MATLAB:
GPU support in a high-level language

Ben Tordoff
Lead developer: GPU and parallel algorithms
MathWorks

Agenda

- **Why add GPU support?**
- **Who is it for?**
- **What does it look like?**

Why add GPU support?

- Customer requests
- Hardware becoming common
- Allows certain algorithms to be accelerated
- An established platform for HPC

Why wait until 2010?

GPU support was first added in Autumn 2010 (R2010b).

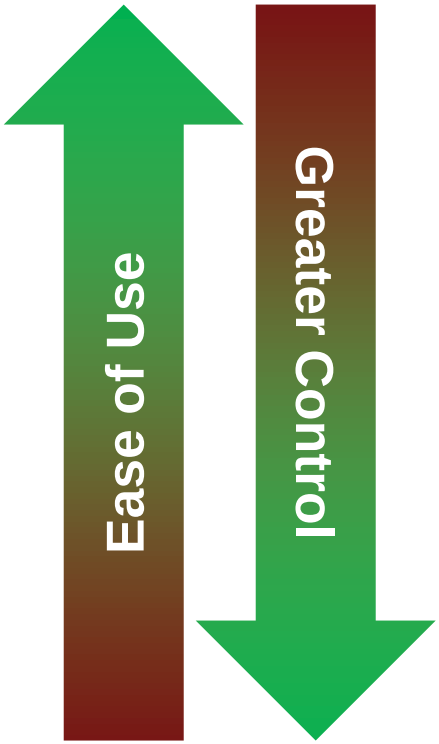
Our requirements:

- Double support
  - Single/double performance inline with expectations
- IEEE Compliant
- Cross-platform
- Mature libraries (FFT, BLAS, LAPACK etc.)

Agenda

- Why add GPU support?
- Who is it for?
- What does it look like?

Who is the target audience?



- Everyday MATLAB users
- Tool builders
- GPU developers

Who is the target audience?

- Everyday MATLAB users
  - Want extra speed
  - Don’t want to have to learn lots of new stuff
  - Don’t want to change code
  - Want to do it all from within MATLAB

Who is the target audience?

- Tool builders
  - Want extra speed
  - Willing to learn new things
  - Willing to customize/optimize code for speed
  - Want to do it all from within MATLAB

Who is the target audience?

- GPU developers
  - Want every ounce of available speed
  - Know MATLAB, C++, CUDA, …
  - Live to customize/optimize code for speed
  - Want to integrate CUDA/C++ code with MATLAB

Agenda

- **Why add GPU support?**
- **Who is it for?**
- **What does it look like?**

Who is the target audience?

- Can we support all the different users with one API?
  - **No**
- Different users want very different levels of control
- Need APIs suited to users

Agenda

- **Why add GPU support?**
- **Who is it for?**
- **What does it look like?**
  - **Everyday MATLAB user**

Example:

Corner Detection on the CPU

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

gaussHalfWidth = max( 1, ceil( 2*gaussSig
ssq = gaussSigma^2;
t = -gaussHalfWidth : gaussHalfWidth;
gaussianKernel1D = exp(-(t.*t)/(2*ssq))/(2*pi*ssq);     % The Gaussian 1D filter
gaussianKernel1D = gaussianKernel1D / sum(gaussianKernel1D);
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_d
trace = smooth_dx2 + smooth_dy2;
score = det - 0.25*edgePhobia*(trace.*trace);
```

1. Calculate derivatives

2. Smooth by convolution

3. Calculate score

Example:

Corner Detection on the GPU

```
cdata = gpuArray( cdata );
```

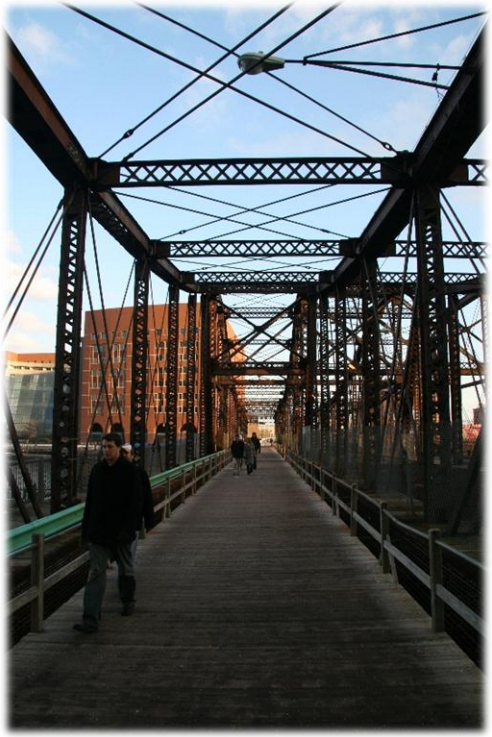
0. Move data to GPU

```
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
gaussianKernel1D = gaussianKernel1D / sum(gaussianKernel1D);
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

Results



- Image is from an 8MP DSLR
  - (3504x2336)
- Host-PC (6-core Intel Xeon @3.5GHz)
  - 0.42 secs
- GPU (NVIDIA Tesla K20c)
  - 0.065 secs (6.5x faster)
    - = 0.036 secs for algorithm
    - + 0.029 secs for data-transfer

Making a `gpuArray`

- To make an array exist on the GPU

```
g = gpuArray( dataOnHost );
g = zeros( argsToZeros, 'gpuArray' );
g = randn( argsToRandn, 'gpuArray' );
    and others...
```

- To move a data back to main memory

```
x = gather( dataOnGPU );
```

- Supports all built-in numeric types plus logicals

```
[complex|][[uint|int][8|16|32|64]|double|single]|logical
```

Why have an API at all?

- Should we just use the GPU without you knowing?

- Answers can be different on the GPU
- Reproducibility is a key requirement for our customers

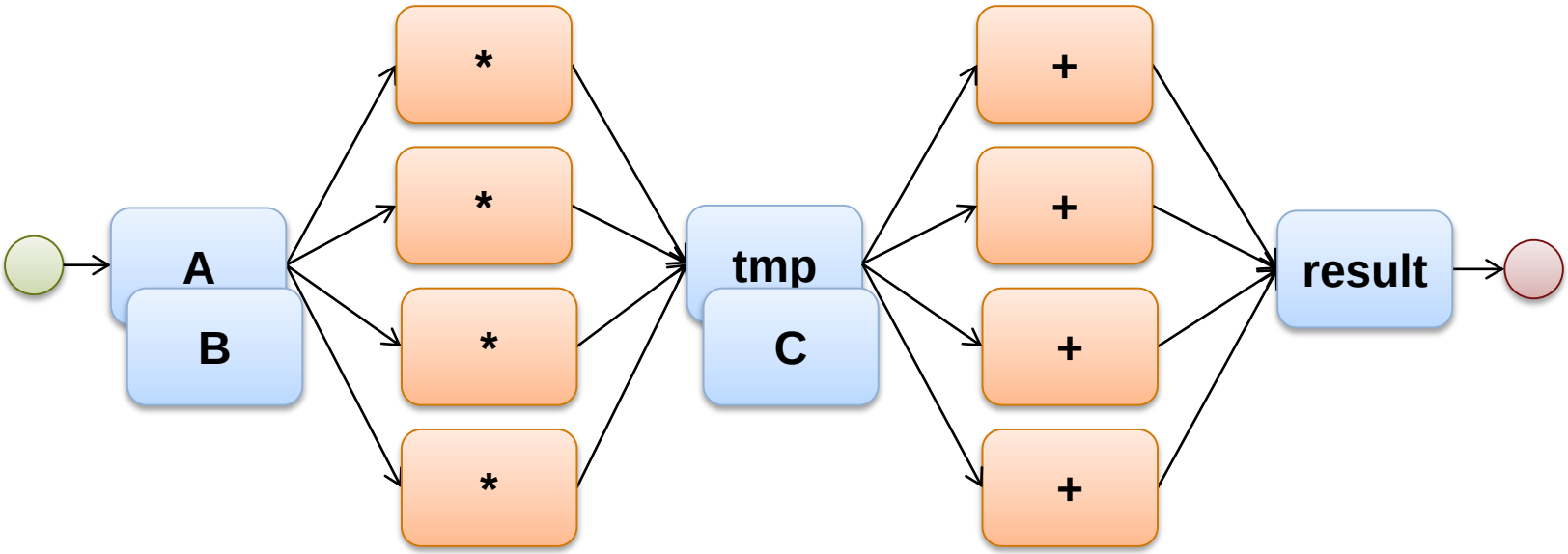
- Transferring data to and from the GPU can be slow
  - For big operations (large linear algebra problems, big FFTs etc.) this might not matter
  - For medium or small operations it would cripple performance
- We need the programmer to tell us when it is worth transferring the data

Agenda

- **Why add GPU support?**
- **Who is it for?**
- **What does it look like?**
  - **Everyday MATLAB user**
  - **Tool builder**

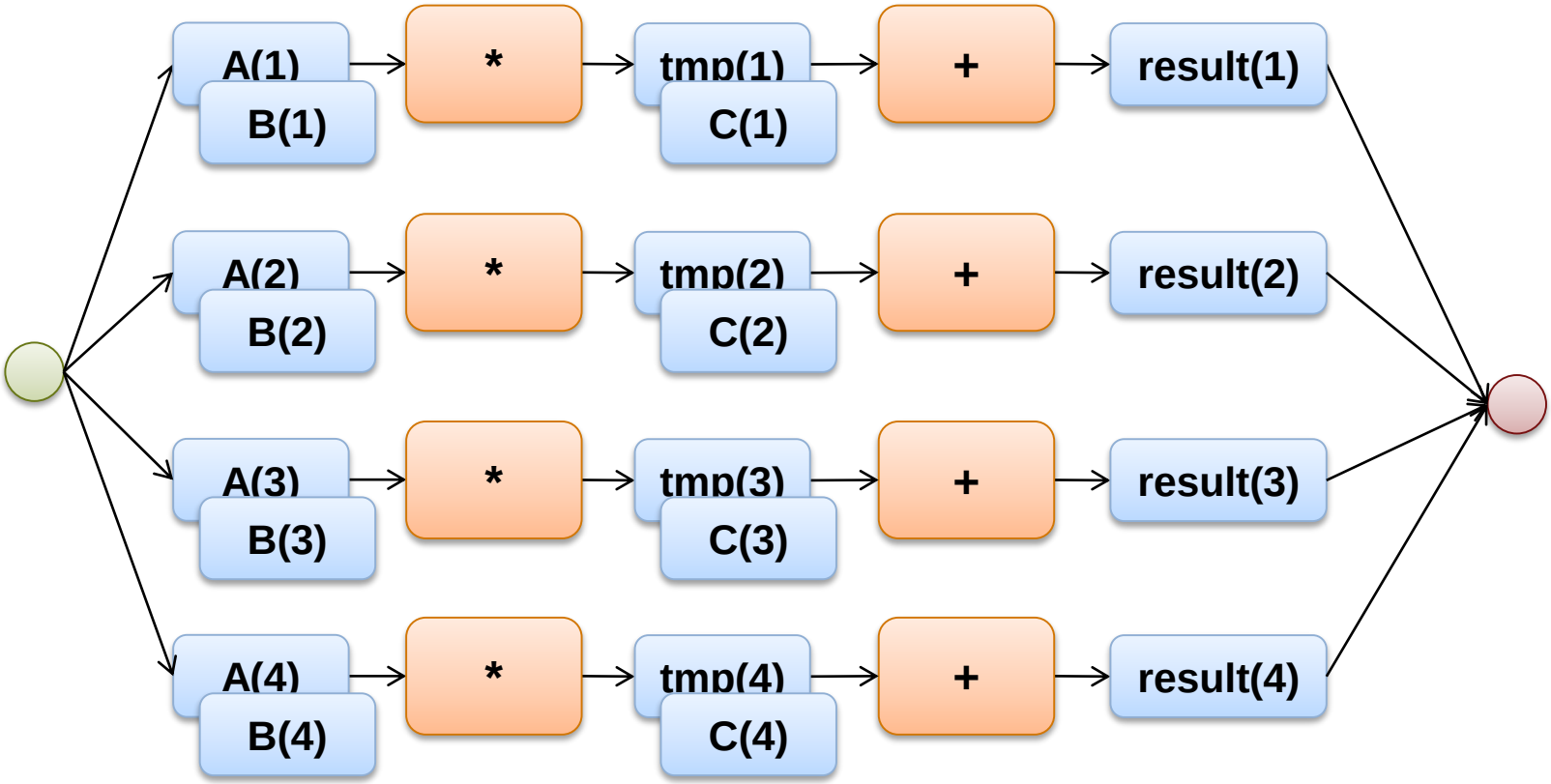
Work pattern: `gpuArray`

```
result = (A .* B) + C
```



Work pattern: `arrayfun`

```
fcn = @(A,B,C) (A .* B) + C;
result = arrayfun(@fcn,A,B,C)
```



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 generate code for the GPU

- your function executes as a single CUDA kernel, with one thread for each element of the input array

Other ways to express parallelism

- PAGEFUN – run a 2-D operation on every page of some N-D arrays.

- E.g. multiply 10,000 3x3 matrices:

```
A = rand(3,3,10000,'gpuArray'); % 10000 3x3 matrices
b = rand(3,1,'gpuArray');

C = pagefun( @mtimes, A, b ); % C will be 3x1x10000
```

Agenda

- **Why add GPU support?**
- **Who is it for?**
- **What does it look like?**
  - **Everyday MATLAB user**
  - **Tool builder**
  - **GPU programmer**

Invoking CUDA Kernels

- Call a CUDA kernel straight from MATLAB
- Use MATLAB as a fast kernel prototyping environment

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

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

% Run
c = feval(kern, a, b);
```

```
__global__
void myKern(double * arg1, double const * arg2)
{
    int const idx = threadIdx.x + blockIdx.x*blockDim.x;
   arg1[idx] += arg2[idx];
}
```

C (MEX) API

- MEX function appears as a standard MATLAB function
- Implemented in a mix of C / C++ and CUDA code
- Can call other CUDA libraries (OpenCV, CuSparse etc.)
- Programmer has to manage data, kernel/library calls, and synchronisation issues

C (MEX) API

```
void mexFunction(int nlhs, mxArray *plhs[],
                 int nrhs, mxArray const *prhs[])
{
    // Initialize the MathWorks GPU API.
    mxInitGPU();

    // Get the input image
    mxGPUArray const * I = mxGPUCreateFromMxArray(prhs[0]);
    // Wrap I with an OpenCV GPU matrix
    float const * d_I = (float const *)(mxGPUGetDataReadOnly(I));
    cv::gpu::GpuMat const cv_image(cv::Size(M, N), CV_8UC1, (void *)d_I);

[snip]

    // Detect corner features using the OpenCV GPU FAST feature detector
    std::vector<cv::KeyPoint> keypoints;
    cv::gpu::FAST_GPU featureDetector(threshold);
    featureDetector(cv_image, cv::gpu::GpuMat(), keypoints);

    // Assign output
    plhs[0] = fastKeyPointToMATLABStruct(keypoints);

    // The mxGPUArray pointers are host-side structures that refer to device
    // data. These must be destroyed before leaving the MEX function.
    mxGPUDestroyGPUArray(I);
}
```

Summary

- Expose the GPU using three levels of API:
  - gpuArray for minimal code change
  - arrayfun, bsxfun, pagefun for optimizing code
  - CUDAKernel and CUDA-MEX for integrating CUDA kernels or libraries