Optimizing CYK parsing on modern processors

Aaron Dunlop
Oregon Health & Science University
dunlopa@cslu.ogi.edu

Brian Roark
Oregon Health & Science University
roark@cslu.ogi.edu

1 Introduction

Recent advances in processing power have enabled CYK parsing with increasingly complex probabilistic grammars (PCFGs). Parsing speed is now governed primarily by the size of the grammar. That is, the overall complexity of exhaustive inference is $n^3|G|$, but for most interesting grammars, the grammar constant is more problematic than the cubic factor. Memory latency in modern architectures exacerbates this limitation; RAM speeds have not kept pace with increases in CPU power, so the execution time of most algorithms is now dominated by memory latency. In fact, execution time often increases super-linearly as a program's working set expands beyond the CPU's cache size, so expanding a PCFG can dramatically increase processing time (Cuppu et al., 2001; Klein and Manning, 2001).

In this paper we describe a modified CYK parsing algorithm which has several beneficial properties vs. standard implementations. Our algorithm: 1) Is cache-efficient on standard CPUs; 2) Parallelizes easily and scales smoothly to large processor counts; and 3) Adapts well to cache-less throughput-oriented processors such as graphics processors (GPUs).

2 Background

2.1 Parsing

We will focus on the inner loop in chart cell population. To populate a cell, we must intersect the potential child non-terminals at each midpoint with the grammar. We normally do this intersection in one of two ways: 1) Examine each potential child pair, looking each pair up in the grammar; 2) Loop through the entire grammar, looking for the children of each rule in the child cells (a particular grammar may favor one approach over the other). In either case, we can do the lookups through a hash table or by a binary search, but both involve unpredictable, non-sequential memory access patterns. This is cache-inefficient, causing repeated processor stalls. We are targeting a more linear access pattern to improve cache hit rate.

2.2 Parallelizing Parsing

We also want to take advantage of multi-core hardware by parallelizing CYK; we can do this in several ways:

Sentence-level: The simplest approach is to parse sentences independently on separate CPU cores. Total parse throughput (sentences/second) scales roughly linearly with the number of cores available, but the time to parse a single sentence is unchanged, so this approach does not help with real-time latency constraints.

Row-level: We generally populate one chart row fully before proceeding to the next higher row. We can process all cells of a row in parallel, but the speedup may not be as great as we would like, for several reasons: 1) High in the chart, we have fewer cells per row, and are forced to leave CPU cores idle. Those higher rows are often the most densely populated, and thus require the most processing. 2) Parallel threads compete for shared caches and memory bandwidth, so the threads populating separate cells may stall one another. 3) Some filtering approaches improve efficiency, but introduce cell-to-cell dependencies within the same row, limiting this approach (e.g. Earley (1970)).

Cell-level: Parallelization within a chart cell is more complex, but avoids the weaknesses of the first two forms of parallelization. If we can fully parallelize cell population, we can make use of all available cores regardless of the cell iteration order or the current position in the chart, and we can expect to share cache and RAM bandwidth between threads.

2.3 GPU Architecture

Graphics processors differ considerably from CPUs. CPU designers spend most of their transistors on cache, speculative execution, and other memory latency reduction techniques.

GPUs instead focus on total throughput, implementing a large number of arithmetic logic units, and devoting less silicon to latency concerns. For example, the NVIDIA GT200 GPU contains 240 individual cores.[1] GPUs generally have wide memory busses, so accesses by sequential threads to sequential addresses can be combined ('coalesced') into a single access, reducing wait times. Threading on a CPU is primarily handled in software, and context switches are expensive, so a thread waiting for memory stalls the CPU. In contrast, GPUs handle threading in hardware, so a thread waiting for memory can be context-switched out while it waits (the GT200 can handle over 30,000 simultaneous threads). This helps hide memory latency, although it pushes some the burden of managing that latency back onto the programmer. On problems scalable to large thread counts, particularly those with sequential memory access patterns, GPUs achieve tremendous speedups. Certain dense matrix operations can be accelerated by 100x or more. Multiplication of sparse matrices and vectors (SpMV) is not as straightforward, but considerable work has been done on optimizing GPU SpMV (Baskaran and Bordawekar, 2008; Bell and Garland, 2009).

3 Methods and Preliminary Results

We transform the grammar intersection within a cell into a series of vector and matrix operations, allowing efficient parallelization and caching. We represent the grammar as a matrix G in which rows represent parent non-terminals and columns pairs of child non-terminals (that is, a sparse matrix of $|V|$ rows and $|V|^2$ columns).

We populate the CYK chart in the normal bottom-to-top, left-to-right order. For each cell, we first populate a vector of possible child non-terminal pairs. For each midpoint, we take the cartesian product of non-terminal pairs from child cells, producing a sparse vector of length $|V|^2$. We then union those midpoint vectors together, choosing the most probable element.

We populate the target cell by multiplying G by the cartesian-product vector in the tropical semiring. That is, for each parent non-terminal (row), we choose the most probable child pair (column) and its probability, rather than summing across all entries in the row. We handle unary rules by a second, similar multiplication.

The cartesian product operations are independent, and thus parallelizable; the union thereof can be done in parallel as well, and each row of the matrix product can be computed separately. So we are able to scale smoothly to large thread counts. Further, the inner loops of both operations access child cell contents and G in order, allowing effective cache prediction on CPUs and memory access coalescing on GPUs.

Preliminary timings of a GPU implementation are somewhat disappointing, but our CPU implementation shows a speedup of 5.4x vs. a filtered grammar-loop implementation. We are exploring grammar transformations to improve GPU memory access patterns in the cartesian-product union.

4 Conclusion

We presented a relatively simple transformation of the standard CYK algorithm, which improves CPU cache efficiently and is parallelizable on CPU or GPU cores, and preliminary results demonstrating the promise of this approach.

References

M. M. Baskaran and R. Bordawekar. 2008. Optimizing sparse Matrix-Vector multiplication on GPUs. Technical report RC24704, NVIDIA Corporation.

N. Bell and M. Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. ACM.

V. Cuppu, B. Jacob, B. Davis, and T. Mudge. 2001. High-Performance DRAMs in workstation environments. *IEEE Transactions on Computers*, 50(11):1133–1153.

J. Earley. 1970. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102.

D. Klein and C. D. Manning. 2001. Parsing with treebank grammars: Empirical bounds, theoretical models, and the structure of the penn treebank. In *Proceedings of 39th Annual Meeting of the ACL*, pages 338–345. ACL, July.

[1] Contrast this to current Intel and AMD CPUs, which contain 4 or 6 cores and process at most 12 simultaneous threads