SIMD code generator

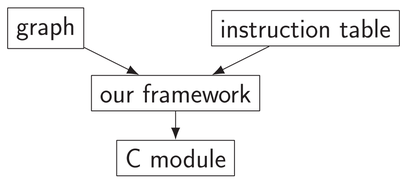
Karel Tuček

Introduction

The center of our interest is a problem of pipelined realisation of a special case of data processing networks. These realisations are supposed to realise computations on series of independent data sets while utilizing SIMD instructions. We also theoretically investigate the possibilities and the problems of employment of control flow in these networks and also to implement a general framework suitable for generation of these realisations. In other words, our aim was the design of a framework which could act as a back-end part of a (trans) compiler. For the purpose of implementation, we targeted transformation of graph-described computations into vectorised C code.

Integration

In yet other words, our generator is a program, which may be invoked from command line or from another programming environment. It takes graph description of a computation and a description of an instruction table and produces a C file which contains a module realising the computation.



Processing of basic blocks

Our idea is that SIMD parallelism should take place on the same instruction realised across multiple data sets. This way, vectorised C code may be easily generated via a topological search of the graph. During this search, instructions are fused across multiple data sets. Sometimes, the representation of data has to be transformed in order to fit its source and destination instruction. We add shortest conversion paths implicitly during the search.

Processing of control flow

If the input graph contains any control flow constructs, we factorise the network with respect to these and connect the resulting components via register-mapped buffers. In other words, we cut the graph into partitions which represent basic blocks without any control flow. Then, we generate a code fragment which crawls over the network and pushes data through, using code fragments generated by the approach presented above. This is necessary, because we represent control flow by means of stream split and merge instructions which fully split data into independent streams.

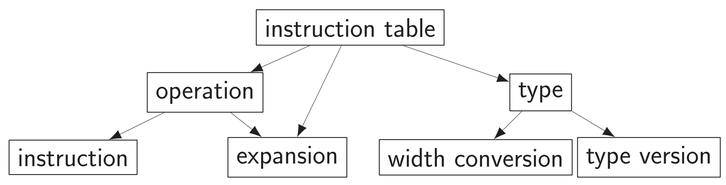
Problems of control flow

This approach presents multiple problems encompassing the question of data ordering and also of buffer sizes, since situations arise that different elements of the same data set have to wait for each other somewhere. We have to ensure that the data fit into the constant sized buffers, that the pipeline does not stall (due to overfilled or underfilled buffers) and that (ideally) all data get processed by vectorised instructions.

Example - instruction table - pseudo SIMD set

The following instruction table is a simplified example of description of a pseudo-vector instruction set. The actual table for production use would contain intrinsic functions in patterns. We reckognize the following constructs: operation, instruction, type, type version, width conversion, expansion, although flat format flattens some of these constructs into a single line. The second figure shows non-flattened hierarchy of the structures.

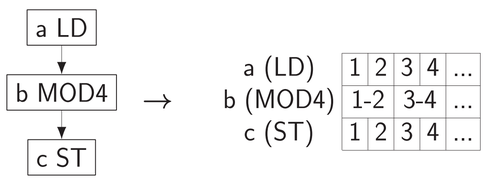
```
#type       type in out pattern                       pattern2
conversion  INT  2  1   $arg1 (b & 0xFFFF0000) >> 16  $arg1 & 0xFFFF
conversion  INT  1  2   $arg1 | ($arg2 << 16)

#type       id   width  pattern
type        INT  1      uint16_t
type        INT  2      uint32_t

#type       id   intpe  outtpe in out pattern
instruction MOD4 INT    INT    1  1   $arg1 % 4
instruction MOD4 INT    INT    2  2   $arg1 & 0x00030003
instruction LD          INT    1  1   $input
instruction ST   INT            1  1   $output
```



Example - generation of basic blocks

We show processing of a modulo 4 computation:

- The left figure shows the input flow graph.
- The table visualises how instructions (cells) are fused together across different data sets (represented by indices).
- The next block shows results produced for the example table. We can see a vectorised and a nonvectorised output fragment. Lines 3,5 and 6 of the vectorised fragment show needed width conversions, e.g., joining outputs of two LD instructions into a single vector.
- The last block shows an example output file.



→

a (LD)	1	2	3	4	...
b (MOD4)	1-2		3-4		...
c (ST)	1	2	3	4	...

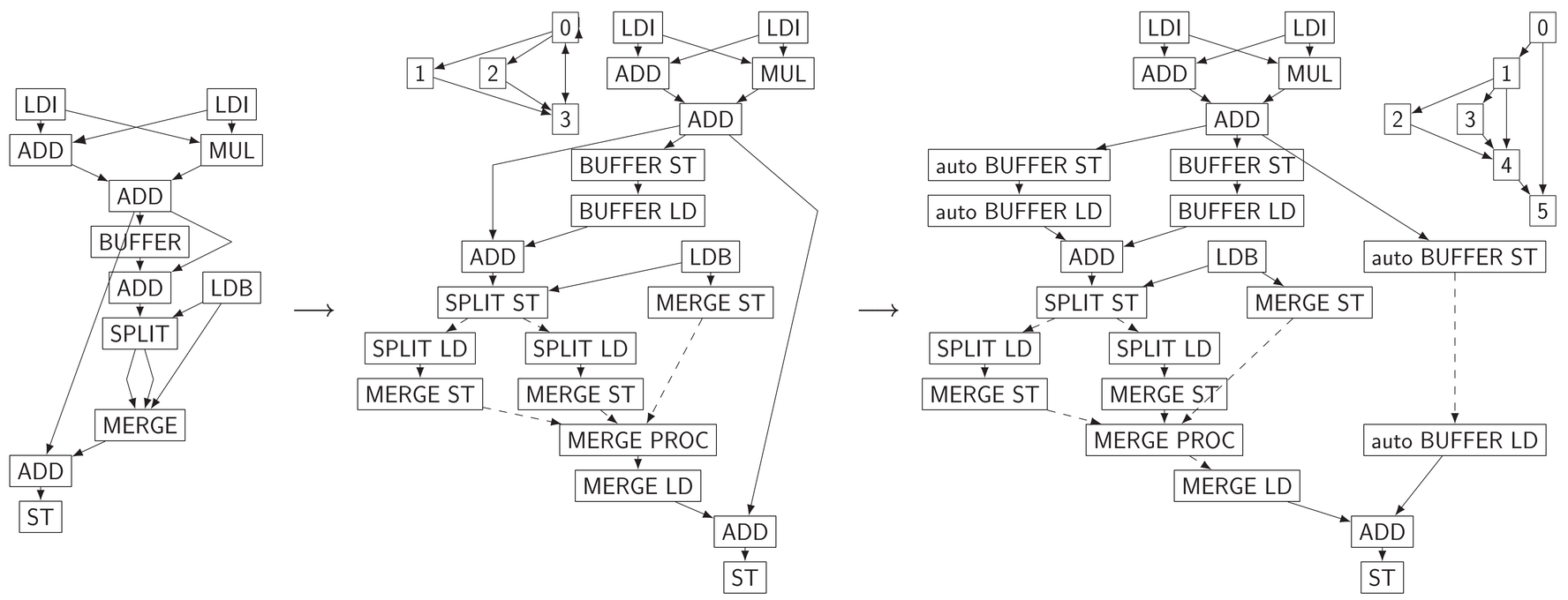
→

```
uint16_t a = input_a[i++];
uint16_t b = a % 4;
output_c[j++] = b;

uint16_t a_0 = input_a[i++];
uint16_t a_1 = input_a[i++];
uint32_t a_conv0 = a_0 | (a_1 << 16);
uint32_t b = a_conv0 & 0x00030003;
uint16_t b_conv0 = b & 0xFFFF;
uint16_t b_conv1 = (b & 0xFFFF0000) >> 16;
output_c[j++] = b_conv0;
output_c[j++] = b_conv1;
```

→

```
void f(uint16_t** input_a,
       uint16_t** output_c,
       int count)
{
  int i = 0;
  int j = 0;
  for(; count >= 2; count -= 2)
  { <vectorised fragment> }
  for(; count >  0; count--)
  { <nonvectorised fragment> }
}
```

Example - control flow preprocessing transformations



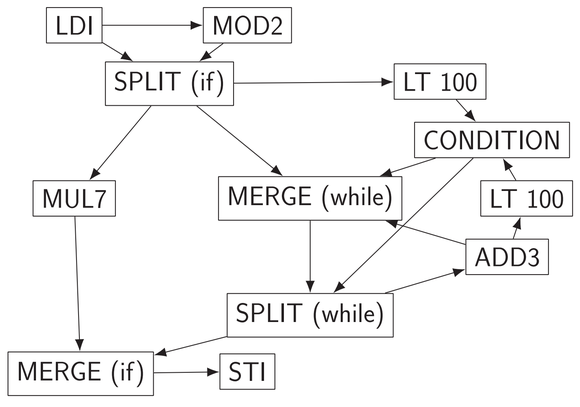
This figure shows transformations needed for employment of control flow. The first transformation replaces simple control-flow operations by new subgraphs, creating new edges which carry data via buffers. The graph of its components of connectedness (w.r.t. solid lines)(a *factor graph*) is not acyclic (as the attached factor graph shows). The second transformation removes all cycles by means of insertion of new buffers.

We need to obtain acyclic factor graph because we wish to process components (aka basic blocks) atomically, i.e., without leaving for processing of other components and especially without re-entering the same block again. Cycles and loops in the factor graph correspond exactly to paths containing buffers which return to their originating componentss.

Example - graph representation of control flow

This figure shows how control flow may be represented in the environment of processing networks. The control flow itself is handled by MERGE, SPLIT, and CONDITION nodes. The SPLIT and MERGE nodes control direction of flow of data elements. The CONDITION node manages behaviour of the loop by ensuring that elements enter and exit the loop when we expect them to do so.

```
int f(int a)
{
  if(a % 2 == 1)
    a = a*7;
  else
    while ( a < 100 )
      a = a+3;
  return a;
}
```



Architecture of our solution

Our implementation is designed to be easily extensible. Namely, it is general with respect to the target language, runtime environment and SIMD extension. For this reason we have decided to design it as a highly sophisticated text processor. Input consists of a file describing the computation and of a file containing an instruction table. Instruction table describes semantics of instruction identifiers and also provides instruction patterns. These patterns are written in a special notation which resembles the system of shell expansion. Our implementation handles graphs without control flow and networks which contain branching. We provide a fully tested instruction table of C operators, implemented via Intel's SSE extension.

Other information

- Bachelor Thesis: SIMD Code Generator
- Year: 2016
- Author: Karel Tuček, kareltucek at centrum.cz
- Supervisor: RNDr. David Bednárek, Ph.D., bednarek at ksi.mff.cuni.cz
- Insitution: Charles University in Prague, Faculty of Mathematics and Physics, Department of Software Engineering