OPTIMIZED GPU KERNELS FOR DEEP LEARNING

Amir Khosrowshahi

GTC
17 Mar 2015
Outline

• About nervana
• Optimizing deep learning at assembler level
• Limited precision for deep learning
• neon benchmarks
About nervana

• A platform for machine intelligence
• enable deep learning at scale
• optimized from algorithms to silicon
Verticals

Medical

Finance

Pharma

Oil&Gas

Agriculture
• Deep learning supplanting traditional approaches everywhere
• Small improvements have large impact
• Customers require clear roadmap that scales to growing need.
nervana platform for deep learning

Data → nervana framework → nervana cloud → Solutions

- train
- explore
- deploy
nervana platform for deep learning

Data → nervana framework → nervana cloud → Solutions

- train
- explore
- deploy

- GPUs
- CPUs
- nervana engine
maxas: a Maxwell Assembler

- Full control of:
  - register allocation
  - instruction ordering
  - control codes
  - barriers, stall counts
- Built-in scheduler (optional)
- Meta-programming
maxas: a Maxwell Assembler

• Full control of:
  • register allocation
  • instruction ordering
  • control codes
    • barriers, stall counts
  • Built-in scheduler (optional)
  • Meta-programming
maxas: a Maxwell Assembler

• Full control of:
  • register allocation
  • instruction ordering
  • control codes
  • barriers, stall counts
• Built-in scheduler (optional)
• Meta-programming

See GitHub repo for docs and examples
ptxas struggles with Instruction Level Parallelism

**Distribution of Number of Instructions Between LDS and Dependant FFMA Operands**

- **Bad**
- **Good**

![Graph showing distribution of number of instructions between LDS and dependent FFMA operands.](chart)

**FFMA Line# - LDS Line#**

Count

- **ptx**
- **cublas**

courtesy Scott Gray
Easy register allocation through **maxas**

```xml
<REGISTER_MAPPING>
  0-63 ~ blk, ldx, ldx4, k, tid1, tid2, tid15, tid15_4, xmad_t0, xmad_end
  3, 2,11,10,19,18,27,26 : cx00y<00-03|32-35>
  7, 6,15,14,23,22,31,30 : cx01y<00-03|32-35>
  1, 0, 9, 8,17,16,25,24 : cx02y<00-03|32-35>
  5, 4,13,12,21,20,29,28 : cx03y<00-03|32-35>
  35,34,43,42,51,50,59,58 : cx32y<00-03|32-35>
  39,38,47,46,55,54,63,62 : cx33y<00-03|32-35>
  33,32,41,40,49,48,57,56 : cx34y<00-03|32-35>
  37,36,45,44,53,52,61,60 : cx35y<00-03|32-35>
  ...
  64-79 : j0Ax<00-03|32-35>, j0By<00-03|32-35>
  80-95 : j1Ax<00-03|32-35>, j1By<00-03|32-35>
  96-111 : loadX0<0-3>, loadX2<0-3>, loadX4<0-3>, loadX6<0-3>
</REGISTER_MAPPING>
```

Register banking for outer products

\[ c = a \, b^t \]
Example GEMM code in maxas

```
259 01:--:--:0  FFMA cx0y2, j0Bx0, j0Ay2, cx0y2;
260 --:1:--:1  LDS.U.128 j1Ay0, [readAs + 4x<1*64 + 00>];
261 --:--:--:1  FFMA cx1y2, j0Bx1, j0Ay2, cx1y2;
262 --:--:--:0  FFMA cx1y0, j0Bx1, j0Ay0, cx1y0;
263 --:--:1:1  LDS.U.128 j1Bx0, [readBs + 4x<1*64 + 00>];
264 --:--:--:1  FFMA cx0y0, j0Bx0, j0Ay0, cx0y0;
265 --:--:--:0  FFMA cx0y3, j0Bx0, j0Ay3, cx0y3;
266 --:--:1:1  LDS.U.128 j1Ay4, [readAs + 4x<1*64 + 32>];
267 --:--:--:1  FFMA cx1y3, j0Bx1, j0Ay3, cx1y3;
268 --:--:--:0  FFMA cx1y1, j0Bx1, j0Ay1, cx1y1;
269 --:--:1:1  LDS.U.128 j1Bx4, [readBs + 4x<1*64 + 32>];
270 --:--:--:1  FFMA cx0y1, j0Bx0, j0Ay1, cx0y1;
271 --:--:--:1  FFMA cx0y6, j0Bx0, j0Ay6, cx0y6;
272 --:--:--:1  FFMA cx1y6, j0Bx1, j0Ay6, cx1y6;
273 ....
274
275 01:--:--:0  FFMA cx0y2, j1Bx0, j1Ay2, cx0y2;
276 --:1:--:1  LDS.U.128 j0Ay0, [readAs + 4x<2*64 + 00>];
277 --:--:--:1  FFMA cx1y2, j1Bx1, j1Ay2, cx1y2;
278 --:--:--:0  FFMA cx1y0, j1Bx1, j1Ay0, cx1y0;
```
Example GEMM code in maxas

```c
259 01:---:---:0  FFMA cx0y2, j0Bx0, j0Ay2, cx0y2;
260  ---:1:---:1  **LDS.U.128** j1Ay0, [readAs + 4x<1*64 + 00>];
261  ---:---:---:1  FFMA cx1y2, j0Bx1, j0Ay2, cx1y2;
262  ---:---:---:0  FFMA cx1y0, j0Bx1, j0Ay0, cx1y0;
263  ---:1:---:1  **LDS.U.128** j1Bx0, [readBs + 4x<1*64 + 00>];
264  ---:---:---:1  FFMA cx0y0, j0Bx0, j0Ay0, cx0y0;
265  ---:---:---:0  FFMA cx0y3, j0Bx0, j0Ay3, cx0y3;
266  ---:1:---:1  **LDS.U.128** j1Ay4, [readAs + 4x<1*64 + 32>];
267  ---:---:---:1  FFMA cx1y3, j0Bx1, j0Ay3, cx1y3;
268  ---:---:---:0  FFMA cx1y1, j0Bx1, j0Ay1, cx1y1;
269  ---:1:---:1  **LDS.U.128** j1Bx4, [readBs + 4x<1*64 + 32>];
270  ---:---:---:1  FFMA cx0y1, j0Bx0, j0Ay1, cx0y1;
271  ---:---:---:1  FFMA cx0y6, j0Bx0, j0Ay6, cx0y6;
272  ---:---:---:1  FFMA cx1y6, j0Bx1, j0Ay6, cx1y6;
273 ....
274
275 01:---:---:0  FFMA cx0y2, j1Bx0, j1Ay2, cx0y2;
276  ---:1:---:1  **LDS.U.128** j0Ay0, [readAs + 4x<2*64 + 00>];
277  ---:---:---:1  FFMA cx1y2, j1Bx1, j1Ay2, cx1y2;
278  ---:---:---:0  FFMA cx1y0, j1Bx1, j1Ay0, cx1y0;
```
Example GEMM code in maxas

```
259  01:-::-:0  FFMA cx0y2, j0Bx0, j0Ay2, cx0y2;
260  --::1::-:1  LDS.U.128 j1Ay0, [readAs + 4x<1*64 + 00>];
261  --::1::-:1  FFMA cx1y2, j0Bx1, j0Ay2, cx1y2;
262  --::1::-:0  FFMA cx1y0, j0Bx1, j0Ay0, cx1y0;
263  --::1::-:1  LDS.U.128 j1Bx0, [readBs + 4x<1*64 + 00>];
264  --::1::-:1  FFMA cx0y0, j0Bx0, j0Ay0, cx0y0;
265  --::1::-:0  FFMA cx0y3, j0Bx0, j0Ay3, cx0y3;
266  --::1::-:1  LDS.U.128 j1Ay4, [readAs + 4x<1*64 + 32>];
267  --::1::-:1  FFMA cx1y3, j0Bx1, j0Ay3, cx1y3;
268  --::1::-:0  FFMA cx1y1, j0Bx1, j0Ay1, cx1y1;
269  --::1::-:1  LDS.U.128 j1Bx4, [readBs + 4x<1*64 + 32>];
270  --::1::-:1  FFMA cx0y1, j0Bx0, j0Ay1, cx0y1;
271  --::1::-:1  FFMA cx0y6, j0Bx0, j0Ay6, cx0y6;
272  --::1::-:1  FFMA cx1y6, j0Bx1, j0Ay6, cx1y6;
273  ....
274  ....
275  01:-::-:0  FFMA cx0y2, j1Bx0, j1Ay2, cx0y2;
276  --::1::-:1  LDS.U.128 j0Ay0, [readAs + 4x<2*64 + 00>];
277  --::1::-:1  FFMA cx1y2, j1Bx1, j1Ay2, cx1y2;
278  --::1::-:0  FFMA cx1y0, j1Bx1, j1Ay0, cx1y0;
```

Load from shared

Fused fp32 multiply add
Example GEMM code in maxas

```
FFMA cx0y2, j0Bx0, j0Ay2, cx0y2;
LDS.U.128 j1Ay0, [readAs + 4x<1*64 + 00>];
FFMA cx1y2, j0Bx1, j0Ay2, cx1y2;
FFMA cx1y0, j0Bx1, j0Ay0, cx1y0;
LDS.U.128 j1Bx0, [readBs + 4x<1*64 + 00>];
FFMA cx0y0, j0Bx0, j0Ay0, cx0y0;
FFMA cx0y3, j0Bx0, j0Ay3, cx0y3;
LDS.U.128 j1Ay4, [readAs + 4x<1*64 + 32>];
FFMA cx1y3, j0Bx1, j0Ay3, cx1y3;
FFMA cx1y1, j0Bx1, j0Ay1, cx1y1;
LDS.U.128 j1Bx4, [readBs + 4x<1*64 + 32>];
FFMA cx0y1, j0Bx0, j0Ay1, cx0y1;
FFMA cx0y6, j0Bx0, j0Ay6, cx0y6;
FFMA cx1y6, j0Bx1, j0Ay6, cx1y6;
...
```

- **Load from shared**
- **Fused fp32 multiply add**

Control Codes
Example GEMM code in maxas

Dual issue instr.

Load from shared

Control Codes

Fused fp32 multiply add
Example GEMM code in maxas

```
01:---:--:0
-::1:--:1
-:-:-:-:1
---1:---:1
-:-:-:-:0
---1:---:1
-:-:-:-:1
....
```

- **Dual issue instr.**
- **Set barrier**
- **Load from shared**
- **Fused fp32 multiply add**
- **Control Codes**
Example GEMM code in maxas

```
259 01:--:--:0
260 :--:--:1
261 --:--:--:1
262 --:--:0
263 --:1:1
264 --:--:1
265 --:--:1
266 --:--:1
267 --:--:1
268 --:--:1
269 --:1:1
270 --:--:1
271 --:--:1
272 --:--:1
273 ....
274 01:--:--:0
275 --:--:1
276 --:--:1
277 --:--:1
278 --:--:1
```

- Dual issue instr.
- Set barrier
- Barrier sync
- Control Codes
- Load from shared
- Fused fp32 multiply add
Convolution kernels for deep learning

Input: C x H x W
Filters: R x S x K
Output: N x P x Q

<table>
<thead>
<tr>
<th>C</th>
<th>Number of input channels</th>
</tr>
</thead>
<tbody>
<tr>
<td>H x W</td>
<td>Input spatial dims</td>
</tr>
<tr>
<td>R x S</td>
<td>Filter spatial dims</td>
</tr>
<tr>
<td>K</td>
<td>Number of filters</td>
</tr>
<tr>
<td>P x Q</td>
<td>Output spatial dims</td>
</tr>
<tr>
<td>N</td>
<td>Mini-batch dim (not shown)</td>
</tr>
</tbody>
</table>
Access patterns for matrix lowering

• Convolution kernels:
Access patterns for matrix lowering

• Convolution kernels: \textbf{fprop}
Access patterns for matrix lowering

- Convolution kernels: \( \text{bprop} \)

\[
\begin{align*}
\delta^1 & \quad \text{P = Q = 2} \\
\delta^0 & \quad \text{N = 3} \quad \text{C = 3} \quad \text{H = W = 3}
\end{align*}
\]
Access patterns for matrix lowering

- Convolution kernels:

\[ \delta^1 \]

\[ \text{P} = Q = 2 \]
\[ \text{K} = 2 \]

Output of the previous layer

Weight updates

\[ \text{C} = 3 \]
\[ \text{K} = 2 \]
\[ \text{R} = S = 2 \]

\[ \text{N} = 3 \]
\[ \text{C} = 3 \]
\[ \text{H} = W = 3 \]
Deep learning with low precision works
Recent advances in deep learning have made the use of large, deep neural networks with tens of millions of parameters suitable for a number of applications that require real-time processing. The sheer size of these networks can represent a challenging computational burden, even for modern CPUs. For this reason, GPUs are routinely used instead to train and run such networks. This paper is a tutorial for students and researchers on some of the techniques that can be used to reduce this computational cost considerably on modern x86 CPUs. We emphasize data layout, batching of the computation, the use of SSE2 instructions, and particularly leverage SSSE3 and SSE4 fixed-point instructions which provide a $3 \times$ improvement over an optimized floating-point baseline. We use speech recognition as an example task, and show that a real-time hybrid hidden Markov model / neural network (HMM/NN) large vocabulary system can be built with a $10 \times$ speedup over an unoptimized baseline and a $4 \times$ speedup over an aggressively optimized floating-point baseline at no cost in accuracy. The techniques described extend readily to neural network training and provide an effective alternative to the use of specialized hardware.

1 Introduction

The recent resurgence of interest in neural networks owes a certain debt to the availability of affordable, powerful GPUs which routinely speed up common operations such as large matrix computations by factors from $5 \times$ to $50 \times$ [1-3]. These enabled researchers to tackle much larger, more difficult machine learning tasks using neural networks, auto-encoders or deep belief networks [4-6]. Due to a variety of factors, including cost, component reliability and programming complexity, GPUs are still however the exception rather than the norm in computing clusters. The question then becomes whether to invest in GPU resources, or whether traditional CPUs can be made to perform fast enough that, using distributed computing, they will yield similar or superior scalability and performance. The purpose of this paper is not to settle this debate, but rather to introduce to neural network researchers some tools which can significantly improve the performance of neural networks on Intel and AMD CPUs in accessible form. Some of these might not be novel to researchers well versed in high-performance computing, but they lay the foundation for improvements going beyond what one might obtain using existing optimized BLAS packages. We will show in particular how one can outperform optimized BLAS packages by a factor of 3 using fixed point arithmetic and SSSE3 / SSE4 instructions.
We simulate the training of a set of state of the art neural networks, the Maxout networks (Goodfellow et al., 2013a), on three benchmark datasets: the MNIST, CIFAR10 and SVHN, with three distinct arithmetics: floating point, fixed point and dynamic fixed point. For each of those datasets and for each of those arithmetics, we assess the impact of the precision of the computations on the final error of the training. We find that very low precision computation is sufficient not just for running trained networks but also for training them. For example, almost state-of-the-art results were obtained on most datasets with around 10 bits for computing activations and gradients, and 12 bits for storing updated parameters.
Deep Learning with Limited Numerical Precision

Suyog Gupta
Ankur Agrawal
Kailash Gopalakrishnan
IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
Pritish Narayanan
IBM Almaden Research Center, San Jose, CA 95120

Abstract
Training of large-scale deep neural networks is often constrained by the available computational resources. We study the effect of limited precision data representation and computation on neural network training. Within the context of low-precision fixed-point computations, we observe the rounding scheme to play a crucial role in determining the network’s behavior during training. Our results show that deep networks can be trained using only 16-bit wide fixed-point number representation when using stochastic rounding, and incur little to no degradation in the classification accuracy. We also demonstrate an energy-efficient hardware accelerator that implements low-precision fixed-point arithmetic with stochastic rounding.
neon: nervana python deep learning library
neon: nervana python deep learning library

• User-friendly, extensible, abstracts parallelism
neon: nervana python deep learning library

- User-friendly, extensible, abstracts parallelism
- Support for many deep learning models
neon: nervana python deep learning library

• User-friendly, extensible, abstracts parallelism
• Support for many deep learning models
• Interface to nervana cloud
neon: nervana python deep learning library

• User-friendly, extensible, abstracts parallelism

• Support for many deep learning models

• Interface to nervana cloud

• Supports multiple backends → nervana engine
  GPU cluster
  CPU cluster (eg. Cray XC30)
  Xeon Phi cluster (soon)
neon: nervana python deep learning library

- User-friendly, extensible, abstracts parallelism
- Support for many deep learning models
- Interface to nervana cloud
- Supports multiple backends
- Multiple limited precision options
neon: nervana python deep learning library

- User-friendly, extensible, abstracts parallelism
- Support for many deep learning models
- Interface to nervana cloud
- Supports multiple backends
- Multiple limited precision options
- Optimized for Maxwell at assembler level

nervana engine
- GPU cluster
- CPU cluster (eg. Cray XC30)
- Xeon Phi cluster (soon)
neon: easy model configuration
neon: easy model configuration

- Dataset

```python
dataset: &ds !obj:datasets.MNIST {
    repo_path: '~data',
    sample_pct: 100,
},
```
neon: easy model configuration

- Dataset
- Weight initialization

```python
weight_inits: [
    &wt_init !obj:params.GaussianValGen {
        loc: 0.0,
        scale: 0.01,
        bias_init: 0.0,
    },
    &wt_initb !obj:params.NodeNormalizedValGen {
        scale: 4.0,
        bias_init: 0.0,
    },
],
```
neon: easy model configuration

- Dataset
- Weight initialization
- Learning rule
neon: easy model configuration

- Dataset
- Weight initialization
- Learning rule
- Model layers and cost

```ruby
model: !obj:models.MLP {
  num_epochs: 30,
  batch_size: &bs 100,
  layers: [
    &datalayer !obj:layers.DataLayer {
      name: d0,
      nout: 784,
    },
    !obj:layers.FCLayer {
      name: h0,
      nout: 100,
      lrule_init: *gdm,
      weight_init: *wt_init,
      activation: !obj:transforms.RectLin {},
    },
    &lastlayer !obj:layers.FCLayer {
      name: output,
      nout: 10,
      lrule_init: *gdm,
      weight_init: *wt_initb,
      activation: !obj:transforms.Logistic {},
    },
    &costlayer !obj:layers.CostLayer {
      name: cost,
      ref_layer: *datalayer,
      cost: !obj:transforms.CrossEntropy {},
    },
  ],
}
```
neon experiments in fp16/32
neon experiments in fp16/32

- Use 16-bit floating point (fp16) as memory format
neon experiments in fp16/32

- Use 16-bit floating point (fp16) as memory format
- Multiply-and-adds use fp32
neon experiments in fp16/32

- Use 16-bit floating point (fp16) as memory format
- Multiply-and-adds use fp32
- Kernel support for:

<table>
<thead>
<tr>
<th>GEMM</th>
<th>Stochastic rounding</th>
<th>Dropout / maxout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv {f,b}prop, update</td>
<td>Max pooling</td>
<td>Statistics collection</td>
</tr>
</tbody>
</table>
neon experiments in fp16/32

- Use 16-bit floating point (fp16) as memory format
- Multiply-and-adds use fp32
- Kernel support for:

<table>
<thead>
<tr>
<th>GEMM</th>
<th>Stochastic rounding</th>
<th>Dropout / maxout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv {f,b}prop, update</td>
<td>Max pooling</td>
<td>Statistics collection</td>
</tr>
</tbody>
</table>

- Python element-wise operations auto-compiled into kernels
neon experiments in fp16/32

- Use 16-bit floating point (fp16) as memory format
- Multiply-and-adds use fp32
- Kernel support for:
  
<table>
<thead>
<tr>
<th>GEMM</th>
<th>Stochastic rounding</th>
<th>Dropout / maxout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv {f,b}prop, update</td>
<td>Max pooling</td>
<td>Statistics collection</td>
</tr>
</tbody>
</table>

- Python element-wise operations auto-compiled into kernels
- fp16 accumulations done carefully to minimize errors
neon experiments in fp16/32

- Use 16-bit floating point (fp16) as memory format
- Multiply-and-adds use fp32
- Kernel support for:

<table>
<thead>
<tr>
<th>GEMM</th>
<th>Stochastic rounding</th>
<th>Dropout / maxout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conv {f,b}prop, update</td>
<td>Max pooling</td>
<td>Statistics collection</td>
</tr>
</tbody>
</table>

- Python element-wise operations auto-compiled into kernels
- fp16 accumulations done carefully to minimize errors
- Working with collaborators (Baidu, Bengio lab) to improve
fp16/32 accuracy

• No accuracy loss going from fp32 to fp16
fp16/32 accuracy

• No accuracy loss going from fp32 to fp16
fp16/32 accuracy

- No accuracy loss going from fp32 to fp16
Speed benchmarks¹: fp16 vs others

5 convolutional layers, forward and backward pass

Lower times are better. Benchmarks on GTX980

¹ Soumith Chintala, github.com/soumith/convnet-benchmarks
Speed benchmarks\(^1\): fp16 vs fp32

5 convolutional layers, forward and backward pass
Lower times are better. Benchmarks on GTX980

\(^1\) Soumith Chintala, [github.com/soumith/convnet-benchmarks](https://github.com/soumith/convnet-benchmarks)
Speed benchmarks: fp16 vs fp32

5 convolutional layers, forward and backward pass
Lower times are better. Benchmarks on GTX980

*some layers do not fit on a 4GB card

Soumith Chintala, github.com/soumith/convnet-benchmarks
Speed benchmarks\(^1\): fp16 vs fp32

![Graph of Speed benchmarks](image)

5 convolutional layers, forward and backward pass
Lower times are better. Benchmarks on GTX980

\(^1\) Soumith Chintala, [github.com/soumith/convnet-benchmarks](https://github.com/soumith/convnet-benchmarks)

*some layers do not fit on a 4GB card
Benchmarks\(^1\) show 2x performance

Raw numbers (averaged over 10 runs)

Maximum practical peak is **4700** gflops.

More than **double speed**\(^2\) with **half memory** storage / bandwidth.

---

\(^1\) Using conventions here: Soumith Chintala, [github.com/soumith/convnet-benchmarks](https://github.com/soumith/convnet-benchmarks)

\(^2\) Numbers are relative to Titan Black (Kepler architecture)
Benchmarks\(^1\) show 2x performance

Raw numbers (averaged over 10 runs)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (msecs)</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg(10) fprop:</td>
<td>43.650</td>
<td>4188.573</td>
</tr>
<tr>
<td>Avg(10) bprop:</td>
<td>94.315</td>
<td>3877.055</td>
</tr>
<tr>
<td>Avg(10) total:</td>
<td>137.965</td>
<td>3975.615</td>
</tr>
</tbody>
</table>

Maximum practical peak is 4700 gflops.

More than \textbf{double speed}\(^2\) with \textbf{half memory} storage / bandwidth.

\(^1\) Using conventions here: Soumith Chintala, [github.com/soumith/convnet-benchmarks](https://github.com/soumith/convnet-benchmarks)

\(^2\) Numbers are relative to Titan Black (Kepler architecture)
Benchmarks\(^1\) show 2x performance

Raw numbers (averaged over 10 runs)

<table>
<thead>
<tr>
<th>Type</th>
<th>Time (msecs)</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg(10) fprop:</td>
<td>172.005</td>
<td>4169.400</td>
</tr>
<tr>
<td>Avg(10) bprop:</td>
<td>355.809</td>
<td>4031.144</td>
</tr>
<tr>
<td>Avg(10) total:</td>
<td>527.815</td>
<td>4076.199</td>
</tr>
</tbody>
</table>

Maximum practical peak is 4700 gflops.

More than **double speed**\(^2\) with **half memory** storage / bandwidth.

---

\(^1\) Using conventions here: Soumith Chintala, [github.com/soumith/convnet-benchmarks](https://github.com/soumith/convnet-benchmarks)

\(^2\) Numbers are relative to Titan Black (Kepler architecture)
Benchmarks\(^1\) show 2x performance

### VGG (N=64)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Average Time (ms)</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>fprop</td>
<td>234.050</td>
<td>4161.347</td>
</tr>
<tr>
<td>bprop</td>
<td>529.052</td>
<td>3681.920</td>
</tr>
<tr>
<td>total</td>
<td>763.102</td>
<td>3828.965</td>
</tr>
</tbody>
</table>

Maximum practical peak is **4700** gflops.

More than **double speed**\(^2\) with **half memory** storage / bandwidth.

---

1 Using conventions here: Soumith Chintala, [github.com/soumith/convnet-benchmarks](https://github.com/soumith/convnet-benchmarks)

2 Numbers are relative to Titan Black (Kepler architecture)
Summary
Summary

- **neon**: User-friendly python library
Summary

- **neon**: User-friendly python library
- **maxas**: Powerful tool for optimizing deep learning
Summary

- **neon**: User-friendly python library
- **maxas**: Powerful tool for optimizing deep learning
- Fast performance, full utilization of GPU
Summary

- **neon**: User-friendly python library
- **maxas**: Powerful tool for optimizing deep learning
- Fast performance, full utilization of GPU
- Limited precision allows for larger models
Summary

- **neon**: User-friendly python library
- **maxas**: Powerful tool for optimizing deep learning
- Fast performance, full utilization of GPU
- Limited precision allows for larger models
- Toolbox for exploring numerical representations
Summary

- **neon**: User-friendly python library
- **maxas**: Powerful tool for optimizing deep learning
- Fast performance, full utilization of GPU
- Limited precision allows for larger models
- Toolbox for exploring numerical representations
GTC 2015

- Contact us at gtc@nervanasys.com
- We are hiring!
  - Cloud engineers
  - GPU experts
  - machine learning engineers
  - software engineers

- Sign up to try neon, our deep learning library.
- We can help solve your problem.