LQCD Optimizations through Automatic Application of Physical Intuition to the Naïve Structure

Richard Sollee

Abstract

Lattice Quantum Chromodynamics are a set of physics problems that are expressed as a summation over the lattice space of a problem as well as a set of weights which leads to an asymptotic runtime for a given problem as $O(L^2W^2)$ where L is the size of the lattice and W is the number of weights. The main part of the summation accesses input variables based on function mappings dependent on the current weight index. The range of these accesses is independent of the weight index which allows precomputing the multiplications based on all possible accesses and then using the precomputed values in the main summation loop. After testing multiple precomputation configurations, precomputing over all indices that depend on weight indices and not precomputing over space provided the fastest runtimes. The asymptotic runtime remains the same with the precomputations but the number of iterations of the loop body in the precomputation remains fixed as the number of weights increases while the main loop does not. This leads to a crossover point where the extra work of precomputing over a larger range than is used leads to less overall floating-point operations because of the amount of reuse in the main loop of the precomputed values.

Introduction

Lattice Quantum Chromodynamics faces massive scalability problems for computing desired outputs because the naïve computation scales with $(L^3)!$ where L is the size of the lattice used. Physicists have used physical intuition about the system to find methods of precomputation which allow the scaling to be dominated by $(3A)!/(3!)^A$ where A is the atomic number of the atom being investigated. The manual implementation and optimization of the code written for these computations takes thousands of lines of code in a DSL for loop computations called Tiramisu (Amarasinghe R. B., 2018), which generates 10s or even 100s of thousands of lines of C or cuda code. This project seeks to create a method of creating a computation using the automatic application of the physicists' optimizations to produce competitive runtimes with significantly less code so that different problem sizes and configurations can be run with minimal setup effort.

LQCD Overview

Quantum Chromodynamics (QCD) is a method of studying the strong force interaction between quarks and gluons which are the fundamental building blocks of atoms. Lattice Quantum Chromodynamics (LQCD) simplifies the computations used in QCD by discretizing space into a lattice which has a finite number of points and defining field values at each of these points. Limiting the number of spatial locations limits the degrees of freedom that need to be considered in the computations which allows computations to be done in physical situations where nonlinearity makes other methods hard or impossible. (Wilson, 1974)

Previous Physicist Work

Previous work on computing correlator functions, a useful number for physicists, using LQCD has implemented optimizations based on the physical structure of the system. One example used precomputations over items in the formula relating to specific quarks to make precomputations they named hadronic blocks. (Detmold, 2013) Another paper which introduced a multi-baryon system used precomputations over the mathematical structure representing the baryons to make precomputations they call baryon blocks. (Amarasinghe S. B., 2021) The code for each of these problems was handwritten for each problem and required hundreds of lines of code for even the simplest optimized case.

Code Repository Setup

A github repository for this project had previously been made to house the python source code and tests for converting the LQCD problems to our IR and converting the IR to a tiramisu function or numpy computation. The tiramisu repository on Github contains the benchmarks the physicists have written to solve a couple of their LQCD problems. An initial investigation into their code revealed issues with their constants that caused segmentation faults. The C++ code the physicists made for their baryon problem was then added to the repository for this project to have greater control over it. The C++ code was debugged to discover the constant issues and resolve them.

Makefiles were created to allow running the benchmarks in any environment where the dependencies have been installed via Spack such as the docker image available from building the dockerfile in the repository. The Makefile obtains the locations of the libraries needed for the compilation by searching through the `CMAKE_PREFIX_PATH` and then performing string operations to get the paths into the desired format.

Intermediate Representation (IR)

The physics problems being solved in our LQCD scenarios can be represented as mathematical formulas. In an earlier version of this work, the formulas were simply represented with a tree structure that included basic mathematical node types such as addition, multiplication, and summation. Upon looking at the optimization work done by the physicists, we realized that the ability to express precomputations would be crucial to speedups. Therefore, we created a new intermediate representation structure that supported "Let" statements for representing precomputations. In addition, the new IR structure supports iterating over symmetry groups and being able to access their parity when in the previous IR the symmetry groups had to be expanded before being put into the IR structure.

Compilers used and IR to compiler code

Overview and Tiramisu Compiler

Prior work to the project also involved creating a system for converting this IR into a tiramisu function that computes the desired result. The Tiramisu compiler is a polyhedral compiler where algorithms can be expressed as computations with indices that are iterated over. The conversion from IR to tiramisu is relatively straight forward with addition and multiplication being their respective operations between tiramisu expressions. Sums in the IR are converted into tiramisu computations where an initial computation initializes the buffer locations to zero and another computation adds the summand for particular index values to the buffer location determined by a subset of those indices. Let statements are represented by creating a computation for the Let statement and a computation for the use statement, then assigning the let value statement to a variable which can be referenced in the use statement and adding an ordering to the computations so that the let statement is computed before the use statement.

Iterations over symmetry groups are implemented through a computation that iterates over the size of the symmetry group. Accesses into the permutations of the symmetry group are done by passing as input to the function an array mapping a number in the range of the size of the permutation group to a specific permutation and then accessing this input value with the iterator index of the symmetry group. The IR also supports stabilizing a permutation group and the conversion into tiramisu code is similar to the plain symmetry group but with some sorting of the permutations by the stabilized index since the value of the stabilized index is variable but the index in the permutation is fixed in the IR.

Switch to Halide compiler

Towards the end of this project, issues were encountered with converting the IR to Tiramisu properly for the precomputations to be done at the desired loop level to ensure the precomputation space was a reasonable size. These issues coupled with some other problems

encountered with Tiramisu development led to the decision to switch to Halide (Ragan-Kelley, 2017) as the compiler. The graduate student I have been working with wrote a new converter which translates the IR into Halide code. After the initial converter was written, the rewritten LQCD IR was not running which led me to debug the Halide converter and get the example working.

Nameless Representation

One potential issue when rewriting the IR is ensuring that we do not duplicate Let statements. In order to ensure we can have one canonical representation of the IR, a function was written to convert a given instance of the IR to what we call a "nameless" instance of the IR. The purpose of the nameless representation is to use a variant of De Bruijn indices to ensure two IR trees are equal as long as the only difference in their structure is the naming of their indices being iterated over. The `get_nameless` function returns the IR with all the indices relabeled so that if two trees are equivalent then a comparison of the tree values returned by calling `get_nameless` with each tree will be equivalent.

This process of creating a new tree was done primarily by match statements. The current expression was matched against the possible IR types and then a new object was returned constructed from children that had `get_nameless` recursively called on them. A special case occurs whenever a Sum is encountered. When a Sum is matched, the free indices and iterating indices are relabeled with increasing integers with the first index encountered in a Sum in the whole tree being labeled with zero. The mapping of old names to new numbers is passed as an optional parameter to `get_nameless` which is used in the recursive case and allows replacing uses of the index in the summand expression with the new index name.

IR Optimizations

Prior work by physicists

The primary goal of the project is being able to achieve the speedups shown in the optimized physicist benchmark code without having to handwrite the optimizations for every new LQCD problem. In the physicist code, they only have one type of precomputation which they do on summations that occur lower in the tree that do not depend on the psi or weight inputs to the program. The part that remains in the precomputation is primarily a summation over the propagator input.

Intuition

The computations being done for the LQCD problem mainly involve summing the result of multiplications between inputs which have their indices accessed based on different function results which depend on the variables being iterated over. The domain is much larger than the range of these index expressions which are used to index into the inputs which are summed over. This leads to the idea of doing precomputations over the range of the index expressions and forming an intermediate result which can then be indexed into based on the index expressions used in the main summation. Doing these precomputations does not change the asymptotic complexity of the algorithm because the main indices must all be summed over but it does allow reducing the number of floating-point operations because many of the iterations of the main loop are all mapping to the same result which then only needs to be computed once.

Initial Rewriting

The initial method for performing rewrites on the given structure performed precomputes using all index expressions. The rewriting function first found all index expressions and then found the ranges that these expressions spanned over. A precompute expression was then formed which replaced all index expressions with indices that iterated over the corresponding ranges. The precomputed expression was then used by placing it in a sum which iterated over the original iterating indices and indexed into the precomputed value using the index expression which had been replaced in the precomputed expression. The initial implementation was simple to create then debug but for any non-trivial lattice size the precompute space was far too large to fit on a reasonably sized machine.

Limiting the Precompute Scope

The next step of the optimizations was to limit the precomputation space so that the precomputation could fit on most machines. The first reduction of the space was accomplished by only precomputing over the lattice space for the sink vertices and recomputing the precomputations for every source vertex in the lattice space. This reduction reduces the space needed by a factor of the lattice size. Switching the location of the precomputation to inside the first iteration of the lattice space does not increase the number of floating-point operations compared to storing the entire precomputation. The number of operations does not increase because the range of the index expressions for the iteration over the lattice are the same as the original iteration over the lattice so the moving of the precomputation inside the loop only decreases space usage.

Theoretical Analysis

For a given LQCD problem, there are parts of the iteration space which are fixed in size and others which can vary. The index expressions which depend on spin and color of the quarks will always have a fixed range and the size of the permutations which are iterated over will also be fixed. The changing parameters to the problem are the size of the lattice and the number of weights given. The loops iterate over the size of the lattice twice as well as the number of weights twice leading to an asymptotic run time of $O(L^2W^2)$ where L is the size of the lattice (which can also be written as $L = lwh$ where the variables on the right hand side are length, width, and height of the 3d lattice respectively) and W is the number of weights. The rewrites of the program are unable to reduce this asymptotic runtime because there is currently not a way to avoid the final iterations over the space and weights. However, the precomputations can reduce the number of floating-point operations done because the space of the precomputations is finite for the given problem size. This finite size of the precomputations prevents drastic increases in floating point operations as the number of weights increases because the index expressions which are precomputed over depend on the weight index for their evaluation but not for their range so once the space is precomputed over then the multiplications for the main part of the expression do not need to be recomputed for the new weight indices and just need to be accessed in memory.

IR Optimization Implementation Analysis

Available Optimizations

The primary decisions for the rewrites rested in deciding whether to precompute over half of the space indices or none and whether to precompute over one weight iterator or both. As a baseline, the physicists' implementation of their optimizations in C was used for comparison. Ideally, the performance of their tiramisu implementation would have also been used but at the time of profiling the compute environment was unable to support running the tiramisu code because of a lack of AVX. The physicists' attempt at optimizations appears to precompute over half of the space indices and one of the two weight indices. Their C implementation of their optimizations takes around 0.075 s to run for a lattice size of 8 and a weight scaling factor of 1. This runtime is around 3 to 4 times longer than two of the quicker optimizations we implemented in Halide and since the lower scaling factor is the worst case for our optimizations, we can expect that they would continue to outperform the physicists C implementation for larger lattice and weight scaling factor sizes.

Optimization Timing Collection

Timing data was collected for running the Halide code generated from the optimized IRs and the unoptimized IR by running the code on the Lanka login node available to the MIT CSAIL

COMMIT group. At the time of data collection, only the login node was available which reports having 18 MB L3 Cache and a base processor speed of 2.00 GHz and a turbo speed of 2.40 GHz. The four different types of precomputes tested where the combinations of deciding whether to precompute over half of the space indices or none and whether to precompute over one weight iterator or both. The code was tested running with lattice sizes of 8 and 27 and scaling the number of weights by 1, 10, and 100 leading to six different test sizes. For each test size, every IR configuration was run 10 times and the median time was taken as the result. The results can be seen in Table 1 in the figures section. Based on the timing data, the best performing precompute optimizations are to not precompute over any of the space indices and to precompute over both α indices.

Theoretical Analysis

There are several possible explanations why not precomputing over any space indices performs better than precomputing over one set and why precomputing over all the α indices performs better than precomputing over just one. When precomputing over one set of space indices the size of the precompute increases compared to not precomputing over the space indices. In addition, no floating-point operations are saved by doing the space precomputation because all the variables which depend only on the space are already multiplied in the precompute statement so the number of floating-point operations does not change because the precomputation loop body will be run the same number of times regardless of whether the space loops are inside out outside of the precomputation. Since the only theoretical difference for the space precomputation appears to be the precomputation size, one possible explanation for the speedup with a smaller precomputation is that more of the precomputation fits in lower-level caches and experiences fewer cache misses.

Precomputing over both α indices creates a larger precomputation space than just iterating over one of them. Based on the other optimization, precomputing over a larger space would seem to be bad. However, when we precompute over the α indices, we do not iterate over their size in the precompute but instead iterate over the sizes of all the index expressions which use them and only actually iterate over the size of the α indices when using the precomputed values. Therefore, since the size of the precomputation space does not depend on α, the number of floating-point operations done in the precomputation should not increase as the number of weight (α) increases. However, if we only precompute over one α index, the number of times the precomputation loop body is run can be expected to increase. This means that precomputing over both α indices should lead to less floating-point operations overall because as the number of weights increases, the results of the floating-point operations done in the precomputation are reused more.

Practical Analysis

To effectively see the advantages of the precomputations, we must discard big O notation and look at the number of operations that are occurring inside the work of each loop. By analyzing the generated Halide statement output of the fastest optimized version, the inside of the precomputation can be seen doing 21 floating point operations per iteration. The main loop using the precomputation then does 4 floating point operations per iteration. In comparison, the unoptimized code does 25 floating-point operations per iteration. This reveals that, when the main loop in the optimized code is run less times than the precompute, there is more work being done on the rewritten code. However, the main loop in the rewritten code does far less work than the naïve loop so the savings of running the main loop eventually outpace the initial extra work done in the precomputation looping.

The following formulas allow us to calculate the expected crossover point for the rewritten code to outperform the naïve code.

$$t_n = 2f_o W^2 L^2$$

$$t_o = L^2(f_l P + 2f_m W^2)$$

In the above equations, t_n, t_o are the expected runtimes of the naïve and optimized versions respectively. $f_o$is the number of floating-point operations in the main body of the naïve loop while f_l, f_m are the number of floating-point operations in the precomputation loop and the main loop respectively of the rewritten loops. L and W are the size of the lattice and the number of weights respectively while P is the size of the iteration space for the precomputation loops in the rewritten structure. The timing for the naïve approach is simple because the main loop is simply run W^2L^2 times with an additional factor of 2 for the symmetry group of size 2 in the problem. For the optimized structure we run the precomputation loop L^2P times because for every pair of spatial coordinates we run the entire precompute size which is fixed. In the optimized structure, we then run the main loop the expected W^2L^2 times with an additional factor of 2 for the symmetry group of size 2 in the problem. To solve for the expected crossover point we can do the following manipulations.

$$2f_o W^2 L^2 \ = \ L^2(f_l P + 2f_m W^2)$$

$$2f_o W^2 \ = \ f_l P + 2f_m W^2$$

$$2f_o W^2 \ - \ 2f_m W^2 \ = \ f_l P$$

$$W^2 = \frac{f_l P}{2(f_o - f_m)}$$

$$\mathrm{W} = \sqrt{\frac{f_l P}{2(f_o - f_m)}}$$

As discussed before, the values for f_o, f_m, f_l are 25, 4, and 21 respectively. Looking at the Halide statement code or the generated IR structure also allows computing P to be 93,312. Plugging these values into the given equation gives an expected crossover point of 216 for the number of weight iterations. Our base problem uses an initial iteration over 24 weights and the scaling factor used in the tables and graphs for the data is a multiplicative factor to the number of weights so dividing 216 by 24 gives us an expected crossover point of 9 for the weights scaling factor. This crossover point can be seen in Figure 1 where it occurs between the scaling factors of 11 and 21 which aligns well with the expected result from looking at the generated Halide statement files.

Figure 1 shows the timing data collected for a lattice size of 8 with an increasing weight scaling factor. For every weight scaling factor, the optimized code and naïve code were each run 9 times with the median value taken for the plot. The plot shows the expected quadratic shape for both the naïve and optimized code because the optimizations do not change the quadratic factor in the weights but instead change the constant factors in front which leads to better performance.

Future Work

There are many opportunities to further speed up the runtimes of these problems. The first area to tackle would be investigating a way to precompute only the values that will be used in the main loop which would decrease the constant factor represented by P in the prior section when calculating runtimes. In addition, parallelizing the computation provides an easy way to gain potential speedups and the problem seems to lend itself to a potentially easy parallelization over the space iterations. There are also physical aspects of the system that could allow more rewritings of the IR which could potentially allow more computation results to be reused. There may also be methods to analyze the inputs for repeated values and partition the iteration space into areas that use different sections of the input which have numerical equivalence.

Individual Work Distribution

Most of the work mentioned above was done by Richard Sollee. He refactored the benchmark tests and Makefiles so that they can easily be run in any environment where spack is used for installation and ensured that the code could be run in the docker container from the provided image so that the outputs can be replicated. He also did the initial creation of the `get_nameless` function as well as adding thorough tests to the testcases and achieving ninety-nine percent code coverage in the function to ensure that the results from the function are a valid IR that can be turned into a tiramisu function. He also did the investigation into the physicist code to determine potential rewrites. He wrote the function which does the rewriting of the IR and did the experiments for determining the optimal rewrite configuration. Teodoro Collin has also assisted in this project with mentorship and was the primary person debugging the physicist

implementation so that the benchmarks ran at all. Teo also created the IR to Halide converter which allowed the new rewrites to be able to be run.

Conclusion

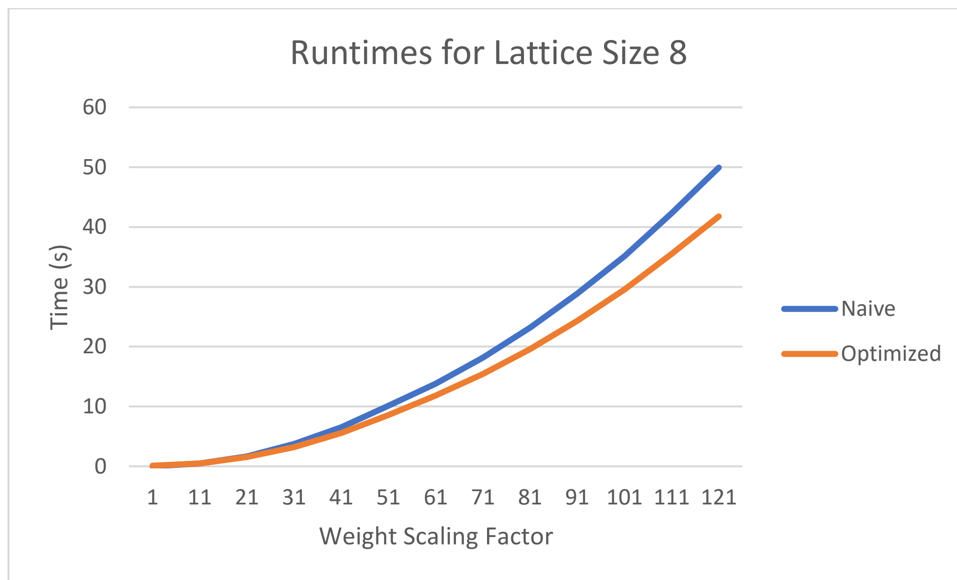
We have provided an automatic optimization system for Lattice Quantum Chromodynamic problems. Our system allows describing an LQCD system which is then turned into an optimized Halide function which can be run to compute the desired results. Several methods of precomputation were tested as optimizations with the fastest optimized function showing a clear speed up over the naïve sum as the number of weights used in the problem increases. Future work includes exploring parallelism as well as reducing the unused parts of the precomputation space.

Figures

Optimization Comparisons (Table 1)

Lattice Size	Weight Scaling	Max Precomp	Precomp Space	Precomp α	Min Precomp
8	1	0.148 s	0.136 s	0.019 s	**0.018 s**
8	10	0.457 s	0.547 s	**0.391 s**	0.411 s
8	100	27.613 s	36.462 s	**21.764 s**	24.276 s
27	1	1.737 s	1.543 s	**0.208 s**	0.210 s
27	10	5.163 s	6.229 s	**4.272 s**	4.700 s
27	100	292.103 s	398.683 s	**222.099 s**	261.735 s

Halide Code Runtimes vs Weight Scaling Factor (Figure 1)



Works Cited

Amarasinghe, R. B. (2018). Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code.

Amarasinghe, S. B. (2021). A variational study of two-nucleon systems with lattice QCD. doi:10.48550/ARXIV.2108.10835.

Detmold, W. &. (2013). Nuclear correlation functions in lattice QCD. Physical Review D, 87(11). doi:10.1103/physrevd.87.114512.

Ragan-Kelley, J. a. (2017). Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing. *Association for Computing Machinery*, 106–115.

Wilson, K. G. (1974). Confinement of quarks. Phys. Rev. D, 10, 2445–2459. doi:10.1103/PhysRevD.10.2445.