Compilers Project: PTX Super-optimization

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 parallel compiler IR and covert it to super-optimal PTX. There are various challenges with this. 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.

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 and other researchers are relevant to the project: 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 is paramount to optimisation as it is be the model encoded for the purposes of super-optimisation.

2.3.2 GPUOcelot

The GPUOcelot project [8] provides a compiler framework perfectly suited for a super-optimizer to fit into.

The project primarily aims to provide a JIT-compiler for CUDA so that CUDA can be launched on the fly. However, this goal leads to utilities for a super-optimiser, especially an object-oriented IR that can be parsed by any processing layers.

A super-optimizer would be one of these intermediate passes.

3 The Program

The program was completed in python for rapid prototyping and testing. The algorithms, being extensible, should be ported to another language for speed (CUDA springs to mind due to the data-parallelism in many places) and tune-ups will be discussed as future work.

There various key modules: a parser, a semantic interpreter, and a code-generator form the core program.

3.1 The Parsing Structures and Semantics

`pyparsing` is the main library this program rests on. Assuming just the instructions, the program can read and parse the most basic arithmetic instructions, handling a few hundred.

Once the basic parsing is completed, the semantics are inferred from the program. The key structure used is an environment, a 4-tuple:

- A map from output to a condition on input.
- The set of inputs.
- The set of outputs.
- The set of all variables "seen".

Each instruction is also encoded as a parser for its arguments and a function that takes an environment and updates it to include the execution of the instruction.

Generally, if the environment read in is (M, I, O, V) and an instruction $o = f(T)$ (where T is some set of inputs) is read, the environment is updated as follows:

- $o \mapsto f(M(T))$ is added to the map (or o is updated). $M(T)$ is used in place of T to propagate equations inferred as the program is read.
- $(T \setminus (V \cup \{o\})) \cup I$. This removes all variables assigned to before since they cannot provide input to the program. Otherwise, unseen variables and past inputs are considered inputs.
- $(O \setminus T) \cup \{o\}$. We remove the inputs to this instruction from the output set of the program since we assume that all outputs will not be re-used. As a workaround, idempotent instructions may be used to produce the output that is re-used. The addition of a simple `add.u32 actual_out, temporary_to_be_reused, 0;` would suffice.
- $V \cup T \cup \{o\}$. We maintain this for the second update above: it is indispensable to know that variables have actually been seen in order to determine which variables are new inputs.

To clarify, here is an example:

```
add.u32 c, a, b;
sub.u32 d, c, b;
add.u32 e, f, g;
```

This means that the unsigned integer variables `a` through `f` are computed as follows:

```
c = a + b;
d = c - b;
e = f + g;
```

It is clear that `d = a` and `c` is not really an output, more a temporary variable, seen through the lens of the third update rule above.

This program is parsed into the following 4-tuple:

$$\begin{aligned} (e \mapsto f + g, d \mapsto a + b - b), \\ \{a, b, c, f, g\}, \\ \{d, e\}, \\ \{a \text{ through } f\} \end{aligned}$$

The current implementation uses the z3 solver and encodes the above in bit-vector theory to capture the nature of the inputs.

3.2 Naive Code-generation

In order to generate every possible program, a brute-force enumeration is produced. We assume that each instruction produces only 1 output (this suffices for all instructions parsed so far).

The first step of the brute-force approach is to try to satisfy the spec with an idempotent program that does not alter the variables. After this, all programs with at least as many instructions as the outputs provided are run through, adding instructions one at a time.

This approach is incredibly naive: in synthesizing even two instruction programs, a lot of time is taken as every two-instruction program is checked without any pruning.

We make this algorithm a little smarter by applying it to only one output at a time.

3.3 An Output-by-output Search

To search the graph of updates, we look at the program output-by-output. This does not require as many "restarts" as a naive traversal since the instructions and temporary variables synthesized for one output can be used for another.

We illustrate the algorithm by running through it on the specification from above:

$$\begin{aligned} &(e \mapsto f + g, d \mapsto a + b - b), \\ &\{a, b, c, f, g\}, \\ &\{d, e\}, \\ &\{a \text{ through } f\} \end{aligned}$$

The output-by-output search arbitrarily begins by searching through all means of generating `e` and quickly finds `add.u32 e, f, g;` finishing its search.

The search for `add.u32 d, a, 0;` terminates in the idempotent generation phase and is fairly quick.

In order to guarantee that the program produced is in fact the shortest, the algorithm must be run on all the permutations of the output variables. Inductively, we can show that the search first generates the shortest program for an output. Then given the program on n outputs, we search for the $n + 1^{st}$ output by using the specification of the program of n outputs. Hence, given an order of outputs, we can be assured that program produced is the shortest. Hence, a search over all permutations guarantees the brevity of the program.

3.4 Disjoint Outputs: an Optimization

We call two outputs disjoint if and only if the production of one does not aid that of the other. The negation of this relation: two outputs being non-disjoint if the production of one helps the production of the other is an equivalence relation on the set of outputs. Hence, we can take equivalence classes of the set of outputs. Within an equivalence class, the permutation of outputs produced may be critical, but by definition, the order in which each class is synthesized does not matter.

We determine disjointness quickly: two outputs are disjoint if and only if the program that produces them in one order is the same length as that that produces them in the other order.

Hence, in o^2 time, we can know the equivalence classes for o outputs. This leads to fewer permutations being checked in the synthesis of the entire program.

The example above is composed of two disjoint outputs, $\{d, e\}$, so the program can process them separately to good effect.

4 Benchmarks, Observations, and Evaluation

One instruction processing and the processing of identities is not only fast but rather pleasant to see working.

From observation, it appears that multiplication is the slowest and most difficult to process instruction. This may be due to the semantics of the encoding. For a `mul.lo.u32 d, x, y;` for instance, the following SAT query is produced: `Extract(0 to 32, Zero-Extend(x, 64), Zero-Extend(y, 64)) == d`. This query may be much slower to process. The test case "long-mul" is an example of this.

The program "many-classes" demonstrates how the classes optimization is, indeed, an optimization with palpable speed-up.

The program "id" demonstrates the identities: so that addition with 0, subtraction and equivalent redundancies are eliminated.

`xor`, `and`, and `or` are supported as demonstrated by "xor-and-or".

"paper-example" is the example above (so that the contents mentioned here can be verified).

4.1 Relating to the Problem Definition

The problem this work intended to solve was to provide a super-optimizer for PTX.

This is certainly a start, but by no means a definitive end. There is much still left to be learned. In fact, the heuristic currently used is naive and may be futile. The nature of PTX is not considered: only the length of the assembly. This would certainly correlate with speed intuitively, but there is no verification of this claim. In fact, confounding variables such as memory arrangement, the loops, and even instructions themselves are not considered.

4.2 Takeaways

There were two takeaways: that alternatives should be considered sooner and that the search space always pruned more.

The Ocelot project [8] showed that though a project may seem promising, its maintenance may not exist. Building the project proved to be impossible as the versions only aligned when the project was built and that astronomical event has passed. In the future, less time should be spent behind one project, especially if it does not build.

The intractability of the naive search sprang up surprisingly quickly: even generating 2 instruction programs proves to be a 2 minute chore. The output-by-output search definitely offers a large speed-up. It is interesting that some of the insights that help are rather simple.

5 Future Work

This is organized by the general facet of the program that would be improved. The last subsections include other general comments and ideas, the first subsection focussing on implementation, the second on theory, and the final on other benchmarks.

5.1 Parsing

Currently, most of PTX is not even parsed. This is since straight-line code is the only facet supported.

A future extension would not only support more instructions but also more of the structures in PTX. Conditional guards, branches, calls, directives, and special identifiers are either not supported or ignored.

This limits capabilities in memory optimization. [5] suggests the advantages of also correctly laying memory out.

Furthermore, loops are a key component of CUDA's usage, hence this program is heavily hindered in its usefulness by not treating loops and conditional structures. This is discussed more in the next sections.

5.2 Semantics

The semantics are lacking in two key ways: type information is not fully stored for any instruction and the "environment" structure discussed above is inflexible in certain cases.

The lack of typing information makes it difficult to handle instructions such as `mul.wide`, where the input and output types vary. Branching instructions incur such troubles as well as a "predicate" type would have to be included.

Furthermore, the environment structure would need to be reasoned about further in order to handle outputs being re-used. This information may have to be included in post-conditions.

Additionally, for branching, remembering environments may aid the reading and understanding of a program. This may also allow a sort of "divide-and-conquer" approach, though programs so generated are optimal given the division of the input, so all divisions would have to be considered (much like how all permutations are currently considered).

The program can also expand by modelling relevant parts in the theory of arrays or floating-point theory.

Additionally, the current implementation does not handle constants. In theory, this is not a huge hinderance: the optimizer may as well treat them as extra inputs, especially since pre-conditions and post-conditions are not handled. An extension would not only include constants, but use them to infer stronger conditions on the program.

5.3 Implementation

The implementation is in python. Other than documentation and obvious refactoring, the implementation should be ported for usability.

Python is not only notoriously slow, but is harder to integrate in the NVCC framework. Hence, compiling with the super-optimiser could be made more streamlined. A framework such as [8] the Ocelot project should be considered.

Furthermore, for runtime improvements, other programming languages should be considered. In fact, the data-parallelism suggests that CUDA should be looked into as a language to implement the program in. The output-by-output algorithm could see permutations in parallel and a faster SAT-solver may be usable.

5.4 Theoretical

Currently, very few ideas from [6] are used since it is unclear how to apply the graph search. The output-by-output algorithm is a step towards some of the LENS algorithm, however, backward generation and the pruning of prior programs are not used. It is unclear how to use them.

In addition, by finding windows with tighter pre and post-conditions, the program may hope 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 would be a key extension on the original program as it is a limitation of the current implementation.

5.5 More 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

With the above testing, we may be able to verify whether our heuristic of program length is admissible. This would be another main area of future work where benchmarks would be invaluable.

6 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] a dynamic compilation framework for gpu computing.