1 HPC Architecture: Vector/SIMD Parallelism

Computer architectures such as the CPUs in your laptop have evolved to accelerate commonly occuring operations in science and engineering. Among these, the multistage, fused multiply-add (fma) operation is one of the most important for linear algebra and STEM applications in general. A classic operation is the `AXPY` operation from the Basic Linear Algebra Subroutines (BLAS-1) set. If $\underline{x}$ and $\underline{y}$ are vectors of length n, the `AXPY` operation reads,

$$\underline{y} \;=\; a\,\underline{x} \,+\, \underline{y}, \tag{1}$$

or, in a loop:

```
for i=1:n
   y(i) = a*x(i) + y(i)
end
```

We see that, for each value of `i`, there is one multiply and one add. It turns out that for the majority of linear algebra operations, the number of additions is roughly equal to the number of additions, so having hardware support for such an operation can offer significant performance benefits. (A notable exception is the fast Fourier transform, or FFT, for which there are asymptotically two additions for each multiplication.)

Execution of multiplication (and/or addition) requires the CPU to take several steps, including fetching the data from memory, evaluating the product, and writing the result. In the product evaluation, there are several steps required between loading of the operands (say, a_i and b_i) into registers and generation of the product (say, $c_i := a_i * b_i$), with each step requiring a single clock cycle. If your CPU is running at 2 GHz and if it takes 4 cycles to execute the product, then your expected rate of multiplication would be one result every 2 nanoseconds, or 500 million results per second. Since, in this case, each result requires a single floating point operation (flop), we say that the processing rate is 500 MFLOPS–500 megaFLOPS, where FLOPS := *floating point operations per second.*

For many vector operations, it is possible to *pipeline* the operands and have multiple calculations in flight at one time, such that one result is produced per clock cycle, i.e., 2 GFLOPS (gigaFLOPS) in the preceding example. Here is a cartoon of how that would work for a 4-stage pipeline. We see that if operands a_1 and b_1 enter on cycle 1, the result emerges on cycle 5. In the interim, (a_2, b_2), (a_3, b_3), and (a_4, b_4), have entered the pipelined operator on successive respective cycles 2, 3, and 4. As result c_1 is written, operands (a_5, b_5) are entering—all assuming that the memory subsystem is able to deliver the data at the required rate, which will be the case if the operands are in the level 1 (L1) cache.

Pipelining also assumes that the operands are "ready", which means that they are not dependent on data that is still in the pipeline from a preceding operation. If the data is not ready for this reason it is referred to as a vector dependency, which can slow down the computation and which should be avoided, if possible.

Availability of an FMA implies that AXPY operations can be executed in about the same time as an isolated multiply or add, which effectively doubles the number of floating point operations per second. Many architectures support multiple FMA units, so that several FMAs may be executed in parallel.

2 Performance Testing

The concepts of pipelined and parallel architectures, vector dependencies, and data caching are important considerations in designing algorithms and deciding which algorithm might be best for a given application. In the following sections we explore the consequences of these features through a sequence of small tests that will reveal how the architecture can influence performance, by as much as an **order-of-magnitude**, in several cases.

3 Tester Background

This section contains a suite of small test loops to analyze the influence of various operations/optimizations on single-core performance. These loops are written in f77 but one could also perform a similar analysis for loops written in C and (perhaps) python.

We start with the following basic loop in cmult1.f (multiplication of a vector by a constant, loop unroll depth=1):

```
c--------------------------------------------------------------------------
      subroutine time_test(kflop,n)

      parameter (nmax=20 000 000)
      common /mydata/ x(nmax)


      kflop=1       ! # flops per entry

      reg = 0.9999999999

      do i=1,n
         x(i)=reg*x(i)
      enddo

      return
      end
c--------------------------------------------------------------------------
```

The test has the following characteristics.

1. `kflop` is an output. It indicates the number of floating point operations (flops) per entry in the vector. Here, only a single flop ("*") is performed for each entry, `i=1:n:  x(i)=reg*x(i)`, so `kflop`=1.
2. `n` is the length of the loop (or vector operation).

Note that the array `x(:)` is stored in a common block. This is by design because it tells the compiler that the data is properly byte-aligned and that `x(1)` is the first entry in a cache line, which allows the compiler to do a better job of optimizing memory management. Our goal with this exercise is to understand the influence of the contents of the loop while, to the extent possible, minimizing confounding factors such as where the data is coming from, etc. In fact, the experiment also reveals the influence of cached versus noncached data. The variable `reg` is a constant that will be loaded and stored in a register for the duration of the loop. Thus, this loop simply fetches a segment of the array `x()` (entries 1 to n), multiplies each entry by a constant, and writes it back.

To measure the time it takes to execute the loop and arrive at a floating point performance indicator, MFLOPS (Millions of flops per second), we execute in base.f the following loop segment:

```
call rone(x,n)              ! Initialize
call time_test(kflop,n)     ! Warm-up

call cpu_time(start)        !
do iloop=1,nloop            !   MAIN
   call time_test(kflop,n)  !   TIMING
enddo                       !   LOOP
call cpu_time(finish)       !

time   = (finish-start)
flops  = (kflop*n*nloop)/time ! floating point operations per second
```

The routine `rone` simply sets x(1:n)=1. Subroutine `cpu_time()` captures the time before and after the start of the routine such that the elapsed time is the difference between these. Note that CPU timers often have limited resolution—a short test might execute completely before the timer is advanced. The run time would appear to take zero seconds and thus make the apparent MFLOPs infinite. To avoid this possibility, we execute the timer test `nloop` times, where `nloop` is large for small values of n but smaller for large values of n. A simple approach is to take `nloop = 20 + 2500000/n`.

A consequence of executing the same code several times in a row is that the data in `x(1:n)` will be cached on the first call to the timing test and will remain so unless n is so large that it will not all fit. In principle, this test thus allows us to measure the cache sizes because we will see a fall-off in performance when n becomes too large. Because we can measure the performance difference, we can also understand what are the costs of bring data from main memory to (say) L2 cache and from L2 to L1.

To round out the test, we have a loop that uses larger and larger values of n, starting with n=1. For each n, we initialize `x(1:n)` and then run the test routine *outside* of the timing loop to be sure that everything is cached (when possible) before commencing the timing.

Finally, we remark that the routine `time_test` is compiled from a different file than the driver routine that calls it. This approach inhibits in-lining of these simple timer kernels and prevents over-optimization by the compiler. An in-lined version of cmult1 would look, for example, like

```
do k=1,nloop
do i=1,n
   x(i)=reg*x(i)
enddo
enddo
```

which a decent optimizing compiler might transform to

```
do i=1,n
do k=1,nloop
   x(i)=reg*x(i)
enddo
enddo
```

This transformation increases the floating point intensity (number of flops per byte of memory transferred to cache) by a factor of nloop. Ways to inhibit this artificial inflation include slightly changing the original loop structure, e.g.,

```
      do k=1,nloop
         do i=1,n
            x(i)=reg*x(i)
         enddo
         x(n)=1.1*x(n)
      enddo
```

The update `x(n)=1.1*x(n)` inhibits use of the transformation above. Alternatively, we can compile `time_test` in isolation, as done here, which prevents the optimizer from exploiting the artificially-simple context in which it is being used for these timing tests.

We make two further remarks about the compilation process. First, we compile everything with type-promotion, which promotes the fortran declared `real` variables and constants to real*8 (i.e., 64-bit floating-point arithmetic or about 16 significant digits). So called FP64 is the standard for large-scale scientific computing. Timings done in FP64 thus provide a more accurate measure of performance than if they were done in lower precision. Second, we consider both optimized (-O3) and nonoptimized (-O0) compilation. Unfortunately, if one compiles (at least with gfortran on a MacBook Pro) without specifying the level of optimization, it defaults to -O0, which produces code that runs about 10× slower than the optimized code. In the next section, we will look at timing data for these two cases and compare other kernels against this baseline data.

Here is the `makefile` used for the tests:

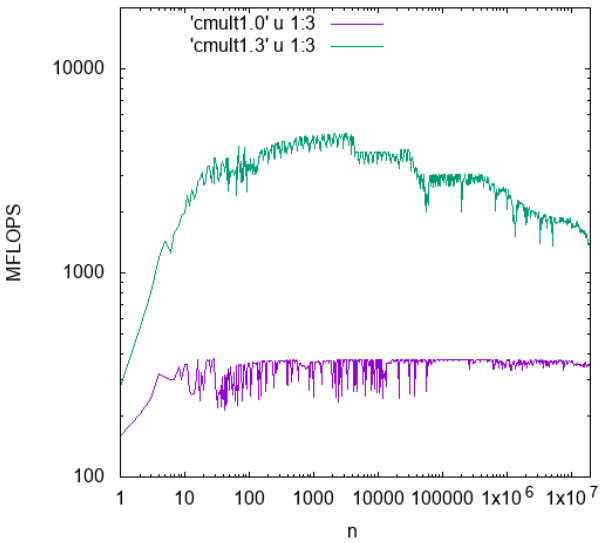
```
CC    = gcc
F77   = gfortran

FLAGS = -O3 -fdefault-real-8

NOBJS = base.o test.o

all: base print

base: $(NOBJS)
$(F77) -o base $(NOBJS)

print:
size base

clean:
'rm' *.o
'rm' base

test.o   : test.f ; $(F77) -c $(FLAGS)  test.f
base.o   : base.f ; $(F77) -c $(FLAGS)  base.f
```

4 Individual Tests

In this section, we present the results for several timing tests. We start with the `cmult1.f` example and then include these results as baseline comparisons for other kernels. My MacBook Pro peaks out around 48 GFLOPS when all four cores are running with boosted clock rates when factoring large matrices (which has $O(n)$ computational intensity). For a single core, the highest FP64 rate that I've observed is around 10+ GFLOPS.

4.1 `cmult`

Here, we consider the performance for the `cmult` loop given in the preceding section. The unoptimized code (cmult1.0) realizes about 400 MFLOPS, while the optimized code peaks around 5 GFLOPS for $n \sim$ 256–4096. For smaller problems the FLOPS is low because the subroutine call overhead is relatively large. For $n > 4096$, performance falls off because the vector no longer fits in the 32KB L1 cache.. There is a similar performance drop-off around 32,768 words, corresponding to the 256 KB L2 cache. There is a 2× performance difference between vectors of length 256–4096 and those in the ≈100,000 range. For $n >$ 500K, performance again starts to degrade. The L3 cache is 2MB per core.

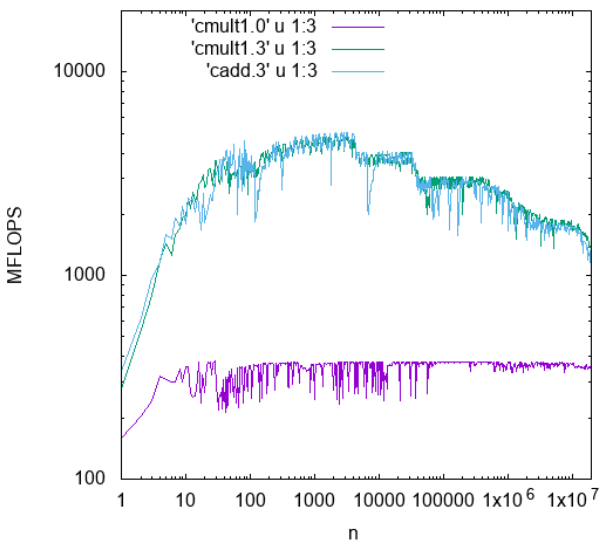



```
subroutine time_test(kflop,n)
parameter (nmax=20 000 000)
common /mydata/ x(nmax)

kflop=1        ! # flops per entry
reg = 0.9999999999

do i=1,n
   x(i)=reg*x(i)  ! cmult
enddo

return
end
```

Figure 1: Code and performance results for `cmult` baseline case: `cmult.0` = -O0, `cmult.3` = -O3.

4.2 `cadd`

We next consider $\underline{x} = \underline{x} + c$ using $-O3$. Here the performance is essentially the same as that observed for `cmult`, as would be expected. CPUs support fused-multiply-add (FMA) units that are designed to deliver one result per clock when executing pipelined arithmetic without stalls or cache misses. In this case we are simply using the adder rather than the multiplier.

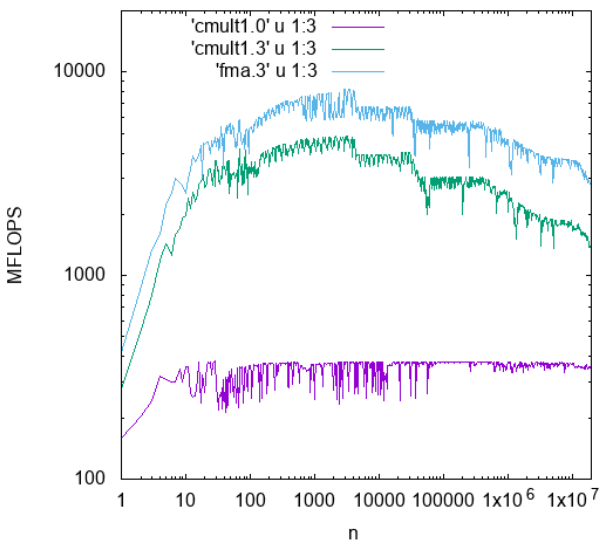



```
subroutine time_test(kflop,n)
parameter (nmax=20 000 000)
common /mydata/ x(nmax)

kflop=1        ! # flops per entry
reg = 0.9999999999

do i=1,n
   x(i)=reg+x(i)  ! cadd
enddo

return
end
```

Figure 2: Code and performance results for `cadd`, shown with baseline `cmult` results.

4.3 `fma`

Here, we exercise the full capabilities of the FMA by executing a loop that has both an add and a multiply. In this case, the performance is ultimately limited by the data transfer rates from the caches or main memory but, for each memory reference, two operations are executed, which doubles the computational intensity and FLOPS.




```
subroutine time_test(kflop,n)
parameter (nmax=20 000 000)
common /mydata/ x(nmax)


kflop=2       ! # flops per entry

r1 = 0.9999999999
r2 = 1.e-9

do i=1,n
   x(i)=r1*x(i)+r2   ! fma
enddo

return
end
```

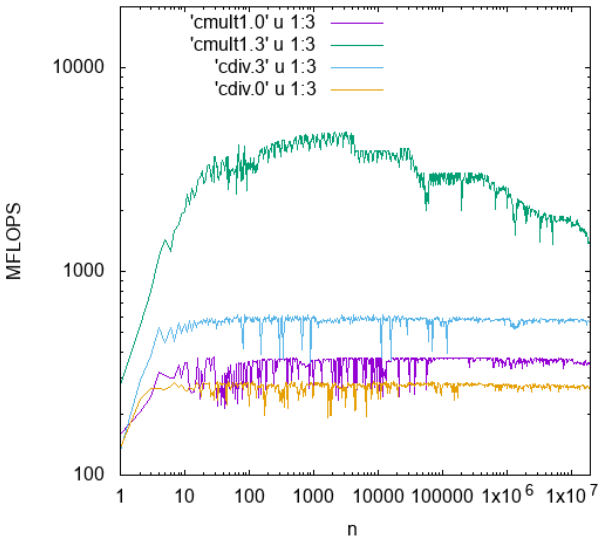
Figure 3: Code and performance results for `fma`, shown with baseline `cmult` results.

4.4 `div`

The next figure shows the FLOPS rate for division, which does not have single-clock hardware support. Division is typically performed with Newton iteration starting with an initial guess that is generated either from a look-up table or through a sign change in the exponent.

We see that `cdiv` with -O3 sustains about 600 MFLOPS whereas the -O0 version is under 280 MFLOPS. Neither show evidence of cache effects because the memory subsystem is able to keep up with this relatively slow operation.

The bottom line is that division is 4–6× slower than multiplication and thus should be avoided, where possible. (For example, take division-by-constant outside of the loop and use multiply-by-inverse instead.)




```
subroutine time_test(kflop,n)
parameter (nmax=20 000 000)
common /mydata/ x(nmax)


kflop=2       ! # flops per entry

r1 = 0.9999999999
r2 = 1.e-9

do i=1,n
   x(i)=r1/x(i)    ! div
enddo

return
end
```

Figure 4: Code and performance results for `div`, shown with baseline `cmult` results.

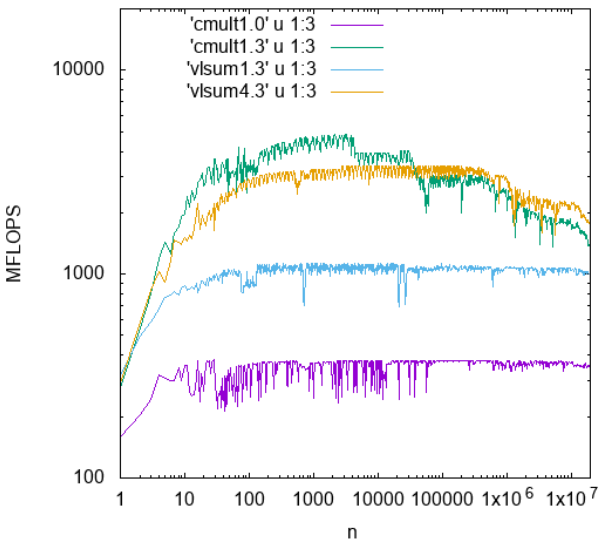
4.5 `vlsum`

Here, we consider a *vector reduction*, which is an operation that takes a vector input but produces a scalar output. As written, the loop `vlsum1`

```
!     vlsum1                          |  !     vlsum4
      s=0                             |        sum0=0
      do i=1,n                        |        sum1=0
         s = s + x(i)    ! vlsum1     |        sum2=0
      enddo                           |        sum3=0
      x(n)=s                          |        do i=1,n,4
                                      |           sum0=sum0+x(i  )  ! vlsum4
                                      |           sum1=sum1+x(i+1)
                                      |           sum2=sum2+x(i+2)
                                      |           sum3=sum3+x(i+3)
                                      |        enddo
                                      |        x(1)=sum0+sum1+sum2+sum3
```

requires the result of the sum s from iteration `i-1` before it can start the addition. This vector dependency (result not ready) prevents pipelined execution of this operation.[1]

In `vlsum4`, we unroll the summation loop such that it has four *independent* computations in the pipeline at one time. It is clear that `sum3` does not depend on `sum0, sum1,` or `sum2` and thus can start as soon as the previous execution of `sum3=sum3+x(i+3)` has completed.[2] We see that the unrolled version sustains roughly the same FLOPS as the baseline `cmult1`-O3, whereas the naive version `vlsum1` version is about 3× slower.




```
subroutine time_test(kflop,n)

parameter (nmax=20 000 000)
common /mydata/ x(nmax)

kflop=1       ! # flops per entry

sum = 0.
do i=1,n
   sum=sum+x(i)  ! vlsum1.f
enddo
x(1)=sum

return
end
```

Figure 5: Code and performance results for vector reductions shown with baseline `cmult` results.

[1] The `x(n)=s` statement in the loop above is designed to prevent the compiler from recognizing that s has no use. In such cases, compilers typically remove the loop altogether (dead code removal), resulting in the appearance of spuriously high MFLOPS.

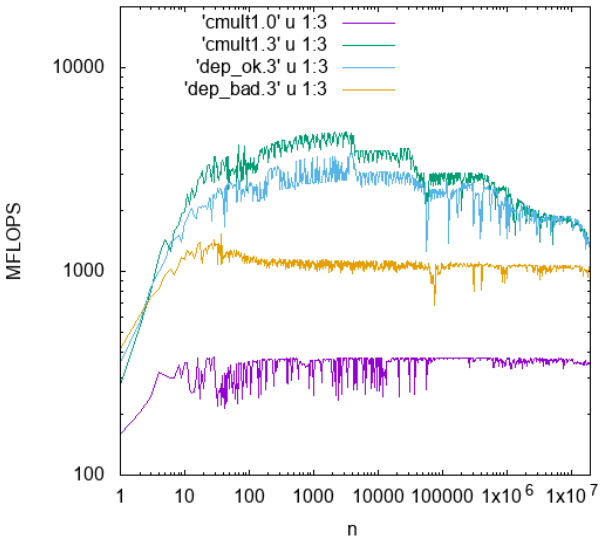
[2] In practice, one has to consider the case where n is not a multiple of 4 by using cleanup code at the end. We don't do that here as we are interested only in the influence on timing.

4.6 Vector Dependencies

Here we consider two similar-looking loop structures

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

do i=1,n
   x(i)=x(i-1)+reg
enddo
```

Notice that the first loop requires `x(i+1)` in order to start the ith step, while the second requires `x(i-1)`. In the first loop, the dependency is *known*, while in the second loop `x(i-1)` is still in the pipeline at the point where we would normally initiate computation of `x(i)`. Thus, the second loop cannot take advantage of pipelined arithmetic (i.e., vectorization) without significant mathematical reformulation. The first loop operates at about 60% of the baseline performance, whereas the dependent loop is about 3× slower.




```
      subroutine time_test(kflop,n)
      parameter (nmax=20 000 000)
      common /mydata/ x(nmax)

      kflop=1        ! # flops per entry

      do i=2,n+1
         x(i)=x(i+1)+reg    ! OK dependency
      enddo

c     do i=1,n
c        x(i)=x(i-1)+reg    ! Bad dependency
c     enddo

      return
      end
```

Figure 6: Code and performance results for dependency tests. shown with baseline `cmult` results.

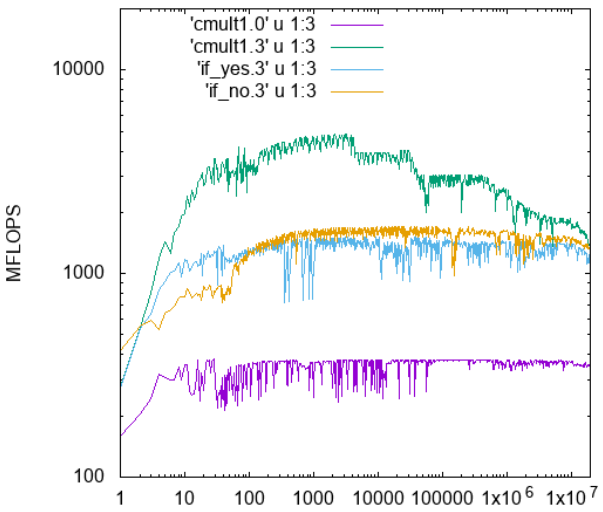
4.7 Branching/Loop-Clutter

Here we consider a loop that has a branch (i.e., an `if` statment).

```
do i=1,n
   if (x(i).lt.3) x(i) = r1*x(i)
enddo
```

Conditionals such as `if` statements inhibit vectorization because the computer cannot simply stream the content of $\underline{x}$ into the registers. Only selected elements go in and then only subsequent to inspection, required for the `if` test. Frequently compilers will assume that the test will be true and will stream that result but halt and reverse the action if it turns out to be false.

We see that cluttering the loop with an `if` statement results in a 2–3× slow down compared to the baseline.

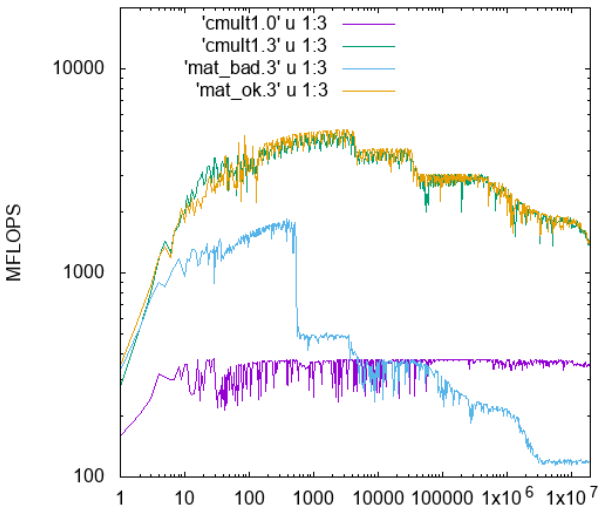



```
      subroutine time_test(kflop,n)
      parameter (nmax=20 000 000)
      common /mydata/ x(nmax)

      kflop=1       ! # flops per entry

      r1 = 0.9999999999999

      do i=1,n
         if (x(i).gt.3) x(i) = r1*x(i)  ! if_no.f
c        if (x(i).lt.3) x(i) = r1*x(i)  ! if_yes.f
      enddo

      return
      end
```

Figure 7: Code and performance results for `if` tests. shown with baseline `cmult` results.

4.8 Unit Stride Addressing

One of the fundamental concerns in HPC is to ensure that the memory system can feed data to the CPU fast enough to keep the CPU busy.

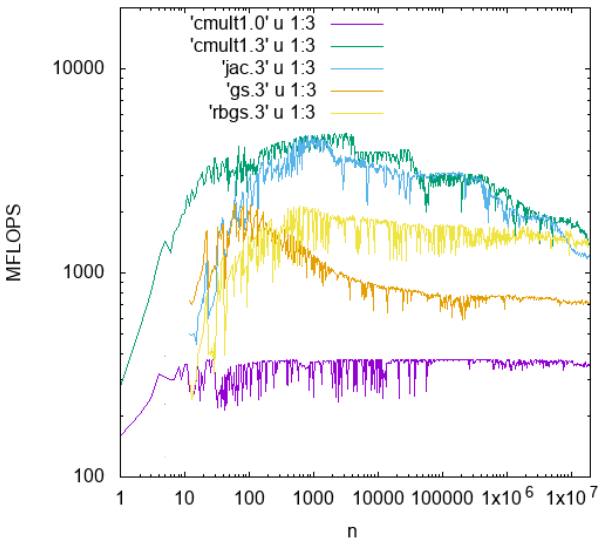



```
      subroutine time_test(kflop,n)

      parameter (nmax=20 000 000)
      common /mydata/ x(10,nmax)

      kflop=1       ! # flops per entry

      reg = 1.e-9
      do i=1,n
         x(1,i)=x(1,i)+reg ! Non-unit-stride
c        x(i,1)=x(i,1)+reg ! Unit-stride
      enddo

      return
      end
```

Figure 8: Code and performance results for unit-stride tests. shown with baseline `cmult` results.

4.9 Jacobi, Gauss-Seidel, Red-Black Gauss-Seidel

It is often argued that GS is easier than Jacobi because you can simply overwrite the results. However, standard GS has a vector dependency that inhibits vectorization and parallelization. One alternative (which also turns out to be good for multigrid) is red-black Gauss-Seidel.




```
subroutine jacobi_update(x,y,f,n)
real x(n,n),y(n,n),f(n,n)

call copy(y,x,n*n)
do j=2,n-1
   jm=j-1
   jp=j+1
   do i=2,n-1
      im=i-1
      ip=i+1
      x(i,j) = .25*(f(i,j)-y(im,j)-y(ip,j)-y(i,jm)-y(i,jp))
   enddo
enddo

return
end
```

```
c-----------------------------------------------------------------
      subroutine gs_update(x,f,n)
      real x(n,n),f(n,n)

      do j=2,n-1
         jm=j-1
         jp=j+1
         do i=2,n-1
            x(i,j) = .25*(f(i,j)-x(i-1,j)-x(i+1,j)-x(i,jm)-x(i,jp))
         enddo
      enddo

      return
      end
c-----------------------------------------------------------------
      subroutine rbgs_update(x,f,n)
      real x(n,n),f(n,n)

      do j=2,n-1
         jm=j-1
         jp=j+1

         do i=2,n-1,2
            x(i,j) = .25*(f(i,j)-x(i-1,j)-x(i+1,j)-x(i,jm)-x(i,jp))
         enddo

         do i=3,n-2,2
            x(i,j) = .25*(f(i,j)-x(i-1,j)-x(i+1,j)-x(i,jm)-x(i,jp))
         enddo

      enddo

      return
      end
c-----------------------------------------------------------------
```

Figure 9: Code and performance results for JAC/GS/RBGS tests. shown with baseline `cmult` results.

4.10 BLAS1, BLAS2, and BLAS3

The Basic Linear Algebra Subroutines (BLAS) were developed to be codified fundamental elements of linear solvers (system solves, eigenvalue solves, etc.) that could be optimized by vendors of high-performance computers, with an original focus on vector machines of late 1970s.

- The BLAS libraries are categorized as follows:
  - BLAS1: vector-vector, e.g., `daxpy:` $\underline{y} = a\underline{x} + \underline{y}$.
  - BLAS2: matrix-vector, e.g., `dgemv:` $\underline{y} = A\underline{x} + \underline{y}$.
  - BLAS3: matrix-matrix, e.g., `dgemm:` $C = AB + C$.
- What is the *computational intensity* for each of these routines?
- The original version of LinPACK was based on vector operations, i.e., BLAS1.
- In the mid- to late-80s, LinPACK was replaced by LaPACK, which was BLAS3-based and over an order-of-magnitude faster.