Project Proposal

Course: ENGR-E 517. Professor: Thomas Sterling.

Alex Shroyer

2021-10-03

ABSTRACT

The J programming language[1] is a high-level, array-oriented, free and open source[2] interpreted language. Originally designed by Kenneth Iverson as a successor to APL, it shares much of APL's expressivity and performance characteristics, but J uses an ASCII characters set. While the J interpreter is well-optimized for modern CPUs (especially those with SIMD and vector extensions), support for multi-core/GPU is limited. This project explores adding GPU support to J, using matrix product as the prototype.

1 J Language Basics

```
   i.3 4  NB. make the first 12 integers, as a 3x4 matrix

0 1  2  3
4 5  6  7
8 9 10 11
```

Most logical and arithemetic operators are defined for all combinations of scalar and n-dimensional array arguments:

```
   10 + i.3 4

10 11 12 13
14 15 16 17
18 19 20 21
```

J generalizes the concept underlying the conventional Σ and Π notation. Therefore, `sum` is not a special symbol, but rather the composition of + with the "insert" operator (`/`).

```
   +/ i.3 4  NB. sum matrix columns, result is a vector

12 15 18 21
```

Functional languages like Haskell sometimes call this concept "fold"[3] or "reduce".

[1]https://www.jsoftware.com/
[2]https://github.com/jsoftware/jsource
[3]https://wiki.haskell.org/Fold

2 Matrix Product

The matrix product operation is worth optimizing because it fundamental to many problems, and also computationally intensive. Because the operation is inherently parallel, it is a good fit for GPU architectures.

J has a generalized inner product `u . v` where `u` and `v` can be user-defined functions. To write a traditional matrix product, use `+/` (sum) for `u` and `*` (multiply) for `v`:

```
   x =: i.3 5  NB. assign 3x5 matrix to x
   y =: i.5 4  NB. assign 5x4 matrix to y
   x +/ . * y  NB. result shape is 3x4

120 130 140 150
320 355 390 425
520 580 640 700
```

More information about matrix product in J can be found on its vocabulary entry[4].

3 Experiments

Experiment A will generate matrixes on the CPU, execute the matrix multiplication on the GPU, and transfer the answer back to the CPU.

Experiment B will generate the data on the GPU instead; the rest will be the same as (A).

I will measure performance for the following:

1. time for (A) for matrixes of various sizes
2. total time for (B) for matrixes of various sizes
3. measure performance of matmul for various matrix sizes on GPU
4. determine break-even data size (if any) where GPU is faster

4 ArrayFire

The ArrayFire[5] project is library for C and C++ which exposes a single interface for interacting with GPUs from various vendors and with different capabilities. Alternatives to ArrayFire include writing CUDA kernels (Nvidia only), or OpenCL code for hardware that supports OpenCL. Using ArrayFire will give some level of cross-platform support, and hopefully enable more people to take advantage of GPU-enhanced J code.

5 Plan

My plan for implementing matrix product on the GPU for J is to prototype using J's external C function call interface[6]. This interface permits calling an external compiled function from within a J interpreter session.

[4] `https://code.jsoftware.com/wiki/Vocabulary/dot#dyadic`
[5] `https://arrayfire.com/`
[6] `https://code.jsoftware.com/wiki/Guides/DLLs/Calling_DLLs`

Below is a simple but complete example of creating and executing an external C function from J:

Write the C code:

```
/* mylib.c */
int foo(int x, char y){
    if(y=='a')
        return x+1000;
    return 10-x;
}
```

Compile it as a dynamic library:

```
rm *.dylib *.o
gcc -c mylib.c
g++ -dynamiclib mylib.o -o mylib.dylib
```

Finally, this function can be called from J. The `cd` function's left argument is a description of the function, including where to find it. The right arguments are passed to the external function. The results are displayed as a list of boxed values (the last boxes are copies of the original right-side arguments):

```
   './mylib.dylib foo i i c' cd 42;'a'


 1042 42 a


   './mylib.dylib foo i i c' cd 42;'b'


 _32 42 b
```

6 C API

Using this `cd` feature of J and the ArrayFire shared library, I plan to call ArrayFire's matrix multiplication function from J. This is complicated by two factors. First, most of ArrayFire's documentation refers to their C++ API, whereas `cd` expects a C interface. Second, the ArrayFire data types do not map directly to J data types, so I must learn how to define these in a way that `cd` understands.