MATLAB: The challenges involved in providing a high-level language on a GPU

Jos Martin

jos.martin@mathworks.co.uk
Agenda

- Why did we introduce GPU support?
- What did we do?
- What did we think about?
Why introduce GPU support (H2 2010)

- Double support
  - Single/double performance inline with expectations
- Operations are IEEE Compliant
- Cross-platform support now available
Summary of Options for Using GPUs

Across one or more GPUs on one or more machines:

- Use GPU with MATLAB built-in functions (gpuArray, gather, etc.)
- Execute MATLAB functions elementwise on the GPU (arrayfun)
- Create kernels from existing CUDA code and PTX files, or integrate with C API
Making a `gpuArray`

- To make an array exist on the GPU

  ```
  g = gpuArray( dataOnCpu );
  g = gpuArray.zeros( argsToZeros );
  g = gpuArray.randn( argsToRandn );
  etc ...
  ```

- Supported types
  - All built-in numeric types
    ```
    [complex|][[uint|int][8|16|32|64]|double|single]
    ```
Using `gpuArray`

- Honestly – it’s just like an ordinary MATLAB array
- Except that the methods that are implemented for it will run on the GPU (over 200 currently and growing)
  - Maybe some of these will be faster on your GPU 😊

- The GPU’ness propagates
  ```matlab
  class( gpuArray(10) + 1 ) == 'gpuArray';
  ```

- Want to get the data back to the CPU
  ```matlab
  c = gather(g);
  ```
Example:
Corner Detection on the CPU

1. Calculate derivatives

\[
\text{dx} = \text{cdata}(2:\text{end}-1,3:\text{end}) - \text{cdata}(2:\text{end}-1,1:\text{end}-2);
\]
\[
\text{dy} = \text{cdata}(3:\text{end},2:\text{end}-1) - \text{cdata}(1:\text{end}-2,2:\text{end}-1);
\]
\[
\text{dx2} = \text{dx} .* \text{dx};
\]
\[
\text{dy2} = \text{dy} .* \text{dy};
\]
\[
\text{dxy} = \text{dx} .* \text{dy};
\]

2. Smooth using convolution

\[
\text{gaussHalfWidth} = \max(1, \text{ceil}(2*\text{gaussSigma})]
\]
\[
\text{ssq} = \text{gaussSigma}^2;
\]
\[
\text{t} = -\text{gaussHalfWidth} : \text{gaussHalfWidth};
\]
\[
\text{gaussianKernel1D} = \exp(-\text{t}.*\text{t}/(2*\text{ssq}))/\sqrt{2\pi}\text{ssq} ; \quad % \text{The Gaussian 1D filter}
\]
\[
\text{smooth_dx2} = \text{conv2}(\text{gaussianKernel1D}, \text{gaussianKernel1D}, \text{dx2}, 'valid');
\]
\[
\text{smooth_dy2} = \text{conv2}(\text{gaussianKernel1D}, \text{gaussianKernel1D}, \text{dy2}, 'valid');
\]
\[
\text{smooth_dxy} = \text{conv2}(\text{gaussianKernel1D}, \text{gaussianKernel1D}, \text{dxy}, 'valid');
\]

3. Calculate score

\[
\text{det} = \text{smooth_dx2} .* \text{smooth_dy2} - \text{smooth_dxy} .* \text{smooth_dxy};
\]
\[
\text{trace} = \text{smooth_dx2} + \text{smooth_dy2};
\]
\[
\text{score} = \text{det} - 0.25*\text{edgePhobia}*(\text{trace}.*\text{trace});
\]
**Example:**

**Corner Detection on the GPU**

```matlab
0. Move data to GPU

cdata = gpuArray( cdata );
dx = cdata(2:end-1,3:end) - cdata(2:end-1,1:end-2);
dy = cdata(3:end,2:end-1) - cdata(1:end-2,2:end-1);
dx2 = dx.*dx;
dy2 = dy.*dy;
dxy = dx.*dy;

gaussHalfWidth = max( 1, ceil( 2*gaussSigma ) );
ssq = gaussSigma^2;
t = -gaussHalfWidth : gaussHalfWidth;
gaussianKernel1D = exp(-(t.*t)/(2*ssq))/(2*pi*ssq);  % The Gaussian 1D filter

smooth_dx2 = conv2( gaussianKernel1D, gaussianKernel1D, dx2, 'valid' );
smooth_dy2 = conv2( gaussianKernel1D, gaussianKernel1D, dy2, 'valid' );
smooth_dxy = conv2( gaussianKernel1D, gaussianKernel1D, dxy, 'valid' );

det = smooth_dx2 .* smooth_dy2 - smooth_dxy .* smooth_dxy;
trace = smooth_dx2 + smooth_dy2;
score = det - 0.25*edgePhobia*(trace.*trace);

score = gather( score );

4. Bring data back
```
MATLAB build functions

- zeros, ones, eye, nan, inf, true, false

- 13a: New way to call these functions:

  \[ \text{zeros(..., 'like', P)} \text{ is an array of zeros with the same data type, sparsity, and complexity (real or complex) as the numeric variable P.} \]

  \[
  \text{>> } g = \text{gpuArray.zeros(10);}
  \text{>> likeG = ones(size(g), 'like', g);}
  \]
Problem

\[ g = \text{gpuArray}(10) + \text{int16}(1); \]

What is class(g)?
Where did the int16’ness go?
Semantic work pattern: **gpuArray**

\[ A \times B + C \]
Actual work pattern: `gpuArray`

```plaintext
a = b*c;
d = e*f;
```
arrayfun

Apply a function to each element of a set of gpuArrays
This method of gpuArray is very similar in behaviour to the MATLAB function arrayfun, except that the actual evaluation of the function happens on the GPU, not on the CPU. Thus any required data not already on the GPU is moved to GPU memory, and the MATLAB function referenced by FUN is then executed on the GPU. All the output arguments are returned as gpuArrays whose data can be retrieved with the GATHER method.

For example

\[ [o1, o2] = \text{arrayfun}(@aFunction, s1, s2, s3) \]
Why is this a good idea?

- We know what inputs are being passed to your function
- We know what code is in your function

  *if* we can type infer all variables in your code

  *then* we can then JIT compile your code for the GPU

- And so your function can then be executed on a single CUDA thread for each element of the input array
Work pattern: `arrayfun`

\[ A \times B + C \]
Mandelbrot
What can I do in an `arrayfun` function?

- All variables in the function need to remain scalar
  - Note that by definition all input args are already scalar.

- Almost all scalar operations are supported, inc. `rand` and `randn`.

- New 13a: Up-level variable access and indexing is supported.
Monte-Carlo and uplevel access
Invoking CUDA Kernels

% Setup
kern = parallel.gpu.CUDAKernel('myKern.ptx', cFcnSig)

% Configure
kern.ThreadBlockSize=[512 1];
kern.GridSize=[1024 1024];

% Run
[c, d] = feval(kern, a, b);
C API

{ /* You might want to do this statically outside the mex function */
    mxInitGPU();

    // This will make the first input exist on the GPU - if already a GPU array this is a no-op
    mxArray const * A = mxGPUCreateFromMxArray(prhs[0]);
    // We assume that the first input is double - you could check if this is the case
    double const * d_A = (double const *) (mxGPUGetDataReadOnly(A));

    /* Create a GPUArray to hold the result and get its underlying pointer. */
    mxArray * B = mxGPUCreateGPUArray(mxGPUGetNumberOfDimensions(A),
                                       mxGPUGetDimensions(A),
                                       mxGPUGetClassID(A),
                                       mxGPUGetComplexity(A),
                                       MX_GPU_DO_NOT_INITIALIZE);
    double * d_B = (double *) (mxGPUGetData(B));

    /* * Call the kernel using the CUDA runtime API. We are using a 1-d grid here,
       * and it would be possible for the number of elements to be too large for
       * the grid. For this example we are not guarding against this possibility.
       */
    int N = (int) (mxGPUGetNumberOfElements(A));
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    // Standard CUDA kernel call using the CUDA runtime.
    TimesTwo<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, N);

    // Device code prototype ...
    void __global__ TimesTwo(double const * const A, double * const B, int const N) { ... };
}
# 7 Dwarfs on a GPU in MATLAB

<table>
<thead>
<tr>
<th>Dwarf</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>dense linear algebra</td>
<td>gpuArray</td>
</tr>
<tr>
<td>spectral methods</td>
<td>gpuArray</td>
</tr>
<tr>
<td>monte carlo</td>
<td>gpuArray or arrayfun</td>
</tr>
<tr>
<td>structured grids</td>
<td>arrayfun, CUDA kernel or C API</td>
</tr>
<tr>
<td>sparse linear algebra</td>
<td>CUDA kernel or C API (cuSparse)</td>
</tr>
<tr>
<td>unstructured grids</td>
<td>CUDA kernel or C API</td>
</tr>
<tr>
<td>N-body methods</td>
<td>arrayfun, CUDA kernel or C API</td>
</tr>
</tbody>
</table>
What did we think about?
Attributes of MATLAB

- Bug-free, Fast, Mathematically Accurate, Usable
- Mathematically Accurate
- Usable
- Bug-free
- Fast

- Add the following for GPU work
  - Code & install size
  - Generality
Why do I care about code size?

[jlmartin@uk-jlmartin01 glnxa64]$ ls -lhS | head -n 10

90M Nov 16 23:58 libnpp.so.5.0.35
69M Nov 17 12:29 libmwgpuruntime.so
57M Nov 16 23:57 libcublas.so.5.0.40
37M Nov 16 23:58 mkl.so
30M Nov 16 23:57 libcufft.so.5.0.35
29M Nov 21 17:38 libmwsimulink.so
29M Nov 17 11:54 libmwhandle_graphics.so
24M Nov 16 23:58 libQtWebKit.so.4.9.2
22M Nov 17 01:03 ps_be
22M Nov 17 01:03 ps_fixpoint

GPU is 246MB of 1.1GB total (257MB of 1GB on win64)
Generality

Quotes I’ve heard:

*We support the general case for GEMM, namely with square matrices*

*Why would you want to take the FFT of a vector who's length contains a three digit prime factor?*
Trade-offs

- Generality
- Usability
- Performance
- Correctness
- Code Size
Performance considerations

- Relatively predictable performance
  - Don’t tune for special cases if it impacts general case
  - Try to have smooth performance characteristics if possible
  - Don’t leave out cases because they don’t work well

- Reproducibility is critical

- It is almost impossible to predict what will optimize a kernel
Performance of rand

![Graph showing RAND performance on the GPU](image)

- Twister (CPU)
- MRG32K3a (R123)
- Threelfry4x64-20 (R123)
- Philox4x32-10 (R123)

Throughput (GB/sec) vs Number of elements
Problem

- I want to map $i_1$ & $i_2$ (a pair of uint32) onto $d$ in $(0, 1)$
- Should I …
  - $\text{ldexp}(i_1, -64) + \text{ldexp}(i_2, -32) + \text{scale}$;
  - $\text{ldexp}(\text{uint64}(i_1, i_2), -64) + \text{scale}$;
  - $\text{double(\text{uint64}(i_1, i_2))}*\text{scale} + \text{scale}$;
  - $\text{double(\text{uint64}(i_1, i_2) + 1)}*\text{scale}$;

For reference:

```c
double ldexp ( double x, int exp );
```

The function returns:

\[
x \times 2^{\exp}
\]
Performance of randn
What is \( \sqrt{-1} \)?

- Reasonable options
  - 0 + 1i (i.e. output becomes complex)
  - Error
  - NaN?

- Implications?
  - Need to consider both `gpuArray` & `arrayfun`
\[ \sqrt{-1} = 0 + 1i \]

\[ A = \sqrt{\text{gpuData}}; \]

gpuArray
- Can you allocate memory when inside a kernel?
  - We would have to re-run the kernel
  - At least only users who need complex would be affected

arrayfun
- In JIT code what would we hold the return of sqrt in?
  - Complexity propagates
  - Everyone is worse off
\texttt{sqrt(-1) returns NaN}

\texttt{A = sqrt(gpuData);}

- Different to every form of the MATLAB language except MATLAB coder code running on a real external target.
\texttt{sqrt(-1)} throws Error

\begin{verbatim}
A = sqrt(gpuData);
\end{verbatim}

gpuArray

- Where do you get the error?
  - Need to ensure that it happens at the right time

arrayfun

- Need to throw an error from a thread in a function
What would you do?
Questions?