GPU Coder:
Automatic CUDA and TensorRT code generation from MATLAB

Girish Venkataramani
Arvind Jayaraman
Jaya Shankar
GPUs and CUDA programming

GPUs are “hardware on steroids”, but, programming them is hard
Consider an example: saxpy

Scalarized MATLAB

```
for i = 1:length(x)
    z(i) = a .* x(i) + y(i);
end
```

Vectorized MATLAB

```
z = a .* x + y;
```

Automatic compilation from an expressive language to a high-performance language
Deep Learning applications

TensorRT is great for inference, ... but, applications require more than inference
GPU Coder is new technology released in September 2017

Accelerated implementation of parallel algorithms on GPUs

Neural Networks
Deep Learning, machine learning

5x faster than TensorFlow
2x faster than mxnet

Image Processing and Computer Vision
Image filtering, feature detection/extraction

60x faster than CPUs for stereo disparity

Signal Processing and Communications
FFT, filtering, cross correlation,

20x faster than CPUs for FFTs
Example: Lidar semantic segmentation
Talk outline

- Introduction

- GPU Coder internals
  - Automatic parallelization
  - Memory optimization
  - Deep learning compilation

- Application demo: Lidar processing in MATLAB using deep learning
GPU Coder automatically extracts parallelism from MATLAB

1. Scalarized MATLAB
   ("for-all" loops)

2. Vectorized MATLAB
   (math operators and library functions)

3. Composite functions in MATLAB
   (maps to cuBlas, cuFFT, cuSolver, cuDNN, TensorRT)
From a loop to a CUDA kernel

for k = 1:n
    t = A(k) .* X(k);
    C(k) = t + Y(k);
end

{ …
    mykernel<<< f(n) >>>( …
    }

static __global__ mykernel(A, X, Y, C, n)
{
    …
}

Extracting parallelism in MATLAB
1. Scalarized MATLAB (for loops)
2. Vectorized MATLAB
3. Composite functions

Dependence analysis to understand the iteration space
From a loop-nesting to CUDA kernels

Imperfect Loops

```
for i = 1:p
  ...(outer prologue code)...
  for j = 1:m
    for k = 1:n
      ...(inner loop)...
      end
    end
  end
  ...(outer epilogue code)...
end
```

Perfect Loops

```
for i = 1:p
  for j = 1:m
    for k = 1:n
      ...(inner loop)...
    end
  end
end
```

Fundamental requirement: Loops need to be contiguous for parallelization
From a loop-nesting to CUDA kernels

**Imperfect Loops**

```matlab
for i = 1:p
    ...(outer prologue code)...
    for j = 1:m
        for k = 1:n
            ...(inner loop)...
        end
    end
    ...(outer epilogue code)...
end
```

**Perfect Loops**

```matlab
for i = 1:p
    ...(outer prologue code)...
    for j = 1:m
        for k = 1:n
            ...(inner loop)...
        end
    end
    if k == n
        ...(outer epilogue code)...
    end
end
end
```

**Fundamental requirement:** Loops need to be contiguous for parallelization
From a loop-nesting to CUDA kernels

Example 1

\[(M \times N)\]

for a = 1:M
  for b = 1:N
    ...(outer prologue code)...
  end
end

\[(K \times P)\]

for c = 1:K
  for d = 1:P
    ...(inner loop)...
  end
end

Example 2

\[(P)\]

for i = 1:P
  for a = 1:M
    for b = 1:N
      ...(inner loop)...
    end
  end
end

\[(M \times N + K \times L)\]

for x = 1:K
  for y = 1:L
    ...(inner loop)...
  end
end

Find parallel loops \rightarrow \text{Partition loop nesting}

Dependence analysis

Heuristic may favor larger iteration space

Fundamental requirement: Loops need to be contiguous for parallelization
From a loop-nesting to CUDA kernels

Example 1

\[
\begin{align*}
(M \times N) & \quad \text{for } a = 1:M \\
& \quad \text{for } b = 1:N \\
& \quad \text{... (outer prologue code) ...} \\
& \quad \text{for } c = 1:K \\
& \quad \text{for } d = 1:P \\
& \quad \text{... (inner loop) ...} \\
& \quad \text{end} \\
& \quad \text{end} \\
& \quad \text{end} \\
(K \times P) & \quad \text{end}
\end{align*}
\]

Example 2

\[
\begin{align*}
(P) & \quad \text{for } i = 1:P \\
& \quad \text{for } a = 1:M \\
& \quad \text{for } b = 1:N \\
& \quad \text{... (inner loop) ...} \\
& \quad \text{end} \\
& \quad \text{end} \\
& \quad \text{end} \\
(M \times N + K \times L) & \quad \text{end}
\end{align*}
\]

- Find parallel loops
- Partition loop nesting
- Create kernel for each partition

Dependence analysis
Heuristic may favor larger iteration space
Use process from single loop conversion

**Fundamental requirement:** Loops need to be contiguous for parallelization
From vectorized MATLAB to CUDA kernels

output(:, 1) = (input(:, 1) – x_im) .* factor;

for i = 1:M
    diff(i) = input(i, 1) – x_im(i);
end
for a = 1:M
    output(i, 1) = diff(i) * factor(i);
end

Assume the following sizes:
- ‘output’ : M x 3
- ‘input’ : M x 3
- ‘x_im’ : M x 1
- ‘factor’ : M x 1

1. Scalarized MATLAB (for loops)
2. Vectorized MATLAB
3. Composite functions
From vectorized MATLAB to CUDA kernels

output(:, 1) = (input(:, 1) – x_im) .* factor;

for i = 1:M
    diff(i) = input(i, 1) – x_im(i);
end
for a = 1:M
    output(i, 1) = diff(i) * factor(i);
end

Assume the following sizes:
- ‘output’ : M x 3
- ‘input’ : M x 3
- ‘x_im’ : M x 1
- ‘factor’ : M x 1

Extracting parallelism in MATLAB
1. Scalarized MATLAB (for loops)
2. Vectorized MATLAB
3. Composite functions

Scalarization
Reduce to for-loops

Loop Fusion
Create larger parallel loops (and hence CUDA kernels)

Scalar Replacement
Reduce temp matrices to temp scalars
From composite functions to optimized CUDA

- Core math
  - Matrix multiply (cuBLAS)
  - Linear algebra (cuSolver)
  - FFT functions (cuFFT)
  - Convolution
  - ...

- Image processing
  - imfilter
  - imresize
  - imerode
  - imdilate
  - bwlabel
  - imwarp
  - ...

- Computer vision

- Neural Networks
  - Deep learning inference (cuDNN, TensorRT)
  - …

Over 300+ MATLAB functions are optimized for CUDA code generation
Talk outline

- Introduction

- GPU Coder internals
  - Automatic parallelization
  - Memory optimization
  - Deep learning compilation

- Application demo: Lidar processing in MATLAB using deep learning
Optimizing CPU-GPU data movement is a challenge

\begin{align*}
A &= \ldots \\
&\ldots \text{for } i = 1:N \\
&\quad \ldots A(i) \\
&\text{end} \\
&\ldots \\
\text{imfilter} \\
&\ldots
\end{align*}

\begin{align*}
A &= \ldots \\
&\ldots \text{cudaMemcpyHtoD}(gA, a); \\
&\text{kernel1}<<<\ldots>>>(gA) \\
&\text{cudaMemcpyDtoH}(\ldots) \\
&\ldots \\
&\text{cudaMemcpyHtoD}(\ldots) \\
&\text{imfilter\_kernel}(\ldots) \\
&\text{cudaMemcpyDtoH}(\ldots) \\
&\ldots
\end{align*}

Where is the ideal placement of memcpy?
GPU Coder optimizes memcpy placement

Assume gA, gB and gC are mapped to GPU memory

```
A(:) = ... 
C(:) = ...

for i = 1:N
    ...
    gB = kernel1(gA);
    gA = kernel2(gB);
    if (some_condition)
        gC = kernel3(gA, gB);
    end
    ...
end
...
... = C;
```

**Generated (pseudo) code**

```
A(:) = ... 
A_isDirtyOnCpu = true;
...
for i = 1:N
    if (A_isDirtyOnCpu)
        cudaMemcpy(gA, A);
        A_isDirtyOnCpu = false;
    end
    gB = kernel1(gA);
    gA = kernel2(gB);
    if (some_condition)
        gC = kernel3(gA, gB);
    end
    ...
end
...
if (C_isDirtyOnGpu)
    cudaMemcpy(C, gC);
    C_isDirtyOnGpu = true;
end
...
... = C;
```
GPU memory hierarchy is deep

<table>
<thead>
<tr>
<th>Memory</th>
<th>Visibility</th>
<th>Heuristics/notes</th>
<th>GPU Coder support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global memory</td>
<td>CPU + GPU</td>
<td>Share data b/w CPU and GPU</td>
<td>Yes</td>
</tr>
<tr>
<td>Local memory/registers</td>
<td>per GPU thread</td>
<td>Thread local data</td>
<td>Yes</td>
</tr>
<tr>
<td>Shared memory</td>
<td>per GPU block</td>
<td>Shared data between threads</td>
<td>Yes</td>
</tr>
<tr>
<td>Texture memory</td>
<td>CPU + GPU</td>
<td>Shared read-only data with 2D alignment</td>
<td>Future</td>
</tr>
<tr>
<td>Constant memory</td>
<td>GPU</td>
<td>Read-only constants</td>
<td>Yes</td>
</tr>
</tbody>
</table>
GPU Coder automatically maps data to shared memory

GPU Coder automatically creates shared memory for many MATLAB image processing functions:

`imfilter`, `imerode`, `imdilate`, `conv2`, ...
GPU Coder runs a host of compiler transforms to generate CUDA

MATLAB

Front-end

Control-flow graph Intermediate representation (CFG – IR)

Loop optimizations

Traditional compiler optimizations

Library function mapping

Scalarization

Loop perfectization

Loop interchange

Loop fusion

Scalar replacement

Parallel loop creation

CUDA kernel creation

cudaMemcpy minimization

Shared memory mapping

CUDA code emission
Demo: Stereo disparity

Left camera

Right camera

Stereo disparity

Disparity map
Easily re-target to Jetson and Drive platforms

- Cross-compile for NVIDIA boards
  - Jetson boards
  - DrivePX2

Two small changes

1. Change build-type to 'lib'
2. Select cross-compile toolchain
Talk outline

- Introduction

- GPU Coder internals
  - Automatic parallelization
  - Memory optimization
  - Deep learning compilation

- Application demo: Lidar processing in MATLAB using deep learning
Deep learning workflow in MATLAB

- **Design in MATLAB**
  - Manage large image sets
  - Automate data labeling
  - Easy access to models

- **Training in MATLAB**
  - Acceleration with GPU’s
  - Scale to clusters

- **Train in MATLAB**
- **Trained DNN**
- **Model importer**
- **Model importer**
- **Model importer**

- **DNN design + training**
- **Model importer**
- **Model importer**
Deep learning workflow in MATLAB

1. Idea
2. Caffe/Model
3. Model importer
4. Train in MATLAB
5. Trained DNN
6. Keras/TensorFlow
7. Model importer
8. Application design
9. Application logic
Deep learning workflow in MATLAB

1. **DNN design + training**
   - Idea
   - Caffe
   - TensorFlow
   - Keras

2. **Model importer**

3. **Train in MATLAB**

4. **Trained DNN**

5. **Application design**
   - Application logic
   - MATLAB

6. **Standalone Deployment**
   - C++/CUDA + TensorRT
   - C++/CUDA + cuDNN

7. **GPU Coder**

---

**Notes:**
- MATLAB supports deep learning with various tools and environments.
- Caffe and TensorFlow are popular frameworks for building deep neural networks.
- Keras integrates seamlessly with these frameworks in MATLAB.
- Standalone部署 allows for deployment outside of MATLAB, utilizing C++/CUDA for efficient execution.
- TensorRT and cuDNN are specialized libraries for optimizing and deploying deep learning models on GPUs.
Deep learning workflow in MATLAB

**DNN design + training**

- **Train in MATLAB**
  - Caffe
  - Keras
  - TensorFlow

- **Model importer**
  - Trained DNN

**Application design**

- **Application logic**
- **MATLAB**

**Standalone Deployment**

- **C++/CUDA + TensorRT**
- **C++/CUDA + cuDNN**

**Standalone Deployment**

- **GPU Coder**
Traffic sign detection and recognition

- Object detection DNN
- Strongest Bounding Box
- Classifier DNN
GPU Coder allows for choice in deployment (cuDNN, TensorRT)
Performance summary (VGG-16) on TitanXP

- GPU Coder (TensorRT int8)
- GPU Coder (TensorRT fp32)
- GPU Coder (cuDNN fp32)
- MATLAB (cuDNN fp32)
MATLAB and GPU Coder support state-of-art classification networks

<table>
<thead>
<tr>
<th>Network</th>
<th># Layers</th>
<th>Size</th>
<th>Frame-rate (GPU Coder)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexnet</td>
<td>25</td>
<td>227 MB</td>
<td>500 Fps</td>
</tr>
<tr>
<td>ResNet50</td>
<td>177</td>
<td>96 MB</td>
<td>160 Fps</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>144</td>
<td>27 MB</td>
<td>190 Fps</td>
</tr>
<tr>
<td>Squeezenet</td>
<td>68</td>
<td>5 MB</td>
<td>615 Fps</td>
</tr>
</tbody>
</table>
Alexnet Inference on NVIDIA Titan Xp

<table>
<thead>
<tr>
<th>Testing platform</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz</td>
</tr>
<tr>
<td>GPU</td>
<td>Pascal Titan Xp</td>
</tr>
<tr>
<td>cuDNN</td>
<td>v7</td>
</tr>
</tbody>
</table>

Frames per second vs Batch Size

- MATLAB GPU Coder + TensorRT 3.0.1 (int8)
- MATLAB GPU Coder + TensorFlow (1.6.0)
- MATLAB GPU Coder + cuDNN

Testing platform:
- CPU: Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz
- GPU: Pascal Titan Xp
- cuDNN: v7
VGG-16 Inference on NVIDIA Titan Xp

Frames per second vs Batch Size

Testing platform:
- CPU: Intel(R) Xeon(R) CPU E5-1650 v3 @ 3.50GHz
- GPU: Pascal Titan Xp
- cuDNN: v7

Frameworks:
- MATLAB GPU Coder + TensorRT 3.0.1 (int8)
- MATLAB GPU Coder + TensorRT 3.0.1
- MATLAB GPU Coder + cuDNN
- mxNet (1.1.0)
- TensorFlow (1.6.0)
Talk outline

- Introduction

- GPU Coder internals
  - Automatic parallelization
  - Memory optimization
  - Deep learning compilation

- Application demo: Lidar processing in MATLAB using deep learning
LiDAR Processing for Autonomous Vehicles

Design Deep learning-based LiDAR algorithm in MATLAB
- **Automate** ground-truth labeling
- **Pre-process** LiDAR data for training
- **GPU accelerated** training

High Performance Inference
Why Use LiDAR and Deep Learning for Autonomous Vehicles?

- **Why use LiDAR?**
  - Provides accurate 3-D structure of scene
  - Required sensor for autonomous driving

- **Why use deep learning?**
  - Best accuracy
  - High-performance inference with GPU Coder
What Does LiDAR Data Look Like?
Lidar processing application design is easy in MATLAB

**DNN design + training**
- Caffe
- Keras
- TensorFlow

**Train in MATLAB**
- Model importer
- Trained DNN

**Application design**
- Application logic

**Standalone Deployment**
- C++/CUDA + TensorRT
- C++/CUDA + cuDNN

**GPU Coder**
Lidar processing application design is easy in MATLAB

DNN design + training

Data prep, labeling

Training

Trained DNN

Application design

Application logic

Standalone Deployment

C++/CUDA + TensorRT

GPU Coder

C++/CUDA + cuDNN
Data preparation and labeling of Lidar is a challenge
Access and Visualize LiDAR Data

**Access Stored LiDAR Data**
- Velodyne file I/O (pcap)
- Individual point clouds (.pcd,ply)

**Visualize LiDAR Data**
- Streaming LiDAR player
- Static point cloud display
- Point cloud differences
LiDAR Preprocessing

ROI + Remove Ground
- Fit plane using RANSAC

Cluster
- Segment clusters using Euclidean distance
Ground Truth Labeling of LiDAR Data
Ground Truth Labeling of LiDAR Data
Ground Truth Labeling of LiDAR Data
Organize Data for Training

Raw Point Cloud Data

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NaN</td>
<td>0.1703</td>
<td>0.2227</td>
<td>0.2505</td>
<td>0.4506</td>
<td>0.5542</td>
<td>NaN</td>
</tr>
<tr>
<td>2</td>
<td>0.0770</td>
<td>0.1032</td>
<td>0.3064</td>
<td>0.4593</td>
<td>0.5234</td>
<td>0.5784</td>
<td>0.6767</td>
</tr>
<tr>
<td>3</td>
<td>NaN</td>
<td>NaN</td>
<td>0.2822</td>
<td>0.4234</td>
<td>0.5332</td>
<td>0.5361</td>
<td>0.6364</td>
</tr>
<tr>
<td>4</td>
<td>0.0834</td>
<td>0.2183</td>
<td>0.2875</td>
<td>0.2901</td>
<td>0.5028</td>
<td>0.6081</td>
<td>NaN</td>
</tr>
<tr>
<td>5</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>0.7061</td>
<td>0.8309</td>
<td>NaN</td>
</tr>
<tr>
<td>6</td>
<td>0.0791</td>
<td>0.3409</td>
<td>0.4531</td>
<td>0.5654</td>
<td>0.7481</td>
<td>0.8279</td>
<td>1.1062</td>
</tr>
<tr>
<td>7</td>
<td>0.0745</td>
<td>0.2363</td>
<td>0.3967</td>
<td>0.5615</td>
<td>0.7447</td>
<td>0.9238</td>
<td>1.1042</td>
</tr>
<tr>
<td>8</td>
<td>0.0651</td>
<td>0.2206</td>
<td>0.3885</td>
<td>0.5512</td>
<td>0.7337</td>
<td>0.9124</td>
<td>1.0923</td>
</tr>
<tr>
<td>9</td>
<td>0.0530</td>
<td>0.2177</td>
<td>0.3790</td>
<td>0.5424</td>
<td>0.7355</td>
<td>0.9542</td>
<td>1.0641</td>
</tr>
<tr>
<td>10</td>
<td>0.0512</td>
<td>0.2130</td>
<td>0.3749</td>
<td>0.5376</td>
<td>0.7265</td>
<td>0.8993</td>
<td>1.0796</td>
</tr>
<tr>
<td>11</td>
<td>0.0419</td>
<td>0.2037</td>
<td>0.2657</td>
<td>0.5283</td>
<td>0.7112</td>
<td>0.8901</td>
<td>1.0706</td>
</tr>
<tr>
<td>12</td>
<td>0.0311</td>
<td>0.1824</td>
<td>0.3295</td>
<td>0.4724</td>
<td>0.6299</td>
<td>0.7765</td>
<td>0.8202</td>
</tr>
<tr>
<td>13</td>
<td>0.0074</td>
<td>0.0459</td>
<td>0.0896</td>
<td>0.1270</td>
<td>0.1690</td>
<td>0.2564</td>
<td>0.2455</td>
</tr>
<tr>
<td>14</td>
<td>0.0046</td>
<td>0.0426</td>
<td>0.0214</td>
<td>0.1174</td>
<td>NaN</td>
<td>0.1951</td>
<td>0.2214</td>
</tr>
<tr>
<td>15</td>
<td>0.0023</td>
<td>0.0407</td>
<td>0.0779</td>
<td>0.1138</td>
<td>0.1520</td>
<td>0.1802</td>
<td>0.2620</td>
</tr>
<tr>
<td>16</td>
<td>0.0011</td>
<td>0.0254</td>
<td>0.0705</td>
<td>0.1180</td>
<td>0.1511</td>
<td>0.1526</td>
<td>0.2270</td>
</tr>
<tr>
<td>17</td>
<td>0.0281</td>
<td>0.0063</td>
<td>0.1033</td>
<td>NaN</td>
<td>0.1793</td>
<td>NaN</td>
<td>0.2530</td>
</tr>
<tr>
<td>18</td>
<td>0.0259</td>
<td>NaN</td>
<td>NaN</td>
<td>0.1375</td>
<td>0.1754</td>
<td>NaN</td>
<td>0.2517</td>
</tr>
<tr>
<td>19</td>
<td>0.0237</td>
<td>0.0621</td>
<td>0.0607</td>
<td>0.1537</td>
<td>0.1776</td>
<td>NaN</td>
<td>0.2515</td>
</tr>
<tr>
<td>20</td>
<td>0.0234</td>
<td>0.0631</td>
<td>0.0992</td>
<td>0.1540</td>
<td>0.1754</td>
<td>0.2162</td>
<td>0.2467</td>
</tr>
<tr>
<td>21</td>
<td>0.0182</td>
<td>0.0525</td>
<td>0.0910</td>
<td>0.1231</td>
<td>0.1627</td>
<td>0.1879</td>
<td>0.2322</td>
</tr>
<tr>
<td>22</td>
<td>0.0168</td>
<td>0.0468</td>
<td>0.0829</td>
<td>0.1143</td>
<td>0.1491</td>
<td>0.1816</td>
<td>0.2182</td>
</tr>
<tr>
<td>23</td>
<td>0.0137</td>
<td>0.0449</td>
<td>0.0752</td>
<td>0.1046</td>
<td>0.1366</td>
<td>0.1669</td>
<td>0.1962</td>
</tr>
<tr>
<td>24</td>
<td>0.0111</td>
<td>0.0401</td>
<td>0.0681</td>
<td>0.0954</td>
<td>0.1252</td>
<td>0.1532</td>
<td>0.1805</td>
</tr>
<tr>
<td>25</td>
<td>0.0093</td>
<td>0.0386</td>
<td>0.0527</td>
<td>0.0883</td>
<td>0.1162</td>
<td>0.1420</td>
<td>0.1673</td>
</tr>
<tr>
<td>26</td>
<td>0.0074</td>
<td>0.0327</td>
<td>0.0571</td>
<td>0.0910</td>
<td>0.1271</td>
<td>0.1514</td>
<td>0.1750</td>
</tr>
<tr>
<td>27</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>28</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>29</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>30</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>31</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>32</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Ground Truth Labels Transformed to Label Mask

Project to 2D
Create Network Architecture

Easy API to create network structure

```
build encoder
  noutputs = 64;
  inputlayerName = 'init_maxpool';
  for blockInd = 1:encoderDepth
    [lgraph, layerOutName] = encoderBlock(lgraph, blockInd, noutputs, inputlayerName);
    noutputs = noutputs * 2;
    inputlayerName = layerOutName;
  end

build decoder
  nInputs = nOutputs;
  inputlayerName = layerOutName;
  for blockInd = encoderDepth:-1:1
    nInputs = maxInputs/2, 64);
    [lgraph, decoderLayerOutName] = decoderBlock(lgraph, blockInd, nInputs, nOutputs, inputlayerName);
    if blockInd == 1
      inputLayerName = [ 'res_add' num2str(blockInd) ];
      lgraph = addLayer(lgraph, additionLayer(1, 'Name', inputLayerName ));
      lgraph = connectLayers(lgraph, [ 'add' num2str(blockInd-1) 'addout' ], [inputLayerName 'in2' ]);
      lgraph = connectLayers(lgraph, decoderLayerOutName, inputLayerName 'in1' );
    end
    nInputs = nInputs/2;
  end
```
Deployment using GPU Coder

- C++/CUDA
- TensorRT
- C++/CUDA + cuDNN
Deep learning workflow in MATLAB

- **DNN design + training**
  - Caffe
  - Keras
  - TensorFlow
- **Model importer**
- **Train in MATLAB**
- **Trained DNN**
- **Application design**
- **Standalone Deployment**
  - C++/CUDA + TensorRT
  - C++/CUDA + cuDNN
- **GPU Coder**
Check Out Deep Learning in MATLAB and GPU Coder

Deep learning in MATLAB

Deep learning On-Ramp: Free self-paced, online training

GPU Coder
https://www.mathworks.com/products/gpu-coder.html