Improving the speed of neural networks on CPUs

Vincent Vanhoucke
Google, Inc.
Mountain View, CA 94043
vanhoucke@google.com

Andrew Senior
Google, Inc.
New York, NY 10011
andrewsenior@google.com

Mark Z. Mao
Google, Inc.
Mountain View, CA 94043
markmao@google.com

Abstract

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.

To illustrate the argument, we will use a neural network achieving state-of-the-art performance on the task of speech recognition for mobile voice input. The basic benchmark setup is described in detail in Section 2, while the full speech recognition evaluation system is described in 6. Section 3 will cover the basics of data layout and floating-point SIMD operations. Section 4 describes the fixed-point implementation of some of the matrix operations. Section 5 introduces further enhancements that are more specifically applicable to the speech recognition task. We'll conclude in Section 6 with performance results on the end-to-end system.

2 Benchmark setup

In the following, we will use as a benchmark a neural network with 5 layers. The input layer has 440 inputs consisting of 11 consecutive, overlapping 25 ms frames of speech, sampled every 10 ms. Each frame consists of 40 log-energies extracted from filterbanks on a Mel frequency scale. Each layer but the last uses a sigmoid as non-linearity. Each intermediate layer consists of 2000 nodes, and the final softmax layer has 7969 outputs which correspond to log-posteriors for context-dependent acoustic states of a HMM/NN speech recognizer. During decoding the softmax output is not normalized. Details of the HMM/NN system can be found in [7] and in Section 6.

The evaluation setup consists of running 100 frames of data through the network on a single CPU on an Intel Xeon DP Quad Core E5640 machine with Ubuntu OS. CPU scaling was disabled and each run was performed a minimum of 5 times and averaged. A summary of the results is presented in Table 1. For results relating to lazy evaluation (†), Table 1 reports performance assuming 30% of the neural network outputs actually need to be computed (see Section 5 for details).

Table 1: Summary of the results

	Section	Time to process 1s of speech	Incremental speed-up
Floating-point baseline	2	3.89 s	
Hand-tuned floating-point	3.2	3.09 s	26%
Floating-point SSE2	3.5	1.36 s	127%
Floating-point GPU	3.7	0.02 to 0.49 s	
8-bit quantization	4.1	1.52 s	-12%
Integer SSSE3	4.2	0.51 s	198%
Integer SSE4	4.3	0.47 s	9%
Batching	5.1	0.36 s	30%
Lazy evaluation †	5.2	$\tilde{0}$.26 s	27%
Batched lazy evaluation †	5.3	$\tilde{0}$.21 s	19%

3 Floating-point implementation

3.1 Memory locality

The most basic tenet of high-performance computing is that once you touch a given memory address, nearby memory past this address gets loaded into the various caches that live on the processor die. This makes the nearby data available to the CPU much faster than it would be if it had to fetch it from memory. The most immediate consequence is that one should strive to have the innermost loops of any numerical computation walk contiguous memory. As an example, consider the multiplication of matrix $A = [a_{i,j}]$ and $B = [b_{k,l}]$. Each entry of matrix $C = AB = [c_{i,l}]$ is:

$$c_{i,l} = \sum_m a_{i,m} b_{m,l} \tag{1}$$

Since the loop variable m walks the columns of A and rows of B, A is best stored in *row-major* order, while B is best stored in *column-major* order. This is even more important when taking Single Instruction, Multiple Data (SIMD) instructions into account.

3.2 Loop unrolling and parallel accumulators

There are several well-known ways to improve over a naive implementation of matrix multiplication. The innermost loop of the matrix multiply operation is a *multiply-and-add* operation: `c += a[i]*b[i]`. To reduce the overhead of checking for loop termination, one can partially unroll the computation by accumulating multiple elements at a time:

```
c += a[i]*b[i] + a[i+1]*b[i+1] + a[i+2]*b[i+2] + a[i+3]*b[i+3]
```

A second technique is to use multiple accumulators in parallel, which gives the compiler more freedom in pipelining operations and distributing them across floating-point units:

```
c0 += a[i]*b[i];
c1 += a[i+1]*b[i+1];
c2 += a[i+2]*b[i+2];
c3 += a[i+3]*b[i+3];
c = c0 + c1 + c2 + c3;
```

On our example benchmark (Table 1), unrolling loops in blocks of 8 and using 4 accumulators in parallel yields a 26% speed improvement.

3.3 SIMD

SIMD instructions are the fundamental building blocks of low-level parallelization on CPUs. These instructions perform multiple operations in parallel on contiguous data, making the issues of data locality even more critical. On Intel and AMD CPUs of the x86 family, they typically operate on 16 bytes worth of data at a time: 2 doubles, 4 floats, 8 shorts or 16 bytes at a time. Various datatypes are defined to represent these, `__m128i`, `__m128` and `__m128d`:

`__m128i`	… 128 bits / 16 chars …							
`__m128i`	short	short	short	short	short	short	short	short
`__m128i`	int		int		int		int	
`__m128`	float		float		float		float	
`__m128d`	double				double			

Fundamental operations on these datatypes use assembly instructions, but are more easily incorporated into C and C++ using thin wrapper functions called intrinsics [8], available for the most popular C and C++ compilers. As an example, if you have two vectors of 4 floats stored in two variables $a = [a_1, a_2, a_3, a_4]$ and $b = [b_1, b_2, b_3, b_4]$ of type `__m128`, you can obtain $c = [a_1 + b_1, a_2 + b_2, a_3 + b_3, a_4 + b_4]$ by writing in C:

```
#include <mmintrin.h>
__m128 c = _mm_add_ps(a, b);
```

3.4 Data layout optimization for SIMD

There are 2 main difficulties in laying out data optimally to leverage SIMD instructions.

First, SIMD instructions generally operate faster on 16 byte blocks that are 16-byte aligned in memory. Being '16-byte aligned' means that the memory address of the first byte is a multiple of 16. As a consequence, if an array of data to be processed is not aligned to 16 bytes, performance will suffer greatly. Forcing 16-byte alignment of a memory block can be achieved in C by replacing calls to `malloc()` with `posix_memalign()`, or by using custom allocators if using the Standard Template Library.

Second, since every instruction operates on a block of 16 bytes, if a data vector is not a multiple of 16 bytes in size, one will have to deal with edge effects. The simplest solution is zero-padding: treat every vector of size N as a vector of size $((N + 15)/16) * 16$ (integer arithmetic), with added zeros at the end. For most linear operations, such as scalar product, sum, matrix multiply, the embedding into such larger vector space is invariant and does not affect the results.

In the following, we'll assume every vector and row (resp. column) in row-major (resp. column-major) matrix is aligned and zero-padded.

3.5 Floating-point SIMD and Intel SSE2

Intel and AMD processors that support SSE2 provide the basic instructions to perform the multiply-and-add step using floating-point SIMD arithmetic: `_mm_add_ps()` and `_mm_mul_ps()`. Here is a simplified example accumulating the scalar product of `__m128 *a` and `__m128 *b` to `__m128 sum`:

```
// c[0] = a[0]*b[0], ... , c[3] = a[3]*b[3]
__m128 c = _mm_mul_ps(*a, *b);
// sum[0] += c[0], ..., sum[3] += c[3]
sum = _mm_add_ps(c, sum);
```

`sum` now contains 4 partial sums which have to be added horizontally to yield the final result using a mechanism such as shuffle instructions available with SSE2, or horizontal sums (`_mm_hadd_ps()`) available with SSE3. Most modern compilers are able to leverage SIMD instructions automatically to some extent. However, it is our experience that automatic vectorization does not come close to realizing the performance gains one can achieve by writing code that leverages them explicitly.

3.6 Comparison to optimized BLAS package

As an illustration of how these simple techniques fare in comparison to off-the-shelf fast matrix libraries, we compared the matrix multiplications in our benchmark with Eigen [9]. Eigen is a very fast library which pays particular attention to cache optimization.

Table 2: Comparison with Eigen

Matrix A	Matrix B	$A \times B$ (Eigen 2.0)	$A \times B$ (Eigen 3.0)	$A \times B$ (Custom)
2000×2000	2000×1	6.0 ms	5.6 ms	1.2 ms
	2000×2	5.8 ms	6.5 ms	2.4 ms
	2000×4	5.1 ms	6.1 ms	4.6 ms
	2000×8	5.0 ms	7.7 ms	9.4 ms
	2000×16	8.7 ms	10.7 ms	19.0 ms

Table 2 shows that our implementation is comparable with Eigen in this specific scenario. Eigen is a much more general library however, so these figures should not be construed as a general statement about Eigen's performance. In this particular context, our implementation appears faster for thin B matrices, but doesn't scale as well for larger ones. The nonlinearity of Eigen's scaling as the data size increases suggests that the library uses block heuristics to optimize its cache behavior.

3.7 Comparison to an optimized GPU implementation

We compared our end-to-end neural network implementation described in Section 2 to one based on CUDAMat [3]. The GPU experiment was run on a NVIDIA Tesla C2070 GPU board mounted on the same machine as used for the other benchmarks.

Because of their massive parallelism, GPUs are very well suited to processing data in batches. Table 3 shows that from a 2.8× performance gain without batching, the GPU throughput scales almost linearly with the batch size for small batches. As we will see in Section 5.1, on a CPU batching only improves a comparable benchmark by a small fraction.

Table 3: GPU Implementation

	Batch size	Processing 1s of speech
CPU	1	1360 ms
GPU	1	490 ms
	2	250 ms
	4	125 ms
	8	66 ms
	128	20 ms

4 Fixed-point implementation

There are several properties of neural networks that make them prime candidates for a fixed-point implementation. First, activations are probabilities in the $[0, 1]$ interval, which means that they can be represented as unsigned integers without much concern for scaling. Second, inputs of all the intermediate layers are activations, whose outputs are compressed through a sigmoid. This causes the dynamic range of the weights to remain bounded, making those good candidates for a signed integer representation. Third, because of the linear nature of the operations and the dynamic range compression of the sigmoid, quantization errors tend to propagate sub-linearly and not cause numerical instability.

4.1 Linear quantization

For reasons that will become clear shortly, we used 8-bit quantization to convert activations into `unsigned char`, intermediate layer weights into `signed char`, with the exception of biases that are encoded as 32-bit `int`. One exception is the input layer, which remains floating-point, to better accommodate the potentially larger dynamic ranges of inputs that are not probabilities. In our particular use-case, the input layer is much smaller (440x2000) than any subsequent layer and doesn't weigh strongly on the overall speed. One of the important benefits of quantizing the majority of the network down to 8 bits is that the total memory footprint of the network consequently shrinks by between $3\times$ and $4\times$.

Weights are scaled by taking their maximum magnitude in each layer and normalizing them to fall in the $[-128, 127]$ range. Biases are scaled by the same amount and linearly quantized to 32 bits. The matrix multiplication at each layer produces a 32-bit integer, which a fast, approximate sigmoid implementation then maps to an 8-bit probability.

Note that a reasonably tuned implementation of this fixed-point network, while outperforming by $2\times$ an equivalent floating-point implementation, is still slower than the SSE2 optimized system (Table 1). It has been observed in many other scenarios that fixed-point implementations don't necessarily compete well with floating-point equivalents on modern CPUs. We will see below how to regain a significant edge.

4.2 Intel SSSE3

The Intel SSSE3 instruction set [8] introduces the `pmaddubsw` instruction (with corresponding intrinsic `_mm_maddubs_epi16()`) which perfectly matches our quantized neural network computation. The instruction takes 16 unsigned 8-bit integers (the activations), 16 signed 8-bit integers (the weights), and performs parallel multiply-and-add operations on them to yield 8 16-bit integers. Note that since 16 bits might not be sufficient to hold the sum-product of 2 signed 8-bits and 2 unsigned 8-bits, the results are saturated to 16 bits, which means that the operation can be an approximation of the true multiply-and-add. In practice, this is not an issue for neural networks since large magnitude outputs tend to be compressed via the sigmoid.

On CPUs which can support this instruction, the fixed-point sum-product of `__m128i *u` and `__m128i *s` to be stored in `__m128 sum` can be written as:

```
// c[0] = saturate(u[0]*s[0] + u[1]*s[1]) ...
// c[7] = saturate(u[14]*s[14] + u[15]*s[15])
__m128i c = _mm_maddubs_epi16(u, s);
// unpack the 4 lowest 16-bit integers into 32 bits.
__m128i lo = _mm_srai_epi32(_mm_unpacklo_epi16(c, c), 16);
// Unpack the 4 highest 16-bit integers into 32 bits.
__m128i hi = _mm_srai_epi32(_mm_unpackhi_epi16(c, c), 16);
// Add them to the 4 32-bit integer accumulators.
sum = _mm_add_epi32(_mm_add_epi32(lo, hi), sum);
```

This results in a 3× speedup of the computation, bringing our benchmark well within real-time.

4.3 Intel SSE4

The SSE4.1 instruction set [8] introduces one small optimization with a single instruction for 16 to 32-bit conversion:

```
// c[0] = saturate(u[0]*s[0] + u[1]*s[1]) ...
// c[7] = saturate(u[14]*s[14] + u[15]*s[15])
__m128i c = _mm_maddubs_epi16(u, s);
// unpack the 4 lowest 16-bit integers into 32 bits.
__m128i lo = _mm_cvtepi16_epi32(c)
// Unpack the 4 highest 16-bit integers into 32 bits.
__m128i hi = _mm_cvtepi16_epi32(_mm_shuffle_epi32(c, 0x4e)));
// Add them to the 4 32-bit integer accumulators.
sum = _mm_add_epi32(_mm_add_epi32(lo, hi), sum);
```

This results in a 9% relative speed improvement over SSSE3 on our benchmark.

5 Further task-specific improvements

All performance numbers in Table 4 pertain to the benchmark described in Section 2: computing 100 frames (1 second) of speech.

Table 4: Effect of batching and lazy evaluation.

Evaluation	Section	Active states	Batch size				Best speed-up
			1	2	4	8	
Eager	5.1	⋆	472 ms	409 ms	374 ms	356 ms	25%
Lazy	5.2	1%	257 ms	203 ms	180 ms	169 ms	64%
		10%	298 ms	243 ms	214 ms	199 ms	58%
		30%	371 ms	308 ms	273 ms	255 ms	46%
		50%	436 ms	366 ms	326 ms	307 ms	35%
		100% †	579 ms	499 ms	458 ms	439 ms	7%
Batched-lazy	5.3	1%	261 ms	201 ms	177 ms	166 ms	65%
		10%	299 ms	231 ms	197 ms	181 ms	62%
		30%	380 ms	286 ms	233 ms	212 ms	55%
		50%	454 ms	337 ms	268 ms	241 ms	49%
		100%	617 ms	450 ms	349 ms	311 ms	34%

5.1 Batching

It is interesting to note that with the optimizations from Section 4.3, the CPU implementation slightly outperforms the GPU implementation described in Section 3.7 *in the absence of batching* (See Tables 1 and 3). While for offline applications, GPUs have a very large advantage over CPUs, it appears that this advantage can become negligible when batch processing is not an option. Batching can however further improve memory locality and be beneficial on CPUs as well. This benefit is to be traded off against a possible increase in latency for real-time applications. In streaming speech recognition, it is common to incorporate a look-ahead of a few hundred ms, at least in the beginning of an utterance, to help improve run-time estimates of speech and noise statistics, which makes it possible to process frames in small batches over tens of ms. To take full advantage of batching, the batches have to be propagated through the neural network layers in bulk, so that every linear computation becomes a matrix-matrix multiply which can take advantage of CPU caching of both weights and activations.

5.2 Lazy evaluation

The traditional GMM/HMM model of speech recognition has a computational advantage over hybrid neural network approaches. During decoding, at every frame, only a fraction of the state scores ever need to be computed. Because every state has its own, small, set of Gaussians, only a fraction of the total parameter space has to be visited at every point. On a small sample of a large vocabulary task, we found for example that on average about 25% to 30% of the states were active at any point in time. Assuming that the GMM system requires roughly the same total number of parameters as a hybrid system, this is a significant reduction in the number of arithmetic operations and memory accesses. In addition, there are several well-known Gaussian selection techniques which further help narrow down the pool of Gaussians that need to be evaluated [10,11]. In the case of dense neural networks, in principle every parameter has to be visited at every frame. The notable exception is the last layer, which only needs to be computed for a given state if the state posterior is needed during decoding. This opens up the possibility of lazy evaluation whereby a state posterior is only computed when needed by the decoder. In our example benchmark for instance, a large fraction of the computation is spent evaluating the last layer, which contains a full 55% of all the parameters.

As shown in Table 4 (comparing row $\star$ and row †), evaluating the final layer in a lazy manner adds inefficiency to the matrix computation and hence introduces a small fixed cost of about 22% relative. In this instance, it is however beneficial even if as many as 50% of the output scores need to be computed, leading to a 14% relative improvement over batching alone.

5.3 Batched lazy evaluation

With lazy evaluation we can no longer simply compute batches of output scores for all outputs across multiple frames, but we can continue to batch the computation of all layers but the last. In addition, We can exploit the piece-wise stationary nature of speech signals, which means that if a state is needed by the decoder at frame t, it is very likely to be needed at frame $t + 1$. Computing a batch of these posteriors for consecutive frames at the same time while the weights are in cache thus provides a further efficiency, at the cost of sometimes computing a given state posterior for a frame when it will not be needed. Table 4 shows the effect of this batch size on computing output scores in batches compared to an equal number of scores computed in random order.

6 Speech recognition evaluation

Table 1 shows that, for the neural network computation, the speed improvements are in the vicinity of $10\times$ against a naive baseline, and around $4\times$ against our best floating-point system. We now evaluate our best floating-point and fixed-point systems in the context of a real speech recognition task, with the overhead of the decoder and the cost of real-task memory access patterns that it entails.

The end-to-end system consists of a hidden Markov model, context-dependent speech recognizer. The neural network described in Section 2 provides posterior probabilities for each state, which are then divided by state priors to yield observation likelihoods. The evaluation task consists of 27400 utterances of mobile spoken input such as search queries and short messages, and uses a large N-gram language model tuned to that combination of tasks. The search is performed using a weighted finite-state transducer decoder [12]. The evaluation was conducted on a cluster of machines replicating a real production environment, and was tuned to typical production parameters.

Table 5 shows that no performance cost is incurred by quantizing the neural network parameters, but the end-to-end system runs $3\times$ faster and within real-time constraints. The real-time factor quoted

Table 5: Speed/accuracy results for large vocabulary speech recognition.

	Word error rate	Insertions	Deletions	Substitutions	90^{th} percentile real-time factor
Floating-point	14.9%	3.0%	2.3%	9.6%	2.91
Fixed-point	14.8%	3.1%	2.1%	9.6%	0.90

is the 90^{th} percentile across all utterances of the ratio of time spent decoding an utterance to the duration of the utterance.

7 Conclusion

Optimization of the kind of large matrix computations that are needed to evaluate neural networks on a CPU is a complex topic in an ever-evolving ecosystem of architectures and performance trade-offs. This paper shows that simple techniques can dramatically enhance the performance of neural network-based systems. Of particular interest are recently-introduced fixed-point SIMD operations in x86 processors which once again tilt the balance of performance in favor of fixed-point arithmetic. We showed that leveraging these faster instructions, a real-time speech recognizer can be built using a very large hybrid network at no cost in accuracy.

References

[1] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlun, Ronak Singhal, and Pradeep Dubey (2010) Debunking the 100× GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU, *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA'10, ACM.

[2] Noriyuki Fujimoto (2008) Faster Matrix-Vector Multiplication on GeForce 8800GTX, *Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, LSPP-402, pp. 1–8.

[3] Volodymyr Mnih (2009) CUDAMat: a CUDA-based matrix class for Python, *Technical Report UTML TR 2009-004*, Department of Computer Science, University of Toronto.

[4] Rajat Raina, Anand Madhavan, and Andrew Y. Ng (2009) Large-scale deep unsupervised learning using graphics processors, *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML'09, ACM.

[5] Kyoung-Su Oh and Keechul Jung (2004) GPU implementation of neural networks *Pattern Recognition*, **37**(6):1311-1314.

[6] Honghoon Jang, Anjin Park, and Keechul Jung (2008) Neural Network Implementation using CUDA and OpenMP, *Proceedings of the 2008 Digital Image Computing: Techniques and Applications*, pp 155–161.

[7] Navdeep Jaitly, Patrick Nguyen, and Vincent Vanhoucke (2012) Application of Pretrained Deep Neural Networks to Large Vocabulary Speech Recognition, Submitted to *ICASSP'12*.

[8] Intel C++ Intrinsics Reference, http://cache-www.intel.com/cd/00/00/34/76/347603_347603.pdf

[9] Eigen, a C++ template library for linear algebra, http://eigen.tuxfamily.org/

[10] Jürgen Fritsch and Ivica Rogina (1996) The bucket box intersection (BBI) algorithm for fast approximative evaluation of diagonal mixture Gaussians, *Proceedings of ICASSP'96*.

[11] Kate M. Knill, Mark J.F. Gales, and Steve J. Young (1996) Use of Gaussian selection in large vocabulary continuous speech recognition using HMMs, *Proceedings of ICSLP'96*.

[12] OpenFst Library, http://www.openfst.org