Speech Recognition on Vector Architectures

by

Adam Louis Janin

B.S. (California Institute of Technology) 1990

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Nelson Morgan, Chair
Jerry Feldman
David Wessel
Kathy Yelick

Fall 2004

The dissertation of Adam Louis Janin is approved:

Chair Date

Date

Date

Date

University of California, Berkeley

Fall 2004

Speech Recognition on Vector Architectures

Abstract

Speech Recognition on Vector Architectures

by

Adam Louis Janin
Doctor of Philosophy in Computer Science

University of California, Berkeley

Nelson Morgan, Chair

From cellphones and PDAs to huge automated call centers, speech recognition is becoming more and more ubiquitous. As demand for automatic speech recognition (ASR) applications increases, so too does the need to run ASR algorithms on a variety of unconventional computer architectures. One such architecture uses a vector processor, which has many benefits in terms of performance, power consumption, price, etc.

This thesis presents and evaluates ASR algorithms ported to run efficiently on vector architectures. A vector simulation library was developed, and is used to evaluate design trade-offs for both the algorithms and the hardware. Two of the three major components of an ASR system vectorize well. The third component, the decoder, vectorizes well for small vocabularies, especially on long vector length architectures, but is difficult to vectorize as the vocabulary grows.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This thesis covers algorithms and analysis of a speech recognition system ported to a simulated vector architecture. The approach is to design and analyze algorithms that will run efficiently on vector architectures, rather than to port to a specific system.

1.1 Automatic Speech Recognition

Automatic speech recognition (ASR) is becoming more and more popular on a variety of computer architectures:

Desktop computers: On the desktop, speech recognition is used both for command and control (e.g. "open notepad"), as well as for general dictation tasks (e.g. writing a letter). Providing efficient speech recognition allows a wider range of applications that exploit speech as input.

Supercomputers: On supercomputers and mainframes, speech recognition is used for applications such as automated call centers, where a large number of simultaneous users need to navigate a voice messaging system. Supercomputers are also useful for experimenting with new algorithms — today's supercomputers are tomorrow's desktops.

Consumer electronic devices: Consumer electronic devices, such as PDAs and cellphones, present a whole new set of challenges. Not only is the hardware less powerful than a typical desktop, but also the application domain (hands-free, uncontrolled acoustic environments) stresses the speech recognition algorithms to their utmost. Battery life also is a major factor.

Special purpose hardware: Video cards and game console systems often come equipped with high speed processors. It may be possible to leverage the computational power of these devices to provide ASR services, much as a video card provides graphics services.

Speech recognition applications require a significant amount of processing power, so the systems must contain a processor capable of providing the required computational speed. The problem is especially acute when real time response is required or when multiple simultaneous users must be supported (e.g. call centers). However, power consumption is also a major concern in many applications. Furthermore, since speech recognition systems tend to be large, complex, and frequently changing, a general purpose computing environment is preferable over a specialized implementation. Currently available processors run the gamut from power-hungry x86-based systems, to specialized application-specific integrated circuits (ASICs).

Data parallel architectures, including vector processors, stream processors, and Single Instruction Multiple Data (SIMD) extensions, are emerging as an attractive way to boost performance. The use of parallelism, rather than a high clock rate, keeps power consumption down, and the simplicity of the data parallel execution model avoids expensive chip and power costs that arise from dynamic parallelism discovery in superscalar architectures. Data parallel architectures push the parallelism discovery problem to the compiler and algorithm designer, and while there are many algorithms in scientific computing and media processing that exhibit extensive data parallelism (see Section 1.3), some algorithms do not fit. In this thesis, we explore the use of data parallelism in ASR, using a vector instruction set as our test vehicle.

Although vector processors generally will execute non-vector code, efficiency on such code is lower than with vectorized code. Amdahl's Law [4] tells us that the speedup we can expect on a vector processor will be limited by the amount of non-vector code in the system. The programming challenge is therefore to discover parallelism in as many parts of the application as possible. Software infrastructure, such as hardware-specific math libraries and vectorizing compilers, ease some of these difficulties by providing ready-made vectorized code for some part of the application, but typically the core algorithms must be ported to run efficiently on vector processors.

For this thesis, vectorized algorithms were developed for a particular ASR system using a vector simulation library. For each algorithm, the vectorized versions were compared to an equivalent scalar algorithm. Results in each case were identical, validating the correctness of the vector algorithms. Where relevant, different algorithms were developed to exploit varying architectural features (e.g. vector length).

The remainder of this chapter provides an overview of vector architectures, including some history. Chapter 2 describes the vector simulation library that was designed and implemented as part of this thesis. Chapter 3 presents a high level view of a speech system, and forms a roadmap for the chapters that follow it. Chapters 4, 5, 6, and 7 contain details about the various components of a speech recognition system designed to run on vector architectures. Finally, Chapter 8 contains conclusions and future work.

1.2 Vector Processors

A vector processor implements a type of data parallelism. Instead of registers holding a single value, vector registers hold multiple values. Vector instructions then operate on all the values, conceptually simultaneously. Figure 1.1 compares scalar addition on scalar registers with vector addition on vector registers.

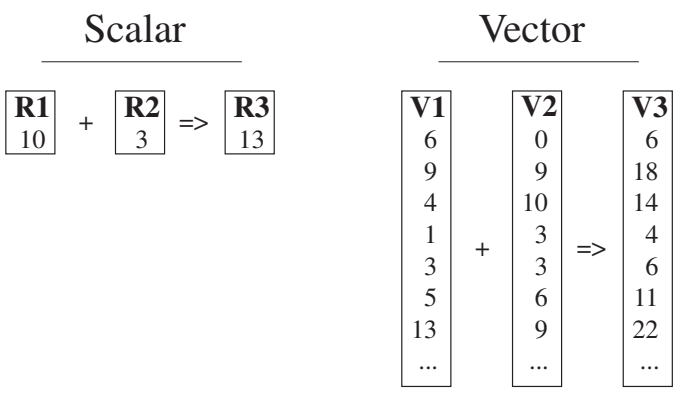

Figure 1.1: Comparison of a scalar operation and a vector operation.

1.2.1 Advantages of Vector Processors

Since the vector instructions explicitly expose data parallelism, it is possible to execute the different elements of a vector operation simultaneously. This improves absolute performance with a modest increase in processor complexity [42]. Furthermore, adding functional units[1] to the chip to allow more vector elements to be executed in parallel can be done fairly easily, as the layout of these additional units is quite regular [35] [7] [39]. The energy consumption can also be made fairly moderate compared with conventional designs.

Another bottleneck on conventional processors, particularly in terms of power consumption, is instruction dispatch [53]. Instruction dispatch is the portion of the pipeline where the instructions are decoded and sent to the functional units. It can be a major consumer of energy in conventional designs, as it cannot be parallelized, and therefore must run at the full clock speed of the chip.

Since a single vector instruction can potentially cause a large number of arithmetic operations to be initiated, the number of instructions per operation is reduced

[1] A functional unit is a sub-component of the processor that performs functions such as arithmetic, memory access, etc.

on a vector processor. This eases the bottleneck at instruction dispatch.

Vector processors can also help with the so-called memory/processor performance gap. As the clock speed of conventional processors has increased, memory latency have not kept pace. This performance gap limits the efficiency of conventional processors on memory intensive tasks [26].

The fact that a single vector instruction can spawn a large number of arithmetic instruction allows efficient pipelining of vector operations. The resulting long pipelines allow memory latency to be hidden, as more memory operations can be "in process" as earlier arithmetic operations complete.

The actual speedup of a particular architecture due to vectorization is dependent both on the algorithm and on low level details of the microarchitecture (e.g. memory hierarchy, bus size, clock speed, etc.). A truly fair comparison requires porting the algorithm to both the vector architecture and to an *equivalent* scalar architecture (e.g. same memory subsystem, same clock rate, etc.). Different optimizations of the algorithm would be required for the scalar and vector cases. Such detailed studies are rare, and are certainly beyond the scope of this thesis. Where available, relevant citations will be quoted below.

1.2.2 Extensions to Conventional Architectures

All of the major chip vendors provide vector extensions to their desktop processors. These vector extensions are implemented primarily using components that already exist on the chip. For example, vector arithmetic is performed using the existing arithmetic logic units (ALUs), but with multiple elements of lower precision types. So a 128 bit ALU might perform 16 simultaneous operations on 8 bit data, or 4 simultaneous operations on 32 bit data, etc. Ideally, the logic to control the operations is the only significant addition to the chip. In many cases, the additional logic requires only a tiny increase (e.g. 0.1% [45]) in the chip area over the conventional design.

With this approach, the number of vector registers is an issue. More vector

registers make it easier to code, and can reduce memory bottlenecks by providing more on-chip storage. However, more registers means more chip area. The size of the registers is also important. More bits in the registers allow more elements to be operated upon at once.

The number of operations that can be performed per clock cycle determines the overall maximum speedup over conventional designs. Note that this does not take into account latency, the amount of time it takes a single instruction to complete. Also, the peak performance may be difficult to achieve, as it is usually achieved with a combination of multiply-adds and memory operations.

Table 1.1 provides some details on selected vector extensions to conventional architectures.

Vendor	Name	Citation	# of registers	Bits/register	Max Ops/cycle
HP	MAX	[44]	32	64	4
Intel	MMX	[55]	8	64	2
Intel	SSE	[59]	8	128	4
Motorola	AltiVec	[50]	32	128	8
Sun	VIS	[38]	32	64	10

Table 1.1: Some details on selected vector extensions.

Since one of the goals of the vector extension design is to minimize the additional chip area, the vector extensions typically do not implement all possible vector instructions. For example, scalar/vector addition, where a scalar is added to each element of a vector, is not generally available. Instead, an instruction is provided that copies a scalar to all elements of a vector, and a vector/vector operation is provided. In Chapter 2, which provides information on the vector simulator used in this research, details will be provided as needed.

In [1], a vector speedup of 2.1× was reported on matrix-matrix multiply for Intel SSE compared to a highly optimized scalar version. [42] reports a speedup of 1.9–2.7× for HP MAX on a variety of video encoding benchmarks. [60] reports 1.1–4.2×

speedup for Sun VIS on a variety of video and image processing benchmarks. Anecdotal evidence seems to support the 2–4× range for speedups of highly vectorizable code (e.g. matrix operations) on conventional architectures with vector extensions.

1.2.3 Supercomputers

Vector processors have a long and successful history in supercomputers. Of the early vector supercomputers, the Cray-1 is probably the best known. Later machines combined multiple processors, each itself a vector processor. In the mid 80s, several so-called "mini-supercomputers" were released that also contained vector processors [56].

Table 1.2 lists some relevant details on a few of the major vector supercomputers. The table lists the year introduced, the clock rate in MHz, the number of vector registers, the number of elements per register[2], and the peak operations per clock cycle.

Machine	Year	Cite	Clock	# of regs	Elems/reg	Ops/cycle
CRAY-1	1976	[64]	80	8	64	4
Hitachi S810	1983	[46]	53	32	256	12
Fujitsu VP2600	1989	[72]	312	64	256	16
NEC SX-3	1990	[73]	345	72	256	16
CRAY X1	2002	[52]	800	32	64	16
NEC SX-6	2002	[52]	500	72	256	16

Table 1.2: Some details on selected vector supercomputers.

Vector supercomputers typically implement a very complete set of vector instructions, since the goal is maximal absolute performance rather than minimal chip area.

The introduction of inexpensive and fast microprocessors spelled the end of an era

[2]All machines use 64 bit data types. Note also that the Fujitsu VP series has configurable registers. The number of registers and number of elements can be adjusted as long as the total is 16K.

for vector supercomputers. Although vector supercomputers are still being produced (the NEC SX-8 was announced as I was writing this paragraph), so-called "scalar parallel" machines have eclipsed pure vector supercomputers for most applications. These machines consist of large numbers of scalar processors, often connected with a fast bus or even a network. These systems are typically much less expensive than the traditional vector supercomputer, mostly because of the commodity nature of their components. Also, memory for the vector supercomputers is typically quite expensive compared to memory cost for scalar parallel machines, as the latter can use commodity memory.

Nevertheless, vector processors will continue to fill a role in the supercomputer market. First, some tasks are very difficult to implement on a multiprocessor machine. Second, a combination of approaches, where each node of a multi-processor consists of a commodity-level vector processor, could yield a system with very high performance. Although no computers on the market currently fill this niche, I believe it is only a matter of time until such a machine is available.

Many papers have been written on vectorization methods for supercomputers. However, only a few present a careful comparison with scalar algorithms on the same architecture. In [5], a vectorized garbage collector was implemented. A vector speedup of 9× was reported vs. a scalar algorithm on the same architecture. In [20], performance of compiled code achieved a maximum of a 6× speedup. An anecdotal rule of thumb seems to be that a speedup of around 5—10× can be expected on highly vectorizable code on vector supercomputers.

1.2.4 Vector Microprocessors

In addition to vector supercomputers, microprocessors have also been constructed that implement vector instructions. Table 1.3 provides some details on two such processors, both developed as part academic research.

Torrent-0 only implements 32 bit integer and fixed point arithmetic, while IRAM supports 16, 32, and 64 bit integer, fixed point, and floating point arithmetic. Also

Name	Year	Citation	# of regs	Bits/reg	Clock	Ops/cycle
Torrent-0	1995	[7]	16	512	40 MHz	24
IRAM	2002	[40]	32	2048	200 MHz	48

Table 1.3: Some details on vector microprocessors.

of note is that IRAM is implemented with "processor in memory" (PIM), where the processor and memory reside on the same chip. This has advantages for power consumption, memory latency, memory bandwidth, and also allows finer architectural control of memory access patterns. Both microprocessors implement a full set of vector operations, with IRAM also providing a few specialized operations (e.g. for reductions and Fast Fourier Transforms). See Chapter 2 for details.

Note the very high operations per clock cycle. These numbers are the maximal achievable, and require a particular mix of operations. For example, on Torrent-0, the 24 operations must be 8 multiplies, 8 adds, and 8 memory operations. If an algorithm does not have this balance of operations, the actual performance will be less.

[7] reported results for a carefully conducted comparison of vector vs. scalar algorithms (the SPECint95 benchmark) on Torrent-0. For the most vectorizable components of the benchmarks, vector speedups of 8–14× were reported. Given the similarity in architecture between IRAM and Torrent-0, one would expect similar (or better) results on IRAM. In fact, [39] reports results for a set of embedded benchmarks on IRAM and for a similar scalar processor, the NEC MIPS VR5000. Although the benchmarks were run on a simulator of IRAM, and the NEC MIPS VR5000 is more capable than a scalar version of IRAM[3], the reported results were also in the 8–14× range.

[3]The VR5000 is a dual-issue MIPS processor running at 250MHz. It also has cache, while IRAM has no cache for vector loads and stores.

1.2.5 Special Purpose Hardware

Several video cards and game consoles use vector processors. Although details on the video cards is typically proprietary, Sony provides many details on the architecture of the Playstation II [66].

The Playstation II uses two architecturally similar vector processors. One is highly specialized for computer graphics rendering, and is not of interest for the current discussion. The other is a general purpose vector processor, tightly coupled to a conventional scalar processor. The vector processors has 32 vector registers. Each register element consists of 128 bits. An element can hold integer types of 8, 16, 32, or 64 bits, or a 32 bit floating point type. Peak performance is 8 operations per clock cycle. The memory subsystem is quite complex, including local caches for the different processors, shared caches, scratch RAM, etc. It also implements some highly specialized instructions that are useful in computer graphics (e.g. reciprocal square-root).

Since special purpose hardware rarely has any scalar equivalent, it is quite difficult to compute vector speedups. For the Playstation II, anecdotal reports of 5–8$\times$ have been circulated, but I know of no published results.

1.3 Related Work

Much work has been done on porting "multimedia" algorithms to particular vector architectures. Many of these algorithms are similar in nature to parts of a speech recognition system. Particularly successful work includes video encoding [43] [70] [67] [51], graphics transformations [70], and basic linear algebra subroutines (BLAS) [10] [6] [30]. References to previous work on architecture-specific algorithms that are presented in this thesis appear in the appropriate chapters.

Chapter 2

Vector Simulation Library

A vector simulation library was designed and implemented as part of this thesis. It enables us to simulate, at a coarse level, many different architectural features of vector processors. The vector simulator implements, as a C library, many of the common opcodes that are present in a vector architecture. The library does not attempt to simulate the performance of any particular architecture (e.g. cache, memory, chaining behavior, etc.). Such simulation is quite complex and is different for every architecture. Instead, the library allows simulation of a subset of the features of a vector processor.

First, the simulator assumes that the architecture is register-based. Each register holds not just a single value, as a scalar processor's register would hold, but rather a set of values. Vector instructions operate on vector registers in a fashion analogous to scalar instructions operating on scalar registers, except that elements are considered pairwise. So, for example, a vector-vector add takes two registers as input, computes the pairwise sum of each of the elements, and stores the results in another vector.

Each architecture has a maximum possible number of elements per vector register, known as the maximum vector length (`MVL`). On some architectures, this value is available either in a register or by using an instruction. This allows portability of the software across hardware with the same ISA (Instruction Set Architecture) but different `MVL` (maximum vector length). In other architectures, `MVL` is assumed to

be known (e.g. at compile time). Although the vector simulation library does allow access to `MVL` at run-time, none of the algorithms depend on this feature.

The instructions in a vector architecture do not necessarily operate on all `MVL` elements. First, the programmer can set the vector length. The vector length is specified either using an operand to the instruction, or, more commonly, by setting a control register to the desired vector length. Typically, if one attempts to set the length to a value greater than `MVL`, then either an exception is raised or the vector length is set to `MVL`. The latter allows code to be written that is independent of `MVL`. All instructions in the simulator operate only on vector elements 1 through the vector length.

In addition to setting the vector length, some architectures allow a mask to be specified. The mask is itself a vector of length `MVL`. If a mask is used for an instruction, then only those elements for which the mask has a non-zero element are touched. The mask can either be an operand to an instruction or a *mask register* can be set. In the simulator, only the instructions so indicated respect the mask settings (e.g. `vaddvv_mask`).

As an example of how the vector length and masks interact, consider the following pseudo-code for a masked, vector add. It takes two vectors as input, and computes the pairwise sum of the elements. It assumes that the vector length is in vl and that the mask is passed in vector register $Mask$ (as an array of booleans). See Algorithm 2.1.

```
procedure ADDVV_MASK(V1, V2, V3, Mask)
    for i ← 1, vl do
        if Mask_i then
            V1_i ← V2_i + V3_i
        end if
    end for
end procedure
```

Algorithm 2.1: Masked vector-vector add

At compile time, the user of the library can specify the precision and type of the elements of the vectors. Although some architectures allow one to change the precision on-the-fly, this library does not allow it. The scalar type is assumed to be the same precision and type as the elements of the vectors. The maximum value of the type is denoted V_{max}, while the minimum is denoted V_{min}.

2.1 Instructions in the Simulation Library

In the following list of instructions, the type `vector` indicates a vector register. The type `scalar` indicates a scalar of the same type as the vector elements. Memory addresses are represented either with `vectorElement*`, if it points to a block of memory with the same type as the vector elements, or `byte*` if it points to raw bytes (signed 8-bit integers). In a function's argument list, "**out** *variable*" indicates that *variable* is modified by the function. "**in/out**" indicates that the variable should be set before entering the function, and that it is modified by the function. With neither "**out**" nor "**in/out**", the variable should be set before entering the function, but the function does not modify the variable.

`integer len = getvl()`

Return the current vector length.

`setvl(integer len)`

Sets the vector length to `len`. If `len` is greater than `MVL`, the simulation library signals an error.

`vaddvv(out vector dest, vector src1, vector src2)`

Pairwise addition of the elements of vector registers `src1` and `src2`.

`vaddsv(out vector dest, scalar src1, vector src2)`

The scalar `src1` is added to separately to the elements of vector register `src2`, with results stored in vector register `dest`.

`vaddvv_mask(out vector dest, vector src1, vector src2, vector mask)`

Masked pairwise addition of the elements of vector registers `src1` and `src2`. Only those elements of `src1`, `src2`, and `dest` for which elements of `mask` are non-zero are operated upon. See Algorithm 2.1 for an example.

`vaddsv_mask(out vector dest, scalar src1, vector src2, vector mask)`

For each element of vector register `src2` for which the mask is non-zero, add the scalar `src1` and store into vector register `dest`.

`scalar v = vextract(vector src, integer position)`

Set `v` to the value of the vector element at the given `position` (e.g. `v = src[position]`).

`vinsert(in/out vector v, scalar src, integer position)`

Set the element of vector register `v` corresponding to the given `position` to the scalar `src` (e.g. `v[position] = src`).

`vload(out vector dest, vectorElement* base)`

Set the vector register `dest` to the elements starting at memory location `base`.

`vload_b(out vector dest, byte* base)`

Same as `vload`, except that memory is assumed to be stored as signed 8-bit integers, and is converted by the library on-the-fly to the internal precision and type of the vector registers as specified at compile-time.

`vloads(out vector dest, vectorElement* base, integer stride)`

Strided vector load. The first element of vector register `dest` is set to the value at memory location `base`. The second element is set to the value at `base + stride`. The third is set to `base + 2·stride`. See Algorithm 2.2.

`vloadx(out vector dest, vectorElement* base, vector offsets)`

Indexed vector load. Use the elements of vector register `offsets` as indices into memory starting at location `base`. See Algorithm 2.3.

procedure VLOADS($dest$, $base$, $stride$)
  for $i \leftarrow 1, vl$ **do**
    $dest[i] \leftarrow base[stride \cdot (i-1)]$
  end for
end procedure

Algorithm 2.2: Strided vector load

procedure VLOADX($dest$, $base$, $Offsets$)
  for $i \leftarrow 1, vl$ **do**
    $dest[i] \leftarrow base[Offsets_i]$
  end for
end procedure

Algorithm 2.3: Indexed vector load

`vmin(out vector dest, vector src1, vector src2)`

Pairwise minimum of the elements of vector registers `src1` and `src2`.

`vmultvv(out vector dest, vector src1, vector src2)`

Pairwise multiplication of the elements of vector registers `src1` and `src2`.

`vmultsv(out vector dest, scalar src1, vector src2)`

The scalar `src1` is multiplied separately by the elements of vector register `src2`, with results stored in the vector `dest`.

`scalar x = vreduce_max(vector src)`

Returns the scalar that is the maximum value of all the elements in `src`.

`scalar x = vreduce_min(vector src)`

Returns the scalar that is the minimum value of all the elements in `src`.

`scalar x = vreduce_sum(vector src)`

Returns the scalar that is the sum of all the elements in `src`.

`vsaddvv(out vector dest, vector src1, vector src2)`

Saturating vector/vector add. Same as `vaddvv`, except that if the sum of

two elements is greater than the maximum or less than the minimum possible given the precision of the representation, the resulting element is pegged to the extreme value. For example, if the vector elements are 8 bit signed integers, then $125 + 10$ will results in 127 and $-125 - 10$ results in -128.

`vsaddsv(out vector dest, scalar src1, vector src2)`

Saturating vector/scalar add. Same as `vaddvv` except that `src1` is a scalar rather than a vector.

`vsets(out vector dest, scalar src)`

Sets all elements of vector register dest to the scalar given by `src`.

`vshift(out vector dest, vector src, integer nelems)`

Shift the elements of vector register `src` by `nelems` to the right (or to the left if `nelems` is negative). For example, if `nelems` equals 1, then `dest[2] = src[1], dest[3] = src[2]`, etc.

`vstore(vector src, vectorElement* dest)`

Store the elements of vector register `src` to memory starting at location `dest`.

`vsubvv(out vector dest, vector src1, vector src2)`

Pairwise subtraction of of the elements of vector registers `src1` and `src2`.

2.2 Reductions

Reductions (such as `vreduce_sum`) are often not available directly. On some architectures, reductions can be implemented by a series of other specialized vector instructions (e.g. vhalf on IRAM [48] allows many reductions in $O(logn)$ time where n is the vector length). On architectures where reductions are not available in any form, scalar operations must be used. Note that the relative cost of using scalar operations to perform a vector reduction is lower if the vector length is shorter.

2.3 Saturating Arithmetic

Saturating arithmetic operations (such as `vsaddvv`) are commonly implemented in vector architectures meant for DSP (Digital Signal Processing) applications. Other architectures instead provide a higher precision version of arithmetic operations, followed by an instruction that reduces the precision and simultaneously saturates. On these architectures, saturating arithmetic typically takes two vector operations. For architectures where neither saturating arithmetic operations nor saturating precision shifts are available, saturating arithmetic can be implemented in terms of vector compares and vector adds.

2.4 Vector/Scalar Operations

Not all architectures provide arithmetic operations between scalars and vectors (e.g. `vaddsv`, `vmultsv`). Instead, one must first copy a scalar to all elements of a vector (e.g. `vsets`), followed by the desired vector/vector operation. This will require one extra vector register and one extra vector operation.

2.5 Vector Shift

Vector shift is central to an efficient small vocabulary decoding algorithm (see Chapter 6). Most architectures provide some means of computing a vector shift, usually through a more general operation. For example, many architectures provide a vector compress operation, which takes all elements of a vector for which the mask is non-zero, and stores them, in order, into a vector register. By setting the mask to `0 1 1 1 1 1 1 ...` and performing a vector compress, elements are shifted to the left by one. All algorithms in this work use `vshift`, as described above.

2.6 Memory

Although details on the efficiency of the memory subsystem are beyond the scope of this thesis, it should be noted that unit-stride loads (`vload`) and stores (`vstore`) are typically more efficient than strided or indexed loads and stores. The exact trade-off is a function not only of the architecture, but also of the algorithm (for example, some architectures can support more simultaneous unit stride loads than strided or indexed loads).

2.7 Compound Instructions

Most architectures implement some combinations of arithmetic operations in a single instruction. For example, since multiply-accumulate is a very common operation (see Algorithm 2.4), many architectures implement it as a single instruction. The vector simulation library described here does not use compound instructions, although many algorithms presented herein could benefit from them.

$total \leftarrow 0$
for $i \leftarrow 1, n$ **do**
    $total \leftarrow total + X_i \cdot Y_i$
end for

Algorithm 2.4: Typical multiply/accumulate

2.8 Strip-mining

It is often the case that the vector length required by an algorithm exceeds the maximum vector length supported by the architecture. In these cases, it is necessary to operate on "strips" of the data of length less than or equal to `MVL`. If the vector length is an integer multiple of `MVL`, then the algorithm may simply run on $V/$`MVL` strips of the input data. More usually, the vector length is not an

integer multiple of `MVL`. Also, the vector length is not always known at compile time (although the algorithms described herein assume that `MVL` is known at compile time). The algorithm for operating on vector lengths longer than `MVL` is known as strip-mining, and is exemplified in Algorithm 2.5. Note that $\lceil x \rceil$ indicates the `ceiling` operation, in which fractions are converted to integers by truncating towards positive infinity. For example, $\lceil 1 \rceil = 1, \lceil 1.2 \rceil = 2, \lceil -1.2 \rceil = -1$.

```
vl ← L%V                                  ▷ Input size L modulo vector length V.
for strip ← 1, ⌈L/V⌉ do
    setvl vl    ▷ Loads the remainder first, then loads chunks of maximum length.
    vload VR1, A                                  ▷ Load from memory location A
    vmultvv VR2, VR1, VR1          ▷ Square elements of VR1, store into VR2
    vstore VR2, B                                 ▷ Store into memory location B
    advance A by vl
    advance B by vl
    vl ← V
end for
```

Algorithm 2.5: Strip mining example. Squares elements of A, stores into B.

Some architectures have direct support for strip mining, including auto-increment and saturation of vector lengths. Auto-increment obviates the need for the "advance X by vl" calls). Saturation of the vector lengths causes the vector length to "max out" at `MVL`. If one attempts to set the vector length to a value greater than `MVL`, the vector length is set to `MVL`. The vector simulator assumes neither is available.

2.9 Chaining

A vector architecture does not always have to complete a vector instruction before starting on the next vector instruction. This overlap of execution is known as "chaining". Whether a new operation can start before an existing operation has completed depends both on algorithmic dependencies and on architectural issues.

If a later instruction requires the results of an earlier instruction, the later instruction must obviously wait until the early instruction completes. Examples include vector reductions followed by other vector operations, memory stores followed by overlapping memory loads, etc.

In addition to intrinsic dependencies, architectural limitations can also disallow chaining. To chain two operations, the hardware must forward the results of the earlier operation to the functional units (or memory load/store) of the later operation. Because this takes chip space, not all combinations of chaining are implemented in any particular vector architecture. Typically, only common, easily implemented patterns are supported. For example, many architectures will chain a memory load followed by an arithmetic operation. The arithmetic operation can therefore start before the final element of the vector load has completed. Note that chaining only speeds up the processing, allowing more operations to complete in a given amount of time. Chaining has no other effect on the code.

Chapter 3

Overview of Speech Recognition

To effectively describe the manner in which I modified the automatic speech recognition (ASR) algorithms to run efficiently on vector architectures, it is necessary to explain "conventional" ASR (automatic speech recognition) in more detail. In this section, I will present an overview of a full speech transcription system. In the following chapters, each component of the full system will be described in more detail, along with specifics on running the components on vector architectures.

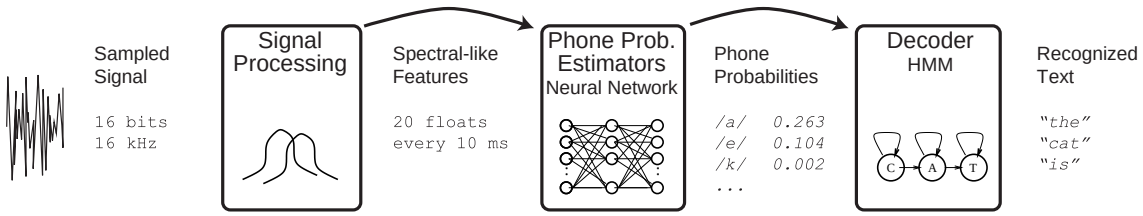

Figure 3.1: ICSI's hybrid speech recognition system.

Figure 3.1 shows a block diagram of ICSI's hybrid speech recognition system [13]. Sounds are digitized from a microphone, and are delivered to the Signal Processing unit, typically at 16,000 values per second and 16 bits per value. Sampling at higher rates helps very little, since meaningful features of speech generally occurs below 8

kHz. Also, sampling with higher precision has not been shown empirically to improve recognition accuracy.

3.1 Signal Processing

The Signal Processing unit performs *feature extraction*, in which the linear amplitude signal is converted to a sparser, spectral-like representation. Typically, the Signal Processor operates on overlapping time windows of 20–32 ms, and produces something like 20 "features" at every time step (typically 8–12 ms). The purpose of the Signal Processor is simply to produce features on which the later processing can operate. While the raw digitized waveform contains all the available information about the acoustic signal, as a practical matter, signal processing is critical. See Chapter 4 for more details.

3.2 Phone Probability Estimation

From the Signal Processor, the features representing a particular time interval are sent to the Phone[1] Probability estimator. This component estimates the probability that the given features represent a particular sound or combination of sounds in the language. For each sound or combination of sounds, the Phone Probability Estimator outputs a value between 0.0 and 1.0, once every time interval. Since the goal is to produce probabilities, the sum of the probabilities of all sounds in a given time interval should equal 1.0. Each phone probability is typically divided by its prior probability, yielding a scaled version of the emission likelihood that is used by the decoder component.

State of the art systems typically use a large number of context dependent phones [19] (e.g. /a/ preceded by a vowel and followed by a dental consonant) and Gaussian

[1]A phoneme is the minimal speech sound which distinguishes two words (e.g. "**S**at" vs. "**C**at"). Allophones are variant pronunciations of a phoneme, often dependent on context (e.g. "**S**at" vs. "wa**S**"). A phone is a set of allophones.

mixture models [58] to estimate their likelihoods[2]. Other groups have successfully used recurrent neural networks [14], decision trees [22], and support vector machines [57]. Although any standard machine learning algorithm can be used, Gaussian mixtures currently provide the most accurate systems.

At ICSI, we implement the Phone Probability Estimator using between 46 and 64 context independent phones and a multi-layer perceptron neural network to estimate their probabilities [12]. Accuracy suffers somewhat compared to context dependent Gaussian mixtures, but the system is smaller and simpler. Also, the vectorization methods presented here would be quite similar for a context dependent Gaussian mixture system, as the matrix operations involved are quite similar. Finally, for large vocabularies and conventional front ends, the Signal Processing and Phone Probability Estimator components typically consume a small fraction of the total computational load of a full ASR system. The decoder, described in the next section, consumes the lion's share. This balance can change for small vocabularies, where the decoder consumes a relatively smaller fraction of the computational load, and for more complex front-end (e.g. Section 5.4), where the Signal Processing and Phone Probability Estimator components consume a relatively greater fraction of the computational load.

Chapter 5 describes methods of running a multi-layer perceptron efficiently on vector architectures.

3.3 Decoding

Conceptually, the decoder takes the sequences of estimates of the phone probabilities, and compares them against models of every possible utterance in the language. It then outputs the most likely utterance. In practice, of course, the search space must be massively pruned for the process to be computationally tractable.

[2]Acoustic likelihoods are related by Bayes' Rule, namely $\frac{P(phone|feature)}{P(phone)} = \frac{P(feature|phone)}{P(feature)}$. Gaussian mixture systems estimate $P(feature|phone)$, while neural networks typically estimate $P(phone|feature)$.

The decoder is normally implemented as a search through Hidden Markov Models (HMMs) of sub-word units (e.g. phones), with one HMM per word in the vocabulary. The sequence of words is computed by combining the probability of each individual word according to an HMM with the *language model*.

The language model provides a score for a given sequence of words based on the likelihood of the sequence according to some model of how words group in the language. The language model may consist of a grammar (for command-and-control applications), or of statistics for runs of words[3] (for a transcription application). Although it may be possible to vectorize the language model evaluation, it is typically not a significant computational bottleneck. Language modeling will not be discussed further in this thesis.

The decoder is often the most computationally and memory intensive component of a large vocabulary ASR system. It is also the most challenging to vectorize. Chapter 6 provides more details on efficient algorithms for decoding when the vocabulary size is relatively small. Chapter 7 discusses issues when the vocabulary becomes large. The remainder of this section describes some aspects of decoding that are relevant to later sections.

3.3.1 Continuous vs. Discrete Decoding

Decoders can be categorized into two major types: *discrete* and *continuous*. Discrete decoders only recognize words (or short phrases) spoken in isolation. Continuous decoders recognize more natural, continuous speech. A common type of continuous decoder, called a stack decoder [54], uses a very slightly modified discrete decoder as its innermost loop. This inner loop is also the most computationally expensive component of a stack decoder. Therefore, if one can make a discrete utterance decoder that runs efficiently on vector architectures, it would be possible to write a continuous speech decoder that would run efficiently as well.

For purposes of this research, I restrict myself to discrete utterance decoders.

[3]For example, the sequence "the cat is" is much more likely to occur than "the cat blue".

Although there are other aspects of a continuous decoder that may be vectorizable (e.g. A* estimation [32]), the discrete decoder is the most computationally expensive component.

3.3.2 Dictionaries

Since one of the major factors that affect decoding is the size and composition of the recognition dictionary, several dictionaries of different sizes were used. The dictionaries consist of some number of words, along with one or more pronunciations for each word. Each pronunciation consists of an ordered list of phones.

In addition to the words, the dictionaries contain information about the phones. Each phone has a minimum duration, specified by the number of states in the phone, as well as additional data describing the distribution of durations (see Chapter 6 for details). All the dictionaries used the same data for the phones, derived from training on Broadcast News [16].

Table 3.1 summarizes some of the statistics of the dictionaries used in this thesis. The following subsections also give a brief description of the dictionaries.

Dictionary	*# of words*	*# of prons*	*Longest word* [4]	*# of states* [5]
Digits	12	12	six	12
Numbers	30	30	sixteen	19
Web	38	79	sixteen	19
SmallBN	983	1000	examinations	28
MedBN	4609	5000	significantly	28
LargeBN	19999	32010	telecommunications	42

Table 3.1: Dictionary statistics

[4]The word containing the largest number of states.
[5]The number of states in the word with the largest number of states.

Digits

The **Digits** dictionary contains single digits from one to nine, plus "zero", "oh", and "ten". Only a single pronunciation per word is used (the pronunciation that occurred most frequently in the training data). This dictionary could be used, for example, in a menu system, for entering simple phone numbers, etc.

Numbers

The **Numbers** dictionary is similar to **Digits**, but contains additional words for larger numbers (e.g. "fifteen", "thirty", "hundred"). It also uses only a single pronunciation per word. It could be used for any task involving only numbers (e.g. credit card numbers, zip codes, flight numbers, more complex menu systems, etc).

Web

This dictionary was used in a web navigation application, where every link on a web page was supplemented with a number. In addition to the words in the **Numbers** dictionary, it also contains words like "home", "bookmarks", "back", "page up", "page down", etc. Also, there are multiple pronunciations per word (providing better accuracy).

Large Broadcast News

The Large Broadcast News dictionary (**LargeBN**) consists of just under 20,000 words, with multiple pronunciations. It was used during the 1998 DARPA evaluations for quick decoding (65,000 words is more typical of a full system).

Medium Broadcast News

The Medium Broadcast News dictionary (**MedBN**) is a subset of **LargeBN**, with 5000 pronunciations chosen at random. Although not realistic in terms of

content, it represents a medium sized dictionary for a dictation task.

Small Broadcast News

Called **SmallBN**, this dictionary is a subset of **MedBN**, with 1000 pronunciations chosen at random.

Chapter 4

Signal Processing

Since the waveform contains all the captured acoustic information, one might think that the Signal Processing component would not be required in an ASR system; the raw audio signal could be fed directly into the Phone Probability Estimator. However, in practice the Signal Processing component turns out to be vitally important in real ASR systems.

Firstly, the Signal Processing unit typically reduces the data rate of the raw audio input, thereby decreasing the computational load of later processing. Secondly, providing features that are more closely related to the desired output (phones) makes the job of the Phone Probability Estimator much easier. For example, if spectral features are useful for determining phone identity, one should feed spectral features into the Phone Probability Estimator directly, rather than requiring it to learn the mapping from the raw data to a spectrum. It is also important not to waste the learning power of the trained system on aspects of the signal that do not generalize well, like waveform shape (for which the Signal Processing unit attempts to compensate). In practice, no working ASR system uses just the raw digitized audio.

Generally, spectral-like features fulfill the requirements of data reduction and data representation. In fact, it is well-known that the human auditory system processes input in a way similar to a spectral filterbank [21]. Most of the successful signal processing front-ends used in speech recognition are at least partially based on human

auditory perception [17] [27].

Figure 4.1 shows one type of signal processing that is commonly used in speech recognition systems, known as Mel-Frequency Cepstral Coefficients (MFCC) [17]. In the following sections, I will outline methods of vectorizing MFCC feature generation. Other signal processing systems use similar methodologies, so much of what is presented could easily be applied to other signal processing algorithms. Previous work on vectorizing speech frontends on a particular architecture can be found in [34].

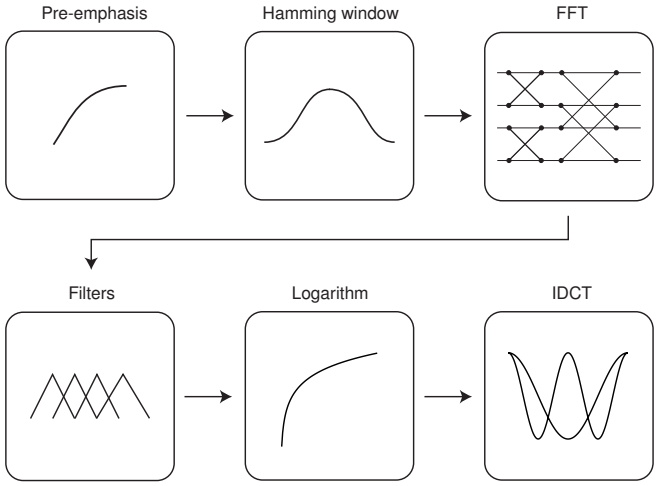

Figure 4.1: Block diagram of Mel-Frequency Cepstral Coefficients algorithm.

4.1 Pre-emphasis

The first step in processing is to apply pre-emphasis to the signal. This acts as a first order high-pass filter. The pre-emphasis is used both to filter out very low frequency components (which typically do not contribute to the intelligibility of speech), and to flatten spectral tilt associated with speech. It also mimics some of the equal loudness characteristics of the human auditory system [18].

Equation 4.1 gives the formula for computing the signal taking pre-emphasis into account. $y(t)$ is the output signal at time frame index t. $x(t)$ is the input at time frame index t. $x(t-1)$ is the input at time frame index $t-1$. α is the pre-emphasis coefficient. A typical value is $\alpha = 0.98$, yielding a high-pass filter with cutoff at about 60 Hz.

$$y(t) = x(t) - \alpha \cdot x(t-1) \tag{4.1}$$

Vectorizing pre-emphasis is quite easy. Simply read a chunk of data, shift it by one, multiply by the pre-emphasis coefficient, and subtract. The only minor complication is strip-mining (see Section 2.8), since the signal will rarely be smaller than the vector length. Algorithm 4.1 shows pseudo-code for performing pre-emphasis.

```
vl ← L%MVL                    ▷ Input size L modulo maximum vector length MVL.
f ← 0                         ▷ Stores the final element of the previous strip.
for strip ← 1, ⌈L/V⌉ do
    setvl vl
    vload VR1, X              ▷ Load signal from memory location X.
    vshift VR2, VR1, 1        ▷ VR2 is X shifted by 1.
    vinsert VR2, f, 0         ▷ Load the final element of the previous strip.
    vmultsv VR3, α, VR2
    vsubvv VR4, VR1, VR3
    vstore VR4, Y             ▷ Store results into memory location Y
    f ← vextract VR1, vl      ▷ Set the final element of the "previous" strip.
    advance X by vl
    advance Y by vl
    vl ← MVL
end for
```

Algorithm 4.1: Vectorizing pre-emphasis.

The algorithm requires $\lceil L/\texttt{MVL} \rceil$ times through the loop, where L is the input size (number of samples in the input utterance), and MVL is the maximum vector length.

Compared to a scalar algorithm, the only "extra" operation is a vector shift and a vector insert. There are two vector memory operations (one load and one store), two arithmetic vector operations, and one vector shift each time through the loop.

4.2 Windowing

Once pre-emphasis has been applied, a window is selected. A typical size for the window is 256 samples, which, at 8 kHz sampling, equals 32 ms. Using a tapered window removes discontinuities at the edges, and has been observed to improve performance. In the frequency domain, this corresponds to reducing the ripples in the frequency domain that would result from using a rectangular (untapered) window (i.e. computing spectra from a range of points without explicit windowing).

A windowing function that is a good compromise between signal distortion and smoothing is the raised cosine known as the Hamming window [25]. Equation 4.2 and Figure 4.2 show the function used in a Hamming window.

$$y(i) = 0.54 - 0.46 \cdot cos(2\pi i/255) \tag{4.2}$$

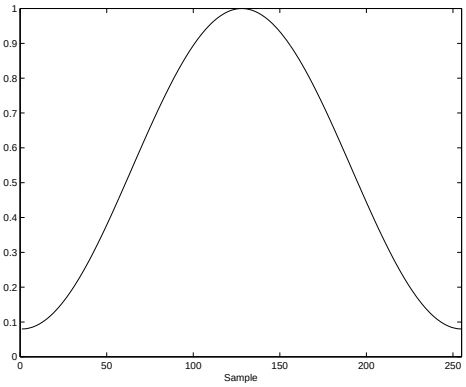

Figure 4.2: Hamming windowing function.

To vectorize the application of the Hamming window, one can pre-compute the coefficients as specified in Equation 4.2, and simply multiply them by the output of the pre-emphasis filter. Modulo strip-mining, this can be done with a single vector multiply. The inner loop will be called $\lceil L/\texttt{MVL} \rceil$ times, where L is the size of the window. Since there is only one vector arithmetic operation each time through the loop, the bottleneck for the algorithm will be memory bandwidth on most architectures.

4.3 Filterbank

The computational bottleneck of MFCC analysis is the computation of the filterbank outputs. Typically, a filterbank is implemented as a Fast Fourier Transform (FFT), followed by an inner product with the various filters. The FFT requires a particular pattern of strided memory access. As the FFT is an important computational kernel, vector architectures quite commonly provide some level of direct support for it. This can either be specialized addressing schemes or methods to quickly move data in ways that facilitate the FFT. Details of FFT implementations are beyond the scope of this thesis, but are well documented in other works [71] [10] [8].

Once the FFT of the windowed, pre-emphasized signal is determined, the output of the filters must be computed. For MFCC features, these are composed of a sequence of triangular filters. Commonly, the spacing is uniform in the linear domain under 1000 Hz, and then linear in the logarithmic domain (exponential in the linear domain) for high frequencies. The number of filters is dependent on the bandwidth. Figure 4.3 shows a set of filters using 5 linear and 8 logarithmic filters[1]. For each filter, the inner product of the output of the FFT and the filter is computed. This results in a single number for each filter.

[1]Typically, more filters would be used. The lower number of filters makes the figure more legible.

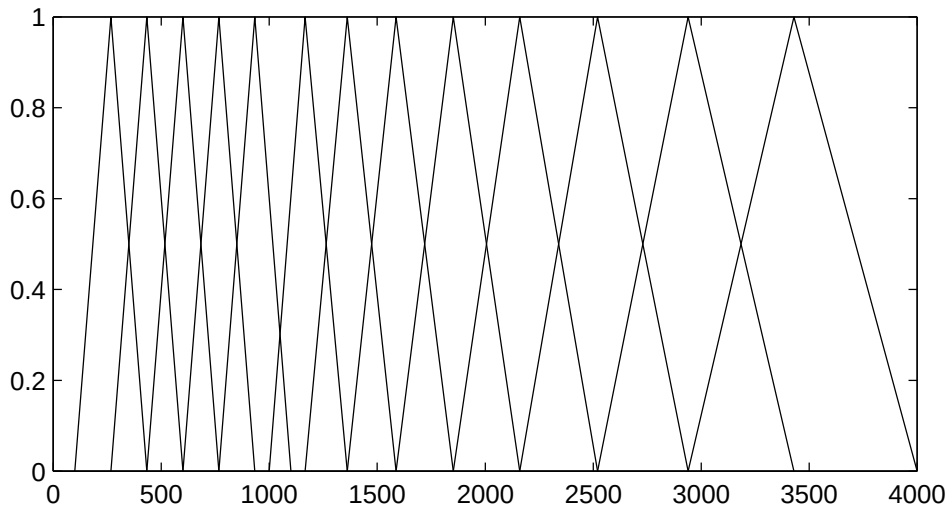

Figure 4.3: Triangular filters.

Vectorizing the filterbank

Since the input signal is discrete, usually sampled at 8 or 16 kHz, the filters can be discrete as well. The outputs of the filterbank can therefore be computed as $O_i = \sum_{f=1}^{N_i} A(f) \cdot F_i(f)$, where the amplitude of the output of the FFT is given by $A(f)$, the filter coefficient of filter i is given by $F_i(f)$, and filter i has N_i coefficients.

Given the typical filter parameters (starting frequency, spacing), N_i ranges in size from 3 elements to about 30 elements. If `MVL` is in this range, an efficient vectorized algorithm for computing the filterbank outputs is a simple strip-mined loop for each filter, followed by `vreduce_sum`, a vector reduction operation (see Section 2.1). For each filter, there is a loop that executes $\lceil L/\texttt{MVL} \rceil$ times, where L is the size of the filter. Each time through the loop, there are three memory accesses (two loads and one store), one multiply, and one vector reduction. Note that the filter sizes are known at compile time, so loop overhead can be minimized.

If `MVL` is much longer than the filter sizes, or if vector reductions are expensive,

the above algorithm can be wasteful. An alternative is to compute the output of several filters at once. This requires an indexed load, which can be expensive on some architectures. Furthermore, the product of the signal and the filters are left in a vector register, as shown in Figure 4.4. These values from each filter must be summed to achieve the output of the filterbank. The computation of these sums will require almost the same amount of work as the algorithm described in the previous paragraph. Unless vector multiplication is much more expensive than the summing, it will be more efficient to perform the multiplication and the summing as described in the previous paragraph.

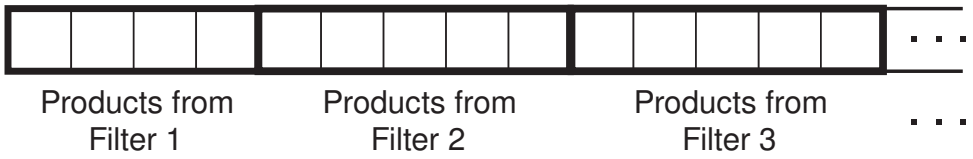

Figure 4.4: Multiple filters per vector register.

4.4 Logarithm

Human perception of loudness is compressive, commonly simulated with a cube root or a logarithm function. This stage mimics human perception by applying the logarithm function to each filter output. It is not a computational bottleneck, since the number of filters is fairly small, and the logarithm is taken independently on each element.

If a vector implementation of this stage is required, a typical approach would be to use a vectorized version of one of the many approximation algorithms for the logarithm function. This is generally straight-forward. The only complication involves algorithms that require table lookup. Such algorithms can be efficiently implemented only if the indexed load operation is available. If the architecture does

not support indexed load, table-driven logarithms may be inefficient. Details of implementation are beyond the scope of this thesis, but have been well-reported in the literature. See, for example, [69] [37] [2].

4.5 DCT

The output from the logarithm stage has high feature-to-feature correlation. To reduce this correlation, it is typical to transform the data. At the same time, the number of outputs can be reduced. A common transform that achieves these goals is the discrete cosine transform [18]. One may implement the discrete cosine function using a "fast" algorithm very similar to the fast Fourier transform described in Section 4.3. Again, details are out of scope for this thesis. Typically, the first 8 to 14 coefficients of the output of the DCT are used for further processing, as described in the following chapters.

4.6 Summary

All the elements of the Signal Processing component vectorize well. The FFT is the most costly element, but is usually optimized on vector architectures. As a result, the Signal Processing component tends to be a minor addition to the computational load of a vectorized ASR system. This balance can shift if multiple signal processing components are used (see Section 5.4). Also, for small vocabulary systems, the Signal Processing component can be a higher fraction of the total computation.

Chapter 5

Phone Probability Estimator

The purpose of the Phone Probability Estimator is to output the probability of each sound in the language given a sub-sample of the input features. Since the context around a sub-sample provides additional information about the identity of the central element, systems often take as input not only the features output from the Signal Processing unit for the current time interval, but also some number of frames before and after the current time interval. Given this context window, the Phone Probability Estimator outputs a vector consisting of numbers between 0 and 1, representing the probability of each phone in the language for the current time interval. Since the outputs are probabilities, and there is a fixed set of possible outputs, the probabilities must sum to 1.0. The number of distinct phones for our standard English language systems (and therefore the output vector length) is between 46 and 64.

As mentioned in Chapter 3, any probabilistic machine-learning algorithm can be used to compute this mapping between input features and output probabilities. At ICSI, we have used multi-layer perceptrons (MLP), which work well on a variety of problems [12]. Furthermore, MLPs (multi-layer perceptrons) are easy to vectorize, and will therefore run efficiently on vector architectures. Similar methods can generally be used for other machine learning algorithms.

A typical MLP Phone Probability Estimator that was used in the Broadcast

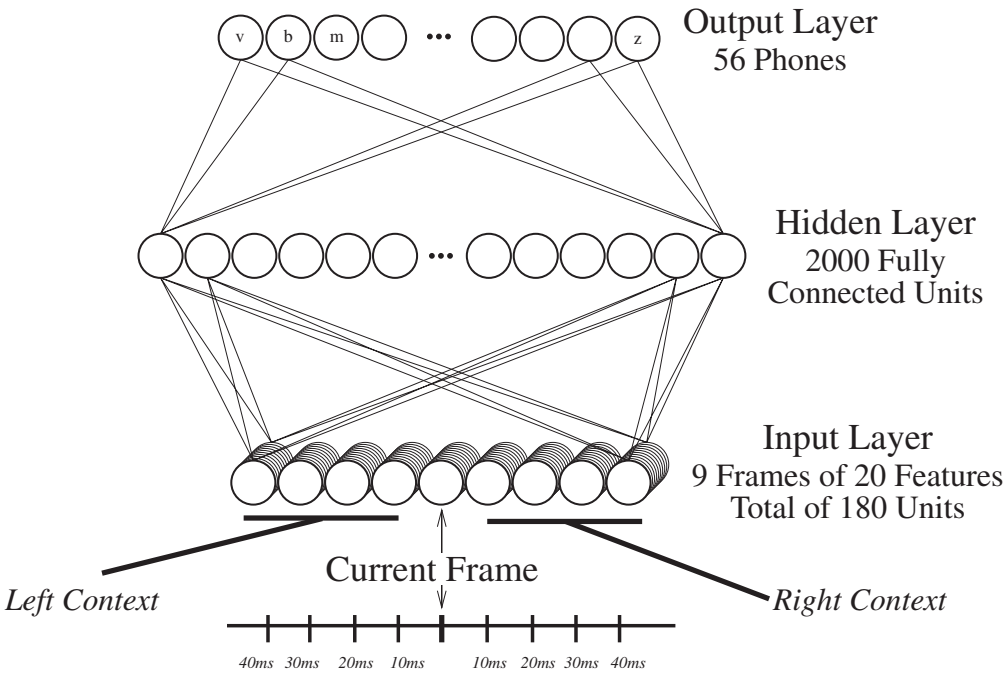

Figure 5.1: Block diagram of an MLP Phone Probability Estimator.

News evaluation [16] is shown in Figure 5.1. A context window of 9 frames and 20 features for a total of 180 input units is shown. The 180 input units are multiplied by the input-to-hidden weight matrix W_{hi}, a matrix consisting of 2000×180 numbers. This yields 2000 numbers at the hidden layer. A sigmoid function is applied to these 2000 numbers, producing the hidden layer values. The processes is repeated with the 56×2000 hidden-to-output weight matrix W_{oh}, followed by a soft-max function. The soft-max takes N inputs q_i and produces N outputs p_i according to equation Equation 5.1. It is easy to see that $\sum p_i = 1$. These outputs can be interpreted as probabilities [15].

$$p_i = \frac{e^{q_i}}{\sum_{j=1}^{N} e^{q_j}} \tag{5.1}$$

The sigmoid and softmax functions are not typically the computational bottleneck of the MLP computation. However, if we fail to vectorize them, Amdahl's Law [4] tells us that they will eventually dominate as the other parts of the algorithm become more efficient. Both sigmoid and softmax require a vectorized version of the exponential function. Similar to the discussion of the logarithm in Section 4.4, the exponential is generally easy to vectorize, although if a table-lookup algorithm is used, architectures that do not support indexed loads will perform poorly. Details of implementation are out of scope, but have been well-covered in the literature [37] [2] [65].

The MLP must be trained on large amounts of data that are typical of the conditions in which the application will be used. The training procedure picks values for W_{hi} and W_{oh} such that the mapping between the input features and the output probabilities is as accurate as possible on the training set. Training procedures for MLPs are out of scope for this thesis, but are computationally similar to the procedure described in this chapter.

5.1 Matrix-Matrix Multiply

For efficiency, several input vectors are usually queued up, and a matrix-matrix multiply (rather than a matrix-vector multiply) is performed. This matrix-matrix multiply is the computational bottleneck of the Phone Probability Estimator.

Previous work on matrix-matrix multiply for specific vector architectures can be found in [30], [1], and others. In this section, I will discuss methods of choosing a particular matrix-matrix multiply algorithm based on architectural variables such as vector length and the efficiency of various operations (e.g. vector reduce).

Matrix-matrix multiply is a very regular operation. Also, the number of arithmetic operations is $O(n^3)$ while the theoretical minimum number of memory accesses is $O(n^2)$ (where n is, for example, the maximal dimension of the matrix). For maximal performance, it is therefore very important to access memory as infrequently as possible. Locality of reference becomes quite important — register access is much faster than cache; cache is faster than memory, etc. The memory subsystem therefore is crucial to the performance of a matrix-matrix multiply on most architectures [41].

To achieve higher performance (through locality of reference) given the memory hierarchy, the matrices are typically broken down into submatrices.

$$A \cdot B \leftrightarrow \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00} \cdot b_{00} + a_{01} \cdot b_{10} & a_{00} \cdot b_{01} + a_{01} \cdot b_{11} \\ a_{10} \cdot b_{00} + a_{11} \cdot b_{10} & a_{10} \cdot b_{01} + a_{11} \cdot b_{11} \end{pmatrix}$$

In the example above, each element (e.g. a_{01}) can itself be a matrix, and the "$\cdot$" operator represents a matrix-matrix multiply. The algorithm then recursively divides the matrix. Efficiency is improved because of data locality. Different divisions can be performed such that, at any point in the algorithm, the current matrix block fits into one level of the memory hierarchy. For example, at the lowest level, the block should fit into the registers of the processor. The next level up might fit into the L1 cache, etc.

It is also possible to organize the recursion to trade off recursive calls to the matrix multiply for additions. These so-called "Strassen-like" algorithms have better

asymptotic performance ($O(n^{log_2 7})$ instead of $O(n^3)$), but have additional overhead compared to the standard algorithm [68] [9]. Typically, only very large matrices benefit from Strassen-like algorithms, and even then only for the first few iterations of the recursion [29].

Optimizing for each particular memory subsystem is itself quite difficult, as the best approach is dependent on many fine-grain details of the memory system design. Results from one architecture do not necessarily translate into other architecture. Therefore, we instead advocate a "generate-and-test" approach, whereby various choices of the blocking sizes are automatically generated and benchmarked, and the most efficient one is chosen for a particular architecture [11] [74]. The vectorized matrix-matrix multiply will simply be the final, smallest block size. Implementing a full generate-and-test methodology is beyond the scope of this thesis. Instead, I present various schemes for implementing the lowest level of the algorithm.

The problem is to compute the product of an input matrix I of size $N \times K$ and a weight matrix W of size $K \times M$, yielding an output matrix of size $N \times M$. Note that N, M, and K are the sizes of the lowest level blocks in the generate-and-test scheme described above.

$$\begin{pmatrix} O_{00} & O_{01} & \cdots & O_{0M} \\ O_{10} & O_{11} & \cdots & O_{1M} \\ \vdots & \vdots & \ddots & \vdots \\ O_{N0} & O_{N1} & \cdots & O_{NM} \end{pmatrix} = \begin{pmatrix} I_{00} & I_{01} & \cdots & I_{0K} \\ I_{10} & I_{11} & \cdots & I_{1K} \\ \vdots & \vdots & \ddots & \vdots \\ I_{N0} & I_{N1} & \cdots & I_{NK} \end{pmatrix} \begin{pmatrix} W_{00} & W_{01} & \cdots & W_{0M} \\ W_{10} & W_{11} & \cdots & W_{1M} \\ \vdots & \vdots & \ddots & \vdots \\ W_{K0} & W_{K1} & \cdots & W_{KM} \end{pmatrix}$$

$$O_{ij} = \sum_{k=0}^{K} I_{ik} \cdot W_{kj}$$

A conventional scalar and naïve method would be to execute the pseudo-code:

To vectorize the above, first one must decide what the vector registers should hold. Then, one must decide the order of the loops (the above order is just one of 6 possible orders even for the naïve algorithm).

```
for n ← 0, N do
    for m ← 0, M do
        Out[n, m] ← 0.0
        for k ← 0, K do
            Out[n, m] ← Out[n, m] + In[n, k] · Weight[k, m]
        end for
    end for
end for
```

Algorithm 5.1: Conventional naïve scalar matrix multiply

5.2 Vectorizing by K

Perhaps the most obvious way to organize the algorithm is to vectorize by the inner, common dimension K. Rows of the input matrix I and columns of the weight matrix W are stored in vector registers. A single scalar value of the output is computed at a time. One may execute either the `N` loop or the `M` loop first. See Algorithm 5.2 and Algorithm 5.3.

```
for n ← 0, N do
    for m ← 0, M do
        Out[n, m] = InnerProduct(In[n, :], Weight[:, m])
    end for
end for
```

Algorithm 5.2: Matrix multiply using inner product, N followed M

```
for m ← 0, M do
    for n ← 0, N do
        Out[n, m] = InnerProduct(In[n, :], Weight[:, m])
    end for
end for
```

Algorithm 5.3: Matrix multiply using inner product, M followed N

Which ordering of the loops is more efficient will depend on details of the memory subsystem and how the matrices are stored in memory. One or the other will likely

result in fewer bank conflicts [23]. We recommend including both orders in the "generate-and-test" methodology, and picking the better of the two.

The problem is now reduced to an efficient implementation of an inner product on a vector architecture.

The inner product takes two equal-length vectors as input and produces a scalar as output. With no error checking, the pseudo-code for a scalar algorithm is shown in Algorithm 5.4.

```
procedure InnerProduct(a, b)
    sum ← 0
    for i ← 1, L do                          ▷ L is the input length
        sum ← sum + a[i] · b[i]
    end for
    return sum
end procedure
```

Algorithm 5.4: Scalar inner product

The first architectural consideration is the vector length. Usually, the greater the vector length, the higher the efficiency of the architecture. However, shorter vector lengths are typically easier to vectorize, since strip-mining overhead can be avoided. The inner product is no exception.

If the vector length is longer than the size of the input, then a vectorized version of Algorithm 5.4 might read something like that of Algorithm 5.5.

```
procedure InnerProduct(A, B)
    setvl vl                                 ▷ vl is the length of A and B
    vload VR1, A
    vload VR2, B
    vmultvv VR3, VR1, VR2
    return vreduce_sum(VR3)
end procedure
```

Algorithm 5.5: Simple vector inner product

Algorithm 5.5 depends on `vreduce_sum` being both available and efficient on the architecture (`vreduce_sum` simply sums all the elements of a vector and returns the scalar result). Frequently, such a function is unavailable as a primitive in the architecture, although methods of implementing it efficiently are often available. See Section 2.2.

If the vector length is shorter than the size of the input (the more common case), then the inner loop of the vectorized algorithm must be strip-mined (see Section 2.8). The partial sums can be stored in a vector register. The final sum must then be reduced. Algorithm 5.6 demonstrates this.

```
procedure InnerProduct(A, B)
    setvl V                  ▷ VR4 collects the partial sums. Set all elements to zero.
    vsets VR4, 0
    vl ← L%V                 ▷ Input size L modulo vector length V.
    for strip ← 1, ⌈L/V⌉ do
        setvl vl
        vload VR1, A
        vload VR2, B
        vmultvv VR3, VR1, VR2;
        vaddvv VR4, VR4, VR3;
        advance A by vl
        advance B by vl
        vl ← V
    end for
    return vreduce_sum(VR4)
end procedure
```

Algorithm 5.6: Strip-mined inner product code

As you can see, even when the vector length is shorter than the input length, `InnerProduct` still requires a call to `vreduce_sum`. However, it only requires one call per call to `InnerProduct`, whereas many multiply/adds are performed. Therefore, this algorithm can be quite efficient if the vector length is short relative to both the

input size and the cost of `vreduce_sum`.

Performance

To perform `InnerProduct` requires $2 \cdot \lceil L/V \rceil$ vector loads, $\lceil L/V \rceil$ vector multiplies, $\lceil L/V \rceil$ vector adds, and one `vreduce_sum`. The vector length in all cases is the maximal vector length V, except for two vector loads, one vector multiply, and one vector add (for the case where V does not evenly divide L).

When calling `InnerProduct` in the inner loop of a matrix-matrix multiply, note that the columns of the second matrix require a strided load. If strided loads are expensive, it is possible to store the transpose of the second matrix, and use unit stride instead. `InnerProduct` must be called a total of $M \cdot N$ times.

A small performance gain can be obtained if the matrix sizes are known at compile time. This is the usual case, since the generate-and-test methodology will pick a particular size for the lowest level blocks. Strip-mining can be avoided if V divides L evenly, and the loops can be unrolled. Better performance can be achieved on some architectures through software pipelining [3].

The algorithm uses four vector registers. There is one memory access per arithmetic operation, leading to $O(n^3)$ memory accesses.

Cache Considerations

Since the size and order of evaluation of the matrix-matrix multiply is computed during the generate-and-test cycle, cache considerations are not directly an issue. The best algorithm should be selected given your memory architecture. That being said, a few general comments can be made. If the local cache is too small to hold the intermediate results (about 4 times the vector length for the inner product version of Algorithm 5.6), then the best block size will probably be the vector length. This allows an inner loop with no strip-mining. Typically this will only be the case if there is no cache at all. If the caches are large enough to hold all the matrices, then

the block size will probably be the matrix size.

Chaining Behavior

The inner loop is a typical load/operate loop. As such, most architectures try to optimize chaining behavior for such cases. Loop unrolling and software pipeline can improve the chaining. See Section 2.9 for discussion of chaining.

Register Blocking

The algorithm as described above requires two memory accesses for each output value. Since memory access is usually quite a bit slower than arithmetic operations, this can be quite inefficient. One method of improving performance is to store elements of the matrices in the vector registers — in effect, using the registers as a cache. This is known as register blocking [74] or tiling. The basic idea is presented in Figure 5.2. Rows of the input matrix and a column of the weight matrix are stored in vector registers. Algorithm 5.7 shows a fragment of the code that computes the matrix product (strip-mining is omitted for brevity). As the number of available registers increases, so too does the size of the algorithm. However, the code is quite regular, and can be generated with an automated procedure.

With this algorithm, only $O(n^2)$ memory accesses are required, rather than the $O(n^3)$ of the previous algorithm. The drawback is that there must be enough vector registers to hold the entire sub-matrix. On architectures with plentiful registers, it is common that the block size that allows the entire sub-matrix to be stored will be the most efficient size, as computed by the generate-and-test method.

5.3 Vectorizing by M or N

Another way to organize the vectorization is to vectorize over one of the "outer" dimensions, M or N. The algorithms to vectorize by M and N are quite similar. In

$$\begin{array}{l} VR1 = \\ VR2 = \\ VR3 = \end{array} \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \\ = & = & = \\ VR4 & VR5 & VR6 \end{pmatrix}$$

Figure 5.2: Register blocking example.

setvl K
vload VR1, In
vloads VR4, $Weight$, M
vmultvv VR0, VR1, VR4
$Out[1,1]$ = vreduce_sum VR0
advance $Weight$ by 1
vloads VR5, $Weight$, M
vmultvv VR0, VR1, VR5
$Out[1,2]$ = vreduce_sum VR0
advance $Weight$ by 1
vloads VR6, $Weight$, M
vmultvv VR0, VR1, VR6
$Out[1,3]$ = vreduce_sum VR0
advance In by K
vload VR2, In
vmultvv VR0, VR2, VR4
$Out[2,1]$ = vreduce_sum VR0
etc.

Algorithm 5.7: Register blocked matrix multiply.

this section, I will only describe vectorization by M. Whether to vectorize by N or M will again depend on the details of the architecture, and should be decided with the same generate-and-test approach as described above.

When vectorizing by M, rows of the second matrix are stored in a vector. This vector is multiplied by a scalar taken from the first matrix. The resulting vectors are summed and stored into rows of the output matrix.

$$O_{n,:} = \sum_{k=0}^{K} I_{n,k} \cdot W_{k,:}$$

Vectorized pseudocode for this ordering can be seen in Algorithm 5.8. For brevity, strip-mining has not been included.

VR1 — A row from the $Weight$ matrix.
VR2 — A scalar from In times VR1.
VR3 — A row of the Out matrix.

```
setvl M                                          ▷ Row size of output.
for n ← 1, N do
    vsets VR3, 0
    for k ← 1, K do
        vload VR1, Weight
        vmultsv VR2, In[n, k], VR1
        vaddvv VR3, VR2, VR3
    end for
    vstore VR3, Out
    advance Weight by M
    advance Out by M
end for
```

Algorithm 5.8: Vectorizing by M

Performance

The number of operations performed by this ordering is exactly the same as in Section 5.2 for vectorizing by K. However, no vector reduction operation is needed. This ordering is therefore a better match for architectures in which reductions are unavailable or expensive.

Cache, Chaining, Register Blocking

Even if reductions are cheap and available, it is possible that this ordering will perform better, depending on the details of the memory hierarchy. The same arguments as in Section 5.2 apply. Once again, generate-and-test is the method of choice.

The innermost loop of Algorithm 5.8 consists of a load, a multiply, and an add. On most architectures, this is extremely efficient, both in terms of chaining, and possibly in terms of compound instructions (e.g. multiply-accumulate). Again, loop unrolling, software pipelining, and register blocking can improve the performance. See Section 5.2.

5.4 Combination of Systems

Previous work has shown the efficacy of combining multiple representations of the audio stream [31]. Multiple frontends, each with its own phone probability estimator, are combined. Figure 5.3 shows on example of combining an MFCC [17] system and a PLP [27] system. The phone probabilities can be combined either using a simple averaging rule, or with more complex methods [49] [36]. Such systems consistently show improvements over either system alone, especially on unseen acoustic conditions.

Because the Signal Processing and Phone Probability Estimating components are both highly compatible with vector architectures, the cost of including multiple

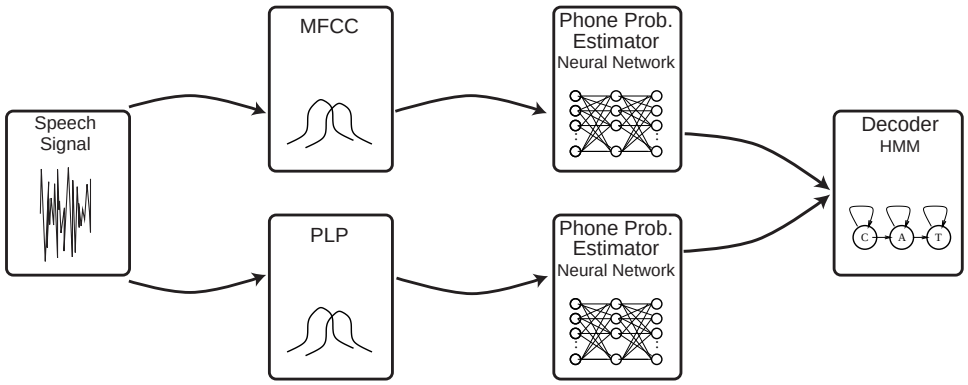

Figure 5.3: Combination of an MFCC system and a PLP system.

copies of these components is usually small compared to the cost of the decoder. Once again, this balance can shift for small vocabularies, where the Signal Processing and Phone Probability Estimating components take up a relatively larger fraction of the total computational load.

Chapter 6

Small Vocabulary Decoder

In this section, small vocabulary decoders are described. These decoders take a list of words and output the most likely words given the probability stream from the Phone Probability Estimator. Note that by "small vocabulary" I do not mean simply that the dictionary has fewer than N words. Rather, the algorithms described in this section evaluate the probability of every word in the dictionary. As the dictionary gets large, this becomes inefficient compared to methods that are able to avoid evaluating low probability words. The advantage of the former type of algorithm in the current context is that they tend to be much more regular, and therefore more amenable to efficient execution on vector processors. For details on large vocabularies, where avoiding extraneous computations is critical, see Chapter 7.

Since the task of the discrete utterance decoder is to compute the likelihood that a stream of phone probabilities match a word, we must start our discussion with how words in the dictionary are modeled. In the simplest case, each word can be thought of as consisting of a finite state machine, with one state per phone in the word, and transitions from phone i to phone i and $i+1$. Figure 6.1 shows an example of such a finite state machine for the word "about". The word starts in the state labeled **ax** (the "uh" sound), stays there for a while, then transitions to the state **b**, stays there for a while, and so on until the end of the word. In real systems, a phone is usually composed of several states, but the left to right ordering is maintained. By

using several states per phone, we impose a minimum duration on the phone equal to the number of states in the phone multiplied by the duration per state.

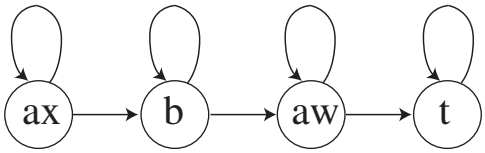

Figure 6.1: Finite state diagram of the word "about".

Given the finite state machine representing a word and the phone probability stream as output from the Phone Probability Estimator, the next task is to compute the likelihood that the finite state machine and the phone stream match. Most ASR systems, including ours, use the so-called "Viterbi approximation"[1]. The Viterbi algorithm can be implemented as a dynamic program, as shown in Figure 6.2. Every entry in the table is filled in according to the following rule:

$$E(t,s) = P(X_t\,|S_s) \cdot \max(\, E(t-1,\ s),\ E(t-1,\ s-1)\,) \tag{6.1}$$

where t is a time index, s is a state index, $E(t,s)$ is the table entry at time t and state s, and $P(X_t|S_s)$ is the scaled likelihood of phone S_s at time t as output by the Phone Probability Estimator.

The above procedure computes the best path through the finite state machine. The probability of a given word is simply the value of the lower right table entry $E(t_{max}, s_{max})$. The discrete utterance decoder computes the score for each word, and outputs either the best word or the top few words.

[1]Briefly, the Viterbi method it is an approximation because it only takes into account the likelihood of the best path, rather than incorporating the likelihood of all possible paths. See, for example, [13] for details on the Viterbi approximation to the Hidden Markov Model.

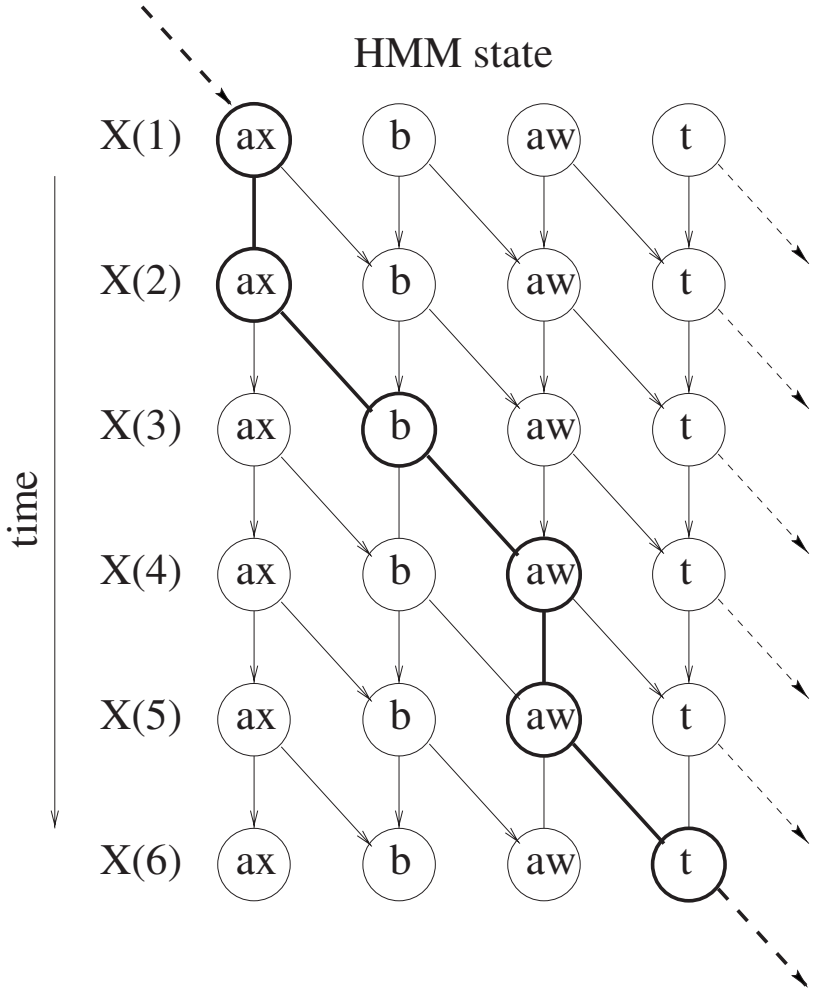

Figure 6.2: Viterbi dynamic program for the word "about". The best path is highlighted.

Some Complications

There are a few complications in the actual algorithm. First, we take $-log$ of both sides of Equation 6.1. The multiplications become additions, and the max becomes a min. This is done both because additions are computationally less expensive than multiplications, and because the numerical stability of performing a sequence of additions is better than performing a sequence of multiplications.

Another complication is that we model the durations of the phone states by assigning a self-loop and an exit probability to the transitions in Figure 6.1. This models the durations as an exponentially decaying distribution. Typically, both the number of states per phone and the self-loop and exit probabilities are selected during the training procedure. For our purposes, we will assume that this information is part of the dictionary.

With these two changes, Equation 6.1 becomes:

$$E(t,s) = -\log P(X_t|S_s) + \min(\, E(t-1,\ s) + \text{self}_s,\ E(t-1,\ s-1) + \text{exit}_{s-1}\,) \quad (6.2)$$

$E(t,s)$ is the table entry for state s at time t, but is this time a log probability that we are minimizing. $logP$ is the log likelihood from the Phone Probability Estimator. self_s is the $-log$ of the self-loop probability of state s, and exit_{s-1} is the $-log$ of the exit probability of the previous state.

6.1 Vectorizing by State

There are several ways in which the discrete utterance decoder can be vectorized. For example, one could try to vectorize along the time axis in Figure 6.2, where a vector register would hold a column of the figure, and each element would hold a different time. However, there is a problem with this approach. $E(t,s)$ is a function of $E(t-1,s)$. The dependency of vector element t on vector element $t-1$ within the vector prevents vectorization along the time axis.

One could also vectorize along the state axis in Figure 6.2. A vector register holds a row in the figure, and each element holds a different state.

One drawback of this arrangement is that words longer than `MVL` (where `MVL` is the maximum vector length as described in Chapter 2) will not fit into a single vector register. Although it is possible to split long words up into multiple registers and stitch them together, the overhead reduces the efficiency of the algorithm. Therefore, this method is not recommended for architectures with short vector lengths.

On architectures with `MVL` much longer than the typical number of states in a word, the arrangement as presented is inefficient. More efficiency can be achieved if we can increase the vector length. A simple method of extending the vector length for vectorization by state is to put words side by side, as in Figure 6.3.

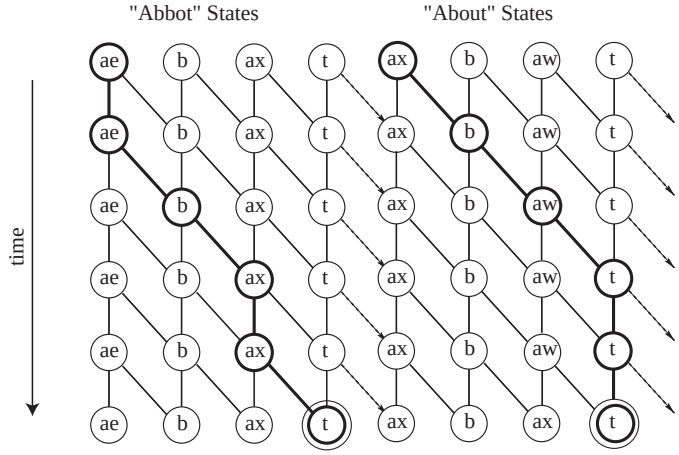

Figure 6.3: Viterbi table for two words, “abbott” and “about”.

Details on an algorithm to simultaneously compute the score for each word in the figure will be presented in Section 6.1.2. Next, an algorithm is presented for picking which words to group together.

6.1.1 Bin packing

Of course, using just two words as in Figure 6.3 is unlikely to be optimal. Rather, it is desirable to match the number of total states (the sum of the number of states in each word) with the vector length of the architecture. This minimizes strip-mining overhead, while maximizing the vector length.

Imagine we have N words in our dictionary. Each word i has n_i states. We want to arrange the words into G groups such that each group has no more than `MVL` total states, while simultaneously minimizing the total number of groups. This is the classic "bin packing" algorithm [47]. Computing the optimal case is NP-complete. However, it has been shown that the best possible polynomial time algorithm is, at the worst, 22% less optimal than the best possible packing [28]. One algorithm that achieves this level is the so-called "ordered first fit" algorithm [33], shown in Algorithm 6.1.

```
Sort N words by the number of states in the word from largest to smallest.
for i ← 1, N do                                          ▷ Word index
    for g ← 1, G do                                      ▷ Group index
        if word i will fit in group g then
            Put word i into group g, incrementing group g's current size.
            last                                         ▷ Exit the inner for loop
        end if
    end for
    if word i did not get put into any group then
        Create a new group g
        Put word i into group g, incrementing group g's current size.
    end if
end for
```

Algorithm 6.1: Ordered first fit bin packing.

Table 6.1 shows the results of using the ordered first fit bin packing algorithm on the dictionaries (see Section 3.3.2 for details on the dictionaries). The first column

lists the dictionary and the total number of states in the whole dictionary. This is the sum of the number of states in each phone in each word in the dictionary. The column "MVL" shows the maximum vector length (the size of the group). "*Best*" lists the best possible packing, equal to $\lceil \frac{States}{\texttt{MVL}} \rceil$. This is the best possible packing, assuming you are allowed to break words up between groups. To compute the optimal packing assuming you cannot break up words is NP-complete, and is therefore not computed. The column "*Actual*" lists the actual number of groups computed by the bin packing algorithm. The closer this is to "*Best*", the better. "*Loss*" lists the percentage of overhead incurred by imperfect packing.

Name / States	mvl	*Best*	*Actual*	*Loss %*
Digits	16	7	7	
97	24	5	5	
	32	4	4	
	48	3	3	
	64	2	2	
	128	1	1	
	256	1	1	
Numbers	16	-	-	
336	24	14	15	7.1
	32	11	11	
	48	7	7	
	64	6	6	
	128	3	3	
	256	2	2	
Web	16	-	-	
941	24	40	41	2.5
	32	30	31	3.3
	48	20	20	
	64	15	15	
	128	8	8	
	256	4	4	

Name / States	mvl	*Best*	*Actual*	*Loss %*
SmallBN	16	-	-	
14793	24	-	-	
	32	463	470	1.5
	48	309	314	1.6
	64	232	235	1.2
	128	116	116	
	256	58	58	
MedBN	16	-	-	
74717	24	-	-	
	32	2335	2370	1.5
	48	1557	1585	1.8
	64	1168	1183	1.3
	128	584	586	0.3
	256	292	294	0.7
LargeBN	16	-	-	
485330	24	-	-	
	32	-	-	
	48	10112	10273	1.6
	64	7584	7681	1.3
	128	3792	3805	0.3
	256	1896	1906	0.5

Table 6.1: Bin packing on dictionaries.

Blank entries in the table represent vector lengths that are too short for the longest word in the dictionary to fit into a single vector register. For example, the

longest word in **Web** is "sixteen", with 19 states. Vector lengths shorter than 19 will therefore not work with this algorithm and dictionary.

Notice that all the dictionaries pack quite well for all legal choices of `MVL`. The worst percentage losses are for cases where only one extra group is created over the best possible case (e.g. 15 actual groups vs. 14 best possible groups for **Numbers** with vector length of 24).

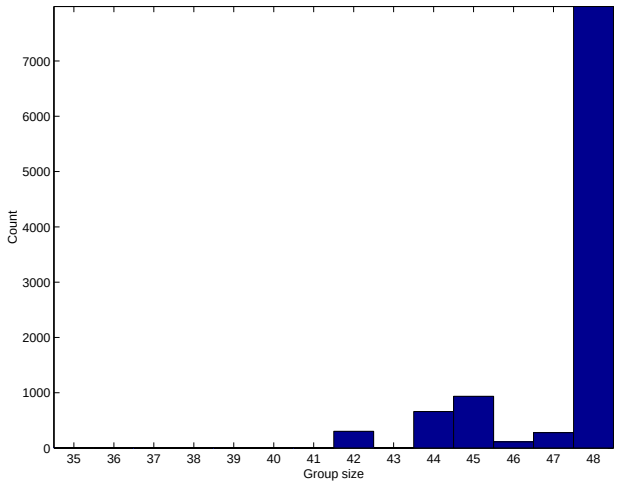

Figure 6.4: Histogram of group size for **LargeBN**, vector length 48

Another measure of the efficacy of the packing is the fullness of each group. Ideally, each group should have nearly the maximal number of states. Figure 6.4 shows a histogram of group size for **LargeBN** with vector length of 48. Notice that almost all groups are fully occupied[2].

One explanation for the good packing is that the distribution of word lengths (in number of states) is nearly normal, having many words near the median length, and few very long words. Figure 6.5 shows a histogram of the number of states per word for **LargeBN**.

[2]The scale of the figure does not allow it to be seen, but one group has size 35. The next smallest group has size 42, which is visible in the plot.

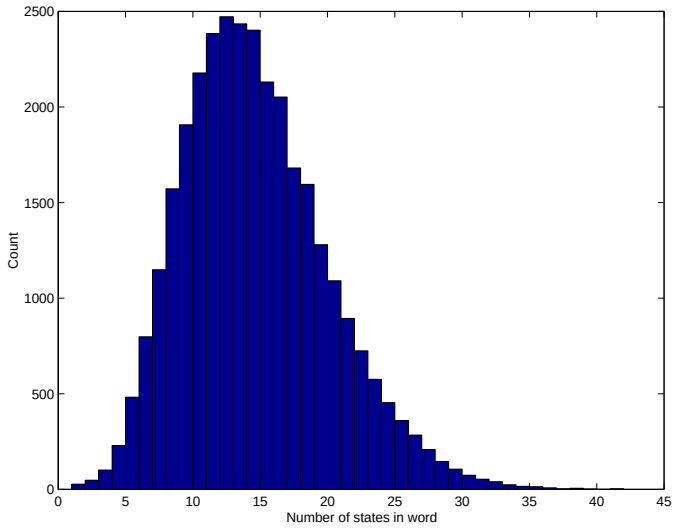

Figure 6.5: Histogram of number of states per word for **LargeBN**

6.1.2 An Algorithm for Vectorizing by State

Now that we have an algorithm for dividing words from the dictionary into groups that can be efficiently evaluated, we present the algorithm for computing the scores for each word.

To compute the best word in a dictionary, the dictionary is split into G groups as presented above. Each group is evaluated, producing one score for each word in the group. The scores can then be sorted to produce an ordered list, or just the best word can be output.

In the current work, the dictionary is assembled ahead of time. Since the algorithm for dividing the dictionary into groups is quite efficient, it would be possible to generate the groups on-the-fly. Either way, certain information (described below) is assumed to be available when a group of words is evaluated.

T — The number of frames in the utterance.

N — The number of states in the group (the sum of the number of states in each word in the group).

P — The number of output probabilities from the Phone Probability Estimator. Also the number of phones in the inventory.

self — The self-loop $-log$ probabilities of each state in the group. The length of **self** is N, and its elements can be determined when the dictionary is assembled.

exit — The exit $-log$ probability of each state in the group. The length of **exit** is N, and its elements can be determined when the dictionary is assembled.

X — The acoustic $-log$ probabilities. Its length is $P \cdot T$; X is computed by the Phone Probability Estimator.

S — The phone state indices. Each element is an offset into X, representing which phone state is being considered. Its length is N. S can be determined when the dictionary is assembled.

The key to the algorithm is that one can set certain elements of **exit** to infinity (zero probability) in such a way as to prevent "bleed through" from one word into another. Consider the dotted lines in Figure 6.3. These lines represent the dependence of elements in the table on the previous state. This dependence, if left unmodified, would cause the first state in one word to be dependent on the last state of the word to the left in the Viterbi table. For example, in Figure 6.3, the state **ax** in "about" is dependent on the state **t** in "abbott". Clearly, this would lead to incorrect results.

The dependence can be eliminated by setting the exit probability of the last state in each word to 0.0. Since **exit** is the $-log$ of this probability, one can set elements of **exit** corresponding to the last state of each word to infinity. This segmenting of the exit probabilities allows all the words in the group to be evaluated simultaneously using a very regular algorithm. Also, the exit probabilities for the group are all shifted one element to the right (so that vector element i gets stored in vector

element $i+1$). The algorithm requires the shifted exit probabilities, so storing them in shifted form saves an operation during decoding.

Some comments on initialization, running, and result reporting will help clarify the algorithm. To start the loop, the first row of the Viterbi table ($t = 0$) must be initialized. All elements get set to v_{max} (the maximum value able to be stored in a vector element) except for the first state in each word, which gets set to the appropriate $-log$ probability. Then, each row is evaluated in order. This represents forward steps in time. Finally, the score for the last state of each word in the group is output. One can either use the single best result, or store all the results for later processing.

The algorithm for decoding a single group is presented in Algorithm 6.2. To decode over all the groups in the dictionary, just repeat for each group.

Notice that the algorithm does not store the Viterbi table to memory. Rather, each row of the table is computed and stored in a vector register. Because each row only depends on the row immediately above it, the intermediate values need not be stored. By not saving these data, we do not have to perform any storage to memory in the inner loop. Although this makes the algorithm quite efficient, it does mean that the "backtrace" is not available. In other words, the algorithm only outputs the score for each word, not the path through the Viterbi table that was taken to arrive at the score. The backtrace would tell you the duration of each phone in the word (for the best path). In some applications, the backtrace may be necessary. In these cases, the algorithm can easily be modified to store the decision at each step (which way the *min* function went), at the cost of an extra store in the inner loop.

The algorithm is very efficient. Other than a small amount of overhead for loop maintenance and the vshift call, the inner loop performs only necessary arithmetic operations. On most architectures, all operations in the inner loop chain. Finally, as was shown in Table 6.1, packing is very efficient — most vector operations have vector length equal to `MVL`. Other than issues of redundant or irrelevant computations as discussed in the next chapter, the only drawback of this algorithm is the inability to run dictionaries with long words on short vector length machines.

VR1 — Row t of the Viterbi table.

VR2 — Row t of the Viterbi table shifted by one to the right.

VR3 — Scaled acoustic negative log probabilities for the group.

▷ Initialize VR1 with the first row of the Viterbi table ($t = 1$).

setvl N

vsets VR1, v_{max}

$i \leftarrow 1$

for each *word* in group **do**

    vinsert VR1, i, $X[S_i]$

    $i = i$+number of states in *word*

end for

▷ Now process the rest of the table.

for $t \leftarrow 2, T$ **do**

    vloadx VR3, S, $X + t \cdot P$ ▷ Load acoustic $-log$ probabilities into VR3.

    vshift VR2, VR1, 1 ▷ Shift previous row by one, store in VR2.

    vsaddvv VR1, VR1, **self** ▷ Add self-loop $-log$ probabilities.

    vsaddvv VR2, VR2, **exit** ▷ Add exit $-log$ probabilities.

    vmin VR1, VR1, VR2

    vsaddvv VR1, VR1, VR3 ▷ Add acoustic $-log$ probabilities.

end for

▷ Extract the results for each word.

$i \leftarrow 1$

for each *word* in group **do**

    $i = i$+number of states in *word*

    Output VR1[i] as score for *word*

end for

Algorithm 6.2: Decode a group of words vectorized by state

6.2 Vectorizing by Word

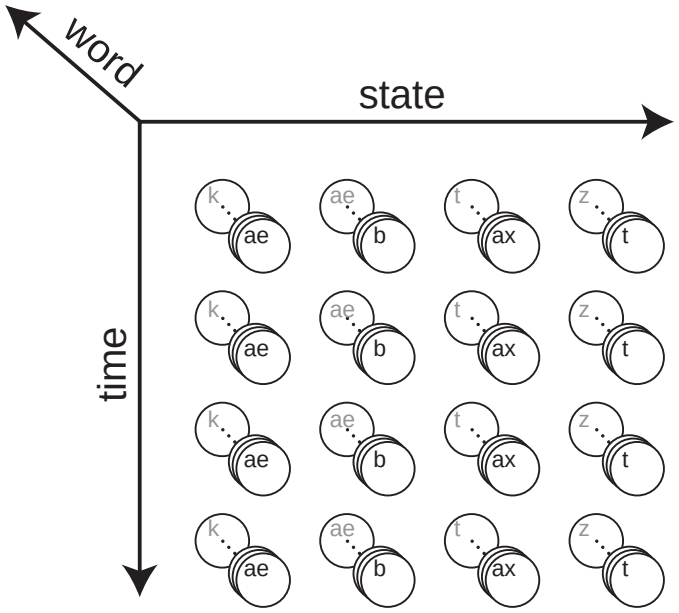

Figure 6.6: Viterbi tables for multiple words with equal number of states

Another approach is to vectorize across words containing similar number of states. Imagine several copies of Figure 6.2 layered one upon the other, with each copy representing a different word (see Figure 6.6). The algorithm proceeds simultaneously, lock-stepped for each word. To cover words that have different numbers of states, one simply repeats the process for the other word lengths, e.g. first all 1-state words are computed, then all 2-state words, etc. A large dictionary will guarantee that the vector lengths are long for all but a very few words. For example, Figure 6.7 shows a histogram of the number of phones in a word for a 65,000 word dictionary.

Of course, it is unlikely that a dictionary will contain exactly `MVL` words with a particular number of states. If words in the various "stacks" of Figure 6.6 have different numbers of states, then it is necessary to avoid computing the values for non-existent entries. One efficient way to do this is to sort the dictionary by the

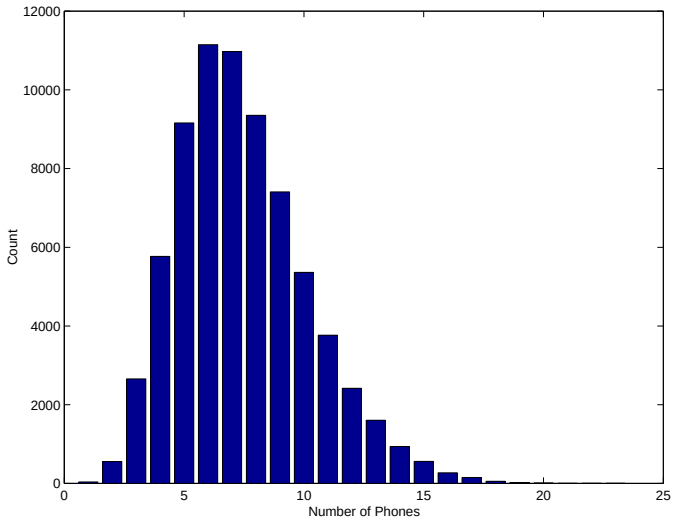

Figure 6.7: Histogram of the number of phones in a word for a 65,000 word dictionary.

number of states in each word from longest to shortest. Then take the first `MVL` words, and create a "stack", lining them up on the left. The width of the Viterbi tables (the number of states) will decrease as you go down the stack. By reducing the vector length as the stack is evaluated, the non-existent entries are avoided. This is illustrated in Figure 6.8. Repeat until all the words have been processed.

Vector architectures rely on the parallelism within a vector to mask memory latency and amortize instruction decode and control costs, so efficiency drops as the vector length decreases. This effect is especially pronounced on vector supercomputers, which may have `MVL` of 64 and need an actual vector length of at least 32 to attain reasonable efficiency. With vector extensions to conventional processors, which have very short vector lengths (typically 2 to 4 words), this is less of a problem.

Table 6.2 provides a lower bound on the efficiency of the algorithm. For each dictionary, the table lists the number of states in the dictionary, `MVL`, and the number of updates of the Viterbi table required assuming that setting the vector length less than `MVL` incurs no penalty. The *Efficiency* column assumes the penalty is

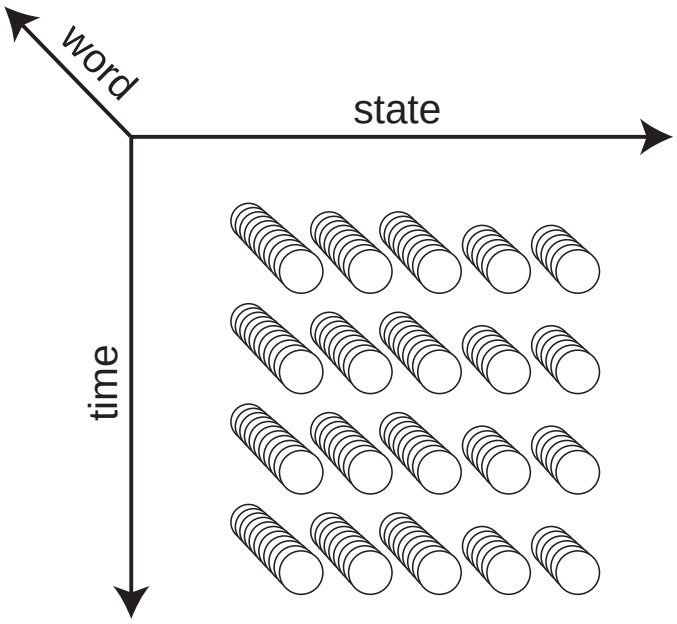

Figure 6.8: Viterbi tables for multiple words with decreasing number of states.

exactly proportional to the difference between MVL and the vector length. On real architectures, the efficiency will be somewhere between the listed efficiency and 100%.

Name / States	mvl	*Updates*	*Efficiency* %
Digits	2	100	97.0
97	4	110	88.2
	8	126	77.0
	16	156	62.2
	24	-	-
	32	-	-
	48	-	-
	64	-	-
Numbers	2	344	97.7
336	4	363	92.6
	8	393	85.5
	16	469	71.6
	24	505	66.5
	32	589	57.0
	48	-	-
	64	-	-
Web	2	950	99.1
941	4	964	97.6
	8	992	94.9
	16	1056	89.1
	24	1088	86.5
	32	1168	80.6
	48	1264	74.4
	64	1360	69.2

Name / States	mvl	*Updates*	*Efficiency* %
SmallBN	2	14806	99.9
14793	4	14834	99.7
	8	14874	99.5
	16	14957	98.9
	24	15054	98.3
	32	15149	97.7
	48	15311	96.6
	64	15583	94.9
MedBN	2	74728	100.0
74717	4	74750	100.0
	8	74802	99.9
	16	74907	99.7
	24	75027	99.6
	32	75099	99.5
	48	75243	99.3
	64	75483	99.0
LargeBN	2	485357	100.0
485330	4	485387	100.0
	8	485475	100.0
	16	485622	99.9
	24	485870	99.9
	32	485942	99.9
	48	486417	99.8
	64	486678	99.7

Table 6.2: Efficiency of vectorizing by word.

Blank entries indicate cases where MVL is greater than the number of pronunciations in the dictionary. Notice that the efficiency drops as MVL increases, especially for small vocabularies. This should not be surprising, since the range of lengths will increase as more words are included in a group.

6.2.1 An Algorithm for Vectorizing by Word

Algorithm 6.3 presents pseudocode for the algorithm. Most of the variables have similar meaning as in Section 6.1.2, but a group consists of words as assembled by the sorting algorithm described in Section 6.2.

T — The number of frames in the utterance.

P — The number of output probabilities from the Phone Probability Estimator. Also the number of phones in the inventory.

N — For each state index s, N_s is the number of words with at least s states. This is the height of a stack in column s of Figure 6.8. N_1 is therefore also the number of words in the group. Since the words are sorted from longest to shortest, $N_i >= N_{i+1}$. **N** can be determined when the dictionary is assembled.

self — The self-loop $-log$ probabilities. For each state index s, there is a different set of self-loop values. Set s has length N_s. The elements of **self** can be determined when the dictionary is assembled.

exit — The exit $-log$ probabilities. As with **self**, there is a different set for each state, and the values can be determined when the dictionary is assembled.

X — The acoustic $-log$ probabilities. Its length is $P \cdot T$; X is computed by the Phone Probability Estimator.

S — The phone state indices. Each element is an offset into X, representing which phone state is being considered. There is a different set of values for each state. The length of set s is N_s. **S** can be determined when the dictionary is assembled.

Unlike the "Vectorize by State" algorithm, this algorithm cannot store the Viterbi table just in vector registers. Rather, one vector register contains a set of scores for all the words in a group at time t and state s of the Viterbi table. To avoid storing the entire Viterbi table in memory, the algorithm proceeds one column (state) at

a time, storing only the current and the previous columns. In Algorithm 6.3, the variables **prev** and **cur** are memory buffers that are large enough to store an entire column of the Viterbi table. The size of **prev** and **cur** is $T \cdot N_1$, although fewer and fewer elements of **prev** and **cur** will be used as the algorithm proceeds across the columns of the Viterbi table. The algorithm maintains **prev** as the values for column $s-1$ and **cur** as the values for column s. After each column is evaluated, the roles of **cur** and **prev** are swapped.

By evaluating one column at a time, the algorithm avoids repeated memory accesses to load **self**, **exit**, and **S**. Instead, these are stored in vector registers. The vector registers need only be updated from memory each time the algorithm moves from one column to the next.

One drawback of this algorithm is the necessity of storing one column of the Viterbi table in memory. This requires an additional load and an additional store each time through the inner loop. Additional memory accesses are also required each time through the outer loop to load **self**, **exit**, and **S**. However, other than the extra memory requirements, the algorithm only performs needed arithmetic operations. On short vector length architectures, where vectorizing by state can be inefficient, this algorithm can perform well.

VR1 – Scores of row t state s of the Viterbi table.
VR2 – Scores of row $t-1$ state $s-1$ of the Viterbi table.
VR3 – Self-loop $-log$ probabilities for state s.
VR4 – Exit $-log$ probabilities for state s.
VR5 – Scaled acoustic $-log$ probabilities.
VR6 – Phone state index for state s.

Set all elements of **prev** to v_{max}
for $s \leftarrow 1$, maximum number of states in group **do**
    setvl N_s ▷ Set the vector length to height of stack at column s.
    vload VR3, **self** for column s
    vload VR4, **exit** for column s
    vload VR6, **S** for column s
    if $s = 1$ **then** ▷ Initialize first row.
        vloadx VR1, VR6, X ▷ First column gets acoustic scores.
    else
        vsets VR1, v_{max} ▷ Others get maximum possible score.
    end if
    for $t \leftarrow 2, T$ **do**
        vload VR2, $prev[(t-1) \cdot N_1]$ ▷ Load scores from $t-1$, $s-1$
        vloadx VR5, VR6, $X + t \cdot P$ ▷ Acoustic scores.
        vsaddvv VR1, VR1, VR3 ▷ Viterbi score at $t-1, s$ plus **self**
        vsaddvv VR2, VR2, VR4 ▷ Viterbi score at $t-1, s-1$ plus **exit**
        vmin VR1, VR1, VR2
        vsaddvv VR1, VR1, VR5
        vstore VR1, $cur[t \cdot N_1]$ ▷ Store score from t, s.
    end for
    vstore VR1, results for column s
    swap **cur** and **prev**
end for

Algorithm 6.3: Vectorize by word via dictionary sorting.

Chapter 7

Large Vocabulary Decoder

The decoding algorithms described in the previous chapter work quite well on small vocabularies. However, as the vocabularies get larger, it is efficacious to implement methods that avoid redundant or irrelevant computations. In this chapter, three such methods are presented. The difficulties of vectorizing such methods are discussed, and tradeoffs with using the small vocabulary methods of Chapter 6 for large vocabularies are presented.

7.1 Tree Structured Lexicons

As the vocabulary gets larger, more and more words end up sharing common prefixes. It is possible to arrange the evaluation of the decoder so that computation of a common prefix occurs only once for all the words that share the prefix. Figure 7.1 is an excerpt from a tree structured lexicon for the **Web** dictionary. The excerpt only includes words that start with "f". An implicit edge from every node to itself is omitted for clarity.

To read the figure, start at the root. Each node represents a speech sounds (phone). At each time step, either stay at the current node, or follow a child node. Continue until a node with a word is reached. For example, the word "four" can be found by starting at "f", staying there for some number of frames, going to "ao",

staying there for some number of frames, and ending at "r". Notice that words that start with the same sounds are all on the same branch. So one can continue after "r [four]" to "w", "axr", "dcl [forward]" to get to "forward".

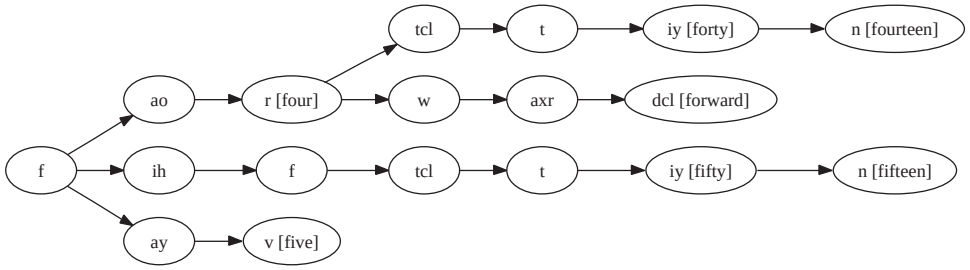

Figure 7.1: Excerpt from tree structured lexicon from the **Web** dictionary.

In the evaluation of the Viterbi table, one need not recompute the values at the nodes for common prefixes. Figure 7.2 shows an example of the evaluation of the word "forward". Notice that once the evaluation of "forward" is complete, all the information for the evaluation of "four" is already present. Looked at another way, if "four" has already been computed, only a little extra work (the last three columns of Figure 7.2) need be done to compute the value of "forward".

One efficient way to traverse the dictionary in a scalar decoder is to do a depth-first search through the tree structured lexicon [63] As the search encounters nodes that represent the end of a word, the algorithm outputs the word's probability. In terms of the Viterbi table, evaluation proceeds by adding columns to the right end of the table as the child of a node is visited, and removing columns from the right end of the table as the search returns to the parent node. The total size of the Viterbi table in such a search is bounded by the length of the longest word in the dictionary (in number of states). As such, this traversal strategy is very efficient in memory. And as desired, each prefix is only evaluated once.

The total savings over evaluating each word in the lexicon are dependent on how many words in the dictionary share prefixes. See Section 7.4 for details.

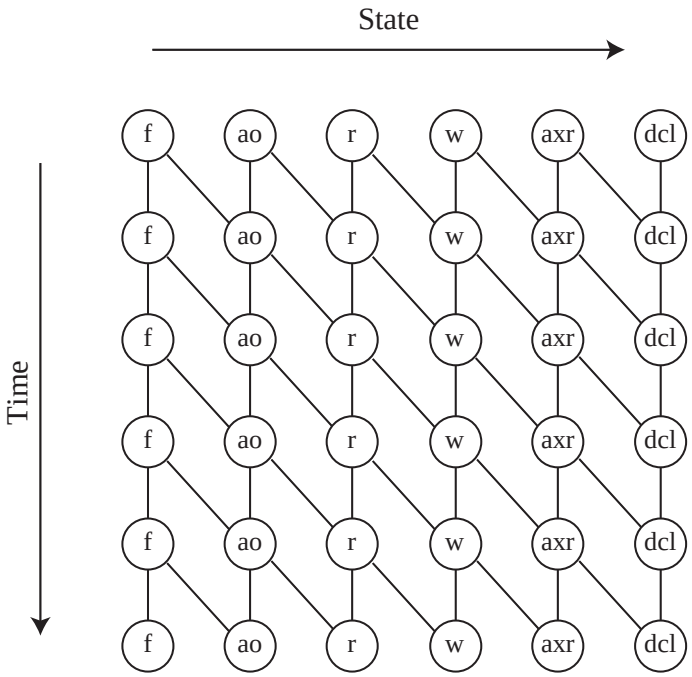

Figure 7.2: Viterbi Table for “four” and “forward””.

7.2 Pruning

In addition to avoiding the unnecessary recomputation of common prefixes, one can also prune the search using a number of strategies. In each case, the goal is to avoid computations where the result is very unlikely to be useful. In the following two subsections, two types of pruning are discussed.

7.2.1 Branch Pruning

The motivation for branch pruning is the observation that if a prefix is very unlikely to match the input stream, then any words starting with that prefix are also unlikely to match the input stream. To implement branch pruning, one keeps track of the current probability that the subword matches the input stream. This value is simply the value at the final time in the rightmost column of the Viterbi table as one traverses the tree structured lexicon. If this probability falls below a threshold, the current node and all its children are not evaluated. Any words occurring along the node's children are assumed to have probability 0.0%.

The threshold can be computed in a number of ways. An absolute value can be chosen. However, the scale of the probabilities is not generally known in advance, and therefore this is not typically used. A more common approach uses beam pruning [62]. In beam pruning, the score of the best match of the input stream to any previously seen subword is stored. The threshold is then picked to be a fixed distance below this best value. Choosing a small value for the distance reduces the amount of computation at the cost of possibly excluding the correct answer.

Instead of beam pruning, one can keep track of the N best fragments that have been seen, and prune anything that falls below the value of the Nth best fragment. This is equivalent to a kind of adaptive beam pruning [24], where the beam is adjusted dynamically to meet a memory constraint. For small vocabularies, N best pruning does not work too well, as few elements are pruned for large values of N (e.g. N near the size of the dictionary), and accuracy suffers if N is too small. To make

the results comparable across dictionaries, only beam pruning is used in the results presented below.

7.2.2 Phone Deactivation Pruning

What is the likelihood that the input stream matches the word "four" if no frame in the input stream contains /f/ with significant probability? Phone deactivation pruning assumes that the likelihood would be very low [61]. More precisely, if any frame i in the input stream has probability for a phone p that is less than a threshold ϵ, then the probability for p in frame i is set to 0.0%. Since the probability that a word matches the stream is proportional the product of the probabilities that each frame matches the correct phone, setting any phone to 0.0 causes the probability that the word matches to fall to 0.0 also. Evaluation of such a word can immediately cease. Varying ϵ allows a time vs. accuracy tradeoff. When combined with branch pruning, phone deactivation pruning will avoid computation of all words that share the common prefix, since their probability will also be 0.0.

7.3 Vectorizing Large Vocabulary Decoders

The advantage of the algorithms in Chapter 6 is that they are very regular, and therefore vectorize quite efficiently. To implement tree structured lexicons and pruning require more irregular computations, which are difficult to perform on a vector architecture. As the vocabulary gets larger and as pruning becomes more aggressive, the benefits of the methods in Section 7.1 and Section 7.2 become greater and greater. Significant speedup through vectorization of regular algorithms is required to outweigh the benefits of a pruned, tree structured scalar algorithm. Section 7.4 presents the tradeoffs between vocabulary size, vector vs. scalar, and pruning, and will show the amount of vectorization speedup required to compensate for the advantages of the more irregular scalar algorithms.

In the rest of this section the use of tree structured lexicons and pruning is

assumed, and it is shown why these algorithms vectorize poorly.

7.3.1 Vectorized Traversal of a Tree Structured Lexicon

To vectorize evaluation over a tree structured lexicon, one must assign nodes in the tree at a particular time to an element of a vector. One can conceptualize this in terms of a token passing algorithm [75]. In the token passing algorithm, there are an unlimited number of movable tokens, each of which has a node to which it is assigned, the current local score (equivalent to the score in the Viterbi table), and the time index. To advance the algorithm, pick a token from a node, copy it to one of the node's children in the tree (including itself), and update the local score. The score of a token at time t going from state r to state s is the likelihood from the phone probability estimator at time t and state s times the transition probability from state r to state s (self-loop probability of r if r is the same as s, and the exit probability of r if r is different from s). If a token already exists at the new node for the same time, remove all but the best scoring token. If all the children have been visited already, remove the original token. Terminate when all tokens are at the end time. Upon termination, each terminal state of a word will contain a token with the score for that word.

Different choices for how to pick which node to update and which child to visit reflect different traversal strategies. As an example, consider evaluation of a single word via the Viterbi table updates, where the entries in the table are updated from top to bottom (time axis) then left to right (state axis). The value of an entry in the Viterbi table at state index s and time t is computed as the minimum of the scores of the entries at $(s-1, t-1)$ and $(s, t-1)$, times the local score. This is equivalent to passing two tokens, one from each source. Under the token passing paradigm, the algorithm will pass the token with state index 1 at time index 1 to state 2 time 2, then pass a token from (2,1) to (2,2), then from (1,2) to (2,3), then (2,2) to (2,3), etc.

To vectorize, one must assign tokens to elements of a vector. All elements are

updated, conceptually simultaneously. The amount of work in an update is quite small — a vector *min* and two vector additions. If instead of a tree, one has a separate set of linear structures for each word, the problem becomes the same as described in the previous chapter. One could pick tokens with the same time, but different (adjacent) states, and end up with the "Vectorizing by State" algorithm of Section 6.1. Or one could pick one token from each word with the earliest time, and end up with the "Vectorizing by Word" algorithm of Section 6.2. With the tree structure, one must account for traversal of the children (branching) and handle leaf nodes (termination).

Branching

At a branch, a token can be passed to any of the children. In the course of the algorithm, a token will eventually be passed to every child, but for the purpose of this discussion, the order is not important. As the token is copied, the old token must remain in place so that the other children can be traversed. Either the new token or the old token must be saved. Either way, a token must be added to a list of tokens to be processed. Maintaining the list is a scalar operation, so each branch introduces a non-vectorizable operation. Since the computation of the token score update is only a few operations, even a small amount of scalar overhead reduces the performance. A large vocabulary will have a large number of branches. For example, the node with the largest number of branches in the **LargeBN** dictionary has 36 children, while the average is just under 2 children.

Termination

When the algorithm reaches a node with no children (a leaf), the token terminates. If not all elements of a vector reach a leaf at the same time, then an element of the vector is introduced that should not be operated upon. Many vector architectures provide a mask register, which allows the algorithm to specify which elements are active. However, as the vector becomes more and more sparse, efficiency is reduced.

Another option is to copy the sparse vector, leaving out the empty elements. This is known as vector compression, and is also supported on most vector architectures. A typical tree structured lexicon of a large dictionary has many such terminations, so compression must be done frequently (for example, the **LargeBN** dictionary has 21830 terminations, nearly 30% of the total number of nodes). This overhead adds to the cost of the vectorized large vocabulary decoder.

7.3.2 Vectorized Pruning

Pruning exacerbates the problem. If a branch is pruned, all tokens at the current node or any child of the current node must be removed. The removal operation itself usually requires a traversal of tree, which is as difficult to vectorize as the evaluation itself. Also, if any of the tokens are currently resident in a vector element, then either the vector must be compressed, or the mask must be set to avoid computation of that element. In either case, the number of elements in the vector shrinks, and efficiency declines. This is very similar to the case described above for when a token reaches a leaf node.

7.3.3 Memory Requirements

For any traversal strategy on the tree, the algorithm must keep track of which nodes have been visited at which time. This is the same as saying that the algorithm must store all the active tokens. The number of tokens can be minimized if one advances each token to the end of the tree as quickly as possible (depth-first traversal of the tree). In such a case, there is only one "active" node, the node in the traversal that is currently the furthest from the root. If there are n nodes in the tree from the root to the active node and t time intervals in the utterance, then only $n \times t$ tokens need to be stored. However, if a strategy other than depth-first is used, there can be more than one active node. The total number of tokens becomes $t \times \sum_{i=1}^{N} n_i$, where N is the number of active nodes, and n_i is the number of nodes between active node i and the root. Not only does more tokens mean more memory storage requirements,

it can also slow the algorithm because of memory hierarchy effects (e.g. if the tokens do not fit in cache).

7.4 Tradeoffs

For the reasons described above, it is very difficult to vectorize a large vocabulary decoder. The amount of work per update is small, and there are intrinsically non-vectorizable components. However, the small vocabulary decoders of Chapter 6 are quite efficient. This section discusses the tradeoffs between a scalar decoder that implements the methods of Section 7.1 and Section 7.2, and the vector decoders of Chapter 6.

To measure the exact tradeoffs between implementations of a scalar vs. vector decoder on a *particular* architecture is beyond the scope of this thesis. Instead, we compare only the number of arithmetic operations used to update the Viterbi table. This provides an estimate of the vector speedup that would be required for the vector algorithms to out-perform scalar algorithms that implement the methods described in Section 7.1 and Section 7.2.

Table 7.1 compares the algorithms of Section 7.1 and Section 7.2 with those of Chapter 6 on the various dictionaries. Since the efficiency of the "Vectorizing by State" algorithm of Section 6.1 and the "Vectorizing by Word" algorithm of Section 6.2 are quite high for their appropriate vector lengths, the table below assumes 100% efficiency for these algorithms.

The **Size** column lists the the number of pronunciations in the dictionary. See Section 3.3.2 for more details on the dictionaries. The remaining entries in the table are the number of times each algorithm updates an entry in the Viterbi table, normalized by the length of the input stream (since each algorithm does work proportional to the length of the input stream). For each algorithm, this involves three additions and a min operation, as presented in Equation 6.2. For the scalar algorithms, the table also lists the vector speedup required for the vector algorithm to

perform equally with the scalar algorithm[1].

The **Vector** column lists the number of updates of the Viterbi table for the dictionaries using the algorithms of the previous chapter. The column **Scalar with Tree Structured Lexicon** lists the number of updates and the required vector speedup for the algorithms of this chapter. The column *No Prune* assumes the dictionary is arranged as a tree structured lexicon as in Section 7.1, but no pruning is performed. The columns labeled *Light Prune* and *Heavy Prune* both use a tree structured lexicon, branch pruning (as outlined in Section 7.2.1), and phone deactivation pruning (as outlined in Section 7.2.2). The heavy pruning settings are appropriate for fast decoding at the cost of some accuracy. The light settings perform more computations, but incur fewer search errors. The particular settings for the values were taken from an evaluation of a speech recognition system on the Broadcast News corpus [16].

		Vector	**Scalar with Tree Structured Lexicon**					
Dictionary	**Size**		*No Prune*		*Light Prune*		*Heavy Prune*	
Digits	12	97	89	1.1×	89	1.1×	89	1.1×
Numbers	30	336	209	1.6×	148	2.3×	133	2.5×
Web	79	941	485	1.9×	329	2.9×	340	2.8×
SmallBN	1000	14793	9720	1.5×	5820	2.5×	3501	4.2×
MedBN	5000	74717	37347	2.0×	18518	4.0×	3721	20.1×
LargeBN	32010	485330	151901	3.2×	43313	11.2×	13633	35.6×

Table 7.1: Number of updates to the Viterbi table and vector architecture speedup required for the vector algorithms to perform equally with the tree structured and pruned scalar algorithms.

Figure 7.3 shows the same data in graphical form. The vertical axis is the vector speedup required. The horizontal axis is the dictionary size (not to scale). Each line represents a different choice for pruning. The horizontal line at 4× indicates the approximate speedup of vector extensions to conventional processors. The line at 10× represents the approximate speedup for vector supercomputers and vector

[1]For example, a vector speedup of 4.0× is required for a vector processor implementing the methods of Chapter 6 to perform equally with an equivalent scalar processor implementing the methods of this chapter with light pruning on the **MedBN** dictionary.

microprocessors. See Section 1.2 for an overview on vector speedups for various architectures.

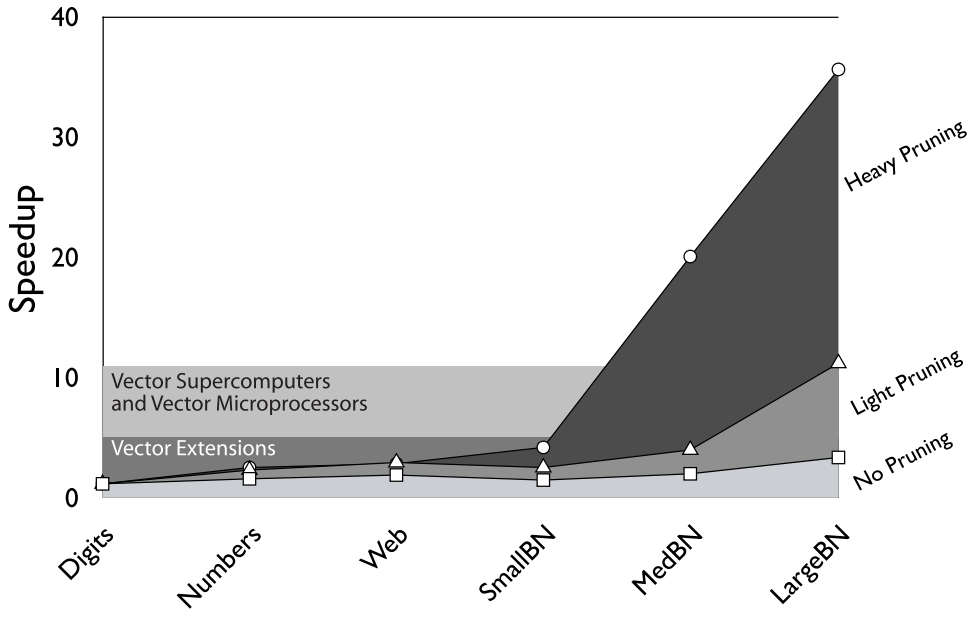

Figure 7.3: Vector architecture speedup required for the vector algorithms to perform equally with the tree structured and pruned scalar algorithms.

Roughly speaking, the speedup provided by a vector architecture must be larger than the indicated speedup for the desired pruned case for the vectorized algorithm to compare favorably with the scalar algorithm.

In the case of long vector length architectures, there is an additional advantage to the "Vectorizing by State" algorithm. It is very memory efficient, both in terms of storage, and in terms of memory accesses in the inner loop. As a result, the vectorize by state algorithm will likely run faster than the typical speedups listed.

For small vocabularies, the vectorized algorithms are competitive with the scalar algorithms. As the vocabulary gets larger, the vector speedup must increase for the vectorized algorithms to be competitive. The problem is exacerbated by high levels of pruning.

Given the fast pace of improvements in the capabilities of scalar processors, it is unlikely that the current batch of vector processors will be competitive for large vocabularies and high pruning levels. However, there is no intrinsic reason that vector processors cannot improve at the same rate as scalar processors. Given an equivalent architecture, a vector processor remains competitive for all but the largest dictionaries and highest pruning levels.

Chapter 8

Conclusions

Speech recognition on vector architectures would benefit many applications, including dictation on the desktop, command and control of PDAs and cellphones, and automated call centers using supercomputers. Low power consumption, high absolute performance, and low cost all contribute to the value accrued to vector architectures. To realize these benefits, speech recognition algorithms must be vectorized to run on these platforms.

For this thesis, a vector simulation library was developed to aid in the analysis of speech recognition algorithms on vector architectures. The vector simulation library implements many of the common opcodes found on vector processors, but does not attempt to simulate the fine details of the architectures (cache, chaining behavior, etc.). Instead, the focus of the research is on generating code that vectorizes well on any vector processor.

Of the three major components of ICSI's hybrid speech recognition system, two vectorize quite well. The signal processing component's principle computational bottleneck is the computation of the filterbank, which is typically implemented using a Fast Fourier Transform (FFT). Since the FFT is used in many, many application for which vector processors are used, most architectures provide some support for FFT computations. The other elements of the signal processing component typically vectorize quite well.

The phone probability estimator used in this work is implemented as a multi-layer perceptron (MLP). The computational bottleneck of an MLP is a matrix-matrix multiply. This operation is quite regular, and vectorizes well. However, for optimal performance, fine details of the memory hierarchy must be taken into account. Since such details vary widely, we advocate a generate-and-test approach, where many algorithms are generated automatically, and the fastest is used. Several algorithms were presented that would form the basis for the automatically generated code.

The case of the final component, the decoder, is divided into small and large vocabularies. For large vocabularies, it is desirable to avoid repeatedly computing common prefixes of words (e.g. "four", "fourteen", "forty", "forward"). Also, one can use several methods to avoid altogether the computation of some of the words. For small vocabularies, the savings using these methods are less important, and it is acceptable to simply evaluate each word in full.

Two algorithms were presented for small vocabularies, where every word in the dictionary is evaluated. The first involves batching together words such that the summed length (in states) of a batch is equal to the vector length. The algorithm vectorizes along the state axis of the Viterbi table. The batches are computed using a bin packing algorithm, and all the dictionaries packed quite well. The algorithm itself vectorizes efficiently, and accesses memory minimally. However, it depends on reasonably long vectors, making it unsuitable for some architectures.

The other algorithm batches together words with similar numbers of states. The algorithm vectorizes by word, such that elements of a vector each hold a state from a different word. The algorithm vectorizes well, although it requires extra memory accesses avoided by the algorithm described above. Also, for long vector length architectures, the efficiency can be low, as not all the vectors will be full. For short vector length architectures, however, the efficiency is excellent for all but the smallest dictionary.

For large vocabularies, no method was found that vectorizes efficiently when pruning and tree structured lexicons are used. The base of the problem is that the tree structured lexicons are bushy and unbalanced. The amount of work necessary

to arrange for a vectorized operation is nearly the same as just performing the operation directly. Comparisons with the vectorizable small vocabulary systems give an indication of the vector speedup required for a particular dictionary to run more efficiently on vector processor than on a scalar processor. The vectorized small vocabulary system is competitive for all but the largest dictionaries and the highest pruning levels.

A possible future direction is a mixed-mode operation, where a scalar algorithm determines what operations to perform, and a vector algorithm does the actual work. If the scalar and vector algorithms can run in parallel, this method may allow efficient vectorization of large vocabulary systems.

Bibliography

[1] D. Aberdeen and J. Baxter. Emmerald: a fast matrix-matrix multiply using Intel's SSE instructions. *Concurrency and Computation: Practice and Experience*, 13(2):103–119, 2001.

[2] R. C. Agarwal, J. W. Cooley, F. G. Gustavson, J. B. Shearer, G. Slishman, and B. Tuckerman. New scalar and vector elementary functions for the IBM System/370. *IBM Journal of Research and Development*, 30(2):126–144, 1986.

[3] V. H. Allan, R. B. Jones, R. M. Lee, and S. J. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, 1995.

[4] G.M. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 30, pages 483–485, Atlantic City, New Jersey, USA, 1967.

[5] A. W. Appel and A. Bendiksen. Vectorized garbage collection. *Journal of Supercomputing*, 3:151–160, 1989.

[6] Apple. Vector libraries web page. http://developer.apple.com/hardware/ve/vector_libraries.html.

[7] K. Asanovic. *Vector Microprocessors*. PhD thesis, University of California at Berkeley, May 1998.

[8] D. Bailey. A high-performance fast Fourier transform algorithm for the Cray-2. *The Journal of Supercomputing*, 1(1):43–60, July 1987.

[9] D. Bailey. Extra-high speed matrix multiplication on the Cray-2. *SIAM Journal on Scientific and Statistical Computing*, 9(3):603–607, May 1988.

[10] R. Bhargava, L. K. John, B. L. Evans, and R. Radhakrishnan. Evaluating MMX technology using DSP and multimedia applications. In *Proceedings of the 31st IEEE International Symposium on Microarchitecture*, pages 37–46, Dallas, Texas, USA, November 1998.

[11] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing*, pages 340–347, Vienna, Austria, July 1997.

[12] H. Bourlard and N. Morgan. Merging multilayer perceptrons & hidden Markov models: Some experiments in continuous speech recognition. In E. Gelenbe, editor, *Artificial Neural Networks: Advances and Applications*. North Holland Press, 1991.

[13] H. Bourlard and N. Morgan. *Connectionist Speech Recognition: A Hybrid Approach*. Kluwer Academic Publishers, 1993.

[14] J. S. Bridle. Alpha-nets: a recurrent neural network architecture with a hidden Markov model interpretation. *Speech Communications*, 9(1):83–92, February 1990.

[15] J. S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In *Neurocomputing: Algorithms, Architectures and Applications*, pages 227–236. Springer, Berlin, 1990.

[16] G. Cook, J. Christie, D. Ellis, E. Fosler-Lussier, Y. Gotoh, B. Kingsbury, N. Morgan, S. Renals, T. Robinson, and G. Williams. The SPRACH system for the transcription of broadcast news. In *DARPA Broadcast News Workshop*, Herndon, Virginia, February 1999.

[17] S. B. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 28(4):357–366, August 1980.

[18] J. Deller, J. Proakis, and J. Hansen. *Discrete-Time Processing of Speech Signals.* Macmillan Publishing, New York, 1993.

[19] L. Deng, M. Lennig, F. Seitz, and P. Mermelstein. Large vocabulary word recognition using context-dependent allophonic hidden Markov models. *Computer Speech and Language*, 4:345–357, 1990.

[20] R. Espasa, M. Valero, D. Padua, M. Jiménez, and E. Ayguadé. Quantitative analysis of vector code. In *Proceedings of the 3rd Euromicro Workshop on Parallel and Distributed Processing*, January 1995.

[21] H. Fletcher. Auditory patterns. *Reviews of Modern Physics*, 22:47–65, 1940.

[22] J. T. Foote. *Tree-based Probability Estimation for HMM Speech Recognition.* PhD thesis, Brown University, Providence, RI, June 1993.

[23] J. Hake and W. Homberg. The impact of memory organization on the performance of matrix multiplication. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 34–40, November 1990.

[24] H. Van Hamme and F. Van Aelten. An adaptive-beam pruning technique for continuous speech recognition. In *Proceedings of the 4th Int'l Conference on Spoken Language Processing (ICSLP-96)*, volume 4, pages 2083–2086, Philadelphia, PA, October 1996.

[25] R. W. Hamming. *Digital Filters.* Englewood Cliffs, New Jersey, 1983.

[26] J. L. Hennessy and D. A. Patterson. *Computer Architecture — A Quantative Approach.* Morgan Kaufmann, 1990.

[27] H. Hermansky. Perceptual linear predictive (PLP) analysis of speech. *J. Acoustical Society of America*, 87(4), April 1990.

[28] P. Hoffman. *The Man Who Loved Only Numbers: The Story of Paul Erdos and the Search for Mathematical Truth.* Hyperion, 1998.

[29] S. Huss-Lederman, E. M. Jacobson, A. Tsao, T. Turnbull, and J. R. Johnson. Implementation of Strassen's algorithm for matrix multiplication. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing*, page 32, 1996.

[30] F. G. Gustafson J. J. Dongarra and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, 1984.

[31] A. Janin, D. W. Ellis, and N. Morgan. Multistream: Ready for prime-time? In *6th European Conference on Speech Communication and Technology (Eurospeech-99)*, volume 2, pages 591–594, Budapest, September 1999.

[32] F. Jelinek. Fast sequential decoding algorithm using a stack. *IBM Journal of Research and Development*, 13(6):675–685, 1969.

[33] D. S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, pages 256–278, August 1974.

[34] S. Joshi and P. Dubey. Some fast speech processing algorithms using AltiVec technology. In *Proceedings IEEE Int'l Conference on Acoustics, Speech, & Signal Processing (ICASSP-99)*, pages 2135–2138, Phoenix, March 1999.

[35] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, and S. Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–47, March 2001.

[36] K. Kirchhoff and J. Bilmes. Dynamic classifier combination in hybrid speech recognition systems using utterance-level confidence values. In *Proceedings IEEE Int'l Conference on Acoustics, Speech, & Signal Processing (ICASSP-99)*, pages 693–696, Phoenix, March 1999.

[37] D. E. Knuth. *The Art of Computer Programming, Vol.2: Semi-Numerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 1969.

[38] L. Kohn, G. Maturana, M. Tremblay, A. Prabhu, and G. Zyner. The visual instruction set (VIS) in UltraSPARC. In *Proceedings of the 40th IEEE Computer Society International Conference*, pages 462–469, March 1995.

[39] C. E. Kozyrakis. *Scalable vector media-processors for embedded systems.* PhD thesis, University of California at Berkeley, 2002.

[40] C. E. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable processors in the billion-transistor era: IRAM. *IEEE Computer*, 30(9):75–78, September 1997.

[41] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74. ACM Press, 1991.

[42] R. Lee. Accelerating multimedia with enhanced microprocessors. *IEEE Micro*, 15(2):22–32, April 1995.

[43] R. Lee. Realtime MPEG video via software decompression on a PA-RISC processor. In *Proceedings of the 40th IEEE Computer Society International Conference*, pages 186–192, San Francisco, CA, USA, March 1995.

[44] R. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, August 1996.

[45] R. Lee. Multimedia extensions for general-purpose processors. *IEEE Workshop on Signal Processing Systems*, pages 9–23, November 1997.

[46] O. Lubeck, J. Moore, and R. Mendez. A benchmark comparison of three supercomputers: Fujitsu VP-200, Hitachi S810/20 and Cray X-MP/2. *Computer*, 18(12):10–24, December 1985.

[47] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley and Sons, 1990.

[48] D. Martin. *Vector Extensions to the MIPS-IV Instruction Set Architecture (The V-IRAM Architecture Manual).* Available through IRAM web pages at http://iram.cs.berkeley.edu, March 2000.

[49] N. Mirghafori and N. Morgan. Combining connectionist multi-band and full-band probability streams for speech recognition of natural numbers. In *Proceedings of the 5th Int'l Conference on Spoken Language Processing (ICSLP-98)*, pages 743–746, Sydney, Australia, November 1998.

[50] Motorola. AltiVec web page. http://motorola.com/altivec.

[51] T. Nguyen, A. Zakhor, and K. Yelick. Performance analysis of an H.263 video encoder on VIRAM. *International Conference on Image Processing (ICIP)*, September 2000.

[52] L. Oliker, A. Canning, J. Carter, J., and S. Ethier. Scientific computations on modern parallel vector systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, Pittsburgh, Pennsylvania, USA, November 2004.

[53] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[54] D. Paul. An efficient A* stack decoder algorithm for continuous speech recognition with a stochastic language model. In *Proceedings IEEE Int'l Conference on Acoustics, Speech, & Signal Processing (ICASSP-92)*, pages 25–28, San Francisco, 1992.

[55] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, 16(4):42–50, August 1996.

[56] R. Perron and C. Mundie. The architecture of the Alliant FX/8 computer. In *Proceedings of the IEEE Computer Society International Conference*, pages 390–394, Washington, D.C., March 1986.

[57] C. Philip and P. Moreno. On the use of support vector machines for phonetic classification. In *Proceedings IEEE Int'l Conference on Acoustics, Speech, & Signal Processing (ICASSP-99)*, Phoenix, March 1999.

[58] L. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2), February 1989.

[59] S. K. Raman, V. Pentkovski, and J. Keshava. Implementing streaming SIMD extensions on the Pentium III processor. *IEEE Micro*, 20(4):47–57, July 2000.

[60] P. Ranganathan, S. V. Adve, and N. P. Jouppi. Performance of image and video processing with general-purpose processors and media ISA extensions. In *Proceedings of the 26th Annual International Symposium Computer Architecture (ISCA)*, pages 124–135, Atlanta, Georgia, USA, May 1999.

[61] S. Renals. Phone deactivation pruning in large vocabulary continuous speech recognition. In *IEEE Signal Processing Letters*, volume 3, pages 4–6, January 1996.

[62] S. Renals and M. Hochberg. Start-synchronous search for large vocabulary continuous speech recognition. In *IEEE Transactions on Speech and Audio Processing*, volume 7, pages 542–553, July 1999.

[63] T. Robinson and J. Christie. Time-first search for large vocabulary speech recognition. In *Proceedings IEEE Int'l Conference on Acoustics, Speech, & Signal Processing (ICASSP-98)*, pages 829–832, Seattle, WA, May 1998.

[64] R. M. Russell. The Cray-1 computer system. *Communications of the ACM*, 21(1):63–72, January 1978.

[65] N. N. Schraudolph. A fast, compact approximation of the exponential function. *Neural Compututing*, 11(4):853–862, May 1999.

[66] Sony. PlayStation web page. http://www.us.playstation.com.

[67] SourceForge. libmpeg2 web page. http://libmpeg2.sourceforge.net.

[68] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[69] P. Tang. Table-driven implementation of the logarithm function in IEEE floating point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378–400, 1990.

[70] S. Thakkar and T. Huff. The Internet streaming SIMD extensions. *Intel Technology Journal*, 3(Q2):1–8, 1999.

[71] R. Thomas and K. Yelick. Efficient FFTs on IRAM. In *Proceedings of the 1st Workshop on Media Processors and DSPs (MICRO-32)*, Haifa, Israel, November 1999.

[72] N. Uchida, M. Hirai, M. Yoshida, and K. Hotta. Fujitsu VP2000 series. In *IEEE Computer Society International Conference (CompCon)*, pages 4–11, Los Alamitos, California, USA, February 1990.

[73] T. Watanabe. The NEC SX-3 supercomputer system. In *IEEE Computer Society International Conference (CompCon)*, pages 303–308, San Francisco, California, USA, February 1991.

[74] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[75] S. J. Young, N. H. Russell, and J. H. S. Thornton. Token passing: A simple conceptual model for connected speech recognition systems. Technical Report TR38, Cambridge University Engineering Department, 1989. Available at http://mi.eng.cam.ac.uk/reports/svr-ftp/young_tr38.ps.Z.