1 Architecture/Algorithm Interactions

- As we develop fast solution algorithms, it is important to understand the landscape of the computer architectures to which they are to be mapped.

- Here, we consider single-core performance issues.
  - We will also discuss distributed-memory multicore performance considerations (e.g., communication overhead) throughout the course, where relevant.

- Most CPU cores today have multiple pipelined functional units (load, add, multiply, etc.) that allow CPUs to produce one or more results per clock cycle, leading to high performance potential.

- Unfortunately, the peak performance of modern processors is rarely realized in practice.

- There are two basic reasons for this situation. The first arises from difficulties in scheduling operations to keep the pipelines and multiple fma (floating-point multiply-add) units full.

- The second, and more serious, results from the fact that improvements in memory access rates have not kept pace with CPU cycle times, making it difficult to keep the CPUs from being data starved.

- The following points should be considered when developing code for modern architectures:
  - reducing vector dependencies
  - eliminating loop clutter
  - increasing data reuse
  - using unit-stride data accesses

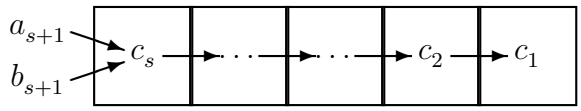
- The first pair addresses the issue of pipelining, while the second focuses on memory subsystem performance.

- While this list is by no means complete, these few basic concepts can easily affect performance by an order-of-magnitude.

2 Pipelining

- Floating-point operations such as addition or multiplication typically require s=2–5 stages to produce a *single* result, with each stage requiring one clock cycle.

- As illustrated below, pipelined functional units allow s operations to be in progress simultaneously, yielding one result per clock cycle.



Pipelined execution of $c_i = a_i + b_i$ for an s-stage pipe.
Result c_1 is ready as a_{s+1} and b_{s+1} enter the pipe.

- In order to realize this potential s-fold increase in performance, the operands entering the pipe must not depend on partially completed results still in the pipe.

- Such dependencies arise, for example, in the the backward substitution phase of a tridiagonal solve:

```
do i=n-1,1,-1
   x(i) = (b(i) - u(i)*x(i+1))/d(i)
enddo
```

- Here, evaluation of $x(i)$ must wait until the result $x(i+1)$ is complete.

- More subtle examples arise from ambiguous references, such as the following.

(1)
```
do i=1,n
     x(i+k) = c*x(i) + a(i)
enddo
```

(2)
```
do i=1,n
     x(ind(i)) = x(ind(i)) + a(i)
enddo
```

- In (1) the loop cannot be scheduled for pipelining unless the sign of k is known at compile time.
- The same is true of (2) because *ind(i)* can potentially point to a dependent address.

- Other operations that inhibit pipelining in compute-intensive loops are subroutine calls, function evaluations, nested loops, *if* statements, and I/O—all of which can lead to ambiguous execution paths.

- In addition, unnecessary divides can significantly reduce performance on some architectures.

- Finally, if the hardware is equipped with multiple function units, these are most likely to come as multiply/add (fma) pairs.

  - If the vector operation does not have an equal number of additions and multiplications, the performance will achieve only a fraction of the peak.

  - For example, FFTs have twice as many additions as multiplications, so the multipliers will be idle at least half the time if both function units have the same number of stages.

  - Fortunately, linear algebra is dominated by operations with an equal number of adds and multiplies.

3 Caches

- By adhering to basic vectorization principles, one should be able to achieve a significant fraction of peak performance *provided there is enough data to keep the processor busy.*

- This caveat is significant because processor and memory performance are on divergent paths, with the former outpacing the latter for several decades. (e.g., [Hennesy & Patterson]).

- One approach to mitigating the issue has been the development of *caches*, that is, small amounts of fast memory between the processor and main memory.

  - Most processors today feature at least a level-one (L1) cache that, under ideal conditions, can provide data fast enough to keep the pipelines full.

  - Some also feature larger L2 caches (and L3, and so on) that are somewhat slower than L1 but still faster than main memory.

  - Typically, data can be delivered at speed to the CPU only if it is already in L1 as a result of an earlier call.

  - Data not in L1 must be loaded from a higher-level cache or main memory, then passed to the CPU.

  - This process, known as a cache miss, can take tens of clock cycles and severely impact performance.

- The following loop will illustrate some basic features of cache behavior.

```
n = 0
for k=1:ntests
   n = floor(1.2*(n+1));

   run TEST(n) on $n$ data and return number of flops; % WARM-UP

   t0 = tic;
   for loop=1:nloop;
       run TEST(n) on $n$ data and return number of flops
   end;
   time_elapsed(n) = toc(t0);

   MFLOPS(n) = (nloop*flops)/time_elapsed(n)
end;
```

- The clock overhead is amortized by multiple calls to the `TEST` routine, and a single call to `TEST` outside the timing loop avoids timing the initial load of data into cache.

- Here, `TEST(n)` is a any one of several mathematical kernels operating on vectors of length n.
  - In a few cases, we compile the code with optimization level O0 (the default), but most are compiled using O3.
  - There are additional optimizations one might consider, but that is not the objective here.
  - This test is designed to understand performance for both cached and uncached data, as both are relevant.
  - It has a side benefit of giving some indication of cache sizes and of loop overhead costs.

4 Cache Behavior

- Data will stay in cache until it is pushed out by other data whose address matches the least-significant-bits of the resident data.
- Typical cache sizes today are 64KB for L1 and 256KB for L2.
- When an entry, a_i, is loaded from memory, an entire *cache line* is pulled into L1 that comprises a_i and any neighboring elements that make up that cache line.
  - cache-line sizes of 64 bytes (8 FP64 words) are typical.
- In this way, if a_{i+1} is used immediately after a_i, it is likely to already be in cache.
- Such a scenario heavily favors *unit-stride addressing*.
- As we will see, main memory is *much* slower than the caches.

- The figure below illustrates the data situation for a 4KB cache at the end of `TEST` in our experimental setup for different values of n.

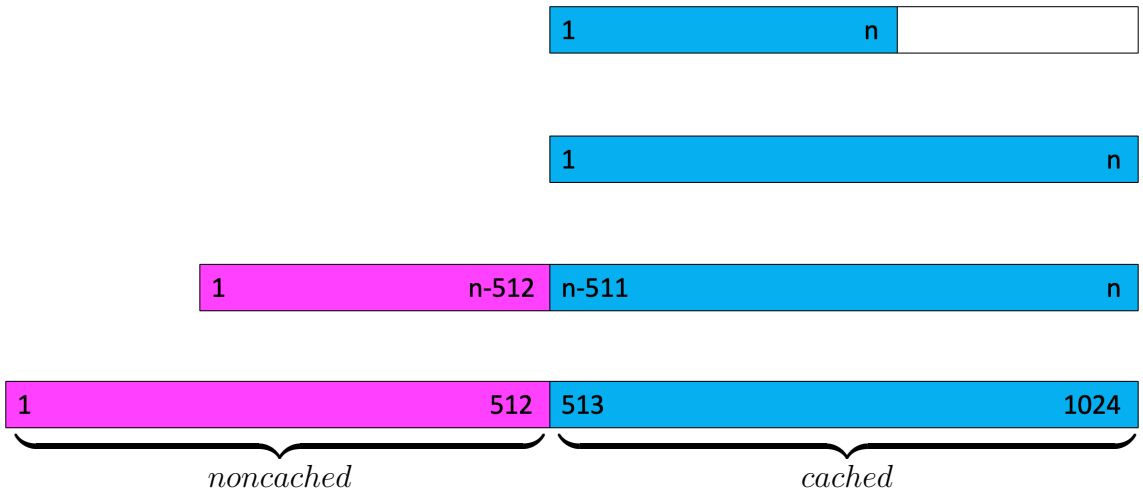


Figure 1: Status for data $[x_1 \ldots x_n]$ at the end of `TEST`: magenta–noncached; blue–cached.

- In the first two cases, data is cached after the warm-up call to `TEST` and ready for the timing loop.

- In the last case, the first half of the loop will be slow because that data is noncached.
  - Because the first half will push out initially the cached data, the second half will also be slow.
  - Data will then reside in L2 until n gets too large, at which point a similar scenario plays out until all the data must come from main memory.

- The actual displacement may vary if the kernel operates on more than one n-vector.