Parallel Quine-McCluskey Algorithm

Team: Emily Allendorf(eallendo), Andrew Wang(herongw)

URL

https://github.com/KingCashVamp/K-Map-Parallelism

SUMMARY

We implemented the Quine-McCluskey algorithm for boolean reduction in CUDA on the GPU and attempted an implementation in OpenMPI of the 8-core Gates machines.

BACKGROUND

Importance of Boolean Expression Reduction

Boolean expression reduction is a fundamental process in the design and optimization of digital circuits. Simplifying these expressions minimizes the number of gates required, reducing hardware complexity, power consumption, and overall cost. It is critical for hardware description compilation and FPGA Synthesis where physical resource constraints make optimization vital.

METHODS OF BOOLEAN REDUCTION

Karnaugh Maps (K-Maps)

Karnaugh Maps (K-maps) are a visual method for simplifying Boolean functions. They represent the truth table of a function in a grid format, with adjacent cells differing by only one variable. Groups of adjacent 1s (or 0s for the Product of Sums approach instead of the Sum of Products approach) are identified and used to generate simplified expressions. K-maps are intuitive and when applied correctly, provide a minimal Boolean representation. However, K-maps are hard to scale and less common in software.

The Quine-McCluskey Algorithm

The Quine-McCluskey method involves finding prime implicants and determining the minimum set of prime implicants that cover all terms (the minimum that can represent equivalent logic as the input). The algorithm can be separated into 3 distinct steps: initialization, finding prime implicants, and finding essential implicants (the final fully reduced boolean expression).

Initialization

A boolean expression can be expressed as a series of terms or a sum of products. For example, if you had 2 variables, A and B, and you had the 3 term boolean expression A'B + AB' + AB you can represent each of the terms as binary encodings of A and B: 01 + 10 + 11. Therefore the minterms would be 1, 2, and 3. These terms must then be sorted into groups of how many ones there are. So, 1 ("01") and 2 ("10") would be placed in group 1 and 3 ("11") would be placed in group 3. This means that for N variables (A, B, C…) there are N+1 groups to sort the minterms into. Through the algorithm it is essential that you keep track of a string representing each term e.g. "11" as well as the minterms involved with that term e.g. 3 (initially all terms are only involved with themselves), a marker to keep track of whether this term has been matched or not, and its group id (the number of 1's there are in the string). This is what became the basis of our "term" struct. Except for the minterms involved, which double on each iteration of the algorithm as terms are matched together (which will be described in the next step), all of the other struct components are of a constant size. We created a "group" struct to keep track of which terms were in each group to be able to access them easily (a hash map). The initialization process ends when all of the initial terms passed in have been sorted into their groups.

Prime Implicants

The next step is the bulk of the algorithm and the most computationally intense. The task is to compare every term to each term in its next adjacent group. So, you compare each term in group 0 to every term in group 1, each term in group 1 to every term in group 2, and so on and so forth until you exhaust all the terms/groups. When you compare the terms you are looking to see if they are one away. For example "10" and "11" are one away from the least significant bit. You can pair or match these terms together to create a new term, replacing the adjacency with an underscore or other character to differentiate. The example pairing then becomes "1_". Then, a new term needs to be constructed with this new string, the terms that were involved to make the pairing, the number of ones/group id, and a tick mark that this term has been matched: "1_", [2, 3], 1, true. This term then has to be resorted into a new set of groups, in this case it will be placed in group 1. While all of this computation can be done independently for each term, the new

groupings must be completely updated and shared among processors before the second iteration can begin. This synchronization is essential because the algorithm depends on fully updated groupings in order to proceed with reducing the terms correctly. It is not correct to simply split up the terms, reduce those fully, and recombine them at the end. This is not logically sound. There is also the case when a term is not matching to anything else at all, the case when the tick mark is not set to true. These can be handled separately by each processor as if they are unmatched in the current iteration they will remain unmatched in all subsequent iterations. However, it is necessary to keep track of these for the final result. Once there are no more possible pairings of any of the terms left, the algorithm has successfully reduced the original expression down to the prime implicants.

Essential Implicants

Once the prime implicants have been generated there is still some work to be done to fully reduce the expression. Each prime implicant covers a certain number of 1's and we want to find the least amount of prime implicants such that all the 1's are covered. This is where the array for each term that keeps track of which minterms are involved is needed. If a minterm is found in one and only one of the prime implicants, then that implicant is considered essential. Compiling all the essential implicants at the end gives the final most optimal solution.

Visualization

To help understand what is actually going on we can look at a simple K-map that lays out all the ones in a graphical matter.

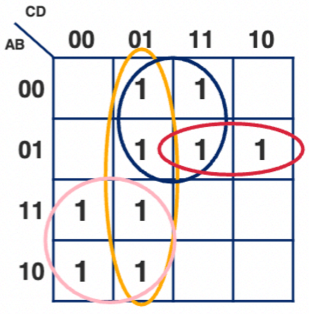

Figure 1: K-map

In this example the circles represent the prime implicants. The essential implicants are then only the circles that are necessary to cover all the ones: blue, red, and pink. The yellow circle in this case is unnecessary as all its ones are covered by the other three circles. These three circles, the essential implicants, represent the minimal boolean expression.

Summary

Data structures: term, group

Inputs: a set of minterms representing a boolean expression

Outputs: the essential implicants (reduced representation of the boolean expression)

Expensive computation and potential for parallelism: each iteration of finding the prime implicants

Workload:

- Dependencies: each iteration is dependent on the last,
- Locality: comparisons are done between adjacent groups (0 and 1, 1 and 0),
- SIMD execution: While certain parts of the Quine-McCluskey algorithm, such as initialization and bitwise comparisons during prime implicant generation, are naturally parallel and amenable to SIMD execution, the need for global synchronization and irregular data operations limits the overall efficiency of SIMD for the entire algorithm.

CUDA

APPROACH (CUDA)

The first approach we chose to take was parallelizing the Quine-Mccluskey algorithm using CUDA. We ran everything on the Gates Machines (Machine 48).

Modifications had to be made to the original serial algorithm to efficiently run on the CUDA architecture. The first of which was to change the use of resizable vectors to set length arrays. We initially used vectors because it was a convenient way to add to an array without knowing its final size. As a result, we were able to utilize the push_back function as well as the .size function that comes with vectors. However, vector push_back is a fundamentally serial operation and does not work with parallel execution unless

many atomic operations are used. This would lead to a negative performance impact on GPU code. As a result, we decided to incorporate set sized arrays in place of vectors. We then added a length variable in each array to keep track of the array size as well as for indexing purposes. Another change that was made was to remove from loops in place of thread indexing.

The approach for CUDA is as follows:

1) Host receives the parsed user input including the number of variables as well as the minterms included
2) Host initializes a global int array of the minterms, a global groupings array for the initial buckets of minterms, and a global remaining minterms array
3) Host launches setBinary kernel where threads are assigned a single minterm value, calculates the binary representation of that single minterm along with its bitcount of ones, then adds that binary value to the thread index in the global array as well as its corresponding groupings bucket.
4) Host initializes a new global grouping array to keep track of the updated groupings, a new global reminder array to keep track of the updated remainder minterms, a global prime implicant array to keep track of the prime implicants, a global unmatched array to keep track of unmatched minterms which would later become prime implicants. Host then launches findprimeImplicant kernel.
5) Device will compare the thread minterm to each of the minterms in the grouping directly after (grouping[i+1]) and if the difference is exactly one, it will add this minterm to the new grouping array with its corresponding bit count of ones
6) The previous process will repeat N times with N representing the number of times there is a difference of one found between a minterm between any of the new groups
7) Host creates a new global essentialPrimeImplicant array then launches find essentialPrimeImplicant kernel to remove the non essential prime implicants
8) Device compares thread primeImplicant to all the other prime implicants looking for a match for all the minterms used

9) Host frees used memory and returns essential prime implicants

One of the challenges we faced was to somehow get all the threads to write to the many global arrays without leading to race conditions or the possibility of overwriting. The first approach to this problem included using a locking and unlocking struct where all the other threads who have not acquired a lock are just busy waiting until the lock is free. Although this theoretically should have worked, we always seemed to hit a deadlock. We believed the issue was that the lock acquisition order was not well defined across all threads. Because we had many different global arrays that a thread had to write to, we needed many different locks for each array. As a result, thread A might have acquired lock 1 and be waiting for lock 2, while thread B has acquired lock 2 and is waiting for lock 1. As a result, we decided to make use of the operations that are inherently built into CUDA architecture such as atomicAdd. Because atomicAdd returned the previous value while also atomically adding to that value, we found that it was useful for the indexing of arrays between threads. This ultimately worked as we used an atomicadd of the array count which allowed threads to add to an array without causing race conditions.

Another challenge we faced was the synchronization of threads during the processing phase. Because of the many operating threads and many array dependencies, it was difficult to pinpoint the places where it was important to ensure all the threads had completed their work. This was made even more difficult by our while loop. We had to ensure that all the threads could reach a syncthreads call as well as ensuring that we minimize the syncthreads calls as each one worsens possible speedup.

Finally, managing the hardware resources efficiently became another concern. CUDA provides the ability to manage the number of threads and blocks used for computation, but fine-tuning these parameters was essential to achieve optimal performance. We had to experiment with different block sizes and grid configurations to find the best combination for the workload, considering factors such as the GPU's compute capability, available memory, and the specific nature of the computation.

RESULTS (CUDA)

We quickly discovered, when running test cases, that the results were heavily dependent on the minterms entered. This is because the Quine-Mccluskey algorithm is unpredictable since with changing minterms, the number of iterations also changes. As a result our main method of measuring performance was to collect the computation time of the program of inputs with varying numbers of minterms while keeping as many of the minterms the same as we can. Our project is mainly concerned with how the Quine-McCluskey algorithm works scales with a lot of variables as well as a lot of minterms as this is the common problem in modern FPGA programs. Hardware description involved an extreme amount of gate and logic reduction to run efficiently. Therefore our results largely looked at the speedup time vs. number of minterms.

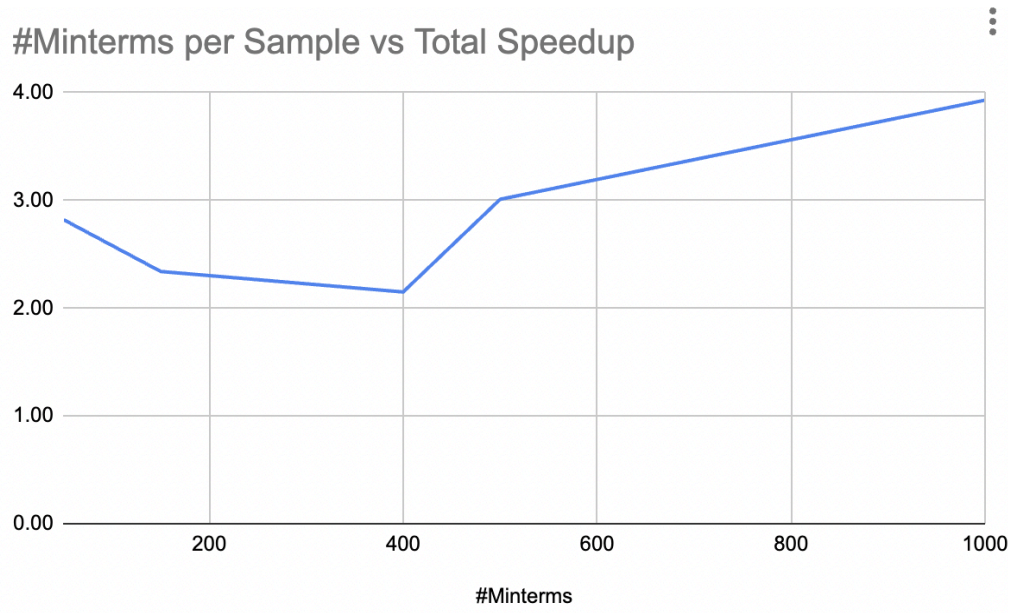

Table 1: Speedup vs Number of Minterms

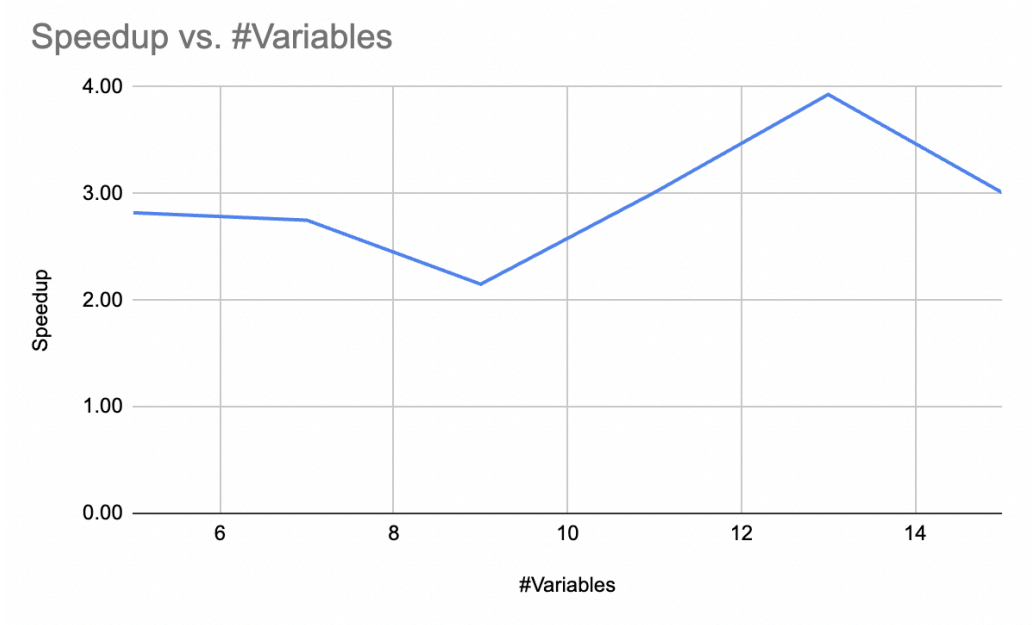

Table 2: Speedup vs Number of Variables

DISCUSSION (CUDA)

Please note that we were only able to get these results based on set minterms. Because we were increasing the number of minterms, for us to try and balance the workload to the best of our abilities, we kept the same first 500 minterms when we tested 1000 minterms for example.

As we can see in table 1, there does appear to be an increase in speedup as we increase the number of minterms. This is because the algorithm has some naturally parallel parts. The first of which is to convert the minterms to their binary values. The other naturally parallel part is the comparison between the groupings. Sequentially, each bucket would have to be compared to the next one when there is not really any dependency between the comparisons. As a result, we did expect there to be a lot more speedup than what was shown, especially for 1000 minterms. One good explanation for this is the fact that the algorithm is unpredictable. The number of iterations is solely dependent on the minterms themselves. As a result, this would lead to poor workload balance. In a SIMD architecture, where all threads in a warp are

supposed to execute the same instruction at the same time, this imbalance results in idle cycles for the threads dealing with independent minterms. For example, one minterm value may be completely independent from the rest because it does not differ by only one with any of the other minterms. This would lead to only one computation of this minterm. In the case that 1000 minterms are being compared, the other minterms may take a lot more than just one computation. This is a serious performance bottleneck.

Another reason for this is the fact that in order to parallelize this algorithm using CUDA, a lot of global memory had to be used. Even though we got rid of vectors, we had to initialize global arrays of set sizes. These set sizes were always either the total number of terms or the total number of variables. This made the memory usage for the CUDA program more memory-bound since the speed of the program is now affected by how fast data can be transferred to and from the host and device. To make it worse, another big challenge was the fact that each thread needed a few arrays to keep track of the data for the minterm that it is responsible for. This leads to a lot of overhead, especially when we launch more threads. These overheads limit the possible speedup.

For table 2, we changed the number of variables while also altering the number of minterms. We do notice an increase in speedup while increasing the number of variables. A good explanation for this is that although different minterms lead to different iterations, having more variables means that the minterms will have more bits, so the overall number of iterations will increase. This would lead to a slightly better work balance between the threads. In other words, the more variables there are, the more bits there are per minterm, and as a result, it will be more unlikely for a specific minterm to have much less computations than all the rest.

We believe we were successful at achieving our goals for our CUDA implementation.

OPEN MPI

APPROACH (MPI)

The next implementation we worked on was an MPI version of the Quine-McCluskey algorithm on the Gates machines with 8 cores. As the iterations progress the number of terms in the lower groups increases since more 1's are replaced with underscores denoting adjacencies. For this reason, our strategy was to break up the work by assigning each processor a set of minterms at the start using the Scatter function. Then, from then on out the processor would be responsible for those terms and the subsequent terms that came as a result of the matching/pairing process. It would be ideal to dynamically assign work to each processor so that, especially on the later iterations, this would avoid the possibility that some processor reduced their terms faster than the others. However, the overhead associated with locking and locking work so that each processor could grab the next available term would be very expensive for a message passing interface so we decided on a statically assigned workload for each processor.

Another implementation we considered for workload distribution was to distribute the work by groups since you are always comparing adjacent groups together which creates good locality. However, this would create a similar and more extreme problem of uneven work distribution because as the iterative process gets closer and closer to the end, the terms left pile up in the lower groups. In all, it was a difficult task to divide up the work evenly because the algorithm, depending on the input, regardless of the scale can be extremely variable. Some inputs will exhibit a lot more adjacencies than others causing the reduction process to be more intensive, while others may have very few possible adjacencies and terminate earlier.

To reiterate, for the MPI implementation we decided on a static work assignment of terms to each processor. From here we needed to make sure, however, to synchronize such that each process had the updated terms after every iteration. This meant that we had to send the information of the pairings to all the other processes to be resorted into their groupings before the next iteration. This provided some challenges because the amount of data to be sent was constantly changing. There was no way to guarantee that each processor would have the same number of terms or the same number of terms it had in the previous iteration which introduced a lot of variability in the message passing.

We also had to make significant changes to our initial serial implementation that relied heavily on vectors, similarly to the CUDA implementation. While sending the vector of wires was fine in assignment 4, this was not a feasible option as we had a vector within our struct itself. This was an issue because it meant that the data we needed to pass to each processor was not contiguous in memory and had to be broken up. While three components of the term struct were of constant size (the string, the tick marks, and the group id), the array with the minterms involved (an array of integers) doubled in size with every iteration which was the reason for using vectors in the first place. This meant that it was not possible to keep all of the elements together in an MPI_Datatype. To address these issues we decided to split up the first three pieces of information into a partial term struct and keep the array of minterms involved as separate data to be sent. Since we also had variable amounts that needed to be sent, we also had to send a header with the amount of terms we were sending beforehand to be able to size the buffers correctly. We then sent the corresponding minterm array data after which the groupings could all be updated by each of the processors. To eliminate the need for a header, we considered sending the terms in batches of equal size. This would ensure that each processor was receiving the same amount of terms each time, but would ultimately require more messages to be sent. A batch size would also be a useful parameter to manipulate based on the scale of the input to balance the size of the messages being sent and the number of messages being sent to find an optimal reduction of overhead in our program.

We intended to implement all of this message passing in rings to reduce the overhead necessary for data communication. Ideally we would have liked to use both asynchronous sends and receives to prevent any processes from running idle, but at this point in the project we didn't have enough time to implement this fully.

DISCUSSION (MPI)

A working implementation for the MPI version was unfortunately not completed by the final deadline, so results on its performance were not collected. However, we will provide a discussion of expected results and limitations of our approach and compare it to that of the CUDA version.

We intended our main data collection to be in the form of traditional speedup graphs of our computation time from 1 to 8 processors on the Gates machines. We intended to also calculate the same metrics as we did for CUDA for comparison purposes. These include speedup graphs sweeping the number of variables involved as well as the number of minterms passed in. The addition of a batch size's parameter impact on speedup would also be an interesting aspect to look at.

SPEEDUP VS. NUMBER OF PROCESSORS: With regard to the traditional speedup graphs, we would expect medium to poor speedup overall when compared to the maximum 8x speedup. This is because, while the algorithm is a good candidate for parallelism because it is intensive and has considerable independent computation, it still has a lot of data dependencies that are especially hard to communicate with message passing due to size variability. In attempting the MPI implementation, we realized that a more intuitive approach would be to use OpenMP with a global set of the current and next groupings. This would allow us to have one set of groups to read from during the current iteration (which wouldn't require any synchronization) and another set of groups to write to in a critical section (would introduce considerable overhead) to replace the current groupings for the next iteration. Our sequential program is mostly just large nested for loops so taking advantage of OpenMPs pragma omp parallel for process would be easy and efficient. An MPI implementation is more convoluted which introduces more complex synchronization and considerable overhead. Ultimately, we can conclude that the main bottleneck of our program is the sheer amount of data that we need to communicate between the processors and the variability in the amount of data associated with each term. One thing we could do to reduce the amount of data would be to eliminate the section of the algorithm that calculated the essential implicants and only do the prime implicants. This would completely eliminate the need for the array with the minterms involved with each term which proved to be the biggest issue. However, this would mean that our final answer would not be fully reduced. There would be a tradeoff in the boolean reduction which would require less hardware on an FPGA, and parallelism which would improve speedup.

SPEEDUP VS NUMBER OF MINTERMS: As we increase the number of minterms passed in we would expect the speedup to increase but plateau at a certain point. This is because when we have so few minterms, not only is the program not particularly computationally intensive, but the variability in the number of iterations is very great. As we get to higher numbers of minterms we would expect that the number of terms assigned to each processor based on our implementation would become more evenly distributed. Therefore, given enough minterms, we would expect our program to be able to overcome the communication overhead between processors, but that this trend would be highly variable before that point. This would be in line with the results of the CUDA implementation. Then as our graph approaches the maximum theoretical speedup (8x given we run these experiments on 8 processors), the graph would start to plateau with increasing the number of minterms. This is because even though the workload distribution would be better with more minterms, the amount of data needed to be sent would be much much greater. This does not just come in the form of more terms being sent (which ideally does not impact the speedup that much because it would also slow down the sequential version of the code), but the array of minterms involved increases exponentially with every iteration. This would result in very space intensive message passes. Optimizations would need to break up these arrays and send them over multiple messages which would be difficult because it is necessary to ensure that they correspond to the correct partial term that was communicated separately.

SPEEDUP VS. NUMBER OF VARIABLES: Given a sufficient amount of minterms are initially passed, we would expect an increase in speedup as the number of variables increases. As we increase the number of variables, (ABCD…) we increase the number of values we can represent, increasing the range of the minterms we can pass in. (The variables are essentially bits.) Our algorithm can theoretically process infinitely many variables but we capped it at 32 (like the values represented with an unsigned int). As the number of variables increases, and the number of minterms stays the same, the data, if random, would become more spread out and less susceptible to matchings. This is because with such variable numbers

the chances that there is an only 1-off change between two would be unlikely. Since one aspect of our implementation is that these unmatched terms are handled only by their assigned processor, it would make our program more parallel, only requiring the communication of these unmatched terms at the very end of the prime implicant generation. This is different from the CUDA implementation where we saw variable results.

REFERENCES

- Figure 1 from 18-240 lecture slides
- Test cases generated by Chat GPT (long lists of minterms/numbers)

DISTRIBUTION OF TOTAL CREDIT:

- Andrew (50%)
- Emily (50%)