SMT-Based Compiler Support for Memory Access Optimization for Data-Parallel Languages

Marek Košta

Abstract. Data-parallel languages like OpenCL and CUDA are an important means to exploit the computational power of today's computing devices. We consider the compilation of such languages for CPUs with SIMD instruction sets. To generate efficient code, one wants to statically decide whether or not certain memory operations access consecutive addresses. Previous work formalized the notion of consecutivity and algorithmically reduced the static decision to satisfiability problems in Presburger Arithmetic, which can be solved using an SMT solver. In this note we describe a novel parallelism-capable library, which provides functions to fully automatize the proposed reduction and SMT solving. We describe how the library analyzes the output of the SMT solver and uses it to compute the final result. We report on experiments, which show how parallel computations perform in our context. The library is intended to be used within an OpenCL compiler.

1. Introduction

Data-parallel languages like OpenCL and CUDA are ubiquitous in today's computing landscape. They follow the so-called SPMD (Single Program, Multiple Data) paradigm, where the technical details of parallelization are abstracted away: The programmer writes a scalar function, called the *kernel*. The kernel is executed in multiple *work items* (sometimes ambiguously called *threads*) by a runtime system. To make every work item perform an individual task, e.g. writing to different elements of an array, special primitives are built into the language to query the *ID* of a work item.

Due to these semantics, the runtime system may choose to execute work items in parallel. On GPUs, this boils down to scheduling each work item to one of the hardware-managed threads via the device driver. On CPUs, the same scheme can be used by employing well-known libraries like pthreads, OpenMP, or MPI to exploit all available cores. In addition, today's CPUs offer another level of parallelism *per core* in the form of SIMD instructions. These are instructions that perform the same operation on multiple input values at once (Single Instruction, Multiple Data). The number of multiple input values, which can be processed by SIMD instructions of a SIMD CPU is called the *SIMD width* of the SIMD CPU.

The historical development of data-parallel languages stemming from GPUs plays a crucial role when compiling them for a SIMD CPU: On the CPU, one has to emulate dynamic features that on GPUs are implemented in hardware. The interesting feature for this note is that GPUs determine at runtime whether or not all work items access memory at *consecutive* addresses. In the positive case, a faster operation is issued. This behavior can be emulated by a CPU compiler that exploits SIMD instructions. To this end, it would introduce code that does the same runtime check, but the cost of this usually outweighs the performance gain. Thus, static analysis has been used to prove at compile time that a memory operation *always* accesses consecutive addresses [3]. This approach generally covers fewer cases than a dynamic runtime check but is still applicable often enough to be of interest. The main drawback of this approach is that it can only handle very simple address computations such

M. Kosta and T. Sturm, eds. Proceedings of the MACIS 2013, Nanning, China, December 11-13, 2013.

as linear translations by constants. A recent publication [5] generalizes this to more interesting linear arithmetic transformations, in particular including integer division and modulo by constants.

The main idea of [5] is to convert the central question "Do consecutive work items access consecutive memory addresses or not?" to a finite set of satisfiability problems in Presburger Arithmetic and to solve them using an SMT solver. This theoretical framework was accompanied by prototypical implementations, which employed several independent software components:

1. Redlog within the computer algebra system Reduce [2],
2. the SMT solver Z3 [1],
3. the WFV OpenCL compiler [4] for code generation.

These components had been combined manually using files as an interface from Reduce to Z3. The output files of Z3 were essentially huge tables mapping possible parameter values of the kernel considered for compilation to truth values stating whether or not the answer to the above consecutivity question is affirmative for the respective parameter value. These tables had to be analyzed using human intelligence in order to obtain concise mathematical descriptions suitable for injection into the WFV OpenCL compiler.

The present note describes work in progress to reduce the heterogeneous environment described above to a monolithic library, which is linked against Z3 and which can in turn be linked against the WFV OpenCL compiler. As a theoretical novelty, this comprises an automatic analysis of the Z3 output resulting in a data structure that can be directly used by the WFV OpenCL compiler.

In Section 2 we summarize the relevant results of the previous publication [5]. Sections 3 and 4 contain the new contributions of this note. Finally, in Section 5 we outline the next steps for this ongoing project.

2. A Summary of Previous Results

The static optimization for a single memory access operation proposed in [5] essentially consists of the following steps:

1. For the considered memory access operation occurring in the OpenCL kernel being compiled, extract the memory access term $e(x, a)$. Here x is a variable corresponding to a work item ID and a is a parameter of the kernel.
2. Using $e(x, a)$ and the SIMD width w of the target CPU, construct a formula $\varphi(a)$ formalizing the consecutivity question for this memory access operation.
3. Instantiate the parameter a in $\varphi(a)$ with all integers from an interval $[l; u]$, obtaining $u - l + 1$ closed formulas.
4. Decide which of these formulas hold.
5. Generate code, which executes a faster memory access operation for those values of parameter a for which $\varphi(a)$ holds.

The constructed formula $\varphi(a)$ formalizes the *consecutivity question* for the given memory access operation in the following sense: If $\varphi(a)$ is true for a particular value of a, then all work items access memory at consecutive addresses for this value of a and a faster memory access operation can be executed. The term $e(x, a)$ is a *Presburger Arithmetic* term, possibly containing modulo and division by constants, denoted by mod_k and div_k, respectively.

Presburger Arithmetic originally refers to the first-order theory of the integers over a countably infinite language $\mathcal{L}$ comprising 0, 1, $+$, $-$, $<$, and infinitely many congruences $\equiv_k$ for $k \in \mathbb{N} \setminus \{0\}$. This, of course, allows to tacitly use also $\leq$, $\neq$, $>$, $\geq$. For this setting, Presburger proved completeness by giving a decision procedure [6]. His decision procedure was based on effective quantifier elimination in combination with the fact that variable-free atomic formulas can be effectively evaluated to either "true" or "false." As a consequence of using quantifier elimination, Presburger required $\equiv_k$ to be a formal part of the language.

In the context of programming languages, Presburger's congruence relations have a counterpart in the binary modulo function, which is naturally paired with integer division. For fixed integer second argument $k \in \mathbb{N} \setminus \{0\}$, both can be encoded in $\mathcal{L}$, e.g., as follows:

$$\begin{aligned} \mathbb{Z} &\models \mathrm{mod}_k(x) = y \longleftrightarrow 0 \leq y \leq k-1 \wedge x \equiv_k y, \\ \mathbb{Z} &\models \mathrm{div}_k(x) = y \longleftrightarrow k \odot y \leq x < k \odot (y+1). \end{aligned}$$

These defining formulas can be generalized to $k \in \mathbb{Z} \setminus \{0\}$ and can be used to systematically translate formulas containing mod_k and div_k to the original Presburger language $\mathcal{L}$. This requires, in general, the introduction of quantifiers with variables that represent sub-terms. Similarly, for decision procedures not based on quantifier elimination, the congruences can be eliminated:

$$\mathbb{Z} \models t \equiv_k 0 \longleftrightarrow \exists x(k \odot x = t).$$

Note that admitting arbitrary terms as second arguments in modulo operations or integer division would lead to an undecidable theory:

$$\mathbb{Z} \models s \neq 0 \longrightarrow t \bmod s = t - (t/s)s, \quad \mathbb{Z} \models s \mid t \longleftrightarrow t \bmod s = 0,$$

and $(\mathbb{Z}, 0, 1, +, -, |)$ is known to be undecidable [7].

There is a variety of decision procedures and complexity results available for Presburger Arithmetic [8, and the references given there]. However, in our case only existential formulas need to be decided for validity. The reason is that the formula $\varphi(a)$ constructed using $e(x, a)$ is a universal formula containing a single free variable a, regardless of the memory access term $e(x, a)$ [5, Section 3]. Systematically using the equivalences for div_k and mod_k above, one can equivalently remove all occurrences of these operations. Therefore, it is in principle possible to use an SMT solver to decide $\neg\varphi(a)$ for a concrete integer a. In practice, a straightforward use of an SMT solver led to unacceptable running times. To improve the performance of an SMT solver, the *modulo elimination* as a preprocessing step was introduced [5, Section 4].

3. New Results

The approach described in Section 2 allows to take advantage of parallelism in the following natural way: Construct $\varphi(a)$, divide the interval $[l; u]$ to n subintervals of approximately same sizes, and do the instantiation together with SMT solving concurrently using n computing entities. A partial result consists of those values of a, for which $\varphi(a)$ is satisfiable in the respective subinterval. Finally, collect and merge the n partial results to obtain the final result, answering the consecutivity question for all values of a in the given interval $[l; u]$.

There are two main ways how to achieve concurrency: processes and threads. The reasons for using threads are the following: Thread creation and communication between threads is much faster than their process counterparts. Moreover, the existing pthreads API, which provides primitives for all thread-related tasks, is a well-established and commonly supported standard. On the other hand, APIs for process management differ between different platforms. This makes portability a difficult task when processes are used.

Recall that for decision, which values of the parameter a yield a satisfiable $\varphi(a)$, an SMT solver with support for linear integer arithmetic can be used. There are numerous different SMT solver libraries with this support available. However, the following reasons make use of the Z3 SMT solver library [1] especially appealing. First, Z3 library directly accepts mod_k and div_k in the input formula. Second, Z3 library is thread-safe and uses a concept called *context*. Context is a data structure, which has to be created before any formulas can be created, manipulated, or solved. There can be a number of context instances in a program using threads. This can be exploited in a threaded application in the following way: Each of the n working threads maintains its own context and carries out instantiations and satisfiability testing exclusively within its context. After a working thread finishes its work, it destroys its context and returns a partial result to the main thread. The main thread merges all partial results to the final result.

To represent partial and final results, we propose an approach based on *periodic sets*. Mathematically, a periodic set is $\{x \mid x \in \mathbb{Z} \wedge a \leq x \leq b \wedge x \equiv_m c\}$. It is represented by four integers: a, b, m, and c. Note that a set containing only one integer is periodic and that union of two periodic sets is not necessarily periodic. However, any finite set of integers can be represented as a sorted list of finitely many disjoint periodic sets. We refer to this as the PS *list representation.*

Main advantages of the PS list representation are: First, the representation is succinct when the represented set is (almost) periodic. Second, given a set in the PS list representation, logical condition, which is true only for numbers in the set can be easily extracted: It is a conjunction of conditions for all periodic sets in the list. Finally, the merge operation (union) of two disjoint sets given in this representation can be done in constant time. Since sets are represented as sorted lists of periodic sets, the merge operation boils down to list concatenation together with a simple check: First, check whether union of two periodic sets (the last one of the first list and the first one of the second list) is a periodic set. If this is the case, merge them before the lists are concatenated. Note that the merge operation is the only operation, which is needed for the above described parallel strategy to work. Merge of two disjoint periodic sets can be done in constant time by comparing moduli and bounds of the sets to be merged.

We implemented the parallel strategy together with the PS list representation described above, creating a new library. The library is implemented in C. It uses pthreads API and is linked against Z3 library. We tested it on Mac OS X and Linux platforms, but the library should be portable to any platform where a C compiler is available. A short description of the function provided by the library can be found in Section 4.

Since the input interval is divided into n disjoint subintervals of approximately the same size, the maximal theoretical speedup is of factor n. In practice, the computing times for different parts can differ, which leads to a smaller speedup factor. Our experiments, which we are going to describe, show that in practice it is possible to come fairly close to the maximal theoretical speedup.

The library was tested on some problems from [5]. The following unexpected behavior was observed across all platforms we tested the library on: Using n working threads the overall user time of a computation was more than n-times higher than the overall user time of the same computation using only one working thread. The real time of the computation was around n-times smaller than the user time, i.e. we observed a speedup of factor close to the maximal theoretical speedup of factor n, but this unreasonably high user time led to no real time speedups or even led to slowdowns in comparison to one working thread. It seems that the user time blow-up is caused by some thread-safeness ensuring mechanisms inside the Z3 library, but the real cause of this remains unclear. There is an intensive effort to analyze and solve this issue. We are in contact with Z3 developers in this matter.

This problem led to the following temporary workaround solution using processes: After a working thread is started, it creates a new process, which executes the actual computations. The working thread only waits for the process and collects its result afterwards. On the one hand, the workaround solution resolved the problem of the rapid user time blow-up. The reason for this seems to be that the concurrent computations are done by independent single-threaded processes. On the other hand, overhead caused by the use of processes is introduced.

The experiments with both implementations were conducted on a 64 core 2.4 GHz Intel Xeon E5-4640 running 64 bit Debian Linux version 6.0.7. All experiments use the modulo elimination as a preprocessing step.

Table 1 shows the experiments with our implementation using threads. The memory access term in this case comes from the Fast Walsh Transform (FWT) benchmark [5]. The actual problem solved is the same in all five cases. Therefore, the same resulting set was obtained. The Threads column shows the number of working threads used to solve the respective problem. The Speedup column shows the speedup factor, i.e. the user time divided by the real time. This table documents the behavior we described above: The speedup factor is close to the maximal theoretical speedup factor (number of threads), but the user time is unreasonably high.

Table 1. Running times and speedups of the implementation using threads applied to the following input: Memory access term $e(x, a) = \mathrm{mod}_a(x) + 2a \odot \mathrm{div}_a(x)$, $a \in \{1, \ldots, 2^{16}\}$, and SIMD width $w = 4$.

User Time Blow-up using Threads

Threads	Result	User Time	Real Time	Speedup
1	$4 \leq a \leq 2^{16} \wedge a \equiv_4 0$	21m 43s	21m 46s	1.0×
4	— " —	26m 41s	7m 44s	3.5×
8	— " —	79m 33s	11m 23s	7.0×
32	— " —	730m 36s	24m 35s	29.7×
64	— " —	1587m 8s	25m 41s	61.8×

Table 2. Running times and speedups of the workaround solution using processes applied to the following input: memory access term $e(x, a) = \mathrm{mod}_a(x) + 2a \odot \mathrm{div}_a(x)$, $a \in \{1, \ldots, 2^{16}\}$, and different SIMD widths: $w = 4$, $w = 8$, and $w = 16$.

Performance of the Workaround Solution on FWT Benchmarks

Processes	w	Result	User Time	Real Time	Speedup
1	4	$4 \leq a \leq 2^{16} \wedge a \equiv_4 0$	13m 40s	13m 42s	1.0×
4	4	— " —	12m 11s	3m 07s	3.9×
8	4	— " —	11m 28s	1m 34s	7.3×
32	4	— " —	11m 02s	22s	30.1×
64	4	— " —	18m 48s	19s	59.4×
1	8	$8 \leq a \leq 2^{16} \wedge a \equiv_8 0$	22m 17s	22m 22s	1.0×
4	8	— " —	20m 05s	5m 11s	3.9×
8	8	— " —	19m 08s	2m 37s	7.3×
32	8	— " —	18m 16s	40s	27.4×
64	8	— " —	30m 50s	31s	59.7×
1	16	$16 \leq a \leq 2^{16} \wedge a \equiv_{16} 0$	38m 23s	38m 29s	1.0×
4	16	— " —	35m 13s	9m 11s	3.8×
8	16	— " —	35m 14s	4m 48s	7.3×
32	16	— " —	33m 27s	1m 14s	27.1×
64	16	— " —	56m 02s	1m 4s	52.5×

Table 2 shows the experiments we conducted with the workaround implementation using processes. The memory access term comes in this case from the FWT benchmark as well. In this case we solved the problems for different SIMD widths w. The Process column shows the number of processes created by working threads. User Time, Real Time, and Speedup columns have the same meaning as in Table 1.

Despite the overhead introduced by process creation and inter-process communication, Table 2 demonstrates that a speedup of factor fairly close to n can be achieved when n working processes are used. Note, however, the relatively big overhead introduced when 64 processes were used.

Comparing the running times of our current implementation using processes with the running times from [5] does not make much sense because of the following reasons: The times in [5] are running times of the Z3 SMT solver only. They do not include the time for writing files and formula construction done in Redlog. Moreover, our implementation does much more: In [5] only a list of parameter values was returned. Our implementation returns a set in the PS list representation, which is succinct and can be directly used further in the code generation process.

4. Short API Description

Now we briefly describe the API provided by the library we described in Section 3.

The three most important functions of the API are: `parse`, `elim`, and `compute`. Function `parse` parses a given SMT-LIB format string describing the memory access term and returns it as an instance of Z3_AST data structure. Z3_AST (abstract syntax tree) is the most fundamental Z3 recursive data structure used to represent formulas and terms as trees. Modulo elimination [5, Section 4] on a Z3_AST term e is realized by `elim`, which returns a Z3_AST term without the modulo operator equivalent to e. The function `compute` carries out the actual computation. Parameters modifying the actual computation are provided as arguments: a memory access term as Z3_AST, the number of threads to be used, and an interval in which the consecutivity question will be decided. Result of `compute` is a set in the PS list representation.

5. Future Directions

We developed a novel library, which can automatically solve the problems coming from the memory access optimization for data-parallel languages described in [5]. To represent the output we came up with the PS list representation, which can be easily used further in the compilation process. The library described in the present note is still a work in progress. The main issue, which has to be resolved is the unreasonable increase of the user time when threads are used.

The main future work is the integration of the library described in the present note with an OpenCL compiler. Another interesting direction is representation of the partial and final results. Experiments with a few arbitrarily chosen memory access terms showed that the output was always a periodic set. This is the case for all computations in this note as well. Therefore, it would be interesting to investigate if an even more efficient representation than the PS list representation could be possibly used. Finally, we would like to try different SMT solvers and compare their performance.

Acknowledgements

This research was supported by the German Transregional Collaborative Research Center SFB/TR 14 AVACS.

References

[1] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340. Springer, 2008.

[2] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.

[3] Ralf Karrenberg and Sebastian Hack. Whole function vectorization. In *CGO*, pages 141–150, 2011.

[4] Ralf Karrenberg and Sebastian Hack. Improving Performance of OpenCL on CPUs. In *Compiler Construction*, volume 7210 of *LNCS*, pages 1–20. Springer, 2012.

[5] Ralf Karrenberg, Marek Košta, and Thomas Sturm. Presburger arithmetic in memory access optimization for data-parallel languages. In Pascal Fontaine, Christophe Ringeissen, and Renate A. Schmidt, editors, *Frontiers of Combining Systems*, volume 8152 of *LNAI*, pages 56–70. Springer, 2013.

[6] Mojzesz Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du premier congres de Mathematiciens des Pays Slaves*, pages 92–101, Warsaw, Poland, 1929.

[7] Julia Robinson. Definability and decision problems in arithmetic. *J. Symb. Log.*, 14(2):98–114, 1949.

[8] Volker Weispfenning. The complexity of almost linear Diophantine problems. *Journal of Symbolic Computation*, 10(5):395–403, November 1990.

Marek Košta
Max Planck Institute for Informatics
Campus E1 4
66123 Saarbrücken, Germany
e-mail: mkosta@mpi-inf.mpg.de