A multi-Teraflop Constituency Parser using GPUs

John Canny **David Hall** **Dan Klein**
UC Berkeley
Berkeley, CA, 94720
canny@berkeley.edu, dlwh,klein@cs.berkeley.edu

Abstract

Constituency parsing with rich grammars remains a computational challenge. Graphics Processing Units (GPUs) have previously been used to accelerate CKY chart evaluation, but gains over CPU parsers were modest. In this paper, we describe a collection of new techniques that enable chart evaluation at close to the GPU's practical maximum speed (a Teraflop), or around a half-trillion rule evaluations per second. Net parser performance on a 4-GPU system is over 1 thousand length-30 sentences/second (1 trillion rules/sec), and 400 general sentences/second for the Berkeley Parser Grammar. The techniques we introduce include grammar compilation, recursive symbol blocking, and cache-sharing.

1 Introduction

Constituency parsing with high accuracy (e.g. latent variable) grammars remains a computational challenge. The $O(Gs^3)$ complexity of full CKY parsing for a grammar with G rules and sentence length s, is daunting. Even with a host of pruning heuristics, the high cost of constituency parsing limits its uses. The most recent Berkeley latent variable grammar for instance, has 1.7 million rules and requires about a billion rule evaluations for inside scoring of a single length-30 sentence. GPUs have previously been used to accelerate CKY evaluation, but gains over CPU parsers were modest. e.g. in Yi et al. (2011) a GPU parser is described for the Berkeley Parser grammar which achieves 5 sentences per second on the first 1000 sentences of Penn Treebank section 22 Marcus et al. (1993), which is comparable with the best CPU parsers Petrov and Klein (2007). Our parser achieves 120 sentences/second per GPU for this sentence set, and over 250 sentences/sec on length ≤ 30 sentences. These results use a Berkeley Grammar approximately twice as big as Yi et al. (2011), an apparent 50x improvement. On a 4-GPU system, we achieve 1000 sentences/sec for length ≤ 30 sentences. This is 2 orders of magnitude faster than CPU implementations that rely heavily on pruning, and 5 orders of magnitude faster than full CKY evaluation on a CPU.

Key to these results is a collection of new techniques that enable GPU parsing at close to the GPU's practical maximum speed (a Teraflop for recent GPUs), or around a half-trillion rule evaluations per second. The techniques are:

1. Grammar compilation, which allows register-to-register code for application of grammar rules. This gives an order of magnitude (10x) speedup over alternative approaches that use shared memory.
2. Symbol/rule blocking of the grammar to respect register, constant and instruction cache limits. This is precondition for 1 above, and the details of the partitioning have a big (> 4x) effect on performance.
3. Sub-block partitioning to distribute rules across the stream processors of the GPU and allow L2 cache acceleration. A factor of 2 improvement.

The code generated by our parser comes close to the theoretical limits of the GPU. 80% of grammar rules

Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing, pages 1898–1907,
Seattle, Washington, USA, 18-21 October 2013.

are evaluated using a single-cycle register-to-register instruction.

2 GPU Design Principles

In this paper, we focus on the architecture of recent NVIDIA® GPUs, though many of the principles we describe here can be applied to other GPUs (e.g. those made by AMD®.) The current NVIDIA® Kepler™ series GPU contains between 2 and 16 "stream processors" or SMX's which share an L2 cache interfacing to the GPUs main memory Anonymous (2013). The SMXs in turn comprise 192 cores which share a memory which is partitioned into "shared memory" and L1 cache. Shared memory supports relatively fast communication between threads in an SMX. Communication between SMXs has to pass through slower main memory.

The execution of instructions within SMXs is virtualized and pipelined - i.e. it is not a simple task to count processors, although there are nominally 192 in the Kepler™ series. Register storage is not attached to cores, instead registers are associated in blocks of 63 or 255 (depending on Kepler™ sub-architecture) with running threads. Because of this, it is usually easier for the programmer to think of the SMXes as 1024 thread processors. These 1024 threads are grouped into 32 groups of 32 threads called *warps*. Each warp of threads shares a program counter and executes code in lock-step. However, execution is not SIMD - all threads do not execute all instructions. When the warp encounters a branching instruction, all branches that are satisfied by some thread will be executed in sequence. Each thread only executes the instructions for its own branch, and idles for the others. NVIDIA® calls this model SIMT (Single Instruction, Multiple Threads). Execution of diverging branches by a warp is called *warp divergence*. While it simplifies programming, warp divergence understandably hurts performance and our first goal is to avoid it.

GPUs are generally optimized for single-precision floating point arithmetic in support of rendering and simulation. Table 1 shows instruction throughput (number of instructions that are executed per cycle on each SMX). The Kepler™ series has two architectural sub-generations (3.0 and 3.5) with significant differences in double-precision support. Data from Anonymous (2012) and NVIDIA (2012).

Instruction type	Architecture	
	3.0	3.5
Shared memory word access	32	32
FP arithmetic +,-,*,FMA	192	192
DP arithmetic +,-,*,FMA	8	64
Integer +,-	160	160
Integer *,FMA	32	32
float sin, exp, log,...	32	32

Table 1: Instructions per cycle per SMX in generation 3.0 and 3.5 Kepler™ devices

In the table, FP is floating point, DP is double precision, and FMA is a single-cycle floating-piont fused multiply-add used in most matrix and vector operations ($A \leftarrow A + B * C$). Note next that floating point (single precision) operations are extremely fast and there is an FPU for each of the 192 processors. Double precision floating point is 3x slower on high-end 3.5 GPUS, and much slower (24x) on the commodity 3.0 machines. While integer addition is fast, integer multiply is much slower. Perhaps most surprising is the speed of single-precision transcendental function evaluation, log, exp, sin, cos, tan, etc., which are as fast as shared memory accesses or integer multiplication, and which amount to a quarter-trillion transcendental evaluations per second on a GTX-680/K10.

PCFG grammar evaluation nominally requires two multiplications and an addition per rule (section 4) which can be written:

$$S_{ij,m} = \sum_{k=1\ldots j;\ n,p\in Q} S_{ik,n} S_{(k+1)j,p} c_{mnp} \qquad (1)$$

i.e. the CKY node scores are sums of products of pairs of scores and a weight. This suggests that at least in principle, it's possible to achieve a trillion rule evaluations per second on a 680 or K10 device, using a * and an FMA operation for each rule. That assumes we are doing register-to-register operations however. If we worked through shared memory (first line of the table), we would be limited to about 80 billion evaluations/sec, 20 times slower. The analysis underscores that high performance for parsing on a GPU is really a challenge of data movement. We next review the different storage types and their

bandwidths, since prudent use of, and movement between storage types is the key to performance.

2.1 Memory Types and Speeds

There are six types of storage on the GPU which matter for us. For each type, we give the capacity and aggregate bandwidth on a typical device (a GTX-680 or K10 running at 1GHz).

Register files These are virtualized and associated with threads rather than processors. 256kB per SMX. Each thread in architecture 3.0 devices can access 63 32-bit registers, or 255 registers for 3.5 devices. Aggregate bandwidth 40 TB/s.

Shared memory/L1 cache is shared by all threads in an SMX. 64kB per SMX partitioned into shared memory and cache functions. Aggregate bandwidth about 1 TB/s.

Constant memory Each SMX has a 48kB read/only cache separate from the L1 cache. It can store grammar constants and has much higher bandwidth than shared memory. Broadast bandwidth 13 TB/s.

Instruction cache is 8 KB per SMX. Aggregate bandwidth 13 TB/s.

L2 cache is 0.5-1.5 MB, shared between all SMXs. Aggregate bandwidth 500 GB/s.

Global memory is 2-6GB typically, and is shared by all SMXs. GPUs use a particularly fast form of SDRAM (compared to CPUs) but it is still much slower than the other memory types above. Aggregate bandwidth about 160 GB/s.

There is one more very important principle: Coalesced main memory access. From the above it can be seen that main memory access is much slower than other memories and can easily bottleneck the calculations. The figure above (160 GB/s) for main memory access assumes such access is *coalesced*. Each thread in the GPU has a thread and a block number which determines where it runs on the hardware. Consecutively-numbered threads should access consecutive main memory locations for fast memory access.

These parameters suggest a set of design principles for peak performance:

1. Maximize use of registers for symbol scores, and minimize use of shared memory (in fact we will not use it at all).
2. Maximize use of constant memory for rule weights, and minimize use of shared memory.
3. Partition the rule set into blocks that respect the limits on number of registers, constant memory (needed for grammar rules probabilities) and instruction cache limits.
4. Minimize main memory access and use L2 cache to speed it up.

Lets look in more detail at how to achieve this.

3 Anatomy of an Efficient GPU Parser

High performance on the GPU requires us to minimize code divergence. This suggests that we do not use a lexicalized grammar or a grammar that is sensitive to the position of a span within the sentence. These kinds of grammars—while highly accurate—have irregular memory access patterns that conflict with SIMD execution. Instead, an unlexicalized approach like that of Johnson (2011) or Klein and Manning (2003), or a latent variable approach like that of Matsuzaki et al. (2005) or Petrov et al. (2006) are more appropriate. We opt for the latter kind: latent variable grammars are fairly small, and their accuracies rival lexicalized approaches.

Our GPU-ized inside algorithm maintains two data structures: parse charts that store scores for each labeled span, as usual, and a "workspace" that is used to actually perform the updates of the inside algorithm. Schematically, this memory layout is represented in Figure 1. A queue is maintained CPU-side that enqueues work items of the form (s, p, l, r), where s is a sentence, and p, l, and r specify the index in the parse chart for parent, left child, and right child, respectively. The outer loop proceeds in increasing span length (or height of parent node scores to be computed). Next the algorithm iterates over the available sentences. Then it iterates over the parent nodes at the current length in that sentences, and finally over all split points for the current parent node. In each case, work items are sent to the queue with that span for all possible split points.

When the queue is full—or when there are no more work items of that length—the queue is flushed

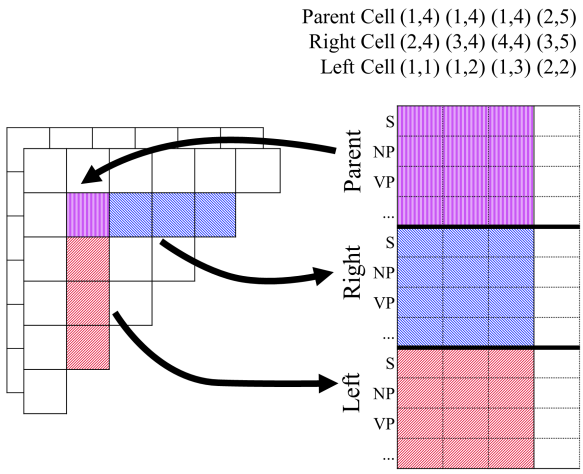

Figure 1: The architecture of the system. Parse charts are stored in triangular arrays laid out consecutively in memory. Scores for left and right children are transposed and copied into the "workspace" array, and the inside updates are calculated for the parent. Scores are then pushed back to the appropriate cell in the parse charts, maxing them with scores that are already there. Transposition ensures that reads and writes are coalesced.

to the GPU, which executes three steps. First, the scores for each left and right child are copied into the corresponding column in the workspace. Then inside updates are applied in parallel for all cells to get parent scores. Then parents are entered back to their appropriate cells in the parse charts. This is typically a many-to one atomic reduction (either a sum for probability scores, or a max for max-sum log probability scores). This process repeats until all span lengths have been processed.

3.1 The Inside Updates

The high-level goal of our parser is to use SIMD parallelism to evaluate the same rule across many spans (1024 threads are currently used to process 8192 spans in each kernel). This approaches allows us to satisfy the GPU performance desiderata from the previous section. As discussed in section 5 each GPU kernel actually processes a small subset of the symbols and rules for the grammar, and kernels are executed in sequence until the entire grammar has been processed. Each thread iterates over the rules in the same order, reading in symbols from the left child and right child arrays in main memory as necessary.

The two-dimensional work arrays must be stored in "symbol-major" order for this to work. That is, the parent VP for one work item is stored next to the parent VP for the next work item, while the VP symbol for the first work item is stored on the next "row" of the work array. The reason the workspace cells are stored in "symbol-major" order is to maximize coalesced access: each thread in the SMX accesses the same symbol for a different work item in parallel, and those work items are in consecutive memory locations.

3.2 The Copy-transpose Operations

Unlike the workspace arrays, the arrays for the parse charts are stored in "span-major" order, transposed from how they are stored in the workspace arrays. That is, for a given span, the NP symbol is next to the same span's VP symbol (for example). This order accelerates both symbol loading and Viterbi search later on. It requires a transpose-copy instead of "non-transposed" copy to move from chart to workspace arrays and back again, but note that a non-transposed copy (or direct access to the chart by the GPU compute kernel) would probably be slower. The reason is that any linear ordering of cells in the triangle table will produce short segments (less than 32 words and often less than 16) of consecutive memory locations. This will lead to many non-coalesced memory accesses. By contrast the span-major representation always uses vectors whose lengths equals the number of symbols (500-1000), and these can be accessed almost entirely with coalesced operations. The copy-transpose operations are quite efficient (the transpose itself is much faster than the I/O), and come close to the 160 GB/s GPU main memory limit.

The reverse copy-transpose (from parent workspace cells to chart) is typically many-to-one, since parent scores derive from multiple splits. They are implemented using atomic reduce operations (either atomic sum or atomic max) to ensure data consistency.

At the heart of our approach is the use of grammar compilation and symbol/rule blocking, described next.

4 Grammar Compilation

Each rule in a probabilistic context-free grammar can be evaluated with an update of the form:

$$S_{ij,m} = \sum_{k=1...j;\ n,p \in Q} S_{ik,n} S_{(k+1)j,p} c_{mnp} \quad (2)$$

where $S_{ij,m}$ is the score for symbol m as a generator of the span of words from position i to j in the input sentence, c_{mnp} is the probability that symbol m generates the binary symbol pair n, p, and Q is the set of symbols. The scores will be stored in a CKY chart indexed by the span ij and the symbol m.

To evaluate (2) as fast as possible, we want to use register variables which are limited in number. The location indices i, j, k can be moved outside the GPU kernel to reduce the variable count. We use symbols P, L and R for respectively the score of the parent, left child and right child in the CKY chart. Then the core relation in (2) can be written as:

$$P_m = \sum_{n,p \in Q} L_n R_p c_{mnp} \quad (3)$$

In the Kepler™ architecture, register arguments are non-indexed, i.e. one cannot access register 3 as an array variable R[i] with i=3[1]. So in order to use register storage for maximum speed, we must open-code the grammar. Symbols like L_3, R_{17} are encoded as variables $L003$ and $R017$, and each rule must appear as a line of C code:

```
P043 += L003*R017*0.023123f;
P019 += L012*R123*6.21354e-7f;
 :        :       :       :
```

Open-coding the grammar likely has a host of performance advantages. It allows both compiler and hardware to "see" what arguments are coming and schedule the operations earlier than a "grammar as data" approach. Note that we show here the sum-product code for computing inner/outer symbol probabilities. For Viterbi parse extraction we replace +,* with max,+ and work on log scores.

L and R variables must be loaded from main memory, while P-values are initialized to zero and then atomically combined (sum or max) with P-values in memory. Loads are performed as late as possible, that is, a load instruction will immediately precede the first use of a symbol:

```
float R031 = right[tid+65*stride];
P001 += L001*R031*1.338202e-001f;
```

where tid is the thread ID plus an offset, and stride is the row dimension of the workspace (typically 8192), and right is the main memory array of right symbol scores. Similarly, atomic updates to P-values occur as early as possible, right after the last update to a value:

```
G020 += L041*R008*6.202160e-001f;
atomicAdd(&par[tid+6*stride],G020);
```

These load/store strategies minimize the active life of each variable and allow reuse of register variables for symbols whose lifetimes do not overlap. This will be critical to successful blocking, described in the next section.

4.1 Common subexpressions

One interesting discovery made by the compiler was that the same L,R pair is repeated in several rules. In hindsight, this is obvious because the symbols in this grammar are splits of base symbols, and so splits of the parent symbol will be involved in rules with each pair of L,R splits. The compiler recognized this by turning the L,R pair into a common subexpression in a register. i.e. the compiler converts

```
P008 += L041*R008*6.200769e-001f;
P009 += L041*R008*6.201930e-001f;
P010 += L041*R008*6.202160e-001f;
```

into

```
float LRtmp = L041*R008;
P008 += LRtmp*6.200769e-001f;
P009 += LRtmp*6.201930e-001f;
P010 += LRtmp*6.202160e-001f;
```

and inspection of the resulting assembly code shows that each rule is compiled into a single fused multiply-add of LRtmp and a value from constant memory into the P symbol register. This allows grammar evaluation to approach the theoretical Gflop limit of the GPU. For this to occur, the rules need to be sorted with matching L,R pairs consecutive. The compiler does not discover this constraint otherwise or reorder instructions to make it possible.

[1] Even if indexing were possible, it is extremely unlikely that such accesses could complete in a single cycle

4.2 Exploiting L2 cache

Finally, we have to generate code to evaluate distinct minor cube rulesets on each of the 8 SMXes concurrently in order to benefit from the L2 cache, as described in the next section. CUDA™ (NVIDIA's GPU programming Platform) does not allow direct control of SMX target, but we can achieve this by running the kernel as 8 thread blocks and then testing the block ID within the kernel and dispatching to one of 8 blocks of rules. The CUDA™ scheduler will execute each thread block on a different SMX which gives the desired distribution of code.

5 Symbol and Rule Blocking

The grammar formula (3) is very sparse. i.e. most productions are impossible and most c_{mnp} are zero. For the Berkeley grammar used here, only 0.2% of potential rules occur. Normally this would be bad news for performance because it suggests low variable re-use. However, the update relation is a *tensor* rather than a matrix product. The re-use rate is determined by the number of rules in which a particular symbol occurs, which is actually very high (more than 1000 on average).

The number of symbols is about 1100 in this grammar, and only a fraction can be stored in a thread's register set at one time (which is either 63 or 255 registers). To compute all productions we will need to break the calculation into smaller groups of variables that can fit in the available register space.

We can visualize this geometrically in figure 2. The vectors of symbols P, L and R form the leading edges of this cube. The cube will be partitioned into smaller subcubes indexed by subsets of those symbols, and containing all the rules that apply between those symbols. The partitioning is chosen so that the symbols in that subset can fit into available register storage. In addition, the partitioning is chosen to induce the same number of rules in each cube - otherwise different code paths in the kernel will run longer than others, and reduce overall performance. This figure is a simplification - in order to balance the number of rules in each subcube, the partitioning is not uniform in number of symbols as the figure suggests.

As can be seen in figure 2, cube partitioning has two levels. The original P-L-R cube is first partitioned into "major" cubes, which are then partitioned into "minor" cubes (2x2x2 in the figure). A major cube holds the symbols and rules that are executed in a single GPU kernel. The minor cubes in a major cube hold the symbols and rules that are executed in a particular SMX. For the GTX-680 or K10 with 8 SMXs, this allows different SMXs to concurrently work on different 2x2x2 subcubes in a major cube. This arrangement substantially reduces main memory bandwidth through the L2 cache (which is shared between SMXes). Each symbol in a major cube will be loaded just once from main memory, but loaded into (up to) 4 different SMXes through the L2 cache. Subcube division for caching in our experiments roughly doubled the kernel's speed.

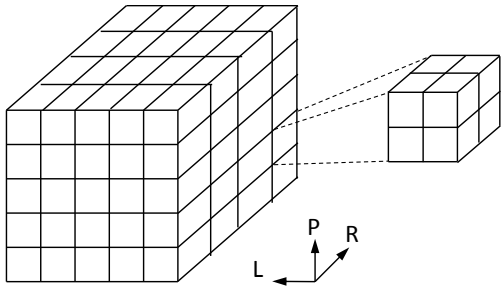

Figure 2: Partition of the cube of symbol combinations into major subcubes (left) and minor subcubes (right).

However, simple partitioning will not work. e.g. if we blocked into groups of 20 P, L, R symbols (in order to fit into 60 registers), we would need 1100/20 = 55 blocks along each edge, and a total of $55^3 \approx 160,000$ cells. Each symbol would need to be loaded $55^2 = 3025$ times, there would be almost no symbol re-use. Throughput would be limited by main memory speed to about 100 Gflops, an order of magnitude slower than our target. Instead, we use a rule partitioning scheme that creates as small a symbol footprint as possible in each cube. We use a spectral method to do this.

Before describing the spectral method we mention an optimization that drops the symbol count by 2. Symbols are either terminal or non-terminal, and in the Berkeley latent variable grammar there are roughly equal numbers of them (503 non-terminals and 631 terminals). All binary rules involve a non-terminal parent. L and R symbols may be either terminal or non-terminal, so there are 4 distinct types

of rules depending on the L, R, types. We handle each of these cases with a different kernel, which rougly halves the number of rules along each edge (it is either 503 or 631 along each edge). Furthermore, these kernels are called in different contexts, and a different number of times. e.g. XX (L, R, both non-terminal) kernels are called $O(s^3)$ times for sentences of length s because both L, R children can occur at every position in the chart. XT and TX kernels (with one terminal and one non-terminal symbol) are called only $O(s^2)$ times since one of L or R must be at the base of the chart. Finally TT kernels (both L and R are terminals) will be called $O(s)$ times. Performance is therefore dominated by the XX kernel.

5.1 Spectral Partitioning

We explored a number of partitioning schemes for both symbol and rule partitioning. In the end we settled on a spectral symbol partitioning scheme. Each symbol is a node in the graph to be partitioned. Each node is assigned a feature vector designed to match it to other nodes with similar symbols occuring in many rules. There was considerable evolution of this feature set to improve partitioning. In the end the vector for a particular P symbol is $a = (a_1, 0.1 * a_2, 0.1 * a_3)$ where a_1 is a vector whose elements are indexed by L, R pairs and whose values represent the number of rules involving both those symbols (and the parent symbol P), a_2 encodes L symbols and counts the number of rules containing that L symbol and P, and a_3 encodes the R symbols and counts rules containing that R symbol and P. This feature vector produces a high similarity between P symbols that exactly share many L,R pairs and lower similarity for shared L and R.

A spectral clustering/partitioning algorithm approximately minimizes the total edge weight of graph cuts. In our case, the total weight of a cut is to first order the product of the number of L,R pairs that occur on each side of the cut, and to second order the count of individual L and R pairs that span the cut. Let S and T be the counts for a particular LR pair or feature, then we are trying to minimize the product S*T while keeping the sum S+T, which is the total occurences of the feature on both sides of the partition, constant. Such a product is minimized when one of S or T is zero. Since many symbols are involved, this typically does not happen to an individual symbol, but this heuristic is successful at making the individual symbol or LR pair distributions across the cuts as unbalanced as possible. i.e. one side of the cut has very few instances of a given symbol. The number of instances of a symbol is an upper bound on the number of subcells in which than symbol occurs, and therefore on the number of times it needs to be loaded from memory. Repeating this operation recursively to produce a 3d cell decomposition also concentrates each symbol in relatively few cells, and so tends to reduce the total register count per cell.

In a bit more detail, from the vectors a above we construct a matrix A whose columns are the feature vectors for each P symbol. Next we construct the symmetric normalized Laplacian L for the adjacency matrix $A^T A$. We then compute the eigendecomposition of L, and extract the eigenvector corresponding to the second-smallest eigenvalue. Each node in the graph is assign a real weight from the corresponding element of this eigenvector. We sort by these weights, and partition the symbols using this sort order. We tried both recursive binary partitioning, and partitioning into k intervals using the original sort order, and obtained better results with the latter.

Partitioning is applied in order P, L, R to generate the major cubes of the rule/symbol partition, and then again to generate minor cubes. This partitioning is far more efficient than a naive partitioning. The XX ruleset for our Berkeley grammar has about 343,000 rules over a 503^3 cube of non-terminal symbols. The optimal PxLxR cube decomposition (optimal in net kernel throughput) for this ruleset was 6x2x2 for major cubes, and then 2x2x2 for minor cubes. This requires 6x2x2=24 GPU kernels, each of which encodes 2x2x2=8 code blocks (recall that each of the 8 SMXs executes a different code block from the same kernel)[2]. Most importantly the reload rate (the mean number of major cells containing a given symbol, or the mean number of times a symbol needs to be reloaded from main memory) drops to about 6 (vs. 3000 for naive partitioning). This is very significant. Each symbol is used on average $343,000/501 \approx 6000$ times overall by the XX ker-

[2]This cube decomposition also respects the constant cache and instruction cache limits

nel. Dropping the reload factor to 6 means that for every 1 main memory load of a symbol, there are approximately 1000 register or L2 cache reuses. A little further calculation shows that L2 cache items are used a little more than twice, so the register reuse rate within kernel code blocks is close to 500 on average. This is what allows teraflop range speeds.

Note that while the maximum number of registers per thread in the GTX-680 or K10 is 63, the average number of variables per minor cube is over 80 for our best-performing kernel, showing a number of variables have non-overlapping lifetimes. Sorting rules lexicographically by (L,R,P) does a good job of minimizing variable lifetime overlap. However the CUDA™ compiler reorders variables anyway with slightly worse performance on average (there seems to be no way around this, other than generating assembly code directly).

6 GPU Viterbi Parse Extraction

In sequential programs for chart generation, it is possible to compute and save a pointer to the best split point and score at each node in the chart. However, here the scores at each node are computed with fine-grained parallelism. The best split point and score cannot be computed until all scores are available. Thus there is a separate Viterbi step after chart scoring.

The gap between GPU and CPU performance is large enough that CPU Viterbi search was a bottleneck, even though it requires asymptotically less work ($O(Gs^2)$ worst case, $O(Gs)$ typical) vs $O(Gs^3)$ to compute the CKY scores. Therefore we wrote a non-recusive GPU-based Viterbi search. Current GPUs support "high-level" recursion, but there is no stack in the SMX. A recursive program must create software stack space in either shared memory or main memory which serious performance impact on small function calls. Instead, we use an iterative version of Viterbi parse extraction which uses pre-allocated array storage to store its output, and such that the partially-complete output array encodes all the information the algorithm needs to proceed - i.e. the output array is also the algorithm's work queue.

Ignoring unaries for the moment, a binary parse tree for a sentence of length n has $2n-1$ nodes, including preterminals, internal nodes, and the root. We can uniquely represent a tree as an array with $2n-1$ elements. In this representation, each index corresponds to a node in prefix (depth-first) order. For example, the root is always at position 0, and the second node will correspond to the root's left child. If this second node has a left child, it will be the third node, otherwise the third node will be the second's right sibling.

We can uniquely identify the topology of the tree by storing the "width" of each node in this array, where the width is the number of words governed by that constituent. For a node at position p, its left child will always be at $p+1$, and its right child will always be at $p + 2 \cdot w_\ell$, where w_ℓ is the width of the left child. The symbol for each node can obviously be stored with the height. For unaries, we require exactly one unary rule per node, with the possibility that it is the identity rule, and so we store two nodes: one for the "pre-unary" symbol, and one for the "post-unary." (Identity unary transitions are removed in post-processing.)

Algorithm 1 Non-recursive Viterbi implementation. The algorithm proceeds left-to-right in depth-first order along the array representing the tree.

```
Input: Sentence length n, parse chart V[i,j]
Output: Array tree of size 2·n−2
  tree[0].preunary ← ROOT
  tree[0].width ← n
  i ← 0                    ▷ Current leftmost position for span
  for p ← 0 to 2·n−2 do
      j ← i + tree[p].width      ▷ Rightmost position
      postu ← BestUnary(V, tree[p].preunary, i, j)
      tree[p].postunary ← parent
      if tree[p].width = 1 then
          i ← i + 1
      else
          lc, rc, k ← BestBinary(V, parent, i, j)
          tree[p + 1].preunary ← lc
          tree[p + 1].width ← k − i
          tree[p + 2·(k−i)].width ← j - k
          tree[p + 2·(k−i)].preunary ← rc
      end if
  end for
```

Armed with this representation, we are ready to describe algorithm 1. The algorithm proceeds in

left-to-right order along the array. First, the symbol of the root is recorded. Then, for each node in the tree, we search for the best unary rule continuing it. If the node is a terminal, then no more nodes can contain the current word, and so we advance the position of the left most child. Otherwise, if the node is a non-terminal, we then find its left and right children, entering their respective symbols and widths into the array representing the tree.

The GPU implementation follows the algorithm outline above although is somewhat technical. Each parse tree is handled by a separate thread block (thread blocks are groups of threads that can communicate through shared memory, and run on a single SMX). Each thread block includes a number of threads which are used to rapidly (in partly parallel fashion) iterate through rulesets and symbol vectors for the BestBinary and BestUnary operations using coalesced memory accesses. Each thread block first loads the complete set of L and R scores for the current split being explored. Recall that these are in consecutive memory locations using the "span-major" ordering, so these loads are coalesced. Then the thread block parallel-iterates through the rules for the current parent symbol, which will be in a contiguous block of memory since the rules are sorted by parent symbol, and again are coalesced. The thread block therefore needs storage for all the L, R symbol scores and in addition working storage proportional to the number of threads (to hold the best child symbol and its score from each thread). The number of threads is chosen to maximize speed: too few will cause each thread to do more work and to run more slowly. Too many will limit the number of thread blocks (since the total threads concurrently running on an SMX is 1024) that can run concurrently. We found 128 to be optimum.

With these techniques, Viterbi search consumes approximately 1% of the parser's running time. Its throughput is around 10 Gflops, and it is 50-100x faster than a CPU reference implementation.

7 Experiments

The parser was tested in an desktop computer with one Intel E5-2650 processor, 64 GB ram, and 2 GTX-690 dual GPUs (effectively 4 GTX-680 GPUs). The high-level parser code is written in a matrix library in the Scala language, which access GPU code through JNI and using the JCUDA wrapper library for CUDA™.

XX-kernel throughput was 900 Gflops per GPU for sum-product calculation (which uses a single FMA for most rules) and 700 Gflops per GPU for max-sum calculations (which requires two instructions for most rules). Net parser throughput including max-sum CKY evaluation, Viterbi scoring traspose-copy etc was between 500 and 600 gigaflops per GPU, or about 2 teraflops total. Parsing max-length-30 sentences from the Penn Treebank test set ran at 250 sentences/sec per GPU, or 1000 sentences/sec total. General sentences were parsed at about half this rate, 120 sentences/sec per GPU, or 480 sentences/sec for the system.

8 Conclusions and Future Work

We described a new approach to GPU constituency parsing with surprisingly fast performance, close to the theoretical limits of the GPU and similar to dense matrix multiplication which achieves the devices highest practical throughput. The resulting parser parses 1000 length-30 sentences per second in a 4-GPU computer. The parser has immediate application to parsing and eventually to parser training. The two highest-priority extensions are:

Addition of pruning: coarse-to-fine score pruning should be applicable to our GPU design as it is to CPU parsers. GPU pruning will not be as granular as CPU pruning and is unlikely to yield as large speedups (4-5 orders of magnitude are common for CPU parser pruning). But on the other hand, we hardly need speedups that large, and 1-2 orders of magnitude would be very useful.

Direct generation of assembly code. Currently our code generator produces ($>$ 1.7 million lines, about same as the number of rules) C source code which must be compiled into GPU binary code. While it takes only 8 seconds to generate the source code, it takes more than an hour to compile it. The compiler evidently applies a number of optimizations that we cannot disable, and this takes time. This is an obstacle to e.g. using this framework to train a parser where there would be frequent updates to the grammar. However, since symbol variables correspond almost one-to-one with registers (modulo

lifetime overlap and reuse, which our code generator is slightly better at than the compiler), there is no reason for our code generator not to generate assembly code directly. Presumably assembly code is much faster to translate into kernel modules than C source, and hopefully this will lead to much faster kernel generation.

8.1 Code Release

The code will be released under a BSD-style open source license once its dependencies are fully integrated. Pre- and Final releases will be here `https://github.com/jcanny/BIDParse`

References

Anonymous. 2012. CUDA C PROGRAMMING GUIDE. Technical Report PG-02829-001-v5.0. Included with CUDA™ Toolkit.

Anonymous. 2013. NVIDIA's next generation CUDA compute architecture: Kepler™ GK110. Technical report. Included with CUDA™ Tootkit.

Mark Johnson. 2011. Parsing in parallel on multiple cores and gpus. In *Proceedings of the Australasian Language Technology Association Workshop 2011*, pages 29–37, Canberra, Australia, December.

Dan Klein and Christopher D. Manning. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430, Sapporo, Japan, July. Association for Computational Linguistics.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of english: The penn treebank. *COMPUTATIONAL LINGUISTICS*, 19(2):313–330.

Takuya Matsuzaki, Yusuke Miyao, and Jun'ichi Tsujii. 2005. Probabilistic CFG with latent annotations. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 75–82, Ann Arbor, Michigan, June. Association for Computational Linguistics.

NVIDIA. 2012. private communication.

Slav Petrov and Dan Klein. 2007. Improved inference for unlexicalized parsing. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, pages 404–411, Rochester, New York, April. Association for Computational Linguistics.

Slav Petrov, Leon Barrett, Romain Thibaux, and Dan Klein. 2006. Learning accurate, compact, and interpretable tree annotation. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 433–440, Sydney, Australia, July. Association for Computational Linguistics.

Youngmin Yi, Chao-Yue Lai, Slav Petrov, and Kurt Keutzer. 2011. Efficient parallel cky parsing on gpus. In *Proceedings of the 2011 Conference on Parsing Technologies*, Dublin, Ireland, October.