Fast Matrix Arithmetic

The 2021 matrix arithmetic upgrades to Glasgow Pascal

Julia challenge

- I was looking into a language for planning
- I benchmarked Julia and Octave
- Julia surprisingly fast, seemed to do matrix multiply faster than Glasgow Pascal.

Matrix addition tested

- Julia

```
for i=1:L

rm1[:,:]=rm1[:,:]+rm2[:,:]
end
```

- Pascal

```
for i :=1 to L do begin
    rm1^:=rm1^+rm2^;
end;
```

Matrix multiply tested

- Julia

```
for i=1:L
    rm3=rm1 *rm2
 end
```

- Pascal

```
for i:=1 to L do
    rm3^:=rm1^ . rm2^;
```

Averages taken

Julia

Averaged over 100 iterations to allow for just in time compilation to be evened out

Pascal

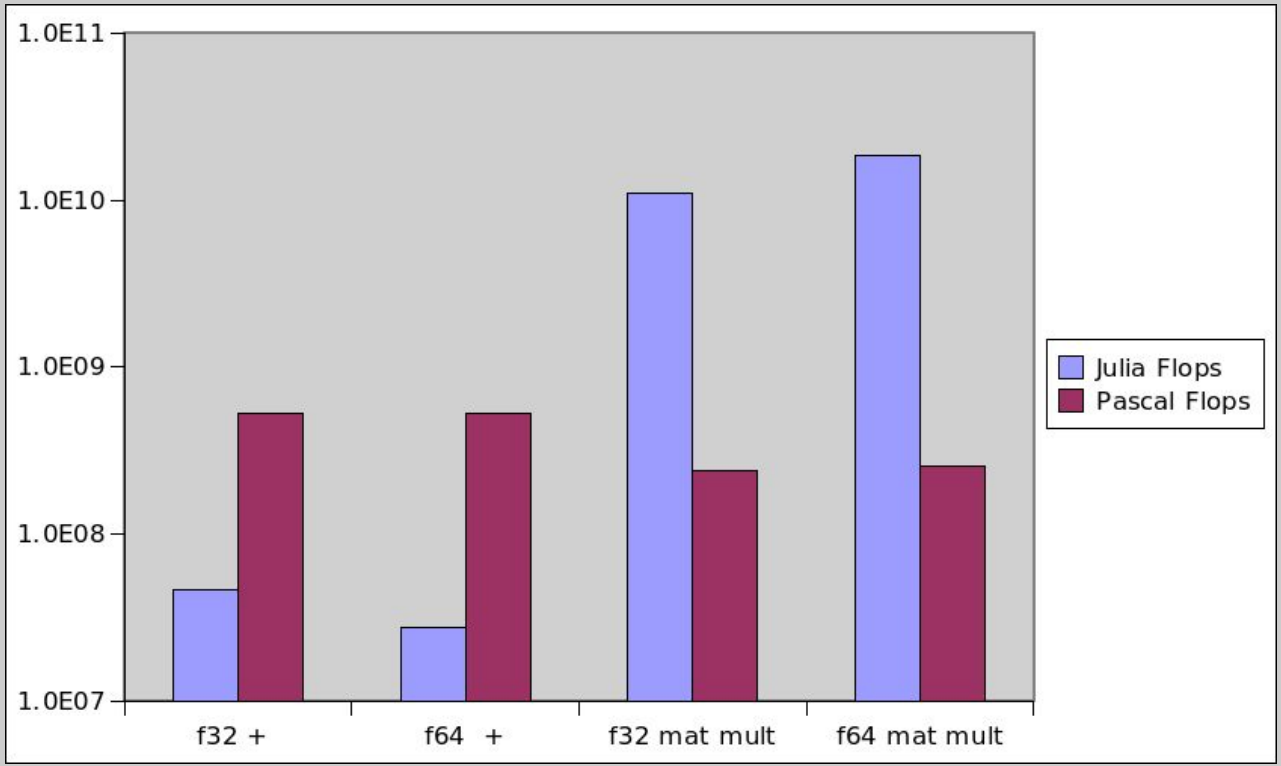
Averaged over 20 iterations just to remove noise

Results

Julia only about 8% the speed of Pascal for matrix addition

Julia 45 times faster for matrix multiply

How could it be so fast



How does Pascal handle . operator

The compiler expands the statement

```
a^:= b^.c^;
```

in place as:

```
for i:=1 to b^.rows do
for j:=1 to c^.cols do
begin
 temp:=0.0;
 for k:= 1 to b^.cols do
  temp:= temp + b^[i,k]*c^[k,j];
 a^[i,j]:=temp
end;
```

How does Pascal handle . operator

The compiler expands the statement

```
a^:= b^.c^;
```

in place as:

```
for i:=1 to b^.rows do
for j:=1 to c^.cols do
begin
 temp:=0.0;
 for k:= 1 to b^.cols do
  temp:= temp + b^[i,k]*c^[k,j];
 a^[i,j]:=temp
end;
```

This is the textbook definition of matrix multiply.

How does Pascal handle . operator

The compiler expands the statement

```
a^:= b^.c^;
```

in place as:

```
for i:=1 to b^.rows do
for j:=1 to c^.cols do
begin
 temp:=0.0;
 for k:= 1 to b^.cols do
  temp:= temp + b^[i,k]*c^[k,j];
 a^[i,j]:=temp
end;
```

This is the textbook definition of matrix multiply.

Why so slow?

250 mega flop when running on one core

Julia around 10 giga flops

How does Pascal handle . operator

The compiler expands the statement

```
a^:= b^.c^;
```

in place as:

```
for i:=1 to b^.rows do
for j:=1 to c^.cols do
begin
 temp:=0.0;
 for k:= 1 to b^.cols do
  temp:= temp + b^[i,k]*c^[k,j];
 a^[i,j]:=temp
end;
```

This is the textbook definition of matrix multiply.

Why so slow?

250 mega flop when running on one core

Julia around 10 giga flops

Is julia multi core?

Try running pascal on 16 cores

Use same machine in all cases

Compile with 16 cores - Pascal speed goes up to 2.27 G flops

Try running pascal on 16 cores

Use same machine in all cases

Compile with 16 cores - Pascal speed goes up to 2.27 G flops

But there is still a 4 fold difference, why?

in situ approach to parallelism

The approach of expanding *in situ* is the obvious approach for any language that supports statically declared array types.

It avoids having to dynamically allocate heap array temporaries whilst evaluating complex expressions ->

- Increase the working set of data degrading cache performance.
- Involve additional computational load in heap management

Not so good for matrix multiply

- The order or access to the second array C, has the row index varying most rapidly.
- 8 adjacent values from each array to be loaded into the cache.
- But the following iteration wants a value from the next row of C.
  - This will cause a cache miss.
  - The organisation of the data also inhibits the compiler from using SIMD instructions.

A fast matrix product

```
function fmatprod(a,b:pmatrix):pmatrix;
var p,c:pmatrix;i,j:integer;
begin
 p:=newmatrix(a^.rows,b^.cols);
 c:=newmatrix(b^.cols,b^.rows);
 c^:= trans b^;
 {$par}for i:= 1 to a^.rows do
       for j:= 1 to b^.cols do
        p^[i,j]:= \+( a^[i] * c^[j] );
 dispose(c);
 fmatprod:=p;
end;
```

Allocate a temporary array c

Load transpose of b into it

Then use + reduction to form the sums

Should be faster

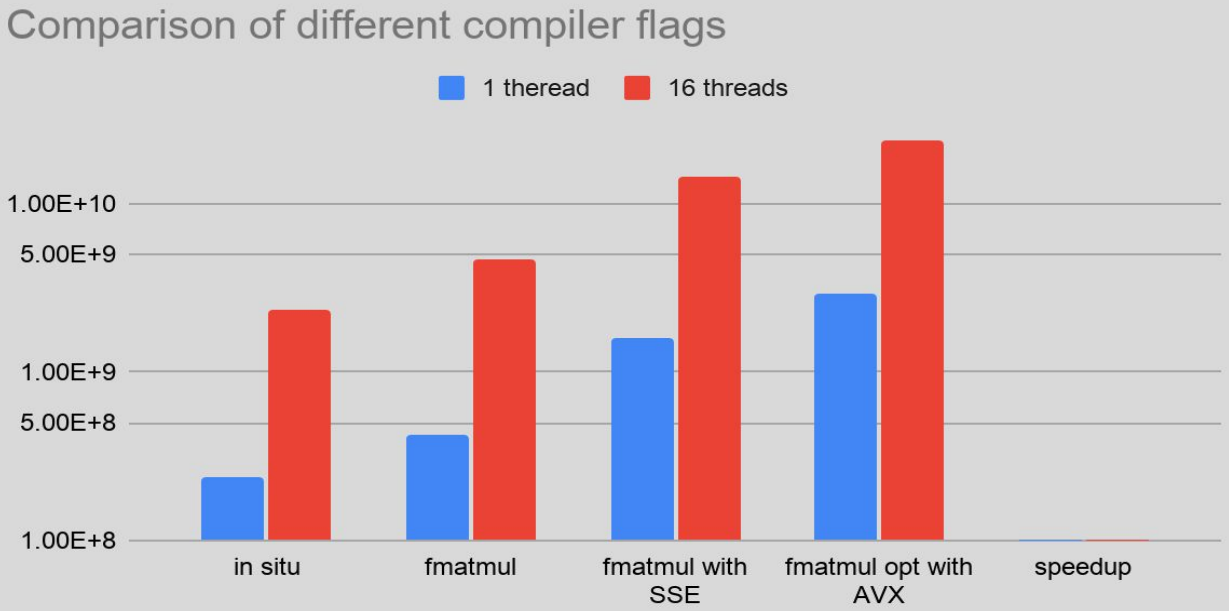
Since C is transposed we can now invoke the built in vector reduction \+ to compute the inner product of the rows.

Since the data is now organised so that sequential accesses are adjacent, both cache use improves and SIMD instructions can be generated.

How did it work

Matrix multiply operations now run at 23.5 Ggflops

97 fold speedup over single thread insitu code
10 times faster than the parallel insitu
Now twice as fast as Julia



Jan 2021 release of Glasgow Compiler

New release of the compiler with dot product operator on matrices on the heap.

Uses the fmatmul method

New manual describing use of improved methods.