APL+Win Experimental System
Scalar Function Optimization

APLNow, LLC
Mark Osborne
05 May 2008

Optimizing Scalar Function Sequences

System Version "88.1.01 May 6 2008 16:39:37 Experimental ALPHA Win/32" is an experimental version of the APL+Win system. This experimental version implements a set of optimizations on groups of certain scalar primitive functions. These optimizations take advantage of the situation in which these primitives are coded to be executed sequentially such as: `xx←a+b×c-d` or `xx←(a+b)×c-d`. In such a situation, generation of intermediate results can be skipped. In doing so, less code is executed and the resulting calculations are faster.

Parenthesized Expressions

Not all uses of such expressions benefit from optimization, so care must be taken in selecting cases to run with optimization in effect. Floating point calculations benefit to a higher degree than integer calculations do. Parenthesized expressions add another twist. In general, parenthesized expressions involving integer arithmetic do not benefit from optimization. Parenthesized expression involving floating point arithmetic may benefit. Parenthesized expressions with floating point arithmetic and less than four functions benefit more than those with four or more functions. These characteristics are due to implementation restrictions.

Optimized Functions and Operands

The scalar primitives supported by this optimization so far include: addition, subtraction, multiplication, division, maximum, minimum, and identity. Any expression including up to twenty of these primitives and either variables or constants may be optimized. The arguments must all be of the same shape and the same type (either integer or floating point). Currently mixed types – where some of the arguments are integer and some are floating point can not be optimized. An expression containing a divide function must contain only floating point values to conform.

If you attempt to optimize an expression that does not fit these requirements, the code will be checked for conformity and then run through the normal execution path. This will cause it to run slightly slower than normal.

Additional Constraints

Expressions must not use axis specifications on the primitives. Expressions that conform, but have intervening functions calls or non-scalar primitives or axis specifications will be effectively optimized on each side of the intervening functions.

The size of the arrays involved also matters. Since there is some overhead in detecting conditions for optimization and setting up the initial conditions, optimizations involving smaller array sizes will benefit less than optimizations involving larger array sizes. Very small array sizes and scalars will likely be effected negatively by optimization.

Future versions of this code will hopefully expand on cases that are conforming and cases that benefit from optimization.

Invoking Optimization

Enabling Optimization is done by setting the value of the 25th element of the system variable ⎕SYS. By default, the value of this element is zero and no attempt is made to optimize APL expressions. If the value of ⎕SYS[25] is set to one, any evaluated APL expressions will be checked for conformity and optimized of they meet the necessary conditions. If they do not conform, they will be evaluated normally. Typical use of optimization would involve surrounding target expressions with assignments to ⎕SYS[25]. Such as:

```
⎕sys[25]←1 ⍝ set optimize mode on
xx←(a+b)×c-d
⎕sys[25]←0 ⍝ set optimize mode off
```

or:

```
⎕sys[25]←1 ⋄ xx←(a+b)×c-d ⋄ ⎕sys[25]←0
```

It is important to note that as more optimizations are added it may become useful to enable specific optimizations with unique values in ⎕sys[25]. Since this is currently an experimental system, we are reserving the possibility of changing the way these values effect optimization in the next version.

Side Effects

There are some side effects to this optimization of code. The first is that it will always involve some conformity checking and some setup to run. So some cases, especially those with small arrays or with a small number of conforming primitives will actually run less efficiently. Additionally, integer and floating point overflows, underflows, infinities, and NaNs may occur at different points in processing compared to the normal path.

Code Examples

Following are some annotated examples of expressions which do and do not benefit from these optimizations:

Assuming a, b, c, d, and e are either all integer arrays or all floating point arrays, the following line will benefit from optimization. The benefit would be higher for floating point data than for integer.

```
x ← a+b×c-d+e+a+b
```

Assuming a, b, c, d, and e are all floating point arrays, the following line will benefit from optimization. For integer values, this is a non-conforming case because divide is assumed to be a floating point producing operation. So for integer values there is no advantage to optimization of this expression:

```
x ← a+b×c-d÷e+a+b
```

Assuming a, b, c, d, and e are all floating point arrays, the following line will benefit from optimization. Parenthesized floating point expressions with fewer than four primitives have special case code to handle them efficiently. For integer values, this is nominally a conforming case, but it is inefficient to implement with optimization because the stack processing involved in handling the parentheses overwhelms the expected advantage. So with integer arguments, even with optimization turned on, this expression would be evaluated normally:

```
x ← (a+b)×c-d
```

Assuming a, b, c, d, and e are all floating point arrays, the following lines will have mixed results in terms of performance gains. The first line runs faster under optimization while the second line runs about the same under optimization and normal paths. The difference is probably due to the functions chosen. In cases of parenthesized code with more than three primitives, it is probably safest to not try to run with optimization. There are several possible coding techniques that we would like to try that may improve this performance area in the future:

```
x ← a⌈(a+b×c)⌊(+d-e)
x ← (a+b)-(c÷d)×(e-a)
```

Timing Examples

The following examples show some typical results for chosen conforming cases which work well under optimization (and some that do not). All of these show results of running 10 iterations of the sample code on floating point or integer vectors of 1,000,000 values. The times are in elapsed seconds. These timings were run on a 1.5Ghz Pentium with 1GB of RAM running Windows XP/SP2. The source for these examples is in the workspace TESTOPTIMIZE03. The function "runall" will run all these examples and more:

General mixed functions with floating point arguments. Note the significant timing improvements:

```
time_F_mix_6[3]
x ← a+b×c-d÷e+a+b
normal:  0 1.583 0 0
optimized:  0 0.881 0 0
```

General mixed functions with floating point arguments in a parenthesized expression. Note that the timing improvements are not very substantial:

```
time_F_mix_6_ALL[3]
x ← a⌈(a+b×c)⌊(+d÷e)
normal:  0 1.592 0 0
optimized:  0 1.372 0 0
```

General mixed functions with floating point arguments in a parenthesized expression. Note that there is no timing improvement in the next two examples. Taken together with the previous example, it is probably not effective to try to run long parenthesized floating point expressions with optimization turned on. However, when there are fewer than four primitives there is improvement. This is shown in the subsequent example:

```
time_F_mix_5_PAREN[3]
(a+b)-(c÷d)×(e-a)
normal:  0 1.642 0 0
optimized:  0 1.652 0 0

time_F_mix_5_NEST[3]
x ← ((a+b)-(c÷d))×(e-a)
normal:  0 1.652 0 0
optimized:  0 1.682 0 0
```

General mixed functions with floating point arguments in a parenthesized expression. Note that there is moderate but significant timing improvement in the next six examples. These cases with three or fewer dyadic primitives are handled by specialized code which enables these cases to be run faster than cases with more than three primitives. In future versions this range may be extended to speed up more cases:

```
time_F_mix_2_PAREN[3]
x ← (a+b)×c
normal:  0 0.641 0 0
optimized:  0 0.54 0 0

time_F_mix_3_P4[3]
x ← a+b×c-d
normal:  0 0.951 0 0
optimized:  0 0.511 0 0

time_F_mix_3_P5[3]
x ← (a+b)×c-d
normal:  0 0.991 0 0
optimized:  0 0.691 0 0

time_F_mix_3_P6[3]
x ← a+(b×c)-d
normal:  0 0.901 0 0
optimized:  0 0.671 0 0

time_F_mix_3_P7[3]
x ← (a+b×c)-d
normal:  0 0.931 0 0
optimized:  0 0.681 0 0

time_F_mix_3_P8[3]
x ← ((a+b)×c)-d
normal:  0 0.911 0 0
optimized:  0 0.661 0 0
```

An example using only one primitive function. This shows no advantage because there are no multiple primitives to combine:

```
time_F_sub_1[3]
a-b
normal:  0 0.33 0 0
optimized:  0 0.321 0 0
```

Some examples with integer arguments. Note that with only two primitives there is no improvement:

```
time_I_mix_2[3]
x ← a+b×c
normal:  0 0.29 0 0
optimized:  0 0.291 0 0
```

More integer cases. With three primitives there is now a performance improvement:

```
time_I_mix_3_P4[3]
x ← a+b×c-d
normal:  0 0.44 0 0
optimized:  0 0.381 0 0
```

More integer cases. With parenthesized expressions there is no performance improvement. In fact the optimized and non-optimized cases run through the same code path:

```
time_I_mix_3_P8[3]
x ← ((a+b)×c)-d
normal:  0 0.451 0 0
optimized:  0 0.451 0 0
```

More integer cases. With more primitives we get more performance improvement:

```
time_I_mix_6[3]
x ← a+b×c-d+e+a+b
normal:  0 0.902 0 0
optimized:  0 0.721 0 0
```

Integer overflow. In this case we get an integer overflow (because some of the values added sum to more than `2⋆31`). In this case, when overflow detection happens, we re-run the expression through the normal floating point arithmetic path. There is no performance gain in such cases:

```
time_I_plus_6_OVF[3]
x ← a+b+c+d+e+a+b
normal:  0 1.673 0 0
optimized:  0 1.672 0 0
```

Technical Details

The core technique used in this optimization stems from the observation that a large percentage of the time spent in the calculation of floating point results is consumed by fetching and storing the operands.

Normal processing of an expression such as `a+b×c` is done by performing the multiplication across all elements of `b` and `c` to produce an intermediate result. Then the addition is performed by adding all the elements of that intermediate result to the corresponding elements of `a` to produce a final result. Functionally two loops are used to perform this. The outer loop is the interpreter main loop which selects the functions to be executed and the operands on which to apply them. The inner loop is the code implemented for each primitive.

The new code inverts these loops, so that we execute `b[n]×c[n]` to produce single result `t` then add that result to `a[n]`. That result is then stored in the result array as `r[n]` and we increment `n` and loop till we process all of the values across each array. This eliminates the construction of any intermediate arrays and all storing values to and fetching values from those intermediate arrays. So the number of instructions executed to evaluate the expression is reduced. An additional advantage of this approach is that it allows checking for floating point overflows to be done less frequently.

This technique is most effective on floating point arrays. It is less effective on integer arrays, but still leads to improvement. This technique works best on expressions with no parentheses. This is because use of parentheses to alter order of evaluation requires a stack mechanism which introduces more instructions in the inner loop. To overcome this issue, there is a separate section of code for each ordering of parentheses possible. For up to three primitives the code will analyze parenthesized expressions in advance and set up a switch statement to choose the appropriate execution path. For this many (up to three) functions in the sequence, there are twelve cases to cover. Three of these do not involve parentheses. Of the nine remaining cases only five represent unique execution orders. This can be extended further, but it is a combinatorial function. The actual primitives involved in the expression are coded as subroutines and executed by setting up pointers to these routines that are dropped into the code.

A Word of Warning

This is still an experimental system. Features and behavior of this system are not guaranteed to be correct and may change or be completely absent in a future version. Do not trust this code on production systems. Workspaces built under this version are compatible with current (Version 8) systems.