Compilers Project Proposal

Heman Gandhi Rutgers University

Abstract

In many up-and-coming fields such as deep learning and crypto-currencies, the SIMD parallelism of GPUs is paramount. Since the data are all that vary over the computation, GPUs perform well.

That said, the GPU is fraught with complexities: the threads must synchronize with one another to the point where even divergent control flow can cause slow-downs.

In handling this, the PTX assembly is structured rather differently from X86 and other common ISAs. This differences in structure are yet to be tackled by super-optimisers and this project hopes to remedy that.

1 Introduction

This project hopes to address the lack of super-optimizers for PTX assemblies. In doing so, the hope is that specialized GPGPU programs may be further optimized automatically.

The hope is to accept either serial or parallel compiler IR and covert it to super-optimal PTX. There are various challenges with this. Should the input be serial code IR, loop synthesis will play a dominant role in this project. Furthermore, various projects seem to have tackled the conversion from serial code into CUDA at a higher-level, so it is likely that a super-optimizer may be more useful in conjunction with these other tools. [1] [2]

At a finer granularity, loops have also been thoroughly studied, in and out of the GPGPU world: general equivalence checking [3], SIMD X86 code [4], and even the optimization of loops in CUDA [5], have been investigated.

However, these prior works are not at the granularity of the super-optimiser. Most of these optimisations handle higher-level data-dependencies and arrange CUDA kernels as their "atomic blocks." In the same vein as recent X86 super-optimization works [6], the hope is to optimise segments of PTX assembly to create the best possible code.

2 Related Works

There are two facets of the study of super-optimisers for CUDA and the synthesis of this delicate code. Firstly, a lot of progress has been made on high-level tools and it is possible that a few insights from them may be useful. Secondly, super-optimisers are not foreign to X86 and more common ISAs so much may be learned from their implementations.

2.1 Super-Optimisers

The key algorithm behind the super-optimiser [6] is a context-aware graph search with various clever optimizations that minimize the number of redundant computations. These optimizations entail keeping the same items pruned: so if the search finds that a class of programs can be pruned, it keeps them pruned across all the relevant restarts and input refinements.

In addition, by finding windows with tighter pre and post-conditions, the super-optimiser hopes to find better code segments that work in the context of the current program. This would be very useful for CUDA programs given the conditional guards built into PTX: the preconditions might be able to be inferred from guards so that the program produced is better in the context of the larger code segment.

However, the conditional hazards of CUDA – that PTX instructions can be prevented from running at the instruction level, means that in order to optimise the bulk of the PTX ISA, conditional branching cannot be ignored. This will have to be a key extension on the original super-optimiser as it is a limitation of the current implementation.

Another interesting work looks into the use of X86 SIMD instructions. It is quite different from simply optimising already SIMD PTX assembly, but would be useful to process serial IRs into even sub-optimal PTX code. The approach would be to relate parts of a loop, as in [4], so that the serial loop can be unrolled by a factor and that would allow loop-equivalence checkers to process the code.

Similarly, loop equivalence checking could be implemented through the data-driven method suggested by [3] where linear algebra is used to solve for relations between the variables. These relations, when maintained over the loop, should lead to similar terminal states for equivalent loops. This may be useful in ensuring loop optimisations should they arise in the PTX super-optimiser.

2.2 GPGPU Synthesizers

In general, there are synthesizers for CUDA code. These synthesisers, however, focus primarily on loops and optimizations thereof. Their input and output formats are not compiler IRs and they do not focus on the granularity of instructions.

2.2.1 Optimizing CUDA Loops

In [5], the interactions of nested loops are studied. The loops are optimized from pseudo-sketches that do not specify the implementation details of every loop, leaving vectorization, unrolling, and memory allocation as choices. The synthesizer then decides on the optimal reasonable choices by essentially trying each one with a dose of clever pruning. A cost function is given and any possible program is only fully evaluated if it could perform better than the bound.

The cost model used to prune the search space may provide insights on optimising memory accesses, should that be tackled as a part of the project.

Additionally, many of the benchmarks provided in the paper are likely relevant for optimising even straight-line PTX.

2.2.2 Alternative Inputs for CUDA Synthesis

Two approaches have also already looked into better approaches to user-input for CUDA synthesis.

In the CUDA community, there is a mature OpenMP-based system [1] that allows programmers to convert sequential code to parallel by helping the compiler through domain-specific directives. This approach does not offer many insights into synthesis algorithms. However, it provides useful benchmarks with a variety of algorithms.

Another input format simply specifies that data dependences between kernels and optimises their layout. [2] Similar to the above, this does not offer too many insights into the synthesis searches needed to super-optimise PTX, but samples the same benchmarks as above so would be a good resource to compare against.

2.3 Nvidia Resources

Various resources from Nvidia would also be needed to encode the ISA and further understand the format of the input.

2.3.1 PTX Semantics

[7] explains the internals of the PTX assembly including the semantics of various instructions.

This will be paramount to optimisation as it will be the model encoded for the purposes of super-optimisation.

2.3.2 NVCC IR

The IR provided by NVCC is documented in a few places and can be readily produced.

[8] contains various samples of the NVCC IR that can be used for initial testing and understanding. It is nicely similar to the LLVM IR.

Furthermore, details and instructions are laid out in [9].

3 Benchmarks

Hacker's delight and other small segments of straight-line code will likely be the first victims of super-optimisation to test the encoding of the PTX ISA.

Most useful benchmarks would arise from the related works aforementioned: the hiCUDA [1] framework's tests and tests from the CUDA loop optimiser [2] would likely be the final benchmarks.

The super-optimiser should take hiCUDA or otherwise well-produced CUDA code and optimise it for a measurable performance increase. These algorithms span a variety of problems and uses, among which may be (from [1]):

1. Black-Scholes Option Pricing
2. Matrix Multiplication
3. N-body Simulation.
4. Coulombic Potential
5. Sum of Absolute Differences
6. Rys Polynomial Equation Solver
7. Magnetic Resonance Imaging FHD

4 Milestones & Timeline

In order to progress on this project over the semester, the following time-line would be paramount:

- 20th Feb.: Proposal received and reviewing starts.
- 27th Feb.: Finished installation of nvvm, hiCUDA, and other relevant tools. Download and finding relevant benchmarks.
- 6th Mar.: Successful encoding of a few bitwise PTX instructions. Testing to be done with samples from hacker's delight.
- 13th Mar.: A few more instructions.
- 20th Mar.: Research into conditional verification and implications on PTX.
- 27th Mar.: Some working conditional test cases.
- 3rd Apr.: More progress on conditional cases.
- 10th Apr.: Beginning to encode process larger benchmarks (like those from [1]).
- 17th Apr.: Debugging, understanding the implications of code size on the optimisation.
- 24th Apr.: Debugging, understanding the implications of code size on the optimisation. Time permitting, investigation into better heuristics for optimisation.

This may be ambitious depending on the complexities of PTX, so some of the larger benchmarks may not be fully tested against. The minimal hope is some optimisation of conditionally evaluated instructions and straight-line code.

5 References

References

[1] Tianyi David Han, Tarek S Abdelrahman. hiCUDA: High-Level GPGPU Programming. *IEEE Transactions on Parallel and Distributed Systems*, 2011.

[2] Hanwoong Jung, Youngmin Yi, Soonhoi Ha. Automatic CUDA Code Synthesis Framework for Multicore CPU and GPU Architectures. *PPAM*, 2011.

[3] Rahul Sharma, Eric Schkufza, Berkley Chruchill, Alex Aiken. Data-Driven Equivalence Checking. *OOPSLA*, 2013.

[4] Gilles Barthe, Juan Manuel Crespo, Sumit Gulwani, César Kunz, Mark Marron. From Relational Verification to SIMD Loop Synthesis. *PPoPP*, 2013.

[5] Ulysse Beaugnon, Antoine Pouille, Marc Pouzet, Jacques Pienaar, Albert Cohen. Optimization Space Pruning without Regrets. *ASPLOS*, 2017.

[6] Phitchaya Mangpo Phothilimthana, Aditya Thakur, Rastislav Bodik, Dinakar Dhurjati. Scaling up Superoptimization. *ASPLOS*, 2016.

[7] Parallel thread execution isa version 6.1.

[8] nvidia-compiler sdk. nvidia-compiler-sdk/nvvmir-samples, Mar 2014.

[9] Mark Harris. Compiling parallel languages with the nvidia compiler sdk.