Optimizing Select Conditions on GPUs

Evangelia A. Sitaridi, Kenneth A. Ross
Dept. of Computer Science, Columbia University
(eva,kar)@cs.columbia.edu

ABSTRACT

Implementations of data processing operators on GPU processors have achieved significant performance improvements over their multicore CPU counterparts. To achieve maximum performance, database operator implementations must take into consideration special features of GPU architectures. A crucial difference is that the unit of execution is a group ("warp") of threads, 32 threads in our target architecture, as opposed to a single thread for CPUs. In the presence of branches, threads in a warp have to follow the same execution path; if some threads diverge then different paths are serialized. Additionally, similarly to CPUs, branches degrade the efficiency of instruction scheduling. Here, we study conjunctive selection queries where branching hurts performance. We compute the optimal execution plan for a conjunctive query, taking branch penalties into account and consider both single-kernel and multi-kernel plans. Our evaluation suggests that divergence affects performance significantly and that our techniques reduce resource underutilization and improve operator performance.

1. INTRODUCTION

Recent increases in GPU memory sizes and the availability of GPU clusters in the cloud make GPU-memory resident databases feasible. Data management operator implementations specialized for GPUs have achieved significant performance improvements.

Threads in modern GPUs are organized into groups called *warps* and the warp size on an Nvidia Tesla C2070 is 32 threads [3]. Multiple warps form larger groups of threads called *thread blocks*, which are executed independently. Threads within a thread block coordinate using fast shared memory which has limited capacity, 48KB on the Tesla C2070. Threads in different thread blocks communicate using slower but larger global memory. Thread blocks running on a GPU execute C-functions called CUDA kernels.

Threads of a warp start executing at the same program address but have their private register state and program counters so they are free to branch independently. When threads in a warp follow different execution paths *thread divergence* is said to occur, and the hardware serializes the different code paths. While the branch followed by a subset of the threads in the warp is executed, the remaining threads are idle, resulting in resource underutilization. Thread divergence significantly affects the performance of core algorithms. For example, in the GPU implementation of mergesort, divergence is identified as one of the major factors affecting performance [4]. There are two sources of divergence: a) A linear search stage, where a warp has to wait for the thread that has the longest gap to search and b) Comparison between variable-length keys. Additionally, GPUs benefit significantly from instruction level parallelism and branches might degrade the efficiency of instruction scheduling.

Here, we are interested in the performance of a table scan combined with the application of a compound selection condition. While a filtering scan might be considered a "lightweight" operator compared to a join, it is important for a number of reasons. As the initial operator in a query plan, a filtering scan will typically process a large volume of data compared with later operators that process filtered data. In data analytics scenarios where there is a limited space budget for secondary indices and/or queries touch relatively large segments of the data, scanning the data may be preferable to index-based access methods. GPUs have significantly higher RAM bandwidth compared to CPUs, so achieving high memory bandwidth for GPU scans has a larger potential payoff. In the presence of expensive select conditions, optimally ordering and grouping of conditions is necessary to maximize query performance.

The following code snippets check the conjunction of two conditions on a two column-table stored in global memory. P1 uses branching-and in which a thread retrieves the value of the second column only if the first condition holds. P2 always checks both conditions, but does not need a branch instruction.

```
P1: check=(t1[tid]==val1)  && (t2[tid]==val2);
P2: check=(t1[tid]==val1)  &  (t2[tid]==val2);
```

Consider the execution of P1 on a GPU where the selectivity of both conditions is 0.25. During the first condition check the threads access contiguous memory locations so there is only one coalesced memory transaction per warp. During the second condition check there will be on average 8 threads of a warp for which the first check is successful, so an additional memory transaction will be issued per warp, even though only a subset of the threads evaluates the sec-

DaMoN'13, June 24 2013, New York, NY, USA

ond condition.[1] In P2, all threads retrieve both values from the two columns but the memory requests can be scheduled more efficiently. We quantify this improvement by running short tests on our target machine. Depending on the selectivity of the first condition the improvement of P2 over P1 can be up to 25%.

An alternative implementation of this compound condition would apply each condition in a separate kernel, with the two kernels executed serially. The first kernel applies the first condition and outputs the records passing the check, which is the subset of records on which the second kernel will apply the second condition. In this way there will be reduced thread divergence but also increased traffic to the global memory.

Branches hurt the performance of CPU programs too. For datasets stored in main memory, branch mispredictions can significantly degrade program performance. Compound select conditions need to be optimized given the CPU, memory characteristics and condition selectivities [14]. Our approach is motivated by [14], but there are significant differences since GPU branch divergence and CPU branch misprediction are different phenomena.

Our contribution includes:

- Suggesting a cost model for estimating the global memory performance of different query execution plans, calibrated by running short tests on our target machine.
- Computing an optimal execution plan for conjunctive queries, minimizing branch penalties and thread divergence.

The rest of the paper is organized in the following way: Section 2 is an overview of related work. Section 3 illustrates the trade-offs between different query plans and analyzes our cost model. We describe our solution in Section 4 and in Section 5 we validate our cost models and present our experimental results. In Section 6 we conclude and discuss how our techniques can be used for more general queries.

2. RELATED WORK

Previous research on database processing on GPUs has demonstrated significant speed-ups [1, 7, 8]. A subset of SQLite commands has been implemented on GPUs. Thread divergence affected performance when the operators of a query plan were fused in a single CUDA kernel as opposed to different serially executed CUDA kernels.

Evaluating multiple predicates in a single-kernel is reminiscent of the kernel fusion technique suggested for data warehousing applications [18]. Kernel fusion combines operators of a query execution plan into a single CUDA kernel. A framework optimizing which query operators to fuse has been suggested [17]. Kernel fusion results in reduced data transfer through the GPU memory hierarchy, but it might increase register and shared memory pressure. Here, we focus on the increased thread divergence caused by the fusion of multiple select operators, and on how branching reduces instruction scheduling efficiency. We assume a columnar memory table layout; [7] assumes a row-store with full records packed into a 32-bit word. We also assume that data are GPU memory resident and that no transfers are required between the CPU and GPU.

We suggest a software optimization for reducing thread divergence. Other software-based solutions eliminate thread divergence by thread-data remapping [19], which has been extended to remove additional code irregularities, e.g., irregular memory references [20]. Iteration delaying and branch distribution reduce the performance impact of branch divergence on programs [10]. Loop-splitting reduces register pressure caused by thread divergence [2]. Software-based branch predication has been suggested for AMD GPUs to reduce branch penalties [16].

Various hardware extensions have been suggested for thread divergence elimination not just for GPUs but for SIMD processors in general: dynamic regrouping of threads into new warps [9], adjusting SIMD width based on branch or memory latency divergence [12], and identifying reconvergence points for threads [6].

Serialization of threads belonging to the same warp is one of the decisive factors affecting performance because it cannot be hidden in current GPU architectures by efficient warp scheduling. Another source of thread serialization within a warp are bank conflicts which can be alleviated by creating replicas of values belonging to different banks [15].

In this paper, we study how branch penalties affect the performance of compound select conditions on a GPU and our solution is adapted from algorithms minimizing branch misprediction rate for main-memory databases running on a CPU [14]. The most efficient plan optimizes the combination of branching-and (&&) and logical-and (&) operators. We suggest two polynomial algorithms: The first also optimizes combinations of branching-and and logical-and operators. The second optimizes which conditions to fuse into a single-kernel. In the presence of expensive predicates the optimal condition ordering is in ascending order of the rank metric: $\frac{selectivity-1}{cost-per-tuple}$ [11]. Optimal execution strategies for select queries involve choosing between late and early materialization [5]. A query plan compilation framework has been suggested that, in addition to minimizing CPU costs, maximizes the lifetime of the data within the registers [13].

3. PROBLEM DESCRIPTION

We assume an OLAP setting and a GPU-friendly column-store memory layout. Data is resident in GPU global memory. Queries posed are the conjunction of equality or range conditions on one or multiple columns. Compound select queries with many conditions are also common in scientific databases. We group all conditions on a single column into one so in the rest of the paper we assume that every condition is applied on a different column.[2] If there is an index on one or more attributes, we could apply the conditions on the indexed attribute(s) and use our technique for the remaining conditions.

3.1 Intra-Kernel Optimizations

In a program implementing a conjunctive select query, all threads in a warp check 32 consecutive values of the column the first condition constrains. If the condition is not satisfied for some of the threads then these threads will not evaluate

[1]Compilers sometimes use branch predication so that some instructions in branches are not skipped, to maximize efficiency of instruction scheduling. Predication adds the overhead of pipeline slots with null operations in some threads on each case of a branch. Typically there is a limit on the number of instructions to which this heuristic is applied [3].

[2]Extending the cost model to conditions like `c1[i]=c2[i]` that mention more than one column is straightforward.

the remaining conditions. However, they will remain idle until all threads in the warp finish condition evaluation. Also, similarly to CPUs, the scheduling of instructions in a branch statement is less efficient. The choice of the most efficient plan depends on the selectivity and the predicate cost. The left-to-right order of conditions is determined by the rank metric mentioned above. We show below the code snippets for all single-kernel plans of a three-condition query.

```
S3:   check = c1[i]<v1 & c2[i]<v2 & c3[i]<v3
S21:  check = (c1[i]<v1 & c2[i]<v2) && c3[i]<v3
S12:  check = c1[i]<v1 && (c2[i]<v2 & c3[i]<v3)
S111: check = c1[i]<v1 && c2[i]<v2 && c3[i]<v3
```

Figure 1 shows the execution path of the three conditions in the S111 plan.

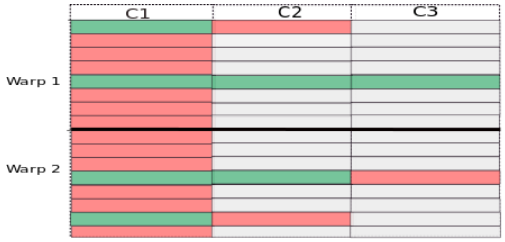

Figure 1: Execution of the S111 plan for two warps. For simplicity of presentation we assume here smaller warp size (8 threads). Red cells signify column values for which the predicate is evaluated to false, green cells are values for which the predicate is true and grey cells correspond to idle threads.

Using the implementation techniques described in [14], S3 needs no branches, and is thus called the "no-branch" plan. S21 and S12 each need one branch (depending on the compiler, which may choose predication rather than branching), and S111 needs two branches.

Figure 2 shows the performance in milliseconds of all plans for a conjunctive query computing a count aggregate with four conditions having the same selectivity, on a 128 million row table. Each plan is implemented in a separate CUDA kernel and they only vary in how they evaluate the four conditions. The no-branch plan S4 is the most efficient for selectivities ≥ 0.4 where the combined selectivity for the first three conditions is greater than $\frac{1}{32}$. In such a case, there will typically be at least one thread per warp satisfying the first three conditions. This means that one memory transaction has to be issued for each of the four columns for all plans, so the time cost of all plans remains constant after this value; the difference in performance comes from branch penalties. In the range [0.2-0.35] the optimal plan is S31. Again we note that for selectivity values ≥ 0.2 there will be one memory transaction for the first three columns. In the range 0.05-0.2 the optimal plan is S211. In CPUs that depend on branch prediction, the branch penalty is highest when the probability that the branch being taken is close to 0.5 [14], a phenomenon we do not see on GPUs.

Based on these results we conclude that conjunctive select queries need to be optimized. Our optimization algorithm uses a cost model estimating the global memory cost of different execution plans. Due to CUDA scheduling, the time cost for a two-column scan is less than twice the cost of a

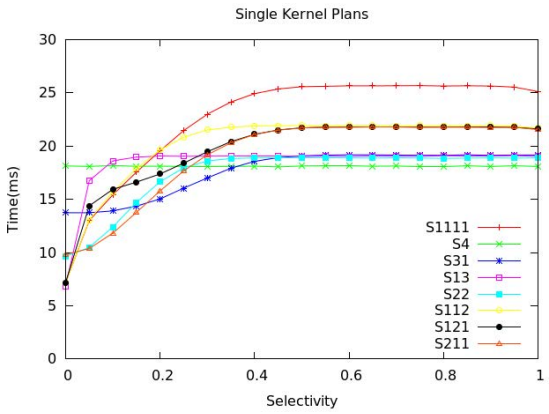

Figure 2: Time performance in ms of different plans for a query with four conditions on a table with 128 million rows.

single-column scan when memory loads from both columns are branch free. If the memory load from the second column is included in a branch and there is at least one thread in a warp fetching the second column value, the cost of the two-column scan is twice the cost of a single-column scan. We will describe now how we quantify the improvement from efficient instruction scheduling of branch-free memory loads. Say the time cost for a k-column table scan is $scan_k$. If r is the number of table rows and w is the warp size then the number of memory transactions for a sequential scan is $\frac{r}{w}$. We account for the improvement from scheduling compared to the expected performance by computing in advance (during a calibration phase) the following rate for each k:

$$r_k = \frac{scan_k}{k \times scan_1}$$

For example, the cost of the plan $(C_1$ `&` C_2 `&` $C_3)$ `&&` C_4 is:

$$r_3 \times \frac{3r}{w} + min(sel_1 * sel_2 * sel_3 * w, 1) * \frac{r}{w}$$

3.2 Multi-kernel Optimization

Depending on the selectivity and the evaluation cost of the conditions, it may be more efficient to execute the query by running separate kernels each checking a subset of the conditions. All but the last kernel write the intermediate query results to the global memory. Depending on the selectivities of the conditions, the extra write cost may be justified by the reduction in thread divergence: Every thread is active in later kernels testing later conditions. In presence of expensive predicates, fused kernels could exhibit higher resource underutilization.

Each query execution plan has one or more kernels, each checking one or more conditions. For example $[c_1][c_2c_3c_4]$ denotes a plan for a four-condition query consisting of two kernels. The first kernel evaluates the first condition and writes the positions/record-ids of the satisfying records into the global memory. The second fused kernel checks all three remaining conditions, with the best of the equivalent single-kernel plans described in the previous section. The space of alternative plans includes the different groupings of the n conditions. Because in optimal plans select conditions are ordered by the rank cost metric, we need only consider

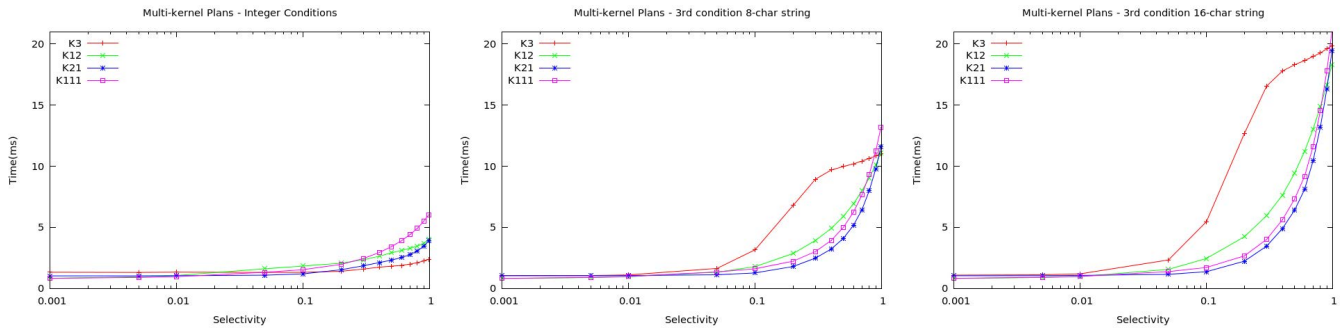

Figure 3: Performance of multi-kernel plans for a query having a single column output on a 12.8 million row table. The selectivity of the first condition is varied and the other two conditions have fixed selectivity 0.5. In the left diagram all columns are integers and in the middle and right diagram the last predicate is a substring match. The x-axis has logarithmic scale.

one left-to-right ordering of the conditions, as before. This means that plan $[c_1][c_2c_3c_4]$ can be denoted unambiguously by `K13`. We call the `K4` plan fuse-only, and the rest of the plan space uses kernel fusion on a subset of the conditions.

Figure 4 shows the execution path of the `K21` plan for three conditions.

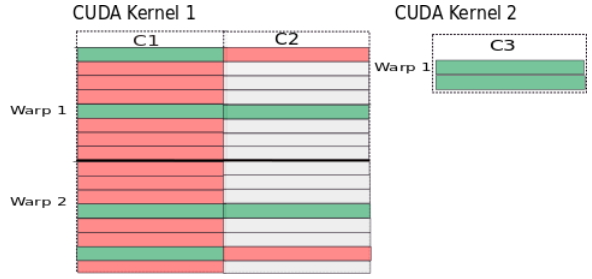

Figure 4: Execution of the K21 plan for two warps. Note that the second kernel needs fewer warps to process the remaining data.

Figure 3 shows the time cost of different plans, for a three condition query and a table with 12.8 million rows. We execute a conjunctive query and vary the selectivity of the first condition. The last two conditions have fixed selectivity 0.5 and the query output is a fourth column. In each kernel only the third condition includes a branch (`K3` is implemented as `S21`). In the first diagram all conditions are integer comparisons but in the second and the third diagram the last predicate is a substring match on a string column with 8 and 16-character strings correspondingly. For varying selectivity and predicate evaluation cost different plans are optimal, so we should optimize plan selection.

In the left diagram for higher values of selectivity (less selective conditions) fuse-only plans are the fastest because multi-kernel plans have increased write cost. In the middle and last diagram K21 is typically the most efficient because it evaluates the expensive predicate on a reduced subset of rows. K111 also evaluates the expensive predicate for fewer rows but due to the increased write cost is faster only for very selective predicates (selectivity ≤ 0.01). In fuse-only plans, even if only one of the values a thread reads satisfies the first two conditions the remaining threads have to wait for this thread to evaluate the expensive string-match operation. For 16-character strings and selectivities in [0.1-0.4] range K3 is 3-5 times slower than K21 and K111 plans although the gap narrows as selectivities increase.

The total cost for a multi-kernel plan is the sum of costs of all individual kernels which can be estimated as suggested in Section 3.1, plus the cost of writing the intermediate query results. In Appendix A we explain how shared memory is used as a buffer to achieve coalesced writes to global memory. While we have described a method that writes the record-ids of intermediate records, we also consider a variant that instead writes all of the column values that are needed in the subsequent kernels. The intermediate results are stored column-wise to be read efficiently from the subsequent kernels. While this variant may need to transfer more data, subsequent accesses will be better aligned for coalesced access.

The cost of writing the intermediate results for plan K111 and a query that selects all columns is:

$$write\ cost_{col} = \sum_{i=1}^{n-1} \frac{sel_i! \times r \times n}{w}$$

where $sel_i!$ is the combined selectivity for conditions 1 to i and $sel_0! = 1$. If the output of a kernel is a record-id list we have to add the cost of reading the list: $\frac{r}{w} \times sel_{i-1}!$, except for the first kernel which reads data sequentially. Also, all kernels but the last have to write the record-ids of the satisfying rows:

$$write\ cost_{rid} = \sum_{i=1}^{n-1} \frac{sel_i! \times r}{w}$$

If we use record-id lists there will be sequential memory access only in the first kernel. For the subsequent kernels the threads will read in a coalesced way only the record-id lists containing the positions of satisfying records but the actual values might not be stored contiguously. To account for that we have to adjust the read cost by using the random access memory bandwidth of our target machine.

4. SOLUTION DESCRIPTION

Here we describe our suggested optimization algorithm minimizing branch penalties. We suggest two different methods. The first method computes the optimal execution plan for single kernel queries with n conditions and the second

method is a multi-kernel optimization algorithm determining which conditions will be applied in a fused kernel. For a small number of conditions we could enumerate all possible plans, but the number of all possible plans grows factorially so we need more efficient alternative algorithms. Also the high performance of GPUs implies that optimization time must be small. We assume that attributes are not correlated. If this is not the case we might want to enumerate all possible condition orders and choose the one with the minimum estimated cost.

Both algorithms have the same complexity. To compute the optimal plan for a query with n conditions the time complexity is $O(n^3)$.

Model Calibration.

To factor-in the improvement by efficient CUDA scheduling of memory transactions for branch-free code we calibrate our model by measuring the scan performance for 1 to n-column scans, where n is the number of conditions in the WHERE part of the query. The calibration step is short because it only has to run once, on a relatively small number of table records. We calibrate our model using tables containing 16 million rows. On our machine, for 4-byte integer scans and 1024-thread block size the rate r_2 for a two-column scan is 0.75. For three and four columns the rate is around 0.68. This means that when measuring the cost of no-branch plan $C_1\&C_2$ the weighed number of memory transactions accounted for by our model will be $2 \times 0.75 = 1.5$ rather than 2. The constants of our model depend on the chosen thread block size/configuration since the performance of CUDA kernels also depends on the kernel configuration. If there are multiple configurations to choose from, then we should compute the described rates for each, and when optimizing a query we must use the rates for the right configuration. Here, we are showing the results for 4-byte integer columns but our model can be used for different data-types.

Single Kernel Optimization.

We present a polynomial dynamic programming algorithm for computing the optimal execution plan with n conditions. We let P denote consecutive nonempty subsets of the n conditions that have been ordered by the rank cost metric. For $n = 4$ there are 10 such subsets: 4 with a single condition, 3 with two conditions, 2 with three conditions, and one with four conditions. In general $|P| = n(n+1)/2$.

Algorithm 1 Single-Kernel Optimization Algorithm

```
Generate all no-branch plans for subsets p ∈ P, storing
their costs in A[p], with null left and right attributes
for all r ∈ P in increasing order //r right child of the plan
do
  for all l ∈ P in increasing order such that
  rank(l.last)+1=rank(r.first) do
    cost=(l && r).cost // computed from A[l] and A[r]
    if cost < A[r ∪ l].cost then
      A[r ∪ l].cost=cost
      A[r ∪ l].left=l
      A[r ∪ l].right=r
    end if
  end for
end for
```

In the initial step we generate all no-branch plans for subsets in P. Our algorithm has an outer and an inner loop corresponding to the right and left subplan. We iterate over plans in increasing order, which means if p_1 is a subset of p_2 then p_1 comes earlier in the loop than p_2. We use any generated subplan as part of the larger plans that are generated in later iterations. In the end, the optimal plan for n conditions is contained in corresponding position of the A array. Each position of this array contains the left and right subplan of the optimal plan, and the cost of this plan. We can retrieve the optimal plan using the left and right subplan.

There are $O(n^2)$ iterations of the outer loop, and $O(n)$ iterations of the inner loop because of the requirement that rank(l.last)+1=rank(r.first). Thus the overall complexity is $O(n^3)$.

Multi-kernel Optimization.

The algorithm that determines the optimal plan by splitting query execution into multiple kernels is similar to the single-kernel optimization algorithm. We use the same algorithm skeleton by varying the cost estimation when combining two plans. When combining two plans the number of kernels in the combined plan will be the sum of the number of kernels in the left and right plan and the cost is the sum of their cost plus the intermediate result writing cost to the global memory. The optimal plan determines which conditions to fuse similarly to how an optimal single kernel plan uses logical-and (&) rather than the branching-and (&&) operator. Internally, the algorithm calls Algorithm 4 to find the best fused plan for the included conditions.

5. EXPERIMENTS

GPU performance was measured using 1024 threads on an Nvidia Tesla C2070 machine with 6GB of RAM and a nominal bandwidth of 144GB/s. Optimization was done on a dual-chip Intel E5620 with nominal bandwidth 25.6GB/s using a single-thread. We assume the data to be GPU resident so we do not measure the time needed to transfer the data to the GPU memory. For reference a single column scan of a 128 million row table computing a count aggregate takes 6.5 ms corresponding to a bandwidth of around 80GB/s. Our kernels are typically memory bound. CUDA C code was compiled using the `nvcc` compiler of the CUDA toolkit 4.2 using full optimization. We used synthetic data with 128M rows and 4-byte integer columns. We ran the following handcompiled queries:

```
Q1: SELECT  C4 FROM  T
  WHERE  C1<v1 AND C2<v2 AND  C3<v3
Q2: SELECT COUNT(*) FROM  T
  WHERE  C1<v1 AND C2<v2 AND  C3<v3 AND  C4<v4
```

Figure 5 validates the model for conjunctive queries executed in a single kernel for query Q2, where all conditions have a common selectivity that is varied. We observe that the trends of different plans in the left diagram (actual times, the same as Figure 2) closely resemble those in the right diagram (optimizer estimates) proving that our cost model can be used in single kernel optimization. Different plans are optimal in different selectivity ranges. For completeness, Figure 6 shows the performance of the same plans on the CPU using 16 threads. CPU plans are 4.5–7 times slower, so even for relatively lightweight operators like compound selections,

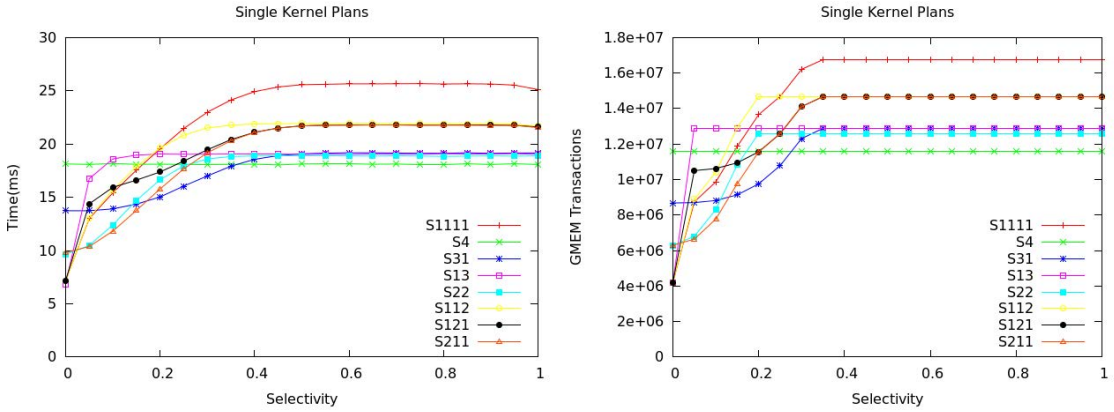

Figure 5: Actual and estimated performance for different single-kernel query plans for Q2.

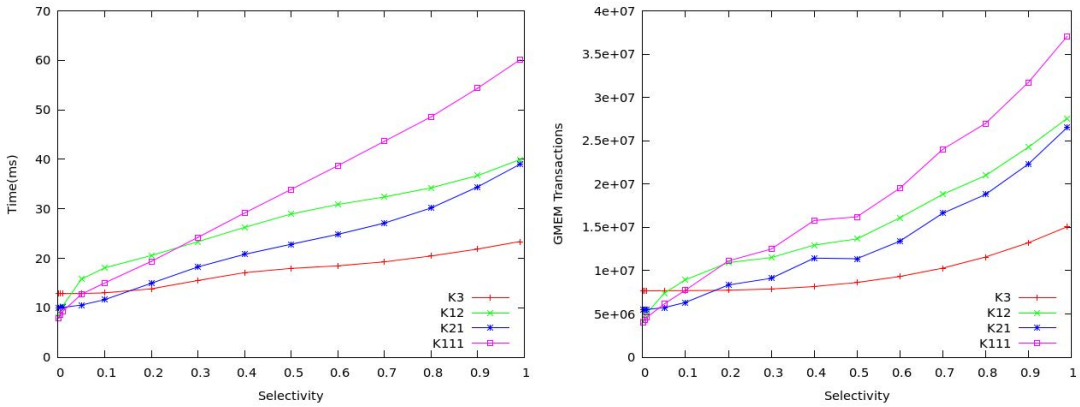

Figure 7: Actual and estimated performance of multi-kernel plans for Q1 varying the selectivity of all conditions.

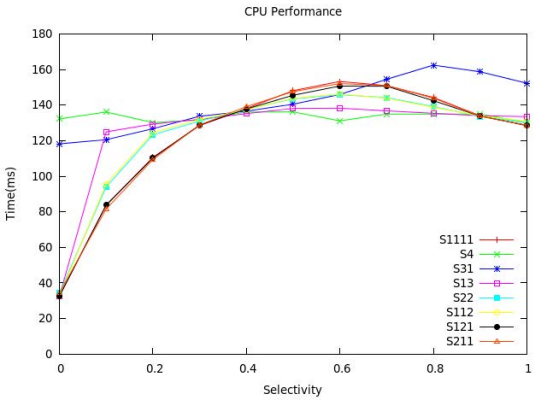

Figure 6: Time performance of different plans on the CPU for Q2.

GPUs offer better performance due to higher memory bandwidth.

In Figure 7 the left figure shows the performance of different plans for Q1 using record-ids as the intermediate data representation between kernels. The three conditions have the same selectivity and the right diagram shows the weighed number of global memory transactions. The similar trends in the two diagrams indicate that memory cost is a valid metric for multi-kernel optimization too. We also note that for such queries it is faster to execute all conditions in a fused kernel when the conditions are not selective, although as we saw in Section 3 the case is different in the presence of expensive predicates. K3 is less efficient for very selective conditions: For selectivities below 0.01 the K111 plan is the fastest, and for selectivities between 0.01 and 0.1, K21 is the most efficient.

In Figure 8 we vary the selectivity of the second condition while the first selectivity is fixed to 0.1 and the third to 0.8. The left diagram shows the performance of single-kernel plans, the middle the performance of multi-kernel plans using rid-lists for intermediate results and the right when writing full column values as intermediates. Multi-kernel plans seem to provide a performance improvement for selective conditions. For example the K21 plan is around 5% faster than the S21 plan for low selectivities. For multiple kernel plans, K21 is the optimal up to 0.2 when writing record-ids and 0.3 when writing column values, because the combined selectivity of the first two conditions is low and it is more efficient to write the reduced set of results in global memory and then apply the third condition. For most plans it is slightly more efficient to write the column values than

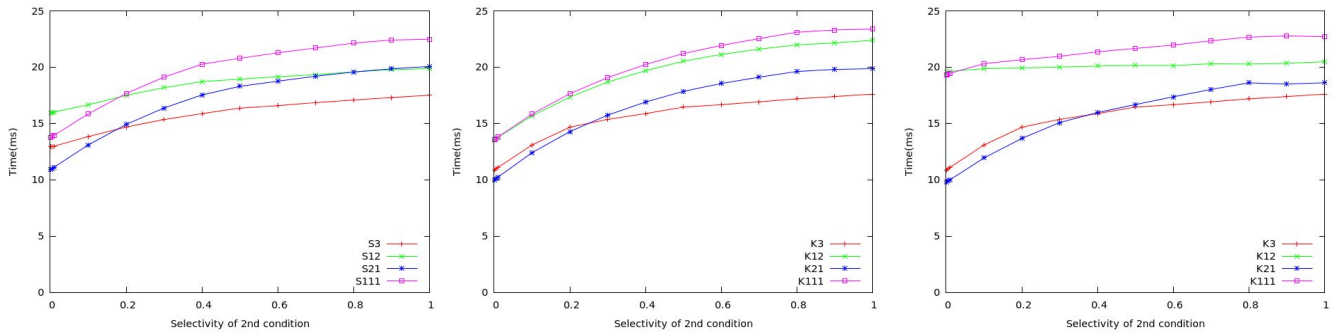

Figure 8: Time performance of different plans for Q1 where the selectivity of the second condition is varied and the first and third condition selectivities are fixed to 0.1 and 0.8. The left diagram shows the performance for single-kernel plans, the middle for multi-kernel plans using rid-lists and the last using materialization.

using rid-lists, except for selective queries on plans such as K111 and K12 where there is more intermediate data to be written. In general, the choice of intermediate result format would depend on the columns in the SELECT clause of the query and the conditions' selectivities.

Figure 9 shows the performance of the multi-kernel optimizer (running on the CPU) as a function of the number of conditions. Optimization time even for relatively many conditions is a fraction of a millisecond suggesting that the optimization cost is low compared to the query cost even in high performance environments like GPUs.

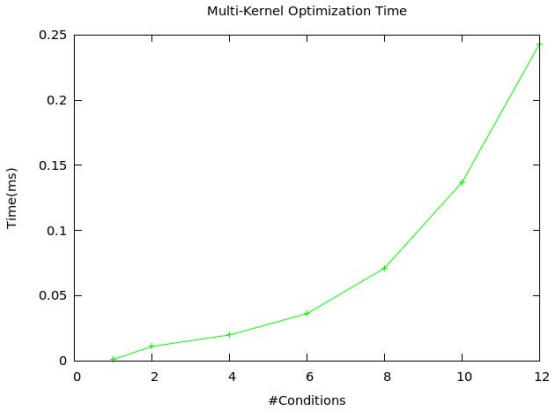

Figure 9: Time performance of the multi-kernel optimizer for an increasing number of conditions.

6. CONCLUSIONS AND FUTURE WORK

Here we studied the effect of branch penalties and thread divergence on the performance of compound select conditions for tables stored in GPU global memory. We suggest a cost model that accurately predicts the performance of query execution plans with varying branching and divergence behavior. Our results suggest that different plans are optimal for different sets of selectivities and predicate evaluation costs, and that our optimization is able to find these plans. We plan to extend our model for disjunctive select queries and to optimize the fusion of common operator combinations, including multi-way joins, in a query execution plan. Our ultimate goal is building a compiler for efficient query plans executing in a GPU.

Acknowledgements

This material is based upon work supported by NSF Grants IIS-1218222 and by an equipment gift from Nvidia Corp.

7. REFERENCES

[1] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*, 2010.

[2] S. Carrillo, J. Siegel, and X. Li. A control-structure splitting optimization for GPGPU. In *ACM conference on Computing frontiers*, 2009.

[3] N. Corporation. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation, April 2012.

[4] A. Davidson, D. Tarjan, M. Garland, and J. D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *InPar*, 2012.

[5] D. J. Dewitt, S. R. Madden, D. J. Abadi, and D. S. Myers. Materialization strategies in a column-oriented DBMS. In *ICDE*, 2007.

[6] G. Diamos, B. Ashbaugh, S. Maiyuran, A. Kerr, H. Wu, and S. Yalamanchili. SIMD re-convergence at thread frontiers. In *MICRO*, 2011.

[7] G. Diamos, H. Wu, A. Lele, J. Wang, and S. Yalamanchili. Efficient relational algebra algorithms and data structures for GPU. 2012.

[8] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. S. GPUQP: query co-processing using graphics processors.

[9] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *MICRO*, Washington, DC, USA, 2007.

[10] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in GPU programs. In *GPGPU*, 2011.

[11] J. Hellerstein. Optimization techniques for queries with expensive methods. *TODS*, 23, 1998.

[12] J. Meng, D. Tarjan, and K. Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3), 2010.

[13] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[14] K. A. Ross. Selection conditions in main memory. *TODS*, 29(1), 2004.

[15] E. A. Sitaridi and K. A. Ross. Ameliorating memory contention of OLAP operators on GPU processors. In *DaMoN*, 2012.

[16] R. Taylor and X. Li. Software-based branch predication for AMD GPUs. *SIGARCH Comput. Archit. News*, 38(4):66–72, Jan. 2011.

[17] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *MICRO*, 2012.

[18] H. Wu, G. Diamos, A. Lele, J. Wang, S. Cadambi, S. Yalamanchili, and S. Chakradhar. Optimizing data warehousing applications for GPUs using kernel fusion/fission. In *PLC Workshop*, 2012.

[19] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining GPU applications on the fly: thread divergence elimination through runtime thread-data remapping. In *ICS*, 2010.

[20] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In *ASPLOS*, 2011.

APPENDIX

A. BUFFERED WRITES

Shared memory is divided into equally-sized buffers equal to the number of warps in a thread block. When threads in a warp read a warp's worth of data a local prefix sum is computed for the values satisfying the condition and these values are written in the shared memory buffer of this warp. To compute the local prefix sum we use CUDA intrinsics ballot (__ballot()) and population count (__popc()). Ballot function returns a 32-bit integer which combines the condition outcome from each thread, where the i-th bit is set if the condition is true for the corresponding thread in the warp. The population count function computes how many bits were set. When the buffer of each warp is full the threads of this warp write its contents to the global memory in a coalesced way.

B. PTX CODE

We show below parts of the ptx assembly-like code for S2 and S11 kernel functions. S11 has two additional branch instructions that depending on the condition selectivity might result in less efficient code.

```
S11 Kernel
        ld.global.u32   %r49, [%rd15];
        .loc 2 168 1
        setp.ge.s32     %p8, %r49, %r21;
        setp.lt.s32     %p9, %r49, %r22;
        and.pred        %p19, %p8, %p9;
        .loc 2 169 1
        @!%p19 bra      BB2_5; //Branch predication
        bra.uni         BB2_4;
BB2_4:
        .loc 2 172 1
        shl.b64         %rd17, %rd5, 2;
        add.s64         %rd18, %rd16, %rd17;
        ld.global.u32   %r51, [%rd18];

S2 Kernel
        ld.global.u32   %r51, [%rd15];
        .loc 2 105 1
        setp.ge.s32     %p5, %r51, %r21;
        setp.lt.s32     %p6, %r51, %r22;
        and.pred        %p7, %p5, %p6;
        .loc 2 109 1
        add.s64         %rd16, %rd3, %rd14;
        ld.global.u32   %r53, [%rd16];
```