Adding GPU Support to an Array-Oriented Language

Alex Shroyer

December 9, 2021

ABSTRACT

This report uses J's foreign function interface to execute GPU instructions, using the ArrayFire library. Timings are shown for a computationally-intensive operation (matrix multiply) on a CPU versus the same operation on a GPU. Using matrices of increasing size, I show that while the CPU is faster for small data sizes, the GPU outperforms at large data sizes. On the test hardware, matrix multiplpication of double-precision matrices larger than 200x200 is faster on the GPU.

Introduction

The J language[1] is primarily an "array-oriented" language in which most operations are implicitly data-parallel and rank polymorphic[2]. Because the language itself is parallel, parallel algorithms are easy to express and easy to analyze. Graphics processing units (GPUs) are massively parallel, so GPU programming should be a natural fit for the J language. However, currently GPU support is not widely used in J, nor in other array-oriented programming languages such as APL[3].

Related Work

This work is influenced by a project called Jfire[4], which provides ArrayFire bindings to J. However, I was unable to run Jfire on my test machine, and I struggled to understand its interface, so ultimately I took my own route.

A closed-source fork of J with OpenMP-based parallelism is available from Monument AI[5]. This interpreter uses OpenMP tasks to implement futures in the language, and also integrates BLAS support for selected linear algebra routines.

Another project related to array languages and the ArrayFire library is IU Alum Aaron Hsu's doctoral thesis, which provides a APL compiler running on a GPU, and also uses ArrayFire as the back-end [6].

Methods

Matrix Multiplication

To multiply matrix A (size $m \times n$) with matrix B (size $n \times p$), producing result C (size $m \times p$), matrix multiply is defined[7] as

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} ... + a_{in}b_{nj} = \Sigma_{k=1}^{n} a_{ik}b_{kj}$$

for $i = 1, ..., m$ and $j = 1, ..., p$.

[1]The J homepage: `https://jsoftware.com`

[2]Rank Polymorphism: `https://prl.ccs.neu.edu/blog/2017/05/04/rank-polymorphism/`

[3]Dyalog APL (a popular modern APL implementation): `https://www.dyalog.com/`

[4]Jfire: ArrayFire bindings for J: `https://github.com/Pascal-J/Jfire`

[5]Monument AI's OpenMP-enabled J interpreter: `https://www.monument.ai/m/parallel`

[6]Aaron Hsu's thesis: `https://scholarworks.iu.edu/dspace/handle/2022/24749`

[7]Matrix multiplication definition: `https://en.wikipedia.org/wiki/Matrix_multiplication`

In mathematics, this is often expressed with the following notation (although recognizing this as matrix multiplication requires contextual hints):

$$C = AB$$

In J, this is expressed with the following notation:

```
C =: A +/ .* B
```

Where `+/` is "sum", `*` is "multiply", and `.` composes these two operations.

For simplicity, these experiments will consider only square matrices of edge length N, consisting of 64-bit floating point data (`double` datatype in C). The data in the matrices are the numbers starting at 0 and counting up to `(N*N)-1`.

Example matrix (N=3):

```
0 1 2
3 4 5
6 7 8
```

Example matrix (N=5):

```
 0  1  2  3  4
 5  6  7  8  9
10 11 12 13 14
15 16 17 18 19
20 21 22 23 24
```

The `matmul` operation in this report is for two identical such matrices as its arguments, or $C = AA$.

C Function Interface

The J interpreter provides a foreign function interface (FFI) to C libraries. For a C function `int func(void* out, double* arr, double y);`, from a library at `/usr/libfunc.so`, the corresponding J FFI call is:

```
p =: ,mema 8*2  NB. allocate space for 2 64-bit words for "out"
'/usr/libfunc.so func i * *d d' cd p; 2.3 10 0.99;
```

The components of the FFI call are:

- `/usr/libfunc.so`, the library path
- `func`, the name of the function
- `i * *d d`, C type codes for the return type (`i` = `int`) and the argument types (`*` = `void*`, `*d` = `double*`, `d` = `double`)
- the `cd` J system function (short for "call dynamic library")
- the runtime arguments, including J data and/or pointers to memory allocated by J

By using this FFI, J programmers can call functions from C libraries. With a little more work, these can be wrapped into J libraries, to hide the implementation details and provide a more familiar programming experience to the J programmer.

ArrayFire

The ArrayFire library is an abstraction layer which provides a common interface for CUDA, OpenCL, and CPU platforms. It has many features, but for this work, the feature of interest is `af_matmul` [8], which does matrix multiplication. Most ArrayFire functions return an integer success code, and the first parameter is an "out" void pointer, intended to be an address where the result data may be stored.

[8] ArrayFire documentation website: https://arrayfire.org/docs/group__blas__func__matmul.htm#ga3f3f29358b44286d19ff2037547649fe

Memory Management

A specific challenge of this assignment was managing memory. Memory in typical J code is automatically reference counted, so the programmer is not concerned with explicit memory management. However, using ArrayFire requires additional memory management.

My initial attempts involved allocating pointers for data from J, providing those pointers to ArrayFire, and freeing the pointers (also in J) after obtaining the result. This worked for small matrix sizes, but I encountered segmentation violations with larger data.

I found that, with a “cold start” of the ArrayFire-calling code, the largest result I could transfer back to J without encountering a segfault was $127x127$. However, a “warm start”, in which I first multiply small matrices (say, $127x127$ each), allowed readaing larger results ($1450x1450$). This cold/warm discrepancy hints that I am improperly reading the data back into J, and so for this work to be of practical use this issue must be resolved.

But for the purpose of this report, because I have validated that the results are correct with small matrices, and also because the segfault occurs *separately* from the matrix multiply operation, I was able to obtain valid performance measurements.

In pseudocode, the cold start failure looks like this:

```
size = 128x128
A,C = allocate(size),allocate(size)
matmul(C,A,A)
copy_to_host(C) # segfault
free(A), free(C)
```

The warm start failure looks like this:

```
size = 1x1
A,C = allocate(size), allocate(size)
matmul(C,A,A)
copy_to_host(C) # fine
free(A), free(C)

# everything is "fine" from 2x2 up through 1449x1449...

size = 1449x1449
A,C = allocate(size), allocate(size)
matmul(C,A,A)
copy_to_host(C) # still fine
free(A), free(C)

# ...but any size > 1449x1449 fails:

size = 1450x1450
A,C = allocate(size),allocate(size)
matmul(C,A,A)
copy_to_host(C) # segfault
free(A), free(C)
```

Because of this issue, timing measurements for this report were taken by modifying the program to this form (pseudocode):

```
size = NxN
A,C = allocate(size),allocate(size)
print(time(matmul(C,A,A)))
```

Results

All timing measurements are captured within J, and show the duration of the matrix multiply operation only.

N	CPU	GPU (cold)	GPU (warm)
10	0.00003	0.00149	0.00003
100	0.00028	0.00157	0.00003
200	0.001639	0.00151	0.00003
500	0.01648	0.00175	0.00011
1000	0.07671	0.00172	0.00012
2000	0.43586	0.00265	0.00014
4000	3.43190	0.00635	0.00025
5000	6.48677	0.00909	0.00036
6000	10.99981	0.01261	0.00047
8000	25.81416	0.02156	0.00068
10000	50.19670	0.03276	0.00106
15000	174.86698	0.03281	0.00213

Table 1: Timing Results

To determine the “cross over” point at which the GPU outperforms the CPU, a square matrix of the first N*N integers was matrix multiplied with itself. For various values of N, representing the length of one side of the square input matrix, the time for the `af_matmul` call is recorded.

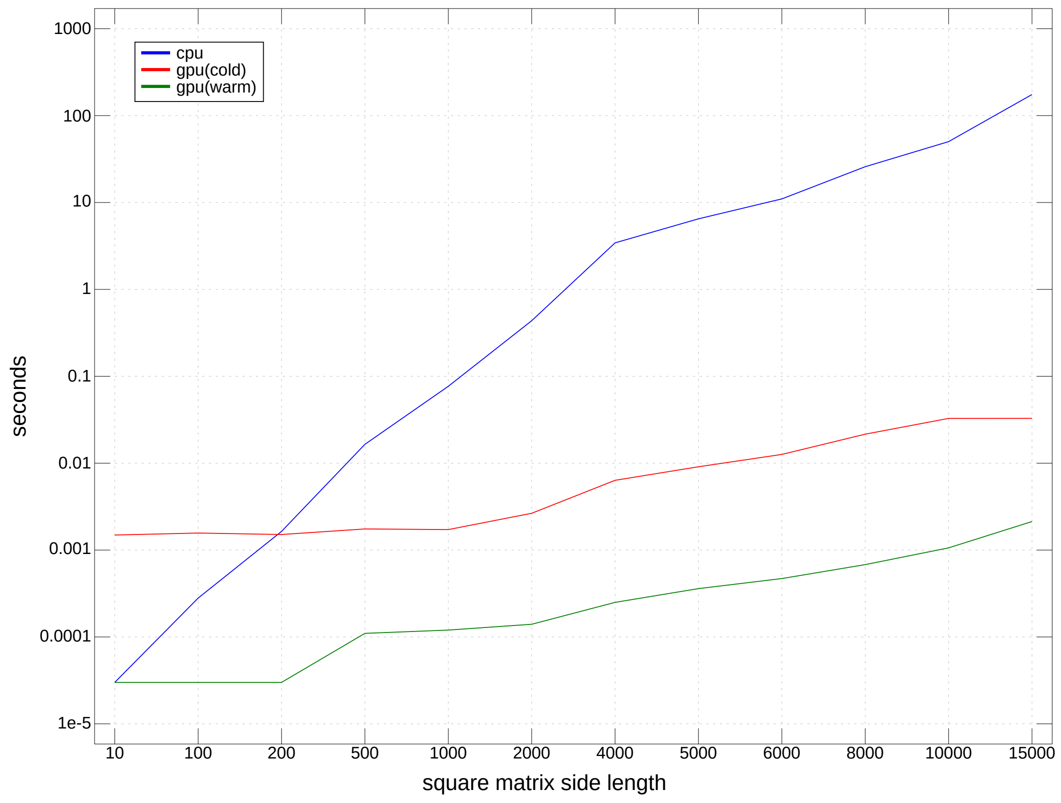

Figure 1: Matrix Multiplication Benchmark

One important caveat is that this does *not* consider the time required to generate the original data, nor does it account for transferring the data to or from the host or device. That said, this clearly shows that the GPU is faster for matrixes larger than 200x200 double-precision floating point data.

Conclusion

These measurements are misleading, because they do not account for data transfer latency between CPU and GPU. In the real world, this is a necessary step, but it adds significant latency. A better measurement would include this latency, and would likely narrow the gap by making the GPU effectively worse than is currently shown.

However, as a baseline, this project successfully shows the minimum data size needed to get GPU benefits, and also shows scaling effect of data size on CPU versus GPU. Because of the inherent latency of copying data from the GPU back to the host CPU, this method does not offer a universal improvement, but rather one that depends on the size of the data.

Further study is needed to (a) fix the segfault issue, (b) measure the impact of memory latency on overall performance, and (c) automatically determine the data size “cross over point” for a given system.

As a final remark, this report compares a recent, high-performance CPU with a much older, low-end GPU. But because the J interpreter only uses 1 of the available CPU cores, another way of thinking about this is in terms of single-core versus many-core performance. Even though the CPU is faster, the parallelism of the GPU is able to get better performance for larger data sizes.

Amdahl’s law is a sobering reminder that parallel hardware speedups are disproportionately hindered by sequential code, so inherently-parallel languages like J provide a practical path toward making better use of existing parallel hardawre.

Appendix: System Specifications

Host (CPU)

- 64GB DDR4
- Intel(R) Core(TM) i9-7980XE CPU @ 2.60GHz
- L1d cache: 576 KiB
- L1i cache: 576 KiB
- L2 cache: 18 MiB
- L3 cache: 24.8 MiB
- Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 s sse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cp uid_fault epb cat_l3 cdp_l3 invpcid_single pti ssbd mba ibrs ibpb stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd avx512bw avx512vl xsaveopt xsav ec xgetbv1 xsaves cqm_llc cqm_occup_llc cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts md_clear flush_l1d

GPU: Nvidia GeForce GTX 745 (rev a2)

- 384 cores
- 4GB DDR3
- 128 bit bus
- 24.79 GFLOPS (theoretical double-precision performance)

Software

- ArrayFire version: `v3.8.0`
- J version: `j902/j64avx2/linux` (compiled with `avx2` support)
- Operating System: `Ubuntu 20.04` (Linux)

Appendix: Code

To populate the table from the Results section, the following code was run multiple times with different arguments for `N`. Results were collected manually and added to the "Timing Results" table.

```
NB. matmul using GPU backend via ArrayFire
NB. requires nvidia gpu with cuda
LIB =: '/opt/arrayfire/lib64/libafcuda.so.3 '
chk =: {{ if. 0<>{.y do. exit echo 'ERROR',":y else. 1{:: y end. }}
af_get_data_ptr    =: LIB,'af_get_data_ptr i i i'
af_iota            =: LIB,'af_iota i * i *l i *l i'
af_matmul          =: LIB,'af_matmul i *x x x i i'
af_sync            =: LIB,'af_sync i i'
f64 =: 2  NB. 2 = ArrayFire type code for double precision float

gpu_time =: {{
 shp =. 2#y
 n_bytes =. 8**/shp
  NB. enough space to do all ops on GPU? (max 4GB, 3.5GB to be safe)
 assert. 3.5>1e9%~n_bytes
 h1 =. mema n_bytes  NB. (host pointer)
 i =. chk af_iota cd (,h1);2;shp;1;(,1);f64

 NB. matmul happens here
 t =. timex 'o1 =. chk af_matmul cd (,h1);({.i);({.i);0;0'
 chk af_sync cd _1
 echo 0j5":t  NB. print timing information to screen

 NB. FIXME - the following code segfaults:
 NB. - cold start segfaults with N=128 or higher
 NB. - warm start segfaults with N=1500 or higher

 NB. copy to host, free memory
 NB. h2 =. mema 8**/shp  NB. allocate CPU memory space for result
 NB. af_get_data_ptr cd h2;{.o1  NB. transfer from device to host (segfaults)
 NB. r=.shp$memr h2,0,(*/shp),8  NB. read memory
 NB. r;t [memf"0 h1,h2  NB. return result and timing information
}}

N =: 10 100 200 500 1000 2000 4000 5000 6000 8000 10000 15000

NB. call this script with one N at a time, note result
NB. gpu_time 1  NB. uncomment to "warm up" ArrayFire
gpu_time "._1{::ARGV

NB. J matmul ("CPU" in report)
```

```
cpu_result =: {{ r;(timex'r=.a+/ .*a=.1+i.2#y') }}
cputest =: {{ 1{::cpu_result y }}
```