N-gram language models for massively parallel devices

Nikolay Bogoychev and **Adam Lopez**
University of Edinburgh
Edinburgh, United Kingdom

Abstract

For many applications, the query speed of N-gram language models is a computational bottleneck. Although massively parallel hardware like GPUs offer a potential solution to this bottleneck, exploiting this hardware requires a careful rethinking of basic algorithms and data structures. We present the first language model designed for such hardware, using B-trees to maximize data parallelism and minimize memory footprint and latency. Compared with a single-threaded instance of KenLM (Heafield, 2011), a highly optimized CPU-based language model, our GPU implementation produces identical results with a smaller memory footprint and a sixfold increase in throughput on a batch query task. When we saturate both devices, the GPU delivers nearly twice the throughput per hardware dollar even when the CPU implementation uses faster data structures.

Our implementation is freely available at `https://github.com/XapaJIaMnu/gLM`

1 Introduction

N-gram language models are ubiquitous in speech and language processing applications such as machine translation, speech recognition, optical character recognition, and predictive text. Because they operate over large vocabularies, they are often a computational bottleneck. For example, in machine translation, Heafield (2013) estimates that decoding a single sentence requires a million language model queries, and Green et al. (2014) estimate that this accounts for more than 50% of decoding CPU time.

To address this problem, we turn to massively parallel hardware architectures, exemplified by general purpose graphics processing units (GPUs), whose memory bandwidth and computational throughput has rapidly outpaced that of CPUs over the last decade (Figure 1). Exploiting this increased power is a tantalizing prospect for any computation-bound problem, so GPUs have begun to attract attention in natural language processing, in problems such as parsing (Canny et al., 2013; Hall et al., 2014), speech recognition (Chong et al., 2009; Chong et al., 2008), and phrase extraction for machine translation (He et al., 2015). As these efforts have shown, it is not trivial to exploit this computational power, because the GPU computational model rewards data parallelism, minimal branching, and minimal access to global memory, patterns ignored by many classic NLP algorithms (Section 2).

We present the first language model data structure designed for this computational model. Our data structure is a trie in which individual nodes are represented by B-trees, which are searched in parallel (Section 3) and arranged compactly in

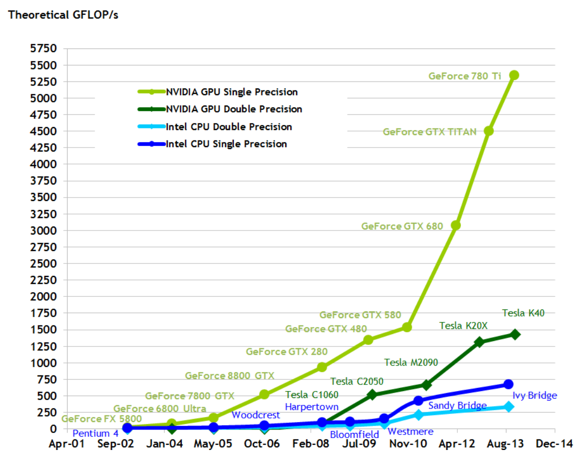

Figure 1: Theoretical floating point performance of CPU and GPU hardware over time (Nvidia Corporation, 2015).

memory (Section 4). Our experiments across a range of parameters in a batch query setting show that this design achieves a throughput six times higher than KenLM (Heafield, 2011), a highly efficient CPU implementation (Section 5). They also show the effects of device saturation and of data structure design decisions.

2 GPU computational model

GPUs and other parallel hardware devices have a different computational profile from widely-used x86 CPUs, so data structures designed for serial models of computation are not appropriate. To produce efficient software for a GPU we must be familiar with its design (Figure 2).

2.1 GPU design

A GPU consists of many simple computational *cores*, which have neither complex caches nor branch predictors to hide latencies. Because they have far fewer circuits than CPU cores, GPU cores are much smaller, and many more of them can fit on a device. So the higher throughput of a GPU is due to the sheer number of cores, each executing a single *thread* of computation (Figure 2). Each core belongs to a *Streaming Multiprocessor (SM)*, and all cores belonging to a SM must execute the same instruction at each time step, with exceptions for branching described below. This execution model is very similar to single instruction, multiple data (SIMD) parallelism.[1]

Computation on a GPU is performed by an inherently parallel function or *kernel*, which defines a *grid* of data elements to which it will be applied, each processed by a *block* of parallel threads. Once scheduled, the kernel executes in parallel on all cores allocated to blocks in the grid. At minimum, it is allocated to a single *warp*—32 cores on our experimental GPU. If fewer cores are requested, a full warp is still allocated, and the unused cores idle.

A GPU offers several memory types, which differ in size and latency (Table 1). Unlike a CPU program, which can treat memory abstractly, a GPU program must explicitly specify in which physical memory each data element resides. This choice has important implications for efficiency that entail design tradeoffs, since memory closer to a core is small and fast, while memory further away is large and slow (Table 1).

[1]Due to differences in register usage and exceptions for branching, this model is not pure SIMD. Nvidia calls it SIMT (single instruction, multiple threads).

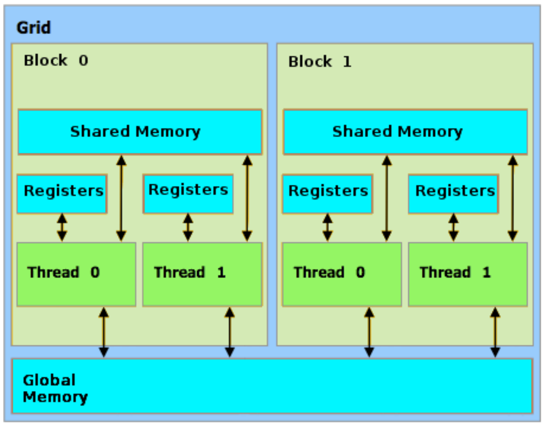

Figure 2: GPU memory hierarchy and computational model (Nvidia Corporation, 2015).

Memory type	Latency	Size
Register	0	4B
Shared	4–8	16KB–96KB
Global GPU	200–800	2GB–12GB
CPU	10K+	16GB–1TB

Table 1: Latency (in clock cycles) and size of different GPU memory types. Estimates are adapted from Nvidia Corporation (2015) and depend on several aspects of hardware configuration.

2.2 Designing efficient GPU algorithms

To design an efficient GPU application we must observe the constraints imposed by the hardware, which dictate several important design principles.

Avoid branching instructions. If a branching instruction occurs, threads that meet the branch condition run while the remainder idle (a *warp divergence*). When the branch completes, threads that don't meet the condition run while the first group idles. So, to maximize performance, code must be designed with little or no branching.

Use small data structures. Total memory on a state-of-the-art GPU is 12GB, expected to rise to 24GB in the next generation. Language models that run on CPU frequently exceed these sizes, so our data structures must have the smallest possible memory footprint.

Minimize global memory accesses. Data in the CPU memory must first be transferred to the device. This is very slow, so data structures must reside in GPU memory. But even when they

Data structure	Size	Query speed	Ease of backoff	Construction time	Lossless
Trie (Heafield, 2011)	Small	Fast	Yes	Fast	Yes
Probing hash table (Heafield, 2011)	Larger	Faster	Yes	Fast	Yes
Double array (Yasuhara et al., 2013)	Larger	Fastest	Yes	Very slow	Yes
Bloom filter (Talbot and Osborne, 2007)	Small	Slow	No	Fast	No

Table 2: A survey of language model data structures and their computational properties.

reside in global GPU memory, latency is high, so wherever possible, data should be accessed from shared or register memory.

Access memory with coalesced reads. When a thread requests a byte from global memory, it is copied to shared memory along with many surrounding bytes (between 32 and 128 depending on the architecture). So, if consecutive threads request consecutive data elements, the data is copied in a single operation (a *coalesced read*), and the delay due to latency is incurred only once for all threads, increasing throughput.

3 A massively parallel language model

Let w be a sentence, w_i its ith word, and N the order of our model. An N-gram language model defines the probability of w as:

$$P(w) = \prod_{i=1}^{|w|} P(w_i | w_{i-1} ... w_{i-N+1}) \qquad (1)$$

A backoff language model (Chen and Goodman, 1999) is defined in terms of n-gram probabilities $P(w_i|w_{i-1}...w_{i-n+1})$ for all n from 1 to N, which are in turn defined by n-gram parameters $\hat{P}(w_i...w_{i-n+1})$ and backoff parameters $\beta(w_{i-1}...w_{i-n+1})$. Usually $\hat{P}(w_i...w_{i-n+1})$ and $\beta(w_{i-1}...w_{i-n+1})$ are probabilities conditioned on $w_{i-1}...w_{i-n+1}$, but to simplify the following exposition, we will simply treat them as numeric parameters, each indexed by a reversed n-gram. If parameter $\hat{P}(w_i...w_{i-n+1})$ is nonzero, then:

$$P(w_i|w_{i-1}...w_{i-n+1}) = \hat{P}(w_i...w_{i-n+1})$$

Otherwise:

$$P(w_i|w_{i-1}...w_{i-n+1}) = P(w_i|w_{i-1}...w_{i-n+2}) \times \beta(w_{i-1}...w_{i-n+1})$$

This recursive definition means that the probability $P(w_i|w_{i-1}...w_{i-N+1})$ required for Equation 1 may depend on multiple parameters. If r $(< N)$ is the largest value for which $\hat{P}(w_i|w_{i-1}...w_{i-r+1})$ is nonzero, then we have:

$$P(w_i|w_{i-1}...w_{i-N+1}) = \hat{P}(w_i...w_{i-r+1}) \prod_{n=r+1}^{N} \beta(w_{i-1}...w_{i-n+1}) \qquad (2)$$

Our data structure must be able to efficiently access these parameters.

3.1 Trie language models

With this computation in mind, we surveyed several popular data structures that have been used to implement N-gram language models on CPU, considering their suitability for adaptation to GPU (Table 2). Since a small memory footprint is crucial, we implemented a variant of the trie data structure of Heafield (2011). We hypothesized that its slower query speed compared to a probing hash table would be compensated for by the throughput of the GPU, a question we return to in Section 5.

A trie language model exploits two important guarantees of backoff estimators: first, if $\hat{P}(w_i...w_{i-n+1})$ is nonzero, then $\hat{P}(w_i...w_{i-m+1})$ is also nonzero, for all $m < n$; second, if $\beta(w_{i-1}...w_{i-n+1})$ is one, then $\beta(w_{i-1}...w_{i-p+1})$ is one, for all $p > n$. Zero-valued n-gram parameters and one-valued backoff parameters are not explicitly stored. To compute $P(w_i|w_{i-1}...w_{i-N+1})$, we iteratively retrieve $\hat{P}(w_i...w_{i-m+1})$ for increasing values of m until we fail to find a match, with the final nonzero value becoming $\hat{P}(w_i...w_{i-r+1})$ in Equation 2. We then iteratively retrieve $\beta(w_{i-1}...w_{i-n+1})$ for increasing values of n starting from $r + 1$ and continuing until $n = N$ or we fail to find a match, multiplying all retrieved terms to compute $P(w_i|w_{i-1}...w_{i-N+1})$ (Equation 2). The trie is designed to execute these iterative parameter retrievals efficiently.

Let Σ be a our vocabulary, Σ^n the set of all n-grams over the vocabulary, and $\Sigma^{[N]}$ the

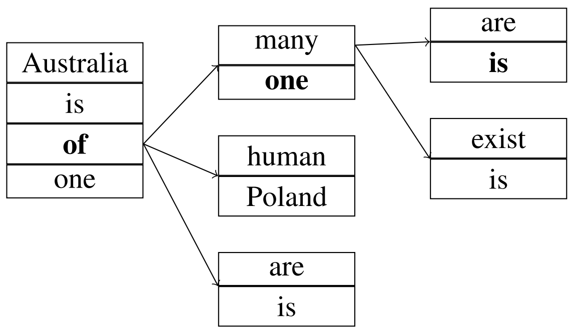

Figure 3: Fragment of a trie showing the path of N-gram **is one of** in bold. A query for the N-gram **every one of** traverses the same path, but since **every** is not among the keys in the final node, it returns the n-gram parameter $\hat{P}(\textbf{of}|\textbf{one})$ and returns to the root to seek the backoff parameter $\beta(\textbf{every one})$. Based on image from Federico et al. (2008).

set $\Sigma^1 \cup ... \cup \Sigma^N$. Given an n-gram *key* $w_i...w_{i-n+1} \in \Sigma^{[N]}$, our goal is to retrieve *value* $\langle \hat{P}(w_i...w_{i-n+1}), \beta(w_i...w_{i-n+1})\rangle$. We assume a bijection from Σ to integers in the range $1, ..., |\Sigma|$, so in practice all keys are sequences of integers.

When $n = 1$, the set of all possible keys is just Σ. For this case, we can store keys with nontrivial values in a sorted array A and their associated values in an array V of equal length so that $V[j]$ is the value associated with key $A[j]$. To retrieve the value associated with key k, we seek j for which $A[j] = k$ and return $V[j]$. Since A is sorted, j can be found efficiently with binary or interpolated search (Figure 4).

When $n > 1$, queries are recursive. For $n < N$, for every $w_i...w_{i-n+1}$ for which $\hat{P}(w_i...w_{i-n+1}) > 0$ or $\beta(w_i...w_{i-n+1}) < 1$, our data structure contains associated arrays $K_{w_i...w_{i-n+1}}$ and $V_{w_i...w_{i-n+1}}$. When key k is located in $A_{w_i...w_{i-n+1}}[j]$, the value stored at $V_{w_i...w_{i-n+1}}[j]$ includes the address of arrays $A_{w_i...w_{i-n+1}k}$ and $V_{w_i...w_{i-n+1}k}$. To find the values associated with an n-gram $w_i...w_{i-n+1}$, we first search the root array A for j_1 such that $A[j_1] = w_i$. We retrieve the address of A_{w_i} from $V[j_1]$, and we then search for j_2 such that $A_{w_i}[j_2] = w_{i-1}$. We continue to iterate this process until we find the value associated with the longest suffix of our n-gram stored in the trie. We therefore iteratively retrieve the parameters needed to compute Equation 2, returning to the root exactly once if backoff parameters are required.

3.1.1 K-ary search and B-trees

On a GPU, the trie search algorithm described above is not efficient because it makes extensive use of binary search, an inherently serial algorithm. However, there is a natural extension of binary search that is well-suited to GPU: K-ary search (Hwu, 2011). Rather than divide an array in two as in binary search, K-ary search divides it into K equal parts and performs $K - 1$ comparisons simultaneously (Figure 5).

To accommodate large language models, the complete trie must reside in global memory, and in this setting, K-ary search on an array is inefficient, since the parallel threads will access non-consecutive memory locations. To avoid this, we require a data structure that places the K elements compared by K-ary search in consecutive memory locations so that they can be copied from global to shared memory with a coalesced read. This data structure is a B-tree (Bayer and McCreight, 1970), which is widely used in databases, filesystems and information retrieval.

Informally, a B-tree generalizes binary trees in exactly the same way that K-ary search generalizes binary search (Figure 6). More formally, a B-tree is a recursive data structure that replaces arrays A and V at each node of the trie. A B-tree node of size K consists of three arrays: a 1-indexed array B of $K - 1$ keys; a 1-indexed array V of $K - 1$ associated values so that $V[j]$ is the value associated with key $B[j]$; and, if the node is not a leaf, a 0-indexed array C of K addresses to child B-trees. The keys in B are sorted, and the subtree at address pointed to by child $C[j]$ represents only key-value pairs for keys between $B[j]$ and $B[j + 1]$ when $1 \leq j < K$, keys less than $B[1]$ when $j = 0$, or keys greater than $B[K]$ when $j = K$.

To find a key k in a B-tree, we start at the root node, and we seek j such that $B[j] \leq k < B[j + 1]$. If $B[j] = k$ we return $V[j]$, otherwise if the node is not a leaf node we return the result of recursively querying the B-tree node at the address $C[j]$ ($C[0]$ if $k < B[1]$ or $C[K]$ if $k > B[K]$). If the key is not found in array B of a leaf, the query fails.

Our complete data structure is a trie in which each node except the root is a B-tree (Figure 7). Since the root contains all possible keys, its keys are simply represented by an array A, which can be indexed in constant time without any search.

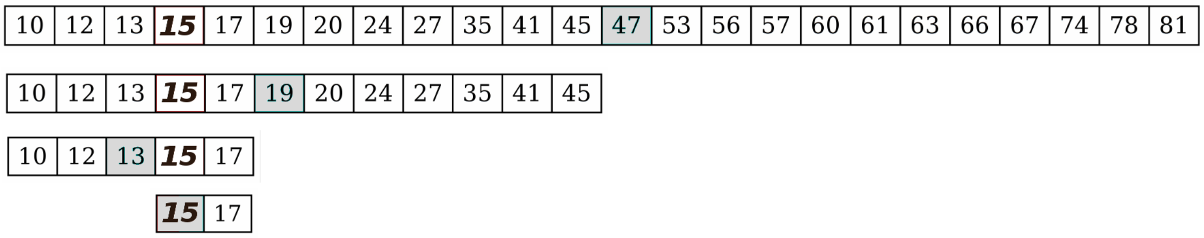

Figure 4: Execution of a binary search for key 15. Each row represents a time step and highlights the element compared to the key. Finding key 15 requires four time steps and four comparisons.

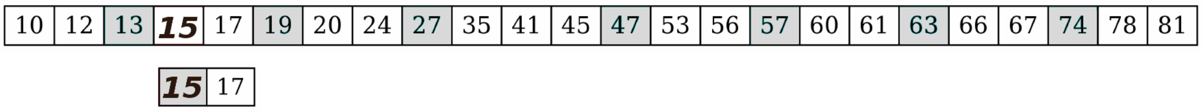

Figure 5: Execution of K-ary search with the same input as Figure 4, for $K = 8$. The first time step executes seven comparisons in parallel, and the query is recovered in two time steps.

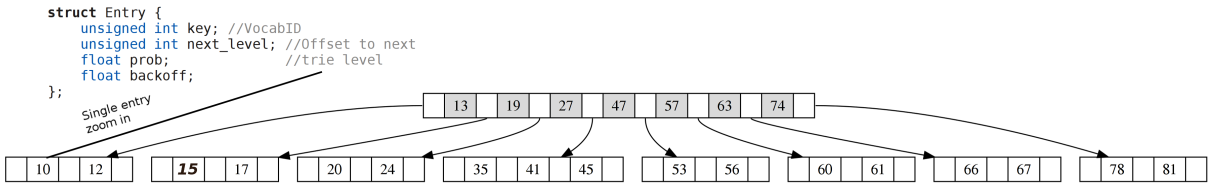
```
struct Entry {
    unsigned int key; //VocabID
    unsigned int next_level; //Offset to next
    float prob;              //trie level
    float backoff;
};
```

Figure 6: In a B-tree, the elements compared in K-ary search are consecutive in memory. We also show the layout of an individual entry.

4 Memory layout and implementation

Each trie node represents a unique n-gram $w_i...w_{i-n+1}$, and if a B-tree node within the trie node contains key w_{i-n}, then it must also contain the associated values $\hat{P}(w_i...w_{i-n})$, $\beta(w_i...w_{i-n})$, and the address of the trie node representing $w_i...w_{i-n}$ (Figure 6, Figure 3). The entire language model is laid out in memory as a single byte array in which trie nodes are visited in breadth-first order and the B-tree representation of each node is also visited in breadth-first order (Figure 7).

Since our device has a 64-bit architecture, pointers can address 18.1 exabytes of memory, far more than available. To save space, our data structure does not store global addresses; it instead stores the difference in addresses between the parent node and each child. Since the array is aligned to four bytes, these relative addresses are divided by four in the representation, and multiplied by four at runtime to obtain the true offset. This enables us to encode relative addresses of 16GB, still larger than the actual device memory. We estimate that relative addresses of this size allow us to store a model containing around one billion n-grams.[2] Unlike CPU language model implementations such as those of Heafield (2011) and Watanabe et al. (2009), we do not employ further compression techniques such as variable-byte encoding or LOUDS, because their runtime decompression algorithms require branching code, which our implementation must avoid.

We optimize the node representation for coalesced reads by storing the keys of each B-tree consecutively in memory, followed by the corresponding values, also stored consecutively (Figure 6). When the data structure is traversed, only key arrays are iteratively copied to shared memory until a value array is needed. This design minimizes the number of reads from global memory.

4.1 Construction

The canonical B-tree construction algorithm (Cormen et al., 2009) produces nodes that are not fully saturated, which is desirable for B-trees that

[2]We estimate this by observing that a model containing 423M n-grams takes 3.8Gb of memory, and assuming an approximately linear scaling, though there is some variance depending on the distribution of the n-grams.

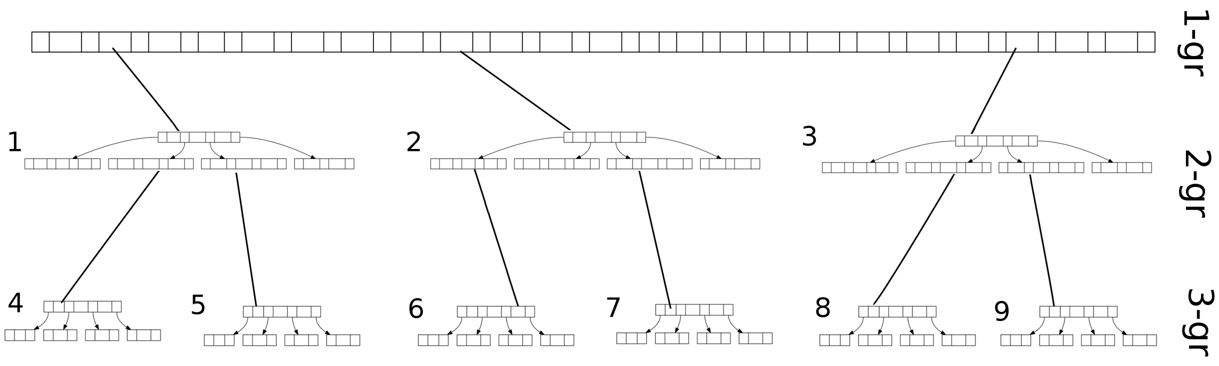

Figure 7: Illustration of the complete data structure, showing a root trie node as an array representing unigrams, and nine B-trees, each representing a single trie node. The trie nodes are numbered according to the order in which they are laid out in memory.

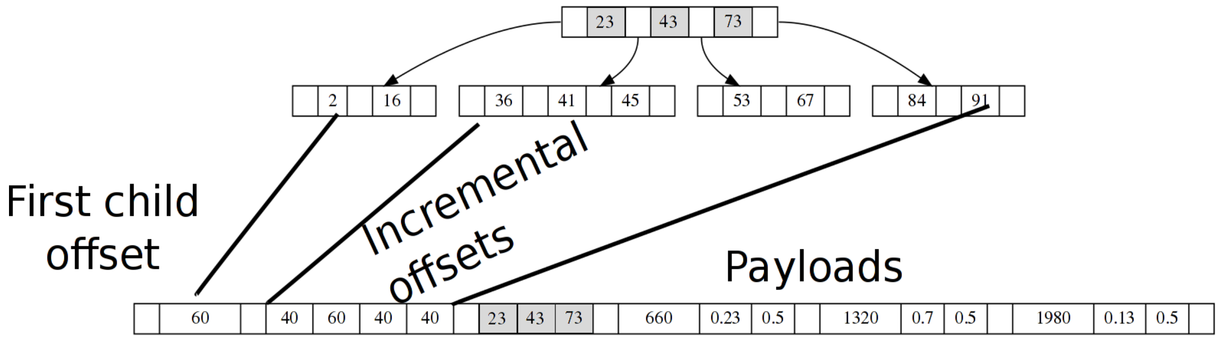

Figure 8: Layout of a single B-tree node for $K = 4$. Relative addresses of the four child B-tree nodes (array C) are followed by three keys (array B), and three values (array V), each consisting of an n-gram probability, backoff, and address of the child trie node.

support insertion. However, our B-trees are immutable, and unsaturated nodes of unpredictable size lead to underutilization of threads, warp divergence, and deeper trees that require more iterations to query. So, we use a construction algorithm inspired by Cesarini and Soda (1983) and Rosenberg and Snyder (1981). It is implemented on CPU, and the resulting array is copied to GPU memory to perform queries.

Since the entire set of keys and values is known in advance for each n-gram, our construction algorithm receives them in sorted order as the array A described in Section 3.1. The procedure then splits this array into K consecutive subarrays of equal size, leaving $K-1$ individual keys between each subarray.[3] These $K-1$ keys become the keys of the root B-tree. The procedure is then applied recursively to each subarray. When applied to an array whose size is less than K, the algorithm returns a leaf node. When applied to an array whose size is greater than or equal to K but less than $2K$, it splits the array into a node containing the first $K-1$ keys, and a single leaf node containing the remaining keys, which becomes a child of the first.

4.2 Batch queries

To fully saturate our GPU we execute many queries simultaneously. A grid receives the complete set of N-gram queries and each block processes a single query by performing a sequence of K-ary searches on B-tree nodes.

5 Experiments

We compared our open-source GPU language model **gLM** with the CPU language model KenLM (Heafield, 2011).[4][5] KenLM can use two quite different language model data structures: a fast probing hash table, and a more compact but slower trie, which inspired our own language model design. Except where noted, our B-tree

[3]Since the size of the array may not be exactly divisible by K, some subarrays may differ in length by one.

[4]https://github.com/XapaJIaMnu/gLM

[5]https://github.com/kpu/kenlm/commit/9495443

node size $K = 31$, and we measure throughput in terms of query speed, which does not include the cost of initializing or copying data structures, or the cost of moving data to or from the GPU.

We performed our GPU experiments on an Nvidia Geforce GTX, a state-of-the-art GPU, released in the first quarter of 2015 and costing 1000 USD. Our CPU experiments were performed on two different devices: one for single-threaded tests and one for multi-threaded tests. For the single-threaded CPU tests, we used an Intel Quad Core i7 4720HQ CPU released in the first quarter of 2015, costing 280 USD, and achieving 85% of the speed of a state-of-the-art consumer-grade CPU when single-threaded. For the multi-threaded CPU tests we used two Intel Xeon E5-2680 CPUs, offering a combined 16 cores and 32 threads, costing at the time of their release 3,500 USD together. Together, their performance specifications are similar to the recently released Intel Xeon E5-2698 v3 (16 cores, 32 threads, costing 3,500USD). The different CPU configurations are favorable to the CPU implementation in their tested condition: the consumer-grade CPU has higher clock speeds in single-threaded mode than the professional-grade CPU; while the professional-grade CPUs provide many more cores (though at lower clock speeds) when fully saturated. Except where noted, CPU throughput is reported for the single-threaded condition.

Except where noted, our language model is the Moses 3.0 release English 5-gram language model, containing 88 million n-grams.[6] Our benchmark task computes perplexity on data extracted from the Common Crawl dataset used for the 2013 Workshop on Machine Translation, which contains 74 million words across 3.2 million sentences.[7] Both gLM and KenLM produce identical perplexities, so we are certain that our implementation is correct. Except where noted, the faster KenLM Probing backend is used. The perplexity task has been used as a basic test of other language model implementations (Osborne et al., 2014; Heafield et al., 2015).

5.1 Query speed

When compared to single-threaded KenLM, our results (Table 3) show that gLM is just over six times faster than the fast probing hash table, and nearly fifteen times faster than the trie data structure, which is quite similar to our own, though slightly smaller due to the use of compression. The raw speed of the GPU is apparent, since we were able to obtain our results with a relatively short engineering effort when compared to that of KenLM, which has been optimized over several years.

LM (threads)	Throughput	Size (GB)
KenLM probing (1)	10.3M	1.8
KenLM probing (16)	49.8M	1.8
KenLM probing (32)	**120.4M**	1.8
KenLM trie (1)	4.5M	**0.8**
gLM	65.5M	1.2

Table 3: Comparison of gLM and KenLM on throughput (N-gram queries per second) and data structure size.

When we fully saturate our professional-grade CPU, using all sixteen cores and sixteen hyperthreads, KenLM is about twice as fast as gLM. However, our CPU costs nearly four times as much as our GPU, so economically, this comparison favors the GPU.

On first glance, the scaling from one to sixteen threads is surprisingly sublinear. This is not due to vastly different computational power of the individual cores, which are actually very similar. It is instead due to scheduling, cache contention, and—most importantly—the fact that our CPUs implement *dynamic overclocking*: the base clock rate of 2.7 GHz at full saturation increases to 3.5 GHz when the professional CPU is underutilized, as when single-threaded; the rates for the consumer-grade CPU similarly increase from 2.6 to 3.6 GHz.[8]

5.2 Effect of B-tree node size

What is the optimal K for our B-tree node size? We hypothesized that the optimal size would be one that approaches the size of a coalesced memory read, which should allow us to maximize parallelism while minimizing global memory accesses and B-tree depth. Since the size of a coalesced read is 128 bytes and keys are four bytes, we hypothesized that the optimal node size would be around $K = 32$, which is also the size of a warp. We tested this by running experiments

[6]http://www.statmt.org/moses/RELEASE-3.0/models/fr-en/lm/europarl.lm.1

[7]http://www.statmt.org/wmt13/training-parallel-commoncrawl.tgz

[8]Intel calls this Intel Turbo Boost.

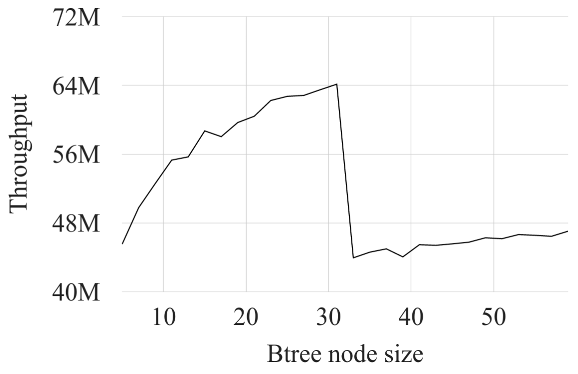

Figure 9: Effect of BTree node size on throughput (ngram queries per second)

that varied K from 5 to 59, and the results (Figure 9) confirmed our hypothesis. As the node size increases, throughput increases until we reach a node size of 33, where it steeply drops. This result highlights the importance of designing data structures that minimize global memory access and maximize parallelism.

We were curious about what effect this node size had on the depth of the B-trees representing each trie node. Measuring this, we discovered that for bigrams, 88% of the trie nodes have a depth of one—we call these *B-stumps*, and they can be exhaustively searched in a single parallel operation. For trigrams, 97% of trie nodes are B-stumps, and for higher order n-grams the percentage exceeds 99%.

5.3 Saturating the GPU

A limitation of our approach is that it is only effective in high-throughput situations that continually saturate the GPU. In situations where a language model is queried only intermittently or only in short bursts, a GPU implementation may not be useful. We wanted to understand the point at which this saturation occurs, so we ran experiments varying the batch size sent to our language model, comparing its behavior with that of KenLM. To understand situations in which the GPU hosts the language model for query by an external GPU, we measure query speed with and without the cost of copying queries to the device.

Our results (Figure 10) suggest that the device is nearly saturated once the batch size reaches a thousand queries, and fully saturated by ten thousand queries. Throughput remains steady as batch size increases beyond this point. Even with the cost of copying batch queries to GPU memory, throughput is more than three times higher than that of single threaded KenLM. We have not included results of multi-threaded KenLM scaling on Figure 10 but they are similar to the single-threaded case: throughput (as shown on Table 3) plateaus at around one hundred sentences per thread.

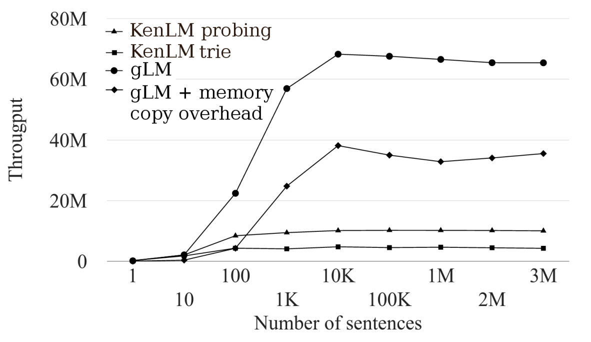

Figure 10: Throughput (N-gram queries per second) vs. batch size for gLM, KenLM probing, and KenLM trie.

	Regular LM	Big LM
KenLM	10.2M	8.2M
KenLM Trie	4.5M	3.0M
gLM	**65.5M**	**55M**

Table 4: Throughput comparison (ngram queries per second) between **gLM** and **KenLM** with a 5 times larger model and a regular language model.

5.4 Effect of model size

To understand the effect of model size on query speed, we built a language model with 423 million n-grams, five times larger than our basic model. The results (Table 4) show an 18% slowdown for gLM and 20% slowdown for KenLM, showing that model size affects both implementations similarly.

5.5 Effect of N-gram order on performance

All experiments so far use an N-gram order of five. We hypothesized that lowering the ngram order of the model would lead to faster query time (Table 5). We observe that N-gram order affects throughput of the GPU language model much more than the CPU one. This is likely due to effects of backoff queries, which are more optimized in KenLM. At higher orders, more backoff queries occur, which reduces throughput for gLM.

	5-gram	4-gram	3-gram
KenLM	10.2M	9.8M	11.5M
KenLM Trie	4.5M	4.5M	5.2M
gLM	**65.5M**	**71.9M**	**93.7M**

Table 5: Throughput comparison (ngram queries per second) achieved using lower order ngram models.

5.6 Effect of templated code

Our implementation initially relied on hard-coded values for parameters such as B-tree node size and N-gram order, which we later replaced with parameters. Surprisingly, we observed that this led to a reduction in throughput from 65.6 million queries per second to 59.0 million, which we traced back to the use of dynamically allocated shared memory, as well as compiler optimizations that only apply to compile-time constants. To remove this effect, we heavily templated our code, using as many compile-time constants as possible, which improves throughput but enables us to change parameters through recompilation.

5.7 Bottlenecks: computation or memory?

On CPU, language models are typically memory-bound: most cycles are spent in random memory accesses, with little computation between accesses. To see if this is true in gLM we experimented with two variants of the benchmark in Figure 3: one in which the GPU core was underclocked, and one in which the memory was underclocked. This effectively simulates two variations in our hardware: A GPU with slower cores but identical memory, and one with slower memory, but identical processing speed. We found that throughput decreases by about 10% when underclocking the cores by 10%. On the other hand, underclocking memory by 25% reduced throughput by 1%. We therefore conclude that gLM is computation-bound. We expect that gLM will continue to improve on parallel devices offering higher theoretical floating point performance.

6 Conclusion

Our language model is implemented on a GPU, but its general design (and much of the actual code) is likely to be useful to other hardware that supports SIMD parallelism, such as the Xeon Phi. Because it uses batch processing, our on-chip language model could be integrated into a machine translation decoder using strategies similar to those used to integrate an on-network language model nearly a decade ago (Brants et al., 2007). An alternative method of integration would be to move the decoder itself to GPU. For phrase-based translation, this would require a translation model and dynamic programming search algorithm on GPU. Translation models have been implemented on GPU by He et al. (2015), while related search algorithms for (Chong et al., 2009; Chong et al., 2008) and parsing (Canny et al., 2013; Hall et al., 2014) have been developed for GPU. We intend to explore these possibilities in future work.

Acknowledgements

This work was conducted within the scope of the Horizon 2020 Innovation Action *Modern MT*, which has received funding from the European Unions Horizon 2020 research and innovation programme under grant agreement No 645487.

We thank Kenneth Heafield, Ulrich Germann, Rico Sennrich, Hieu Hoang, Federico Fancellu, Nathan Schneider, Naomi Saphra, Sorcha Gilroy, Clara Vania and the anonymous reviewers for productive discussion of this work and helpful comments on previous drafts of the paper. Any errors are our own.

References

R. Bayer and E. McCreight. 1970. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141.

T. Brants, A. C. Popat, P. Xu, F. J. Och, and J. Dean. 2007. Large language models in machine translation. In *In Proceedings of EMNLP-CoNLL*.

J. Canny, D. Hall, and D. Klein. 2013. A multi-teraflop constituency parser using GPUs. In *Proceedings of EMNLP*.

F. Cesarini and G. Soda. 1983. An algorithm to construct a compact B-tree in case of ordered keys. *Information Processing Letters*, 17(1):13–16.

S. F. Chen and J. Goodman. 1999. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–393.

J. Chong, Y. Yi, A. Faria, N. R. Satish, and K. Keutzer. 2008. Data-parallel large vocabulary continuous speech recognition on graphics processors. Technical Report UCB/EECS-2008-69, EECS Department, University of California, Berkeley, May.

J. Chong, E. Gonina, Y. Yi, and K. Keutzer. 2009. A fully data parallel WFST-based large vocabulary continuous speech recognition on a graphics processing unit. In *Proceedings of Interspeech*.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

M. Federico, N. Bertoldi, and M. Cettolo. 2008. IRSTLM: an open source toolkit for handling large scale language models. In *Proceedings of Interspeech*, pages 1618–1621. ISCA.

S. Green, D. Cer, and C. Manning. 2014. Phrasal: A toolkit for new directions in statistical machine translation. In *Proceedings of WMT*.

D. Hall, T. Berg-Kirkpatrick, and D. Klein. 2014. Sparser, better, faster GPU parsing. In *Proceedings of ACL*.

H. He, J. Lin, and A. Lopez. 2015. Gappy pattern matching on GPUs for on-demand extraction of hierarchical translation grammars. *TACL*, 3:87–100.

K. Heafield, R. Kshirsagar, and S. Barona. 2015. Language identification and modeling in specialized hardware. In *Proceedings of ACL-IJCNLP*, July.

K. Heafield. 2011. KenLM: faster and smaller language model queries. In *Proceedings of WMT*, pages 187–197, July.

K. Heafield. 2013. *Efficient Language Modeling Algorithms with Applications to Statistical Machine Translation*. Ph.D. thesis, Carnegie Mellon University, September.

W.-m. W. Hwu. 2011. *GPU Computing Gems Emerald Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition.

Nvidia Corporation. 2015. *Nvidia CUDA Compute Unified Device Architecture Programming Guide*. Nvidia Corporation. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

M. Osborne, A. Lall, and B. V. Durme. 2014. Exponential reservoir sampling for streaming language models. In *Proceedings of ACL*, pages 687–692.

A. L. Rosenberg and L. Snyder. 1981. Time- and space-optimality in B-trees. *ACM Trans. Database Syst.*, 6(1):174–193, Mar.

D. Talbot and M. Osborne. 2007. Smoothed Bloom filter language models: Tera-scale LMs on the cheap. In *Proceedings of EMNLP-CoNLL*, pages 468–476.

T. Watanabe, H. Tsukada, and H. Isozaki. 2009. A succinct N-gram language model. In *Proc. of ACL-IJCNLP*.

M. Yasuhara, T. Tanaka, J. ya Norimatsu, and M. Yamamoto. 2013. An efficient language model using double-array structures. In *EMNLP*, pages 222–232.