Portable Support for Explicit Vectorisation in C

Artjoms Šinkarovs[1] and Sven-Bodo Scholz[2]

[1] University of Hertfordshire, Hatfield, Hertfordshire, AL10 9AB, United Kingdom
[2] Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, United Kingdom

Abstract. In pursuit of requirements of modern software, high-performance computing often offers a narrowly-tailored solutions that reduce portability of software. As one of such examples in this paper, we consider the situation with SIMD accelerators, whose importance seriously increased in the last decade. Firstly appeared in the early 90es, being oriented exclusively on graphics acceleration, nowadays SIMD CPU extensions are used in a variety of fields. The lack of standard and incompatibility of instruction sets through the different types of CPUs substantially increase the complexity of developing portable applications within the extensions. In this paper we present an abstraction layer implemented as a set of C language extensions within the GNU GCC compiler which provides an interface for SIMD vectors and operations independently from the architecture. First of all, these abstractions allow to exploit SIMD extensions of a CPU explicitly, which is useful when auto-vectoriser fails. Secondly, the abstractions are general enough to be mapped to any hardware supporting SIMD paradigms; hence the new abstractions could be considered as a step forward to a new C language standard.

1 Introduction

Starting from the early 90es most of the computational platforms have incorporated Single Instruction Multiple Data (SIMD) extensions into their processors. The SIMD mechanisms allow architectures to explore data-parallelism by executing one and the same operation on multiple data objects packed into a vector register that holds several scalar values. Efficient use of SIMD instructions proves increasingly important for achieving excellent performance in a variety of applications. Most computationally intensive applications spend most of their time inside a few hot-spots, typically the innermost loops. These very often apply arithmetic operations on large sets of indexed data, a situation very amenable to SIMD instructions. If applicable, such a use of SIMD instructions immediately increases performance of the loop several times depending on the size of the scalar data type. On current hardware one can expect speed-ups of factor two or four when operating with floating point numbers. Besides the immediate gains, the use of SIMD operations is typically orthogonal to any gains obtained from multi-threaded execution, i.e., in case of a successful SIMD optimisation the overall speed-up can be obtained by multiplying the speed-up due to the use of SIMD instructions with the number of cores available. Finally, we can observe that the length of vector registers provided by new architectures doubles every several years: Starting with 64 bit registers, the SSE architecture increased this to 128 bits, AVX operates on 256-bits and the MIC architecture already has 512-bit long vector registers. This suggests that algorithms suitable for SIMD operations should directly benefit from these innovations.

In an ideal world we would like to see all the inner-loop refactoring to happen automatically under the hood of a compiler. A lot of very successful research in this direction has been done already [9,12,10,3]. Most modern compilers including GNU's GCC, Intel's ICC, and LLVM are equipped with some form of auto-vectoriser. However, even the smartest auto-paralleliser is bound to be limited to the code pattern it has been programmed to recognise.

This paper is concerned with the situation the programmer is left in if this happens, i.e., when the auto-vectoriser fails despite dealing with some code that can be transformed into a SIMD suitable form. In that situation, it would be desirable if the programmer could explicitly instruct the compiler where to insert SIMD instructions. While this can always be achieved by inserting inline-assembly into the program, this constitutes an inherently non-portable solution. Not only does this imply a lock-in into a particular architecture, it also inhibits an immediate benefit from larger vector sizes in the next generation.

Furthermore, it puts quite some additional burden on the programmer as he has to acquire an in-depth understanding of the architecture that is being targeted. He also has to make sure that the interfacing between the inline assembly and the C context is handled properly which either requires some difficult to read and maintain wrapper code or the use of intrinsic operations, which are provided by most modern compilers. However, these are typically translated into literal wrappers which improves on readability only but does not resolve any of the other issues.

Last but not least, an inline assembly approach inhibits any optimisations across these operations such as constant propagation, code reorganisation or any optimisation that requires in-depth knowledge of the operation.

In this paper we propose a set of C language extensions which provide a full support for vectors and vector-operations independently from the architecture. The vector operations are being dispatched to the SIMD extensions of a CPU if they are present, or implemented with a number of scalar operations, otherwise. The key design criteria were to come up with an as small as possible set of extensions that is large enough to i) benefit from the various existing SIMD instruction sets available on the market today, and to ii) enable the programmer to conveniently express various SIMD applications. Rather than inventing a completely new set of abstractions that suit these criteria, we build on the set of abstractions for SIMD operations that have been proposed in the context of OpenCL [8]. However, in contrast to OpenCL, we have integrated these operations into the GNU GCC compiler [5].

The rest of the paper is organized as follows: in Section 2 we present a part of BZIP2 compression as a running example. We discuss its potential for the use of SIMD operations and to what extent these can be achieved without making use of our proposed extensions. Section 3 describes the extensions we propose and explains to what extent these have been integrated into GCC v4.7. Section 4 illustrates how these extensions can be applied to our running example and quantifies its effect on the overall runtime of BZIP2 in a platform independent manner. After an extensive discussion of related work in Section 5, we conclude with a brief summary of the benefits and disadvantages of the approach we have taken in Section 6 and Section 7 provides suggestions for future work.

2 Motivation

As a running example we will consider a Move To Front (MTF) transformation which is used in modern compressing algorithms. This algorithm can be vectorised, however the pattern of the vectorisation is non-trivial and none of the auto-vectorisers we have tried out succeeded.

2.1 Move To Front (MTF) Algorithm

In order to improve compression algorithms which use Burrows-Wheeler Transformation (BWT) [2] as an additional post-processing step one can use Move To Front (MTF) transformation. After applying BWT we expect to get a string containing groups of repeating characters; for example '`aaaabbbccc`'. In order to decrease an entropy of the message and improve the efficiency of further Huffman encoding [6] we replace every symbol of the message with its position in the alphabet and move the symbol in the alphabet to the first place (or to the front as name suggests). In our case after encoding the string we will get '`0000100200`' assuming that the initial alphabet is '`abc`'. The MTF is being used in BZIP2 compression; the reverse version (unMTF) is used while decompression. In this paper we consider the latter algorithm and its vectorisation. The trivial implementation of the unMTF is the following:

```
char unMTF(char alphabet[256], int idx)
{
  char c = alphabet[idx];

  for (; idx > 0; idx--)
    alphabet[idx] = aplhabet[idx-1];

  return alphabet[0] = c;
}
```

The encoded message is an array of positions in the alphabet. To decode the message we read it from left to right and we use the same initial alphabet as during the encoding. The algorithm step replaces each element of the message with the symbol of the alphabet pointed by the element and moves the character to the front of the alphabet. The algorithm implemented as above is inefficient – it has $O(N)$ worst case complexity, where N is a length of the alphabet. The implementation used in BZIP2 reduces the worst case complexity to $O(\sqrt{N})$ by dividing the alphabet in $\sqrt{N}$ chunks, performing then above transformation on a single chunk where the given element is located and updating all the front-facing chunks applying constant-time operation. The implementation of this approach looks as following:

```
#define N 4096
char alphabet[N];
short ptr[16] = {N-256, N-256+16, N-256+16*2, N-256+16*3, ... };

void rotate_segment(char *v, int idx)
{
  if (idx == 0)
    return;
  do
    v[idx] = v[idx-1];
  while (--idx);
}

void rearrange_alphabet ()
{
  int i, j, k = N-1;
  for (i = 15; i >= 0; i--)
    {
      for (j = 15; j >= 0; j--)
        alphabet[k] = alphabet[ptr[i] + j], k--;
      ptr[i] = k + 1;
    }
}

void unMTF(int idx)
{
  int i, q, r, c;

  if (idx == 0)
    return;

  q = idx / 16;
  r = idx % 16;
  c = alphabet[ptr[q] + r];

  rotate_segment(&alphabet[ptr[q]], r);

  ptr[q]++;
  for (i = q; i > 0; i--)
    {
      ptr[i]--;
      alphabet[ptr[i]] = alphabet[ptr[i-1]+15];
    }

  alphabet[--ptr[0]] = c;
  if (ptr[0] == 0)
    rearrange_alphabet ();
}
```

The main idea of the approach is based on the following observation: shifting a whole chunk n symbols to the right can be achieved by decreasing the starting position of the chunk by n and filling n symbols in the front. In order to change the starting position of a chunk we have to allocate an alphabet-array which must be bigger than the length of the alphabet. We also have to store the starting positions of all the chunks – we use an array called `ptr` for that purpose. As each unMTF step potentially moves chunks to the left, eventually the first chunk will reach the first position in the alphabet-array, in that case the alphabet-array has to be rearranged by putting all the chunks at the end of the array; this is done using `rearrange_alphabet` function. In order to perform an unMTF with a single chunk `rotate_segment` function is being used.

In BZIP2 the length of the alphabet is 256 which after dividing into chunks gives us 16 chunks each of which is 16 characters long. Conveniently enough standard SIMD register these days is 128-bit long which is exactly one chunk. It means that `rearrange_array` can move chunks with two vector instructions rather than with 16 scalar. The fact that most of the SIMD architectures support permutations within a vector gives us a chance to implement a vectorised version of `rotate_segment`. Now, how the desired vectorisation can be expressed?

Auto-vectorisers we tried out did not consider any of the functions suitable for the vectorisation. There are several reasons for that: first of all, `roatate_segment` signature does not contain any information about the maximal values of `idx`, so a compiler can only deduce this information from the calling context. Secondly, a compiler needs to apply a cost model to prove that the transformation is beneficial, but this is not an easy task as a potential vectorisation may increase a number of instructions which affects an instruction pipeline; or add conditions which affect branch prediction; or change the memory access; etc. Without the knowledge that a particular function is a hot-spot a compiler can take a decision not to vectorise a function even if it is possible in theory.

In order to express `rotate_segment` explicitly in a portable SIMD way we have to have an interface for vector permutation. In GCC it was impossible before we added it with version 4.7. Alternatively one can express permutation using inline assembly, but disregard the fact it is non-portable, even for one architecture one may end-up creating several variants of the code. For example: Intel SSE3 has a `pshufb` instruction which does a byte-level permutation; any lower version of SSE support 32-bit elements permutations only which require a programmer to come-up with vector shifting and masking scheme which is less efficient and in case the architecture uses AVX another version of the code is needed.

Vectorisation of `rearrange_alphabet` can be done in a portable way starting from GCC v3.2, declaring a variable of vector type and for every chunk loading it to the variable and storing back into the memory. The code for the function looks as following:

```
typedef char __attribute__ ((vector_size (16), aligned (4))) xchar;
#define unaligned(x)  ((xchar *)x)
void rearrange_alphabet ()
{
  int i;
  for (i = 15; i >= 0; i--)
    {
      vector (16, char) vec = *unaligned (&alphabet[ptr[i]]);
      short idx = N-256+16*i;

      *(vector (16, char) *)&alphabet[idx] = vec;
      ptr[i] = idx;
    }
 }
```

Some architectures like for example Intel, differentiate aligned and unaligned vector loads providing two separate instructions for this purpose. In the code above we have to take care of the cases when a vector-assignment access unaligned memory. In order to inform a compiler we mark potentially unaligned memory by converting it to the vector type with minimal alignment.

3 C Vector Extensions

It turned that vector permutation is not the only missing feature which makes vector programming framework incomplete. Analysing common operations in different SIMD accelerators and combining them with scalar operations available in C we managed to identify the set of operations which is complete enough to program most of the SIMD algorithms. As we base our framework on GCC, the following features were missing there:

1. Vector indexing in the same style arrays are being indexed.
2. Vector element-wise and whole vector shifting. Element-wise shifting should take a special care when a target supports vector/scalar combination.
3. Scalar/vector and vector/scalar operations like `1 + {2, 3}`.
4. Vector comparison using standard comparison operations: `>`, `<`, etc.
5. Vector permutations in its most generic way.

Now we would like to give an idea about all the interfaces available in GCC with respect to the explicit vectorisation. The first step towards vector programming is to declare a vector type which can be done using a notion of attributes. Consider the following variable declaration example:

```
int __attribute__ ((vector_size (16))) var;
```

The `int` type specifies the base type of the vector, and the attribute specifies the length of the vector type measured in bytes. In the declaration above, given that int is a 32-bit type we define a vector of 4 ints. The basic type of the vector can be both signed and unsigned integer types: `char`, `short`, `int`, `long` and `long long`. In addition float and double can be used to define a floating-point vector types. The size of the vector type can be any number which is a power of two. Vector types are treated in the same way as C base types, it means that one can create variables of vector types, create pointers to vector types and use `sizeof` operator, use vector type when declaring a function argument or function return type, make type-casts. In the latter case, casting from one vector type to another, one must make sure the types are of the same size.

Defining a vector variable one can assign a constant value using an array notation. Consider the following example:

```
#define vector(elcount, type) \
__attribute__((vector_size((elcount)*sizeof(type)))) type

vector (4, float) pp = {3., .1, .4, .1};
```

Here we declare a pp variable of vector type of 4 floats and initialize the variable with values 3.0, 0.1, 0.4 and 0.1. Keep in mind, that if the initialisation vector contains less elements than the type, the missing elements will be implicitly filled with zeroes without a warning being produced. For example:

```
vector (8, short) v = {1, 2, 3};
```

In this case the compiler would initialize v with {1, 2, 3, 0, 0, 0, 0, 0}.

Constant vector-values are defined in the same way as when initialising a variable, but with an explicit type-cast. For example:

```
vector (4, int) a, b;
a = b + (vector (4, int)){1, 2, 3, 4};
```

Vector types can be used within a subset of normal C operations. Currently GCC allows using the following operations on vector types: +, -, *, /, %, unary minus, >>, <<, ^, |, &, ~. All the binary operations perform an element-wise operation on vector elements. For example:

```
vector (4, int) a, b, c;
a = b + c;
```

This code for each of the four elements of b will add the corresponding four elements of c and store the result in a.

Assigning expression of vector types, it is allowed to use a short form of the binary operation like +=, -=, etc. The semantics of the operation is going to be the same as in scalar case. For example:

```
vector (4, int) a, b, c;
a += b;          /* a = a + b;   */
b <<= c;         /* b = b << c;  */
```

Following the OpenCL conventions we also allow both scalar/vector and vector/scalar variants of binary expression. In that case the scalar is transformed to the vector of corresponding type where all the elements are equal to the given scalar. Consider the following example:

```
vector (4, float) a, b, c;

a = b + 1.;      /* a = b + {1., 1., 1., 1.}  */
a = 1. + b;      /* a = {1., 1., 1., 1.} + b  */
```

Note that the transformation will happen only when the scalar can be safely transformed to the vector type. For example, the following code would produce an error, as converting long to int includes a truncation.

```
vector (4, int) a, b;
long l;

a = b + l;       /* Error, cannot convert long to int.  */
```

When shifting is used on the vector types, we do not follow OpenCL in case when right-hand side of the shifting expression is greater than $\log_2 N$. Following the C standard of scalar shifting we leave this situation undefined. This is choice was made deliberately to avoid runtime masking of the right-hand side.

Vectors can be indexed in the same way as if they were arrays with the same number of elements and base-type. Out of bound access invoke undefined behaviour at runtime, however, the warnings can be enabled using `-Warray-bounds`. Consider the following example:

```
vector (4, int) a = {1, 2, 3, 4};
int sum = a[0] + a[1] + a[2] + a[3];
```

Vector comparison is supported within the standard comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=`. Comparison operands can be either both integer type or both real type. Comparisons between integer-type vectors and real-type vectors are not supported. The result of vector comparison is a vector of the same width and number of elements as the comparison operands with a signed integer base type. Vectors are compared element-wise producing `0` when comparison is false and `-1` (constant of the appropriate type where all the bits are set) otherwise. Consider the following example:

```
vector (4, int) a = {1,2,3,4};
vector (4, int) b = {3,2,1,4};
vector (4, int) c;

c = a > b;       /* The result would be {0, 0,-1, 0}  */
c = a == b;      /* The result would be {0,-1, 0,-1}  */
```

Vector shuffling is available using functions `__builtin_shuffle (vec, mask)` and `__builtin_shuffle (vec0, vec1, mask)`. Both functions construct a permutation of elements from one or two vectors and return a vector of the same type as the input vector(s). The mask is an integral vector with the same width (W) and element count (N) as the output vector.

The elements of the input vectors are numbered in memory ordering of `vec0` beginning at 0 and `vec1` beginning at N. The elements of mask are considered modulo N in the single-operand case and modulo $2 * N$ in the two-operand case. Consider the following example:

```
vector (4, int) a = {1,2,3,4};
vector (4, int) b = {5,6,7,8};
vector (4, int) mask1 = {0,1,1,3};
vector (4, int) mask2 = {0,4,2,5};
vector (4, int) res;

res = __builtin_shuffle (a, mask1);      /* res is {1,2,2,4}  */
res = __builtin_shuffle (a, b, mask2);   /* res is {1,5,3,6}  */
```

The contribution of this work is not only in the identification of the operation set, but also in implementing most of the missing parts in GCC. Currently version 4.7 will include all the missing features except whole vector-shifting. Comparing vector and scalar arithmetic operations the following operations are not available in vector mode: `++`, `--`, `!`, `&&`, `||`.

4 Case-study

In this section we will demonstrate a vectorised version of the running example and evaluate performance. We are not going to make extensive performance measurements, cause the figures mainly depend on the quality of a particular code-generator. The main purpose is to demonstrate that using GCC vector extensions we can address complicated patterns producing a code which is portable across all the platforms supported by the compiler and which can efficiently use SIMD accelerators in case they are present.

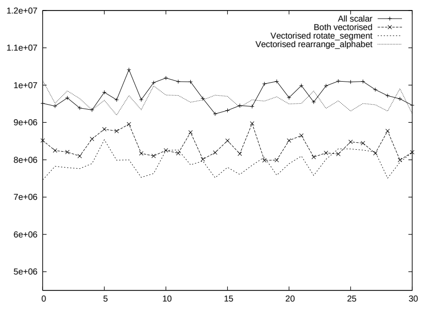
In order to implement `rotate_segment` in a vectorised way we would load a 16-symbol chunk into a vector register, perform a permutation in-place and store it back into the memory. All the chunks are potentially unaligned, so we mark it in the same way as in `rearrange_alphabet`. The code looks as following:

```
const vector (16, char) perms[16] = {
  (vector (16, char)){0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15},
  (vector (16, char)){1,0,2,3,4,5,6,7,8,9,10,11,12,13,14,15},
  (vector (16, char)){2,0,1,3,4,5,6,7,8,9,10,11,12,13,14,15},
  /* ... */
};

void rotate_segment (char *v, int idx)
{
  vector (16, char) t, vec;

  if (idx == 0)
    return;

  t = *unaligned (v);
  vec = __builtin_shuffle (t, perms[idx]);
  *unaligned (v) = vec;
}
```

As all the permutation masks are static, the compiler can perform an optimisation of each particular permutation. For example, it can replace the case when `idx` is one with something like: `swap (v[0], v[1])`. Note that in OpenCL framework permutation is a library function call and the above optimisation cannot be achieved unless link time optimisation is used, in which case the library must be compiled with special flags.

Consider Fig. 1 which shows how the vectorised and non-vectorised versions of `rotate_segment` and `rearrange_alphabet` impact performance of unMTF. The base-line of the experiment is a fully scalar implementation. The best performance improvement on both architectures we observe when `rotate_segment` is vectorised and `rearrange_alphabet` is not: we have about 20% and 30% speed-ups on according architectures. We observe the negative impact from vectorising `rearrange_alphabet`. In order to explain this fact we have to realise that unaligned move on Intel is expensive and it also creates additional pressure on the memory. The operation itself happens quite rarely (once per length of the alphabet-array) so the initialisation overheads are bigger than the performance gain.

As another experiment we would like to include the newly implemented function in the box-standard implementation of BZIP2 in order to see the impact of this function on the overall decompression process. As an input data we use encoded 347MB video-file and as a measurement unix `time` command taking user's time. On Intel Core2 duo we observed 18% speed-up (36.6s vs 30.2s) but on the Core i5, the run-time difference was in order of measurement error (26s


(a) Intel Core2 duo, 2GB RAM, 4MB cache

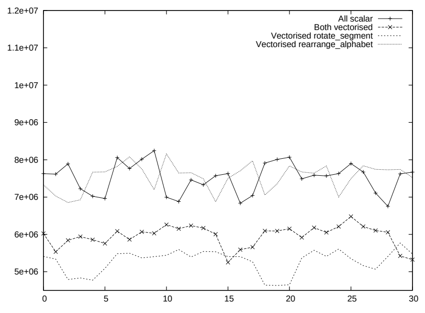

(b) Intel Core i5, 4GB RAM, 6MB cache

Fig. 1. Vectorised and non-vectorised versions of unMTF measured in clock-cycles. All the measurements were repeated 30 times sequentially using an output of a previous run as an input for the next one. The first input is a file of 10^5 random characters.

both). As BZIP2 is a pipe-line we can see that the improvement of a single part may not affect whole the process; hence unMTF is not a hot-spot on Core i5 processor. Identifying a new hot-spot of the algorithm requires an analysis which is outside of the scope of this paper.

We have demonstrated how easily one can experiment with the potential benefits of using SIMD extensions. Keep in mind that all the results on both architectures were obtained from one and the same source code which would also work on any architecture supported by GCC.

5 Related work

In this section we will make an overview of existing concepts and approaches which allow to exploit SIMD extensions of a CPU.

5.1 Automatic vectorisation

Automatic vectorisation is an implicit way of recognizing loop patterns and rewriting them using vector instructions during the optimisation cycle of a compiler. Most modern compilers like GCC, LLVM and Intel are equipped with auto-vectorisers. Automatic vectorisation is a complicated and expensive process, occupying about 25 000 lines of code in GCC; it has to consider data dependencies of the loops, internals of the target architecture and use a cost-model to determine whether it is reasonable to vectorise a given loop.

Automatic vectoriser does not require any effort form the side of a programmer, literally a programmer just compiles the code and the vcetoriser uses knowledge and heuristics to do the job. However, the downside of the approach is lack of chances to influence the decision of the vectoriser. The number of supported patterns is always limited, and in the cases of non-trivial data-dependencies the vectoriser would give-up. In order to get the best performance from an auto-vectoriser in case of floating-point operations one have to specify flags that violate IEEE and ISO implementation of floating point. As an example we can consider the case of horizontal sums:

```
float *array, result;

for (i = 0; i < N; i++)
  result += array[i];
```

```
float *array, result;
float_vec reg;

/* Assume N % 4 == 0  */
for (i = 0; i < N; i += 4)
  reg += *(float_vec *)&array[i];

result = reg[0] + reg[1]
         + reg[2] + reg[3];
```

According to IEEE floating point standard the order of operations can change the result; hence the above optimisation is illegal. In order to legalise it in GCC one needs to specify `-ffast-math` flag when compiling and it is impossible to use it on a given loop only. It means that in order to make auto-vectoriser perform the optimisation, a programmer has to switch a flag potentially violating all the floating-point operations.

The auto-vectoriser cannot properly handle the loops with the control-flow or uncountable loops, e.g. `while (*x != NULL)`.

5.2 Virtual Instruction Set

R.Bocchino et al in [1] address the problem of portability of SIMD instructions by introducing a virtual instruction set specially designed for vector operations. The main design criteria is to provide an abstraction for various vector architectures and architecture classes. Architecture classes include sub-word SIMD like Intel SSE and PowerPC Altivec and streaming processors like RSVP. The Low Level Instruction Set (LLVA) allow both arbitrary length and fixed vectors providing asynchronous load and store semantics for long vectors and introducing alignment attributes. The compilation scheme involves a compiler which can generate a portable vector code and a translator with full information about the target architecture and system configuration.

The LLVA approach provides a portable standard for SIMD operations. However, this approach raises several practical and theoretical questions. Practically, the architecture exists only as a prototype with implemented translator for several ISA-s. It means that in order to integrate LLVA in any existing compiler, we will have to provide a translation from the IL of a compiler to the LLVA. Assuming that we did that, we will have to implement the translators for all the targets we want to support. Keep in mind that LLVA provides instructions only for vector operations, it means that all the non-vector operations has to be integrated into the representation as well. Assuming that we did that as well, we come to the point when we will have to instruct our auto-vectoriser and possibly other optimisations to generate the code using LLVA. How can we estimate the cost of the operation, if we don't know the target architecture?

As several architecture classes are supported within the LLVA, and there are mechanisms, allowing careful tuning for each processor class, how efficient would it be to run an LLVA code tuned for class A on class B?

5.3 OpenCL

In terms of portability our approach is very similar to OpenCL and we even borrow the syntax of SIMD operations, however, there is a number of important distinctions. The main purpose of Open Computing Language (OpenCL) is to provide a unified framework which supports heterogeneous architectures. The main stress is a combination of CPUs and GPUs. OpenCL provides a C99-based language for programming kernels and set of APIs for controlling the host. OpenCL supports both data-based and task-based parallel programming models. Amongst the other APIs, OpenCL provides an abstraction for SIMD operations.

Obviously the intention of OpenCL is very different from ours. OpenCL operates with a large-scale problem and tries to include in the framework as much instruments as possible; where our approach solves a single issue. The OpenCL C programming language is based on ISO/IEC 9899:1999 C language standard (a.k.a C99) [7], but it also introduce a number of restrictions. Most importantly, OpenCL tries to cover all the undefined or ambiguous cases of C99 standard. For example, basic types, like `int`, `char` and `long` get a fixed size; C99 in this cases fixes only the relation of the type-sizes i.e `char` $\leq$ `int` $\leq$ `long`. Defining bit-shift operations `e1 << e2`, OpenCL states that only lower $\log_2 N$ bits of `e2` will be used during the operation; C99 in this case states that if `e2 >` $log_2 N$, the result is undefined.

Arguably the restrictions of OpenCL increase the portability of programs but at the same time they remove backward-compatibility with ANSI C code. Practically it means that the existing C code may not work within the OpenCL compiler.

Technically OpenCL provides a set of libraries and header files, but the actual compilation is done by a C compiler of users choice. This is a key difference from the approach we are taking, as we implement SIMD operations as an integral part of the C language; hence as a part of the compiler. Decoupling a framework from the compiler gives you a freedom when you choose a compiler, but in terms of SIMD operations we see the following problems with this approach:

1. In order to define a SIMD vector OpenCL provides `type`n construction, where n can be 2,3,4,8 or 16 and the `type` is a basic scalar type, e.g. `char`, `int`, `float`, etc. As far as there is no way to override selection operator `[]` in C, OpenCL introduces a new scheme for enumerating vector components introducing notion of `lo, hi` components `x, y, z, w`, etc. The vector type is mapped to the hardware-specific SIMD vector type or static array in case SIMD accelerators are not present within the architecture. Such a design

makes it rather complicated to support vectors of arbitrary length, as each `type`n is defined as a new structure and chosen indexing scheme leads to the combinatorial explosion. Also, each time when the length of vector register doubles, the standard correction is required. For example, currently, it is impossible to define a `char32` type, however, it is supported by Intel AVX. Our approach allows to define a vector of arbitrary length, where the length is a power of two. To index elements we use a standard selection `[]` operator and during the compilation vector operations are compiled to the longest vectors supported by the architecture.
2. Basic vector operations like arithmetic, comparison, shuffling are whether aliases to the intrinsic functions or external functions defined in the library. From the performance point of view both cases are harmful as they decrease a chance for optimisations. Library function call prohibits even from the simple constant propagation; intrinsic functions normally do not participate in the optimisation cycle. If vector operations would be inlined, the compiler can generate a better code with respect to the pipelining and register pressure.
3. OpenCL SDKs are mainly closed-source products which are released for the combination of hardware architecture and operating system. It means that there is a chance that all these products perform slightly different. Our approach does not solve this problem fully, as code-generators are unique for every hardware architecture as well, however, the most of the optimisations happen in the middle-end. Also, the fact that GCC compiler is an open-sourced product gives a better chance to identify the reason of the undesired behaviour.

6 Conclusions

In this work we have demonstrated a framework which allows to encode explicit vector computations in a portable fashion. The framework is implemented as a set of C language extensions within the GCC compiler preserving backward compatibility with ANSI C standards. This approach is beneficial because of the following reasons:

1. A programmer or a compiler which uses C as a target language gets a chance to express vector computations of any complexity at the level of C without involving any additional libraries or frameworks and not taking care about the specifics of any particular architecture, but being sure that the vector code would be executed within the SIMD accelerators in case they are present.
2. Backward compatibility makes it easy to change an existing software by rewriting a certain function or code region using vector types and operators.
3. Internal representation of vector operations is shared with the auto-vectoriser which means that any architecture that supports auto-vectorisation supports explicit vectorisation and vice-versa and any improvement affects both parts of the compiler.

4. Vector operations within the GCC are fully-fledged members of the flow-graph; hence they participate in the optimisation cycle similarly to the scalar operations.

The downsides of the proposed approach are the following:

1. Currently the framework is supported only within the GCC compiler, which means that one will have difficulties in case of moving code-base to the different compiler. One of the solutions to this problem is to make the language extensions fit into a new C language standard, but unfortunately this is a long and complicated process.
2. The price for the abstraction layer is a potential inability to use the newest SIMD instructions. There is always a gap in time between the architecture becoming available on the market and the code generator being able to use it correctly. There is no good solution here, however the good thing is that a programmer cannot and doesn't have to do much about it. Whenever a given pattern would be incorporated in a code-generator, the code should start to work faster automatically.

7 Future Work

In the near term we aim to implement a support for the missing operations available in a scalar mode, but not available in a vector mode. These operations are: `++`, `--`, `&&`, `||`, `!`. After that it is necessary to make more careful research with respect to the common SIMD patterns supported by various architectures and provide according C constructions. For example, currently there are no facilities to use horizontal vector instructions like sum or min/max of the entire elements.

Another important issue that we aim to address is vector alignment. Some ISAs have different instructions to load/store vectors from aligned and unaligned memory. Normally unaligned variant of the instruction is less efficient. Currently in C we cannot annotate a chunk of memory being aligned/unaligned; hence generate an efficient load/store instruction. In auto-vectorisers the similar problem is partially solved [4,11] by transforming the computation. As we are in explicit mode, our goal is to provide annotations and mechanisms to emit a suitable instruction. As C arrays and pointers are the same in its declaration, `aligned` attribute does not give the desired effect, as applied on a pointer it means that the pointer is aligned (not the memory where it is pointing). In case attribute is applied to a type then in case of an array it means that each element of the array becomes aligned which is not true. A solution here is to introduce a new attribute for aligning a chunk of memory. After that pointer dereferences and pointer-arithmetics must be analysed in order to see if the alignment gets changed. Unfortunately in general case such an analysis is impossible as variables and functions can be external. However, we hope to address this problem by including the analysis in the Link Time Optimisation (LTO) cycle when all the information is statically available.

In the long term we hope to include describe extensions into a new C language standard. That would give a chance to use various compilers for one and the same code. It seems that it is already too late to apply for C1X standard, but the upcoming one could be just the right time.

8 Acknowledgments

This project started in terms of Google Summer Of Code 2010, with Artjoms Šinkarovs being a student and Richard Güenther being a mentor. The authors wish to thank GCC community for their help during the coding. Special thanks to Richard Güenther, Richard Henderson, Joseph S. Myers and Jakub Jelinek for their valuable comments and patch reviews.

References

1. Robert L. Bocchino and Vikram S. Adve. Vector LLVA: a virtual vector instruction set for media processing. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 46–56, New York, NY, USA, 2006. ACM.
2. M. Burrows, D. J. Wheeler, M. Burrows, and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.
3. Kuan-Hsu Chen, Bor-Yeh Shen, and Wuu Yang. An automatic superword vectorization in LLVM. In *16th Workshop on Compiler Techniques for High-Performance and Embedded Computing*, pages 19–27, Taipei, 2010.
4. Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, volume 39, pages 82–93, New York, NY, USA, May 2004. ACM Press.
5. Free Software Foundation. GCC. http://gcc.gnu.org.
6. David Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, September 1952.
7. ISO/IEC. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standardization, Geneva, Switzerland., 1999.
8. Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2*, 15 November 2011.
9. Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 281–294, Washington, DC, USA, 2006. IEEE Computer Society.
10. Gang Ren, Peng Wu, and David Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. In *In 16th International Workshop of Languages and Compilers for Parallel Computing*, pages 420–435, 2003.
11. Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures, 2002.
12. N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extensions. *International Journal of Parallel Programming*, 28:363–400, 2000.