Presburger Arithmetic in Memory Access Optimization for Data-Parallel Languages

Ralf Karrenberg[1], Marek Košta[2], and Thomas Sturm[2]

[1] Saarland University, 66123 Saarbrücken, Germany
karrenberg@cs.uni-saarland.de
[2] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany
{mkosta,sturm}@mpi-inf.mpg.de

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. We formalize the notion of consecutivity and algorithmically reduce the static decision to satisfiability problems in Presburger Arithmetic. We introduce a preprocessing technique on these SMT problems, which makes it feasible to apply an off-the-shelf SMT solver. We show that a prototypical OpenCL CPU driver based on our approach generates more efficient code than any other state-of-the-art driver.

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). This saves both execution time and power consumption. Fig. 1 depicts variants of how the runtime system could choose to execute a 2-dimensional grid of work items.

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

P. Fontaine, C. Ringeissen, and R.A. Schmidt (Eds.): FroCoS 2013, LNAI 8152, pp. 56–70, 2013.

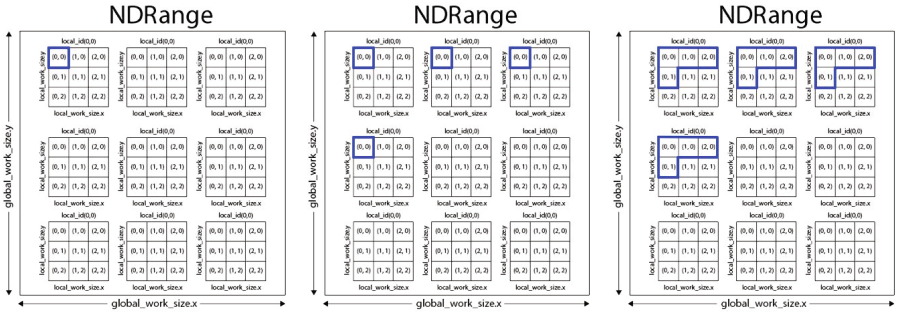

Fig. 1. The OpenCL execution model and examples how a driver could choose to execute work items in parallel (work items marked blue). Left: sequential execution. Middle: multi-threaded execution (4 cores). Right: multi-threaded execution with "Whole-Function Vectorization" (4 cores, SIMD width 4, details in Sect. 2).[1]

to emulate dynamic features that on GPUs are implemented in hardware. The interesting feature for this paper 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 [10]. This approach generally covers fewer cases than a dynamic runtime check but is still applicable often enough to be of interest. Until now, however, it could only handle very simple address computations such as linear translations by constants. In this paper, we generalize this to more interesting linear arithmetic transformations, in particular including integer division and modulo by constants. Our approach can—to a certain degree—also handle non-constant inputs. Our key idea 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*.

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 [16]. 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.

[1] Graphics modified from the official Khronos OpenCL specification.

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:

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

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 [17].

There is a variety of decision procedures and complexity results available for Presburger Arithmetic [20, and the references given there]. Our input considered here is limited to the existential fragment, for which SMT solvers, in spite of their possible incompleteness, are an interesting choice. For our practical computations we chose Z3 [5], which has the advantage to directly accept mod_k and div_k in the input.

The original contributions of this paper are the following:

1. We formalize in Presburger Arithmetic the notion of consecutivity and all relevant conditions that have to be decided for static optimization of memory operations.
2. Our formalization allows to consider address computations that involve div_k and mod_k and limited occurrences of non-constant inputs.
3. We introduce *modulo elimination* as a general preprocessing technique for Presburger terms. This makes it feasible to decide our formalizations using an off-the-shelf SMT solver.
4. The feasibility of our approach is documented by comprehensive, systematic computations.
5. Our computations establish new benchmarks based on current research problems in compiler construction. In this capacity, they are of general interest for the SMT community.
6. We show that a prototypical OpenCL CPU driver based on our approach generates more efficient code than any other state-of-the-art driver, including Intel and AMD.

```
__kernel void
simple(float* in,
       float* out) {
  int tid = get_global_id();
  out[tid] = in[tid];
}
```

```
__kernel void
fastWalshTransform(float* tArray,
                   int    step) {
  int tid   = get_global_id();
  int group = tid % step;
  int pair  = 2*step*(tid/step)
              + group;
  int match = pair + step;
  float T1      = tArray[pair];
  float T2      = tArray[match];
  tArray[pair]  = T1 + T2;
  tArray[match] = T1 - T2;
}
```

Fig. 2. OpenCL kernels with simple (left) and complex (right) memory address computations: `tid`, `pair`, and `match`. The function `get_global_id()` returns the work item ID, which allows each work item to access different array positions.

The structure of the paper is as follows: In Sect. 2, we summarize the relevant notions from data-parallel languages and compilers and make precise the problem addressed in this paper. In Sect. 3, we formalize this problem as a set of Presburger satisfiability problems and perform first computational experiments. Sect. 4 details how modulo elimination significantly improves performance of the SMT solver. In Sect. 5, we discuss possibilities for code generation. Sect. 6 experimentally evaluates the achieved performance gain. Sect. 7 discusses related work. In Sect. 8, we summarize our results and discuss possible future work.

2 Memory Access Optimization for Data-Parallel Languages

We want the compiler of a language like CUDA or OpenCL to prove for as many memory access operations as possible that the addresses that are accessed by consecutive work items are contiguous in memory. If this property can be proven, CPU compilers for these languages can generate faster code. Recall from the introduction that our target architecture are CPUs with SIMD instruction sets, to which we are going to simply refer to as CPUs. We generally consider load and store operations, for which array accesses are the most prominent example.

Consider the two OpenCL kernels in Fig. 2. The kernel on the right-hand side, `fastWalshTransform`, is taken from the AMD APP SDK v2.8.[2] In this code, the array accesses depend on the value `tid` obtained from calls to `get_global_id()`. These calls return different values for different work items and consecutive values for consecutive work items. It is easy to see that the `simple` kernel always accesses contiguous memory locations due to the direct

[2] `developer.amd.com/sdks/AMDAPPSDK`

```
__kernel void
fastWalshTransform(float* tArray,
                   int    step)
{
  int tid    = get_global_id();
  if (tid % 2 != 0) return;
  int2 tidV  = (int2)(tid,tid+1);
  int2 stepV = step;
  int2 group = tidV % stepV;
  int2 pair  = 2*step*(tidV/stepV)
               + group;
  int2 match = pair + stepV;
  float2 T1  = (float2)(tArray[pair.x],
                        tArray[pair.y]);
  float2 T2  = (float2)(tArray[match.x],
                        tArray[match.y]);
  float2 X   = T1 + T2;
  float2 Y   = T1 - T2;
  tArray[pair.x]  = X.x;
  tArray[pair.y]  = X.y;
  tArray[match.x] = Y.x;
  tArray[match.y] = Y.y;
}
```

```
__kernel void
fastWalshTransform(float* tArray,
                   int    step)
{
  if (step <= 0 || step % 2 != 0) {
    // Omitted code:
    // Execute kernel as on the left.
    return;
  }
  int tid   = get_global_id();
  if (tid % 2 != 0) return;
  int group = tid % step;
  int pair  = 2*step*(tid/step)
              + group;
  int match = pair + step;
  float2 T1 = *((float2*)(tArray+pair));
  float2 T2 = *((float2*)(tArray+match));
  *((float2*)(tArray+pair))  = T1 + T2;
  *((float2*)(tArray+match)) = T1 - T2;
}
```

Fig. 3. Result of WFV manually applied at source level to the `fastWalshTransform` kernel of Fig. 2 ($w = 2$). Left: Conservative WFV requires sequential execution. Right: WFV with our approach proves consecutivity of the memory addresses for certain values of `step`, which allows to generate a variant with more efficient code.

use of `tid`. In contrast, the access pattern of `fastWalshTransform` is not obvious. Experimentally, one would observe that, depending on `step`, there is a considerable number of accesses that actually are consecutive. At this point, it is important to understand a fundamental difference between GPUs and CPUs: In GPUs, there is dedicated hardware to dynamically *coalesce* to a single access all memory accesses of work items running in parallel whenever possible. This yields significant speedup by preventing sequential execution of the accesses. On CPUs, there is no such hardware support. Therefore, without compiler optimization, the memory operations considered here would be executed sequentially.

Modern compilers apply a technique called *Whole-Function Vectorization* (WFV) [10]. WFV transforms a kernel to execute w work items at once, where w is usually the *SIMD width*. The SIMD width of a CPU is the number of single-precision values that fit into a vector register. Typical values for w are 4 for the SSE, AltiVec, and NEON instruction sets, 8 for AVX, or 16 for LRBni. An interesting recent development is AMDs introduction of the Sea Islands series of GPUs which have a vector instruction set with a SIMD width of 64. In the context of WFV, a vectorized kernel executes w work items at once for every single hardware thread. The values of each work item are kept in one cell of a vector register instead of a scalar register. Thus, WFV can increase performance of applications by a factor as large as w.

Unfortunately, WFV has drawbacks which can significantly reduce this theoretical speedup or even result in slowdowns [11]. In this paper, we focus on one

specific drawback, which arises in presence of memory access operations: In order to have the kernel compute w work items at once, accesses to `tid` are transformed to return a vector of w consecutive values. Accordingly, each dependent operation has to be transformed into its vector counterpart, e.g. a scalar addition becomes a vector addition. Unfortunately, the vector counterparts for memory operations available in most of today's SIMD instruction sets only support accessing consecutive addresses. Therefore, if the addresses are non-consecutive, w sequential operations have to be used, reducing the overall gain of WFV compared to the theoretical factor w. This is exemplified by the left-hand side kernel in Fig. 3, where `tidV` is the above-mentioned, vector-valued `tid`. To make use of a vector load or store, the compiler has to automatically prove that the address computation will never produce non-consecutive values.[3] However, the expression tree that corresponds to this address computation may consist of arbitrary code. This will, in general, lead to undecidable problems. Current approaches are limited to expressions with linear translations by constants [10]. Our new approach extends the class of expressions that can be analyzed to Presburger Arithmetic and functions definable therein. This covers in particular integer division and modulo operations by constants as well as certain occurrences of input variables.

To this end, our compiler translates the expression tree that yields the memory address to a term that depends on `tid` and a possible input parameter. For example, the address of the second load operation of the FastWalshTransform kernel in Fig. 2 is given by `tArray[match]`, where the term obtained for `match` is

```
2*step * (tid / step) + (tid % step) + step.        (1)
```

Notice that `step` is an input value that is constant for all work items during one execution of the kernel.

3 Translation to Presburger Arithmetic

We are now going to switch to a more mathematical notation: The variable t is going to denote the `tid` and a is going to denote the input. For integer division and modulo, we introduce unary functions div_k and mod_k for $k \in \mathbb{Z} \setminus \{0\}$, which emphasizes the fact that the divisors and moduli are limited to numbers. For our example term (1), we obtain

$$e(t, a) = 2a \odot \mathrm{div}_a(t) + \mathrm{mod}_a(t) + a. \tag{2}$$

At this point, let us give the precise definitions of mod_k and div_k:

$$x = k \odot \mathrm{div}_k(x) + \mathrm{mod}_k(x), \quad \text{where} \quad |\mathrm{mod}_k(x)| < |k|. \tag{3}$$

[3] In addition, a store must be proven to be always executed by all work items to not produce false side effects. To keep things simple, we consider this out of the scope of this paper.

It is well-known that this definition does not uniquely specify $\mathrm{div}_k(x)$ and $\mathrm{mod}_k(x)$. SMT-LIB Version 2 resolves this issue by making the convention that $\mathrm{mod}_k(x) \geq 0$.[4] As long as both k and x are non-negative, common programming languages agree with this convention. However, when negative numbers are involved, OpenCL follows the C99 standard, which in contrast to SMT-LIB requires that $\mathrm{sign}(\mathrm{mod}_k(x)) = \mathrm{sign}(x)$. In our setting, we observe that the arguments of mod_k generally are positive expressions involving the `tid` such that both conventions happen to coincide.

Let us analyze a single memory access with respect to the following *consecutivity question*: "Do w consecutive work items access consecutive memory addresses when doing this memory access or not?" Using the corresponding term $e(t, a)$, the following equation holds if and only if the work items t and $t + 1$ access consecutive memory locations for input a:

$$e(t, a) + 1 = e(t + 1, a).$$

The following conjunction generalizes this equation to w consecutive work items t, $\ldots$, $t + w - 1$:

$$\bigwedge_{i=0}^{w-2} e(t + i, a) + 1 = e(t + i + 1, a).$$

Recall from the previous section that these groups of w work items naturally start at 0 so that only conjunctions are relevant where t is divisible by w. The following Presburger formula formally adds this constraint:

$$\varphi(w, a) = \forall t \Big(t \geq 0 \wedge t \equiv_w 0 \longrightarrow \bigwedge_{i=0}^{w-2} e(t + i, a) + 1 = e(t + i + 1, a) \Big).$$

For given $w \in \mathbb{N}$ and $\alpha, \beta \in \mathbb{Z}$ with $\alpha \leq \beta - 1$, the answer to our consecutivity question for w and $a \in \{\alpha, \ldots, \beta - 1\}$ is given by the set

$$A_{w,\alpha,\beta} = \{\, a \in \mathbb{Z} \mid \mathbb{Z} \models \varphi(w, a) \wedge \alpha \leq a < \beta \,\}.$$

We essentially compute $A_{w,\alpha,\beta}$ by at most $(w - 1)(\beta - \alpha - 1)$ many applications of an SMT solver to the $w - 1$ disjuncts of $\neg\varphi(w, a)$ for $a \in \{\alpha, \ldots, \beta - 1\}$, where

$$\neg\varphi(w, a) = \bigvee_{i=0}^{w-2} \exists t \big(t \geq 0 \wedge t \equiv_w 0 \wedge e(t + i, a) + 1 \neq e(t + i + 1, a) \big).$$

Notice that, when obtaining "sat" for some $i \in \{0, \ldots, w - 2\}$, the remaining problems of the disjunction need not be computed.

Our answer $A_{w,\alpha,\beta}$ consists of those a for which the SMT solver yields "unsat." Note that besides "sat" or "unsat," the solver can also yield "unknown," which we treat like "sat." This underapproximation does not affect the correctness of our approach. We only miss optimization opportunities when generating code

[4] `smtlib.cs.uiowa.edu/theories/Ints.smt2`

Table 1. Running times of Z3 applied to $\neg\varphi(w, a)$ for $e(t, a)$ as in (2). In all three cases, $\alpha = 1$ and $\beta = 2^{16}$ so that $a \in \{1, \ldots, 2^{16} - 1\}$ with a time limit of one minute per call.

FastWalshTransform Problem Set using Z3					
w	Sat	Unsat	Unknown	Timeouts	CPU Time
4	16,243	48,931	0	361	14 h
8	7,694	54,510	0	3,331	97 h
16	2,773	52,468	0	10,294	256 h

later on. The same holds for possible timeouts when imposing reasonable time limits on the single solver calls. Later in Sect. 5, we are going to discuss how compact representations for $A_{w,\alpha,\beta}$ can be obtained.

Table 1 shows running times and results for the application of Z3 version 4.3.1 [5] to the consecutivity question for our FastWalshTransform kernel.[5] Alternatives to Z3 include CVC4 [1] and MathSAT5 [3]. These SMT solvers, however, do not directly support div_k and mod_k, which makes them less interesting for our application here.

Obviously, our obtained running times are too high to be of any practical interest. We suspect that the div_a and mod_a operators occurring in (2), which have to be resolved into the classical Presburger language, significantly contribute to that complexity.

4 Modulo Elimination as a Preprocessing Step

We are now going to considerably improve the running times observed in the previous section. The basic idea is to eliminate, in a preprocessing step, all occurrences of mod_k in favor of div_k. To this end, we use the definition (3) of div_k and mod_k as a rewrite rule:

$$\mathrm{mod}_k(x) \to x - k \odot \mathrm{div}_k(x).$$

The validity of this *modulo elimination* rule is not affected by the discussion after the statement of definition (3). After finitely many applications of the rule to $\neg\varphi(w, a)$, we arrive at equivalent input to the SMT solver that does not contain any modulo operations.

Applying modulo elimination to $e(t, a)$ from (2) results in $a + t + a \odot \mathrm{div}_a(t)$. Using this reduced term instead of the original one inside $\neg\varphi(w, a)$ makes a significant difference in performance for the benchmarks from Table 1. The improved results are collected in Table 2. Notice that the CPU times related by the respective speedup factors refer to different subsets of finished computations due to timeouts. Nevertheless, for practical purposes, these speedups are exactly the

[5] All our SMT computations have been performed on a 2.4 GHz Intel Xeon E5-4640 running Debian Linux 64 bit.

Table 2. Running times of Z3 applied to $\neg\varphi(w, a)$ for $e(t, a)$ obtained by applying modulo elimination to the term in (2). In all three cases, $\alpha = 1$ and $\beta = 2^{16}$ so that $a \in \{1, \ldots, 2^{16} - 1\}$ with a time limit of one minute per call.

FastWalshTransform Problem Set with Modulo Elimination using Z3

w	Sat	Unsat	Unknown	Timeouts	CPU Time	Speedup
4	16,383	49,152	0	0	4 min	210×
8	8,191	57,344	0	0	5 min	1164×
16	4,095	61,128	0	312	334 min	46×

interesting information. We have verified that our modulo elimination causes at least 50 percent of the reported speedups.

These running times are not suitable for just-in-time compilation. For offline release compilation, however, they are fine. Note that the computations can be perfectly parallelized such that the use of, say, 64 virtual cores will result in a corresponding speedup factor. We estimate that on such a system, the FastWalshTransform problem set for $w = 4$ could be computed in less than seventy hours even for $\alpha = -2^{31}$ and $\beta = 2^{31}$ covering the full range of signed 32 bit integers. In general, there are situations where an optimization for a subset of the range will result in a good tradeoff between compilation time and performance gain.

Let us understand why modulo elimination is so successful on our types of examples: The relevant equation $e(t+1, a) - e(t, a) - 1 = 0$ for FastWalshTransform with $e(t, a)$ taken from (2) is given by

$$2a \odot \mathrm{div}_a(t+1) - 2a \odot \mathrm{div}_a(t) + \mathrm{mod}_a(t+1) - \mathrm{mod}_a(t) - 1 = 0.$$

Modulo elimination yields:

$$2a \odot \mathrm{div}_a(t+1) - 2a \odot \mathrm{div}_a(t) + t + 1 - a \odot \mathrm{div}_a(t+1) - t + a \odot \mathrm{div}_a(t) - 1 = 0,$$

which simplifies to $a \odot \mathrm{div}_a(t+1) - a \odot \mathrm{div}_a(t) = 0$. Such patterns do not occur accidentally: Programmers using data-parallel languages try hard to access memory only consecutively to get the best performance. This leads to specific patterns within address computation expressions. Using particularly div_k and mod_k, consecutivity can hardly be achieved without patterns similar to the one discussed above.

To conclude this section, we are now going to discuss an additional example, for which our approach turns out to be suitable even for just-in-time compilation. This is a kernel, called `bitonicSort`, also taken from the AMD APP SDK. The expression we are interested in here is

$$e(t, a) = 2^{a+1} \odot \mathrm{div}_{2^a}(t) + \mathrm{mod}_{2^a}(t) + 2^a. \tag{4}$$

The input parameter a occurs exclusively as an exponent. This restricts the reasonable range of values to consider to $\{0, \ldots, 62\}$ on a 64 bit architecture.

Table 3. Running times of Z3 applied to $\neg\varphi(w, a)$ for $e(t, a)$ as in (4). In all three cases, $\alpha = 0$ and $\beta = 63$ so that $a \in \{0, \ldots, 62\}$ with a time limit of one minute per call.

BitonicSort Problem Set using Z3					
w	Sat	Unsat	Unknown	Timeouts	CPU Time
4	54	2	0	7	8 min
8	52	3	0	8	16 min
16	40	4	0	19	41 min

Table 4. Running times of Z3 applied to $\neg\varphi(w, a)$ for $e(t, a)$ obtained by applying modulo elimination to the term in (4). In all three cases, $\alpha = 0$ and $\beta = 63$ so that $a \in \{0, \ldots, 62\}$ with a time limit of one minute per call.

BitonicSort Problem Set with Modulo Elimination using Z3						
w	Sat	Unsat	Unknown	Timeouts	CPU Time	Speedup
4	61	2	0	0	0.7 s	686×
8	60	3	0	0	1.5 s	640×
16	59	4	0	0	3.7 s	665×

Table 4 shows the relevant running times. Noticing the similarities between (4) and (2), it is clear that modulo elimination leads to similar simplifications in the corresponding equation. The timings in Table 3 confirm this: The speedups are similar to FastWalshTransform.

5 From SMT Solving Results to Code

Recall from Sect. 3 that the answer obtained there to our consecutivity question "Do w consecutive work items access consecutive memory addresses when doing this memory access or not?" is the set $A_{w,\alpha,\beta}$ of inputs $a \in \{\alpha, \ldots, \beta - 1\}$ for which the answer is affirmative. The respective sets $A_{w,\alpha,\beta}$ for all our problem sets are collected in Table 5.

Our goal is now to produce during code generation a case distinction such that for input contained in $A_{w,\alpha,\beta}$, more efficient code including vector memory operations will be executed. The right-hand side of Fig. 3 shows the automatically generated code for the `fastWalshTransform` kernel for $w = 2$ without imposing bounds α and β. For readability reasons, we use OpenCL notation instead of the LLVM intermediate representation [13], which we actually use at that stage of compilation. The corresponding condition

```
step <= 0 || step % 2 != 0
```
(5)

in the first if-statement describes the complement of the set $A_{w,\alpha,\beta}$ obtained from our SMT solving step.

Table 5. Output from the SMT solving step for all our problem sets. We have $X \subseteq \{\, a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_{16} 0 \,\}$ with $|X| = 312$, i.e., timeouts occur only for input a with $a \equiv_{16} 0$.

SMT Solving Step Output

Problem Set	w	α	β	$A_{w,\alpha,\beta}$
FastWalshTransform	4	1	2^{16}	$\{\, a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_4 0 \,\}$
FastWalshTransform	8	1	2^{16}	$\{\, a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_8 0 \,\}$
FastWalshTransform	16	1	2^{16}	$\{\, a \in \mathbb{Z} \mid 1 \leq a < 2^{16} \wedge a \equiv_{16} 0 \,\} \setminus X$
BitonicSort	4	0	63	$\{0, \ldots, 62\} \setminus \{0, 1\}$
BitonicSort	8	0	63	$\{0, \ldots, 62\} \setminus \{0, 1, 2\}$
BitonicSort	16	0	63	$\{0, \ldots, 62\} \setminus \{0, 1, 2, 3\}$

Due to our independent runs of the SMT solver for all possible choices of a, the sets $A_{w,\alpha,\beta}$ are obtained explicitly as lists of elements. From these, we have to generate implicit descriptions like (5). One approach for this is to represent the characteristic functions of the sets $A_{w,\alpha,\beta}$ as bit strings and to use incremental finite automata minimization to obtain minimal regular expressions. These are finally transformed into quantifier-free Presburger conditions. Alternatively, one could apply automatic synthesis techniques as suggested by Gulwani et al. [7]. At present, this step is not automated yet.

6 Evaluation: OpenCL Performance

We evaluate the effect of our improved code generation for memory accesses for the applications that contain the kernels we discussed throughout the paper: FastWalshTransform and BitonicSort. To this end, we have hooked into the WFV OpenCL driver [11], employing our SMT-based approach to generate machine code.

In each respective kernel, there are actually several memory operations, which happen to lead to the same satisfiability problems $\neg\varphi(w, a)$. It is worth noting that for the majority of kernels that we found in the AMD APP SDK, the memory address computations are so simple that the relevant equations are decided already via term simplification, i.e. without any non-trivial SMT solving. Nevertheless, our technique is to our knowledge the first one that enables the compiler to generate better code in less simple cases such as FastWalshTransform and BitonicSort. Hence, if maximum performance of a kernel with complex memory operations is desired, our approach is the only currently available option.

In Table 6, we report kernel execution times of our SMT-enhanced driver in different configurations.[6] Each measurement shows the median of 1000 individual

[6] These experiments were conducted on a Core 2 Quad at 2.8 GHz with 8 GB of RAM running Ubuntu Linux 64 bit. The vector instruction set is Intel's SSE 4.2, yielding a SIMD width of four 32 bit values.

Table 6. Median of kernel execution times for 1000 executions of various OpenCL CPU drivers: Non-vectorized (Scalar), vectorized (WFV), and vectorized with our SMT-based optimization (WFV+SMT). As a reference, we also include the performance of the latest Intel and AMD implementations.[7] The Speedup column shows the effect of our SMT-based memory access optimization, comparing WFV+SMT to WFV.

OpenCL Kernel Performance (milliseconds)							
Application	Array Size	AMD	Intel	Scalar	WFV	WFV+SMT	Speedup
FastWalshTransform	16,777,216	413	313	303	309	299	1.03×
BitonicSort	1,048,576	894	680	236	121	58	2.09×

runs per configuration per benchmark without warm-up. Although the machine was not rebooted after every run, the numbers reported here are as realistic as possible for one cold-started, arbitrary run of the application.

The results clearly demonstrate the applicability of our approach. For FastWalshTransform, this is the first time that we were able to beat the scalar implementation with the WFV-based one [11]. It turns out, however, that the optimized code can be executed in only one out of w cases, which limits the performance gain. The situation is different for BitonicSort. Here, the optimized code is executed in the majority of cases. The factor of 2.09 for BitonicSort is huge, and also the 3 percent speedup for FastWalshTransform is relevant.

It is remarkable that in spite of including a WFV implementation, the Intel driver *refuses* to vectorize either of the two kernels. This means that Intel's heuristics deem the code to not benefit from vectorization. The reasons are probably that they consider the memory operations to dominate the runtime and that the heuristics have to assume that these operations are not consecutive. The AMD CPU driver does not use WFV.

7 Related Work

The basic analysis of memory address computations for linear translations by constants, which we extend, was introduced as part of the *vectorization analysis* of WFV [10,11]. Coutinho et al. [4] proposed a similar *divergence analysis*, which identifies values that remain the same for all work items but does not analyze consecutivity.

An increasing number of OpenCL drivers is being developed by different software vendors for all kinds of platforms from GPUs to mobile devices. For comparison purposes, the x86 CPU compiler from Intel is most interesting, although most details about the underlying implementation are not disclosed. According to our experimental results, its analyses are not as advanced as ours. The Portland Group implemented an x86 CPU driver for CUDA, which also makes use of

[7] `software.intel.com/en-us/vcsource/tools/opencl`, the Intel SDK for OpenCL Applications XE 2013 Beta.

both multi-threading and WFV.[8] Again, no details are publicly available. The AMD driver, the POCL project [9], TwinPeaks [8], MCUDA [18], and Ocelot [6] are other notable CPU implementations of the OpenCL and CUDA APIs, but none of them employ WFV.

For GPUs, various approaches exist to analyze memory access patterns for coalescing. However, none of the static approaches can handle integer division, modulo by constants, or non-constant inputs. CuMAPz [12] and the official CUDA Visual Profiler perform dynamic analyses of memory access patterns and report non-coalesced operations to the programmer. Yang et al. [21] implemented a static compiler optimization to improve non-coalesced accesses using shared memory. Li et al. [14] proposed an SMT-based approach for verification of GPU kernels. This was extended by Lv et al. [15] to also profile coalescing. Tripakis et al. [19] use an SMT solver to prove non-interference of SPMD programs. GPUVerify [2] is a tool that uses Z3 to prove OpenCL and CUDA kernels to be free from race-conditions and barrier divergence. None of these SMT-based techniques is concerned with automatic code optimization but only with verification.

8 Conclusions and Future Work

We have improved the state-of-the art in CPU code generation for memory access operations in data-parallel languages. The key idea is to prove at compile time that certain memory operations access consecutive addresses. This task is automatically translated to sets of formal problems that can be processed by an off-the-shelf SMT solver. We have introduced modulo elimination, a preprocessing technique on those formal problems which makes the approach practically feasible. The SMT output admits to construct case distinctions that are crucial for generating efficient code. Our performance measurements have demonstrated that our generated code is more efficient than all previous approaches including proprietary implementations by Intel and AMD.

To turn our prototypical environment into one integrated software system, a few gaps have to be closed: The compiler has to be linked with the SMT solver to minimize communication overhead. The code generation via finite automata or automatic synthesis as suggested in Sect. 5 has not yet been implemented.

On the other hand, the results achieved here open up interesting perspectives for future work in the areas combined in this paper: Modulo elimination should be included in SMT solvers with appropriate heuristics. More generally, sophisticated translation of integer division and modulo operations in special cases might boost performance also in other application areas. It appears promising to generate from the final SMT input $\neg\varphi(w, a)$ equivalent integer linear programs and to compare the performance of corresponding software. On the compiler side, a next step would be to automatically deduce tight ranges for the input values, e.g., from their data types or even from their usage. It is quite clear that our approach can be adapted to GPUs with SIMD instruction sets that have recently entered the market, e.g., the latest AMD Sea Islands series.

[8] The Portland Group, Inc. PGI CUDA-x86.

Acknowledgements. We would like to thank Sebastian Hack for helpful discussions. This research was supported in part by the German Transregional Collaborative Research Center SFB/TR 14 AVACS and in part by the German Federal Ministry of Education and Research (BMBF).

References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
2. Betts, A., Chong, N., Donaldson, A., Qadeer, S., Thomson, P.: Gpuverify: a verifier for gpu kernels. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 113–132. ACM, New York (2012)
3. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013)
4. Coutinho, B., Sampaio, D., Pereira, F.M.Q., Meira, W.: Divergence analysis and optimizations. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 320–329 (2011)
5. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Diamos, G.F., Kerr, A.R., Yalamanchili, S., Clark, N.: Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, pp. 353–364. ACM, New York (2010)
7. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, pp. 62–73. ACM, New York (2011)
8. Gummaraju, J., Morichetti, L., Houston, M., Sander, B., Gaster, B.R., Zheng, B.: Twin peaks: a software platform for heterogeneous computing on general-purpose and graphics processors. In: PACT, pp. 205–216. ACM, New York (2010)
9. Jaskelainen, P.O., de La Lama, C.S., Huerta, P., Takala, J.: OpenCL-based Design Methodology for Application-Specific Processors. In: SAMOS 2010, pp. 223–230 (July 2010)
10. Karrenberg, R., Hack, S.: Whole function vectorization. In: CGO, pp. 141–150 (2011)
11. Karrenberg, R., Hack, S.: Improving Performance of OpenCL on CPUs. In: O'Boyle, M. (ed.) CC 2012. LNCS, vol. 7210, pp. 1–20. Springer, Heidelberg (2012)
12. Kim, Y., Shrivastava, A.: Cumapz: a tool to analyze memory access patterns in CUDA. In: Proceedings of the 48th Design Automation Conference, DAC 2011, pp. 128–133. ACM, New York (2011)
13. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: CGO (March 2004)
14. Li, G., Gopalakrishnan, G.: Scalable smt-based verification of GPU kernel functions. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, pp. 187–196. ACM, New York (2010)

15. Lv, J., Li, G., Humphrey, A., Gopalakrishnan, G.: Performance degradation analysis of gpu kernels. In: Proceedings of the Workshop on Exploiting Concurrency Efficiently and Correctly 2011, EC2 2011 (2011)
16. Presburger, M.: 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, Warsaw, Poland, pp. 92–101 (1929)
17. Robinson, J.: Definability and decision problems in arithmetic. J. Symb. Log. 14(2), 98–114 (1949)
18. Stratton, J.A., Stone, S.S., Hwu, W.-M.W.: MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 16–30. Springer, Heidelberg (2008)
19. Tripakis, S., Stergiou, C., Lublinerman, R.: Checking equivalence of spmd programs using non-interference. Technical Report UCB/EECS-2010-11, EECS Department, University of California, Berkeley (January 2010)
20. Weispfenning, V.: The complexity of almost linear Diophantine problems. Journal of Symbolic Computation 10(5), 395–403 (1990)
21. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, pp. 86–97. ACM, New York (2010)