ArrayFire: a GPU acceleration platform

James Malcolm, Pavan Yalamanchili Chris McClanahan, Vishwanath Venugopalakrishnan, Krunal Patel, John Melonakos

AccelerEyes, 75 5th Street NW STE 204, Atlanta, GA 30308

ABSTRACT

ArrayFire is a GPU matrix library for the rapid development of general purpose GPU (GPGPU) computing applications within C, C++, Fortran, and Python. ArrayFire contains a simple API and provides full GPU compute capability on CUDA and OpenCL capable devices. ArrayFire provides thousands of GPU-tuned functions including linear algebra, convolutions, reductions, and FFTs as well as signal, image, statistics, and graphics libraries. We will further describe how ArrayFire enables development of GPU computing applications and highlight some of its key functionality using examples of how it works in real code.*

Keywords: CUDA, GPU, GPGPU, ArrayFire, OpenCL

1. INTRODUCTION

ArrayFire is a software platform developed at AccelerEyes[1] that allows users and programmers to rapidly develop data-parallel programs in C, C++, Fortran, and Python. ArrayFire provides a simple, high-level matrix abstraction over low-level GPU APIs such as CUDA,[7] OpenCL, and OpenGL along with thousands of GPU-tuned functions to allow users in science, engineering, and finance to take full advantage of GPU hardware. The combination of an easy-to-use array interface, automatic memory management, on-the-fly compilation, parallel GPU `for`-loop construction, and interactive hardware-accelerated graphics library, make ArrayFire well suited for rapid prototyping of data-parallel algorithms as well as building deployed applications.

Over the last decade GPUs have proliferated both consumer and developer computers. Despite the growing list of success stories, GPU software development adoption has had a slow rise. The slowness of the rise is attributable to the difficulty in programming GPUs.[2]

Cg, GLSL, HLSL, and Brook marked the beginning of stream programming, a precursor to general purpose GPU programming, where computation is mapped onto the graphics pipeline and consequently subject to various constraints. Following on the heels of these technologies, CUDA and OpenCL introduced a more generally programmable software architecture, easier than stream programming but harder to program than standard single-threaded C/C++. However, even with these advances there remains a steep learning curve for the average programmer to reach success with CUDA, OpenCL, etc.

Several previous companies also made attempts to reach this goal. One of the first such companies, PeakStream (now at Google) built a C/C++ runtime and library of functions providing a rich toolset to GPU development. RapidMind (now at Intel) built a flexible middle-layer supporting various frontend languages and backend hardware targets. Both of these sought to bridge the gap between the hardware and developers. ArrayFire, the latest platform for GPU development attempts to bridge this gap by porting high-level functions onto the underlying hardware. The goal is to deliver programmability and portability without sacrificing performance.

2. ARRAYFIRE

2.1 Language support: C++, Fortran, Python

ArrayFire revolves around a single matrix object (array) which can contain floating point values (single- and double-precision), real or complex values, and boolean data. Heres a classic example showing Monte-Carlo estimation of Pi in the vectorized ArrayFire notation: create two vectors of samples (x,y) and count how many fell inside one quadrant of the unit circle.

*The project was supported by Grant Number R44MH088087 from the National Institute Of Mental Health. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institute Of Mental Health or the National Institutes of Health.

Modeling and Simulation for Defense Systems and Applications VII, edited by Eric J. Kelmelis,
Proc. of SPIE Vol. 8403, 84030A · © 2012 SPIE · CCC code: 0277-786X/12/$18 · doi: 10.1117/12.921122

```
int n = 20e6;                              // use 20m samples
array x = randu(n,1), y = randu(n,1);   // uniform random (x,y) samples
double pi = 4 * sum<float>(hypot(x,y) < 1) / n;
```

Figure 1. Using ArrayFire to estimate the value of Pi via random sampling

ArrayFire arrays are multidimensional, can be generated via simple matrix creation functions such as (ones, randu, etc.), and can be manipulated with arithmetic and functions. Additionally, ArrayFire provides a parallel FOR-loop implementation, gfor, which executes arbitrarily many instances of independent routines in a data-parallel fashion. More details and examples can be found in the online reference[†].

2.2 Example: Stereo Disparity

```
array left = get_image(left), right = get_image(right);
array kernel= ones(5,5), sad = zeros(5,5);
gfor (i = 0; i < nshifts; ++i) {
    array shifted = shift(right, 0, i);
    array diff = abs(left- shifted);
    sad(span,span,i) = convolve(diff, kernel);
}
array disparity = min(sad, 3);
```

Figure 2. Using ArrayFire to compute the horizontal disparity between two images using plane sweeping and a 5-by-5 correlation window. All disparity computations and the final minimum are executed in parallel while the source code is kept clean and simple without diving into low-level GPU APIs.

In the simple example depicted above (Figure 2), we show the computation of stereo disparity between two images. This code runs on all devices that ArrayFire supports and remains concise without the need for low-level GPU programming APIs such as CUDA or OpenCL. ArrayFire computes the horizontal disparity between two images using plane-sweeping and a 5-by-5 correlation window. All disparity computations and the final minimum are executed in parallel while the source code is kept clean and simple without diving into low-level GPU APIs.

The program is described line by line. The declaration of left, right, and sad indicate that we will be working with 4-byte floating point arrays and that left and right are imagery that we have acquired. We assume that all arrays are the same size. We also create kernel, a single precision sliding 5-by-5 window.

The second line indicates via the gfor keyword that we will be computing the following block nshifts + 1 times from 0 to nshifts in parallel. All subsequent commands are still executed only once, but instead of describing one computation, each command invokes nshifts + 1 computations. Thus, the next command shifts the right image 0 pixels, 1 pixel, 2 pixels, etc. and stores the results (if compiled and executed immediately) as a handle in the variable shifted. The remaining lines compute the sum of the absolute difference between the two images (multiple times as is the case with the first line) in a sliding 5-by-5 window. Finally, the minimum over all computed sums of absolute differences (SADs) is taken. Note that during execution ArrayFire may be compiling commands together lazily and all the previously discussed operations may be merged into one batch of execution of the GPU and dispatched.

3. LAZY EVALUATION

ArrayFire uses a technique called lazy evaluation (also known as Just-In-Time computation, or JIT) in order to maximize the speedup available to the users programs. This patent-pending technology both generates computational device code on-the-fly and optimizes memory transfers to maximize throughput while maintaining flexibility.

The simplest way to perform a computation with the GPU would be to perform each operation on the GPU as the user enters/runs their code. This would work, and might offer some speedup, but not nearly as much

[†]http://www.accelereyes.com/arrayfire/c

as is possible. Every time ArrayFire ports to the GPU, there is a cost in time. In addition to the actual time for computing values, there is also time spent communicating instructions and transferring data to and from the GPU. ArrayFire keeps track of the computations the user is performing. Rather than performing the computation immediately, ArrayFire saves the formula and input to compute a given value for faster execution the second time around.

Consider the following bit of code:

```
array I = randu(128);
array A = cos(I);
array B = pow(A, 2);
array C = 1 - B;
```

Behind the scenes, ArrayFire will build up these partial expressions, but waits to actually evaluate (execute) any device code:

```
array I = randu(128);
array A = cos(I);
array B = pow(cos(I), 2); // builds on A
array C = 1 - pow(cos(I), 2); // builds on both A and B
```

It doesn't matter if `I` is changed before `C` is evaluated, ArrayFire keeps track of the original input, so you will still get the correct answer. Once you need the value of `C`, it will be computed automatically. This saves time versus having to perform kernel executions on the GPU at each step because the entire computation of `C` can be batched as a single kernel and not as each operation is encountered. ArrayFire uses this information to further optimize the bigger picture version of your computation that would otherwise be impossible when performing each operation one-at-a-time. The other benefit from this is that you can run code to compute a value which you never end up actually using; ArrayFire can skip the computation all together. If the memory needed to compute `C` or the complexity of the computation starts to get extreme, ArrayFire employs techniques to pre-compute parts of `C` on its own.

4. INDEXING

ArrayFire provides several flexible and efficient methods for subscripted indexing into arrays. Just like standard C/C++, Python, and Fortran, subscripting in ArrayFire is zero-based, i.e. `A(0)` is the first element. Indexing using ArrayFire can be done with mixtures of:

- integer scalars
- `seq` representing linear sequence
- `end` representing last element of a dimension
- `span` representing entire dimension
- `row(i)` or `col(i)` specifying a single row or column
- `rows(first,last)` or `cols(first,last)` specifying a span of rows or columns

Examples of subscripted array indexing:

```
array A = randu(3,3);
array a1 = A(0);   // first element
array a2 = A(0,1); // first row, second column

A(end);   // last element
```

```
A(-1);    // also last element
A(end-1); // second-to-last element

A(1,span);       // second row
A.row(end);      // last row
A.cols(1,end);   // all but first column

float b_host[] = {0,1,2,3,4,5,6,7,8,9};
array b(b_host, 10, dim4(1,10));
b(seq(3));       //  {0,1,2}
b(seq(1,7));     //  {1,2,3,4,5,6,7}
b(seq(1,2,7));   // {1,3,4,7}
b(seq(0,2,end)); // {0,2,4,6,8}
```

Indexing of arrays is important when considering performance. Because memory movement and rearrangement operations can be expensive, it is easy to let subscripting in one area of your program eat away at the speedups obtained from fast number crunching in another area. Avoid subscripting to access scalar values; instead, subscript to reference an entire vector or matrix of elements. Data is stored in column major order (same as Fortran), so its preferable to index contiguous spans of memory, e.g. referencing an entire column (`A(span,2)`), rather than scattered accesses, e.g. referencing a row which skips memory between elements (`A(2,span)`). More examples can be found online[‡].

5. GRAPHICS

General purpose computing aside, the GPU is first and foremost a graphics processing machine. Reflecting this reality, ArrayFire ships with a graphics package both with free and commercial versions. Unique to ArrayFire, its graphics engine is designed from the ground-up with GPGPU computing in mind: all primitives draw from memory on the card. Therefore, compute and visualization are both handled by the GPU without any memory transfers between the host CPU and the device. The GPU is left to do what it does best: parallel processing and graphics output while the CPU is left to serial computation and direction of the graphics backend.

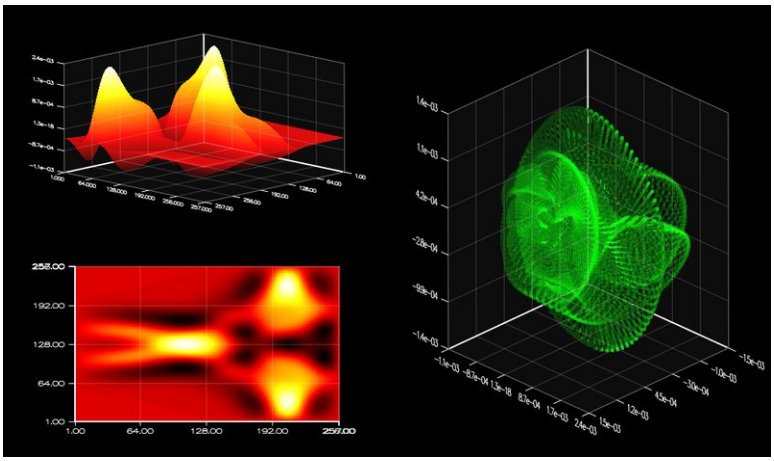


Figure 3. Shallow water simulation (`examples/swe`) with subplots showing various views of the field.

Several basic plotting primitives are included with the graphics package including:

- 2D and 3D line plotting

[‡]`http://accelereyes.com/arrayfire/c/page_quickref.htm`

- Volume rendering
- Overlaying plots for comparison analysis
- Image visualization
- and many more

Working with the graphics backend is meant to be easy so that you can concentrate on your problem instead of OpenGL, multiple threads, etc. For example, drawing Figure 3 was easily accomplished with the following C++ code:

```
subfigure(2,2,1); plot3d(A);
subfigure(2,2,3); imgplot(A);
subfigure(2,1,2); points(dx, dy, dz);
```

6. GFOR

ArrayFire provides the only GPU-enabled data-parallel loop: `gfor`. Use `gfor` in your code in place of standard `for`-loops to batch process data parallel operations. For example, the following `for`-loop calculates several matrix multiplications serially:

```
for (int i = 0; i < 2; i++) {
     C(span,span,i) = A(span,span,i) * B;
}
```

The same three matrix multiplications can be carried out in one pass instead of three by utilizing a `gfor`-loop,

```
gfor (array i, 2) {
    C(span,span,i) = A(span,span,i) * B;
}
```

Whereas the former loop serially computes each matrix multiplication, the latter loop computes all matrix multiplications in one pass. Similarly, `gfor` can be used for other such embarrassingly parallel codes in a straightforward fashion. A good way of thinking about gfor is to consider it as syntactic sugar: `gfor` serves as an iterative style of writing otherwise vectorized algorithms.

7. MULTI GPU

ArrayFire can scale from a single GPU to multi-GPU configuration using just a few function calls. The GPU to be used for computation is selected by calling `deviceset()`. This gives the count of GPUs licensed to run ArrayFire. To compute on multiple GPUs simultaneously, we partition input data (if necessary) and move these data chunks onto each GPU. This is done by calling deviceset() in a loop selecting each GPU in the system and constructing array objects.

```
int ngpus = devicecout(); // Total GPUs in the system
float *h_in = new float[n*n]; // Host buffer containing input data

// Loop over gpus and move partitioned data onto each GPU
for (int i = 0; i < ngpus; i++) {
     deviceset(i); // Switch to specified GPU
     Input[i] = array(n, n/ngpus, h_in + (n*n/ngpus * i));
}
```

This is followed by another loop that calls the FFT to operate on these individual chunks of data, all running simultaneously on each GPU in the system.

```
for (int i = 0; i < ngpus; i++) {
     deviceset(i);
     Output[i] = fft(Input[i]); // All GPUs compute FFT in parallel
}
```

ArrayFire also provides a synchronization function `sync()` that blocks the host thread until all devices finish all queued up function calls. To get data back on the host, we run another loop and move back the data to host buffer.

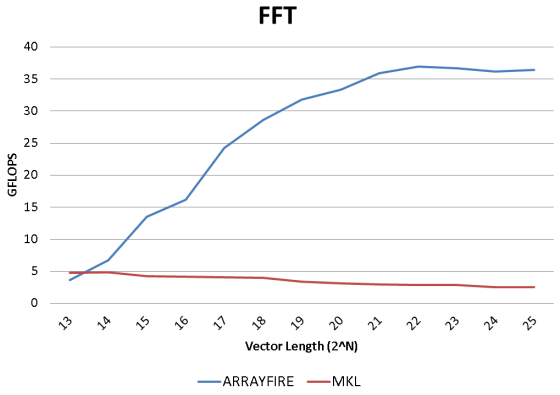
```
sync() // Block host thread until all GPUs finish
for (int i = 0; i < ngpus; i++) {
     deviceset(i);
     h_out[i] = Output[i].host<float>(); // Copy results back to host buffer
}
```

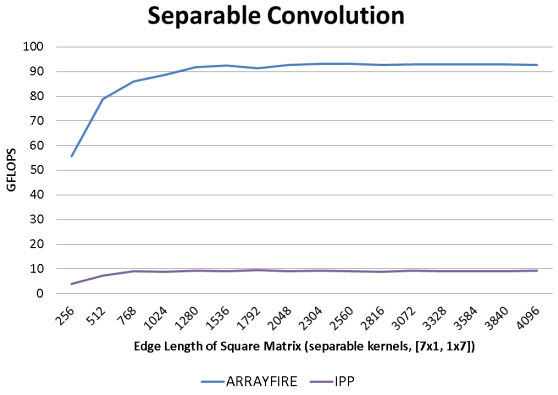
ArrayFire with its user-friendly API makes it very easy to scale from single GPU to multiple GPUs as shown above.

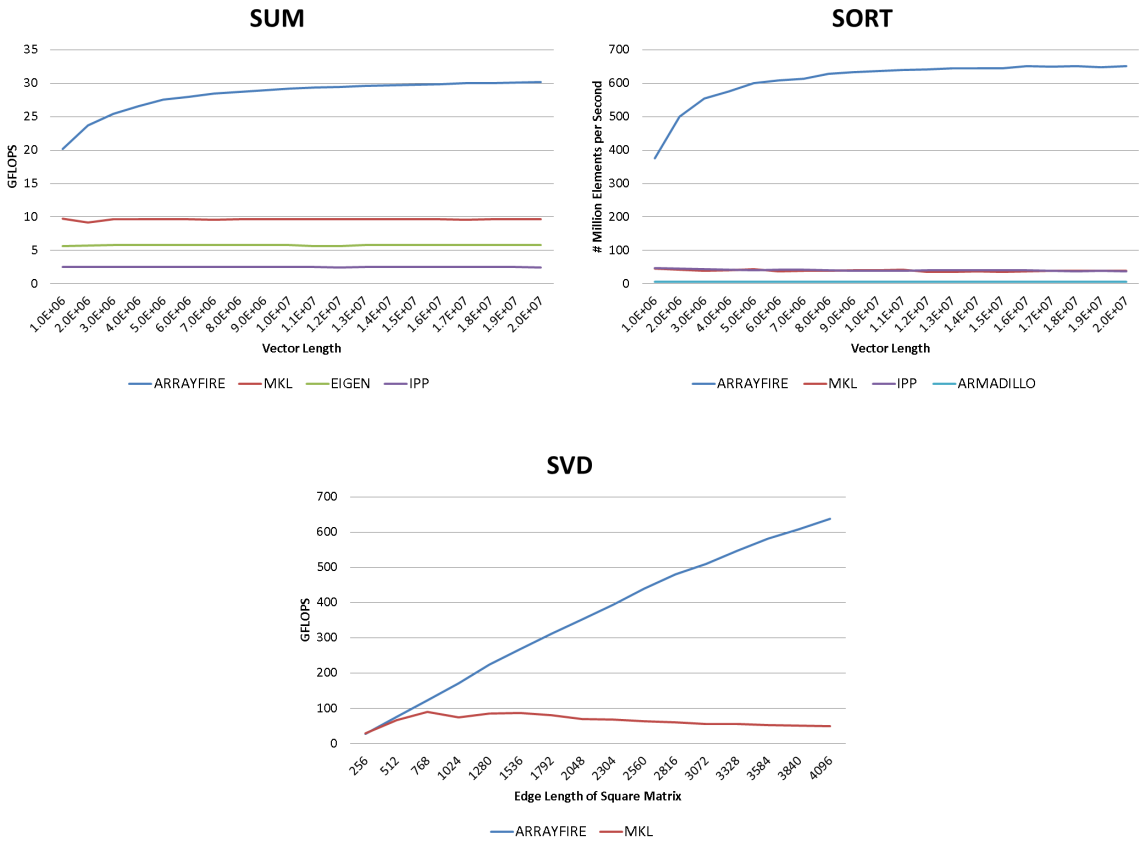
8. BENCHMARKS

ArrayFire is fast. It uses the parallel processing power of the GPUs to provide significant performance benefits over other commercial and freely available libraries. We present sample benchmarks from the wide range of functionality that ArrayFire supports. All trend lines indicate throughput: higher trend lines mean better performance. All benchmarks have been run using the following specifications:

- Hardware: Intel i7 920, Tesla C2050
- Software: ArrayFire 1.0, Intels MKL 10.3, Intels IPP 7.0, Eigen 3.0.







9. CONCLUSION

The emergence of new software tools and cost-effective GPU-based computers are ushering in a new era for scientists, engineers and analysts working on technical problems. The platform, ArrayFire, described in this paper offers a more productive platform for non-computer scientists looking for ways to leverage the computational capabilities of GPUs. ArrayFire is a fast GPU matrix library for development of general purpose GPU computing applications within C, C++, Fortran, and Python. We presented ArrayFire's simple API and described how it enables rapid development of GPU computing applications. Key elements of ArrayFires functionality were described along with examples of how it works in real code. We then closed with benchmarking results showing that ArrayFire provides a fast approach to GPU computing.

REFERENCES

[1] AccelerEyes. Addr.: 800 W Peachtree St NW, Atlanta, GA 30308, USA. URL: `http://www.accelereyes.com`.

[2] Andy Webb: "MATLABs Racing Jacket". Automated Trader, vol. 16, no. 1, pp 54–61. 2010.

[3] AccelerEyes: "Jacket v1.7.1: Getting Started Guide". URL: `http://www.accelereyes.com/services/documentation`.

[4] Mark Harris, "Optimizing CUDA". *SuperComputing 2007 Tutorial*, Reno, NV, USA. November 2007.

[5] V. Volkov and J. W. Demmel,"Benchmarking GPUs to Tune Dense Linear Algebra". *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pp. 1-11, Austin, Texas, USA, 2008.

[6] Torben's Corner. Available at the Wiki of AccelerEyes at: URL: `http://www.accelereyes.com/wiki/index.php?title=Torben's_Corner`.

[7] nVidia, "CUDA programming guide 1.1," Available at http://developer.download.nvidia.com/compute/cuda/1_1/NVIDIA_CUDA_Programming_Guide_1.1.pdf
[8] nVidia, "CUDA CUBLAS Library 1.1," Available at http://developer.download.nvidia.com/compute/cuda/1_1/CUBLAS_Library_1.1.pdf
[9] nVidia, "CUDA CUFFT Library 1.1," Available at http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf
[10] nVidia, "PTX: Parallel Thread Execution ISA Version 1.1", Available at http://www.nvidia.com/object/cuda_develop.html
[11] Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., and Hanrahan, P., "Brook for GPUs: Stream computing on graphics hardware." *Transactions on Graphics and Visualization* vol. 23, no. 3, Aug. 2004.
[12] Tarditi, D., Puri, S., and Oglesby, J.. Accelerator: Using data parallelism to program GPUs for General-Purpose uses. In *International Conference on Architectural Support for Programming Languages and Operating Systems* (2006)
[13] McCool, M. and Toit, S.D., *Metaprogramming GPUs with Sh*. A K Peters, 2004.
[14] Mark, W.R., Glanville, R.S., Akeley, K., and Kilgard, M.J. "Cg: A system for programming graphics in a c-like language." *Transactions on Graphics and Visualization* vol. 22, no. 3, pp. 896-907, 2003
[15] Matt Pharr, ed., *GPU Gems 2*, Addison-Wesley, 2005.
[16] Hubert Nguyen, ed., *GPU Gems 3*, Addison-Wesley, 2007.
[17] D. Göddeke, *GPGPU Basic Math Tutorial*, tech. report 300, Fachbereic Mathematik, Universitt Dortmund, 2005