Presburger Arithmetic in Memory Access Optimization for Data-Parallel Languages

Marek Košta
(joint work with R. Karrenberg and T. Sturm)

Max Planck Institute for Informatics

18.9.2013

Data-Parallel Languages

Single Program Multiple Data (SPMD) Paradigm

- Technical details of parallelization are abstracted away.
- The programmer writes a scalar function code, called the **kernel**.
- The kernel is executed in multiple **work items** by a runtime system.
- Work items can be viewed as threads which differ only in their **ID**.
- Work items can query their ID to execute different tasks.

Examples of Data-Parallel Languages

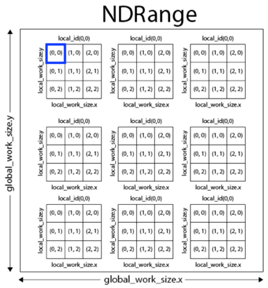
OpenCL (Khronos Group), CUDA (NVIDIA), PVM (University of Tennessee)

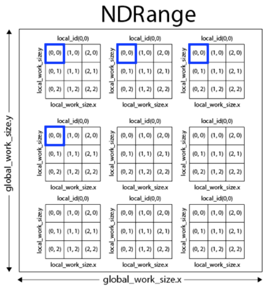
Execution of the Work Items

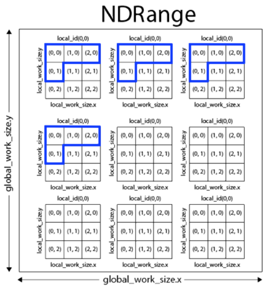
The runtime system decides how the work items will be executed. This task is platform-dependent.

- On GPU, this is straightforward: One work item corresponds to one hardware-managed thread.
- On CPU, external libraries (pthreads, OpenMP or MPI) have to be employed to obtain the wanted effect: One work item running on one CPU core and all CPU cores busy.

In this talk we consider compilation of data-parallel languages for **SIMD CPUs**.







Single Instruction Multiple Data

- SIMD is another level of parallelism which modern CPUs offer.
- Execution of the same operation on multiple input data at once, i.e. **vectorization**.
- The SIMD width w of a CPU is the number of single-precision values that fit into one vector register. Typical values for w are 4, 8 or 16.
- A technique called **Whole-Function Vectorization** (WFV) transforms a kernel so that w work items can be executed at once by a single hardware thread (CPU core).
- Therefore, WFV can increase performance of application by a factor as large as w.
- In practice, WFV has drawbacks such that applying WFV can even result in slowdowns. . .

WFV Applied to Compilation of OpenCL for SIMD capable CPUs

The Main Idea of WFV

To compute w work items at once do the following:

- Transform accesses to tid (ID of a work item) to return a vector of w consecutive values, always starting at nw, where $n \geq 0$.
- Transform each operation into its vector counterpart, e.g. addition becomes a scalar addition.

Problem!

Vector counterparts for memory operations work only for consecutive addresses. If the addresses are non-consecutive, w sequential operations have to be used. This can dramatically decrease performance!

This problem does not exist on GPUs. There is dedicated hardware to dynamically coalesce more memory accesses to a single one whenever possible.

When are the accessed addresses consecutive?

An easy example:

```
__kernel void
shift(float* in,
      float* out,
      int a) {
  int tid = get_global_id();
  out[tid] = in[tid+1];
}
```

A not so obvious example:

```
__kernel void
fwtExcerpt(float* tArray,
           int    step) {
  int tid   = get_global_id();
  int group = tid % step;
  int pair  = 2*step*(tid/step)
              + group;
  float num    = tArray[pair];
  tArray[pair] = num;
}
```

Memory accesses in the left-hand side example, in[tid+1], are consecutive because tids are consecutive. Memory access pattern of the left-hand side example, tArray[pair], is more complicated: The accessed addresses are consecutive only in some cases.
Without compiler optimization, the memory operations in both cases would be executed sequentially.

Problem Formulation

Consecutivity Question

Given a kernel and one particular memory access in it: If executed by work items with consecutive tids, will the accessed memory locations be contiguous?

We ask the consecutivity question statically, **not** at runtime.
Reason: Consecutivity check could be done at runtime (by generating appropriate code) but the time spent on checking outperforms the gains in most cases.

Allowed Operations

Hardness of the consecutivity question depends on the arithmetic operations allowed in the expression describing the accessed address. This is in general undecidable. Therefore, we restrict ourselves to expressions in **Presburger Arithmetic** with division and modulo by constants. Current state-of-the-art techniques can handle only translations by constants.

Presburger Arithmetic

The Original Theory

First-order theory (with equality) of the integers with countably infinite language $\mathcal{L}$ consisting of 0, 1, $+$, $-$, $<$ and infinitely many congruences $\equiv_k$ for $k \in \mathbb{N} \setminus \{0\}$. Predicates $\leq$, $\neq$, $>$ and $\geq$ can be used as well: They are **definable** in $\mathcal{L}$.

For this theory, Presburger has given a decision procedure based on effective quantifier elimination. Congruences are part of the language because of the QE procedure: Theory with language without them does not admit QE.

Modulo and Division by a Constant

For fixed integer $k \in \mathbb{N} \setminus \{0\}$, these can be encoded in $\mathcal{L}$ as follows:

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

Problem Formalization (1)

Memory Address Term

We translate each expression describing an address to a term e.
Term $e(t, a)$ describing a memory location to read from (write to) is a Presburger term containing only variables t (tid) and a (possible input).
Integer functions $\text{mod}_k(x)$ and $\text{div}_k(x)$ are allowed for constant k only.

Two work items with consecutive tids access contiguous memory locations for input a if and only if

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

Any w work items with consecutive tids access contiguous memory locations for input a if and only if

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

is true.

Problem Formalization (2)

Because WFV assigns tids in such a way that w consecutive work items start at nw, where $n \geq 0$, the only conjunctions relevant are those where w divides t.

Formalization of the Consecutivity Question in PA

For one particular memory access with memory address term $e(t, a)$ and fixed values for w and a the following formula is true if and only the answer to the consecutivity question is true:

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

Presburger Arithmetic and SMT-LIB2

Important Technical Issue

We defined mod_k and div_k for positive k only. For negative k, SMT-LIB2 Ints theory and OpenCL standard (following C99 standard) differ. In SMT-LIB2, $\text{mod}_k(x)$ is always non-negative in contrast to OpenCL standard where $\text{mod}_k(x)$ can be negative.

In our case this is no problem because k is positive in the majority of applications. If negative k was needed, translation to Presburger Arithmetic would be a little bit more complicated.
There are more logics involving integers defined by SMT-LIB2 standard. For our purposes QF_NIA will be sufficient because our input is limited to the existential fragment of Presburger Arithmetic with mod_k and div_k.

Using an SMT Solver to Answer the Consecutivity Question

Negating $\varphi(w, a)$ and moving the existential quantifier into the disjunction yields

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

For given w and a the consecutivity question can be answered by at most $w-1$ many calls to an SMT solver.
For given $w \in \mathbb{N}$ and $\alpha, \beta \in \mathbb{Z}$ with $\alpha \leq \beta - 1$, the answer to the consecutivity question for w and $a \in \{\alpha, \ldots, \beta-1\}$ is given by the set

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

which can be computed by at most $(w-1)(\beta-\alpha-1)$ many applications of an SMT solver.

SMT Solving – FastWalshTransform Example

Which SMT solver do we use?

We use Z3 (version 4.3.1) because it supports mod and div symbols off-the-shelf. Neither CVC4 (version 1.1), nor MathSAT5 (version 5.1.3) support mod and div symbols in the input directly.

Running times of Z3 applied to $\neg\varphi(w, a)$ where

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

In all three cases $\alpha = 1$ and $\beta = 2^{16}$. Time limit was set to one minute per call.

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

Set $A_{w,\alpha,\beta}$ consists of those a for which we get “unsat”.

Modulo Elimination as a Preprocessing Step

Running times of the SMT solving are inacceptable. To improve the performance of an SMT solver we use the following equation:

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

- Given t, we can use this equation (from left to right) to obtain an equivalent term t' which does not contain mod_k operators.
- This has to be done from bottom to top in the term tree of t.
- Doing this for all terms of $\varphi(w, a)$ we obtain equivalent $\varphi'(w, a)$.
- Note that $|\varphi'(w, a)|$ can be $2^{|\varphi(w,a)|}$ in the worst case.

SMT Solving with Preprocessing

Running times of Z3 applied to $\neg\varphi'(w, a)$ obtained by the preprocessing described above. In all three cases $\alpha = 1$ and $\beta = 2^{16}$. Time limit was set to one minute per call.

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

- Running times are not fast enough for just-in-time compilation.
- Running times are acceptable for offline compilation.
- Speedup factors are promising.

Code Generation

- As result of the SMT solving step we obtain the set $A_{w,\alpha,\beta}$ explicitly as a list of elements.
- If $a \in A_{w,\alpha,\beta}$ we know that a vector memory operation can be executed.
- Our aim is to produce a succinct description D of $A_{w,\alpha,\beta}$ in the form of a quantifier-free formula. Example: $a > 5$, $\mathrm{mod}_2(a) = 0$.
- This can be done employing finite automata minimization or by automatic synthesis techniques.
- Based on this description D, the compiler generates code executing the following: Check if D holds. If yes, then vectorized memory operation is executed, otherwise w memory operations are executed.
- This step is not automated yet.

Performance Evaluation

- We tried our technique on kernels that we found in the AMD APP SDK.
- In the majority of cases the memory address computations are either trivial enough to be decided without our SMT solving approach or too complicated.
- However, our technique is the only one which can generate vectorized code in non-trivial cases.

Median of kernel execution times for 1000 executions: Non-vectorized (Scalar), vectorized (WFV) and vectorized with our SMT-based optimization (WFV+SMT). The Speedup column shows the effect of our SMT-based approach, comparing WFV+SMT to WFV.

OpenCL Kernel Performance (milliseconds)

Application	AMD	Intel	Scalar	WFV	WFV+SMT	Speedup
FWT	413	313	303	309	299	1.03×
BS	894	680	236	121	58	2.09×

Memory Access Optimization Process

For one memory access operation in the kernel code a compiler executes the following:

1. Extract the memory access term describing the address.
2. If it is not a Presburger Arithmetic term: Our approach cannot be used: Generate code for the serialized memory access and terminate.
3. Construct term $e(w, a)$ and use it to construct $\varphi(w, a)$.
4. Apply modulo elimination to obtain $\varphi'(w, a)$.
5. Use an SMT solver to decide $\neg\varphi'(w, a)$ for an interesting set of inputs. Result of this step is set $A_{w,\alpha,\beta}$ of those a where the answer is “unsat”.
6. By the process described above, generate code which executes vectorized memory accesses when possible.

Results

1. We formalized in Presburger Arithmetic the consecutivity question and conditions that have to be decided to statically optimize memory operations.
2. Our formalization allows to consider address computations that involve div_k and mod_k and limited occurrences of an input variable.
3. We used 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 in reasonable time.
4. We made systematic computational experiments to show that our approach is feasible.
5. Our computations established new benchmarks based on current research problems in compiler construction. These benchmarks could be added to SMT-LIB benchmark library.
6. We showed 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.

Future Work

- Try to do something more clever than brute-force.
- Investigate in which cases the consecutivity question could be answered by using linear integer programming solvers.
- Link the compiler and SMT solver together to minimize communication overhead.
- Implement and compare performance of automatic synthesis techniques and incremental finite automata minimization.
- Investigate usability of the modulo elimination and other heuristics for translation of integer modulo and division operations.

Thank you for your attention!