that "target-dependent" world of shades which the "abstract machine" targeted by the standards doesn't cover. The immediate, cold-blooded replies of "C++ doesn't know what a thread is" in mountains of mailing list and newsgroup questions dealing with parallelism will forever serve as a reminder of this past.

But all of that came to an end with C++11. The C++ standards commitee realized the language won't be able to stay relevant for much longer unless it aligns itself with the times and finally recognizes the existence of threads, synchronization mechanisms, atomic operations and memory models - right there in the standard, forcing C++ compiler and library vendors to implement these for all supported platforms. This is, IMHO, one of the biggest positive changes in the avalanche of improvements delivered by the C++11 edition of the language.

This post is not a tutorial on C++11 threads, but it uses them as the main threading mechanism to demonstrate its points. It starts with a basic example but then quickly veers off into the specialized area of thread affinities, hardware topologies and performance implications of hyperthreading. It does as much as feasible in portable C++, clearly marking the deviations into platform-specific calls for the really specialized stuff.

Logical CPUs, cores and threads

Most modern machines are multi-CPU. Whether these CPUs are divided into sockets and hardware cores depends on the machine, of course, but the OS sees a number of "logical" CPUs that can execute tasks concurrently.

The easiest way to get this information on Linux is to `cat /proc/cpuinfo`, which lists the system's CPUs in order, providing some infromation about each (such as current frequency,

cache size, etc). On my (8-CPU) machine:

```
$ cat /proc/cpuinfo
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model               : 60
model name  : Intel(R) Core(TM) i7-4771 CPU @ 3.50GHz
[...]
stepping    : 3
microcode   : 0x7
cpu MHz             : 3501.000
cache size  : 8192 KB
physical id : 0
siblings    : 8
core id             : 0
cpu cores   : 4
apicid              : 0
[...]

processor   : 1
vendor_id   : GenuineIntel
cpu family  : 6
[...]

[...]
processor   : 7
vendor_id   : GenuineIntel
cpu family  : 6
```

A summary output can be obtained from `lscpu`:

```
$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                8
On-line CPU(s) list:   0-7
Thread(s) per core:    2
Core(s) per socket:    4
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 60
Stepping:              3
CPU MHz:               3501.000
BogoMIPS:              6984.09
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K
NUMA node0 CPU(s):     0-7
```

Here it's also very easy to see that the machine has 4 cores, each having two HW threads (see hyperthreading (https://en.wikipedia.org/wiki/Hyper-threading)). And yet the OS sees them as 8 "CPUs" numbered 0-7.

Launching a thread per CPU

The C++11 threading library gracefully made available a utility function that we can use to find out how many CPUs the machine has, so that we could plan our parallelism strategy. The function is called `hardware_concurrency`, and here is a complete example that uses it to launch an appropriate number of threads. The following is just a code snippet; full code samples for this post, along with a Makefile for Linux can be found in this repository (https://github.com/eliben/code-for-blog/tree/master/2016/threads-affinity).

```
int main(int argc, const char** argv) {
  unsigned num_cpus = std::thread::hardware_concurrency();
  std::cout << "Launching " << num_cpus << " threads\n";

  // A mutex ensures orderly access to std::cout from multiple threads.
  std::mutex iomutex;
  std::vector<std::thread> threads(num_cpus);
  for (unsigned i = 0; i < num_cpus; ++i) {
    threads[i] = std::thread([&iomutex, i] {
      {
        // Use a lexical scope and lock_guard to safely lock the mutex only for
        // the duration of std::cout usage.
        std::lock_guard<std::mutex> iolock(iomutex);
        std::cout << "Thread #" << i << " is running\n";
      }

      // Simulate important work done by the tread by sleeping for a bit...
      std::this_thread::sleep_for(std::chrono::milliseconds(200));

    });
  }

  for (auto& t : threads) {
    t.join();
  }
  return 0;
}
```

A `std::thread` is a thin wrapper around a platform-specific thread object; this is something we'll use to our advantage shortly. So when we launch a `std::thread`, and actual OS thread is launched. This is fairly low-level thread control, but in this article I won't detour into higher-level constructs like *task-based parallelism*, leaving this to some future post.

Thread affinity

So we know how to query the system for the number of CPUs it has, and how to launch any number of threads. Now let's do something a bit more advanced.

All modern OSes support setting CPU *affinity* per thread. Affinity means that instead of being free to run the thread on any CPU it feels like, the OS scheduler is asked to only schedule a given thread to a single CPU or a pre-defined set of CPUs. By default, the affinity covers all logical CPUs in the system, so the OS can pick any of them for any thread, based on its scheduling considerations. In addition, the OS will sometimes migrate threads between CPUs if it makes sense to the scheduler (though it should try to miminize migrations because of

the loss of warm caches on the core from which the thread was migrated). Let's observe this in action with another code sample:

```
int main(int argc, const char** argv) {
  constexpr unsigned num_threads = 4;
  // A mutex ensures orderly access to std::cout from multiple threads.
  std::mutex iomutex;
  std::vector<std::thread> threads(num_threads);
  for (unsigned i = 0; i < num_threads; ++i) {
    threads[i] = std::thread([&iomutex, i] {
      while (1) {
        {
          // Use a lexical scope and lock_guard to safely lock the mutex only
          // for the duration of std::cout usage.
          std::lock_guard<std::mutex> iolock(iomutex);
          std::cout << "Thread #" << i << ": on CPU " << sched_getcpu() << "\n";
        }

        // Simulate important work done by the tread by sleeping for a bit...
        std::this_thread::sleep_for(std::chrono::milliseconds(900));
      }
    });
  }

  for (auto& t : threads) {
    t.join();
  }
  return 0;
}
```

This sample launches four threads that loop infinitely, sleeping and reporting which CPU they run on. The reporting is done via the `sched_getcpu` function (glibc specific - other platforms will have other APIs with similar functionality). Here's a sample run:

```
$ ./launch-threads-report-cpu
Thread #0: on CPU 5
Thread #1: on CPU 5
Thread #2: on CPU 2
Thread #3: on CPU 5
Thread #0: on CPU 2
Thread #1: on CPU 5
Thread #2: on CPU 3
Thread #3: on CPU 5
Thread #0: on CPU 3
Thread #2: on CPU 7
Thread #1: on CPU 5
Thread #3: on CPU 0
Thread #0: on CPU 3
Thread #2: on CPU 7
Thread #1: on CPU 5
Thread #3: on CPU 0
Thread #0: on CPU 3
Thread #2: on CPU 7
Thread #1: on CPU 5
Thread #3: on CPU 0
^C
```

Some observations: the threads are sometimes scheduled onto the same CPU, and sometimes onto different CPUs. Also, there's quite a bit of migration going on. Eventually, the scheduler managed to place each thread onto a different CPU, and keep it there. Different constraints (such as system load) could result in a different scheduling, of course.

Now let's rerun the same sample, but this time using `taskset` to restrict the affinity of the process to only two CPUs - 5 and 6:

```
$ taskset -c 5,6 ./launch-threads-report-cpu
Thread #0: on CPU 5
Thread #2: on CPU 6
Thread #1: on CPU 5
Thread #3: on CPU 6
Thread #0: on CPU 5
Thread #2: on CPU 6
Thread #1: on CPU 5
Thread #3: on CPU 6
Thread #0: on CPU 5
Thread #1: on CPU 5
Thread #2: on CPU 6
Thread #3: on CPU 6
Thread #0: on CPU 5
Thread #1: on CPU 6
Thread #2: on CPU 6
Thread #3: on CPU 6
^C
```

As expected, though there's some migration happening here, all threads remain faithfully locked to CPUs 5 and 6, as instructed.

Detour - thread IDs and native handles

Even though the C++11 standard added a thread library, it can't standardize *everything*. OSes differ in how they implement and manage threads, and exposing every possible thread implementation detail in the C++ standard can be overly restrictive. Instead, in addition to defining many threading concepts in a standard way, the thread library also lets us interact with platform-specific threading APIs by exposing *native handles*. These handles can then be passed into low-level platform-specific APIs (such as POSIX threads on Linux or Windows API on Windows) to exert finer grained control over the program.

Here's an example program that launches a single thread, and then queries its thread ID along with the native handle:

```
int main(int argc, const char** argv) {
  std::mutex iomutex;
  std::thread t = std::thread([&iomutex] {
    {
      std::lock_guard<std::mutex> iolock(iomutex);
      std::cout << "Thread: my id = " << std::this_thread::get_id() << "\n"
                << "        my pthread id = " << pthread_self() << "\n";
    }
  });

  {
    std::lock_guard<std::mutex> iolock(iomutex);
    std::cout << "Launched t: id = " << t.get_id() << "\n"
              << "            native_handle = " << t.native_handle() << "\n";
  }

  t.join();
  return 0;
}
```

The output of one particular run on my machine is:

```
$ ./thread-id-native-handle
Launched t: id = 140249046939392
            native_handle = 140249046939392
Thread: my id = 140249046939392
        my pthread id = 140249046939392
```

Both the main thread (the default thread running `main` on entry) and the spawned thread obtain the thread's ID - a standard defined concept (http://en.cppreference.com/w/cpp/thread/thread/id) for an opaque type that we can print, hold in a container (for example, mapping it to something in a `hash_map`), but not much other than that. Moreover, the thread object has the `native_handle` method that returns an "implementation defined type" for a handle that will be recognized by the platform-speficic APIs. In the output shown above two things are notable:

1. The thread ID is actually equal to the native handle.
2. Moreover, both are equal to the numeric pthread ID returned by `pthread_self`.

While the equality of `native_handle` to the pthread ID is something the standard definitely implies [1], the first one is surprising. It looks like an implementation artifact one definitely shouldn't rely upon. I examined the source code of a recent libc++ (http://libcxx.llvm.org/) and found that a `pthread_t id` is used as both the "native" handle and the actual "id" of a `thread` object [2].

All of this is taking us pretty far off the main topic of this article, so let's recap. The most important take-away from this detour section is that the underlying platform-specific thread handle is available by means of the `native_handle` method of a `std::thread`. This native handle on POSIX platforms is, in fact, the `pthread_t` ID of the thread, so a call to `pthread_self` within the thread itself is a perfectly valid way to obtain the same handle.

Setting CPU affinity programatically

As we've seen earlier, command-line tools like `taskset` let us control the CPU affinity of a whole process. Sometimes, however, we'd like to do something more fine-grained and set the affinities of specific threads from *within* the program. How do we do that?

On Linux, we can use the pthread-specific pthread_setaffinity_np (http://man7.org/linux/man-pages/man3/pthread_setaffinity_np.3.html) function. Here's an example that reproduces what we did before, but this time from inside the program. In fact, let's go a bit more fancy and pin each thread to a single known CPU by setting its affinity:

```
int main(int argc, const char** argv) {
  constexpr unsigned num_threads = 4;
  // A mutex ensures orderly access to std::cout from multiple threads.
  std::mutex iomutex;
  std::vector<std::thread> threads(num_threads);
  for (unsigned i = 0; i < num_threads; ++i) {
    threads[i] = std::thread([&iomutex, i] {
      std::this_thread::sleep_for(std::chrono::milliseconds(20));
      while (1) {
        {
          // Use a lexical scope and lock_guard to safely lock the mutex only
          // for the duration of std::cout usage.
          std::lock_guard<std::mutex> iolock(iomutex);
          std::cout << "Thread #" << i << ": on CPU " << sched_getcpu() << "\n";
        }

        // Simulate important work done by the tread by sleeping for a bit...
        std::this_thread::sleep_for(std::chrono::milliseconds(900));
      }
    });

    // Create a cpu_set_t object representing a set of CPUs. Clear it and mark
    // only CPU i as set.
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(i, &cpuset);
    int rc = pthread_setaffinity_np(threads[i].native_handle(),
                                    sizeof(cpu_set_t), &cpuset);
    if (rc != 0) {
      std::cerr << "Error calling pthread_setaffinity_np: " << rc << "\n";
    }
  }

  for (auto& t : threads) {
    t.join();
  }
  return 0;
}
```

Note how we use the `native_handle` method discussed earlier in order to pass the underlying native handle to the pthread call (it takes a `pthread_t` ID as its first argument). The output of this program on my machine is:

```
$ ./set-affinity
Thread #0: on CPU 0
Thread #1: on CPU 1
Thread #2: on CPU 2
Thread #3: on CPU 3
Thread #0: on CPU 0
Thread #1: on CPU 1
Thread #2: on CPU 2
Thread #3: on CPU 3
Thread #0: on CPU 0
Thread #1: on CPU 1
Thread #2: on CPU 2
Thread #3: on CPU 3
^C
```

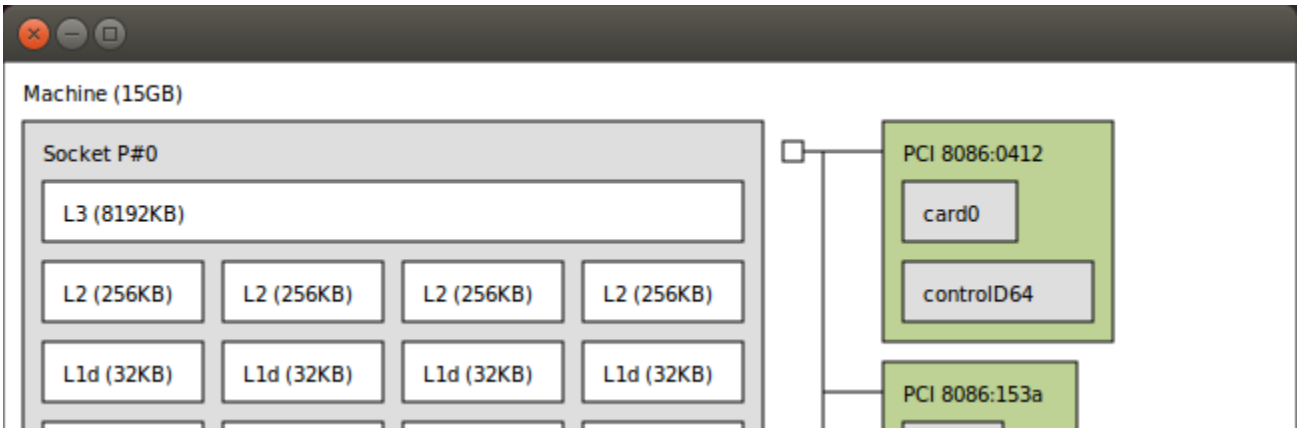
The threads get pinned to single CPUs exactly as requested.

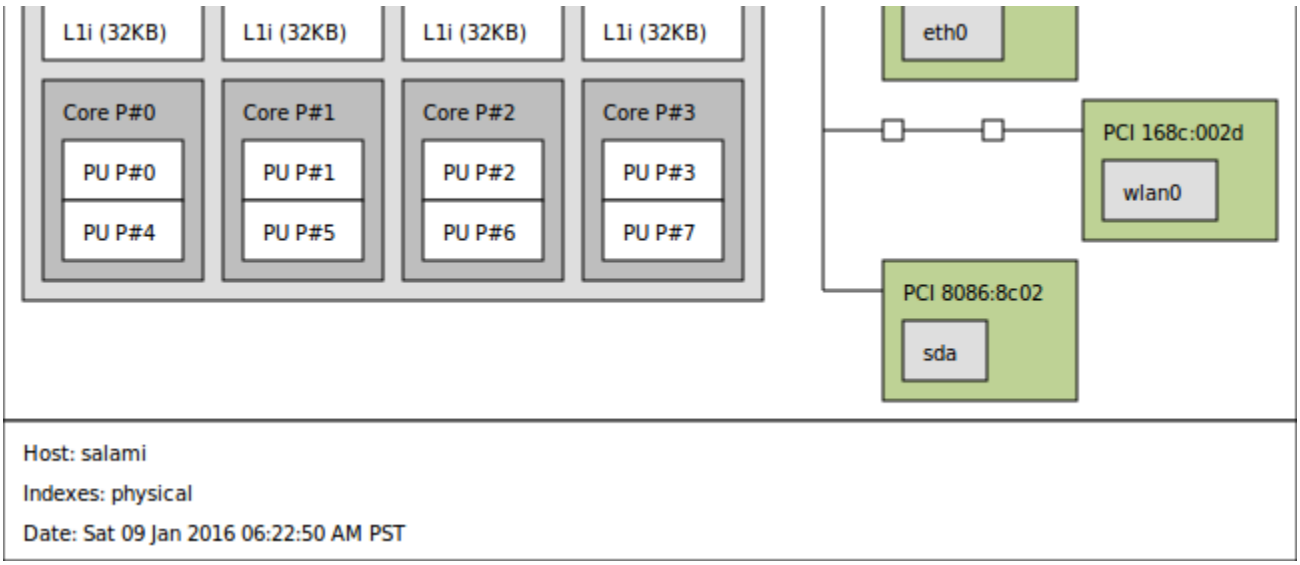
Sharing a core with hyperthreading

Now's time for the really fun stuff. We've learned about CPU topologies a bit, and then developed progressively more complex programs using the C++ threading library and POSIX calls to fine-tune our use of the CPUs in a given machine, up to selecting exactly which thread runs on which CPU.

But why any of this matters? Why would you want to pin threads to certain CPUs? Doesn't it make more sense to let the OS do what it's good at and manage the threads for you? Well, in most cases yes, but not always.

See, not all CPUs are alike. If you have a modern processor in your machine, it most likely has multiple cores, each with multiple hardware threads - usually 2. For example as I've shown in the beginning of the article, my (Haswell) processor has 4 cores, each with 2 threads, for a total of HW 8-threads - 8 logical CPUs for the OS. I can use the excellent `lstopo` tool to display the topology of my processor:





An alternative non-graphical way to see which threads share the same core is to look at a special system file that exists per logical CPU. For example, for CPU 0:

```
$ cat /sys/devices/system/cpu/cpu0/topology/thread_siblings_list
0,4
```

More powerful (server-class) processors will have multiple sockets, each with a multi-core CPU. For example, at work I have a machine with 2 sockets, each of which is a 8-core CPU with hyper-threading enabled: a total of 32 hardware threads. An even more general case is usually brought under the umberlla of NUMA (https://en.wikipedia.org/wiki/Non-uniform_memory_access), where the OS can take charge of multiple very-loosely connected CPUs that don't even share the same system memory and bus.

The important question to ask is - what *do* hardware threads share, and how does it affect the programs we write. Take another look at the `lstopo` diagram shown above. It's easy to see that L1 and L2 caches are shared between the two threads in every core. L3 is shared among all cores. For multi-socket machines. cores on the same socket share L3 but each socket usually has its own L3. In NUMA, each processor usually has access to its own DRAM, and some communication mechanism is used for one processor to access the DRAM of another processor.

Caches isn't the only thing threads within a core share, however. They also share many of the core's execution resources, like the execution engine, system bus interface, instruction fetch and decode units, branch predictors and so on [3].

So if you've wondered why hyper-threading is sometimes considered a trick played by CPU vendors, now you know. Since the two threads on a core share so much, they are not fully independent CPUs in the general sense. True, for some workloads this arrangement is beneficial, but for some it's not. Sometimes it can even be harmful, as the hordes of "how to

disable hyper-threading to improve app X's performance" threads online imply.

Performance demos of core sharing vs. separate cores

I've implemented a benchmark that lets me run different floating-point "workloads" on different logical CPUs in parallel threads, and compare how long these workloads take to finish. Each workload gets its own large `float` array, and has to compute a single `float` result. The benchmark figures out which workloads to run and on which CPUs from the user's input, prepares the inputs and then unleashes all the workloads in parallel in separate threads, using the APIs we've seen earlier to set the precise CPU affinity of each thread as requested. If you're interested, the full benchmark along with a `Makefile` for Linux is available here (https://github.com/eliben/code-for-blog/tree/master/2016/threads-affinity); in the rest of the post I'll just paste short code snippets and results.

I'll be focusing on two workloads. The first is a simple accumulator:

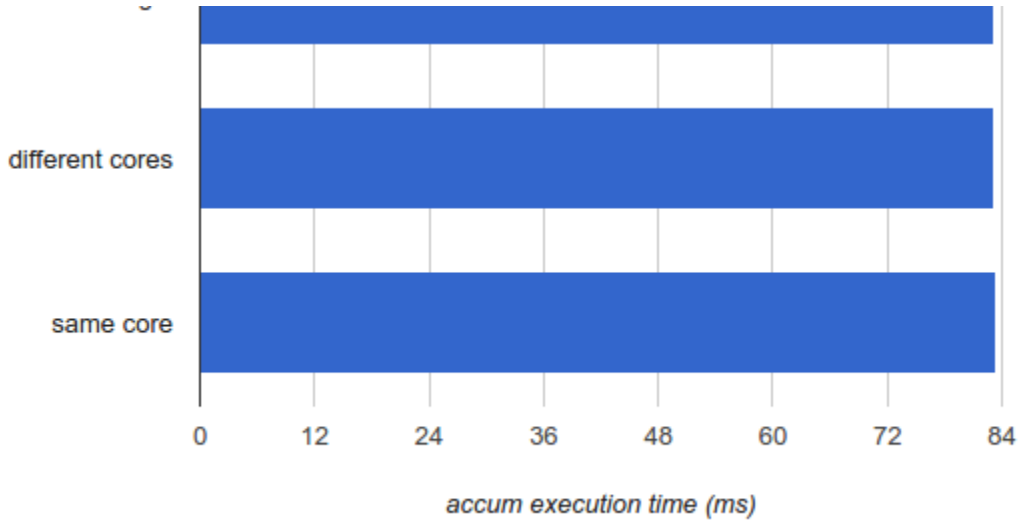
```
void workload_accum(const std::vector<float>& data, float& result) {
  auto t1 = hires_clock::now();
  float rt = 0;
  for (size_t i = 0; i < data.size(); ++i) {
    rt += data[i];
  }
  result = rt;

  // ... runtime reporting code
}
```

It adds up all the floats in the input array together. This is akin to what `std::accumulate` would do.

Now I'll run three tests:

1. Run `accum` on a single CPU, to get a baseline performance number. Measure how long it takes.
2. Run two `accum` instances on different cores. Measure how long each instance takes.
3. Run two `accum` instances on two threads of the same core [4]. Measure how long each instance takes.

The reported numbers (here and in what follows) is execution time for an array of 100 million floats as input of a single workload. I'll average them over a few runs:



This clearly shows that when a thread running `accum` shares a core with another thread running `accum`, its runtime doesn't change at all. This has good news and bad news. The good news is that this particular workload is well suitable for hyper-threading, because apparently two threads running on the same core manage not to disturb each other. The bad news is that precisely for the same reason it's not a great single-thread implementation, since quite obviously it doesn't use the processor's resources optimally.

To give a bit more details, let's look at the disassembly of the inner loop of `workload_accum`:

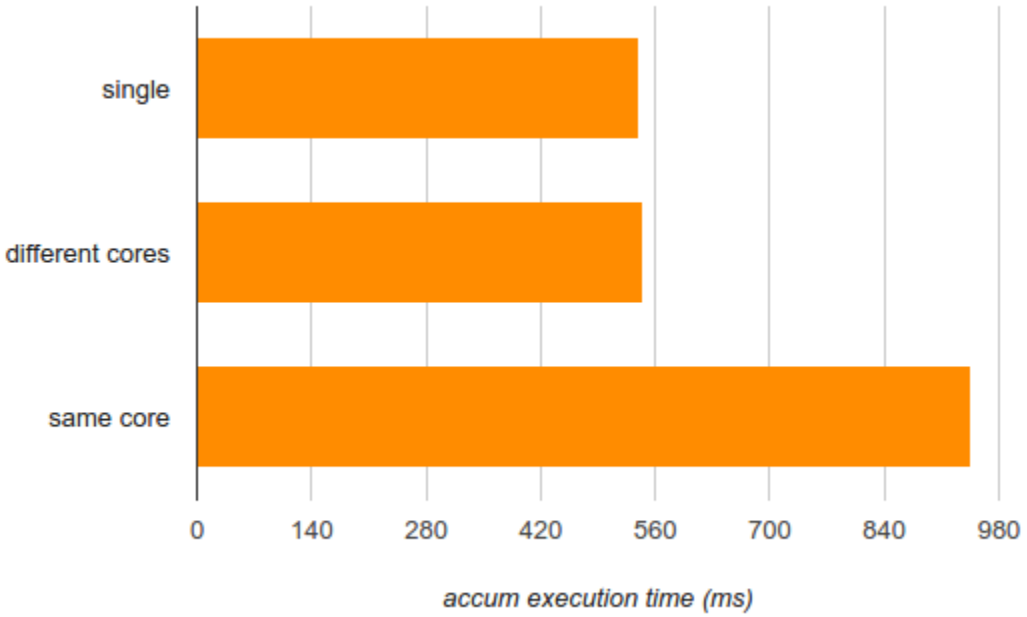
```
4028b0:       f3 41 0f 58 04 90       addss  (%r8,%rdx,4),%xmm0
4028b6:       48 83 c2 01             add    $0x1,%rdx
4028ba:       48 39 ca                cmp    %rcx,%rdx
4028bd:       75 f1                   jne    4028b0
```

Pretty straightforward. The compiler uses the `addss` SSE instruction to add floats together in the low 32 bits of a SSE (128-bit) register. On Haswell, the latency of this instruction is 3 cycles. The latency, and not the throughput, is important here because we keep adding into `xmm0`. So one addition has to finish entirely before the next one begins [5]. Moreover, while Haswell has 8 execution units, `addss` uses only one of them. This is a fairly low utilization of the hardware. Therefore, it makes sense that two threads running on the same core manage not to trample over each other.

As a different example, consider a slightly more complex workload:

```
void workload_sin(const std::vector<float>& data, float& result) {
  auto t1 = hires_clock::now();
  float rt = 0;
  for (size_t i = 0; i < data.size(); ++i) {
    rt += std::sin(data[i]);
  }
  result = rt;

  // ... runtime reporting code
}
```

Here instead of just adding the numbers up, we add their sines up. Now, `std::sin` is a pretty convoluted function that runs a reduced Taylor series polynomial approximation, and has a lot of number crunching inside it (along with a lookup table, usually). This should keep the execution units of a core more busy than simple addition. Let's check the three different modes of running again:



This is more interesting. While running on different cores didn't harm the performance of a single thread (so the computation is nicely parallelizable), running on the same core *did* hurt it - a lot (by more than 75%).

Again, there's good news here and bad news here. The good news is that even on the same core, if you want to crunch as many numbers as possible, two threads put together will be faster than a single thread (945 ms to crunch two input arrays, while a single thread would take 540 * 2 = 1080 ms to achieve the same). The bad news is that if you care about

latency, running multiple threads on the same core actually *hurts* it - the threads compete over the execution units of the core and slow each other down.

A note on portability

So far the examples in this article were Linux-specific. However, everything we went through here is available for multiple platforms, and there are portable libraries one can use to leverage this. They will be a bit more cumbersome and verbose to use than the native APIs, but if you need cross-platform portability, that's not a big price to pay. A good portable library I found useful is hwloc (https://www.open-mpi.org/projects/hwloc/), which is part of the Open MPI project. It's highly portable - running on Linux, Solaris, *BSD, Windows, you name it. In fact, the `lstopo` tool I mentioned earlier is built on `hwloc`.

`hwloc` is a generic C API that enables one to query the topology of the system (including sockets, cores, caches, NUMA nodes, etc.) as well as setting and querying affinities. I won't spend much time on it, but I did include a simple example (https://github.com/eliben/code-for-blog/blob/master/2016/threads-affinity/hwloc-example.cpp) with the source repository for this article. It shows the system's topology and binds the calling thread to a certain logical processor. It also shows how to build a program using `hwloc`. If you care about portability, I hope you will find the example useful. And if you know of any other cool uses for `hwloc`, or about other portable libraries for this purpose - drop me a line!

Closing words

So, what have we learned? We've seen how to examine and set thread affinity. We've also learned how to control placement of threads on logical CPUs by using the C++ standard threading library in conjunction with POSIX calls, and the bridging native handles exposed by the C++ threading library for this purpose. Next we've seen how we can figure out the exact hardware topology of the processor and select which threads share a core, and which threads run on different cores, and why this really matters.

The conclusion, as it always is with performance-critical code, is that measurement is the single most important thing. There are so many variables to control in modern performance tuning that it's very hard to predict in advance what will be faster, and why. Different workloads have very different CPU utilization characteristics, which makes them more or less suitable for sharing a CPU core, sharing a socket or sharing a NUMA node. Yes, the OS sees 8 CPUs on my machine, and the standard threading library even lets me query this number in a portable way; but not all of these CPUs are alike - and this is important to understand in order to squeeze the best performance out of the machine.

I haven't gone very deep into analyzing the micro-op level performance of the two presented

workloads, because that's really not the focus of this article. That said, I hope this article provides another angle to figure out what matters in multi-threaded performance. Physical resource sharing is not always taking into account when figuring out how to parallelize an algorithm - but as we've seen here, *it really should*.

---

[1] Though it can't guarantee it, since the C++ standard "doesn't know" what POSIX is.

[2] The same is done in the POSIX port of libstdc++ (though the code is somewhat more convoluted if you want to check on your own).

[3] For more details see the Wikipedia page on hyper-threading (https://en.wikipedia.org/wiki/Hyper-threading) and this post (http://www.agner.org/optimize/blog/read.php?i=6) by Agner Fog.

[4] The knowledge of which CPUs belong to the same core or different cores is taken from the `lstopo` diagram for my machine.

[5] There are ways to optimize this loop, like manually unrolling it to use several XMM registers, or even better - use the `addps` instruction to add up 4 floats at the same time. This isn't strictly safe, though, since floating-point addition is not associative. The compiler would need to see a `-ffast-math` flag to enable such optimizations.

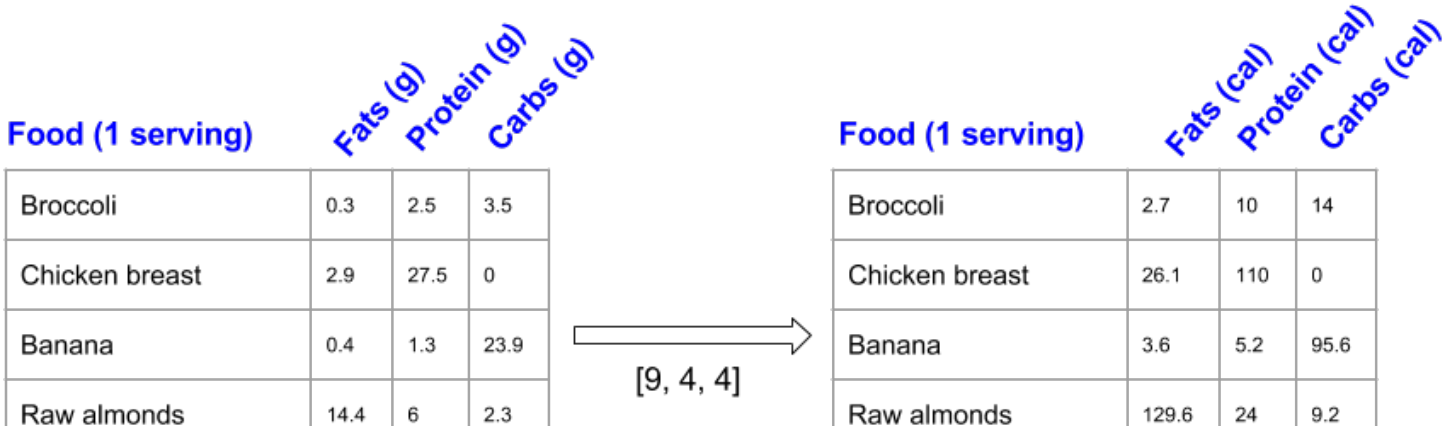
~~~

Summary of reading: October - December 2015 (http://eli.thegreenplace.net/2015/summary-of-reading-october-december-2015/)

December 31, 2015 at 07:20

- "1776" by David McCullough - on one hand I'm a bit disappointed, but on the other I'm not. And to be honest, my disappointment has only myself to blame. I should've researched better. I was looking for an introductory popular history book about the American independence war, and hey presto - here's a Pulitzer prize winner! I figured the name is 1776 because that's the year the declaration of independence was signed, but not so. The name is 1776 is because the book only tells about what happened in... that's right, 1776. Which is a tiny sliver of the actual story of the American revolution. To be sure, the story is very well told and enjoyable to read... but, this is not what I was expecting. It would be really awesome if this would be part of an 8-volume series (1775, 1776, 1777 etc.) but to the best of my knowledge it's not. A more to-the-point disappointing fact that the book spent very little on the actual declaration of independence, the process leading to it, and so on. Which is surprising, right? The book
~~~

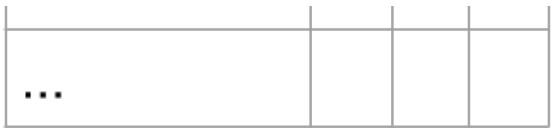
instead covers, in excruciating detail, the battle of New York / Brooklyn and the ensuing flight of the continental army to Philadelphia (with some coverage of the siege of Boston in the beginning). So if these specific events interest you in depth, the book is great. Otherwise, this would definitely not be the first (or second, or third) book I'd read on the topic of the revolutionary war.

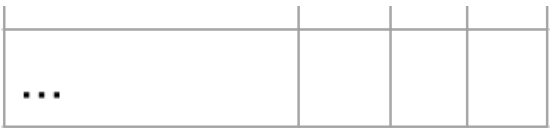
- "The Worst Journey in the World" by Apsley Cherry-Garrard - a semi-autobiographic account of Scott's expedition to reach the South Pole in 1911. This book was written in 1922 by one of the expedition's team members, and collects his own diary entries with selected entries from other members (including Scott himself). Overall well written and told - fascinating insight into early Polar exploration. One thing I think was lacking is some more context on how and why certain things were done (i.e. setting up rations, depots and so on) - these things would certainly be very interesting from an engineering/planning point of view.
- "The Achievement Habit" by Bernard Roth - a huge disappointment. I've fallen for an overly pompous short description and not reading the reviews carefully enough on Amazon. The author of the book is certainly an impressive individual, but instead of a systematic approach I expected, this book is just a biographic diary of a guy reminiscing on his life experiences. Sure, as a general feel-good book it has the usual set of good ideas - but nothing special or insightful.
- "Do No Harm: Stories of Life, Death, and Brain Surgery" by Henry Marsh - an auto-biographic account of a senior neurosurgeon in the UK about his work experiences, focusing on dealing with patients and with inevitable mistakes. Extremely well written in an emphatic and thoughtful voice. The British directness and sense of humor is also much appreciated. I also think his notes on the difference between public healthcare (NHS in the UK) and private care (in the US and the UK) are poignant. Highly recommended overall, with a word of caution for hypochondriacs - the book describes quite a few very scary medical conditions in gory detail.
- "Diaspora" by Greg Egan - a "hard science fiction" book (meaning - strong emphasis on scientific detail) following the quest of humanity's descendants (conscious software, what else...) to better understand the universe and probe beyond the edges of known physics. Pretty heavy reading, replete with hard-core physics and mathematics to such extent that it makes one wonder whether the information is real or the author is making it all up. One of the weirder books I've read in a while, for sure.
- "Battle Cry of Freedom: The Civil War Era" by James McPherson - an amazingly thorough and well-written history of the American civil war. At almost 1000 pages this book is very long and dense, but totally worth it. The author spends a lot of time on all the important aspects surrounding the war - including the political situation that led to it, the economical situation in both South and North before and during the war, the personalities involved and of course all the major battles of the war.

Re-reads:

- "Dreaming in Code" by Scott Rosenberg
- "The Making of the Atomic Bomb" by Richard Rhodes
- "Kafka on the Shore" by Haruki Murakami

~~~

Broadcasting arrays in Numpy (http://eli.thegreenplace.net/2015/broadcasting-arrays-in-numpy/)

December 22, 2015 at 06:00

Broadcasting is Numpy's terminology for performing mathematical operations between arrays with different shapes. This article will explain why broadcasting is useful, how to use it and touch upon some of its performance implications.

Motivating example

Say we have a large data set; each datum is a list of parameters. In Numpy terms, we have a 2-D array, where each row is a datum and the number of rows is the size of the data set. Suppose we want to apply some sort of scaling to all these data - every parameter gets its own scaling factor; in other words, every parameter is multiplied by some factor.

Just to have something tangible to think about, let's count calories in foods using a macro-nutrient breakdown. Roughly put, the caloric parts of food are made of fats (9 calories per gram), protein (4 calories per gram) and carbs (4 calories per gram). So if we list some foods (our data), and for each food list its macro-nutrient breakdown (parameters), we can then multiply each nutrient by its caloric value (apply scaling) to compute the caloric breakdown of each food item [1]:


~~~





With this transformation, we can now compute all kinds of useful information. For example, what is the total number of calories in some food. Or, given a breakdown of my dinner - how much calories did I get from protein. And so on.

Let's see a naive way of producing this computation with Numpy:

```
In [65]: macros = array([
  [0.3, 2.5, 3.5],
  [2.9, 27.5, 0],
  [0.4, 1.3, 23.9],
  [14.4, 6, 2.3]])

# Create a new array filled with zeros, of the same shape as macros.
In [67]: result = zeros_like(macros)

In [69]: cal_per_macro = array([9, 4, 4])

# Now multiply each row of macros by cal_per_macro. In Numpy, `*` is
# element-wise multiplication between two arrays.
In [70]: for i in xrange(macros.shape[0]):
   ....:     result[i, :] = macros[i, :] * cal_per_macro
   ....:

In [71]: result
Out[71]:
array([[   2.7,   10. ,   14. ],
       [  26.1,  110. ,    0. ],
       [   3.6,    5.2,   95.6],
       [ 129.6,   24. ,    9.2]])
```

This is a reasonable approach when coding in a low-level programming language: allocate the output, loop over input performing some operation, write result into output. In Numpy, however, this is fairly bad for performance because the looping is done in (slow) Python code instead of internally by Numpy in (fast) C code.

Since element-wise operators like `*` work on arbitrary shapes, a better way would be to delegate all the looping to Numpy, by "stretching" the `cal_per_macro` array vertically and then performing element-wise multiplication with `macros`; this moves the per-row loop from above into Numpy itself, where it can run much more efficiently:

```
# Use the 'tile' function to replicate cal_per_macro over the number
# of rows 'macros' has (rows is the first element of the shape tuple for
# a 2-D array).
In [72]: cal_per_macro_stretch = tile(cal_per_macro, (macros.shape[0], 1))

In [73]: cal_per_macro_stretch
Out[73]:
array([[9, 4, 4],
       [9, 4, 4],
       [9, 4, 4],
       [9, 4, 4]])

In [74]: macros * cal_per_macro_stretch
Out[74]:
array([[   2.7,   10. ,   14. ],
       [  26.1,  110. ,    0. ],
       [   3.6,    5.2,   95.6],
       [ 129.6,   24. ,    9.2]])
```

Nice, it's shorter too. And much, much faster! To measure the speed I created a large random data set, with 1 million rows of 10 parameters each. The loop-in-Python method takes ~2.3 seconds to churn through it. The stretching method takes 30 *milliseconds*, a ~75x speedup.

And now, finally, comes the interesting part. You see, the operation we just did - stretching one array so that its shape matches that of another and then applying some element-wise operation between them - is actually pretty common. This often happens when we want to take a lower-dimensional array and use it to perform a computation along some axis of a higher-dimensional array. In fact, when taken to the extreme this is exactly what happens when we perform an operation between an array and a scalar - the scalar is *stretched* across the whole array so that the element-wise operation gets the same scalar value for each element it computes.

Numpy generalizes this concept into *broadcasting* - a set of rules that permit element-wise computations between arrays of different shapes, as long as some constraints apply. We'll discuss the actual constraints later, but for the case at hand a simple example will suffice: our original `macros` array is 4x3 (4 rows by 3 columns). `cal_per_macro` is a 3-element array. Since its length matches the number of columns in `macros`, it's pretty natural to apply some operation between `cal_per_macro` and every row of `macros` - each row of `macros` has the exact same size as `cal_per_macro`, so the element-wise operation makes perfect sense.

Indicentally, this lets Numpy achieve two separate goals - usefulness as well as more consistent and general semantics. Binary operators like `*` are element-wise, but what

happens when we apply them between arrays of different shapes? Should it work or should it be rejected? If it works, how should it work? Broadcasting defines the semantics of these operations.

Back to our example. Here's yet another way to compute the result data:

```
In [75]: macros * cal_per_macro
Out[75]:
array([[   2.7,   10. ,   14. ],
       [  26.1,  110. ,    0. ],
       [   3.6,    5.2,   95.6],
       [ 129.6,   24. ,    9.2]])
```

Simple and elegant, and the fastest approach to boot [2].

Defining broadcasting

Broadcasting is often described as an operation between a "smaller" and a "larger" array. This doesn't necessarily have to be the case, as broadcasting applies also to arrays of the same size, though with different shapes. Therefore, I believe the following definition of broadcasting is the most useful one.

Element-wise operations on arrays are only valid when the arrays' shapes are either equal or compatible. The equal shapes case is trivial - this is the stretched array from the example above. What does "compatible" mean, though?

To determine if two shapes are compatible, Numpy compares their dimensions, starting with the trailing ones and working its way backwards [3]. If two dimensions are equal, or if one of them equals 1, the comparison continues. Otherwise, you'll see a `ValueError` raised (saying something like "operands could not be broadcast together with shapes ...").

When one of the shapes runs out of dimensions (because it has less dimensions than the other shape), Numpy will use 1 in the comparison process until the other shape's dimensions run out as well.

Once Numpy determines that two shapes are compatible, the shape of the result is simply the maximum of the two shapes' sizes in each dimension.

Put a little bit more formally, here's a pseudo-algorithm:

```
Inputs: array A with m dimensions; array B with n dimensions
p = max(m, n)
if m < p:
    left-pad A's shape with 1s until it also has p dimensions
else if n < p:
    left-pad B's shape with 1s until is also has p dimensions
result_dims = new list with p elements
for i in p-1 ... 0:
    A_dim_i = A.shape[i]
    B_dim_i = B.shape[i]
    if A_dim_i != 1 and B_dim_i != 1 and A_dim_i != B_dim_i:
        raise ValueError("could not broadcast")
    else:
        result_dims[i] = max(A_dim_i, B_dim_i)
```

Examples

The definition above is precise and complete; to get a feel for it, we'll need a few examples.

I'm using the Numpy convention of describing shapes as tuples. `macros` is a 4-by-3 array, meaning that it has 4 rows with 3 columns each, or 4x3. The Numpy way of describing the shape of `macros` is `(4, 3)`:

```
In [80]: macros.shape
Out[80]: (4, 3)
```

When we computed the caloric table using broadcasting, what we did was an operation between `macros` - a `(4, 3)` array, and `cal_per_macro`, a `(3,)` array [4]. Therefore, following the broadcasting rules outlined above, the shape `(3,)` is left-padded with 1 to make comparison with `(4, 3)` possible. The shapes are then deemed compatible and the result shape is `(4, 3)`, which is exactly what we observed.

Schematically:

```
(4, 3)                  (4, 3)
         == padding ==>          == result ==> (4, 3)
(3,)                    (1, 3)
```

Here's another example, broadcasting between a 3-D and a 1-D array:

```
(3,)                      (1, 1, 3)
           == padding ==>             == result ==> (5, 4, 3)
(5, 4, 3)                 (5, 4, 3)
```

Note, however, that only left-padding with 1s is allowed. Therefore:

```
(5,)                      (1, 1, 5)
          == padding ==>            ==> error (5 != 3)
(5, 4, 3)                 (5, 4, 3)
```

Theoretically, had the broadcasting rules been less rigid - we could say that this broadcasting is valid if we *right-pad* `(5,)` with 1s. However, this is not how the rules are defined - therefore these shapes are incompatible.

Broadcasting is valid between higher-dimensional arrays too:

```
(5, 4, 3)                    (1, 5, 4, 3)
             == padding ==>               == result ==> (6, 5, 4, 3)
(6, 5, 4, 3)                 (6, 5, 4, 3)
```

Also, in the beginning of the article I mentioned that broadcasting does not necessarily occur between arrays of different number of dimensions. It's perfectly valid to broadcast arrays with the same number of dimensions, as long as they are compatible:

```
(5, 4, 1)
          == no padding needed ==> result ==> (5, 4, 3)
(5, 1, 3)
```

Finally, scalars are treated specially as 1-dimensional arrays with size 1:

```
In [93]: ones((4, 3)) + 1
Out[93]:
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])

# Is the same as:

In [94]: one = ones((1, 1))

In [95]: one
Out[95]: array([[ 1.]])

In [96]: ones((4, 3)) + one
Out[96]:
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
```

Explicit broadcasting with numpy.broadcast

In the examples above, we've seen how Numpy employs broadcasting behind the scenes to match together arrays that have compatible, but not similar, shapes. We can also ask Numpy for a more explicit exposure of broadcasting, using the `numpy.broadcast` class:

```
In [103]: macros.shape
Out[103]: (4, 3)

In [104]: cal_per_macro.shape
Out[104]: (3,)

In [105]: b = broadcast(macros, cal_per_macro)
```

Now `b` is an object of type `numpy.broadcast`, and we can query it for the result shape of broadcasting, as well as use it to iterate over pairs of elements from the input arrays in the order matched by broadcasting them:

```
In [108]: b.shape
Out[108]: (4, 3)

In [120]: for i, j in b:
              print '{0}:  {1}  {2}'.format(b.index, i, j)
   .....:
1:  0.3  9
2:  2.5  4
3:  3.5  4
4:  2.9  9
5:  27.5  4
6:  0.0  4
7:  0.4  9
8:  1.3  4
9:  23.9  4
10:  14.4  9
11:  6.0  4
12:  2.3  4
```

This lets us see very explicitly how the "stretching" of `cal_per_macro` is done to match the shape of `macros`. So if you ever want to perform some complex computation on two arrays whose shapes aren't similar but compatible, and you want to use broadcasting, `numpy.broadcast` can help.

Computing outer products with broadcasting

As another cool example of broadcasting rules, consider the outer product of two vectors.

In linear algebra, it is customary to deal with column vectors by default, using a transpose for row vector. Therefore, given two vectors x and y, their "outer product" is defined as xy^T. Treating x and y as Nx1 matrices this matrix multiplication results in:

$$xy^T = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_N \end{bmatrix} [y_1, y_2, ..., y_N] = \begin{bmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_N \\ x_2y_1 & x_2y_2 & \cdots & x_2y_N \\ \vdots & & & \\ x_Ny_1 & x_Ny_2 & \cdots & x_Ny_N \end{bmatrix}$$

How can we implement this in Numpy? Note that the shape of the row vector is `(1, N)` [5]. The shape of the column vector is `(N, 1)`. Therefore, if we apply an element-wise operation between them, broadcasting will kick in, find that the shapes are compatible and the result shape is `(N, N)`. The row vector is going to be "stretched" over N rows and the column vector over N columns - so we'll get the outer product! Here's an interactive session that demonstrates this:

```
In [137]: ten = arange(1, 11)

In [138]: ten
Out[138]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

In [139]: ten.shape
Out[139]: (10,)

# Using Numpy's reshape method to convert the row vector into a
# column vector.
In [140]: ten.reshape((10, 1))
Out[140]:
array([[ 1],
       [ 2],
       [ 3],
       [ 4],
       [ 5],
       [ 6],
       [ 7],
       [ 8],
       [ 9],
       [10]])

In [141]: ten.reshape((10, 1)).shape
Out[141]: (10, 1)

# Let's see what the 'broadcast' class tells us about the resulting
# shape of broadcasting ten and its column-vector version
In [142]: broadcast(ten, ten.reshape((10, 1))).shape
Out[142]: (10, 10)

In [143]: ten * ten.reshape((10, 1))
Out[143]:
array([[  1,   2,   3,   4,   5,   6,   7,   8,   9,  10],
       [  2,   4,   6,   8,  10,  12,  14,  16,  18,  20],
       [  3,   6,   9,  12,  15,  18,  21,  24,  27,  30],
       [  4,   8,  12,  16,  20,  24,  28,  32,  36,  40],
       [  5,  10,  15,  20,  25,  30,  35,  40,  45,  50],
       [  6,  12,  18,  24,  30,  36,  42,  48,  54,  60],
       [  7,  14,  21,  28,  35,  42,  49,  56,  63,  70],
       [  8,  16,  24,  32,  40,  48,  56,  64,  72,  80],
       [  9,  18,  27,  36,  45,  54,  63,  72,  81,  90],
       [ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]])
```

The output should be familiar to anyone who's finished grade school, of course.

Interestingly, even though Numpy has a function named `outer` that computes the outer product between two vectors, my timings show that at least in this particular case broadcasting multiplication as shown above is more than twice as fast - so be sure to always measure.

Use the right tool for the job

I'll end this article with another educational example that demonstrates a problem that can be solved in two different ways, one of which is much more efficient because it uses the right tool for the job.

Back to the original example of counting calories in foods. Suppose I just want to know how many calories each serving of food has (total from fats, protein and carbs).

Given the `macros` data and a `cal_per_macro` breakdown, we can use the broadcasting multiplication as seen before to compute a "calories per macro" table efficiently, for each food. All that's left is to add together the columns in each row into a sum - this will be the number of calories per serving in that food:

```
In [160]: macros * cal_per_macro
Out[160]:
array([[   2.7,   10. ,   14. ],
       [  26.1,  110. ,    0. ],
       [   3.6,    5.2,   95.6],
       [ 129.6,   24. ,    9.2]])

In [161]: sum(macros * cal_per_macro, axis=1)
Out[161]: array([  26.7,  136.1,  104.4,  162.8])
```

Here I'm using the `axis` parameter of the `sum` function to tell Numpy to sum only over axis 1 (columns), rather than computing the sum of the whole multi-dimensional array.

Looks easy. But is there a better way? Indeed, if you think for a moment about the operation we've just performed, a natural solution emerges. We've taken a vector (`cal_per_macro`), element-wise multiplied it with each row of `macros` and then added up the results. In other words, we've computed the dot-product of `cal_per_macro` with each row of `macros`. In linear algebra there's already an operation that will do this for the whole input table: matrix multiplication. You can work out the details on paper, but it's easy to see that multiplying the matrix `macros` on the right by `cal_per_macro` as a column vector, we get the same result. Let's check:

```
# Create a column vector out of cal_per_macro
In [168]: cal_per_macro_col_vec = cal_per_macro.reshape((3, 1))

# Use the 'dot' function for matrix multiplication. Starting with Python 3.5,
# we'll be able to use an operator instead: macros @ cal_per_macro_col_vec
In [169]: macros.dot(cal_per_macro_col_vec)
Out[169]:
array([[  26.7],
       [ 136.1],
       [ 104.4],
       [ 162.8]])
```

On my machine, using `dot` is 4-5x faster than composing `sum` with element-wise multiplication. Even though the latter is implemented in optimized C code in the guts of Numpy, it has the disadvantage of moving too much data around - computing the intermediate matrix representing the broadcasted multiplication is not really necessary for the end product. `dot`, on the other hand, performs the operation in one step using a highly optimized BLAS routine (https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms).

---

[1] For the pedantic: I'm taking these numbers from http://www.calorieking.com (http://www.calorieking.com), and I subtract the fiber from total carbs because it doesn't count for the calories.

[2] About 30% faster than the "stretching" method. This is mostly due to the creation of the `..._stretch` array, which takes time. Once the stretched array is there, the broadcasting method is ~5% faster - this difference being due to a better use of memory (we don't *really* have to create the whole stretched array, do we? It's just repeating the same data so why waste so much memory?)

[3] For the shape `(4, 3, 2)` the trailing dimension is 2, and working from 2 "backwards" produces: 2, then 3, then 4.

[4] Following the usual Python convention, single-element tuples also have a comma, which helps us destinguish them from other entities.

[5] More precisely, `(1, N)` is the shape of a 1-by-N matrix (matrix with one row and N columns). An actual row vector is just a 1D array with the single-dimension shape `(10,)`. For most purposes, the two are equivalent in Numpy.

↑ Back to top