GPU-Accelerated Recurrent Neural Networks  
*OpenCLLink and SymbolicC*

**Zdenek Buk**  
bukz1@fel.cvut.cz  
Computational Intelligence Group  
Department of Computer Science  
Faculty of Electrical Engineering,  
Czech Technical University in Prague

---

**Abstract**

The paper presents application of OpenCLLink in Wolfram Mathematica to accelerate fully recurrent neural networks using GPU. We also show the idea of automatically generated parts of source code using SymbolicC.

**Introduction**

The general purpose graphics processing units (GPU) represent a modern, powerful and cheap parallel platform. Since the neural networks consist of many relatively simple neurons - computation units, it seems natural to use the GPU multicore architecture to accelerate the evaluation and learning of neural networks. There are many approaches to the acceleration - the core question is, how the network or group of networks will be mapped to the GPU structure. We have chosen the very intuitive version where single thread (kernel) represents one neuron in the network. We have used the fully recurrent neural network (FRNN) with the realtime recurrent learning algorithm (RTRL).

There are two main platforms for GPU programming - CUDA which can be used with nvidia GPUs and OpenCL which represent the multi platform technology. Both of these platforms are available in Wolfram Mathematica through OpenCLLink or CUDALink. To easier porting the code between this two platforms we will show the usage of SymbolicC package in Wolfram Mathematica.

**Neural Networks**

Neural networks consist of many interconnected neurons. It can be differentiate them by information representation and processing as spiking networks or so-called traditional neural networks. It can be differentiated also by topology as feed-forward or recurrent networks.

The following figure shows a typical neuron with multiple inputs and single output. Each neuron has an activation function which transforms the neuron’s potential to the output value.

![Model of single neuron with multiple inputs and single output value.](image)

\[ y = f \left( \sum_{j=1}^{n} w_j x_j + \Theta \right) = f \left( \sum_{j=0}^{n} w_j x_j \right), \text{ where } x_0 = 1 \text{ and } w_0 = \Theta \]  

(1)

The well known feed-forward topology is shown in the following image. The network is divided into particular layers. There is a complete connection between the layers.

![Example of feed-forward neural network with three external inputs two hidden layers and one output layer with single neuron.](image)

The feed-forward kind of neural network is not able to handle time context of processed data, the output depends only on current combination of input values. The extension of such network with the backward connections leads to recurrent neural networks. The figure below shows an example of so-called Jordan-Elman neural network with explicit recurrent layer representing the state of the system. Output of
this network depends not only on the current configuration of input values but also on the previous states of the network (previous input values transformed by hidden layers).

Example of recurrent neural network - Jordan-Elman network, there are two “state” neurons which handle the previous outputs from the hidden layer.

**GPU Accelerated Computing**

GPUs are primarily oriented to parallel data manipulation - this is mainly derived from the graphics rendering based on operations applied to huge amount of graphics data elements (pixels, vertices, etc.). Serial processing became inefficient with increasing number of these elements. The standard CPUs handle many different tasks at the “same” time, so it’s optimized for this purpose (caching, context switching, etc.). Even on many-core CPUs all tasks are not executed simultaneously, but each task has its amount of processor time and when this time is spent the operating system switches the context to another task.

In contrast, GPU handles just one task at the time, but executed many times in parallel. It is not possible to execute more different programs (tasks) on the same GPU at the same time. The data-parallel processing model maps data elements to threads (working items/units).

*Thread* is running binary code (the functionality is described by *Kernel*) executed many times in parallel. To map threads on the data we need to organize the threads somehow. The GPU memory is exposed to programmer as a linear array of elements. Each thread has an automatically generated identification used for memory object identification. For easier mapping the threads are organized into multi-dimensional virtual grid of threads.

![OpenCL Execution Model](image-source: The OpenCL Specification v1.2, Khronos OpenCL Working Group, 2012)

NVidia CUDA technology uses five-dimensional structure - for better orientation it is divided into three-dimensional blocks of threads and two-dimensional grid of blocks. The OpenCL organization is similar but uses different terminology, see the following table.

<table>
<thead>
<tr>
<th>CUDA Terminology</th>
<th>OpenCL Terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>Index space</td>
</tr>
<tr>
<td>Block</td>
<td>Work-group</td>
</tr>
<tr>
<td>Thread</td>
<td>Work-item</td>
</tr>
</tbody>
</table>

The unique thread identification is evaluated from coordinates in the block and coordinates within the grid. The block and grid dimensions have to be set up before executing the code.

In OpenCL the id of current kernel can be calculated as:

```c
get_global_id(0)
```

in CUDA as:
In "Implementation" section we will show, how to generate such parts of code using SymbolicC automatically dependent on particular technology used.

**Methods**

**Fully Recurrent Neural Networks**

We have chosen the fully recurrent neural networks as the most general topology. The network is represented as a complete symmetric directed graph. See the example of FRNN in the figure below.

![Fully Recurrent Neural Network](image)

A simple fully recurrent neural network example. The network contains two fully interconnected neurons and two external inputs. Each neuron contains an activation function.

For training such networks we are using the realtime recurrent learning algorithm described by Williams [A]. The core is the calculation of \( p \) variables - this is the most computationally expensive part of the algorithm:

\[
    p_i(t + 1) = f_i(s_i(t)) \sum_{j \in U} w_{ij} p_j(t) + \delta_{ik} z_i(t),
\]

where \( \delta_{ik} \) denotes the Kronecker delta.

In our implementation and experiment we use the sigmoid activation function.

\[
    f_i(x) = \frac{1}{1 + e^{-x}},
\]

\[
    s_i(t) = \sum_{j \in U \cup I} w_{ij} z_j(t),
\]

\[
    z_i(t) = \begin{cases} 
    x_i(t) & \text{if } k \in I \\
    y_i(t) & \text{if } k \in U 
    \end{cases}
\]

**OpenCLLink**

"OpenCLLink allows the Wolfram Language to use the OpenCL parallel computing language. It contains functions that facilitate loading user-defined OpenCL functions into the Wolfram Language. OpenCLLink also integrates OpenCL with existing Wolfram Language development tools, allowing a high degree of automation and control." [OpenCLLink Package Overview / Wolfram Documentation]

\( \text{Needs}[\text{"OpenCLLink"} ] \)

If the OpenCL is available on our system, we can easily load and execute the kernel without needs of memory allocation and other programming overheads.

\( \text{Needs}[\text{OpenCL}] \)

\( \text{Out}[2]= \text{True} \)

The following example shows a code which increment all values in given list in parallel.
Applying Wolfram Language manipulation code.

Calling the `incFun` function will automatically start all the kernels, and applies `incVec` code to all values in the list in parallel. Please note the `index` value which represent the “position” of the particular kernel in the block/data.

SymbolicC

“The Wolfram Language’s core tree-oriented symbolic structure makes it well suited to working with a hierarchical view of C code as Wolfram Language expressions. This supports the use of the Wolfram Language for the creation, manipulation, and optimization of C code. It is used extensively for the Wolfram Language’s code generation tools. In addition, you can use SymbolicC for your own code manipulation purposes.” [SymbolicC User Guide / Wolfram Documentation]

Here is an example of the symbolic representation of simple C function.

```plaintext
Needs["SymbolicC"]
code = CFun[fun, int, name, {}, {CAssign[a, b], CReturn[COperator[Plus, {a, b}]]}] ];
```

We can easily generate the plain C code from this symbolic representation or do some symbolic changes.

```plaintext
ToCCodeString[code]
int name() 
{ 
  a = b; 
  return a + b; 
}
```

Applying the rewriting rules we are able to change the code easily:
ToCCodeString[code /.{int -> float, Plus -> Times}]

float name()
{
    a = b;
    return a + b;
}

It is also possible to symbolically represent the OpenCL programs/kernel.

\begin{verbatim}
In[24]:= src = SymbolicOpenCLFunction["incVec",
        {{CPointerType["__global", mint]}, "in"}, {mint, "length"}],
        CBlock[
            CDeclare[int, CAssign["index", SymbolicOpenCLCalculateKernelIndex[1]]],
            CIf[COMperator[Less, {"index", "length"}],
                CAssign[CArray["in", "index"], COMperator[Plus, {CArray["in", "index"], 1}]]
            ]]
        ] // ToCCodeString

Out[24]= ___kernel void incVec(__global mint* in, mint length)
        {
            int index = get_global_id(0);
            if (index < length)
                in[index] = in[index] + 1;
        }

In[26]:= incFun2 = OpenCLFunctionLoad[src, "incVec", {{_Integer}, _Integer}, 256]

Out[26]= OpenCLFunction[<>, incVec, {{_Integer}, Integer64}]

In[27]:= incFun2[[1, 2, 3, 4], 4]

Out[27]= [[2, 3, 4, 5]]
\end{verbatim}

**Implementation**

**Data Structures**

The fully recurrent neural networks (see the example below) are represented by weight matrix. During the training process the particular weights are adapted by a learning algorithm. The figure below shows the mapping of the particular weight into the matrix. The green values shows the weight of input values, the blue weights represent the internal connections between neurons and red values represent the bias.

![Diagram of neural network](image)

An example of Fully Recurrent Neural Networks with 3 neurons and 2 external inputs. The figure shows also the mapping of particular weights into the weight matrix as used in our application. "1" represents the virtual input with constant value 1 used for the bias implementation.

We are calculating each neuron in separate thread, so the mapping to the kernel ID is obvious - each line is processed by one thread and we just multiply the global ID by number of columns of the weight matrix.

```
   w_offset = get_global_id(0) * (num_of_neurons + num_of_inputs + 1);
```
Except the weight matrix the FRNN and RTRL algorithm uses the three-dimensional array $p_{ij}^k$, where $i$ and $k$ go through all neurons (1 to $n$) and $j$ goes through all the neurons and external inputs including the bias represented by constant “virtual input” (1 to $n+m+1$). We have used the following mapping:

```c
// k = get_global_id(0)
p_offset = get_global_id(0) * (num_of_neurons + num_of_inputs + 1) * num_of_neurons;
```

### Context Handling

The recurrent neural networks work in time context, which is the main benefit of this kind of networks. We would like to use the neural network as a function in Mathematica. In pure functional approach we need to handle the state in the parameter of the evaluation function.

The complicated approach is shown in the code below, see the explicit context represented by the $y_{t}$ parameter or the context represented by the network object $net$.

```plaintext
// Simplified version - the idea *
y_{t.1} = netEval[net, y_0, x];
y_{t.2} = netEval[net, y_{t-1}, x];

/* "Full" version - consider that the network has more complex state */
{y_{t.1}, net_{t-1}} = netEval[net, x];
{y_{t.2}, net_{t-2}} = netEval[net_{t-1}, x];
```

Functional-based ”Context handling”.

In the following code the context is stored in the $net$ structure.

```plaintext
net = netInit[...];

(* Desired way to use the neural network function. The context is hidden and does not need to be explicitly specified. *)

net = netInit[...];
y_{t.1} = net[x];
y_{t.2} = net[x];
```

The neural network itself behaves as a evaluation function.

In reference CPU implementation we have used the TagSet to define the $netParameter$ function which is used to access all the parameters of neural network by unique ID.

```plaintext
netId = Unique[FRNNNet];
net = {netId, weight matrix, other parameters...};

Evaluate[netId] /: netParameters[netId] := net;
Apply[FRNNNet, Take[net,1]] (* Return only the id of network -- [[1]] *)
```

The network is represented just as ID. And the evaluation function is defined as follows:

```plaintext
FRNNNet[model___][xx_, opts___?OptionQ] := Module [

...,
netId = FRNNNet[model][[1]];(* net contains all the parameters like weight matrix, etc. *)
...
```

With this approach we are able to create many independent neural networks at the same time.

In OpenCL implementation the situation is easier in the way how we work with the global data - all the parameters are allocated in the GPU memory and Mathematica works just with the references to this memory. Here are examples of FRNN structures in OpenCL implementation:
We have shown some examples of SymbolicC package above. Here is the code to generate the index of current kernel in OpenCL and CUDA:

```wolfram
Needs["OpenCLLink"]
SymbolicOpenCLCalculateKernelIndex[1] // ToCCodeString
```

```wolfram
Needs["CUDALink"]
SymbolicCUDA CalculateKernelIndex[1] // ToCCodeString
```

Our kernel which evaluates the neuron in FRNN works with the weight matrix, \( p \) array, outputs (previous outputs are needed to calculate the current outputs), input values \( x \), and number of neurons and external inputs. \( y \) and \( p \) are allocated as two arrays each - the previous one and the current one - we can’t modify the \( y \) array directly from the kernel, because of the parallel evaluation of neurons - all neurons needs to have consistent input values (those are based on previous outputs). After the parallel evaluation the values from \( y \) are copied into \( \text{prev}_{-y} \). Evaluation of \( p \) array uses the same approach.
SymbolicOpenCLFunction["net_eval",
{
{CPointerType["__global", "Real_t"], "w"},
{CPointerType["__global", "Real_t"], "prev_p"},
{CPointerType["__global", "Real_t"], "prev_y"},
{CPointerType["__global", "Real_t"], "x"},
{mint", "n"},
{mint", "m"},
{CPointerType["__global", "Real_t"], "p"},
{CPointerType["__global", "Real_t"], "y"}
}
] // ToCCodeString

SymbolicCUDAFunction["net_eval",
{
{CPointerType["Real_t"], "w"},
{CPointerType["Real_t"], "prev_p"},
{CPointerType["Real_t"], "prev_y"},
{CPointerType["Real_t"], "x"},
{mint", "n"},
{mint", "m"},
{CPointerType["Real_t"], "p"},
{CPointerType["Real_t"], "y"}
}
] // ToCCodeString

SymbolicOpenCLFunction["net_eval",
{
{CPointerType["__global", type_], var_} -> {CPointerType["__global", type]], var}
}

Using this representations we are able to create the platform (OpenCL or CUDA) independent code:

gpuFunction["net_eval",
{
{CPointerType["Real_t"], "w"},
{CPointerType["Real_t"], "prev_p"},
{CPointerType["Real_t"], "prev_y"},
{CPointerType["Real_t"], "x"},
{mint", "n"},
{mint", "m"},
{CPointerType["Real_t"], "p"},
{CPointerType["Real_t"], "y"}
}
] // ToCCodeString

SymbolicCUDAFunction["net_eval",
{
{CPointerType["__global", type_], var_} -> {CPointerType["__global", type]], var}
}

ClearAll[gpuFunction]

gpuFunction["CUDAKernel", x__] := SymbolicCUDAFunction[x]
gpuFunction["OpenCLKernel", x__] := SymbolicOpenCLFunction[x] /. 
{CPointerType["__global", type_], var_} -> {CPointerType["__global", type]], var}
gpuFunction[x__] := gpuFunction["OpenCLKernel", x]

gpuFunction["net_eval",
{
{CPointerType["Real_t"], "w"},
{CPointerType["Real_t"], "prev_p"},
{CPointerType["Real_t"], "prev_y"},
{CPointerType["Real_t"], "x"},
{mint", "n"},
{mint", "m"},
{CPointerType["Real_t"], "p"},
{CPointerType["Real_t"], "y"}
}
] // ToCCodeString
We have tested the code for selected input values and weight matrices and compared the results (outputs and $p$ matrix) with the reference implementation in Mathematica. We have also compared the performance of both implementations.

**CPU vs. GPU**

To compare the CPU and GPU implementation, we are using our reference implementation of FRNN and RTRL in Mathematica (available online: http://evolution.felk.cvut.cz/recurrent/).

```mathematica
Needs["RecurrentNetworks"]
```

Initialization of the network with 2 neurons and 1 external input. The weight matrix in initialized randomly by default.

```mathematica
cpuNet = InitializeFRNNet[{{1}}, {{1}}, 2];
```

Evaluate the network response for constant input.

```mathematica
Table[cpuNet[{{0.1}}, {5}]
    {0.492178, 0.516173}, {0.492501, 0.533771},
    {0.492405, 0.533971}, {0.492403, 0.533971}, {0.492403, 0.533971}]
```

We will measure time to evaluate the response of the network which includes calculation of the derivatives and the $p[i]$ values, which takes most of the time in RTRL. The evaluation in done for number of neurons from 1 to 32 and each neural network is evaluated in 10 steps. The final time is an average of theses 10 values.

```mathematica
nSteps = 10;
AbsoluteTiming[
cpuResults = Table[
cpuNet = InitializeFRNNet[{{1}}, {{2}}, neurons];
{neurons,
    First[AbsoluteTiming[Table[cpuNet[{}], {nSteps}]]] / nSteps}
],
{neurons, 1, 32}]
]
{280.941156, Null}
```

We do the same measurements for OpenCL implementation of the neural network.

```mathematica
gpuNet = OpenCLFRNNInitialize[2, 1, 1];
```
res = OpenCLFRNNEvaluate[gpuNet]
OpenCLMemory[<738019847>, Float]

After evaluating the network all the parameters and also the output values are updated in the data structures allocated in the GPU memory. We can access this memory by OpenCLMemoryGet function.

OpenCLMemoryGet[res]
{0.53743, 0.53743}

Free the allocated GPU memory:

OpenCLFRNNUnload[gpuNet]

Performance measurement of the OpenCL implementation.

nSteps = 10;
Module[{tmp},
  AbsoluteTiming[
    gpuResults = Table[
      gpuNet = OpenCLFRNNInitialize[neurons, 1, 1];
      tmp = {neurons, First[AbsoluteTiming[Table[OpenCLFRNNEvaluate[gpuNet], {nSteps}]]] / nSteps};
      OpenCLFRNNUnload[gpuNet];
      tmp,
      {neurons, 1, 64}
    ];
  ]
]
{97.349304, Null}

For very low number of neurons we can see that there is some significant overhead of the OpenCL implementation (allocation of the memory, copying from host memory to GPU memory, etc.). The graph below shows the detail of the performance comparison for 1 to 8 neurons and we can see that the performance of the OpenCL implementation is worse in most of these cases. From network size higher than 7 or 8 neurons, the performance of the OpenCL implementation is better than the CPU version and the overhead is negligible.
Conclusion

We have shown the accelerated version of fully recurrent neural network implemented using the OpenCLLink technology in Wolfram Mathematica. We have compared the performance with the reference CPU implementation. The OpenCLLink allows us to easily implement the core of the problem, without worrying about the system-level issues and we can focus mainly on the kernel code. In combination with the SymbolicC package, we are able to write easily portable code or even use the completely automatically generated code.

References


Appendix

Here is an example of the generated OpenCL kernel for FRNN evaluation:

```c
__kernel void net_eval(
    __global Real_t *w,
    __global Real_t *prev_p,
    __global Real_t *prev_y,
    __global Real_t *x,
    mint n,
    mint m,
    __global Real_t *p,
    __global Real_t *y
) {

    const mint k = get_global_id(0);
    const mint w_offset = k * (n + m + l);
    const mint p_offset = k * (n + m + l) * n;

    __local Real_t s;
    __local Real_t sum;
    __local Real_t z;

    if (k < n) {
        s = 0.0f;
        for (int j = 0; j < n+m+1; j++) {
            if (j < n) {
                // Process previous outputs
                s += w[w_offset + j] * prev_y[j];
            }
            if (j == n) {
                // Thresholds
                s += w[w_offset + j];
            }
            if (j > n) {
                // External inputs
                s += w[w_offset + j] * x[j - n - 1];
            }
        }
    }
    __syncthreads();
    sum += s;
    s = 0.0f;
    for (int j = 0; j < m; j++) {
        // Receive inputs
        s += w[w_offset + j] * x[j];
    }
    z = s;
    __syncthreads();
    sum += z;
    // Output
    for (int j = 0; j < n; j++) {
        y[j] = sum;
    }
}
```
void net_eval(__global Real_t* w, __global Real_t* prev_p, __global Real_t* prev_y, __global Real_t* x, mint n, mint m, __global Real_t* p, __global Real_t* y)
{
    const mint k = get_global_id(0);
    const mint w_offset = k * (n + m + 1);
    const mint p_offset = k * (n + m + 1) * n;

    __local Real_t s; __local Real_t sum; __local Real_t z;

    if (k < n)
    {
        s = 0.0f;
        for (int j = 0; j < m + n + 1; j++)
        {
            if (j < n)
            {
                // Process previous outputs
                s += w[w_offset + j] * prev_y[j];
            }
            if (j == n)
            {
                // Thresholds
                s += w[w_offset + j];
            }
            if (j > n)
            {
                // External inputs
                s += w[w_offset + j] * x[j - n - 1];
            }
        }
        y[k] = (1.0f / (1.0f + exp(-s)));
    }
    else
    {
        sum = 0.0f;
        for (int l = 0; l < n; l++)
        {
            sum += w[l] * prev_p[(l + i) * (n + m + 1) + l * n + j];
        }
        if (i == k)
        {
            if (j < n) {
                z = prev_y[j];
            }
            if (j == n) {
                z = 1.0f;
            }
            if (j > n) {
                z = x[j - n - 1];
            }
            sum += z;
        }
        p[p_offset + i * (n + m + 1) + j] = y[k] * (1 - y[k]) * sum;
    }
}

barrier(CLK_GLOBAL_MEM_FENCE);
prev_y[k] = y[k];
}