CUDA Accelerated Image Processing Libraries (S3559)

A look at ArrayFire and NPP

James Malcolm
AccelerEyes
Image Processing (IP)
Algorithms

- morphology - erode, dilate, open, close
- convolution - smoothing, sharpening
- filtering - mean, median
- signal processing often included
Libraries

CPU
- Intel Performance Primitives (IPP)
- OpenCV

GPU
- ArrayFire
- NVIDIA Perf. Primitives (NPP)
- OpenCV (in development)
NPP by NVIDIA

Overview, History, Features, Examples

Slides courtesy of Frank Jargstorff @ NVIDIA
NPP from NVIDIA

- Library of high-performance C functions (primitives) for image and signal processing.
- Running on NVIDIA’s CUDA Architecture GPUs.
- Implemented on top of the CUDA Runtime Interface.
- Modeled after Intel’s Integrated Performance Primitives (IPP).
Library vs. Framework

- NPP is a library—not a framework.
  - NPP strives to impose the least amount of policy with respect to how the primitives can be used and how image (and signal) data is passed to the functions and how it is stored.
  - NPP never allocates or frees memory by itself.
  - Keeps the number of data-structures used throughout the API to a bare minimum.
  - No inversion-of-control (i.e. no call-back functions).

- Why this matters:
  - This design minimizes the amount of code changes needed to integrate NPP into existing projects.
Memory Management I

- NPP leaves all memory management to the developer:
  - Allocating memory for image and signal storage on the host (CPU) as well as on the device (GPU).
  - Moving image and signal data from host to device for processing and moving results back from device to host.
- Large data (images, signals, filter coefficients) are passed to and from NPP via pointers. E.g.:

```c
NppStatus
nppiAddC_8u_C1RSfs(const Npp8u * pSrc, int nSrcStep, const Npp8u nConstant,
Npp8u * pDst, int nDstStep, NppiSize oSizeROI,
    int nScaleFactor)
```

- All pointers in the NPP API are device pointers, unless explicitly stated otherwise.
Region-of-Interest (ROI)

- Majority of NPP image-processing functions operate on ROIs.
- In most cases, ROI given simply as (width, height) pair.
- First pixel in ROI is pointed to by the image-data pointers passed to the function (pDst, pSrc, etc.)
- In order to move the ROI to start at a pixel other than (0,0) user has to perform pointer-arithmetic to compute the pointer address of the first pixel \((x, y)\) in the ROI (assuming 8u_C1 image):
  
  \[
pSrc = pImage + nImageStep \times y + x
\]
Region-of-Interest

pImage [pointer to pixel (0,0)]

LineStep (in bytes)

pDst [pointer to first pixel in ROI]

Image Data

ROI (width, height)

Line Padding

Image Height

Image Width
Naming Conventions

- **Data types descriptors:**
  - 8u = 8-bit unsigned integer. Similarly: 16u
  - 16s = 16-bit signed integer. Similarly : 8s, 32s, 64s
  - 32f = 32-bit floating-point (IEEE). Similarly : 64f
  - 16sc = Complex value consisting of two 16s values. Similarly: 32sc, 32fc, 64fc.

- **Actual NPP type definitions** are prefixed with “Npp”. Examples: Npp8u, Npp32f, Npp64sc.
Memory Management II

- Several NPP primitives require temporary storage space in order to perform their duties.
- Management of these temporary memory buffers is the developer’s responsibility:
  - Each function requiring temporary buffers has a companion function that computes the amount of memory (in bytes) required by the primitive.
  - These “ComputeBufferSize” functions often require some or all of the primitive’s parameters to be passed to them, since the temporary memory buffer size may be parameter dependent.
- Temporary memory is unstructured. The primitive makes no assumptions about buffer’s content.
Temporary Buffer Example

// computes the minimum of all pixels in the ROI
NppStatus
nppiMin_8u_C1R(const Npp8u * pSrc, int nSrcStep, NppiSize oROI, // input image & ROI
Npp8u * pDeviceBuffer, // temp buffer
Npp8u * pMin) // result

// computes the necessary temporary buffer size for nppiMin_8u_C1R
NppStatus
nppiMinGetBufferSize_8u_C1R(NppiSize oSizeROI, // temp buffer size depends on ROI
int * hpBufferSize) // result
NPP Primitive Naming

nppiAddC_8u_C1RSfs

- C1 = Single Channel Image
- R = Region-of-Interest required
- Sfs = Scale-factor for integer-result scaling.

Primitive name. This one adds a constant to each pixel value.

This primitive operates on 8-bit unsigned pixel data.

All primitives are prefixed with “npp”. An additional “i” indicates an image-processing primitive. The “npps” prefix implies a signal-processing primitive.
Functionality

- Most NPP image-processing primitives support a range of data types and color channels:
  - Npp8u, Npp16u/s, Npp32s, Npp32f, etc.
  - 1, 3, and 4 channel images.
NPP Features

- arithmetic, statistics
- erode, dilate, interpolate
- affine/perspective warp
- rotate, resize, transpose, channel swap
- colorspace conversions, histograms
- convolution, smoothing
- graph cuts, DCT

more online: https://developer.nvidia.com/npp
ArrayFire
World's fastest, largest GPU library.
C, C++, Fortran
CUDA and OpenCL
v1.9
accelereyes.com/arrayfire
Licensing

Free
  Full performance, single GPU, internet connection to license server

Pro
  Multi-GPU, full linear algebra, offline
estimate $\pi$
estimate \pi

```c
#include <stdio.h>
#include <arrayfire.h>
using namespace af;
int main() {
    // 20 million random samples
    int n = 20e6;
    array x = randu(n,1), y = randu(n,1);
    // how many fell inside unit circle?
    float pi = 4 * sum<float>(sqrt(x*x+y*y)<1) / n;
    printf("pi = %g\n", pi);
    return 0;
}
```
```plaintext
array x = randu(n, f32);
array y = randu(n, f64);
array z = rand(n, u32);
```
ND Support

- Vectors
- Matrices
- Volume
Subscripting

ArrayFire Keywords: end, span

A(1,1)  A(1,span)

A(end,1)  A(end,span)

A(span,span,2)
generate arrays

constant(0,3) // 3-by-1 column of zeros, single-precision constant
(1,3,2,f64) // 3-by-2 matrix, double-precision
randu(1,8) // row vector (1x8) of random values (uniform)
randn(2,2) // square matrix (2x2) random values (normal)
identity(3,3) // 3-by-3 identity
randu(5,7,c32) // complex random values
create arrays from CPU data

float hA[] = {0,1,2,3,4,5};
array A(2,3,hA); // 2x3 matrix, single-precision
print(A);
// A = [ 0 2 4 ]      Note: Fortran storage order
//     [ 1 3 5 ]
array R = randu(3,3);
array C = constant(1,3,3) + complex(sin(R)); // C is c32

// rescale complex values to unit circle
array a = randn(5,c32);
print(a / abs(a));
// calculate L-2 norm of every column
sqrt(sum(pow(X, 2)))  // norm of every column vector
sqrt(sum(pow(X, 2), 0))  // ..same
sqrt(sum(pow(X, 2), 1))  // norm of every row vector
array A = randu(3,3);
array a1 = A(0);       // first element
array a2 = A(0,1);     // first row, second column
A(1,span);             // second row
A.row(end);            // last row
A.cols(1,end);         // all but first column

float b_ptr[] = {0,1,2,3,4,5,6,7,8,9};
array b(1,10,b_ptr);
b(seq(3));             // {0,1,2}
b(seq(1,7));           // {1,2,3,4,5,6,7}
b(seq(1,2,7));         // {1,3,5,7}
b(seq(0,2,end));       // {0,2,4,6,8}
// setting entries to a constant
A(span) = 4;    // fill entire array
A.row(0) = -1; // first row
A(seq(3)) = 3.1415; // first three elements

// copy in another matrix
array B = constant(1,4,4,f64);
B.row(0) = randu(1,4,f32); // set row (upcast)

// use arrays to reference into other arrays
float h_inds[] = {0, 4, 2, 1}; // zero-based
array inds(1,4,h_inds);
B(inds) = randu(4,1); // set to random
// matrix factorization
array L, U;
lu(L, U, randu(n,n));

// linear systems: A x = b
array A = randu(n,n), b = randu(n,1);
array x = solve(A,b);
C, C++, and Fortran

// C
float *d_x;
cudaMalloc(&d_x, 4*4*sizeof(float));
af_randu_S(d_x, 4*4);

! Fortran
type(array)
dst = randu(4,4)
Compute + Render

asynchronous
non-blocking
throttled at 35 Hz
#include <arrayfire.h>
using namespace af;

int main() {
    // Infinite number of random 3d surfaces
    const int n = 256;
    while (1) {
        array x = randu(n, n);
        // 3d surface plot
        surface(x);
    }
    return 0;
}
Graphics Commands

non-blocking primitives

- `surface` surface plotting (2d data)
- `plot2` line plotting
- `image` intensity image visualization (grayscale, color)
- `arrows` vector fields
- `plot3` scatter plot
- `volume` volume rendering for 3d data
Graphics Commands

utility commands

fig("sub", nx, ny, i);
fig("color", table);
fig("clear");
fig("draw"); (blocking)
fig(); new
fig("title", str);
fig("close");
GFOR

data parallel for-loops
gfor

- GPUs are great for big arrays
- But often you have a lot of little problems
- Solution: treat them like independent tiles and launch kernels over all at once
Example: Matrix Multiply

Serial matrix multiplications (3 kernel launches)

for (i = 0; i < 3; i++)
    C(span, span, i) = A(span, span, i) * B;

iteration i = 1

\[
\begin{align*}
\text{C}(,,1) & = \begin{bmatrix}
\end{bmatrix} \\
\text{A}(,,1) & = \begin{bmatrix}
\end{bmatrix} \\
\text{B} & = \begin{bmatrix}
\end{bmatrix}
\end{align*}
\]
Example: Matrix Multiply

Serial matrix multiplications (3 kernel launches)

```c
for (i = 0; i < 3; i++)
    C(span,span,i) = A(span,span,i) * B;
```

**iteration i = 1**

```
C(,,1) = A(,,1) * B
```

**iteration i = 2**

```
C(,,2) = A(,,2) * B
```
Example: Matrix Multiply

Serial matrix multiplications (3 kernel launches)

```c
for (i = 0; i < 3; i++)
    C(span, span, i) = A(span, span, i) * B;
```

*iteration i = 1*

\[
C(\text{span}, 1) = A(\text{span}, 1) \times B
\]

*iteration i = 2*

\[
C(\text{span}, 2) = A(\text{span}, 2) \times B
\]

*iteration i = 3*

\[
C(\text{span}, 3) = A(\text{span}, 3) \times B
\]
GFOR: data parallel for-loop

Good for lots of small problems

Serial matrix multiplications (3 kernel launches)

```c
for (i = 0; i < 3; i++)
    C(span, span, i) = A(span, span, i) * B;
```

Parallel matrix multiplications (1 kernel launch)

```c
gfor (array i, 3)
    C(span, span, i) = A(span, span, i) * B;
```

Example: Matrix Multiply

Parallel matrix multiplications (1 kernel launch)

\[
gfor \ (\text{array } i, \ 3) \\
C(\text{span}, \text{span}, i) = A(\text{span}, \text{span}, i) \times B;
\]

\text{simultaneous iterations } i = 1:3
Example: Matrix Multiply

Parallel matrix multiplications (1 kernel launch)

\[
gfor \ (\text{array } i, \ 3) \\
C(\text{span},\text{span},i) = A(\text{span},\text{span},i) \times B;
\]

Think of GFOR as compiling 1 stacked kernel with all iterations.
simultaneous iterations \( i = 1:3 \)

\[
\begin{align*}
C(\text{,,1:3}) &= \ast \ast \ast \\
A(\text{,,1:3}) &= \ast \ast \ast \\
B &= \ast \ast \ast
\end{align*}
\]
Example: Matrix Multiply

Parallel matrix multiplications (1 kernel launch)

gfor (array i, 3)
    C(span,span,i) = A(span,span,i) * B;

Think of GFOR as compiling 1 stacked kernel with all iterations.
simultaneous iterations i = 1:3
Example: Summing over Columns

Think of GFOR as “syntactic sugar” to write vectorized code in an iterative style.

Three passes to sum all columns of B

```plaintext
for (i = 0; i < 3; i++)
    A(i) = sum(B(span,i));
```

One pass to sum all columns of B

```plaintext
gfor (array i, 3)
    A(i) = sum(B(span,i));
```

Both equivalent to “\texttt{sum(B)}”, but latter is faster (more explicitly written)
array *y = new array[n];
for (int i = 0; i < n; ++i) {
    deviceset(i); // change GPUs
    array x = randu(5,5); // add work to GPU’s queue
    y[i] = fft(x); // more work in queue
}

// all GPUs are now computing simultaneously
Hundreds of Functions...

- **reductions**
  - sum, min, max, count, prod
  - vectors, columns, rows, etc

- **convolutions**
  - 2D, 3D, ND

- **interpolate & scale**
  - vectors, matrices rescaling

- **FFT's**
  - 2D, 3D, ND

- **image processing**
  - filter, rotate, erode, dilate, morph, resize, rgb2gray, histograms

- **dense linear algebra**
  - LU, QR, Cholesky, SVD, Eigenvalues, Inversion, Solvers, Determinant, Matrix Power

- **sorting**
  - along any dimension sort detection

and many more...
ArrayFire Image Processing

- arithmetic, statistics
- erode, dilate, interpolate
- rotate, resize, transpose, channel swap
- colorspace conversions
- histograms binning/equalization
- IIR/FIR filtering, convolution, smoothing
- bilateral, mean shift, median filtering

more online: [http://accelereyes.com/arrayfire/c](http://accelereyes.com/arrayfire/c)
Performance

ArrayFire v MKL, IPP, Eigen, ...
1D Convolution

MB/second vs. Signal Length $2^N$, with 10k element kernel length

ARRAYFIRE, MKL, IPP
2D convolution

GFLOPS

Edge Length of Square Matrix (25x25 kernel)

ARRAYFIRE

IPP
Easy To Maintain

Write your code once
Each new release improves the speed of your code
Tesla, Fermi, Kepler, ..

Behind the scenes...we update
Licensing

Free

Full performance, single GPU, internet connection to license server

Pro

Multi-GPU, full linear algebra, offline
Build vs Buy

ArrayFire eliminates hidden costs of software development.
Key factors in software choice:

- Performance
- Portability
- Community
- Programmability
- Scalability
- Interop
Performance vs. Programmability

- Faster
  - Raw CUDA
- Slower
  - SSE or AVX
  - Compiler Directives

Time-consuming vs. Easy-to-use
Example: local window operations

- lots of operations require fetching a local window
  - median filtering, smoothing, morphology, ...
  - often the dominant factor in performance
- Not all algorithms scale across all hardware

<table>
<thead>
<tr>
<th></th>
<th>Tesla C1060</th>
<th>Quadro 2000</th>
<th>Tesla C2075</th>
<th>Tesla K20c</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td>395 ms</td>
<td>599</td>
<td>244</td>
<td>102</td>
</tr>
<tr>
<td>method 1</td>
<td><strong>270</strong></td>
<td>703</td>
<td>241</td>
<td>103</td>
</tr>
<tr>
<td>method 2</td>
<td>699</td>
<td><strong>407</strong></td>
<td><strong>138</strong></td>
<td>116</td>
</tr>
<tr>
<td>textures</td>
<td>380</td>
<td>522</td>
<td>202</td>
<td><strong>98</strong></td>
</tr>
</tbody>
</table>
Portability

CUDA
Open source LLVM compiler

OpenCL
Runs on AMD, Intel, and NVIDIA
Provides CPU fallback
Scalability

Laptops … workstations … servers
   No code change

Multi-GPU
   CUDA, Directives: user managed, low-level
   ArrayFire: simple!

   for (int i = 0; i < ndevice; ++i) {
      deviceset(i);
      out[i] = fft(in[i]);
   }
Community

Topics on GPU forum activity

NVIDIA – 25,277 topics
AMD – 4,886
AccelerEyes – 1,533
PGI – 569
Interop

Integrate CUDA kernels, device pointers, ...

array x = randu(n);
float *dx = x.device<float>();
kernel<<<blk,thr>>>(n,dx);

Incrementally port critical sections of code
A case study:

NORTHROP GRUMMAN

UAV IMAGING
NGC Falls Church – UAV Imaging

Project: Ground vehicle detection
NGC Falls Church – UAV Imaging

Project: Ground vehicle detection (IRAD)
Baseline: 4.0 s (IPP enhanced CPU code)
Goal: 0.4 s (real-time)
Problem: Optimized CPU code 10X too slow. Too many small image patches of apparently not well-suited for GPUs.
NGC Falls Church – UAV Imaging

Solution #1: Use ArrayFire (and Jacket)  
Avoid low-level hassle with raw CUDA

Solution #2: Use AccelerEyes’ consulting  
Expert at GPU algorithms

Code Result: 0.4 second goal achieved

Project Result: Code port in <1 month
Examples

Image Processing with ArrayFire and NPP
Manipulation

Swap Channels

**Using NPP:**

```c
sFilename = "path_to_image"; // filename of source image
npp::ImageCPU_8u_C4 oHostSrc; // host image object for 4 channel source image
npp::loadImage(sFilename, oHostSrc); // load image from disk
npp::ImageNPP_8u_C4 oDeviceSrc(oHostSrc); // copy source host image to device
NppiSize oSizeROI = {oDeviceSrc.width(), oDeviceSrc.height()}; // set ROI
const int aDSTOrder[4] = [0,1,0,3] // set order to swap bChannel with rChannel
NppStatus errorStatus = nppiSwapChannels_8u_C4IR
    (oDeviceSrc.data(), oDeviceSrc.pitch(), oSizeROI, aDSTOrder); // swap channels
```
Using ArrayFire:

```cpp
array tmp = img(span, span, 0);  // save the R channel
img(span, span, 0) = img(span, span, 2);  // R channel gets values of B
img(span, span, 2) = tmp;  // B channel gets value of R
```

Can also do it this way:

```cpp
array swapped = join(2, img(span, span, 2),  // blue
                    img(span, span, 1),  // green
                    img(span, span, 0));  // red
```

Or simply:

```cpp
array swapped = img(span, span, seq(2,-1,0));
```
Transposing an image in C++ using the NEON Parallel Processing (NPP) library.

```cpp
sFilename = "path_to_image"; // file name of source image
npp::ImageCPU_8u_C1 oHostSrc; // host image object
npp::loadImage(sFilename, oHostSrc); // load gray scale image
npp::ImageNPP_8u_C1 oDeviceSrc(oHostSrc); // copy image to device
NppiSize oSizeROI = {oDeviceSrc.width(), oDeviceSrc.height()}; // ROI
npp::ImageNPP_8u_C1 oDeviceDst(oSizeROI.width, oSizeROI.height); // destination device image

NppStatus errorStatus = nppiTranspose_8u_C1R
    (oDeviceSrc.data(), oDeviceSrc.pitch(), oDeviceDst.data(),
     oDeviceDst.pitch(), oSizeROI); // calculate transpose
```
Using ArrayFire:

```plaintext
array img = loadimage("image.jpg", false);  // load grayscale image from disk
array img_T = img.T();  // transpose
```
Grayscale
Box filter blur
Gaussian blur
Image Negative
Erosion

ArrayFire

// erode an image, 8-neighbor connectivity
array mask8 = constant(1,3,3);
array img_out = erode(img_in, mask8);

// erode an image, 4-neighbor connectivity
const float h_mask4[] = { 0.0, 1.0, 0.0,
                        1.0, 1.0, 1.0,
                        0.0, 1.0, 0.0 };
array mask4 = array(3, 3, h_mask4);
array img_out = erode(img_in, mask4);

NPP

nppeErode_8u_C1R (pSrc, nSrcStep, pDst, nDstStep, oSizeROI, pMask,
                 oMaskSize, oAnchor)
Filtering

ArrayFire:

array R = convolve(img, ker); // 1, 2 and 3d convolution filter
array R = convolve(fcol, frow, img); // Separable convolution
array R = filter(img, ker); // 2d correlation filter

NPP:

// Separable filtering
nppiFilterRow_8u_C1R(in, istep, out, ostep, ROI, ker, ksize, off, div);
nppiFilterColumn_8u_C1R(in, istep, out, ostep, ROI, ker, ksize, off, div);
// 2D filter
nppiFilter_8u_C1R(in, istep, out, ostep, ROI, ker, ksize, off, div);
Histograms

ArrayFire

```c
int nbins = 256;
array hist = histogram(img, nbins);
```

NPP

```c
int nbins = 256;
nppiHistogramRangeGetBufferSize_32f_C1R(imgROI, nbins, &buffsize);
nppiHistogramRange_32fC1R(d_img, lineStep, imgROI,
                           d_hist, d_binSizes, nbins+1, d_buff);
```
Transforms

ArrayFire

array half = resize(0.5, img);

array rot90 = rotate(img, af::Pi/2);

array warped = approx2(img, xLocations, yLocations);

NPP

\[
x' = c_{00} \ast x + c_{01} \ast y + c_{02} \\
y' = c_{10} \ast x + c_{11} \ast y + c_{12}
\]

C = \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \end{bmatrix}

nppiWarpAffine_32f_C1R (pSrc, oSrcSize, nSrcStep, oSrcROI, pDst, nDstStep, oDstROI, C, NPPI_INTER_LINEAR)
Image smoothing

ArrayFire:

array S = bilateral(I, sigma_r, sigma_c);
array M = meanshift(I, sigma_r, sigma_c, iter);
array R = medfilt(img, 3, 3);

// Gaussian blur
array gker = gaussiankernel(ncols, ncols);
array res = convolve(img, gker);
FFT

ArrayFire

array R1 = fft2(I);                              // 2d fft. check fft, fft3
array R2 = fft2(I, M, N);                        // fft2 with padding
array R3 = ifft2(fft2(I, M, N) * fft2(K, M, N)); // convolve using fft2

CUFFT by NVIDIA

cufftPlanMany(plan, rank, dims,                   // rank 1, 2, 3
              NULL, 1, 0, NULL, 1, 0,          // Used for strided FFTs
type, batch);                                   // type CUFFT_FORWARD or CUFFT_BACKWARD

cufftExecR2C(plan, in, out);                     // Forward. cufftExecC2R for Backward
cufftExecC2C(plan, in, out, type);              // Z for double precision
AccelerEyes

Custom development
Port CPU code
Extend existing code to use the GPU
Numerical, image, signal, algebra, ..

Training courses
1-4 days
CUDA, OpenCL
Hands on work with your code
Questions?
NPP is distributed with NVIDIA's CUDA Toolkit

ArrayFire is distributed by AccelerEyes.com