A HACK JOB

C. T. KELLEY

Abstract. Simon, I'm proposing that you split the half-precision sums into blocks, sum each block in half, and accumulate the block sums in a higher precision. This will do two things. The first is to make overflow very unlikely. The second is to reduce the memory footprint when your summation algorithm needs extra memory.

I have done some testing with the Julia `sum` command and will show you the codes and some results. I hope this helps.

1. Introduction. Simon, I liked your thesis. The point you make in the introduction that the computer industry could stop supporting high precision is, as you said, something many people are worried about. You should look at [1] and [2] to get more ammunition for that point.

I've put this document, the codes I used, and an IJulia notebook in a GitHub repo at

https://github.com/ctkelley/HalfTime

I'm using an Mac Mini with an M2 Pro chip (8 performance cores and 4 efficiency cores). I am not threading the loops at this point to make things simple. Threading is important and needs to be done at some point. I have a sketch of an algorithm for threading in § 1.2.4 but have not written any code.

I'm running Julia version 1.12-beta4 for my normal work. The notebook uses the 1.11 kernel because that's the version you probably have.

The first thing I want to show you is that half precision really works in this environment. I'll make a half precision vector and copy that array into single and double. I'll compare the timings and the results for summation. Half precision is faster than single, but not by the factor of 2 you'd expect. You see the factor of 2 more clearly for matrix operations where there's more work.

```
julia> using BenchmarkTools

julia> N=4096; x=rand(Float16,N); x32=Float32.(x); x64=Float64.(x);

julia> @btime sum($x); @btime(sum($x32)); @btime(sum($x64));
  140.006 ns (0 allocations: 0 bytes)
  211.136 ns (0 allocations: 0 bytes)
  368.763 ns (0 allocations: 0 bytes)

julia> err16=abs(s - s64)/abs(s64)
5.52983e-05
```

So the error is not that bad.

The timings are not the factors of 2 your might expect. The sum command may not use the SIMD registers optimally, especially for half precision. SIMD performance will get much better with Julia v1.13, which is under development and is the nightly build.

Summation has recently become an active point of discussion in the Julia Discourse. See

https://discourse.julialang.org/t/performance-challenge-can-you-write-a-faster-sum/130456/3

One thing you should probably do is benchmark your summation against the Julia command for both speed and accuracy.

The source for the Julia sum command is pretty opaque. You can get to it with this mess.

```
julia> x=rand(Float16,2);

julia> @edit sum(x)
```

After that your default editor will open sum.jl from the Julia source. I did not find that useful. It does

indicate that sum does not do the obvious thing and puts the Julia Discourse discussion in some context.

1.1. Proposal. I suggest a way to avoid overflow and reduce the memory burden by splitting the vector you're summing into blocks, sum the blocks in half precision, but accumulate the block sums in a higher precision. I'm using double for the higher precision.

Here's the algorithm. I'm using `sum` to denote a generic summation algorithm. In the codes on the repo, I'm using the Julia `sum` command. Here M is the blocksize and I'm assuming that the length N of the vector $\mathbf{x}$ is an integer multiple of M

---

blocksum$(\mathbf{x}, M)$
  $N = \text{length}(\mathbf{x})$
  $NB = N/M$
  $sumd = 0.0$
  for $ib = 1$ *to* M **do**
    $ilow = (ib - 1) \times M + 1$
    $ihigh = ib \times M$
    $sumd = sumd + sum(\mathbf{x}[ilow : ihigh])$
  end for
  Return *sumd*.

Here's the Julia code.

```
function blocksum(x, M)
    N=length(x);
# Stink test for consistency between array length and block size.
    xb=N/M
    NB=Int(floor(xb));
    (NB == xb) || error("blocksize wrong")
#
# Accumulate the block sums in double. Be aware that the
# block sums themselves are still in the precision of x.
#
    sumd=0.0
    @views for ib = 1:NB
        ilow=(ib-1)*M + 1
        ihigh=ib*M
        sumd += sum(x[ilow:ihigh])
    end
    sumd
end
```

1.2. Why is this a good idea?. There are a few reasons you might want to do this. You pretty much eliminate overflow, can reduce memory cost, and can squeeze parallelism from summation methods that are not naturally parallel.

1.2.1. It does no harm. Algorithm **blocksum** does not do the same operations as `sum` and may get different results. In particular you cannot expect the error to be reduced significantly. For the example I did above you get

```
julia> sb=blocksum(x, 128);

julia> abs(sb-s64)/abs(s64)
7.06857e-05
```

So the error is roughly the same, but still a bit larger. Since `eps(Float16) = 9.76562e-04`, both sums are acceptable.

1.2.2. Avoiding Overflow. If the blocks are small enough, one can avoid overflow by accumulating the block sums in higher precision. Even if the sum is an overflow in half precision, we at least get a result.

Here's an example of a vector where the sum overflows and produces an `Inf`.

```
julia> N=512*512; x=rand(Float16,N); x64=Float64.(x);

julia> s16=sum(x)
Inf

julia> s64=sum(x64)
1.30854e+05

julia> sblock=blocksum(x,512)
1.30851e+05
```

So `blocksum` gets pretty close to the double precision sum.

1.2.3. Reduction memory cost. If your summation method needs auxiliary storage, you can allocate dimension NB sized arrays for that instead of size N. For example suppose your summation algorithm needs two extra vectors. If you put then in a $N \times 2$ array $\mathbf{A}_S$. Then the call might look like

`summation(x, A)`

The blocked version of this would allocate a smaller array of size $NB \times 2$ and use over and over again for each block.

---

blocksum2$(\mathbf{x}, M)$
  $N = \text{length}(\mathbf{x})$
  $NB = N/M$
  $sumd = 0.0$
  Allocate an array $\mathbf{A}_S(NB, 2)$
  for $ib = 1$ *to* M **do**
    $ilow = (ib - 1) \times M + 1$
    $ihigh = ib \times M$
    $sumd = sumd + summation(\mathbf{x}[ilow : ihigh], \mathbf{A}_S)$
  end for
  Return *sumd*.

1.2.4. Parallelism. Even if algorithm within `summation` does not parallelize, you can wrap the whole thing in a loop that does. So if you have NT threads you would first split $\mathbf{x}$ into NT parts for each thread and then split each of those into blocks of size M. So the threaded loop would look like this.

---

blocksumThreaded$(\mathbf{x}, M, NT)$
  $N1 = N/NT$
  Allocate a vector $sumT(NT)$
  for $ithread = 1 : NT$ **do**
    $ilowT = (ithread - 1) \times N1 + 1$
    $highT = ithread \times N1$
    $sumt[ithread] = blocksum2(\mathbf{x}[ilow : ihigh], M)$
  end for
  $sumdt = sum(sumT)$
  Return *sumdt*

So even if `summation` does not parallelize well, you still get the benefits of threading.

Of course, you should preallocate all the auxiliary storage in advance and manage that within the loop.

REFERENCES

[1] E. Deelman, J. Dongarra, B. Hendrickson, A. Randles, D. Reed, E. Seidel, and K. Yelick, *High-performance computing at a crossroads*, Science, 387 (2025), pp. 829–831, https://doi.org/10.1126/science.adu0801.
[2] J. Dongarra, J. Gunnels, H. Bayraktar, A. Haidar, and D. Ernst, *Hardware trends impacting floating-point computations in scientific applications*, 2024, https://arxiv.org/abs/2411.12090, https://arxiv.org/abs/2411.12090.