15-745 Project Milestone Report

Joey Fernau (rfernau); Norman Ponte (nponte)

April 14, 2017

Restate Motivation

So far, our motivation is still very similar to the originally stated motivation. Most of the insight so far has come in narrowing down what our optimizations will do and how they interact with each other. Our motivation is still to create a project which annotates code and then using runtime information modifies the source code to improve function calls and fix SIMD divergence in warps.

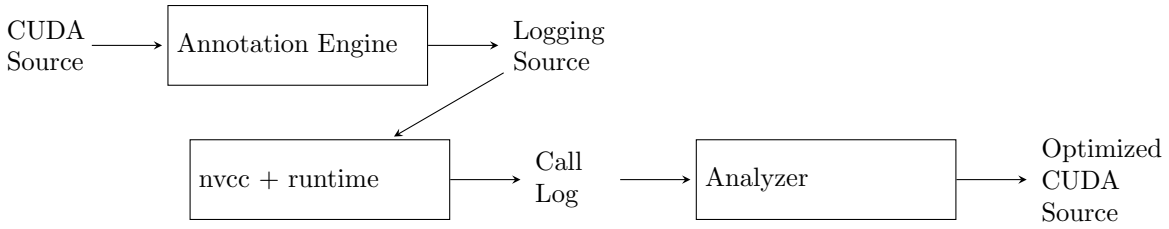

Figure 1: System Overview

Proof of Concept

One of the first steps we wanted to take in this project was to have a proof of concept that the changes our run-time compiler would make to the source code would lead to improvements. To do this we wrote a program and manually modified it with the changes we would want our system to do. This was just a small proof to ourselves that in the best case our system can improve the performance of a given program. The transformations we applied were two-fold. The first one was to break up the kernel function into a switch based on arguments which calls more optimized versions of the original kernel function. The second transformation was to manually clean up the divergence in warps by mapping threads which take the same path to the same warp (see figures 2&3). Note that both transformations must occur for speedup as explained next.

First Optimization [Device Function Argument Optimizer]

We tested the first optimization using an example where the optimized functions ran much less code if the first argument was different and we found ability to speed up the code by around **1.05x**, which is negligible. We were not expecting speedup with just this transformation alone. The key problem with just this optimization alone was that the warp divergence caused the instances that ran the shorter code have to wait on the threads that ran the longer code. This means minor speedups are just due to threads being scheduled to the correct warp already. The key insight we got from this test was that we need to reschedule CUDA threads to run on the same warp. While there are cases where this optimization alone provides speedups, we will only see substantial speedups when this is combined with warp scheduling.

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;

if (idx < N)
  output[idx] = helper_func(input1[idx], input2[idx]);
```

Figure 2: Original Code

```
int idx = threadIdx.x + blockIdx.x * blockDim.x;
int ridx = warp_remap[idx];

if (idx < N)
  output[idx] = helper_func(input1[ridx], input2[ridx]);
```

Figure 3: Warp Optimization

Second Optimization [Warp Scheduling]

The second optimization involves warp scheduling. For this we just manually mapped threads to a different input combining inputs into a warp that run the same code. This allowed us to have whole warps running a much shorter section of code. This optimization is also nothing without the other since all the code is running the same code in the first instance and therefore there isn't a chance for divergence in the warps. When we combine both of these optimizations we got speedups of around **2x** from our simple example.

Updated Timeline

Week of	Todo list for week
Mar 26	Make a test program to determine what optimizations will give the best speedup to deliver a proof our concept.
Apr 2	Write+debug python script that adds in annotations to the source code where we will gather information on the running kernel program.
Apr 9	Write program to modify the source code to include the optimized function calls and fix problems with divergence. Write Milestone Report to be submitted on Apr. 14.
Apr 16	Run custom tests in order to debug and determine performance of the system. Profile the code in order to determine what is creating speedup. Consider other optimizations to apply (not just the function argument analysis), start implementing them.
Apr 23	Further optimize thread warp mapping edge cases. Start to test system on real world problems determine the characteristics of the system under different programs looking for edge cases and good/bad examples of our optimizations. Implement more of our additional optimizations.
Apr 30	Explore possibility of doing this dynamic analysis at run-time (as opposed to all at compile-time), similar to a JIT compiler. Continue implementing additional optimizations.
May 7	Finish final report (due on May 9th)

Evaluation Plan

Our original stated evaluation is to run our compiler/system on our own custom test cases and open source benchmarks. So far we have one custom test case. This is the code that we hand optimized in terms of optimized functions and thread index remapping. This allows us to have a baseline for the first test we will run our system on giving us an idea of how close our system performs to hand done optimizations. Our current work has also given us a better idea of what type of programs to use to evaluate our system. We suspect that our system will perform well in cases where the kernel function has a high amount of divergence and many of the threads run different segments of code.

Another key to our evaluation will be to profile our code in order to objectively show the divergence and where the modified code is getting the performance increases and where it is spending most of the time during the run-time. For our evaluation we want to present the complete picture of our optimizations and their best use cases and where the overhead might not be worth the increase in performance.

Refer to figure 1. Our system currently: (1) takes in CUDA source as input, (2) uses the Annotation Engine to insert CUDA thread friendly printf's that log device function call types and values, (3) runs this annotated CUDA source (compiled with nvcc) to output a call log, (4) inputs this call log into the Analyzer. The analyzer currently has implemented histogram counters of function call argument values. For each device function and for each argument of the function, a counter is used to keep track of how many times this argument was a particular value. This will allow use to determine which arguments should be kept variable and which arguments should be fixed. If an argument's value has a very high histogram count, an optimized function should be made with this value fixed.

Concerns

So far our concerns remain the same. Our motivation is to be able to run our system on any code and achieve greater then or equal performance then the original code with more falling into the former category. Our high level concern is that we provide the user of our system with greater or equal performance of the original code. We have proven our concept works in a specific use case but a high amount of our concern comes from being to generate these optimizations manually and making sure our optimizations do not have the ability to make the code worse. Since our system takes the source code annotates it and then runs it and then modifies the source code our system will most likely take a non-significant fraction of time to run and therefore we have to promise to the user to deliver speedups which justify the long compile time and source code modification.

Our main technical concern to our performance improvements is memory management. Memory management is very important to GPU code generation as due to having a smaller cache maintaining memory in cache is very important to achieving high performance. Having the *warp_remap* array remap the original indices so that we no longer have consecutive memory accesses is going to slow down our optimized code as we make more fetches from main memory. We are looking into ways of limiting these effects using scatter and gather intrinsic instructions and perhaps using shared memory to better utilize the cache and avoid other undesirable memory effects.