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

Marek Košta

Max Planck Institute for Informatics

MACIS 2013, Nanning, China
December 12, 2013

Our Memory Access Optimization Problem

What are we doing?

- SIMD paradigm, vectorization
- data-parallel languages: **OpenCL**, CUDA, etc. focusing on GPUs
- goal: compile OpenCL for SIMD capable CPUs

Definition

kernel = function to be compiled to vectorized code
work item = one entity executing “one coordinate” of the vectorized code
SIMD width w = number of entities
ID = unique identifier of an entity

One central problem when switching from GPUs to SIMD CPUs

- vectorized CPU instructions resemble to GPU instructions
- but: CPUs lack hardware support for recognizing aligned memory accesses
- our idea: solve this **consecutivity question** in software at compile time

Our Memory Access Optimization Problem

What are we doing?

- SIMD paradigm, vectorization
- data-parallel languages: **OpenCL**, CUDA, etc. focusing on GPUs
- goal: compile OpenCL for SIMD capable CPUs

Definition

kernel = function to be compiled to vectorized code
work item = one entity executing “one coordinate” of the vectorized code
SIMD width w = number of entities
ID = unique identifier of an entity

One central problem when switching from GPUs to SIMD CPUs

- vectorized CPU instructions resemble to GPU instructions
- but: CPUs lack hardware support for recognizing aligned memory accesses
- our idea: solve this **consecutivity question** in software at compile time

Our Memory Access Optimization Problem

What are we doing?

- SIMD paradigm, vectorization
- data-parallel languages: **OpenCL**, CUDA, etc. focusing on GPUs
- goal: compile OpenCL for SIMD capable CPUs

Definition

kernel = function to be compiled to vectorized code
work item = one entity executing "one coordinate" of the vectorized code
SIMD width w = number of entities
ID = unique identifier of an entity

One central problem when switching from GPUs to SIMD CPUs

- vectorized CPU instructions resemble to GPU instructions
- but: CPUs lack hardware support for recognizing aligned memory accesses
- our idea: solve this **consecutivity question** in software at compile time

A Standard OpenCL Benchmark Kernel

```
__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;
}
```

One arithmetically non-trivial memory access expression

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

x – the ID of a work item

a – the function argument

A Standard OpenCL Benchmark Kernel

```
__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;
}
```

One arithmetically non-trivial memory access expression

$e(x, a) = 2a \odot \mathrm{div}_a(x) + \mathrm{mod}_a(x)$
x – the ID of a work item
a – the function argument

Our Approach (1)

1. Formalize the **consecutivity question**:

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

2. Solve it, obtaining a set of values a, for which $\varphi(w, a)$ holds.
3. Generate code: More efficient code is executed when the consecutivity question is true for the value of the input parameter.

The input parameter is treated as a constant and instantiated.

Reason: Integer arithmetic with binary division and/or modulo is undecidable.

Our Approach (1)

1. Formalize the **consecutivity question**:

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

2. Solve it, obtaining a set of values a, for which $\varphi(w, a)$ holds.
3. Generate code: More efficient code is executed when the consecutivity question is true for the value of the input parameter.

The input parameter is treated as a constant and instantiated.

Reason: Integer arithmetic with binary division and/or modulo is undecidable.

Our Approach (1)

1. Formalize the **consecutivity question**:

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

2. Solve it, obtaining a set of values a, for which $\varphi(w, a)$ holds.
3. Generate code: More efficient code is executed when the consecutivity question is true for the value of the input parameter.

The input parameter is treated as a constant and instantiated.

Reason: Integer arithmetic with binary division and/or modulo is undecidable.

Our Approach (1)

1. Formalize the **consecutivity question**:

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

2. Solve it, obtaining a set of values *a*, for which $\varphi(w,a)$ holds.
3. Generate code: More efficient code is executed when the consecutivity question is true for the value of the input parameter.

The input parameter is treated as a constant and instantiated.

Reason: Integer arithmetic with binary division and/or modulo is undecidable.

Our Approach (2)

- Instantiation of a leads to **many** SMT problems, e.g., $a = 1, \ldots, 2^{16}$.
- Direct use of Z3 turned out to be infeasible in practice.

Our Solution

modulo elimination as a preprocessing step: Systematically apply $\mathrm{mod}_a(x) \to a - a\,\mathrm{div}_a(x)$ before SMT-solving.

- solving 2^{16} instances by Z3 (v4.3.1):

w	Timeouts	CPU Time	Elim Timeouts	Elim CPU Time	Speedup
4	361	14 h	0	4 min	210×
8	3,331	97 h	0	5 min	1164×
16	10,294	256 h	312	334 min	46×

- performance gain: The generated code is from 1.03 to 2 times faster.
- significant advantage of our approach: performance can not be worse
- better performance than any state-of-the-art compilers including Intel and AMD on examples from AMD APP SDK

Our Approach (2)

- Instantiation of a leads to **many** SMT problems, e.g., $a = 1, \ldots, 2^{16}$.
- Direct use of Z3 turned out to be infeasible in practice.

Our Solution

modulo elimination as a preprocessing step: Systematically apply $\mathrm{mod}_a(x) \to a - a\,\mathrm{div}_a(x)$ before SMT-solving.

- solving 2^{16} instances by Z3 (v4.3.1):

w	Timeouts	CPU Time	Elim Timeouts	Elim CPU Time	Speedup
4	361	14 h	0	4 min	210×
8	3,331	97 h	0	5 min	1164×
16	10,294	256 h	312	334 min	46×

- performance gain: The generated code is from 1.03 to 2 times faster.
- significant advantage of our approach: performance can not be worse
- better performance than any state-of-the-art compilers including Intel and AMD on examples from AMD APP SDK

Our Approach (2)

- Instantiation of a leads to **many** SMT problems, e.g., $a = 1, \ldots, 2^{16}$.
- Direct use of Z3 turned out to be infeasible in practice.

Our Solution

modulo elimination as a preprocessing step: Systematically apply $\text{mod}_a(x) \to a - a\,\text{div}_a(x)$ before SMT-solving.

- solving 2^{16} instances by Z3 (v4.3.1):

w	Timeouts	CPU Time	Elim Timeouts	Elim CPU Time	Speedup
4	361	14 h	0	4 min	210×
8	3,331	97 h	0	5 min	1164×
16	10,294	256 h	312	334 min	46×

- performance gain: The generated code is from 1.03 to 2 times faster.
- significant advantage of our approach: performance can not be worse
- better performance than any state-of-the-art compilers including Intel and AMD on examples from AMD APP SDK

Our Approach (2)

- Instantiation of a leads to **many** SMT problems, e.g., $a = 1, \dots, 2^{16}$.
- Direct use of Z3 turned out to be infeasible in practice.

Our Solution

modulo elimination as a preprocessing step: Systematically apply $\mathrm{mod}_a(x) \to a - a\,\mathrm{div}_a(x)$ before SMT-solving.

- solving 2^{16} instances by Z3 (v4.3.1):

w	Timeouts	CPU Time	Elim Timeouts	Elim CPU Time	Speedup
4	361	14 h	0	4 min	210×
8	3,331	97 h	0	5 min	1164×
16	10,294	256 h	312	334 min	46×

- performance gain: The generated code is from 1.03 to 2 times faster.
- significant advantage of our approach: performance can not be worse
- better performance than any state-of-the-art compilers including Intel and AMD on examples from AMD APP SDK

Our Approach (2)

- Instantiation of a leads to **many** SMT problems, e.g., $a = 1, \dots, 2^{16}$.
- Direct use of Z3 turned out to be infeasible in practice.

Our Solution

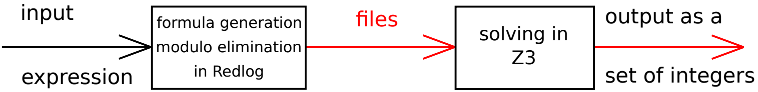
modulo elimination as a preprocessing step: Systematically apply $\mathrm{mod}_a(x) \to a - a\,\mathrm{div}_a(x)$ before SMT-solving.

- solving 2^{16} instances by Z3 (v4.3.1):

w	Timeouts	CPU Time	Elim Timeouts	Elim CPU Time	Speedup
4	361	14 h	0	4 min	210×
8	3,331	97 h	0	5 min	1164×
16	10,294	256 h	312	334 min	46×

- performance gain: The generated code is from 1.03 to 2 times faster.
- significant advantage of our approach: performance can not be worse
- better performance than any state-of-the-art compilers including Intel and AMD on examples from AMD APP SDK

Proof of Concept “System”



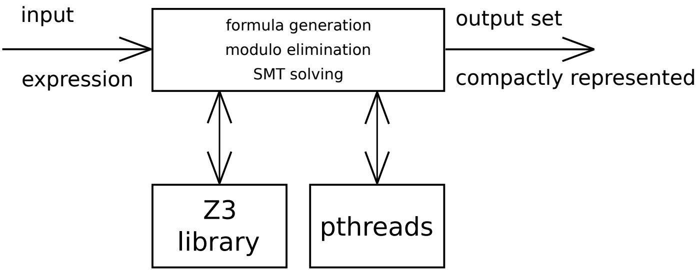
Drawbacks

- redundant combination of two systems
- communication through files
- scripting needed to make it work
- no library available → Usage within a more complex system is not possible.

Advantages

- flexible environment for rapid prototyping

Current Work in Progress



Design Aims

- multithreading for the e.g. 2^{16} instances of *a*
- directly linkable to a compiler with a suitable API
- minimal infrastructure needed
- fast and portable

Short System Description

Solving Process

Input: a memory access expression $e(x, a)$ and interval $[u; l]$

Output: a sorted list of disjoint periodic sets

1. Construct the formula $\varphi(w, a)$ and divide $[u; l]$ into n parts.
2. Spawn n working threads.
3. Collect and merge partial solutions.

Problem with Threads

- The real running time does not scale!
- Reason for this is unacceptable increase in the user time.
- We are in contact with Z3 developers about this issue.

Workaround Solution

Each thread spawns a process, which carries the actual computation.

Short System Description

Solving Process

Input: a memory access expression $e(x, a)$ and interval $[u; l]$

Output: a sorted list of disjoint periodic sets

1. Construct the formula $\varphi(w, a)$ and divide $[u; l]$ into n parts.
2. Spawn n working threads.
3. Collect and merge partial solutions.

Problem with Threads

- The real running time does not scale!
- Reason for this is unacceptable increase in the user time.
- We are in contact with Z3 developers about this issue.

Workaround Solution

Each thread spawns a process, which carries the actual computation.

Short System Description

Solving Process

Input: a memory access expression $e(x, a)$ and interval $[u; l]$

Output: a sorted list of disjoint periodic sets

1. Construct the formula $\varphi(w, a)$ and divide $[u; l]$ into n parts.
2. Spawn n working threads.
3. Collect and merge partial solutions.

Problem with Threads

- The real running time does not scale!
- Reason for this is unacceptable increase in the user time.
- We are in contact with Z3 developers about this issue.

Workaround Solution

Each thread spawns a process, which carries the actual computation.

Compact Representation of the Answer Set

At present we restrict ourselves to sorted lists of disjoint **periodic sets**:

- A periodic set is $\{x \mid x \in \mathbb{Z} \wedge a \leq x \leq b \wedge x \equiv_m c\}$.
- straightforward representation
- Implemented operations: `add_point` and `merge` (union).

Pros

- (in practice) constant space
- `add_point` and `merge` operations take constant time
- compatible with concurrency

Cons

- limited expressive power
- possible generalizations:
  - non-disjoint sets in the list
  - incremental automata minimization

Compact Representation of the Answer Set

At present we restrict ourselves to sorted lists of disjoint **periodic sets**:

- A periodic set is $\{x \mid x \in \mathbb{Z} \wedge a \leq x \leq b \wedge x \equiv_m c\}$.
- straightforward representation
- Implemented operations: `add_point` and `merge` (union).

Pros

- (in practice) constant space
- `add_point` and `merge` operations take constant time
- compatible with concurrency

Cons

- limited expressive power
- possible generalizations:
  - non-disjoint sets in the list
  - incremental automata minimization

SMT vs. ILP

Characteristics of our formulas

- simple Boolean structure
- numerous different but comparatively simple instances
- linear integer arithmetic with division and modulo by constants

- This can be transformed into ILP.
- Surprise: Gurobi v5.5 cannot compete with Z3.

Key Observation

- ILP solver: better performance in SAT cases
- SMT solver: **outperforms** ILP in UNSAT cases

SMT vs. ILP

Characteristics of our formulas

- simple Boolean structure
- numerous different but comparatively simple instances
- linear integer arithmetic with division and modulo by constants

- This can be transformed into ILP.
- Surprise: Gurobi v5.5 cannot compete with Z3.

Key Observation

- ILP solver: better performance in SAT cases
- SMT solver: **outperforms** ILP in UNSAT cases

SMT vs. ILP

Characteristics of our formulas

- simple Boolean structure
- numerous different but comparatively simple instances
- linear integer arithmetic with division and modulo by constants

- This can be transformed into ILP.
- Surprise: Gurobi v5.5 cannot compete with Z3.

Key Observation

- ILP solver: better performance in SAT cases
- SMT solver: **outperforms** ILP in UNSAT cases

Future Work

- combination with a compiler (cooperation with S. Hack and R. Karrenberg at Saarland University)
- study possible combinations of ILP and SMT
- consider other theories than integers to the extent supported by SMT solvers
- more experiments
- more expressive and still efficient representation of the answer set

Summary

1. Introduced memory access optimization problem.
2. Formalized the consecutivity question.
3. Described the old “system”, which was used for some experimentation.
4. Current work: Development of a monolithic system.
5. Properties: Parallelism, compact answer sets representation, and easy-to-use in combination with a compiler.
6. Reported on implementation issues, which need to be resolved.

Additional Kernel Example

```
__kernel void
bitonicSort(__global uint * tArray,
            const uint stage,
            const uint passOfStage,
            const uint width,
            const uint direction) {
  uint tid = get_global_id();
  uint pairDistance = 1 << (stage - passOfStage);
  uint blockWidth   = 2 * pairDistance;
  uint leftId  = (tid % pairDistance) + (tid / pairDistance) * blockWidth;
  uint rightId = leftId + pairDistance;
  uint leftElement  = tArray[leftId];
  uint rightElement = tArray[rightId];
           .          .
           .          .
           .          .
  if (. . .) {
     tArray[leftId]  = lesser;
     tArray[rightId] = greater;
  } else
     . . .
}
```

Memory access expression

$$e(x,a) = 2^{a+1} \odot \mathrm{div}_{2^a}(x) + \mathrm{mod}_{2^a}(x) + 2^a$$